Solar Car Data Acquisition System

Submitted To

Dr. James Wiley
Dr. Gary Hallock
Heng-Lu Chang
Amal Kouttab
University of Texas Solar Vehicle Team

Prepared By

Maxwell Archibald
James Creamer
Travis McClure
Beau Roland
Minh Van-Dinh
Christian Young

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Fall 2017

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

TABLES

FIGURES

EXECUTIVE SUMMARY

The University of Texas Solar Vehicle Team (UTSVT) tasked our team to design a data acquisition system to collect sensor data while creating a visualization of the up-to-date data. The previous UTSVT data acquisition system was slow to boot and had limited sensor access during runtime. Our design solution improved upon the boot time and sensor access limitations. The design of the system can be broken into three major sections: sensor data gathering, data logging and transmission, and data display. These sections bring together a design solution for the UTSVT data acquisition system.

The sensor data gathering design uses Nucleo-144 microcontrollers to rotate through the sensors in the system before sending the data through the system. The coding structure of the Nucleo-144 was broken into two major functions: initialization and data acquisition. The initialization focused on sensor configuration for the specific communication protocol and sensor settings. The data acquisition code cycled through the sensors before organizing the data and placing the data into a buffer. The buffer would send out all of the messages to the Controller Area Network (CAN) bus based on the message format that the system modules agreed on.

Once the data is collected by the Nucleo-144, the data is transmitted through the CAN bus to the Raspberry Pi 3 (RPi3). The CAN bus was chosen for this communication because of its reliability when used in electromagnetic interference (EMI) environments. The messages are compiled to log files before packaged into an UDP packet. The UDP packet is then sent via ethernet to another Raspberry Pi 3 stationed in the pit where the UDP packet is received and sorted into a readable format. UDP was selected over the other various communication protocols because of it was easy to manipulate, had fast socket reconnection time, and had fast packet transmission speed.

The RPi3 stationed in the pit receives and process the UDP packets, hosts the InfluxDB database client and Grafana display server, and stores sensor data on the local SD card. Upon UDP packet receipt, the RPi3 executes a logging script that writes the data to the correct CSV file and then inserts the data to InfluxDB. Grafana then continuously queries InfluxDB for new data points to display the data on its graphical interface in near real time.

1.0 INTRODUCTION

The purpose of this report is to provide the UTSVT with a detailed report on the design and implementation of a data acquisition system. The UTSVT requires a data acquisition system to gather data from the sensors located on the solar vehicle and to store this data in an easily accessible location for future reference. Additionally, the system must log and display the data in a remote location in the race pit or chase vehicle.

The report has seven main sections: design problem, design solution, design implementation, test and evaluation, time and cost constraints, safety and ethical aspects of the design, and recommendations. The design problem section discusses the current UTSVT's solar vehicle data acquisition system and specifications for a new data acquisition system. The design solution section describes the design for the solar car data acquisition system developed over the course of this project. The design implementation section elaborates on the design changes made throughout the duration of the project as well as the reasoning behind them. The test and evaluation section outlines the tests performed and investigate their results. The time and cost section constraints elaborates on the total price of the design as well as issues with time that occurred throughout or project. The safety and ethical aspects of the design section illustrates the main concerns that encompases this desing. The recommendation section defines improvements to the team's design going forward.

2.0 DESIGN PROBLEM

The following section describes the UTSVT's current solar vehicle data acquisition system and its failings, their requirements for a new system, and the specifications that define these requirements. The problem background explains the basic operation of the previous data acquisition system and identifies the failures of that system during races. The system requirements section identifies the primary needs of the UTSVT, specifically the continued safety of the driver, local storage of collected data, and a continuous remote data stream. Finally, the project specifications section defines the specifications that meet these needs in four main categories: inputs and output specifications, user interface specifications, operating environment specifications, and performance specifications.

2.1 Problem Background

The previous UTSVT data acquisition system collected data from sensors inside the solar vehicle, but this system presented issues with boot times and reliability. The previous system used microcontrollers to gather sensor data before sending this data to a single board computer inside the vehicle. The single board computer then logs the data in local storage. This system, however, booted in two to three minutes and did not reliably collect sensor data. This caused data collection to be limited in use, as data logs could be incomplete. The boot time also restricted vehicle operation, as the system provided critical operation information to the driver. Additionally, the data acquisition system frequently crashed, prevented the collection of any sensor data. Overall, the system prevented timely access to up to date sensor data. The UTSVT requires a data acquisition system that solves this problem.

2.2 System Requirements

The UTSVT required a data acquisition system to gather sensor information from their solar car and display the sensor data. As mentioned in the problem background, the previous data acquisition system booted slowly, so a new system had to address this issue. In addition, the UTSVT required the system to provide a flow of data from the car during a race by transmitting the data off the car while still storing the data locally. This requires the data acquisition system to include a remote computer that provides easy access to the collected data. The system needs to operate during race conditions since that will be the most valuable time to use the system. Finally, the system must not interfere with other car operations for driver safety and car efficiency.

2.3 Project Specifications

For the data acquisition system to fulfill the UTSVT's requirements, the project must meet all applicable constraints in the form of specifications. The following sections break down the constraints into inputs and output, user interface, operating environment, and performance specifications. The inputs and output specifications define constraints associated with the sensors sending information to the data acquisition system and the data the system must provide to its users. The user interface specifications explain constraints relevant to the user's interaction with the system, specifically the different ways in which the UTSVT and the public will view the collected information. The operating environment specifications define the conditions the system

will encounter during a race. Lastly, the performance specifications show the minimum standards the system must possess to meet the needs of the UTSVT.

2.3.1 Inputs and Output Specifications

The data acquisition system must gather information on the performance of the solar car and display that information to the UTSVT. An unknown number of sensors on the solar car will provide unprocessed data to the system, communicating over the I^2C, SPI, serial, and CAN protocols. The system will interpret the data and store it securely in removable media while displaying the information to users as shown in Figure 1.

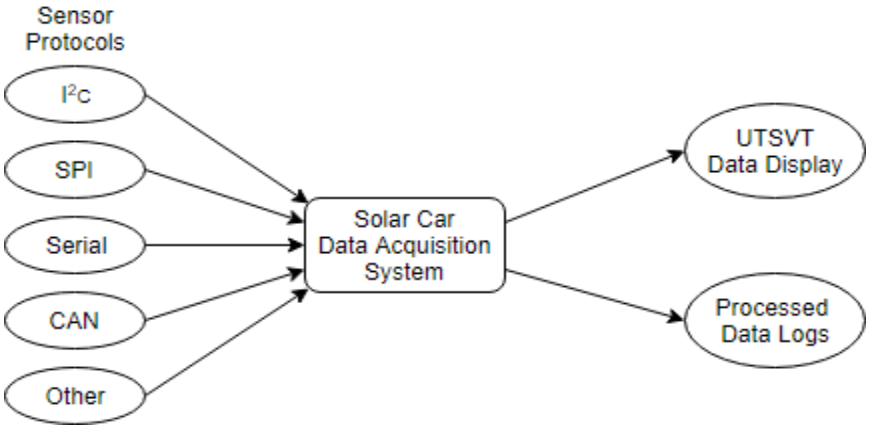

Figure 1. Data Acquisition I/O Diagram

2.3.2 User Interface Specifications

The solar car's data acquisition display must provide easy access to the solar car data. The in-car system will not require an interface since the system will automatically run when powered, users of the remote display should have the ability to customize the data displayed. Customization should involve the ability to select displayed groups of data through menu interaction as well as the option to display a history of the displayed groups via another menu. The output data logs must have an easily readable form such as a text file Tabor an appropriate equivalent.

2.3.3 Operating Environment Specifications

The UTSVT requires a system that can perform properly in racing conditions. The system must be able to operate in the temperature range and vehicle speed as listed below in Table 1. The solar car will operate at a distance from the race pit, so the system must consider this distance to ensure that the pit crew receives the necessary data. The battery voltage, system voltage, and vehicle low voltage are constraints from the solar car itself that should be considered in wiring and power design.

Table 1. Operating Environment Specifications

Environment Variables	**Specifications**
Temperature	0°C to 49°C
Remote Transmission Range	2.5 km to 8.0 km
Vehicle Speed	0 km/h to 104.2 km/h [1]
System Voltage	0 V to 5 V
Vehicle Battery Voltage	100 V to 120 V
Vehicle Low Voltage	12 V
Software Environment	Linux-based systems

2.3.4 Performance Specifications

The new data acquisition system must meet the constraints as shown on the next page in Table 2 to fulfill the needs of the UTSVT. The maximum transmission bandwidth is a limit based on the radio modem used on the solar car to transmit data. The power consumption limit minimizes the overall impact the new system has on the solar vehicle's battery and as a result overall operation. The system needs to consistently run for at least eight hours during the car's operation to guarantee the pit crew can advise the driver for the duration of a race while storing the data. Lastly, the system must have a minimum of 4 GB of storage space to prevent the loss of data.

Table 2. Performance Specifications

System Variables	Specifications
Maximum Transmission Bandwidth	0.5 Mb/s
Total Power Consumption	0 W to 10 W
Minimum Local Storage Size	4 GB
Minimum Continuous Active Time	8 hours

3.0 DESIGN SOLUTION

This section presents the design for the solar car data acquisition system developed over the course of this project. The solar vehicle data acquisition system design collects sensor data from across the vehicle in order to store and remotely display that data to a race pit or chase vehicle. As seen in Figure 2, the design contains two major subsystems: the in-car acquisition system and data display system. The in-car acquisition system gathers raw sensor data over the specified communication protocols: I^2C, CAN, SPI, and serial communication. The subsystem then logs this data in a local SD card storage system, packages the data into User Datagram Protocol (UDP) packets and transmits the packets to the data display system in the race pit. The data display system receives the UDP packets before logging the packet contents in a redundant local SD card storage system and database. Using this database, the data display system hosts a local server to allow the UTSVT to view race sensor data while the vehicle is operating.

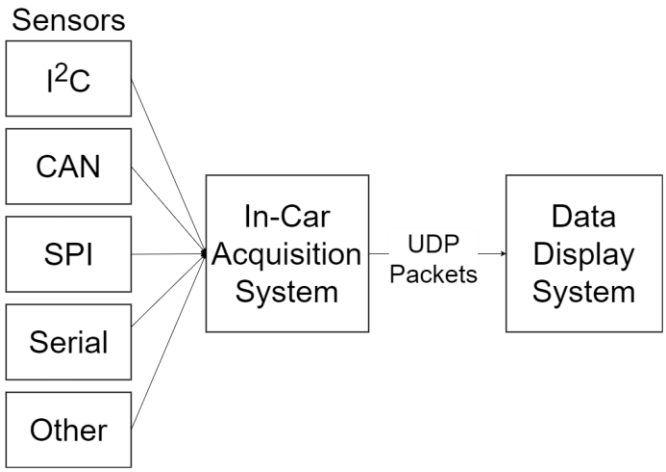

Figure 2. High Level Block Diagram

3.1 In-Car Acquisition System

This section examines each component of the in-car acquisition system in detail, specifically the STM Nucleo-144s, CAN communication bus, and Raspberry Pi 3. The in-car acquisition system gathers sensor data over the specified communication protocols using the STM Nucleo-144s as shown in Figure A.1 of Appendix A. After collecting this data, the Nucleo-144s send this data to a Raspberry Pi 3 for data logging in local SD card storage and transmission to the data display system via UDP. Sending this data to the RPi3 is completed via a CAN communication bus, which allows the design to include two Nucleos for data acquisition.

3.1.1 STM Nucleo-144

In our design solution, we use STM Nucleo-144 discovery boards to interface the sensors. These boards offer a wide range of library support, hardware interface ability, and a large number of available GPIOs that can be used to connect several sensors. For ease of use, we created an interface board to easily connect sensors to the microcontroller and to easily connect the microcontroller's CAN hardware to an external CAN bus. This daughter board's schematic is shown below [b#1].

The data acquisition software design consists of two parts: an initialization phase and a acquisition phase. In the acquisition phase, the microcontroller sets up the hardware. This includes setting up a system clock, instantiating a configurable number of sensors, calibrating these sensors, allocating memory for the data that is collected from the sensors, and setting up a periodic timer interrupt that controls the overall data output rate. In addition, when constructing a new sensor object, the software also saves the data sample rate at which to collect this specific sensors data. The acquisition phase is made up of a large software loop. This loop begins with going through each sensor and determining whether it's time to sample the sensor. This is done by comparing the system clock with the last time the software sampled the sensor and the sensor's desired sample period. If system time is passed the time to sample the sensor, the software reads the sensor and stores the data in the reserved memory allocated earlier in the initialization phase.

After completing the sensor loop, the software handles CAN communication. The software checks to see if a interrupt flag has been raised signaling the processor that it is time to package the sensor data collected into CAN messages. This flag is raised with the periodic timer instantiated earlier and acknowledged after pushing these CAN messages into a CAN FIFO. The last portion of the acquisition loop is send the next element in the FIFO out on the CAN bus if it's not empty. A detailed block diagram of the software blocks is shown in Figure B.1 of Appendix B.

3.1.2 CAN Bus

The CAN bus transmits data between the microcontrollers and the RPi3. The CAN bus requires a message to contain eight bytes of data and two bytes for the message identifier. Since the Nucleo-144 used in the team's design has a CAN transceiver chip on the board, the microcontroller will send this message to a CAN chip separate from its board. We used a CAN translator chip, ADM3054, with the Nucleo-144 to convert the signal. The signal is converted to CAN high and low which are the main lines of the CAN bus. Two chips, the ADM3054 and MCP2515, convert the CAN bus lines to SPI to be read by the RPi3. Once the connections are set up, the CAN high and low lines need two 120 Ohm termination resistors between the lines at each end of the bus. The termination resistors are required for the receivers to interpret the signals correctly. Finally, the RPi3 has to set up communication with the MCP2515 with its native driver which will allow the RPi3 to receive sent CAN messages.

3.1.3 Raspberry Pi 3

The RPi3 in the in-car acquisition system receives the incoming CAN bus messages while logging data and sending UDP packets to the RPi3 in the pit. To do this, the RPi3 runs a script on boot up written in C++. This script creates a socket specifically to handle the full incoming CAN bus message interface. While the messages are piped into the RPi3 via the CAN bus, the script parses through the message and places them into a buffer based on which STM Nucleo-144 sent the message. At the same time, the script sends the UDP packet as seen in Figure A.2 of Appendix A. Once the buffer is filled with data, the script empties the data into a Nucleo specific file with the timestamps and the data.

3.2 Data Display System

This section focuses on each of the major components in the data display system, specifically the pit-side Raspberry Pi 3, InfluxDB, and Grafana. The data display system is responsible for organizing and displaying the data received from the in-car acquisition system. The pit-side Raspberry Pi 3 receives the UDP packets transmitted by the in-car Raspberry Pi 3, as shown in Figure C.1 of Appendix C. The data is saved to a local SD card, stored in InfluxDB, and finally queried by Grafana to display the data to the UTSVT.

3.2.1 Raspberry Pi 3

The pit RPi3 handles the the receipt of the incoming UDP packets from the in-car acquisition system as well as the storage and display of the sensor data inside the packets. Figure C.2 of Appendix C outlines the software flow on the pit RPi3. Upon start up, the Pit RPi3 sets up InfluxDB in Python and the UDP in C++ client to receive and process sensor data from the in-car acquisition system. The logger was written in Python because InfluxDB natively supported the language, and the UDP client was written in C++ because the UDP socket needed both ends to be written in the same language to function without the use of an Application Programming Interface (API).

In order to communicate between the two languages, we implemented a wrapper on the pit RPi3. By exporting the necessary UDP client functions written in C++ into a Python module then importing it into a Python script, the pit RPi3 successfully received packets for data processing. After a successful initialization, the Pit RPi3 enters a main loop where it continuously polls the UDP server for UDP packets. Once the UDP client receives a packet, it is read into different JSON messages for each sensor which each have a function that compiles the sensor data into a character array for the UDP client. The sensor data contains a timestamp for data collection, a sensor ID, and all sensor values. The UDP client module converts the separate sensor data into a list of strings using Python data structures and passes the data to the Python script. Finally, the Python script appends the sensor data to different comma separated values files for each sensor, logs the data on to InfluxDB for data visualization, and reenters the loop to receive another UDP packet.

3.2.2 InfluxDB and Grafana

InfluxDB and Grafana serve as a database creation and management tool for sensor data and a data visualization dashboard, respectively. Figure C.3 of Appendix C outlines the interaction among InfluxDB, Grafana, and the user browser. With the proper credentials, a user can log in to Grafana and view the different sets of data on a browser dashboard. Each time a user views new points by selecting a set of data, Grafana requests that data from InfluxDB through a query, and the InfluxDB sends the data over to Grafana to display on that dashboard so that the user can view it. This is done so that the user can observe the data being filled in on the dashboard in near-real time.

4.0 DESIGN IMPLEMENTATION

The following sections explain the rationale behind the major components of the data acquisition system design. The design of the in-car acquisition system was motivated by the capacity for sensor input and reliable communication between devices. The data display system was designed while focusing on ease of use and decreasing overhead. These sections provide more detail about this rationale in the context of the in-car acquisition and data display subsystems.

4.1 In-Car Acquisition System

The in-car acquisition system consists of three parts: microcontrollers that interface sensors into the system, a Raspberry Pi 3 that stores the data and sends the data to the race pit, and a robust data communication protocol. We chose the STM Nucleo-144 discovery boards as a starting development platform for our microcontroller interfaces due to the wide support of the MBED library and the vast number of available GPIOs. The Raspberry Pi 3 was chosen as an in car acquisition hub due to the team's relative familiarity, its online support, and its portability. Throughout the development process, the in-car acquisition system underwent several changes. The following explains the major design changes and the reasonings behind them.

4.1.1 STM Nucleo-144

One of the requirements of the UTSVT data acquisition is the need for the system software to not only be efficient but to also be easily understood and adaptable. This is due to the nature that the prototype is likely to be integrated into the UTSVT and expanded upon to have more features. It was decided that the acquisition software was to be written in the C++ language. This allows

future developers to lean on the well tested support of the MBED library. In addition, due to the nature of inheritance in C++, the system software is divided into separate functional blocks. Each of these software blocks can contain the more detailed instructions that perform a particular function whereas the main software can be written with simpler control logic with the use of accessor functions to these software modules. As a result, the main software is relatively simple allowing for easy code changes. Likewise, the software modules can updated easier to provide a new feature without the need for the system software to change.

Throughout the development process, the main software control logic varied greatly over it's structure. For instance, the original acquisition code was organized into separate functional threads as shown in the figure shown below [B.2]. This allows for several modules to be running concurrently and has the potential to offer high data throughput. As discussed in more detail in section 5.1.1 Maximum Data Sample Rate Test, the data rate was limited to 166 Hz. This is much lower than required data throughput required by the senior design problem. In theory, the data dependencies between threads and the raw overhead of a single processor to switch its context between threads were the main reasons for this data limit. Therefore, the main code structure was reorganized to be non-threaded as shown in the figure B.2 in the appendix. This structural change not only removed the need of data locks that prevented Read-Modify-Write structural hazards but, the structural changes prove to increase the data bandwidth past 1000 Hz after rerunning the same test in section 5.1.1.

4.1.2 CAN Bus

The CAN bus was selected for in-car communication due to its superiority to Ethernet communication in this application. The CAN bus was the in-car system communication because of its reliability in EMI environments. This bus is reliable compared to Ethernet due to its robust error correction, message acknowledgement, and fault confinement. Ethernet is less reliable because of the protocol's focus on speed rather than reliability. The CAN bus error correction forces the message to be sent repeatedly until the correct message is received ensuring correct signals. The message acknowledgement forces the receiver to send a signal confirming that the message request was seen. Finally, the CAN bus will force a module off the bus if it does not follow proper

communication protocol. The fault confinement preserves the integrity of the bus so the bus is not muddied by bad messages.

4.1.3 Raspberry Pi 3

We chose a Raspberry Pi 3 to handle data transmission and storage due to its high usability and functionality. Additionally, most team members have had experience with the RPi in the past so it was easy to implement into the design plan. RPis are also easily replaceable in the event that one breaks. We installed Linux on the RPi because it is free, common, and easy to use. Additionally, the Raspberry Pi 3 can run the Linux operating system. We chose UDP over alternative transmission protocols due to its speed and reliability. UDP does not implement a handshaking protocol for communication, which means that it does not wait for an acknowledge signal to be returned when a packet is received. This increases the speed of transmission at a small cost to reliability. When sensors are polled hundreds of times a second, a few packets can be lost while still maintaining accurate data measurements. The UDP server was coded in C++ to maintain language consistency for the in-car acquisition system. This was to minimize the necessary number of programming languages for the entire system and to reduce complexity for the in-car acquisition system.

4.2 Data Display System

The data display system consists of a Raspberry Pi 3 running InfluxDB and Grafana. The reasoning behind the use of these components in the design solution was primarily motivated by ease of use for the project team and UTSVT, familiarity with the systems, and minimizing installation overhead. The following sections explain this reasoning in more detail.

4.2.1 Raspberry Pi 3

We chose a Raspberry Pi 3 to host the data display system for the same reasons it was chosen for the in-car acquisition system. Additionally, the use of Linux was beneficial for the RPi in the pit because Linux does not crash if WiFi users attempt to overload the system. This is especially useful for when users are trying to connect to the Grafana server. We wrote the UDP client in C++ to mirror the programming language of the entire in-car acquisition system, specifically the UDP server. It is impossible to create a UDP socket that uses different languages for the transmitter and

receiver without the use of an external library or API, so we wrote both sides of the UDP socket in C++. For a similar reason, we chose Python as our programming language for our logger that stores and inserts data to the SD card and InfluxDB, respectively. InfluxDB natively supports Python for data insertion and only supports C++ through the use of an open source project on GitHub. Because of this dependency, we opted to write a simple wrapper to connect the UDP client to the logger. Finally, Python was chosen for the logger because it has a multitude of tools for string manipulation. This makes string manipulation easy for storing the data in sensor-specific comma-separated values (CSV) files and updating configuration files when sensors are added.

4.2.2 InfluxDB and Grafana

InfluxDB was selected for its superior read and write times and its compatibility with the Raspberry Pi as a database. In comparison to a CSV file storage system, a database like InfluxDB has faster read speeds as the number of data sets collected grows. Among the database solutions, InfluxDB lacked package dependencies when installing it on the Raspberry Pi 3, making it easy to set up on the native Raspbian OS.

Grafana was chosen for its ability to customize its layout, limit processing load on the Raspberry Pi, and adopt external tools with ease. Like InfluxDB, Grafana has no overhead concerning dependencies on the RPi3, and taking the limited power of the RPi3 into consideration, Grafana pushes the processing of the graphs to the users' devices, considerably lightening the processing load on the RPi3. Grafana is able to request specific data sets from InfluxDB, and a user with permissions can log in to Grafana to organize these data sets into more coherent groups before presenting the graphs to other users. Finally, Grafana is able to aggregate data over time to perform more complex analytics and supports the use of plugins that enable it to use the functionality of other tools, substantially increasing its power and flexibility.

5.0 TEST AND EVALUATION

The team ran tests on the system to understand its limits and to prove the prototype met the specifications given by the UTSVT. The system was separated into four different types of tests: data acquisition tests, data transmission tests, data display tests, and full system tests. The follow section will explain the data acquisition testing done on the Nucleo-144 microcontrollers and the

RPi3. The next section will detail the tests run on the UDP data transmission and InfluxDB capabilities. The data display section explains the tests run on Grafana's limits and functionality. The final section explains the tests run on the system when it was fully assembled.

5.1 Data Acquisition Testing

This section describes tests that were conducted to evaluate the system's performance when gathering sensor data and transmitting it across the solar car. As explained in the design solution, the in-car acquisition reads sensor data from the installed sensors and sends this data to the in-car RPi3 via a CAN bus. To ensure that the system reliably provides the most up to date data, a maximum data sample rate test was conducted alongside a data overflow test. To examine the robustness of the CAN bus, we performed a device boot order test and the maximum CAN message frequency test. Finally, the power consumption test was completed to confirm compliance with the power budget outlined in the performance specifications.

5.1.1 Maximum Data Sample Rate Test

Data acquisition in the Nucleo is organized into two parts: data sampling portions and a data collection portion. Given that there is a central memory location holding the sensor data, there existed shared memory that dictates the current sensor data states. To prevent inadvertent write after read structural hazards, a mutex was provided as a blocking mechanism. For more information on the overlying structure, refer to Figure B.2 of Appendix B.

Due to this structural overhead, the purpose of this test was to find the maximum sampling rate that can be pushed through the software pipeline. This test achieved this by modifying the data sampling portions to update the sensor data at increasing faster sample rates and monitoring the overall duration to update the sensor data in the shared memory. The data sampling thread compared the last time it sampled the last sample timestamp with the system timer clock to see if it needed to read the single sensor's data. If the system time was greater than the time needing to sample, then the acquisition thread read the data and push this data to memory. However, this push to data memory could be delayed if the collection portion is using the shared memory by locking the mutex. In this test, this sampling process loops collected 25 samples in total, and used the overall duration to calculate the structural sample rate. If this overall sample rate agreed with the

desired sample rate, then it was determined that the structural overhead does not impede data collection.

5.1.1.1 Test Results

After initial testing, the time to read the sensor data without storing the data took 1800 µs. This provided a theoretical maximum data rate of 556 Hz, but as seen in Table 3 below, as the data sampling rate exceeded 100 Hz, the time of the additional overhead prevented the system from maintaining the desired rate. This was a fifth of the theoretical maximum data rate without overhead, which was a much lower sampling rate than expected.

Table 3. Data Sampling Rate Timings

Data Sample Rate	Desired Period	Duration for 25 Samples	Actual Sample Period
1 Hz	1000 ms	25109 ms	1004 ms
2 Hz	500 ms	12720 ms	508 ms
10 Hz	100 ms	2595 ms	103 ms
25 Hz	44 ms	1095 ms	44 ms
100 Hz	10 ms	359 ms	14 ms
166 Hz	6 ms	360 ms	14 ms
NOTE 1 - These results are given for this set up using 1 sensor connected via a 400 Hz I2C bus. NOTE 2 - No printf/log statements are included within the time recording section to prevent distortion of time.			

5.1.1.2 Results Analysis

The mutex locking mechanism mentioned in the design implementation section severely limited the system's ability to pull sensor data. This was confirmed when running the same test without starting the collector data handling thread. This extra unit test case yielded an actual sample rate of 147 Hz when configured at a sensor sample rate of 166 Hz, which was much closer to the actual desired functionality. After observing this, the team restructured data read protection as described earlier in the report.

5.1.2 Data Overflow Test

The purpose of this test was to show that the CAN library was able to support overflow conditions in the native Nucleo CAN hardware. The test began by setting the hardware to be in a local CAN loopback mode and creating a single static CAN message. It read the number of overflows that were generated. Next, the program sent CAN messages and checked to see that the number of overflows did not increase. In theory, sending three or fewer messages within a CAN frame period would not generate an overflow but transmitting four or more CAN messages within the same period would generate an overflow.

5.1.2.1 Test Results

After several attempts to fill the mailboxes with CAN messages, the library failed to generate an overflow. The CAN message write function discarded data when the mailboxes were full. In addition, the source code provided access to the number of generated overflow generation conditions but failed to increment this number.

5.1.2.2 Results Analysis

The CAN message's write function seemed to always report a successful operation. The code under development did not return an error if the buffer was full and even threw away data in the case that it was full. This test determined that the design solution required its own wrapping CAN controller class that handled the CAN transmission, buffering, and overflow control.

5.1.3 Device Boot Order Test

The purpose of this test was to ensure loss of power and the boot order would not affect the reliability of the communications. All of these tests were performed with a CAN message being sent every second. We tested the following boot order options to observe the effects of the boot order on data collection. First, we provided power to both the Nucleo and the Raspberry Pi 3 by giving an external 5V source to the 5V pins at exactly the same time. We then provided power to the Nucleo 15 seconds before the Raspberry Pi 3 and vice versa. After a connection was verified, we cut the power to the Nucleo to simulate a blackout and powered the Nucleo back on. We repeated the simulated blackout with the Raspberry Pi 3. To ensure damage would not be done to

any of the devices during these tests partial loss of power tests were omitted. This test was performed five times for consistency checks.

5.1.3.1 Test Results

During every single test, the devices were able to eventually reestablish a connection without having any long term effects on the performance. However, some tests proved to cause losses in data. When both devices were powered on in parallel, The Nucleo started sending data more quickly than the RPi3 could start the script to catch the data and initialize the interfaces. Four CAN messages were lost on average over the course of five iterations as shown in Table 4. The boot order tests with delayed start ups caused no issues with the CAN bus connection. Eight CAN messages were lost on average due to the RPi3's boot up time. For the blackout simulations, the connection performed as expected with logging coming back four seconds after the Nucleo initialization. The RPi3 on the other hand was not able to get messages for sixteen seconds after performing a system reboot and was not able to get messages for nine seconds after powering back up.

Table 4. Device Boot Order Test Observations

Boot Order Type	CAN Messages Lost
Parallel Boot of Nucleo and RPi3	4
Nucleo Boot First	8
RPi3 Boot First	0
Complete Loss of Power for Nucleo	0
Complete Loss of Power for RPi3	9
NOTE 1 - The Nucleo boot time was 1 second. NOTE 2 - The RPi3 boot time was 9 seconds.	

5.1.3.2 Results Analysis

These tests showed that the system is able to become operational in less than 10 seconds after any form of power loss. The team was concerned a boot order or blackouts would cause issues in the

transfer of data. This test proved that the system is robust enough to handle any form of interruption with the modular devices.

5.1.4 Maximum CAN Message Frequency Test

The CAN maximum bus frequency test's purpose was to find the frequency threshold for the bus's functionality. The test was conducted by increasing the frequency setting based on common bitrates while measuring the bus lines. The common bitrates ranged from 10000 to 1000000 bits per second. The frequency setting was updated by changing the CAN transceiver frequency on the Nucleo-144 and the RPi3. After changing the frequency, the CAN high and low bus lines, CAN transceiver pins, and RPi3 were analyzed while sending a message between devices.

5.1.4.1 Test Results

Further analysis of the testing results concluded that when the CAN frequency was set to 800 Kb/s the CAN bus was more successful than the 1 Mb/s frequency setting. This was confirmed by the received acknowledge signal and the message data measurements on the CAN high and low bus lines as shown in Figure D.1 of Appendix D. The CAN bus was unable to reliably send messages at 1 Mb/s as shown in the Figure in D.2 of Appendix D.

5.1.4.2 Results Analysis

The CAN bus was unsuccessful with frequencies that were expected to pass based on the CAN protocol frequency ratings. The CAN bus is rated for 1 Mb/s according to the Texas Instruments CAN datasheet [2]. As seen in Figure D.1 of Appendix D, the signal shows the whole message was sent at 800 Kb/s. However, when the frequency setting was increased to 1 Mb/s, the only signal that was caught was the acknowledge signal shown in Figure D.2 of Appendix D.

5.1.5 Power Consumption Test

The power consumption test provides information about the wattage the data acquisition system is drawing during sensor gathering. The test was done using a DC power source providing 5 V and a clamp at 600 mA to avoid faulty connections damaging the system. We connected the 5 V source to the power lines of each respective component separately and measured the current draw of the device. After the modular tests, we tested the full system and increased the clamp to 1100 mA. For

devices with large fluctuations in current draw, we recorded their current draw at one second intervals for thirty seconds and averaged the current draw over that time.

5.1.5.1 Test Results

The test results explain how the modules reacted during the thirty seconds of power testing. During CAN message sending, the Nucleo consumed 154 mA as seen in Table 5 after oscillating 10 mA after initially booting up the Nucleo. The CAN bus takes a constant 13 mA to keep the bus functional. The RPi3 oscillated between 120 mA and 507 mA over thirty seconds. This RPi3 has HDMI, Wifi, Bluetooth, and SSH enabled and running during this test. The RPi3 averaged 242 mA over the thirty seconds of collecting data. The full system averaged 752 mA after oscillating during the RPi3 boot sequence.

Table 5. Device Under Test Current Readings

Device Under Test	Current (mA)
RPi3	242
CAN Bus	13
STM Nucleo-144	154
Full System	752

5.1.5.2 Results Analysis

The power consumption tests prove that the system falls within the specifications given by UTSVT. UTSVT required our whole system to consume less than 10 W and our full system fluctuated to 5.5 W during the RPi3 boot process and then settled at 3.76 W after initialization finished. The majority of the power consumption was from the RPi3 with only two Nucleo-144 connected.

5.2 Data Transmission Testing

This section describes tests that were conducted to evaluate the system's performance when sending and receiving data over the existing wireless network. As previously mentioned, the

system transmits data between the solar vehicle and a race pit via a wireless network connection. Connection to the network may be temporarily lost due to road conditions, so evaluating the system's ability to reconnect to the network was necessary in the intermittent network connection test. This network also has a bandwidth maximum of 0.5 Mb/s. In order to confirm the system's ability to fully utilize the bandwidth with the UDP protocol, testing of the maximum UDP packet size and maximum UDP packet transmission rate was conducted.

5.2.1 Intermittent Network Connection Test

The intermittent network connection test confirms the system's ability to reconnect to a wireless network in the event that connection is temporarily lost. This simulates the possible inconsistent connection during a race. To perform this test, we sent a packet with a timestamp and integer at a rate of 1 packet per second between the two RPis. After the pit RPi successfully received packets from the car RPi for 50 seconds, we toggled power to the network router to terminate and reinstate the wireless connection. We then recorded the timestamp of the next packet received by the UDP client and compared the value to the timestamp at transmission. We determined a successful reconnection time between network power loss and next packet receipt to be less than 45 seconds.

5.2.1.1 Test Results

Performing the test three times provided the results shown in Table 6 on the next page, which also includes the required time for the network router to reboot and the time required to establish a connection while accounting for the network router reboot time. With three test iterations, we see an average connection reset time of 43.66 seconds.

Table 6. UDP Transmission Connection Timings

Test Iteration	Connection Reset Time (seconds)	Router Boot Time (seconds)	Actual Reset Time (seconds)
1	43	19	24
2	42	19	23
3	46	19	27
Average	43.66	19	24.66

5.2.1.2 Results Analysis

Given the average connection time under 45 seconds, we determined that the current UDP transmission system met the project needs. Noting that the connection time includes a router reboot time not present in a race scenario, we expected connection times averaging just below 25 seconds in normal usage. From these results, we concluded that a connection that cuts out more frequently than once every 30 seconds prevents the data display system from functioning effectively, as it does not receive near real-time data.

5.2.2 Maximum Packet Size Test

The purpose of the maximum packet size test was to determine the limitations of the UDP socket in regards to packet size. The packet size was increased to 9200 bytes and was tested to see if the client could receive a message of this size. The 9200 byte packet size was sent to the client ten times to ensure functionality. This was repeated four times at 9200 byte, 1024 byte, 512 byte, and 256 byte packets. We ceased testing at a packet size of 9200 bytes due to the unlikeliness of using packets of this size, which could be cut off mid transmission due to an intermittent connection.

5.2.2.1 Test Results

The RPi UDP client successfully received all 10 packets without loss for all tested packet sizes. Table 7 documents these results, beginning with the maximum tested packet size of 9200 bytes.

Table 7. UDP Packet Message Reception Results

Packet Size (bytes)	Packets Sent	Packets Not Received
9200	10	0
1024	10	0
512	10	0
256	10	0

5.2.2.2 Results Analysis

The UTSVT's transmission network has a limited bandwidth of only 0.5 Megabits per second. Due to recent UTSVT experiments with the ZigBee protocol, the project developed for UDP packets no greater than 100 bytes in order to minimize potential redesigns if the transmission method changes and creates new system constraints. Having the ability to send a 9200 byte packets indicates that the 100 byte message is more than capable of being received by the client. Additionally, with the UDP packet size determined we determined the maximum packet rate r_p allowed by the network bandwidth B through the equation

$$r_p = 0.125B/s, \quad (1)$$

where s is the maximum packet size in bytes. In this operating environment, we saw that B is 0.5 Mb/s and s is 100 bytes, resulting in a maximum packet transmission rate of 625 packets per second. With this information, we tested the system's ability to send packets at this rate in the maximum packet transmission rate test.

5.2.3 Maximum Packet Transmission Rate Test

The purpose of the maximum packet transmission rate test was to test the rate that different size packets can effectively be received by the pit RPi. The size of the packet was manipulated to be 512 bytes, 256 bytes, and 128 bytes. For each different size packet, 10,000 packets were sent from one RPi to another RPi. This test was completed 10 times for each packet size. The number of packets received was recorded for each test.

5.2.3.1 Test Results

The test results for each packet size is included in Appendix E. Receiving a packet size of 512 bytes with no wait time showed the receiver lost an average of 168 packets out of 10,000. Comparing the 512 byte packet to the 256 byte packet with no wait time showed the 256 byte message lost an average of 116 packets out of 10,000. The 128 byte packet received every packet that was sent to it, with an average of 0 packets lost out of 10,000.

The 512 byte packet performed at a better rate with a 5 microsecond wait between each packet and an average of 28 packets lost out of 10,000. The 256 byte packet received more packets with an average of 14.1 packets lost out of 10,000. The 128 byte packet received all the packets sent to it, with an average of 0 packets lost out of 10,000.

Comparing the 5 microsecond wait to the 10 microsecond wait shows the 256 byte packet and the 128 byte packet size received all of the packets sent to it, with an average of 0 packets lost out of 10,000. The 512 byte packet size lost an average of 9 packets out of 10,000.

5.2.3.2 Results Analysis

The maximum packet transmission rate test showed that a bigger packet size requires a slower sending rate to receive all of the packets. Additionally, once the test packet size decreases below a threshold of 128 bytes, transmission is consistent provided a functioning network. Given the recommended packet size of 100 bytes or less, we saw that the maximum transmission rate of 128 packets per second for that size was not only possible, but limited by the network bandwidth and not the system itself.

5.3 Data Display Testing

The data display testing section examines the tests performed to confirm the data display subsystem's compliance with the project specifications identified earlier in this report. The data capacity test validated InfluxDB's ability to meet the continuous active time required of the project. In order to confirm the available data transmission rate from the network into InfluxDB, we performed the data inflow test. Lastly, we performed the concurrent user test to verify Grafana's support for multiple users as required in the user interface specifications.

5.3.1 Data Capacity Test

The data capacity test indicates the Raspberry Pi 3's capability to log data to InfluxDB for the specified continuous active time of 8 hours. To confirm this capability, we sent data at a rate of 2.25 Kb/s into InfluxDB for over twice the specified continuous active time. Additionally, we recorded the virtual and resident memory usage in kilobytes every 5 seconds for 16 hours and 20 minutes with a shell script to monitor InfluxDB's memory usage. For this test, we defined success

as InfluxDB's ability to function for twice the minimum continuous active time, a total of 16 hours.

5.3.1.1 Test Results

Performing the data capacity test provided InfluxDB's memory usage on the RPi3 as shown in Figure 3 on the next page. Virtual memory usage is shown as a dark black point near the top of the scatter plot. Resident memory usage is shown as a light grey point near the x-axis of the plot.

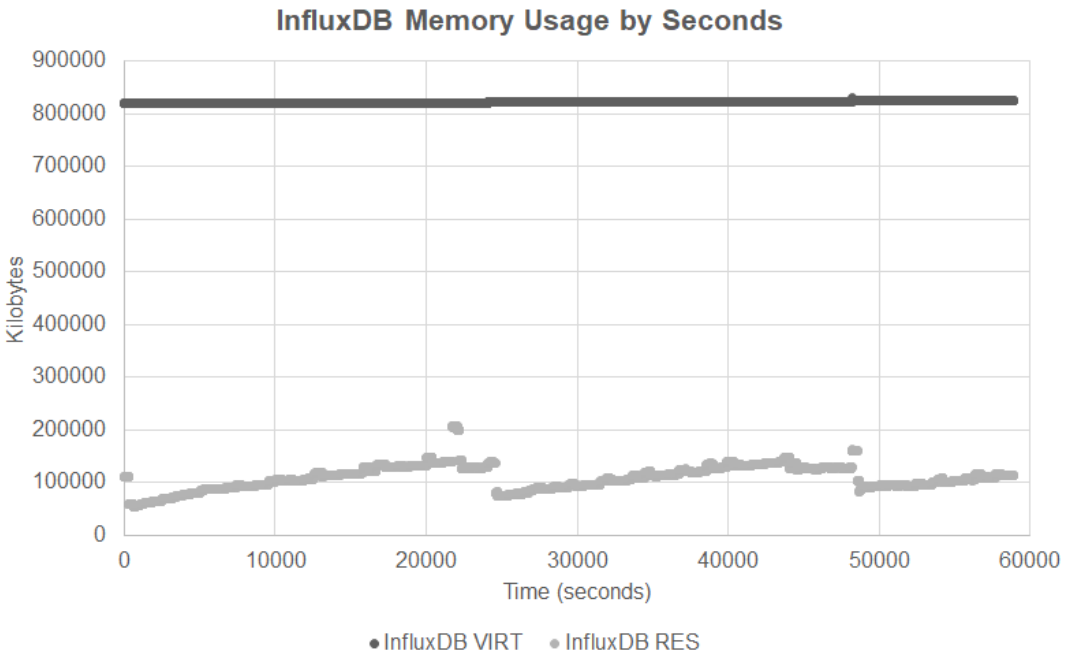

Figure 3. InfluxDB Memory Usage by Seconds

5.3.1.2 Results Analysis

Since InfluxDB successfully ran for the full 16 hours, the data display system met our criterion for success. More importantly, we were able to predict the virtual and resident memory usage past the running time of the data capacity test. Virtual memory usage during this test started around 800,000 KB and slightly increased every time the resident memory usage dropped from a peak, shown at approximately 2500 and 4900 seconds. Since there were no spikes in virtual memory usage, we took no action to change virtual memory usage. Resident memory usage increased to roughly 150,000 KB before falling below 100,000 KB and repeating the process. We assumed that

the RPi3 will repeat this process as necessary, ensuring that we do not exceed the amount of available resident memory. We also noted that Grafana and Python displayed negligible usages capacity-wise, with Grafana usage being dependent on the number of users. Overall, we confirmed that the data display system will meet the project specifications by running for more than the minimum continuous active time.

5.3.2 Data Inflow Test

The data inflow test indicates the Raspberry Pi 3's ability to receive data from the Python UDP receiver. We defined a successful test as one that shows InfluxDB, and by extension the RPi3, can receive two kilobits per second. We determined two kilobits per second was a sufficient metric since this would be handling approximately three of our UDP packets per second. To perform this test, we ran a Python logging script that sent a set amount of randomly-generated data to InfluxDB every second. If InfluxDB can handle this script at two kilobits per second, the RPi3 passes our test. We ran this test for 8 hours at the following data rates: 0, 0.625, 1.75, 2.1875, 3.125, and 9.375 kilobits per second. In order to track system performance, we used the same shell script as the data capacity test, providing us with the percent usage of one of the four CPU cores and the memory usage percentage taken at five-second intervals.

5.3.2.1 Test Results

By conducting the data inflow test, we collected resource usage for Grafana, InfluxDB, and Python on the RPi3. For every data input rate tested, InfluxDB remained did not crash during the full 8 hours. We have provided the average resource usage for InfluxDB, Grafana, and the Python logging script in Figures F.1, F.2, and F.3 of Appendix F. Each average is calculated from the 5 second interval measurements taken during the 8 hour test runs.

5.3.2.2 Results Analysis

All results gathered during the data inflow test suggest that the usage of an RPi3 in our design will meet the specifications of the UTSVT. InfluxDB usage showed a linear growth with increasing data input rate, with CPU usage at the 2.1875 kilobit per second mark remaining below five percent. Five percent of one CPU core is acceptably low for our design, and higher integer rates showed similarly low CPU usage. Memory usage is more likely to cause a test failure in the future,

as the slope of increase is higher than CPU usage. We saw that 9.375 kilobits per second averages at 13%, so usage would need to increase by an order of magnitude over our test before our design would expect failure. Grafana usage was extremely stable over integer rate, suggesting that the data input rate change has little impact on Grafana performance. Python, however, also showed a linear increase in CPU usage, but the the starting point is so low that InfluxDB would fail well before the Python logging script. Overall, we saw no necessary changes to the system design as a result of this test.

5.3.3 Concurrent User Test

The concurrent user test shows Grafana's ability to display visual data to multiple users. For this test, two kilobits per second were sent to InfluxDB, and five users used Grafana simultaneously. The users also attempted to refresh Grafana as quickly as possible so as to query InfluxDB for more data and put stress upon it. If neither InfluxDB nor Grafana services crash by the end of the test, it is successful. To log the performance data, we used the same logging script from the data capacity and data inflow tests.

5.3.3.1 Test Results

While performing the concurrent user test, we gathered CPU usage data on the RPi3. InfluxDB occupied more of the CPU than Grafana, which remained around 0% CPU usage as shown in Figure 4 below. Users began Grafana access between 0 and 150 seconds, and users began manually refreshing Grafana at 500 seconds.

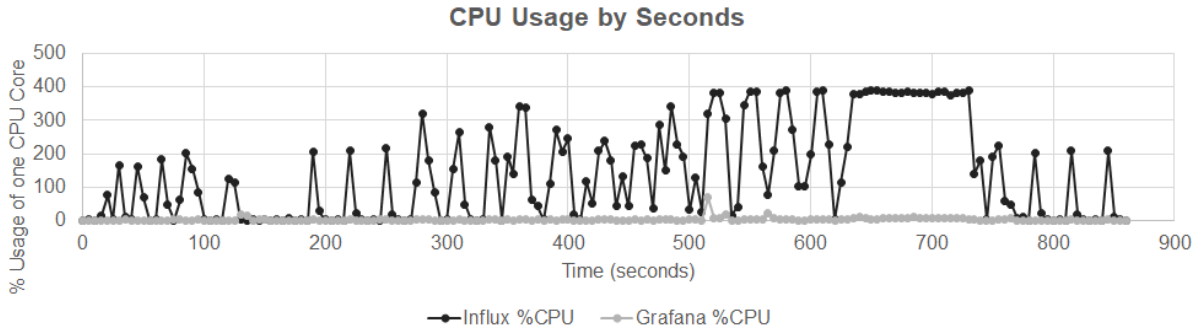

Figure 4. CPU Usage by Seconds

5.3.3.2 Results Analysis

Through the results found in the concurrent user test, we concluded that InfluxDB and Grafana met the specifications mandating support for multiple users. The RPi's CPU usage was stretched to its maximum, but InfluxDB did not crash - it only slowed down. After processing all the manual refreshing, CPU usage decreased to normal levels. To prevent heavy CPU usage causing InfluxDB performance issues, we limited the number of users that can access Grafana simultaneously through the iptables utility program.

5.4 System Testing

After completing the module testing described earlier in this section, we repeated the module tests with the integrated modules to confirm the functionality of the system. Since the module tests were comprehensive, the only additional test required was the validation of general system functionality. As a result, we saw no changes in test results compared to the module tests and we confirmed the successful operation of the design.

6.0 TIME AND COST CONSIDERATIONS

We met most time and budget constraints of the project. We were given until open house to complete the project, and we were not given a budget due to the organization of the project. The major scheduling roadblock was the implementation of the CAN bus, and we experienced no major budget issues.

6.1 CAN Implementation

The implementation of CAN took longer than expected due to many reasons. First, CAN was new to all members of the team, which meant that there was a learning curve necessary to understand and implement CAN in our system. Second, CAN only works when two or more connected modules are configured for CAN so significant development was needed before we were able to confirm successful functionality. Third, the CAN bus does not give any feedback as to where errors occur so debugging was difficult. Finally, we experienced an unexpected hardware failure because one of the custom-ordered daughterboards arrived defective because it had connection issues that caused multiple modules to not communicate with each other properly. Modules failed to detect when the CAN bus was in use so data would continuously be sent and overwritten by

other modules, eventually overloading the CAN bus. This delayed the development of CAN by two weeks.

6.2 Bill of Materials

The base system without any sensors costs $199.52 as seen in Figure G1 of Appendix G. Theoretically, one should be able to build upon the base system with compatible sensors to create a complete system. The incurred cost of the project over the two semesters was $390.79 as seen in Figure G2 of Appendix G. This total includes all spare and unused parts that were ordered as well as the prices of the two sensors that were included for the open house demo system.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

This project considered the safety of the driver when developing the new data acquisition system. The data acquisition system will function while the solar vehicle is moving with a driver, so an improperly-designed system could result in a danger to the driver. Because of this risk, the hardware and software created in this project does not interfere with the safe operation of the solar car or the safety of the driver. The in-car acquisition system is isolated from the rest of the vehicle by connecting only to the low power voltage line and the sensors being read. By limiting these connections, a system or server crash remains a modular failure separate from the system that operates the car. Overall, the solar car and the data acquisition system are effectively two independent systems.

The system inside the car and in the pit should have network security to prohibit any unauthorized access to the wireless network. Since the car has not been assembled, the access security cannot be finalized with our design. The full security of the system is outside the scope of our project since we only connect to the network via an existing Ethernet modem. We leave the full security measures up to the UTSVT to ensure authorized users are the only ones with access to wireless network.

8.0 RECOMMENDATIONS

The following section outlines possible changes to the design solution presented in this report and potential avenues to expand the functionality of the design. If the needs of the UTSVT change in

the future, minor changes to the design may be preferable to a completely new system. The recommended design solution changes section provides options to lower power consumption in the solar vehicle and improve the display capabilities of Grafana in the event either of these topics become concerns. Additionally, the recommended design solution expansion section explains the possible options of adding public Grafana access via the internet and detecting power loss inside the solar vehicle.

8.1 Recommended Design Solution Changes

To improve the power usage of the in-car acquisition system, we recommend two options: replacing the Nucleo-144s in the design with low footprint Nucleo-32 microcontrollers and replacing the RPi3 inside the car with a Raspberry Pi Zero. Since the Nucleo-144 is used as a sensor hub, the excessive amount of GPIO ports are unnecessary for its purpose in the design. We would recommend that the design solution should shift to Nucleo-32 since it uses close to half the amount of current as the Nucleo-144. The Nucleo-32 has less GPIO ports but since we will use the CAN bus, connecting more modules will not affect the functionality of the system. The Raspberry Pi Zero requires less power than the RPi3, but it does not have a built-in Ethernet port. Adding an Ethernet port is possible, but this would increase power consumption as well. Instead, we strongly recommend investigating alternate low power methods of network communication to maximize the benefit of the Raspberry Pi Zero.

To improve Grafana's ability to display GPS location, we recommend changing the InfluxDB implementation to another Grafana-supported database such as Elasticsearch. While InfluxDB responded to queries for GPS location data, the format of the query restricted Grafana's ability to filter out data points. This resulted in a cluttered user display when attempting to view the solar vehicle's current position. Elasticsearch has a built-in geo-point data type, allowing for easier usage of GPS data [3]. However, Elasticsearch installation on the RPi3 involves more overhead than InfluxDB, making this a cautious recommendation.

8.2 Recommended Design Solution Expansion

In order to allow the public to view solar vehicle data, we recommend adding a web server with Grafana and InfluxDB. The current design solution can only display data to members of the

UTSVT inside the race pit. If a stable internet connection in the race pit is available, we recommend forwarding the UDP packets to a web server. Once there, the server can run the same Python logging script to populate InfluxDB. This would allow public users to go to a website for Grafana access instead of a local IP. The pit would have access to a separate server so there interface was not affected by the number of users accessing the public display.

The RPi3 needs to safely shutdown before power is removed otherwise the system has the risk of becoming corrupted. We recommend adding an undervoltage signal to the RPi hat used in the in-car acquisition system to detect when there is loss of power. During the development of the system, we found that power loss during data logging to the SD card has a chance to corrupt the card and ruin the data inside. Detecting low voltage in the power supply would allow the system to safely shutdown prior to power loss, preventing possible corruption. We attempted to use internal GPIO pins in the RPi3 to detect low voltage, but found that the detection was too unreliable to warrant further development. Instead, we recommend adding an external low voltage signal line to the RPi hat used in the in-car acquisition system. This should improve the reliability of detection to a level where further development of power loss detection functionality is possible.

9.0 CONCLUSION

This report explained the completed data acquisition system for the UTSVT. The previous solar vehicle had a slow boot time and limited access to sensor data when the system was running. A new data acquisition system was requested that could gather data from the sensors on the new solar vehicle and display the data in either a race pit or chase vehicle. The main design was split between two main subsystems: the in-car acquisition system that gathers sensor data and the data display system that receives and displays the sensor data. The in-car acquisition system's main components are the STM Nucleo-144, CAN bus, and a RPi3. The data display system's main components are the RPi3, InfluxDB, and Grafana. The CAN bus was implemented due to its high reliability in EMI, while the STM Nucleo-144 was selected for its large number of GPIO pins and processing power. In both subsystems, the RPi3 was chosen for its flexible functionality and ease of use. InfluxDB was selected for its superior read and write times and its relatively low overhead. Grafana was chosen for its customizable layout and the limited load on the RPi. Module tests were performed on both subsystems, confirming the functionality of the data acquisition, data

transmission, and data display portions of the design solution. After the tests were completed, full system's reliability was confirmed in the context of the project specifications. After completing the prototype, the team realized the major setback to the schedule was completing the CAN bus. Since the CAN bus gives little feedback, the team had difficulty finding and fixing the error. This setback put pressure onto the team in the later weeks of the project. Even with the schedule setback, The safety of the solar vehicle driver was prioritized by guaranteeing that the new data acquisition system had no impact on the safe operation of the solar vehicle. The team identified the most relevant recommendations such as replacing the Nucleo-144s with Nucleo-32 microcontrollers, replacing the RPi3 with a Raspberry Pi Zero, and changing InfluxDB to Elasticsearch. Additionally, possible extensions to the design include a public display of the data and adding an undervoltage signal to the RPi to ensure safe shutdown. Overall, the project team provided a completed data acquisition system with clear steps for improvement and expansion.

REFERENCES

[1] Innovators Educational Foundation. (2017, Feb. 1). *American Solar Challenge 2018 Regulations (Rev. A)* [Online]. Available: http://www.americansolarchallenge.org/ASC/wp-content/uploads/2016/08/ASC2018-Regs-External-Revision-A.pdf.

[2] Texas Instruments, "Controller Area Network Physical Layer Requirements", SLLA270 datasheet, Jan. 2008. Available: http://www.ti.com/lit/an/slla270/slla270.pdf.

[3] D. Lee. (2017, Oct. 26). *Worldmap Panel* [Online]. Available: https://grafana.com/plugins/grafana-worldmap-panel.

APPENDIX A – IN-CAR ACQUISITION SYSTEM DIAGRAMS

APPENDIX A – IN-CAR ACQUISITION SYSTEM DIAGRAMS

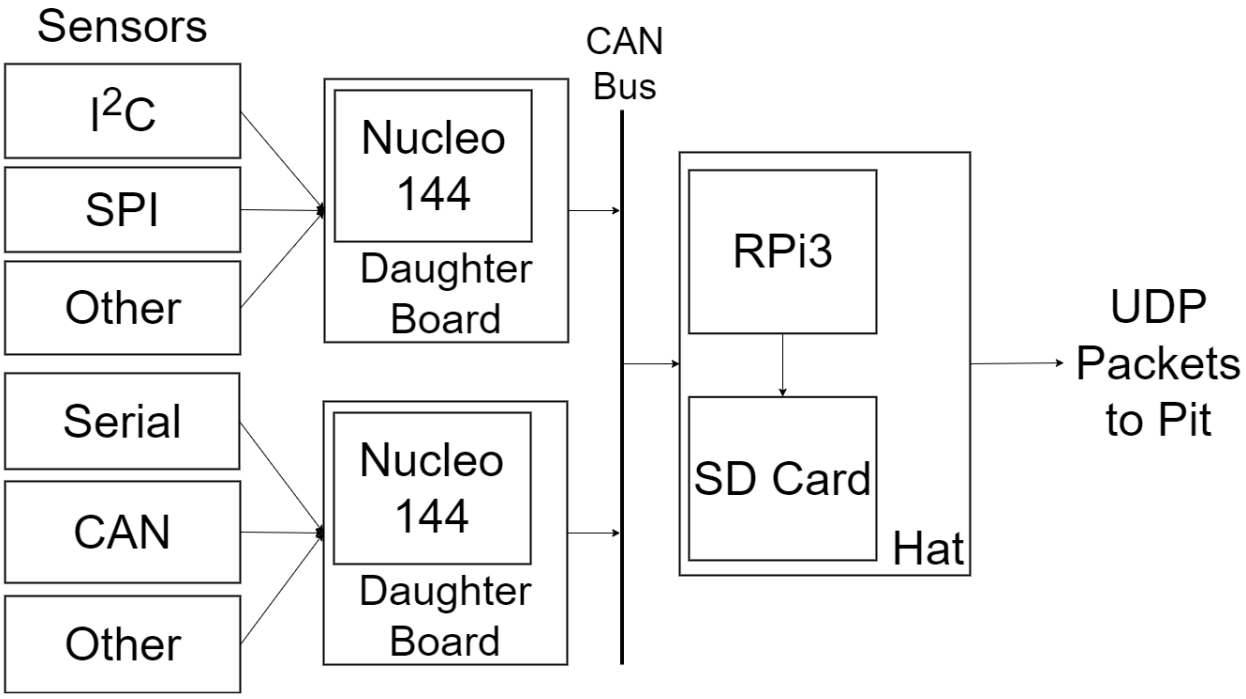

Figure A.1. In-Car Acquisition System Block Diagram

APPENDIX A – IN-CAR ACQUISITION SYSTEM DIAGRAMS

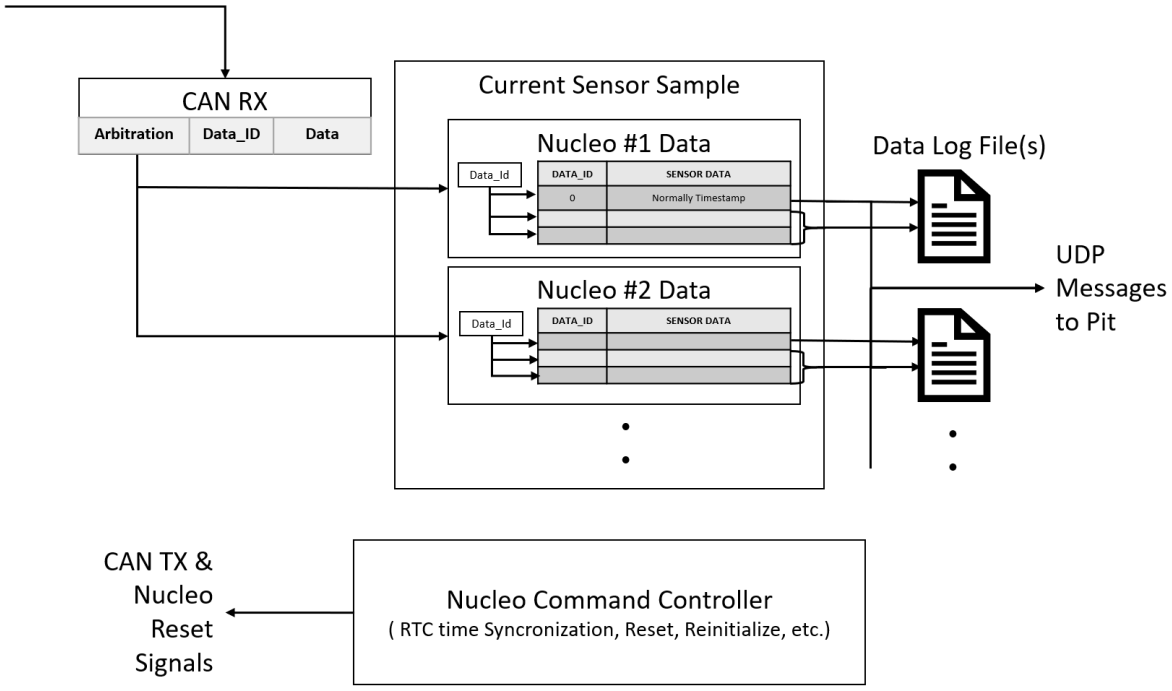

Figure A.2. Software Structure of the In-Car RPi3

APPENDIX B – ADDITIONAL NUCLEO DOCUMENTATION

APPENDIX B – ADDITIONAL NUCLEO DOCUMENTATION

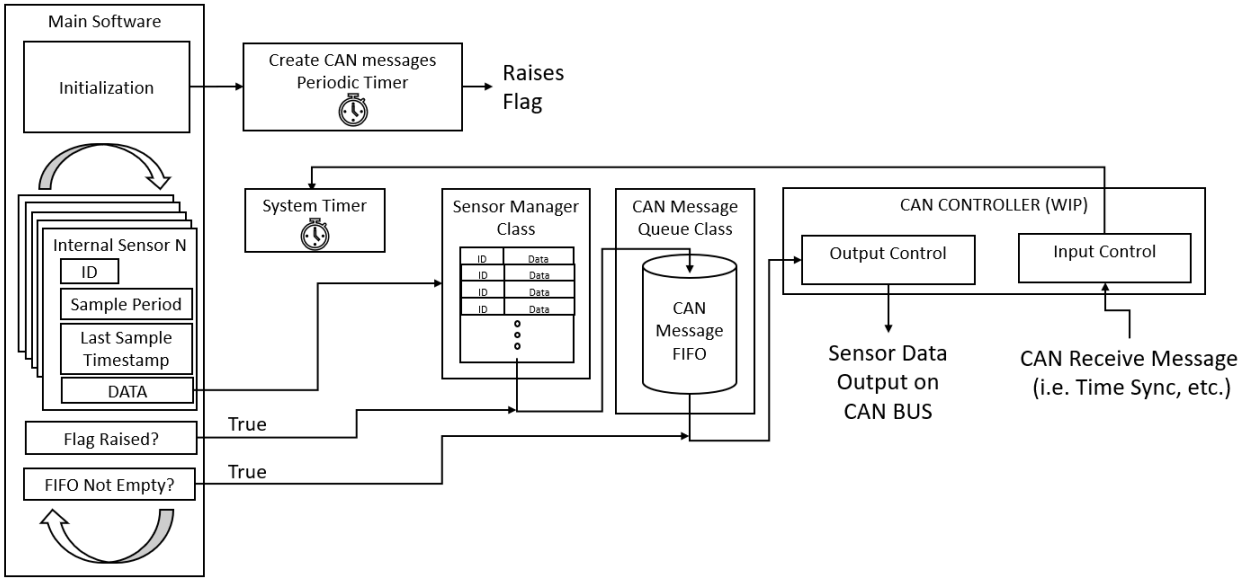

Figure B.1. Final Nucleo Software Structure

APPENDIX B – ADDITIONAL NUCLEO DOCUMENTATION

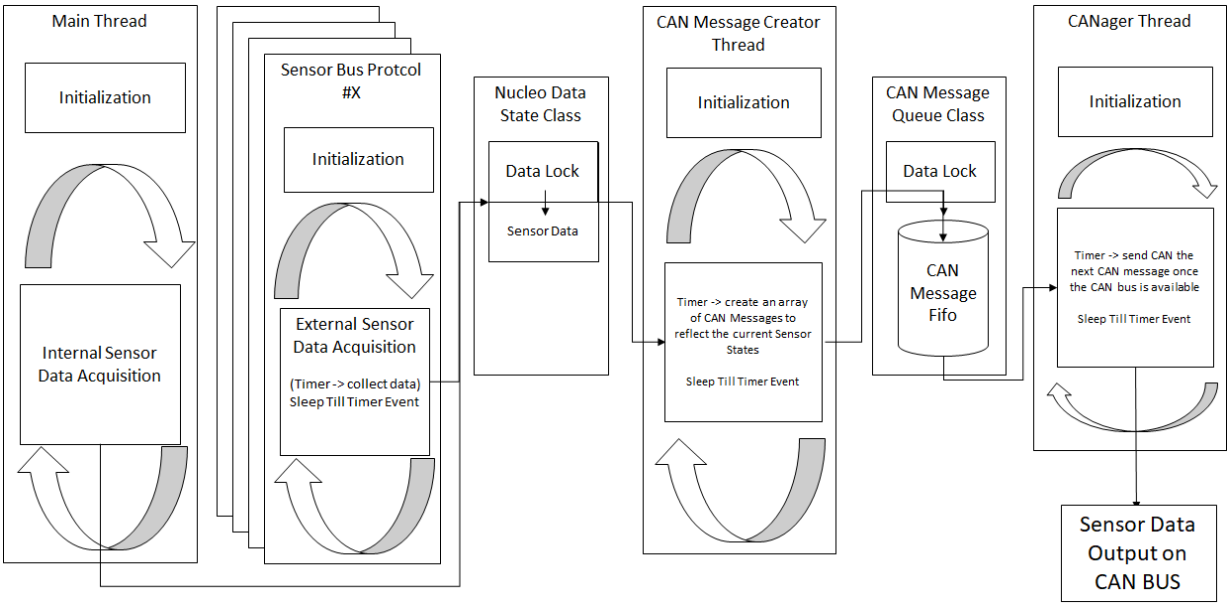

Figure B.2. Previous Threaded Nucleo Software Structure

APPENDIX C – DATA DISPLAY SYSTEM DIAGRAMS

APPENDIX C - DATA DISPLAY SYSTEM DIAGRAMS

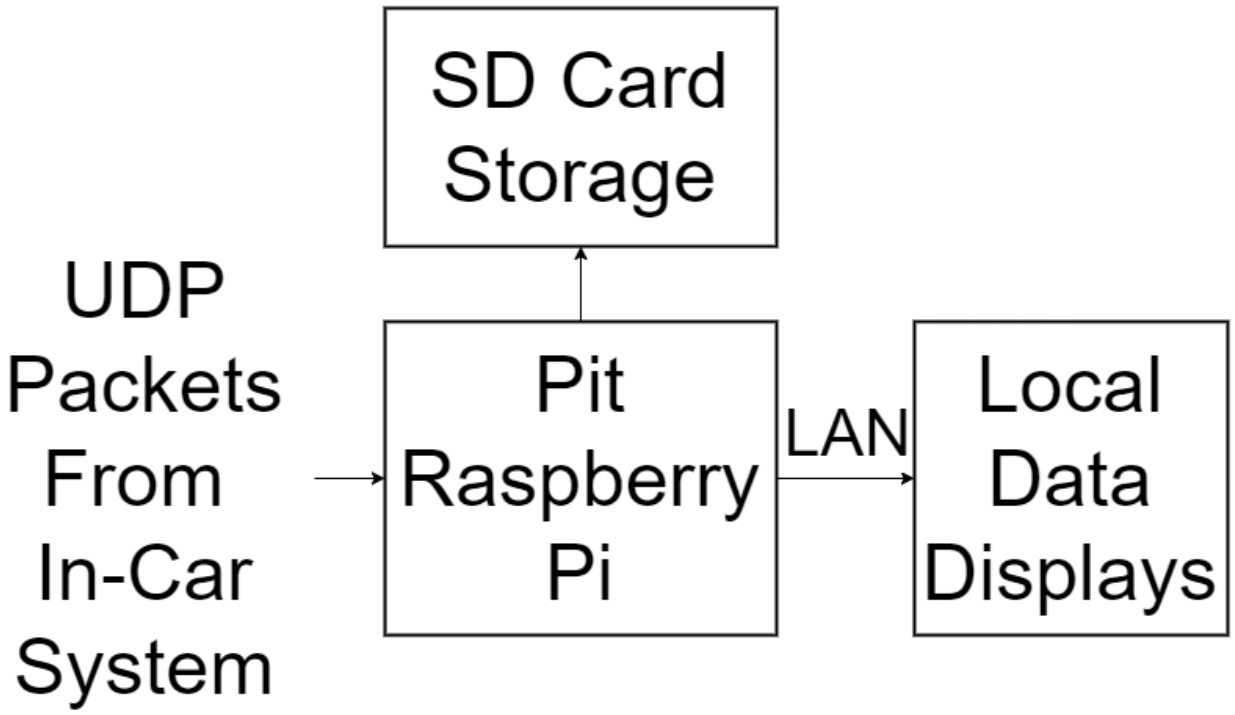

Figure C.1. Data Display System Block Diagram

APPENDIX C - DATA DISPLAY SYSTEM DIAGRAMS

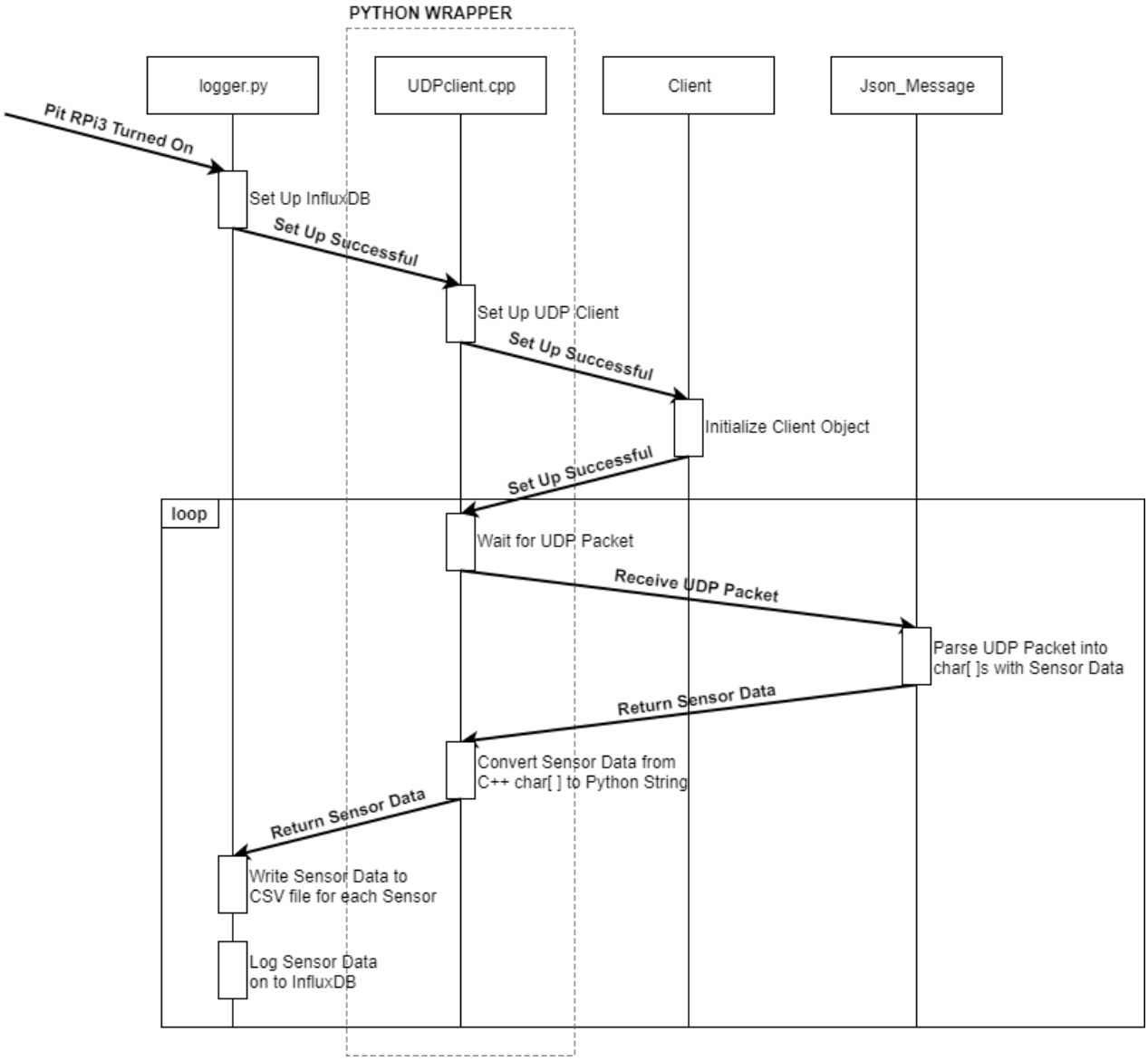

Figure C.2. Pit RPi3 Software Sequence Diagram

APPENDIX C - DATA DISPLAY SYSTEM DIAGRAMS

Figure C.3. InfluxDB and Grafana Interaction

APPENDIX D – CAN FREQUENCY TEST RESULTS

APPENDIX D - CAN FREQUENCY TEST RESULTS

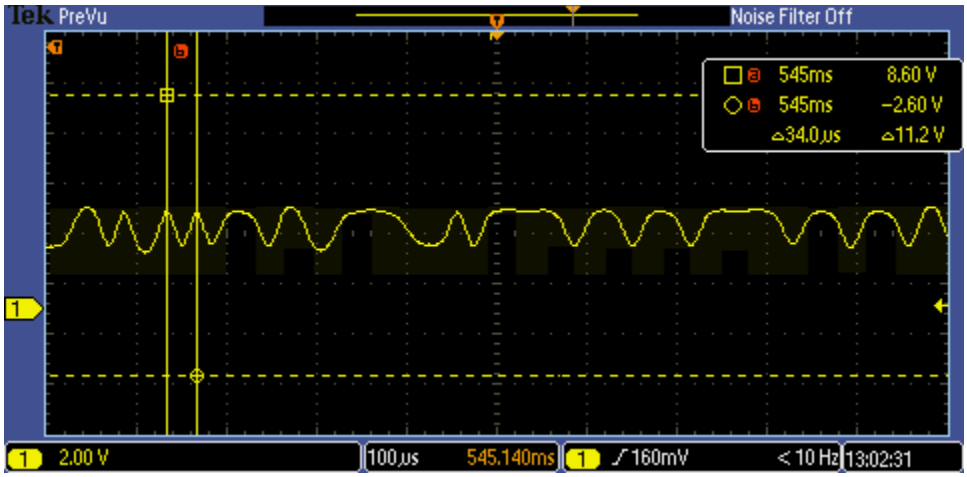

Figure D.1. CAN 800 Kb/s CAN High Oscilloscope

APPENDIX D - CAN FREQUENCY TEST RESULTS

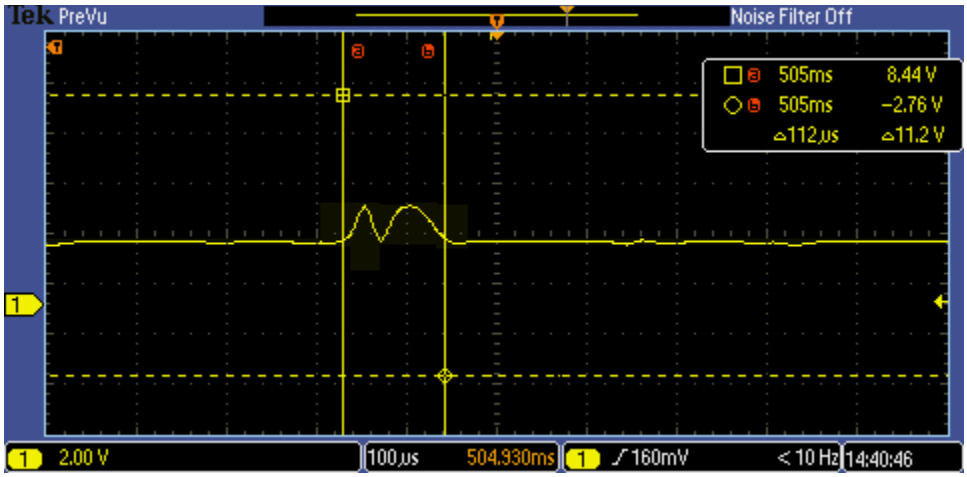

Figure D.2. CAN 1Mb/s CAN High Oscilloscope

APPENDIX D - CAN FREQUENCY TEST RESULTS

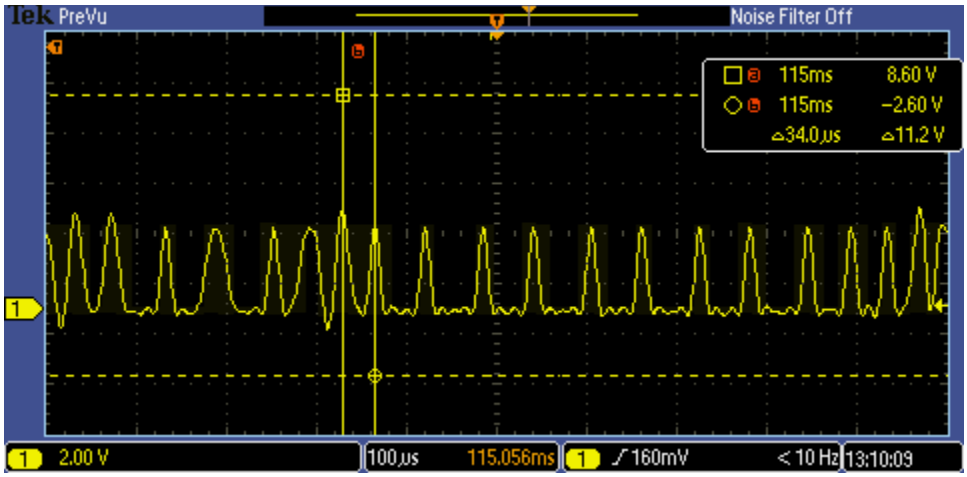

Figure D.3. CAN 800 Kb/s CAN Tx Oscilloscope Reading

APPENDIX D - CAN FREQUENCY TEST RESULTS

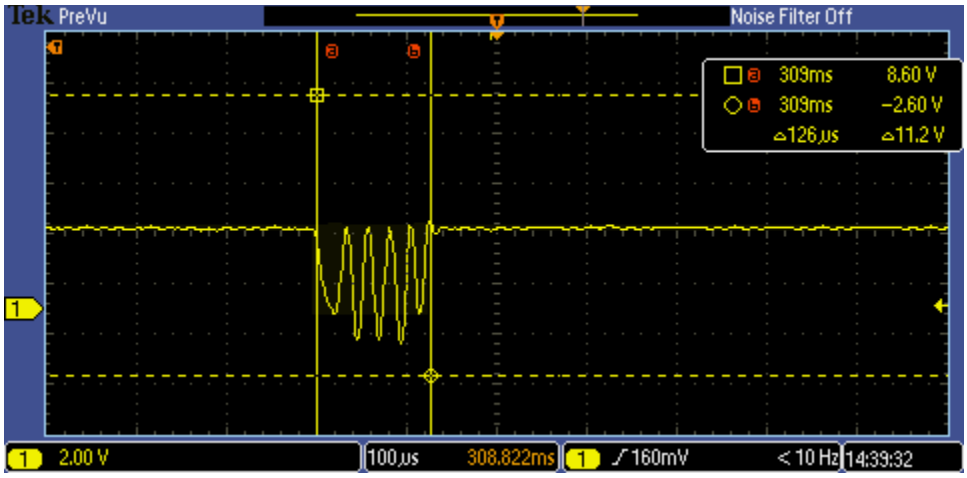

Figure D.4. CAN 1 Mb/s CAN Tx Oscilloscope Reading

APPENDIX E – PACKET TRANSMISSION RATE TEST RESULTS

APPENDIX E – PACKET TRANSMISSION RATE TEST RESULTS

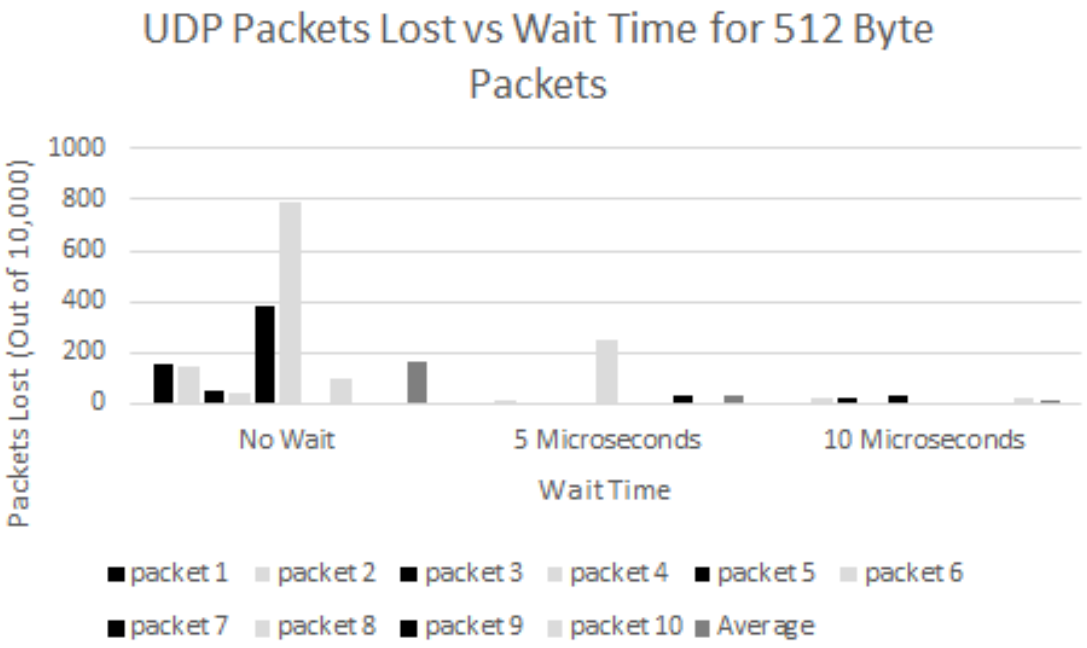

Figure E.1. UDP Packets Lost vs Wait Time for 512 Byte Packets

APPENDIX E – PACKET TRANSMISSION RATE TEST RESULTS

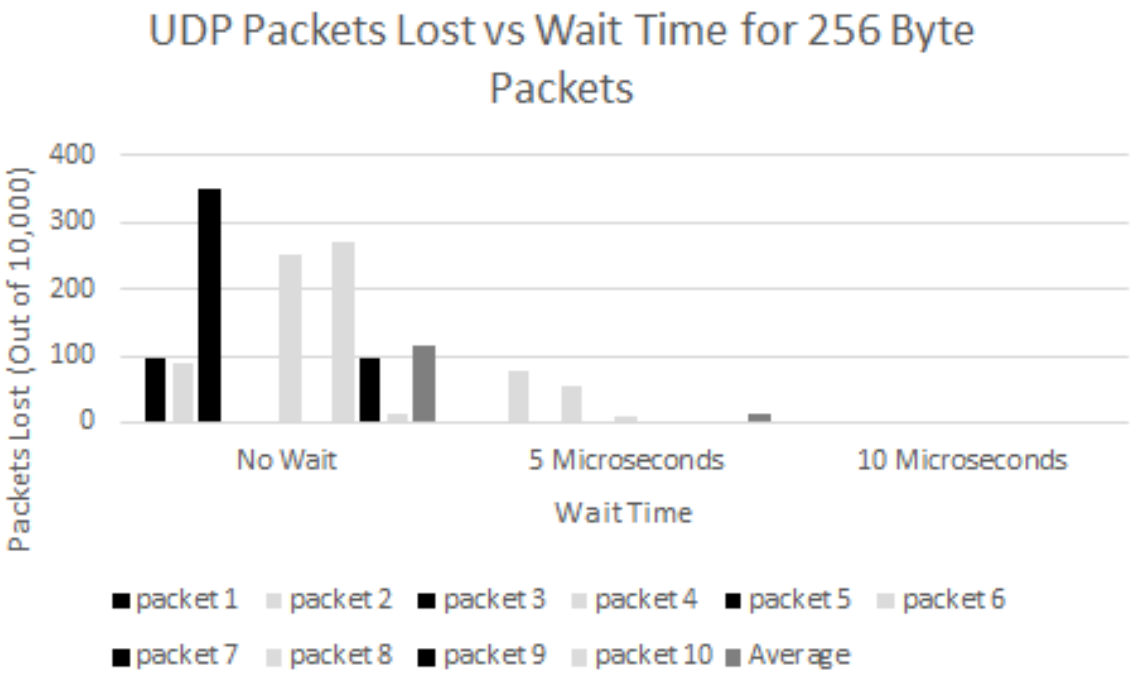

Figure E.2. UDP Packets Lost vs Wait Time for 256 Byte Packets

APPENDIX E – PACKET TRANSMISSION RATE TEST RESULTS

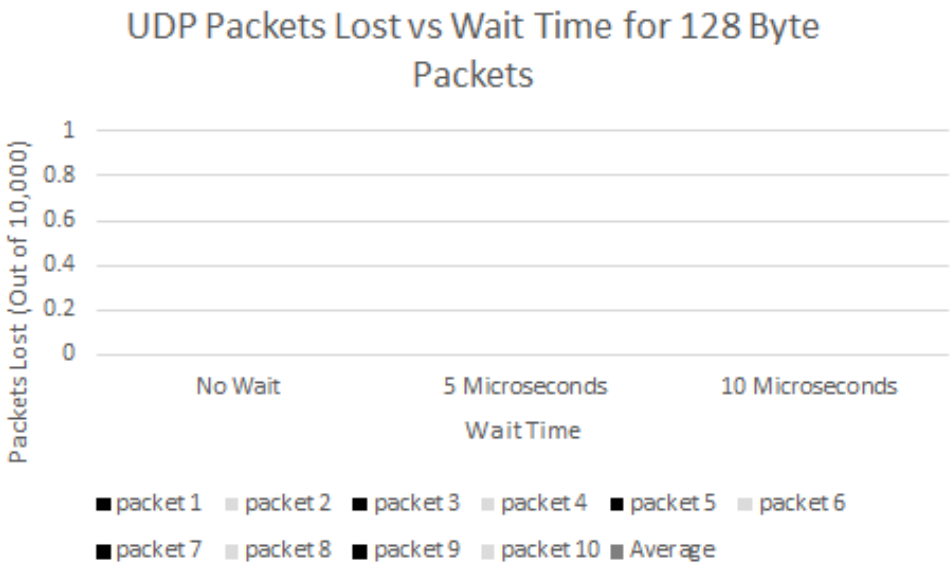

Figure E.3. UDP Packets Lost vs Wait Time for 128 Byte Packets

APPENDIX F – DATA INFLOW TEST RESULTS

APPENDIX F – DATA INFLOW TEST RESULTS

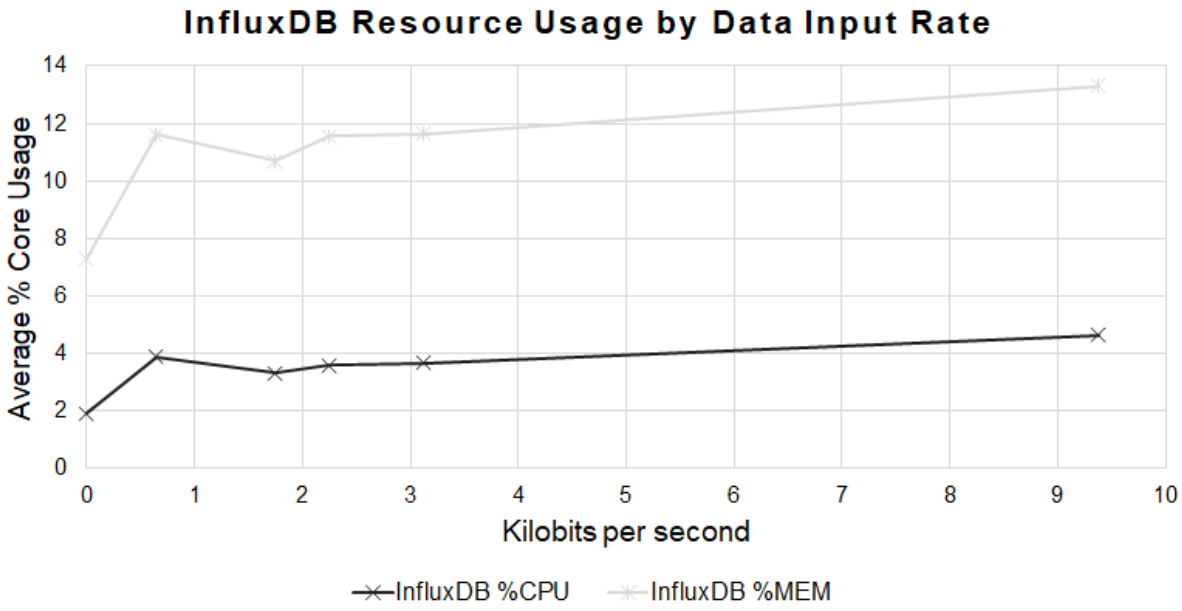

Figure F.1. InfluxDB Resource Usage by Data Input Rate

APPENDIX F – DATA INFLOW TEST RESULTS

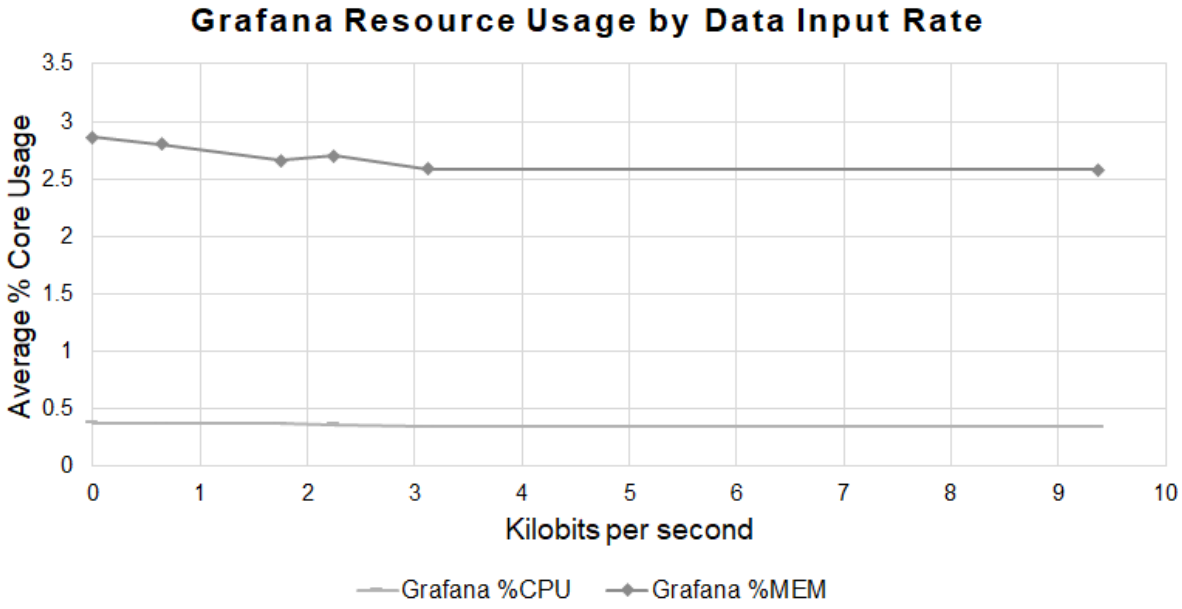

Figure F.2. Grafana Resource Usage by Data Input Rate

APPENDIX F – DATA INFLOW TEST RESULTS

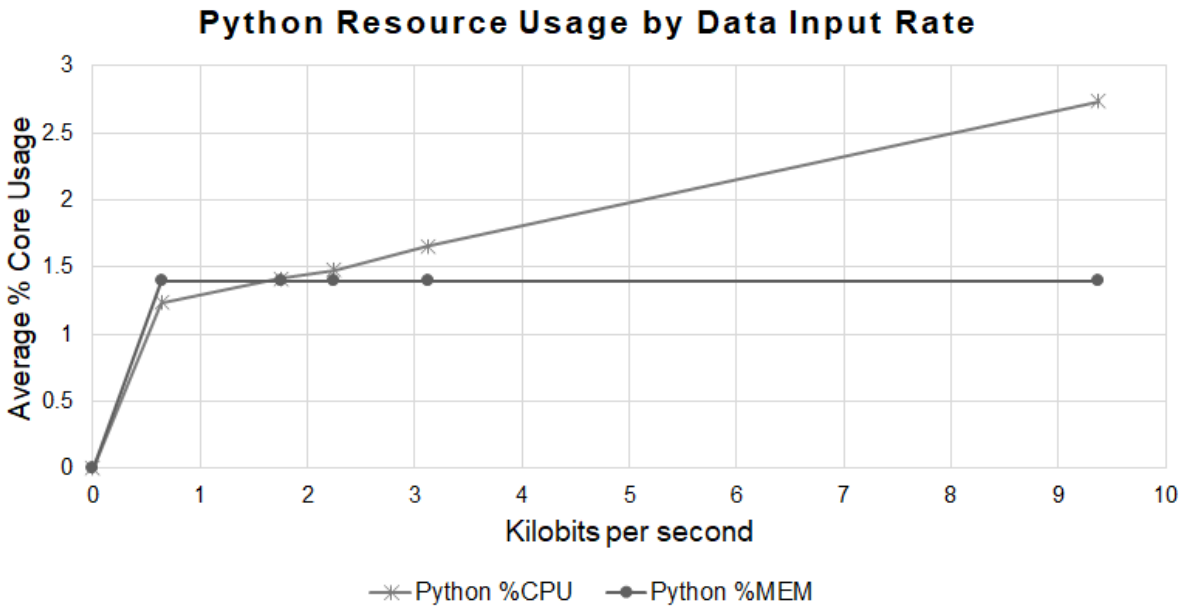

Figure F.3. Python Resource Usage by Data Input Rate

APPENDIX G – BILL OF MATERIALS

APPENDIX G – BILL OF MATERIALS

Table G.1. Bill of Materials for Base Data Acquisition System

Part	Quantity	Cost per Unit	Cost
3M9531-ND 0.1" Pitch Header	2	4.20	8.4
43025-0600 Male Connectors	3	0.43	1.29
43045-0620 Female Connectors	3	3.54	10.62
8 GB Class 10 microSD Cards	2	8.99	17.98
825 Jumper Wires	1	**3.95**	3.95
ADM3054 CAN Interface	2	6.23	12.46
BK-916-CT-ND Battery Holder	1	0.50	0.5
Custom Raspberry Pi Hat	1	5.00	5
Custom STM Hat	2	18.00	36
DS3231 Real-Time Clock	1	8.50	8.5
MCP2515 SPI Interface CAN Controller	2	2.09	4.18
Raspberry Pi 3	2	34.99	69.98
STM32 Nucleo-144	2	10.33	20.66
		Total Cost	**199.52**

APPENDIX G – BILL OF MATERIALS

Table G.2. Bill of Materials Accumulated Over Project Lifetime

Part	Quantity	Cost per Unit	Cost
1005 LEDS Chip Mount LEDs	5	0.24	1.2
2N3904 BJT	4	0.4	1.6
3M9531-ND 0.1" Pitch Header	2	4.20	8.4
43025-0600 Male Connectors	4	0.43	1.72
43045-0620 Female Connectors	4	3.54	14.16
8 GB Class 10 microSD Cards	3	8.99	26.97
825 Jumper Wires	1	3.95	3.95
ADM3053 CAN Interface	2	11.82	23.64
ADM3054 CAN Interface	2	6.23	12.46
BK-916-CT-ND Battery Holder	1	0.50	0.5
BNO055 Orientation Sensor	2	34.95	69.9
Custom Raspberry Pi Hat	3	5.00	15
Custom STM Hat	3	18.00	54
DS3231 Real-Time Clock	1	8.50	8.5
MCP2515 SPI Interface CAN Controller	2	2.09	4.18
Raspberry Pi 3	3	34.99	104.97
STM32 Nucleo-144	2	10.33	20.66
TLP172GFCT-ND MOSFET	4	1.77	7.08
TSL2561 Luminosity Sensor	2	5.95	11.9
		Total Cost	**390.79**