Washington University in St. Louis Formula Racing Data Acquisition and Telemetry System

Andy Wang
Washington University in St. Louis
St. Louis, Missouri, United States

Chris Donnell
Washington University in St. Louis
St. Louis, Missouri, United States

Shane Warga
Washington University in St. Louis
St. Louis, Missouri, United States

Figure 1. CAD of WUFR20 Full Car Model

Abstract

The Data Acquisition system is an essential subsystem of WashU Formula Racing. We gather, log, and distribute data around the car, allowing every other system to focus on sound, data-driven designs. To do this, we utilize custom boards for sensing, logging data, sending telemetry, and providing driver feedback via dashboard. This gives us full control over our data flow and allows us to easily adapt to changing demands from every system year-to-year.

This paper is documenting our attempt at creating a live telemetry system on the car (as well as efforts to update and refine the data acquisition system as a whole). A telemetry system hasn't been implemented in many years on the team, and in this respect we are essentially starting from scratch.

1 Introduction

Formula SAE (FSAE) is an international engineering design competition where students are asked to "conceive, design, fabricate, develop, and compete with small, formula-style vehicles". Every year the WashU Formula Racing team (WUFR) attends a competition in Michigan where our car is tested against cars from other universities across the world. Teams are highly interdisciplinary, with members specializing in everything from fabricating custom engine parts to programming custom data collection and control systems. An emphasis is placed on good engineering practices and rigorously tested designs. Both sound design and actual performance are tested in equal parts, so even the best performing car is never going to win if the team's engineers can't explain their design process and provide justification for the choices they've made.

The data acquisition (DAQ) system allows our team to gather data and perform analysis on the car's various other systems. Over 20 sensors generate data around the car, and it's the DAQ system's job to capture all that data and relay it to the other systems. They can then use the data gathered during testing to validate their designs and come up with improved designs the following year. The data gathered during the competition itself is used to evaluate our final performance versus other university teams. Without a DAQ system, our team would have absolutely no way to validate our designs, which as mentioned above is a critical part of the FSAE competition.

All the data generated by the various sensors is gathered by our custom sensor boards. The sensor boards bundle the data into CAN frames and transmit them over CAN, where the data logging board receives and records it to an

SD card, which can be read after each testing run. Data is also presented to the driver through a physical dashboard, and a number of open- and closed-loop control systems exist on the car (including engine control, wing kit activation, and paddle shifting). Because the DAQ system has access to all the data on the car, multiple onboard controllers, and the requisite CS experience, the system is suited for expansion in any number of directions.

The project we're documenting in this paper would take the current DAQ infrastructure and integrate a wireless broadband telemetry system and an AWS supported dashboard website to provide a system for live feedback during testing and competition runs. This would allow us to detect faults quicker, collect data more reliably, and provide feedback to drivers and designers on the order of seconds rather than hours.

2 Requirements

To make this a robust and reliable system for the team we've laid out a set of minimum requirements:

1. Sensors must be delivering high fidelity data consistently, at an appropriate sample rate (will vary between sensors).
2. Sensor data needs to be logged and available to the team at a later date.
3. Sensor data needs to be transmitted consistently to the cloud (though this can be at a much lower data rate than for logging).
4. Lastly, usability and maintainability are essential to the long term needs of the team, to ensure the system can be passed on to future members.

The last point is of particular importance to us, as all of the authors of this paper are graduating from the team this year and will need to pass on the systems we have developed to new, less experienced members.

In order to meet these requirements, a number of new/revised systems needed to be developed. A redesign of our sensor boards was necessary to make changes to the system and simplify it in the process, and changing our data logging board from another microcontroller to a Raspberry Pi 4 was necessary in order to allow for telemetry to be added. An entirely new web interface also needed to be developed.

At each step of the process, we must be revisiting previous design decisions and ensuring every part of our system aligns with the greater design goals of the car. The project will continue into the spring semester, with designs being incorporated into testing and potentially being brought to competition in May.

2.1 Sensor Boards

Our previous sensor board designs were based around the ATSAMC21 chip, which utilizes onboard ADCs for sensing and an onboard CAN protocol controller to send and receive data from across the car.

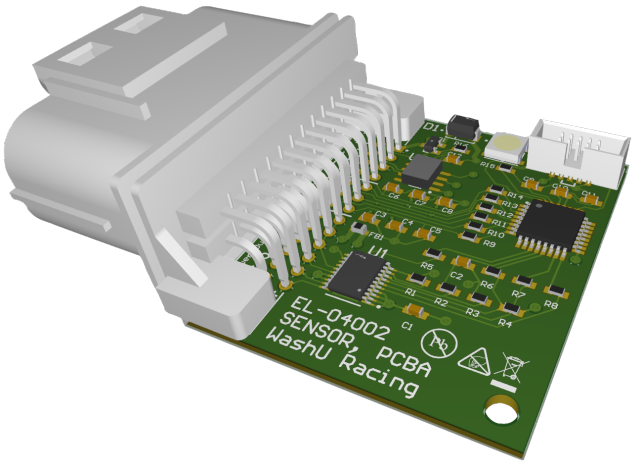

Figure 2. CAD of previous sensor board design

This design is functional but has some major limitations. Limited clock speed on the ATSAMC21 boards and only 6 ADC channels means we have to deploy many sensor boards to achieve full coverage (as we've added more and more sensors to the car, this has become more of an issue - we have well over 20 sensors requiring 5 sensor boards at the moment). In addition, multiple years of adding features has led to an increasingly complex code base which has become difficult to effectively iterate on.

For our redesign, we want to stick to these core requirements:

- sense values from sensors
  - most are 5V
  - need adequate resolution and sampling rate
  - cover all 20+ sensors on the car
- control actuators
  - digital outputs for things like the shifter
  - PWM for servo control
- transmit collected data to be logged
  - medium distance transmission, up to 2m, past very electrically noisy environment
  - we have used a CAN bus in the past for this and it has worked nicely

Beyond handling those things, the biggest priority is keeping the system simple, easy to use, and easy to build on. Ideally this would mean simplifying the code base and reducing the number of sensor boards on the car. We only physically need two sensor boards, one in the front and one in the rear of the car, to achieve full coverage, so eliminating additional board would be nice.

2.2 Data Acquisition and Telemetry Board

In general, for our system to serve its purpose to the fullest extent there must be a way to record data both locally and send data over wireless. Recording data locally can be done with extremely low latency and thus is best for high frequency sensors and, later on, proper data analysis. On the other hand, it would be useful to get live feedback from the system during testing. To serve these two functions our data acquisition and telemetry board must be able to perform these distinct tasks asynchronously. This requirement is caused by the discrepancy in throughput of logging and wireless messaging. CAN frames will be recorded as they are received while the most recent messages are sent over LTE. Doing this asynchronously will limit bottlenecks due the potentially high latency of MQTT protocols while also ensuring that critical data is sent with priority due to implicit priority messaging system that is part of the CAN network. Furthermore, the data logging and telemetry board will encounter a high volume of data so processing times must be minimal or the processor must be fast, in the best case both.

To be more specific, the data acquisition and telemetry board must be able to interface with two distinct CAN FD networks to both send and receive messages up to 8 mbit per second. We need two network connections to separate engine critical data from the Performance Electronics engine control unit (ECU) and the non engine sensor network. When these separate networks need to talk to each other, the data logging board will also need to pass these messages back and forth. Next, on a received message, data is sent to be recorded in a format standard to CAN logging practices (Database CAN) and in accordance to our physical sensor network. And, secondly, take that CAN message and construct a minimal MQTT message that can be sent to a connected UDP server, in our case AWS IoT.

2.3 Front-end

Based on the properties of the sensor data we collected and the expectations we set before designing the project, we defined the requirements on the front-end interface as follows:

- Since the sensors are collecting thousands of data per second, the dashboard should be able to handle real-time, frequently updating data values without any visible lag.
- The dashboard is also required to have emphasis on certain data fields by enlarging the size of those components as the driver may value some values over other less important ones.

2.4 Cloud Computing

To support the real-time display of sensor data, the data transmission and analysis process is critical to the overall performance of the system. To satisfy the project's goal and allow processing of the large amount of data, we set the following requirements on the cloud computing part of the system:

- The data transmission system should be able to handle data with high volume
- The data analysis process should be efficient and lightweight for each piece of data to allow the low-latency visualization
- The host of the system should be reliable and easy to maintain
- The system should be capable of handling volatile changes in the data input (e.g. adding a new sensor or taking out a sensor)

3 Design and Implementation

3.1 Sensor Boards

In order to fulfill the requirements given in section 2.1, we knew we had to select a microcontroller because we want the real-time guarantees and low level hardware access that microcontrollers have.

We hoped to find a microcontroller with many analog inputs for the numerous sensors we have, but we weren't particularly happy with our options, with most boards going up to only 5 or 6, occasionally even 8 analog inputs. In order to cover the whole car with 8 analog inputs we'd need three or four sensor boards, when ideally we'd keep the total down to only two (one board in the front and one in the back of the car). We discovered that higher channel ADCs could be bought and added on to a microcontroller and decided that would be the best option for us. We elected to add on the ADS7953 ADC which can provide 16 analog channels at up to 12-bit resolution, with SPI serial communication is very fast.

We chose to build our sensor boards on the Raspberry Pi Pico, a low-cost, high-performance microcontroller device with flexible digital interfaces. We program them primarily in Micropython [1], which allows for rapid prototyping and quick changes in software. In addition to fulfilling the requirements of the sensor board, it also keeps our code simple and clean, and allows for useful programming techniques like object-oriented programming and Python-like threading. We add on the CANPico sock to enable sending data via CAN messages[2], which will be received by the data logger and telemetry board for saving.

The Raspberry Pi Pico also conforms to our other external requirements. The Pico has ample IO, including 16 PWM channels for controlling servos. The Pico is also very well supported, with documentation being maintained by Raspberry Pi, so we expect for it to be a very stable platform for new members to learn and build.

3.2 Data Acquisition and Telemetry Board

3.2.1 Processing Unit. Due to availability, simplicity, and overall performance we have chosen to use a Raspberry Pi

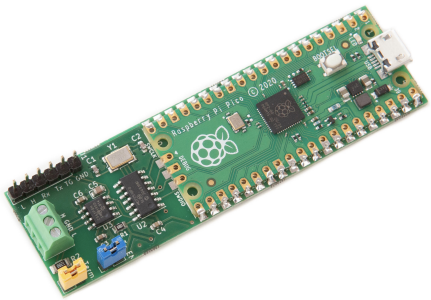

Figure 3. CAD of CANPico Board and Sock

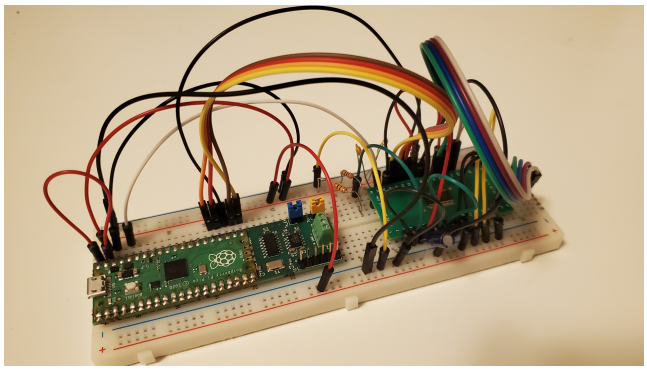

Figure 4. Sensor Board Design Laid Out on a Breadboard

4 as our data acquisition and telemetry board. The Raspberry Pi 4 satisfies many of our requirements without much need for optimization which leaves further design room in our system. With a Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz, 4GB of LPDDR4-3200 SDRAM memory constraints and processing times for our desired data rate are achievable without lower level language optimization. In fact we can perform all our necessary task simply using Python and some niche Linux kernel features. The Raspberry Pi 4 is multithreading capable and Linux itself has many features that support concurrent tasks. Additionally, the expansive I.O. compatibility of the Raspberry Pi 4 let us use a set of third party expansions that resolve both CAN and LTE interfacing on a hardware level and to some extent a software level as well. It is important to note that our Raspberry Pi 4 uses Raspian-lite 32-bit os which is a debian distribution of Linux from Raspberry Pi that has a greater level of hardware support than alternatives and has many extraneous feature stripped out such as the GNOME graphical shell freeing up additional processing.

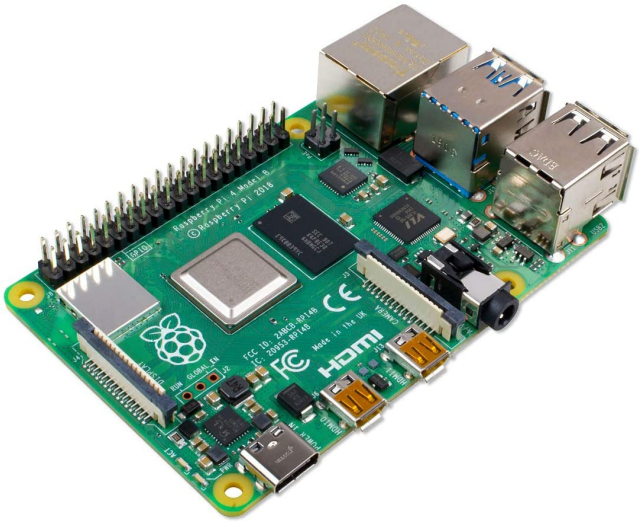

Figure 5. Raspberry Pi 4

3.2.2 CAN. Considering the Raspberry Pi 4 can't interface directly with a CAN Bus, we need an expansion board which can support two distinct CAN FD networks.

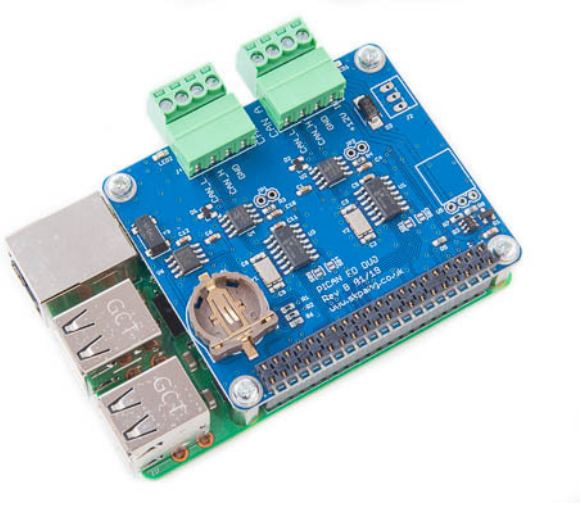

Figure 6. CAN FD Duo Hat from Copperhill Technologies

This Raspberry Pi 4 hat satisfies all of our requirements and has a few added benefits. First, there is support for a real-time clock which can be used to override the network base Linux clock allowing for more accurate data logging and telemetry by extension. Secondly, the CAN FD DUO has virtual CAN support through socket-can letting us use Linux networking features.

3.2.3 LTE. With the use of the Raspberry Pi 4 and the built in networking features of Raspian-lite we can use the SixFab LTE kit to create a WWAN connection on the Raspberry Pi 4 which treats LTE like wireless LAN. This simplifies the

connection process and lets us use network priority features so if there is a stable wireless LAN or Ethernet connection the system will prioritize those. The consequences of that are that we are always using the fastest and most stable connection and that LTE will be online in the case where there are no other networks, given that there is a strong enough signal. This also turns out to be a good cost saving measure as LTE can be costly when sending large amounts of data.

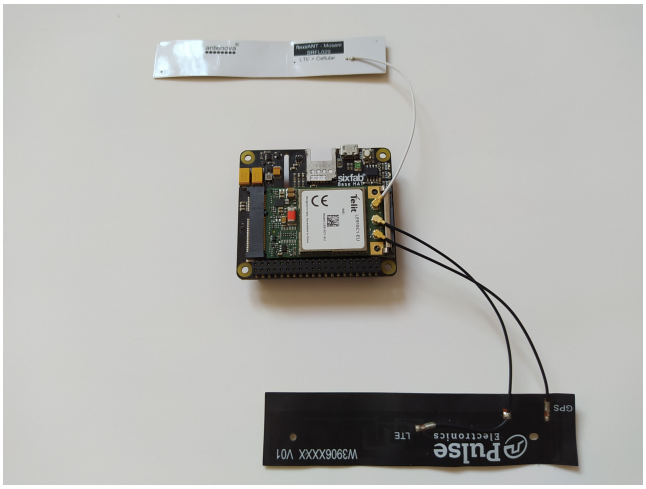

Figure 7. SixFab LTE Hat for Raspberry Pi 4

3.2.4 Drivers. Almost all of the functionality of this system is driven by a single python script, a crontab boot script that establishes virtual connections for physical CAN to socket-can, and some additional third party drivers from the hats discussed previously. With that, the python script uses a notifier-listener reactor pattern to asynchronously schedule tasks to be performed. This allows the two critical sections to perform at their respective highest rates. In practice, we can recorded thousands of CAN messages per minutes locally and send hundreds over LTE.

3.3 Front-end

For the design of the front-end interface, we utilized InVision Studio for sketching out the layout of the UI. The design was initially inspired by Formula 1 race car dashboards on the steering wheels as it is a proven way of efficiently displaying engine data in an interactive way. In comparison to a traditional gauge cluster on a production car, this digitized way of displaying data provides more clarity and is more friendly to the driver's and team members' dynamic vision as many sensor values are changing much more frequently on our race car than the ones on a traditional production car. By creating the sketch of the dashboard first, our programmer is able to visualize the goal of the front-end goal with specific requirements on dimensions which made the development process smoother. By contrast, the RPM dial at the top visualizes the revolutions per minute value of the engine. The specific value of this piece of data is not critical, but the driver cares about when the engine reaches the revolution limit, which is indicated by the red area of the bar, at which the driver need to shift gears.

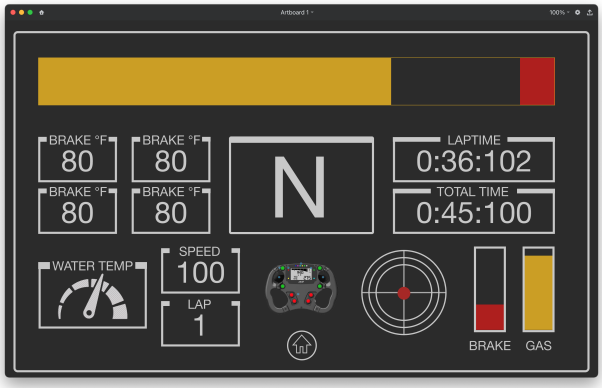

Figure 8. Design Sketch of the Dashboard Using InVision Studio.

As illustrated above, the components are given different sizes based on their importance and visual priorities. The display format is also differentiated to fit the nature of different data values. For example, the brake temperatures are displayed as figures since it is a sensitive value and showing the specific numbers is the most straight-forward way to display it.

3.4 Cloud Computing

The design of the cloud computing part of the system is mostly based on the Amazon Web Services. AWS contains a suite of services that perfectly meets the demands of our project. What we need for the system includes a wireless data publishing and receiving service, a server-less function that can decode the raw data and publish it back to the stream, and a cloud server for hosting the actual website. With AWS, we are able to acquire the resources we need at a low cost. Compared to what we had in previous implementations, where we used to have a local server specifically for hosting the API and the database that stores the data logs, using AWS allows us to simplify the structure of the system while maintaining the same functionalities. The components used in the system and the flow of data is shown below:

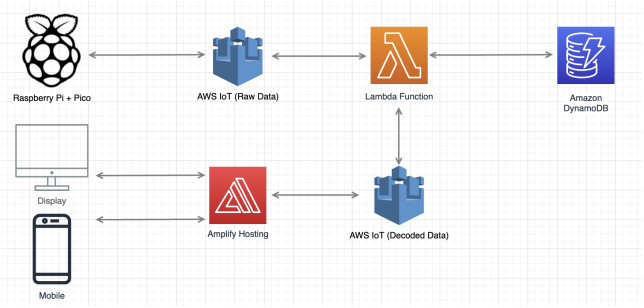

Figure 9. Structure Diagram of the Cloud Computing Components Involved in the System.

Figure 10. Screenshot of the Dashboard Website.

To explain the implementation of the cloud computing system, we would first summarize the hardware components as Raspberry Pi + Pico and assume that the hardware system is sending raw, binary data out to the cloud [4]. Starting from here, as the raw data is published to AWS IoT [5], AWS IoT uses a protocol called MQTT, a data publishing and subscribing protocol that is widely used in the field of Internet of Things for transmitting data, to subscribe to the topic that the data is published to and record that data in real-time. The data analysis is accomplished by triggering an AWS Lambda Function [3] from AWS IoT such that for each message received, an AWS Lambda Function is called with the message as one of the input parameters. The AWS Lambda Function is a Python script supported with a local decoding file that includes the linearization functions for each individual types of sensors. The Python script would take in the message as a JSON object and determine which sensor the message is describing based on the name of the sensor read from a field of the JSON object. Then, it would find the corresponding linearization function in the local decoding file and decode the binary raw data into a floating point number. This decoded data is then published to another AWS IoT topic that is subscribed by the website client [6]. Finally, the client displays the data. This process would usually operate at a frequency of 10 times per second. A screenshot of the final product is attached below:

4 Lessons Learned

4.1 Design Phase

During the development process, we decided to change the hardware setup of the system to a completely different structure. This resulted in a decent amount of time allocated to switching to the new system. In fact, with a more thorough planning process before the implementation process started, this change during the development process could be avoided.

4.2 Implementation Phase

Because of the delays during the design phase, we also didn't get to do as much testing of the system as we would have liked. While we did get the system fully operational, it took a big push at the end and there were a few smaller issues that we didn't get to fully resolve by the end of the project.

In addition, due to external factors we did not get to have the car fully built to test the system by the end of the project. We were depending on testing in a real-world situation, so there was a lack of feedback to help refine the result. In future projects, it would be reasonable to have a simulation plan as a back-up validation process to make sure the system would work in real-world situations so that iterations on prototypes can be achieved.

5 Conclusion and Future Work

In this paper, we've detailed how we went about implementing a wireless telemetry system and integrating it with the sensor network on our FSAE car. This will greatly benefit the WashU Formula Racing team during testing, since now the ground crew will have a much better idea of how the car is performing which lets them guide the driver better. In addition, each system will be able to get near instant feedback on their systems and evaluate if everything is performing as expected or if there might be a fault on the car.

Since the car is a much longer term project than a single semester class, it is assumed that we have only completed a first iteration on the telemetry system. In the spring we intend to further test and improve essentially every system, but this project has laid out an amazing groundwork and is an easy jumping off point for people to build upon. Some specific areas we expect to revisit are <lessons learned ideas>.

Beyond just the direct work for the racing team, this project and the research that went into it could potentially have many benefits. This kind of network setup could readily be applied to modern cars, either retrofitting data systems on current/older cars or helping in designing newer cars with improved data systems. The skills to make this kind of sensor network are also relevant in many other domains such as IoT systems and robotics. Using the skills and techniques we developed in this project, each member is equipped to work on many other types of systems.

6 Source Code

All source code for this paper can be found here. https://github.com/WURacing/DAQ-Project Note that this is part of the WUFR organization Github and this repository will be made private once the grading period has ended.

References

[1] Paul Sokolovsky Damien P. George and contributors. 2021. *MicroPython documentation.* Retrieved Dec 14, 2021 from http://docs.micropython.org/en/latest/

[2] CANIS AUTOMOTIVE LABS. 2021. *The CANPico Board.* Retrieved Dec 6, 2021 from https://kentindell.github.io/canpico

[3] Amazon Web Services. 2021. *Building Lambda functions with Python.* https://docs.aws.amazon.com/lambda/latest/dg/lambda-python.html

[4] Amazon Web Services. 2021. *Connect a Raspberry Pi or another device.* Retrieved Dec 03, 2021 from https://docs.aws.amazon.com/iot/latest/developerguide/connecting-to-existing-device.html

[5] Amazon Web Services. 2021. *What is AWS IoT?* Retrieved Dec 14, 2021 from https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[6] James Thesken. 2020. Building an AWS IoT App with React.js. (April 2020). https://theskenengineering.com/building-a-react-js-app-with-aws-iot/