NEXT GENERATION IMPLEMENTATION OF AN IMPROVED TELEMETRY SYSTEM FOR MONITORING AN OFF-ROAD RACECAR

Daniel Fuehrer, Harrison Pearl, Oliver Sjostrom
Haseeb Irfan, James Nguyen
Faculty Advisor:
Dr. Michael Marcellin
Department of Electrical and Computer Engineering
The University of Arizona, Tucson, AZ 85721

ABSTRACT

The University of Arizona Baja Wildcat Racing Team designs and fabricates an off-road vehicle for an international competition. A telemetry system serves as an important backbone for design optimization and problem detection. The system consists of an array of sensors, microcontrollers to read them, and the core module. The core module is the heart of the system and it compiles, transmits, and stores the data locally. A custom screen displays speed data to the driver. The software saw a total redesign. The primary changes were the implementation of a data management class to improve I2C communications, and the generation of data structures using python scripts to reduce the risk of data loss. The software will be the primary focus, with detailed design justification, and hardware updates will also be discussed.

KEY WORDS: Mobile Interface for Visualization and Control of Off-grid Telemetry Systems, Baja SAE, I2C, Microcontrollers

INTRODUCTION

Baja is a collegiate design series overseen by the Society of Automotive Engineers (SAE). The goal of Baja Racing is to create a buggy-style, single-seater, off-road vehicle which can drive in all weather conditions and terrains. As part of the Baja team, the goal of the electrical sub-team is to develop a telemetry system that has the capability to transmit data from the car to a remote computer over 900 MHz radio using the core module. To this end, the car features a variety of sensors including GPS, IMUs, Hall effect sensors, and brake pressure sensors. In the past academic year, the team has worked to add a battery board, which provides battery level

monitoring, protection circuitry including reverse polarity, overvoltage, and overcurrent protection to the system while also regulating the hot-swappable 12V drill batteries to power the sensors and the microcontrollers. By using two drill batteries in parallel, the system has an easy and lightweight solution to replacing batteries and not losing power while doing so. Additionally, the speed is now displayed to the driver on a screen consisting of 16 x 9 individually addressable pixels. The software behind the system saw an overhaul, with a new I2C data management class to streamline the storage and organization of data carried by the communication bus. Finally, python scripts were developed to generate code that previously had to be written manually, thus making the process more efficient.

MOTIVATION

Given its status as a university organization, the electrical team on the Wildcat Racing team holds the primary objective of providing a learning opportunity for its members. In developing and fabricating a telemetry system, students gain real-world experience with writing code, constructing a wiring harness, and utilizing sensors. As they face obstacles, students learn to solve technical problems as they arise, all as part of the larger system. Additionally, as a part of the SAE Baja organization, the team has the objective of developing a working telemetry system which can transmit useful and relevant information from the vehicle to a remote computer, including speed, GPS position, and brake line pressure. Such actionable information can offer vital feedback to drivers in a racing environment and hence, a superior racing experience.

HARDWARE

The telemetry system comprises sensors and microcontrollers that measure, interpret, and transmit valuable information about the car's operation. The system has four different sensors. Two hall effect sensors were placed on each of the front brake calipers, close enough to the wheels to measure wheel RPM data. Four IMUs are placed on each suspension link. These are a source of acceleration data which is combined with the RPM data from the hall effect sensors to calculate speed. Two brake pressure sensors, in the cockpit underneath the floor panel, monitor the front and rear brake pressure. Finally, the core module is responsible for collecting data throughout the car and transmitting data back to the pit using a Xbee transceiver. The pit uses a custom program MIVCOTS, Mobile Interface for Visualization and Control of Off-grid Telemetry Systems, to track the car's position in real-time. The core module also interfaces with a custom pixel array screen to display the speed from the GPS sensor to the driver.

Figure 1. LED front (left) and back (right) with individually addressable pixels. A microcontroller is attached to the back.

The microcontrollers handle packaging the data and sending it to the core module. Many sensors cannot natively communicate over I2C so microcontrollers do the computation and communicate that data to the core module over I2C.

This year, we added a new battery board to the system. The battery board in Figure 2 is designed to monitor the current draw for each component, regulate all the voltages, provide protections like surge, overvoltage, reverse voltage, and monitor battery levels. Most of these features weren't functional but the board was still able to distribute power. Two rechargeable lithium ion drill batteries provided power to the battery board. Drill batteries are easily swappable, lighter, and safer than other options. We also did a respin of the LED board this year, which includes fixing the oscillator circuitry and making a bypass for the onboard microcontroller. We added additional capacitors from the previous year, yet the board still produced sub-optimal results. The screen only situationally displayed data. When not attached to the vehicle, the screen functioned properly. When attached to the vehicle, and pushed with the engine off, the screen continued to function. However, when the engine was on, the screen would flicker random colors. This could have resulted from poor construction and the vibration from the engine was dislodging the pixel or it could possibly be electrical interference from the spark plug since the wiring harness and other components are unshielded. Further testing will show what the problem is. Once the problem is identified, a proper screen shall be constructed with dampeners or the wiring harness shall be wrapped in shielding.

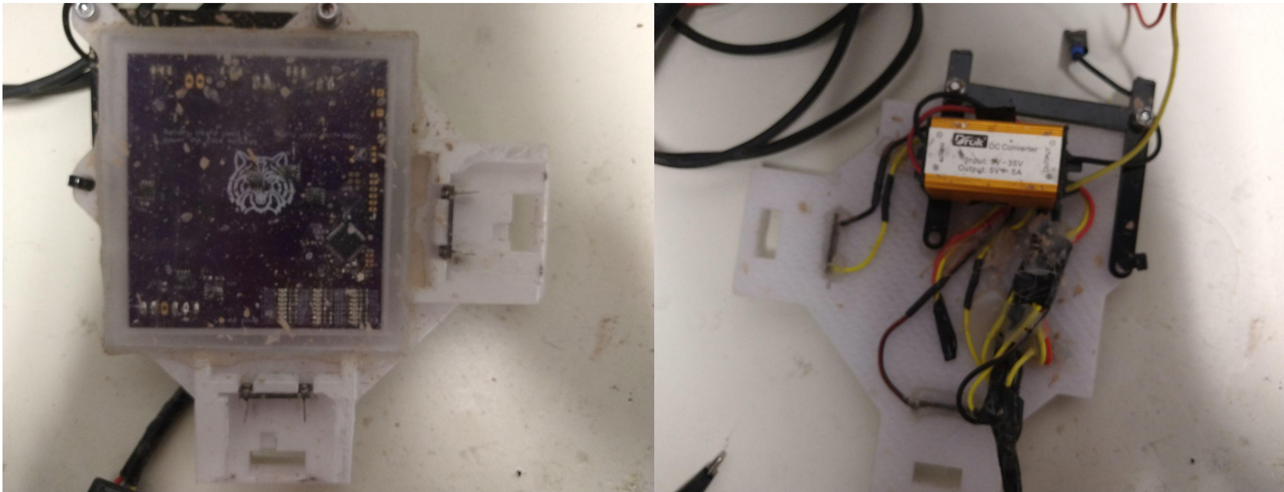

Figure 2. Battery board front (left) and back (right).

Other components of the system also experienced some failures. The USB hub wasn't fully functional, meaning each microcontroller individually needed code flashed onto it. A potential cause is a manufacturing or transportation issue with ordered components. All 5 of the new USB boards assembled were nonfunctional and the PCB design for the USB boards were unchanged from the successful ones in the past. There was also some difficulty with receiving data in the pit. The Xbees had to be reset to the default channel, risking interference from any other teams using the default channel. Finally, there was some issue with the hall effect sensors impacting the wheels of the car. This was likely an installation mistake, however, moving them to the top of the brake caliper would help reduce future risk of the same issue. Although the hardware saw some minor improvements, mostly, the primary focus was software.

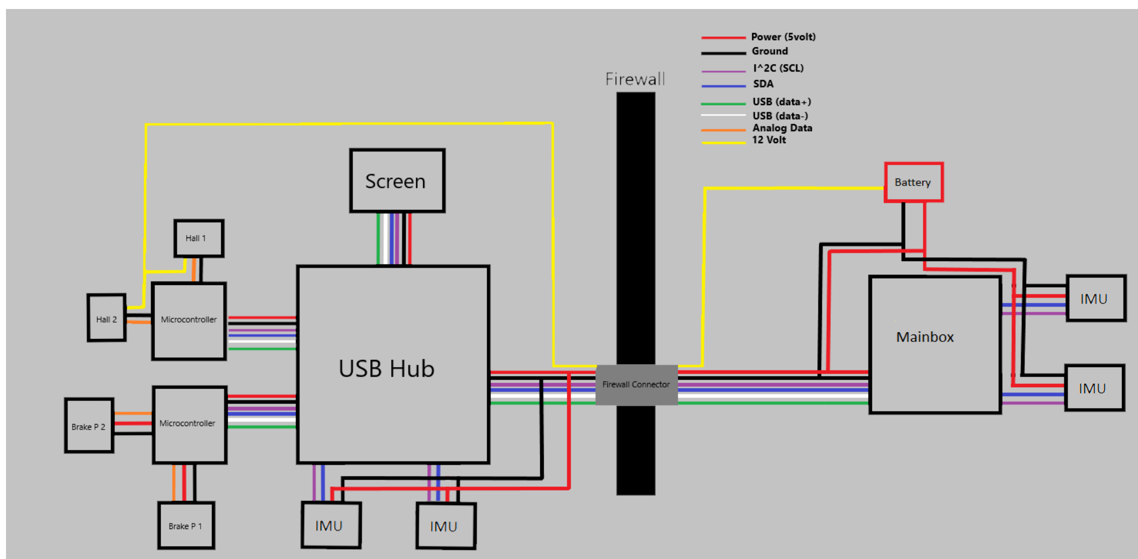

Figure 3. Basic hardware diagram

SOFTWARE

The motivation for the redesign of our vehicle's software was to achieve more efficient transmission of data over the I2C communication protocol and reduce the possibility of error in the generation of system critical code. We are able to achieve these goals by automating tedious programming processes and adhering to agreed upon style guidelines.

At the core of our vehicles software is our custom libraries which are generated for each individual sensor type on the vehicle. By writing our own library for each sensor we are able to precisely control the operation of the microcontrollers on our vehicles and the communication and storage of data between the microcontrollers and the main box. Additionally, by generating these libraries we are able to standardize our sensor structs which package our data in an object to be transmitted to the main box over the I2C protocol.

To manage each sensor struct we programmed management classes for the mirontrollers. The management class is also generated automatically. It contains methods that handle the transmission and receiving of the data held in the sensor structs. These management classes make use of the Arduino “Wire” I2C library to send the data obtained from the sensors connected to the microcontroller to the main board. We did this as the “Wire” library is simple and standardized across all of the microcontrollers and architectures we used. This microcontroller unit class (MCU class) does have to make up for limitations in place within this library, such as the fixed buffer size limit of a maximum of 32 bytes in the AVR library. Overall, the management classes set a standard interface for handling the transmission of data between microcontrollers and ensures the master and slave agree on the data being sent by using the same generated C++ files. One important feature of these classes is that they allow the main board to specify which data it wants to receive by sending a bit array first. After reading the bit array, the other microcontroller sends back a bit array of flags for which data has been updated since the last request and follows with the updated and requested data. This way, the main board can focus on specific sensors to prioritize and have a consistent way to get their data even though many different sensors are attached to a given microcontroller.

At the center of the system is the main processor. This microcontroller collects the data from all the sensors and microcontrollers, compiles it into a buffer, saves it to its SD card, and sends the data over radio to the pit. The operation of the main processor is about as simple as the other microcontrollers. It stores the data for each sensor in the sensor structs and uses the management classes to receive data over I2C from the other microcontrollers. Beyond storing and transmitting the data, it also has additional functionality. First it can choose how often to receive data from each sensor with a simple state machine and timers. This state machine could be made to dynamically update its frequencies on receiving a command over radio to be able to prioritize certain data in real time from the pit. The other additional action it does is relaying data to the

custom screen in front of the driver, in this case speed from the GPS. Alongside the main board is another microcontroller with an attached SD card. This microcontroller reads the GPS data, logs the raw NMEA strings to the SD card in case of data loss, and relays the data to the main processor.

In order to produce well written programs to be executed on the microcontroller we compiled a set style of programming which was created on the basis of being pragmatic and simple. Starting with variable types, floating point numbers are to be avoided. This is a limitation of the microcontrollers, as they do not have Floating Point Units; in order to perform mathematical floating point operations, the microcontroller must make use of a floating point library. By only using integers we are able to perform mathematical operations with the same precision as floating points numbers by storing a scale factor. Using integers in this fashion is more efficient than using floating points, and is the primary reason for their use.

Another edict of our style guide is the use of static allocation for all memory to avoid dynamic allocation. This is done as we know exactly how much memory is required for our variables since the data storage requirements are a direct consequence of the hardware attached. Reinforcing this decision is the fact that dynamic allocations are not only less efficient from a processing power perspective but would allow for potential memory leaks. By using static memory allocation we avoid the possibility of over allocating memory and having the stack and heap collide.

The final criteria for our style guide relates to documenting our code. When writing a function or script, all functions are to be well commented with inputs, outputs, and descriptions. By commenting our code base, members of the team who did not write the code are able to debug the code they may be running in a test setting. As well, having well commented code is in general a good practice and reinforces positive programming habits.

We developed python scripts for automating the generation of two of the most crucial but tedious pieces of code to write. Both of these are structures that have very much in common with the other structures of their type, so they can be scripted reliably. The first structure to be generated by a script was the MCU classes that manage the data and I2C transmission for the subservient microcontrollers. The second file to be generated is the sensor structs that hold the data for each sensor on both the main board and the slave microcontrollers. Both of these are prone to errors that can lose data from small changes. For instance, the sensor structs take advantage of the bit alignment of the microprocessors so they can operate identically on the different architectures used in the system. This means that something as simple as changing the size or order of a variable can lead to loss of data on transfer. The MCU classes similarly can fail to pack or unpack the data buffers correctly if a change is not made in all the required places. Both scripts work off the same concept: at the core they parse a JSON file to know how to construct the structure correctly based on their template and then write it to a new file. Both scripts also have

a GUI built for them to generate the JSON file and use the first component to generate the C++ files. Overall, these scripts bring consistency to the generated code, allow for easy changes and generation of new files for new hardware, and can allow the code to be written more explicitly for its use case rather than relying on a lot of abstraction to reduce the amount of new code to write.

CONCLUSION

Having seen the telemetry system utilized at the competition, there were a number of lessons that we intend to explore in the future. First, more testing is required on the individual hardware components before they are assembled with the rest of the car. This will allow the team to isolate potential failures to the connections rather than the parts, clarifying the troubleshooting process. Next, the failure of the screen at high speeds despite normal performance when created reveals that more testing is required under realistic conditions for the car so that possible flaws may be brought to light ahead of time. Finally, we found that the code became easier to write as well as teach to new members following a reduction in polymorphism. The team intends to continue to phase it out wherever possible in the future so as to continue this progress.

ACKNOWLEDGMENTS

We’d like to thank Dr. Michael Marcellin for his guidance and help making this project possible.