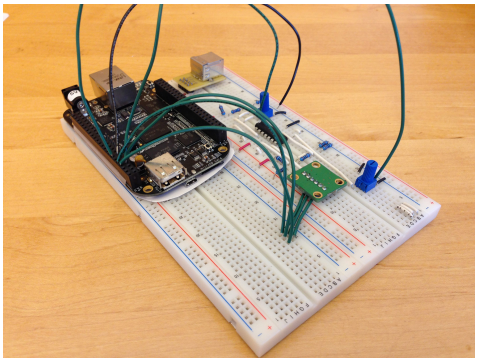
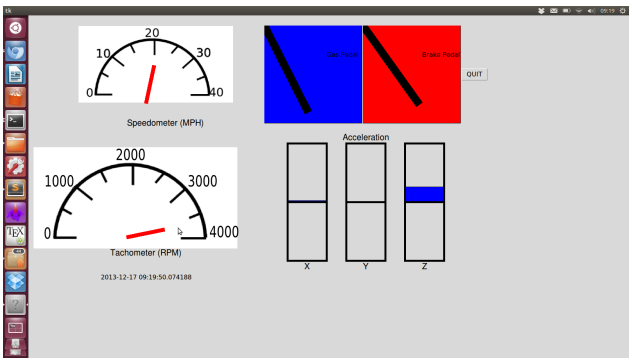
Designing Sensor Data Processing Software for Olin's Mini-Baja Team

Final Project Report





Justin Poh & Zoe Fiddler

Software Design, Fall 2013

PREFACE

This report discusses code which may be found at the following repository:

https://github.com/JPWS2013/SoftDes_Project

In order to run this code, you will need a beaglebone black with the Adafruit_BBIO library and pyserial installed.

The following files will need to be copied to the beaglebone black:

1. bb_datamethods.py
2. bb_model.py
3. controllerclassdefs.py
4. trace_playback.py
5. mainprogramme.py

To run the code:

1. Plug in a suitable USB cable between the beaglebone black and a laptop that contains the module Dash.py
2. Open a terminal and ssh into the beaglebone black
3. Open another terminal and Run Dash.py on the laptop
4. In the terminal that is commanding the beaglebone black through ssh, run mainprogramme.py

1. Introduction

The aim of this project was to create the framework and software to read and process data from sensors on a Mini-Baja car and visualize those results in a meaningful way for pit crew members or the driver. This project was motivated by the needs of Phoenix Racing, Olin's Mini-Baja team. With the software, the team can be informed of racing conditions or performance problems on the car.

Because the team had already opted to use a beaglebone black to read sensor data, the software written for this project was tailored to be used on a BeagleBone Black. A dashboard was also written to visualize the data in order to demonstrate the capability of the software. Although the team intends to use cellular 4G to transmit and store the data to a database, this project mimics that process by running a module on a laptop and transferring data from the beaglebone to the laptop using the serial port.

This report begins with the original project proposal, including the original aim, and steps through each iteration of the design. The final sections discuss the final design refinements made as well as an analysis and reflection of the outcome.

2. The First Iteration: Original Project Proposal

Submitted on 25th October 2013

2.1 - The Project

The aim of this project was to create the software to process data read from sensors on the car and produce meaningful results that would benefit Phoenix Racing, Olin's Mini-Baja team. With the software, the team could be informed of racing conditions or performance problems on the car. The software was intended to run on a beaglebone or raspberry pi and we wanted to write a virtual dashboard that would help a user to visualize the data being output from our software.

2.2 - Minimum Deliverable

The minimum deliverable we planned on having was software that was capable of processing the data and presenting it in a python Graphical User Interface (GUI) such as a virtual dashboard

that would allow a user to visualize the data being processed. We also wanted to write an Application Programming Interface (API) in Flask that will allow the Mini-Baja team to use the API we wrote to manipulate the data being transmitted from the car for whatever purpose they saw fit.

2.3 - Maximum Deliverable

The maximum deliverable we planned on having was processing the data, presenting it using a javascript GUI and setting up the API. This would have involved learning some javascript but would have been much more useful to the team, as it could actually be placed into the web-app they would be creating.

2.4 - Foreseeable Problems

We faced several foreseeable problems. Firstly, none of the sensors were actually up on the car yet, so we had to use dummy data. We also had to make guesses as to what the data from each sensor would actually look like since we were unfamiliar with some of the sensors the Mini-Baja team planned on using.

Secondly, this project would ideally use both flask and javascript, neither of which we knew. While learning one of these may have been possible, learning both would have probably been too much work. We prioritized learning flask, but still hoped to find the time to do both.

Finally, we wanted to keep communicating with the baja team to find out what data would be most useful on the dashboard and to make sure they were satisfied with the results. This was difficult since early in the year, they were not sure what the final car would look like, so they did not necessarily know what they needed.

3. Second Iteration: Original Design Proposal

Submitted on 8th November 2013

3.1 - Updated Project Goal

For the updated project goal, we decided the project would have several major components. The first module would be a trace playback module to simulate live data coming from the sensors in cases where the physical sensor was unavailable. The second module would run on the beaglebone and would contain modules and functions that read raw data from the various sensors or from the trace playback module when the physical sensor was not available. This module then processed those readings to produce meaningful data that could be visualized by the module running on a laptop. Finally, the third module would run on a laptop that acted like a server requesting information from the beaglebone, as it would do when the beaglebone runs on the car. This module would consist of a dashboard to visualize the processed data from the beaglebone and logic or threshold checks for critical driving conditions.

3.2 - First Major Design Decision

The first problem we encountered since the original proposal was how to obtain sensor data when a majority of the sensors didn't exist yet. If we wrote everything as one programme that read, processed and output sensor data, we wouldn't be able to test anything because the basis of that programme required being able to read data off of sensors, which we didn't have.

This led us to decide that we needed to write another module that pretended to be a sensor or a group of sensors. However, we then discovered that even if we were able to write a program that pretended to be a sensor, it would still interact with our main programme in a way that would not be comparable to having the physical sensor.

Finally we decided on our current implementation because we realized we needed to restructure the way we were thinking about our implementation. By modularizing our implementation into 3 modules as described above, it would allow us to write each module independently and then insert helper functions to bridge the modules where necessary. This would allow us to continue to make progress on parts of the project without having progress be dependent on the availability of physical sensors or other hardware.

Another reason for the choice of structure was the way in which the software would be transferred to the team at the end of the project. We realized that our original implementation of writing a standard programme all in one script would not be ideal when transferred back to the team because how we made use of our functions would not necessarily be the way the team would want to make use of the functions. Hence, modularizing our code in the way that we did would allow the team to take the modules apart and make changes to them independently without affecting the rest of the software.

3.3 - Development Plan

The first part of this project was to write a Python module that ran on a laptop and read data from a trace playback module, which essentially generated data which mimicked sensor data readouts as much as possible. This allowed us to mimic sensor data without actually owning the beaglebone. The second part was then to hook up a few of the sensors from the car, most likely accelerometer and potentiometer, to the beaglebone, and have a laptop running the python module read the data. We then started on our first iteration of the dashboard to present this data in a useful way. We had not yet decided whether this initial dashboard would be a Python or Javascript GUI. Once we had achieved these steps, we would mainly be adding or mimicking more of the sensors on the car and adding them to the dashboard.

3.4 - Foreseeable Problems

We again had several foreseeable problems in addition to those discussed in section 2.4. Firstly, the car was now broken so we lacked any real data for us to process. This simply meant that our dummy data could potentially look nothing like the data we would be receiving from the car because we did not have a way to figure out what data to expect. Secondly, we still wanted to keep communicating with the Mini-Baja team to find out what data would be most useful on the dashboard and to make sure they were satisfied with the results. This remained difficult since they were still not sure what the final car will look like, so they did not necessarily know what they needed. Finally, we were concerned that writing an API to bridge the server module when reading trace playback data and when reading live data might be tricky.

4. Third Iteration: Original Design Refinement

Submitted on 22nd November 2013

4.1 - Significant Changes

One of the significant changes we made in our project is deciding to implement a Model-View-Controller model for our code architecture. We decided on a Model-View-Controller design because it represented the components of our project well and this is the implementation that exists in our final design of the software. For this iteration, the model was a collection running on a laptop that stored the data read from the sensors. When fully implemented by Olin's Mini-Baja team, the model will likely be a web server. The View was initially a text-view. This was eventually swapped out and is now a dashboard to visualize the data. Finally, the controller is all the sensors that read data and publish them to the model. This information is represented in the following diagram:

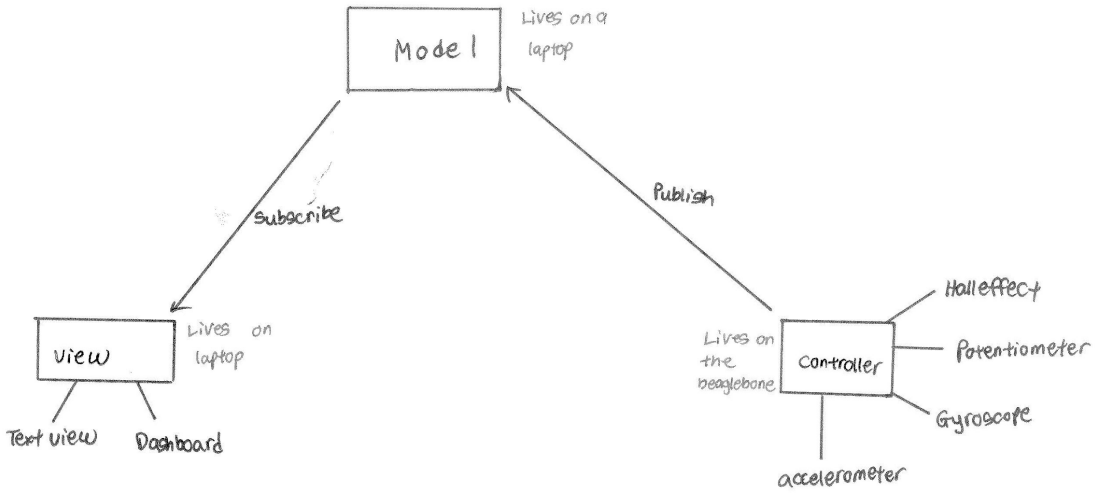


Figure 1: Diagram of Model-View-Controller Model Being Implemented

As can be seen in the diagram, the controller consists of a group of sensors. We have implemented this by creating a class called "Sensor" which inherits from object. We then created classes named after each sensor that all inherit from the parent class Sensor. The model was then essentially a database of sensor data received from the controller. Finally, the view was initially a text-view in the command prompt but eventually became a dashboard which visualized the data received by the model.

Another major change we made was to implement a publisher-subscriber model. We chose to use a publisher-subscriber model because it allowed us to most closely mimic the actual information flow we would like to have between our sensors and our displays. The sensor instances that make up the controller all published to the model and the view would subscribe to the model. This mimics the natural way information would flow if the information were to be transferred manually and so we have chosen this architecture for our project.

4.2 - Abstraction and Language

In our programme, we have chosen to make use of language that mimics the natural way information would be referred to and passed if it were done manually. We also took advantage of domain terminology to make the programme as natural as possible. For instance, we have purposely chosen the parent class to be named “Sensor” because the child classes that inherit from the Sensor class are all types of sensors. Each of the subclasses of Sensor are then the specific types of sensors. This thus mimics the fact that, for example, an accelerometer is a type of sensor. We also store attributes in the instances of the class of sensor. For example, an instance of Potentiometer would contain attributes related to the total resistance of the potentiometer and the maximum sweep angle. This mimics the physical sensor having a data sheet with properties that a user would need in order to make use of the sensor. By storing such information as attributes in an instance, it mimics the idea that there are multiple types of the same sensor and a particular instance has properties that may be different from another instance of the same sensor. As such, we have designed the programme terminology and functionality to mimic the physical products as closely as possible. We are also using domain-based terminology to name class methods that will make it easier to understand what each class method does. Overall, this has made it easier for us to communicate information about our programme because the class names are already derived from domain terminology, just as one would communicate the same idea using entirely natural language.

5. Final Iteration - Final Report

5.1 - Final Design Refinement

As part of our final iteration, we have made several major changes to our project. Firstly, in order to have the beaglebone communicate with the laptop and send data, we had to divide the model into two. One part of the model was run by the beaglebone and contained methods that would transmit data over the serial port to the laptop using pyserial. The second part of the model was run by the laptop and contained methods for storing the data it received over the serial port in a suitable data structure. The data structure we chose will be discussed shortly.The UML class diagrams below represents the new relationships between the classes that we have implemented on the Beaglebone and on the computer.

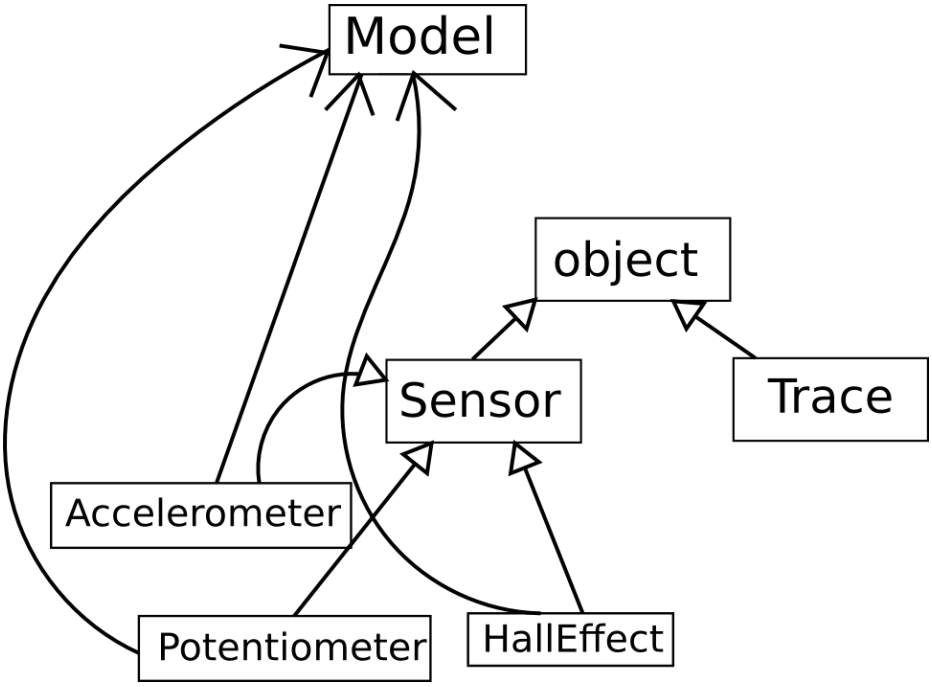


Figure 2: UML Class Diagram showing relationship between classes on the Beaglebone (only IS-A and HAS-A relationships)

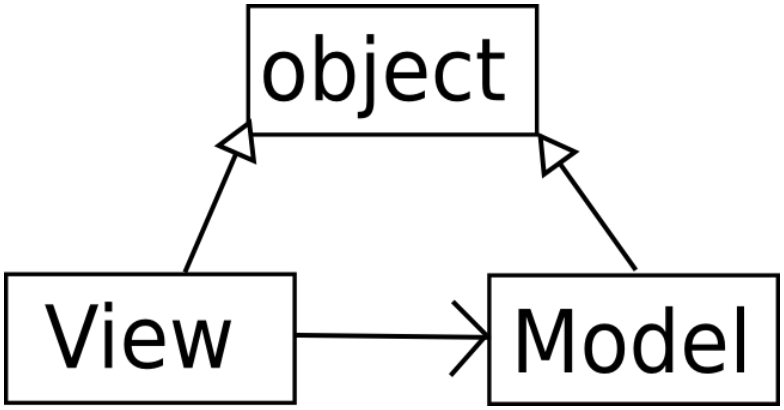


Figure 3: UML class diagram showing the relationship between classes on the computer (only IS-A and HAS-A relationships)

Secondly, we chose to store the sensor data in nested dictionaries. The top level dictionary contains the sensor IDs as keys and the data as values. This data is, in itself, a dictionary where the keys are string versions of datetime objects that represented the date and time at which the data was taken and the values are the actual sensor data, which may be a list or a single value, depending on the sensor. We chose dictionaries because value retrieval is efficient. For this project, efficiency was necessary because of the high frequency of reading and writing of data that our software requires. For these reasons, we chose nested dictionaries for the data structure of our model.

Thirdly, we modified our original decision to use a publisher-subscriber model. Instead of having the model inform the view whenever new data was received, the view would query the model for new data and store and display it. Originally, we had planned to have separate model and view modules that would be imported and run from a common script, similar to the way the controller and model modules are used on the beaglebone. However, in order to write a GUI in Tkinter, having 2 separate modules would not work because once Tkinter ran its mainloop, no other function outside Tkinter could be called. In order to solve this problem, we modified our code such that it now only has a view that contains the model within it. The view then queries the beaglebone model for new data and then stores and displays this updated data. As such, it is no longer a true publisher-subscriber model.

Fourthly, we discovered we had to insert pauses between successive transmissions from the beaglebone in order to be compatible with the rate at which we had set the Tkinter dashboard to query for data.

Finally, we also added the use of the python pickle module in order to output the sensor data to a text file once the GUI was quit by the user. This ensured that the data that had been collected would still be available even after the programme had ended.

5.2 - Analysis of the Outcome

We did not quite achieve the minimum deliverable that we set for ourselves, although we did achieve most of it. The one part of the minimum deliverable that we set for ourselves and did not achieve was learning flask to write the API for the mini-baja team. We think this was mainly the result of the difficulties we experienced at the beginning of the project trying to define the scope of our project and precisely what we wanted to do. In the end, our project also relied heavily on the Model-View-Controller architecture, which we had no experience in using. Thus, it took significantly longer than we anticipated to figure out what we wanted to implement and how to implement that architecture in our software.

Another thing that we did not anticipate spending as much time as we did on was setting up and interfacing with the beaglebone and building a test circuit with the sensors. Because neither of us had ever worked with a beaglebone before, we spent a lot of time learning how to use the beaglebone and how to access its General Purpose Input/Output (GPIO) pins. We also spent a significant amount of time learning how to transmit data over the serial port. We had both done this previously using other platforms but not the beaglebone and it turned out to be more difficult than we had anticipated and thus we spent far more time than we thought we would have had to on this part of the project.

On the other hand, we accomplished our goal of creating the API for the Beaglebone to collect data. In this case, because of our design, although the API we have written transmits data over USB, it can be easily modified to transmit that same data over a different communication protocol. This thus also meets one of the goals we set for ourselves which was to make the software we wrote as easy for the team to modify as possible.

In terms of data processing and visualization, we were also able to collect real data, process it and display it in real time. We also managed to create a dashboard that displays the data in an easily readable format. Although we would have liked to create a more sophisticated dashboard,

we decided to design a simple dashboard that would function more as a proof of concept than a final product because we were using python while the team has chosen to use javascript for their final dashboard. Thus we chose to use a simple dashboard that would demonstrate how the dashboard would, in theory, allow baja team members to monitor the drivers progress once fully populated with relevant data visualizations.

Finally, we also built a testing circuit to clearly see the effects of real-world interactions with the sensors, such as adjusting a potentiometer or tilting the breadboard, changing the dashboard in real time.

Overall, we are satisfied with the outcome. Although the code will, of course, need more development and refinement, we think we have successfully designed a foundation for the baja team to continue with our framework and implementation in order to collect and process data from the car.

5.3 - Reflection on Design

We think the best decision we made was to use a Model-View-Controller architecture and to constantly be mindful of ensuring that interfaces were standardized and consistent. This ensured that the Model, View and Controller were all implemented with interfaces that allowed any part to be swapped easily without needing major rewrites in the other parts. This design benefitted us many times including when we swapped the trace playback module for the module that would read the actual sensor data and when we had to rewrite the model to be in two parts instead of one. By keeping the parts separate with standardized interfaces, we were able to make these changes without needing to make major changes to the other modules.

One of the things we would do differently next time would be to implement the serial communication much earlier. For this project, we designed everything to run on a laptop first before we started to implement it on the beaglebone. This led to us making major changes to the model because the serial communication now had to be written into the model since it was not necessary when everything was running on the laptop. If we had implemented serial communication much earlier, we would have realized the model needed to be in two parts in the first place and avoided the major changes we had to make later.

5.4 - Division of Labor

Division of labor was another aspect of the project that benefitted from our Model-View-Controller implementation. We were definitely able to divide the project in a way that allowed us to each work independently on our own module(s) and then meet to integrate our work together because we designed our software to be modular in the first place. Since we made sure to use standardized interfaces throughout our modules, we were able to ensure that modules could be swapped in and out without needing to rewrite the module that was calling or importing it. In particular, it was very useful in implementing the dashboard because we used a command prompt-based text view for a long time because the dashboard took us a long time to implement. However, once we had the implementation, all we needed to do was swap the text view out for the dashboard. Because we had standardized interfaces, the swap was done with minimal changes to other modules.

5.5 - Bug Report

The largest bug we had involved getting the Tkinter GUI to update continuously and read and store data from the Beaglebone using pyserial simultaneously. To display the GUI, we wanted to use the Tkinter method mainloop(), so we could still interact with the GUI while it was running. This created a problem because once mainloop is running, no other programs can run, meaning that data collection and storage would stop. Initially we tried to solve this using threads. The view would be one thread and the model running on the laptop reading from the beaglebone would run on the other thread. This would allow the view to read from the model which was being updated continuously. This proved to be problematic because we discovered Tkinter did not interact well with threads and created complications which we were unable to deal with. We solved this problem by combining the two modules into one and creating a reference to an instance of the model inside of the view. The view now contains the model and this allows it to read from the Beaglebone and write the data to the model. We then created an update function within the view which reads the data from the Beaglebone, writes the data to the model, then changes the dashboard to display it. The update function is also configured as a time-based function which calls itself again after a given interval. We set that interval to be less than the interval the Beaglebone uses to transmit data. This way, we are storing all of the data, and are able to continuously read and store data while the dashboard is running.

5.6 - Conclusion

The aim of this project was to create the framework and software to read and process data from sensors on a Mini-Baja car and visualize those results in a meaningful way for pit crew members or the driver. Through the course of this project, we have developed a relatively easy to use framework that would allow Olin's mini-baja team to continue developing the software for more sensors. We have also demonstrated the potential for a dashboard-style visualization of the data and how it might be useful to the mini-baja team. Overall, we think we have met the goals we set ourselves for this project.