Software Systems Final Report

Amanda Lee, Rachel Mathew, Shane Skikne, Michael Warner

1. What subjects did we explore?

For our final software systems project we decided to explore embedded systems and options for optimizing writing data to an SD card. For the embedded systems part of this project we mounted a distance sensor and a photoresistor to a cart that is powered by an Arduino Uno. Using timer interrupts we can take data from the sensor and then use the distance sensor data to determine if the cart needs to turn away from an oncoming object. By programming in a path for the cart to take, we can take a range of data that can be stored on an SD card. This data can be written to the SD card using interrupts. In order to write data to the SD card in the most effective way we conducted a series of tests that helped us determine how much data we should be writing and how often.

2. What resources did we use?

For hardware we used an Arduino Uno, a cart with two DC motors mounted to it that we salvaged from the POE lab, various electrical components that we picked up from the ECE stockroom, an SD card and SD card reader that we purchased. The specific sensors that we mounted to our cart were a photoresistor and a short range distance sensor.

The software was done in Arduino which is an extension of C/C++. We used the Arduino documentation, and examples that we found online to guide our work. We also referenced Homework 4 which was where we used timer interrupts to make the SoftSysSynth.

3. What was our learning goal, how did that go, and what did we get out of this module? Have we achieved our goals?

Mike: I wanted to learn more about the Arduino, specifically what its limitations are. From this project, I definitely have a better understanding of the maximum sampling rates, maximum writing rates to an SD card, and maxing out the onboard memory on the Arduino.

Amanda:

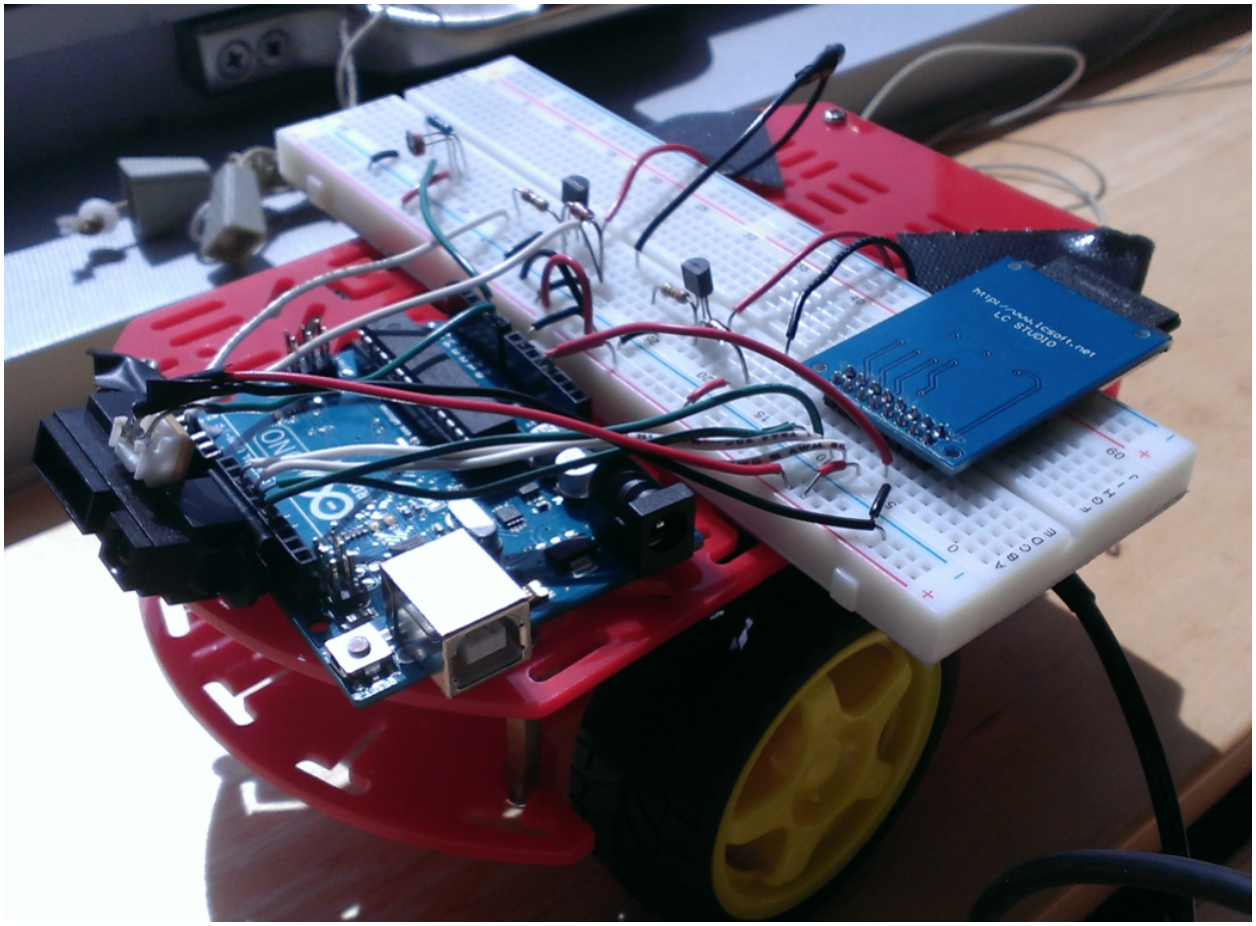
My initial goal was to learn more about debugging embedded systems and possibly gain a better understanding of C concepts. While I did end up doing a lot of debugging, most of it was due to hardware issues (messing up the circuit, motor connection problems, batteries). It was nice to be able to look at some hardware for this project, but it was really frustrating and difficult to fix things that seemed to be breaking for no reason. If there had been more time I think it would have been nice to write more test code that would be able to indicate whether the problem was due to hardware or software. Overall, I may not have achieved my initial goal but I did learn a lot and have a great experience.

What was our learning goal, how did that go, and what did we get out of this module? Have we achieved our goals?

Shane: I wanted to apply some of the interesting aspects of C I've learned and explore some different relationships in the software and hardware. While I didn't get to do much new work in C, I did get a really cool opportunity to explore different relationships between hardware and software. My favorite was thinking about and optimizing data storage forced me to think about all the different factors at play and how to optimize as many of them as possible. I think got a lot of practice is taking into considerations many different limitations when planning and figuring out how to organize them. I also got some practice in ensuring that my plan was as efficeint as expected once the code was implemented. While I might not have achieved all my goals, I definitely got a lot out of this project.

Rachel: I initially wanted to learn more about performance testing, and how memory management works on the Arduino. I was able to learn about the different types of memory with the Arduino, and look at the performance. I definitely think that there is a lot more to learn about performance, a lot of time for this project was spent debugging hardware issues.

4. Deliverables:



Above is a picture of our final embedded system; the motor powered cart with the mounted Arduino, sensors, breadboard, and SD card reader. Both the short range distance sensor and the photocell are pointed at the front of the card so that the system can sense what is ahead of it. Initially, were using a timer interrupt at 2kHz which would take data from both sensors and sent the distance data to check whether the cart was approaching an object. If the cart is approaching something, it will turn off the right motor in order to turn right until it is no longer blocked by an object. This timer interrupt was eliminated in the final code system because the SD card was using Arduino provided interrupts. The final system can be plugged into the mounted battery pack and it will gather data.

We ran into a number of hardware challenges with the embedded system that took a lot of our time. While the motors that we started with were fine, some of the wires started falling off taking the majority of the connection to the motor. After many failed attempts of trying to solder on new wire to a connection the size of a dot, we finally decided to switch it out with a motor we found in

the POE lab. Because we only replaced a single motor, the other motor is now being finnicky and would probably have to be replaced in the future.

After creating our embedded system we performed a series of tests to assess the performance of the SD card. Taking into account performance, speed, and reliability we found a method that we believe to be most efficient. Namely creating a buffer of 256 bytes in SRAM allowing it to fill up with data before saving it to the SD card.

For our first performance test we wrote a series of increasing sized blocks and tracked the time that it takes for the card to write to SRAM. As with all tests presented here, we used the micros() function to time how long each write takes.

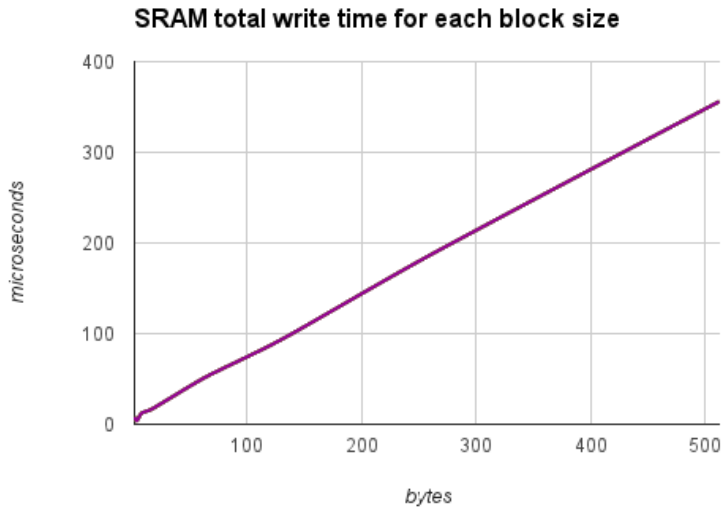


Figure 1: Writing blocks to SRAM. Linear increase in time for each new byte written.

From Figure 1, we were able to calculate the speed to write a block of a particular size. As you would expect given by the linear increase in time for increasing block sizes, the byte time for a particular byte size remains relatively constant, as shown in Figure 2.

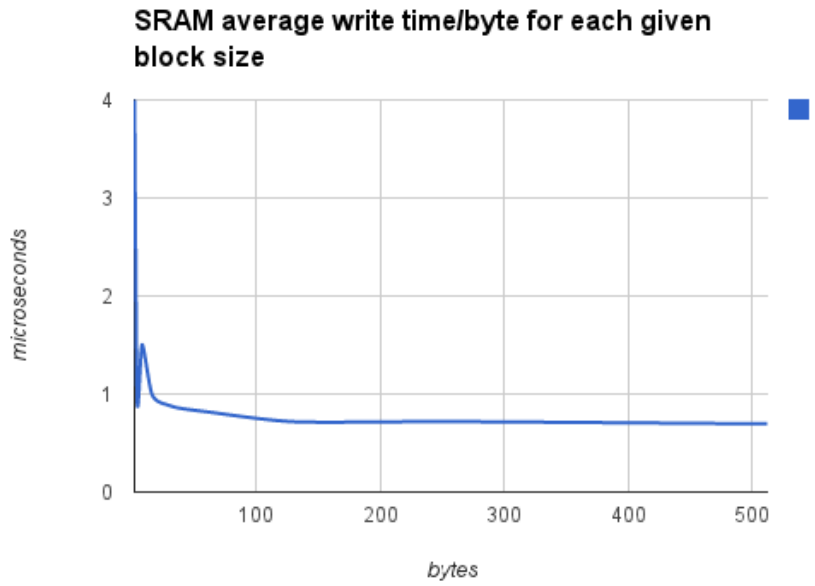


Figure 2: Average time per byte for SRAM. Relatively constant and very fast at an average 1.25us/byte

After experimenting with SRAM, we started to experiment with EEPROM memory. This memory is non volatile, meaning that it will persist after the Arduino will store these values after the Arduino has been turned off. In order to write to this memory, Arduino has a library that allows you to write and read to EEProm. We performed the same two tests for the EEProm that we did for the SRAM and obtained similar results, only that it is dramatically slower.

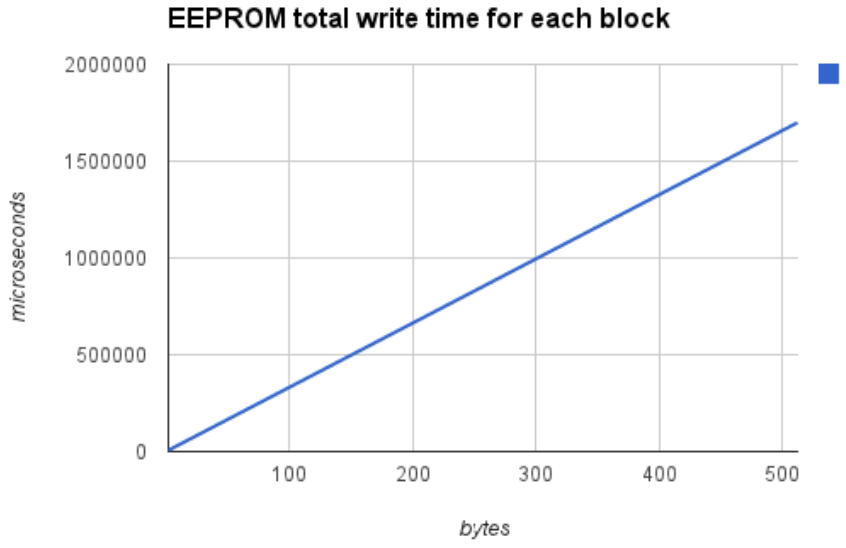


Figure 3: Writing blocks to EEprom. Linear increase in time for each new byte written.

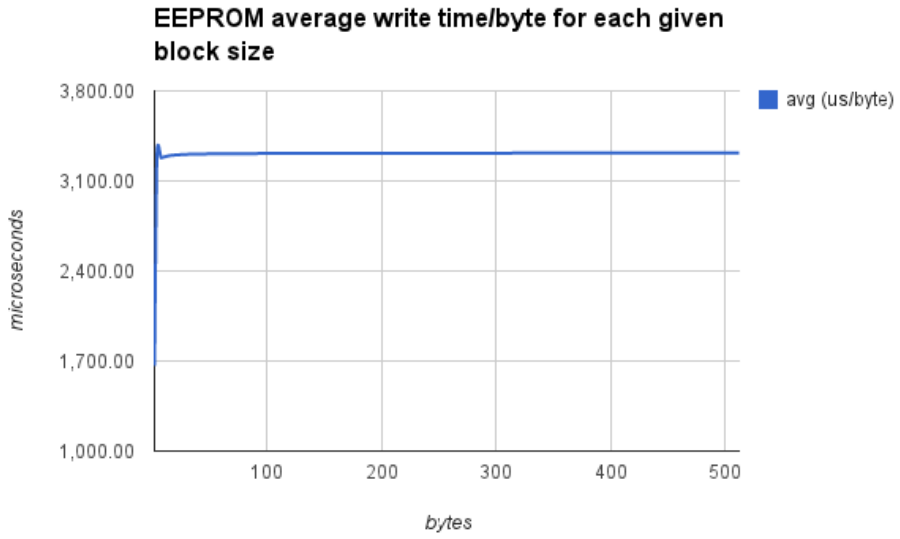
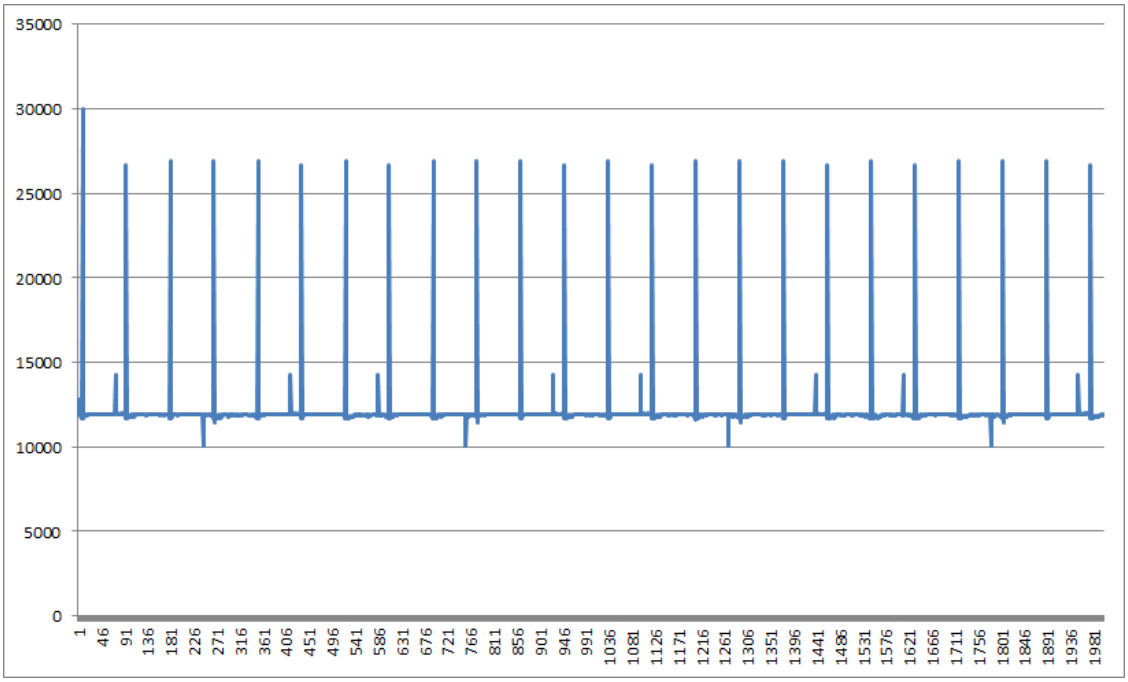


Figure 4: Average time per byte for EEprom. Relatively constant but much slower than SRAM at an average of 3300 us/byte.

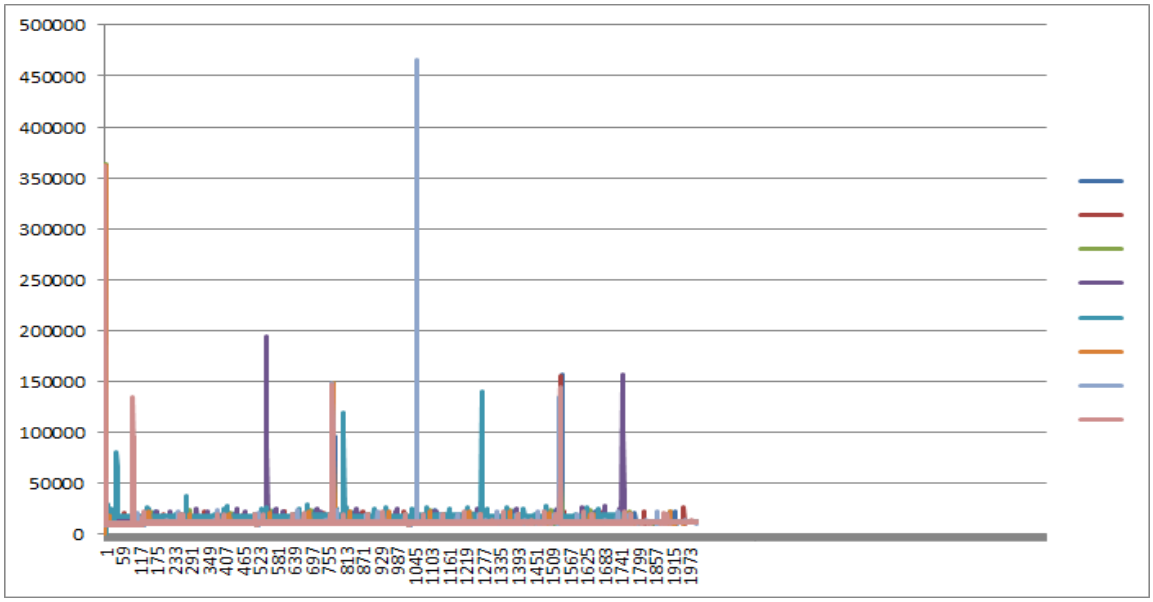
After testing write speeds for different block sizes we looked at the performance of writing individual bytes to the SD card. The graph we obtained showed us that the amount of time taken to write to a byte spiked roughly every 86 bytes. Taking up to 30,000us to write to the card. We are not quite sure what this behaviour is a result of.



(vertical axis is microseconds, horizontal axis represents the test#)

Figure 5: Time taken to write single bytes to the SD card.

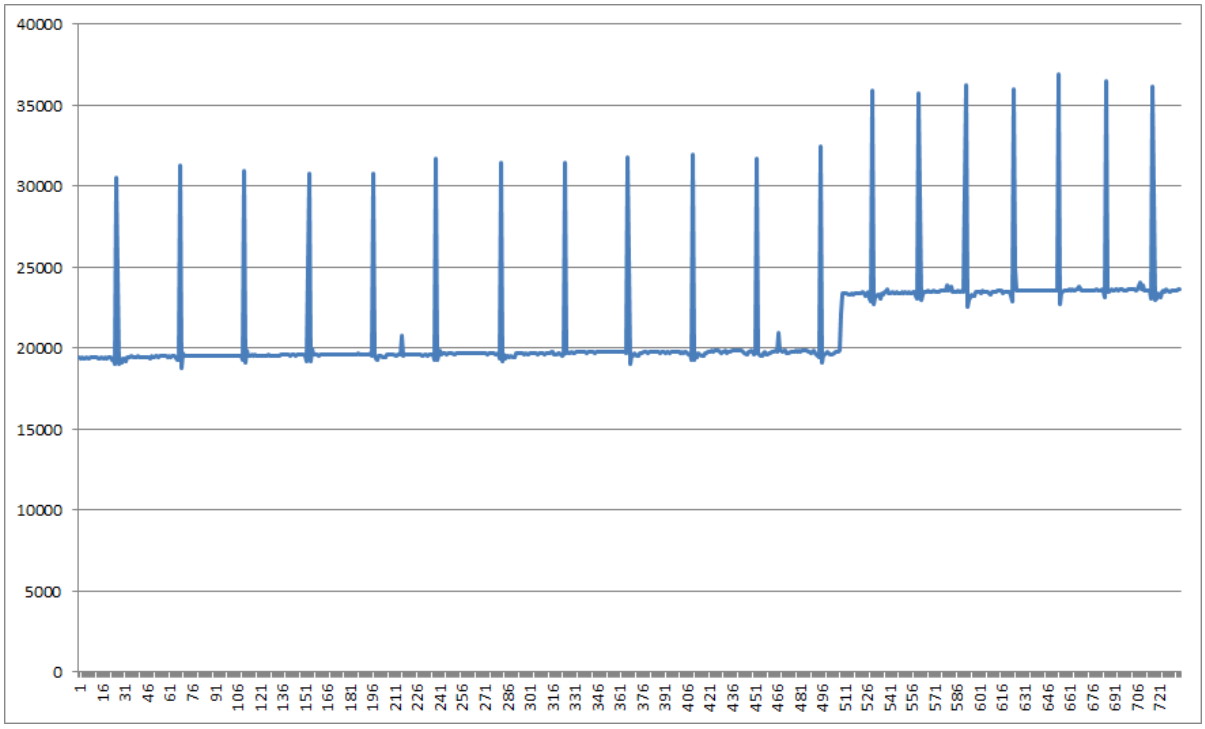
After looking at the performance of writing single bytes we experimented with writing increasing block sizes to the SD card. As shown in Figure 6 of writing blocks of 256 bytes, the 30,000us worst case write time breaks down around 15kB. Huge random spikes can be seen up to 400,000 microseconds. According to some research online, these spikes can be attributed to the SD card erasing large blocks of flash memory and rearranging them.



(vertical axis is microseconds, horizontal axis represents the test#)

Figure 6: Time taken to write blocks of 256 bytes to the SD card

Interestingly the SD card takes consistently longer to write files longer than 512 bytes, as shown in the clear jump in Figure 7 and 8. This is due to the fact that the FAT32 filesystem, which the SD card is formatted to, writes its sectors in 512 bytes. The extra time incurred is due to the filesystem accounting for writing past that sector.

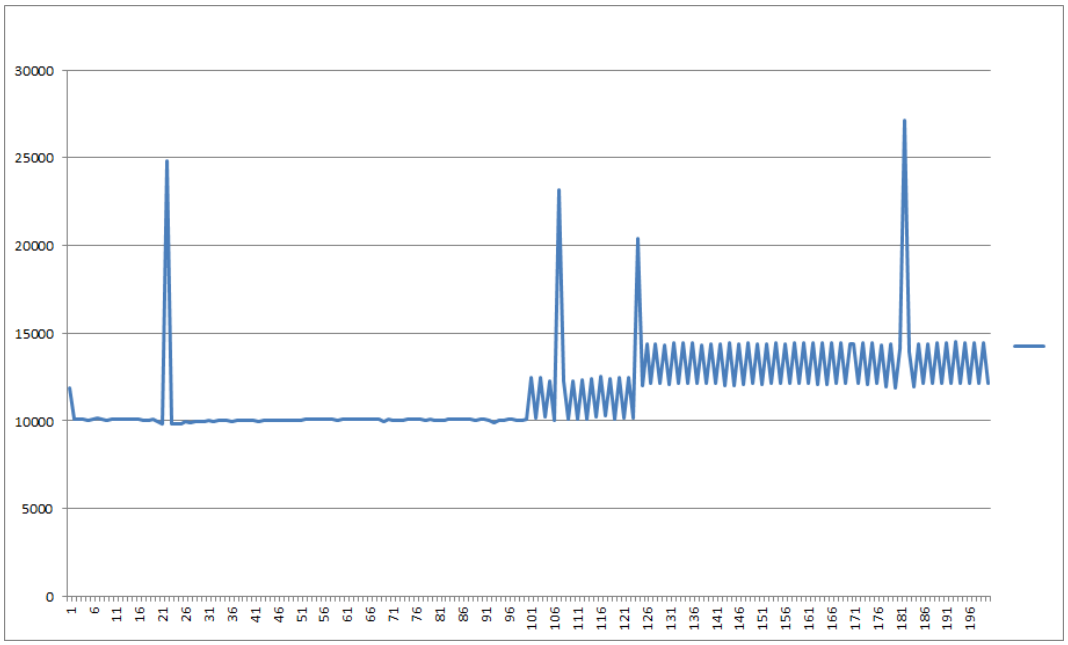


(vertical axis represent microseconds, horizontal axis is the size of the block in bytes written to the SD card)

Figure 7: Time taken to write a byte to the SD card with increasing block sizes.

(vertical axis represent microseconds, horizontal axis is the size of the block in bytes written to the SD card)

Figure 8. Time taken to write a byte to the SD card with increasing block sizes for many tests.



(vertical axis represent microseconds, horizontal axis is the test#)

Figure 9. Two different tests showing the penalty for writing the first byte.

Figure 9 shows two sets of tests taken one after another. Tests 1-100 are the time taken for writing a single byte to the SD card, while tests 101-200 show the time taken for writing a block of 256 bytes to the SD card. The average time to write a single byte is 12000 us while for 256 bytes the time is 13,000us, or nearly 50us/byte. This shows that there is an incredible penalty for writing the first byte to a file. To maximize efficiency, then, it is imperative to write to the file in as large of blocks as possible.

SD card write strategies

All of this data informs our strategy to writing to an SD card. To sample data and writing that data to the SD card at a maximum sampling rate, it makes sense to use buffers. The SRAM technically has 2kB of memory that can be used for the buffers, but because this memory is occupied by other variables used by the sketch, we found that 730 bytes was the maximum we could use. Thus, two buffers of 256 bytes was chosen.

One buffer is constantly being written to the SD card, while the other buffer is being filled by samples taken by the sensor using interrupts. When the buffer is filled, it signals the mutex to allow a block of samples to be written. Then the other buffer is filled while the first buffer is being written to the SD card.

If your total data is less than 15kB, then you can write your buffers at a period of the worst case scenario of 30,000 us. This ensures that even in the worst case time taken to write your buffer to the SD card, you will not miss any samples.

1/(30,000*10^-6s) *256 bytes = 8500Hz

If your data is greater than a total of 15kB, you can write your buffers at a period of the worst case scenario of 400,000 us. The much larger period is due to the spurious spikes caused by the flash rearranging itself.

1/(400,000*10^-6s) *256 bytes = 640Hz