Greg Edelston
Victoria Coleman

# Final Project, Final Report, Final Destination

*Memory management for long-term Arduino data collection*

## Introduction

Initially, the project was intended to be done jointly with the group that did serial communication with a sensor to record the energy uses on Olin campus. Our goal was to collect sensor data at regular intervals, signal when the device is full on memory and thus ready for an upload, upload data to the computer and free it as it is uploaded, and maximize the length of time that the device can run without need for an upload. This project was particularly compelling to us, because it showed a very relevant, real-world application of what we had learned -- relevant enough that we had been explicitly told that facilities wanted this device.

When the subteam working directly with the sensor informed us that they would not in fact be able to collect data from the sensor itself, we split into separate teams. Our team became much more focused on minimizing the size of each datum and effectively transferring data to the computer for longer-term storage.

## Background/resources

The most useful resource to us was the Stino plugin for Sublime Text, which allowed us to compile and upload Arduino code without having to use the tremendously frustrating Arduino IDE. Having discovered this tool, we hope to never again have to use the Arduino IDE.

We also discovered that the Linux command line can act as a serial port using /dev/tty, which we found very convenient for quick testing. This significantly expedited the writing of our data transfer protocol. However, this also required some cleverness on our part, since any calls to printf() would create output to the terminal, which would then be interpreted by the serial port. This tool was thus useful to us only if we were very careful about our debugging.

## Work and Learning

The issues we set out to tackle were how to work with the little amount of memory on the arduino's EEProm, how to allow the data collection to run as long as possible since this application has a large time-scale, and to make it a robust system for the user to use. Some of the main decisions to make were what data structure to make, how to handle the communication between the computer and arduino, and how to store the time information.

We chose to implement the main data structure as a linked list simulating a queue. This had a few advantages over another possible implementations such as a circular buffer. A linked

list was sufficient for making the system “human-proof” for uploading: if the Arduino was unplugged from the computer before the data transfer was completed, then the untransferred data would stay in memory, and the first untransferred datum would become the new head of the queue. The linked list also held an advantage over the circular buffer because it was more extensible. With a circular buffer, we would have had to declare from the outset how many elements there would be, which would require knowing the size of one datum and the amount of memory at hand. The linked list, which adds new elements as necessary, allowed us to fill memory “to the brim,” regardless of the size of the data struct. That in mind, it would still be possible to make a dynamic system that initially checks how much memory there is and then determines how many elements will fit, but that takes much more overhead if a more simple solution will be sufficient.

To send the data to the computer, we decided to use the robust ping system for sending data. In this way, the user could plug in an request information from the arduino one datum at a time. In this way, the user does not have to wait for a complete download before unplugging it takes a long time (although this may not be an issue for the arduino since it has so little memory, it felt useful to be able to do because it can be practical approach with other memory storage units). With that method, only one datum will be lost at most when the user unexpectedly disconnects from the arduino. The protocol we used is that the computer will send a bit of information to the arduino, and the arduino would send back information in the following format:

“T _ _ _ _ _ _ M _ L _”

The “T” represents that the header that the next 6 bytes are time stamp information, the byte after “M” represents the most significant byte while the “L” represents the least significant bit. The computer reads this information over serial and saves the data to a file for future analysis. The best solution for the application would be to have a supplemental storage device or to transfer the information over the internet as soon as it is received, but we were interested in dealing with the arduino by itself and dealing with the close memory restrictions.

Another decision was what data to store in list: because we collect data at uniform, known times, and time is such a memory hog, it is unnecessary to store the full time-stamp for every piece of data. There is a known reference time that would be when datum 0 was stored, but the list would only have indices. The indexes allow us to back-solve the time from the reference point and send the complete time-stamp over serial with the datum when requested. By doing it this way, the data is not 7 bytes instead of 12, which increases our memory capacity by over 70%..

With these considerations in pace, we created a system that stores data into memory along with time information for data collection from sensors. Since the data collection side was unable to collect data due to hardware issues, we have self-generated data for proof of concept. The data collects then stops collecting when it runs out of memory. We could have had it delete the old data easily with simple list manipulation similar to exercises we’ve done in class, but we were dealing with serial issues. On the computer end, a script written in C pings the serial port to request each new piece of information and writes the received information to a file. In this way, the computer can disconnect from serial with at maximum only one piece of data lost. The arduino then starts or continuing to store data into memory.

What we did not accomplish was full integration because we ran out of time and arduino was being finicky. Based on testing by typing in the terminal and seeing what the arduino prints, we believe that reading of the time stamp will work when it is figured out why the serial data goes bad part way through of a bit, and it wouldn't be too much effort to extend it to the data as well and have it tested.

## Conclusions and Future Work

During this project, we learned a lot about serial communication and further solidified our familiarity with c, which was a nice practice. If we were implementing this device "in real life," we would have set it up with a Wi-Fi shield. That way, we would not have to worry about minimizing the data size or leaving large delays between samples, because we could upload and free data as soon as it is collected.

If Wi-Fi were not an option, we would choose a much larger microcontroller, perhaps the Arduino Mega. We were quite surprised to learn just how little memory an Uno really has. The Mega has four times as much memory as the Uno, which would significantly aid our device's runtime.

We also considered implementing smarter sampling delays. For example, if the readings are growing erratic, it would be useful to sample more frequently in order to better understand the erratic data, whereas a long stretch of relatively unchanging data might not be the most valuable use of our precious memory. A simple way to implement this would be an inverse relationship between the sample delay and the recent change in data with respect to time.