Prusto Watch #4: A Retrospective

May 2018

The past few months of working on the Prusto Watch has been both frustrating and rewarding. This post will go through what I did and what I learned during the process.

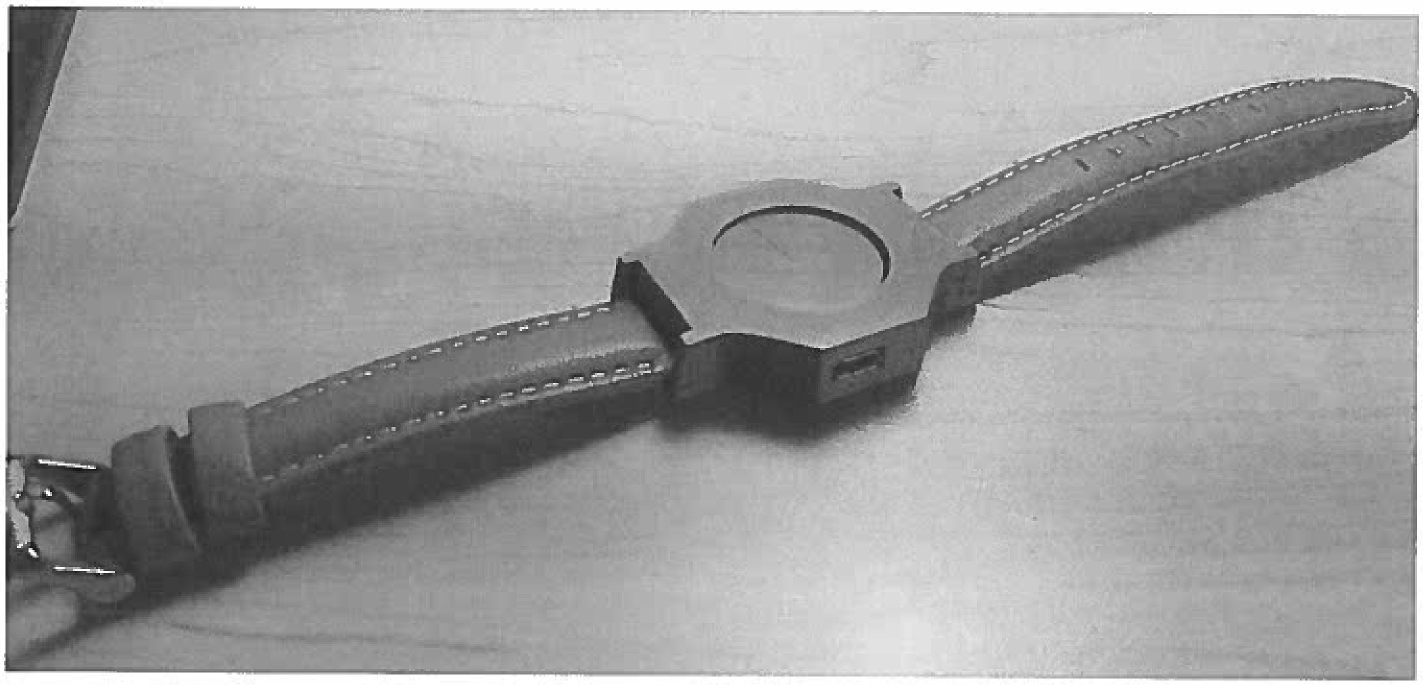
Project Summary So Far

First, I will start off with a summary of what has been accomplished so far in the project.

- Four PCB revisions, with the final revision being a great form factor and also a two-layer board.
- Functioning BLE, Display and button peripherals.
- Two libraries written in Rust which are hardware-agnostic. One for the BLE module and one for the memory display.
- Bug fix for the `embedded-graphics` library's Bresenham line-drawing algorithm.
- Developed a simulator for the `embedded-graphics` library.
- Various additions to the `stm32f30x-hal` library (although not upstreamed due to maintainer issues, I've found some of my additions present in other forks).
- Basic setup for the Prusto Watch firmware using the RTFM framework.
- 3D printable case design which fits all the hardware and is only 9.5mm thick! (It is surprisingly pleasant to wear)

Even though the Prusto Watch is not a fully usable smart watch (yet), I think I have made significant progress in getting something very close to a usable smart watch. Ideally over the next few weeks I will be able to develop out the firmware side of the watch and make it more functional.

The most recent Prusto Watch in it's case shown below:

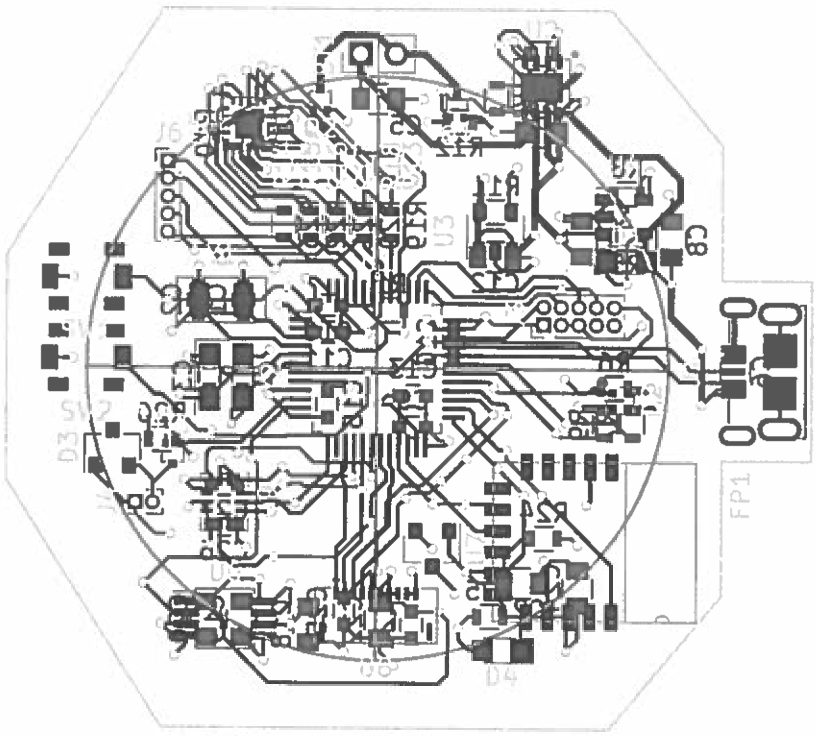


Lessons Learned

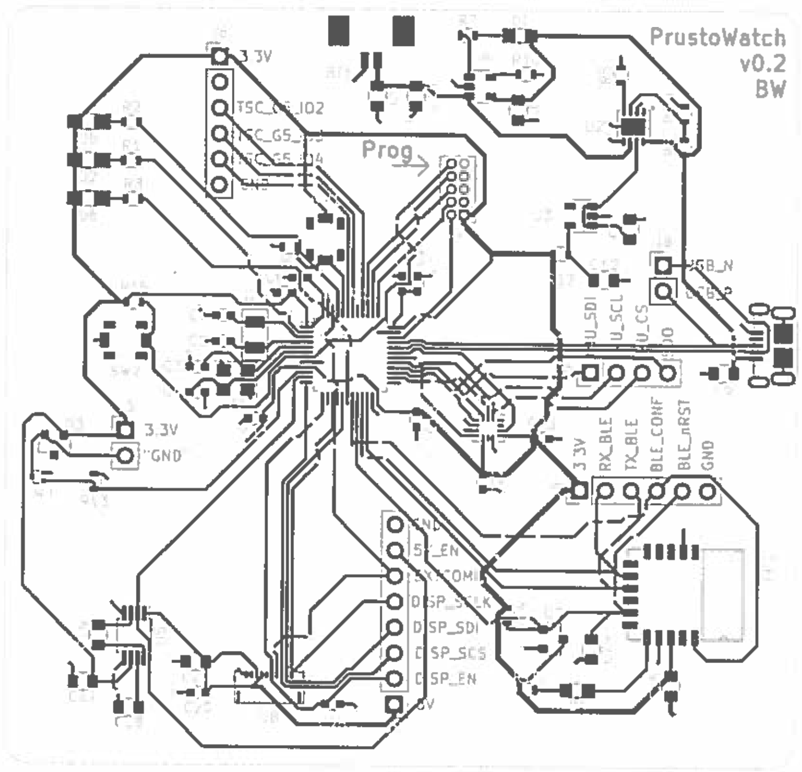
The main goal of this project was to learn and to document much of what I had learned. So I will go over some of the major points of failure of the project, since they seem to coincide with where I learned the most.

1. How to Design Hardware For Debugging

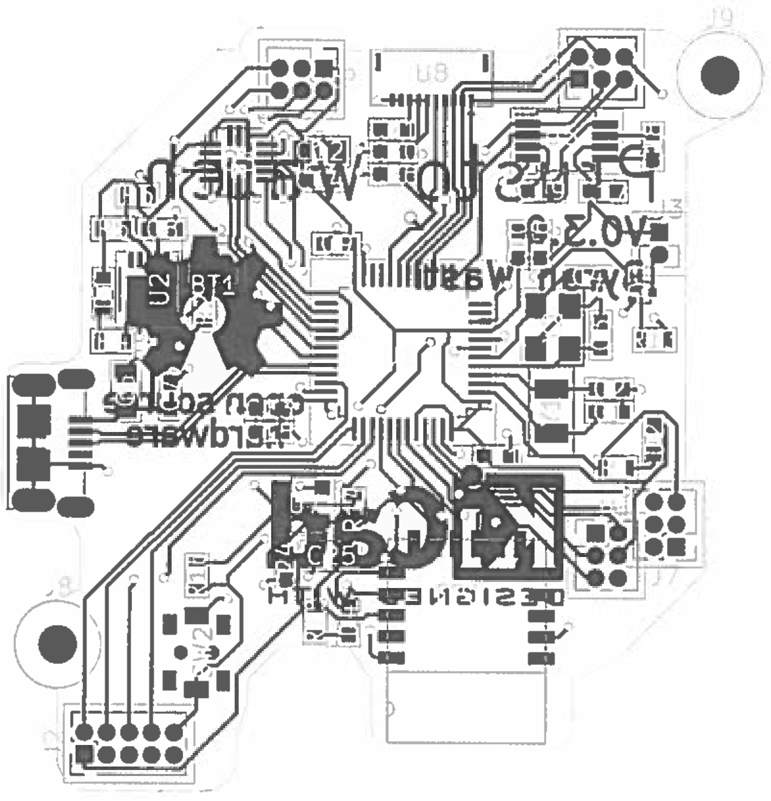
The most significant time sink, and also the most avoidable, was dealing with hardware issues. One of the biggest issues I seemed to continually run into was being unable to easily scope or otherwise debug what was going wrong. By the fourth revision I had finally figured out a system that worked, but not without three revisions of headaches.



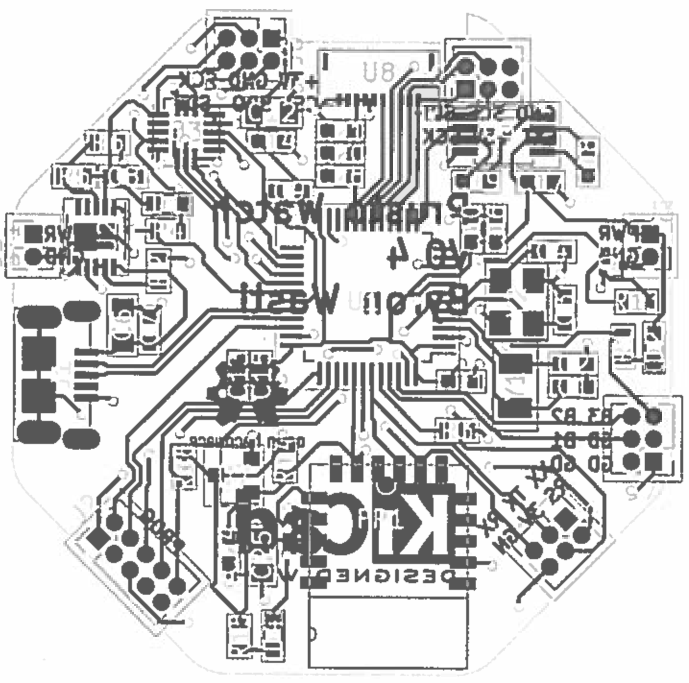
The first PCB (shown above) I naively did not put any headers or other scoping points, and thus was basically unable to know what was going wrong. I learned how important it is to put debugging points on hardware, especially on the first revision of a board.



The second PCB (above) I put large headers on, and was able to make a lot of progress. However, I was unsure of which direction the connector should face on the LCD. I spent far too many hours debugging "software" issues when I just needed to flip the connector. I learned that if I am unsure of a particular connection, I should just design a board with all of the possibilities at once so that I can test them all in parallel.



The third PCB revision (above) I made sure to have headers, however to save space I made them 1.27mm pitch thinking it would be easy to just solder leads to them for testing. This worked for the most part, but was very unwieldy and difficult to solder correctly. I also did not put silkscreen for any of the connections because I did not have space on the top layer. I quickly realized once working with the board that silkscreen on the back would have worked just as well. Finally, I thought having such a weird board shape would be fine when designing a case, which was far from the truth



The final PCB revision (above) had a board shape designed in CAD, so it was very easy to design a case for. Additionally, I ordered 1.27mm pitch header breakouts which made it much easier to quickly test various peripherals. Finally, I made sure all of the external connections did not interfere with either the battery or the display, which made the debugging components available even when the watch is fully assembled.

Overall, the main lessons I learned was to design hardware with a focus on being able to debug it. Even having a pad available to test voltage was a sanity saver at various points of development.

2. Using the Bus Pirate to Verify if it is a Software or Hardware Issue

One of the most challenging parts of embedded development is the fact that issues can either be hardware or software. In order to deal with this in the past I relied on knowing firmware really well, and basically trusting that my firmware was correct. This works surprisingly effectively with AVR and other simple microcontrollers, since the firmware is straightforward and there isn't much complexity going on. However, Arm is significantly more complicated than AVR and I was unfamiliar with it. Thus, many of the issues I ran into I had no idea if they were hardware or firmware or both.

This is where I learned just *how* useful a device like the Bus Pirate is. The Bus Pirate allowed me to quickly verify if problems in communication with a peripheral was hardware or software, since I could completely rely on the Bus Pirate getting the communication protocol correct. If I had known about solutions like the Bus Pirate at the start of this project I could have saved a lot of time debugging silly issues (like the LCD connector).

3. The Importance of Knowing the Microcontroller

For a while I relied on my very basic understanding of Arm in order to get firmware development done. I avoided diving deeply into how Arm worked until I ran into a very strange error, which I talk about here. By putting off truly learning the details of how an Arm chip works, I wasted time "trying" things until it worked instead of being able to effectively read the MCU's datasheet. I also missed out on learning to use awesome debugging tools, especially GDB, effectively.

Although now that I have a much better understanding of how Arm works and how to debug it effectively, the next project that I use an Arm chip on will go much more smoothly.

4. Using Rust to Start Was Probably a Bad Idea (but it thankfully worked out)

At the beginning of the project I was excited by the idea of using Rust, a language which I think has amazing promise for systems development, as the main programming language for the Prusto Watch. However, at the time Rust barely had support for Arm chips.

Through the project Rust actually gained mainline support of Arm, although only on the nightly branch. This meant that every few weeks things would break with no documentation. In general, the documentation is not exactly stellar for Rust on embedded (which makes sense given how new the ecosystem is).

In retrospect, it was a really bad idea to use Rust, especially since I did not know how to develop for Arm using C and thus couldn't even reference what worked in C. If Rust's Arm support had been poorly implemented or otherwise buggy, I would have been in a whole heap of trouble.

However, in general things worked really well. I was also able to contribute to the embedded ecosystem, and I plan on continuing to contribute. From my experience I think Rust has amazing potential to make embedded development much nicer and more safe, and it is progressing in a really awesome way.

Other (Smaller) Things Learned

Through the project I also learned a number of smaller things.

I learned:

- How to place components by hand using tweezers. Since I was using 0.6mm thick PCBs instead of the standard 1.6mm thickness, the pick-and-place was not able to properly "let go" of components, and they would not be in the correct placement. I was forced to use tweezers to place components, which I got pretty good at by the end.
- The importance of having components ordered before you need them. Twice I was burned because I did not realize I was out of a certain component, or really low on a component, while I

was soldering up a new board. I ended up ordering an SMD case for keeping all my components, and developing a system for keeping track of components I am low on.
- Getting super thin PCBs may not be the best idea. Once I went to 0.6mm thick boards they began becoming extremely unreliable. Depending on how they were held or looked at they would fail to turn on, or power certain components, or even allow me to program the MCU.

Whats Next

I plan on continuing firmware development of the Prusto Watch to make it into a usable smart watch of sorts. Depending on how that goes I will continue refining the hardware and firmware. It would be awesome to have a community that is developing a fully open source smart watch, so once I get it to a certain point I hope to have a proper announcement and get others working on it.

Overall, I think this project has been a success regardless of where it ends up since I have learned a significant amount about hardware development as well as firmware development with Arm chips.
