**PCB Design**

This week was mostly spent finishing the schematic and PCB layout for the flight controller. Both had reached a good point last week, and so the remaining efforts were implementing the IMU recommended designs, touching up the PCB component positions, silkscreen references, routing power traces, and cleaning up traces and ground planes to maximize signal integrity and current capacity, and minimize voltage drop and noise.

Again, I won’t show how the sausage is made too much, as it is a very, very iterative process and stuff keeps moving around on the PCB until I am satisfied with its position.

However, as things move around, the optimal location to have the signal pins originate off the microcontroller can change. So, in addition to selecting optimal final locations for the components, I frequently checked the datasheet, and a handy ST tool to check which UARTs, SPI busses, etc. could be enabled on which IO pins.

Graphical user interface

Description automatically generated with medium confidence

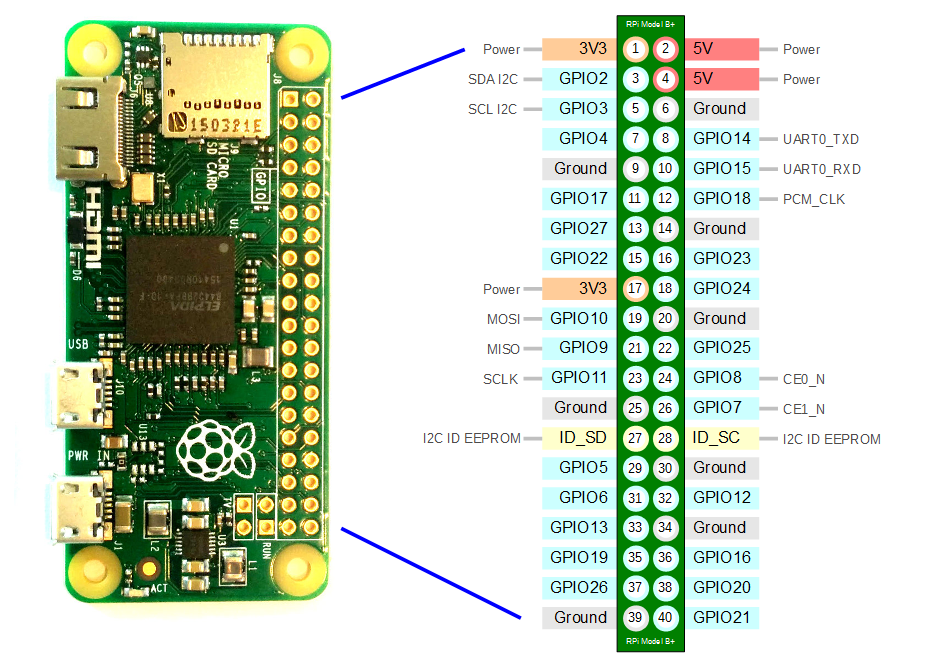
Using this tool, I could validate that my pin configuration was workable with my physical layout, and then implement that on my schematic.

Implementing the IMU was mildly interesting, as I want the capability to use either the MPU9250, the LSM6DS3, or both. So in addition to adding the recommended parts, caps, etc., I added a network of 0 ohm resistor switches to enable the desired configuration as needed.

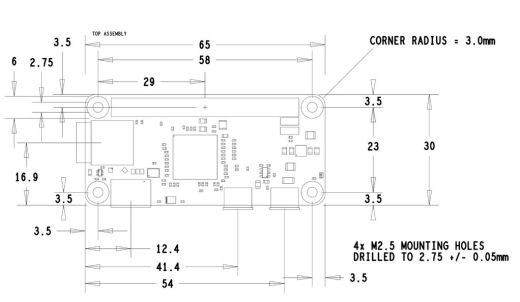
Diagram, schematic

Description automatically generated

I also added the PI Zero connector that I had put off for last, selecting a part of the Pi header to interface to that contains the UART, +5V, and ground. This is pins 1 through 10 at the far edge of the PI PCB. This ended up being a relatively convenient spot after I moved a couple things out of its way.



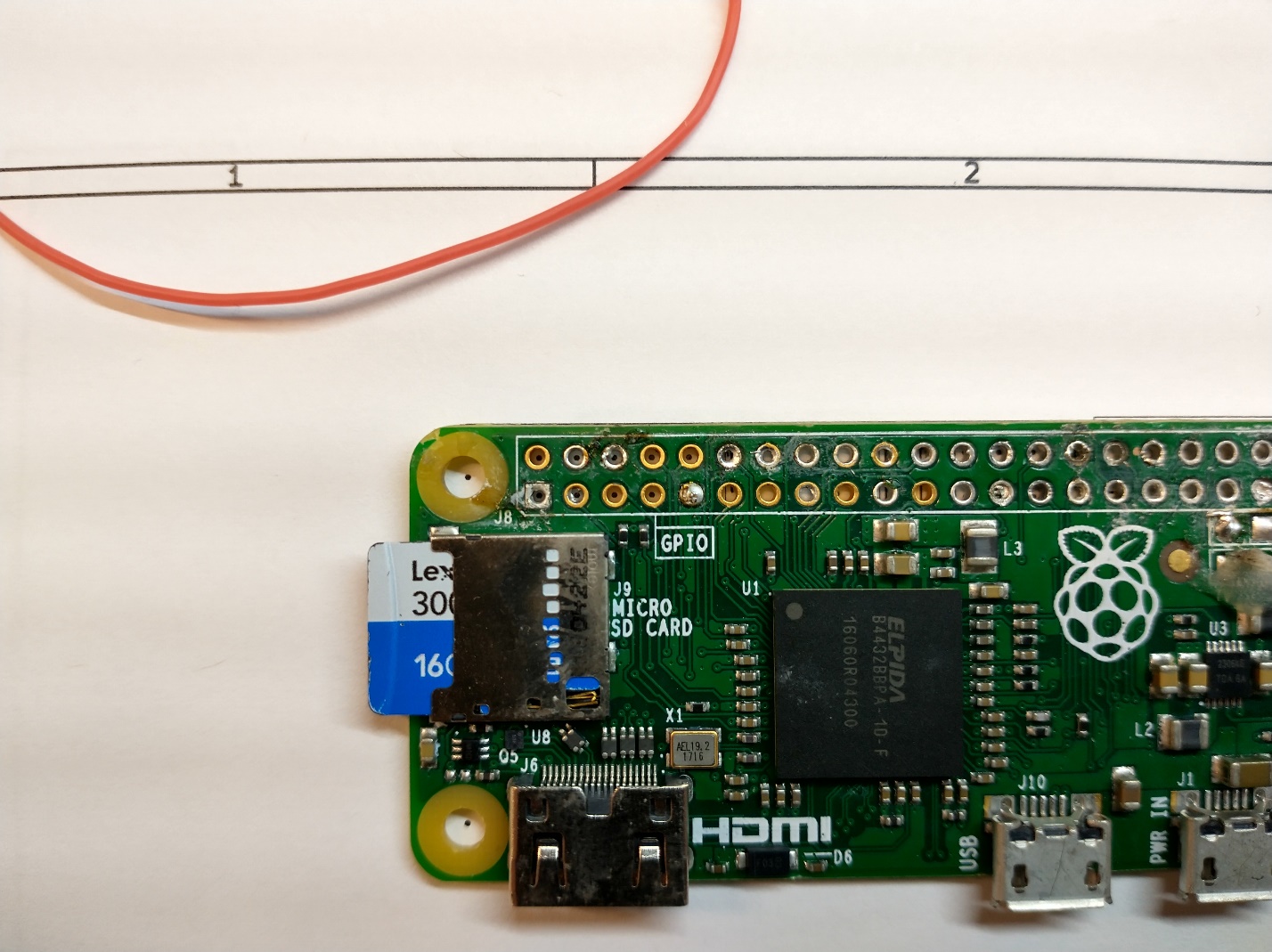
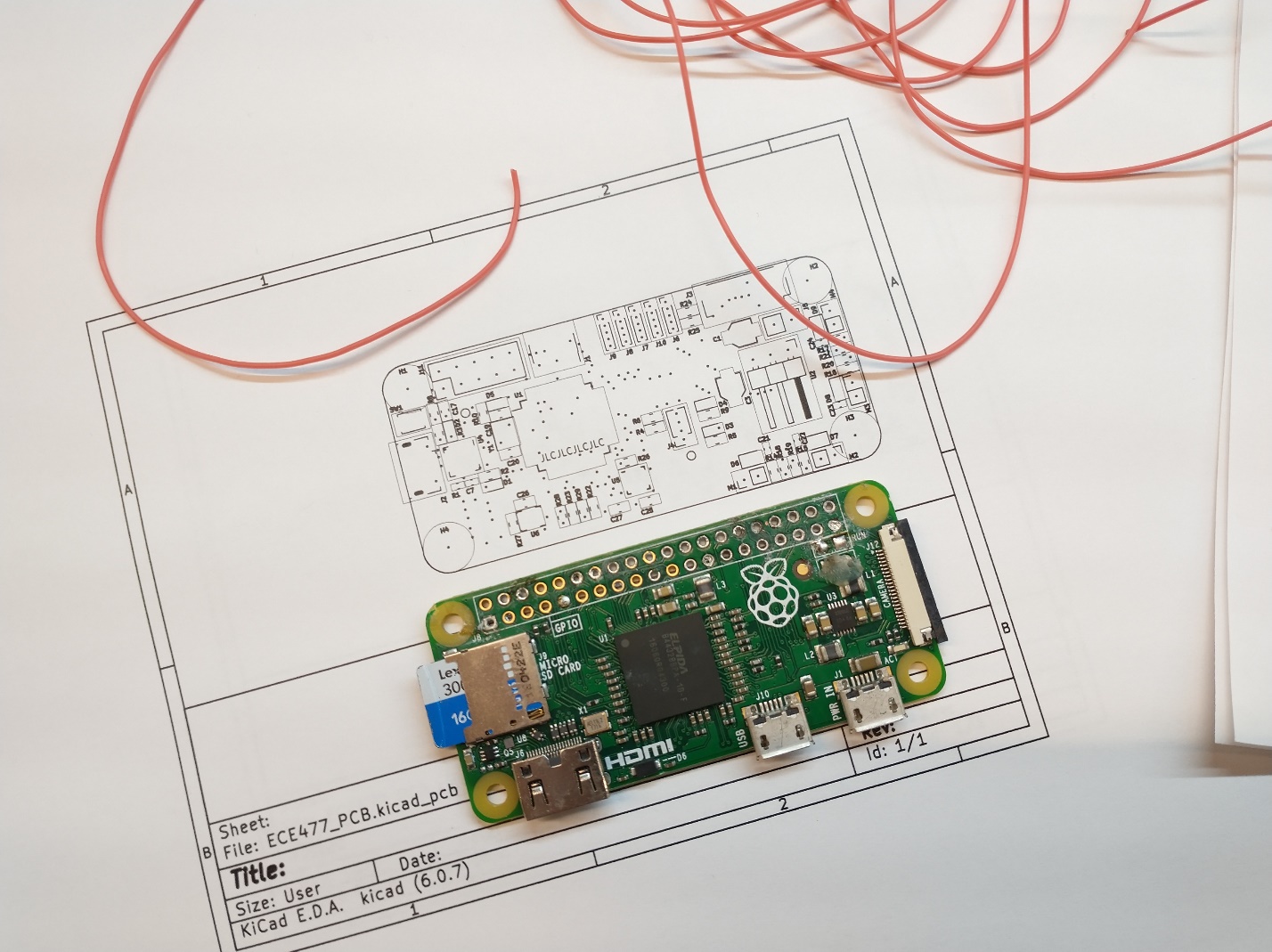
Now that the part was in the ~mostly correct spot, I had to ensure that the connector would line up perfectly by looking up the exact dimensions of the board, and by verifying the setup by printing a to-scale version of the board and checking it against the Pi Zero I own.



Using these measurements, I worked to place it in what I believed was the best position. Then I printed off the too scale version and compared. I noticed that while all the corner holes were properly aligned, the connector was about a millimeter too far to the right. I then readjusted and printed again. Much better! A visual check shows the pins at the exact right spot.

Diagram

Description automatically generated



After much additional moving and redrawing, I got the board to a physically and electrically satisfactory point, with correct footprint positions, trace widths, silkscreen, etc.

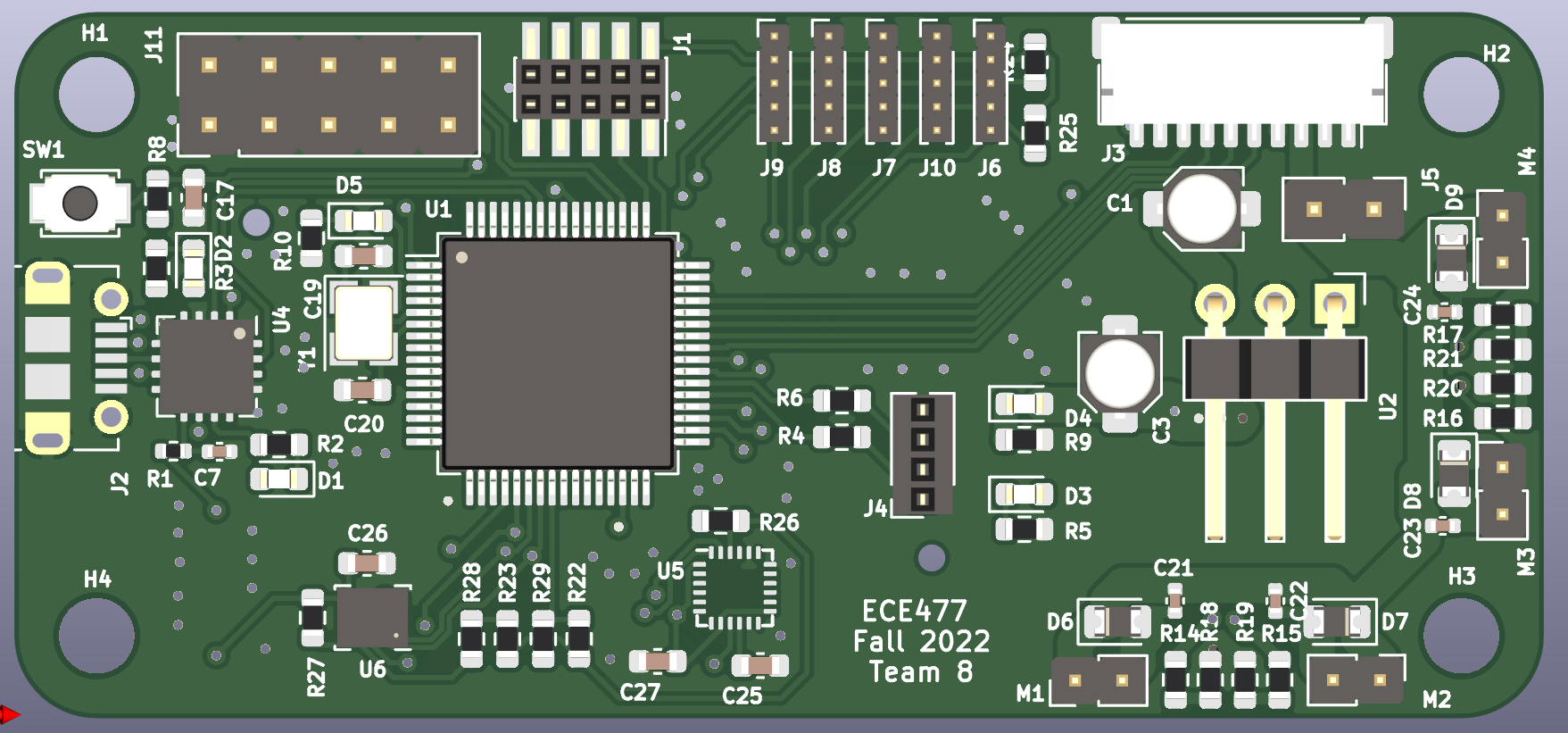
The result looks like this.

A picture containing schematic

Description automatically generated

A screenshot of a game

Description automatically generated with medium confidence



A circuit board with many chips

Description automatically generated with low confidence

**Radio Code**

I also revisited my radio CRSF interpretation code this week, in order to improve my relatively hacky code – which worked but performed almost no data validation (dangerous!), and was very inefficient.

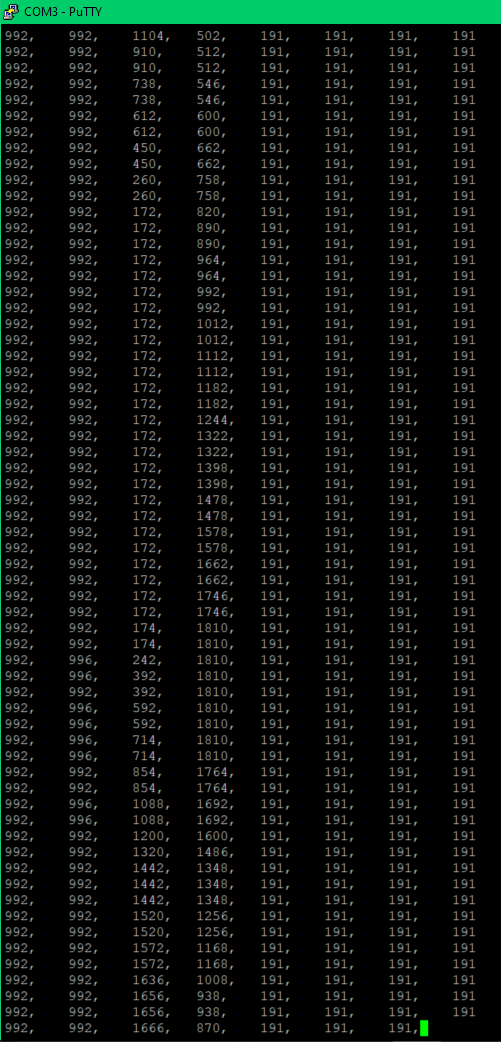
For the new design, I chose a circular buffer, where the ‘start’ of the buffer would be shifted up until the end of the buffer, and then back around, rather than shifting the buffer contents while looking for a valid packet. This is significantly more efficient. Once it sees a valid starting character, it places a struct pointer at that location in memory, and begins to decode the data from that position, pulling out descriptors like the frame length from the point indicated my the struct definition. If needed, the ‘rolled over’ part of memory in the circular buffer is copied over to free space at the end of the buffer so the data frame is contiguous. Then the address, frame type, and CRC are checked. If these are valid, then the channel data is copied over to the global location for the latest radio channel data.

This system actually worked almost perfectly first try, which was great! I only had to fix a couple pointer casts and offsets, and the system was good to go. Another benefit is I went to the source code on the ELRS site and used their #defines and enums, so that my code is using not only the right values, but the canonical names for each field.

Graphical user interface, text, application, email

Description automatically generated

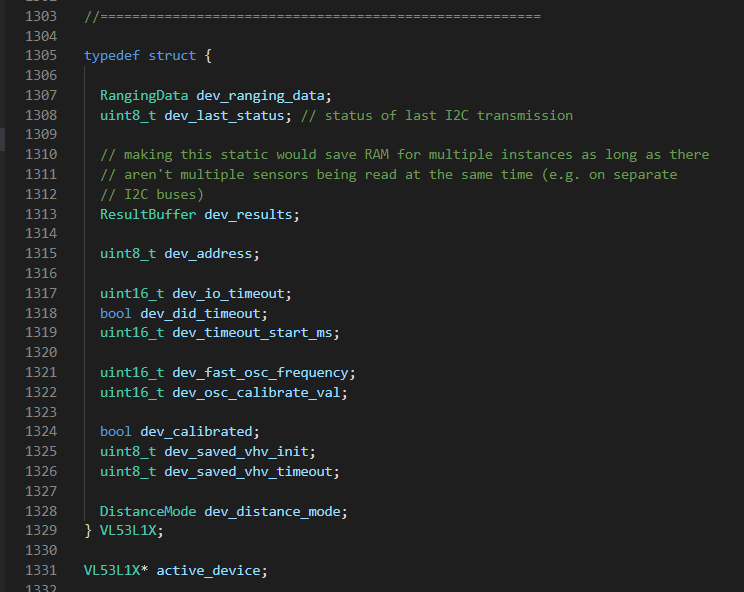
(A quick screenshot to help visualize it a bit)



Data happily streaming in with channel data.

In addition to the radio receiver, I spent time taking the available Arduino driver (a derivative of ST’s required base driver) and removing it’s C++ syntax and porting it to just C, and preparing it for use with our implementation of a I2C bus.

Probably the most fundamental change was moving from data contained in a class instance, to storing all device data in a struct. This results in needing a struct for each specific LiDAR device we use. So I both created a device descriptor struct, and then #define’s translating every reference to their original variable names to a reference to the device descriptor struct pointer’s data.



Text

Description automatically generated

Text

Description automatically generated

We’ll see how this works out, but it seems like a relatively good way to implement instantiation (ish) without all the memory management of C++ on this particular embedded setup.