Prototype Build for the Omnicom portable SDR platform

Haoran S .Diao

December 31, 2017

`Note To MIT:` This document describes my current active project. It is likely that by the time you read this much more progress has been made. I apologize if it is unacceptable to send in an unfinished project, but I felt this one was the one that aligned most with the efforts I'm making. If you want to read more, project updates will start popping up on my personal site http://hairydiode.xyz, any code I write for this project will also be on my github https://github.com/knolax. The hardware I'm building here was built so I could start testing a distributed network idea I had, so for more context on what i'm going got you can check that out http://hairydiode.xyz/network.

Abstract

Digital radio communications in computers currently rely on opaque, protocol specific radio hardware. These radio chips usually run propretary firmware and require proprietary drivers which creates a source of vulnerability. In addition, the protocol specific nature of these chips limit computers to a preset set of protocols. As of late, robust and powerful Software Defined Radios such as the LimeSDR have become widespread, allowing for many possibilities in pentesting and development of new radio protocols because SDRs are general purpose radios where signal processing is done in software. However, SDRs have never been integrated into portable form-factor akin to a mobile phone, even though that's one of the most common form-factors for a communications device. I am building the *Omnicom* as a SDR based device with this form-factor in mind. The primary use cases here are pentesting, and network protocol development. This isn't one of those "on the whim" projects because I actually need something like this to develop a network protocol idea that I've been thinking of.

1 Design Requirements

So given that the Omnicom is going to be mostly used as a development tool, i'm focussing more so on robustness than polish. Other than that the specifications are as follows:

Portable It has to be small enough to carry around and self contained, but not necessary small enough to fit in your pocket.

Tactile User Interface I mean specifically a keyboard here. Personally I despise touchscreens, any sort of Human Input Device that isn't a keyboard requires way too much hand-eye coordination to be fast AND accurate at the same time.

Reasonably Durable The Omnicom has to be mechanically sound, properly mounted components and all that jazz, this requirement is more of a challenge to myself since most of my previous electronics were pretty frail.

Mechanically Sound Electrical Connections - That means I have to keep in mind proper potting staking of all the electrical connections.

Spacious Interiors Since I'm using commercial parts I'm going to have to use standard cables, which take up some space.

Use Mostly Commercial Parts Altough it's fun to fabricate my own components, it's more reliable and less labor intensive to use commercial parts. The power supply is also just a commercial power brick, since it's safer for me not to deal with my own LiIon charging circuit.

Familiar Operating System - The Omnicom has to run Linux, preferabbly Arch Linux, my daily driver. A familiar system means more software support and less grief for me when I have to write some of the drivers.

2 Bill Of Materials

Custom Mechanical Keyboard This means a custom PCB and custom drivers.

Custom PCB I'm not going to use point to point construction for this because of the requirement for mechanical soundness and because I value my sanity. This PCB also doubled as a way for me to connect the LCD touchscreen's I2C interface and power pins to the Raspberry Pi.

30 tactile microswitches Like I said, tactile feedback is important to me, so I'm using industrial microswitches for this keyboard.

10 diodes for each of the columns IN4007 MIC Through-hole diodes

Standard Male Header Pins w/ standoffs Connections for the Raspberry Pi and LCD.

Osoyoo "3.5 Inch" HDMI LCD with I2C Touchscreen This is from one of those shady reseller screens with no documentation I bought off of amazon. It was originally meant to be a break-out board for another model of the Raspberry PI, but the actual screen itself just accepts an HDMI input. I chose this screen because the HDMI input meant I didn't have to mess with drivers to get it to work, so it didn't matter that it it came with bad documentation.

Standard Female Header Pins Since this screen was mean for another model of Pi, and I needed to have it in a position far away from the Pi, I needed to connector between it and the Pi's GPIO. This only connects the power pins and I2C so it wasn't that much of a deal.

Commercial 4 foot HDMI cable No Way i'm going to make my own slim HDMI cable. The smallest HDMI cable that I could by was 4-foot. But that's a feature not a bug! If I keep the HDMI cable external and spool it up in the back, It means I can easily connect the Omnicom to an external screen.

Raspberry Pi Zero Now a days there are lot of commercial singleboard computers out there. What I had lying around and what I knew could run the OS I wanted however was a Raspberry Pi Zero. It's definitely underpowered for SDR signal processing, and the blockchain operations I'm doing for my distributed network, but for now it'll do before I move on to a more powerful platform.

2 USB-micro to USB type A adapters For Robustness's sake.

Custom made "slim" USB-Male to USB-Male Cable Standard USB connectors just didn't have that extra centimeter of slimness I wanted, anded weren't the correct length, so I made my own USB cables by canibalizing some USB junk I had around.

Custom made "slim" USB-Male to USB-Female Cable I had to make two of these, one to connect the Pi to the power supply and one to connect it to the SDR.

Commercial Power Brick of unknown manufacture I didn't want to deal with Lithium battery charging, So i just used a commercial USB power-brick I got as a promotional handout at the security conference. I wasn't terribly concerned with battery-life or the amount of current it could supply. It should be enough for the Pi, SDR and Screen, and If i want to give more power to the SDR, there's also an 9v power jack on that.

LimeSDR Software Defined Radio The part that really matters. The LimeSDR is a full duplex SDR with a decent frequency range that came out last year, and it's meant to be plugged into a laptop. It's also has a pretty decent software ecosystem around it, which is important because I'm not planning on writing too much custom code for this project.

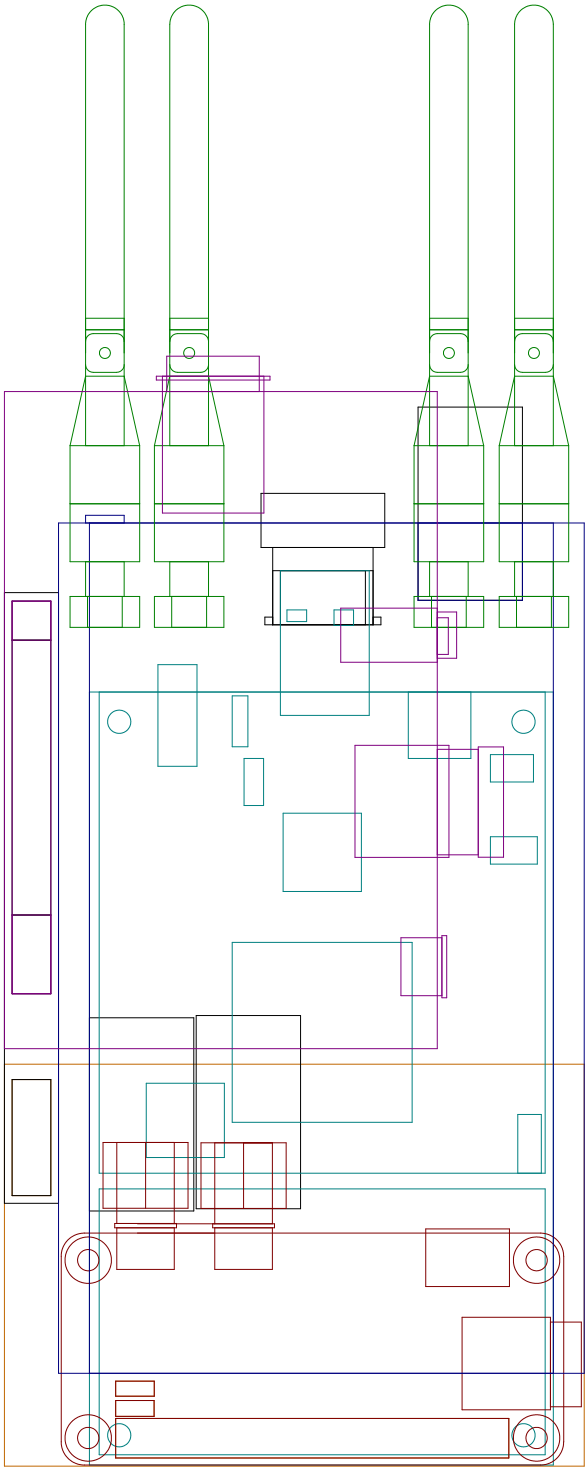
Custom frame for everything to be mounted on In order for the Omnicom to both be portable, and properly mount all these components, I have to use a pretty unconventional layout which requires a custom frame to mount everything in.

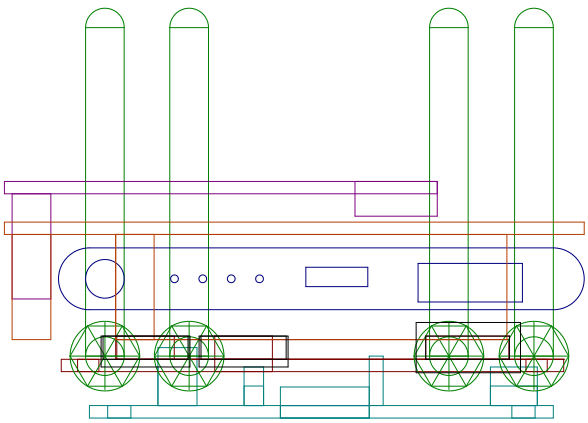
3 Internal Layout and Frame

3.1 Layout

So I have 3 PCBs, an LCD and a powerbrick that I need to put together in a way for me to fit in my hand, and also minimize the amound of wires I have to use. To make mounting easier, I had everything be in 4 "layers".

What I did was I had the SDR on the bottom since it was the largest component. Above the SDR I mounted the SDR antenna and room for the USB connectors going into the SDR and powerbrick, this formed the first layer.Then in the layer on top of that we have the Pi, which since it only took up part of that layer, left room for all the internal cables to come through. The next kayer was the powerbrick, but it was positioned slightly above the Pi's GPIO so that on that same layer I could connect te keyboard PCB to the Pi. The keyboard PCB and the screen then completely covered the Powerbrick, with female headerpins at the side connecting them. With this layout I could basically have all the ports of the SDR and Powerbrick at the top, accesible as is, and also have room for the USB cable connectors. I cadded the layout in `LibreCad` and you can see it on the next page.

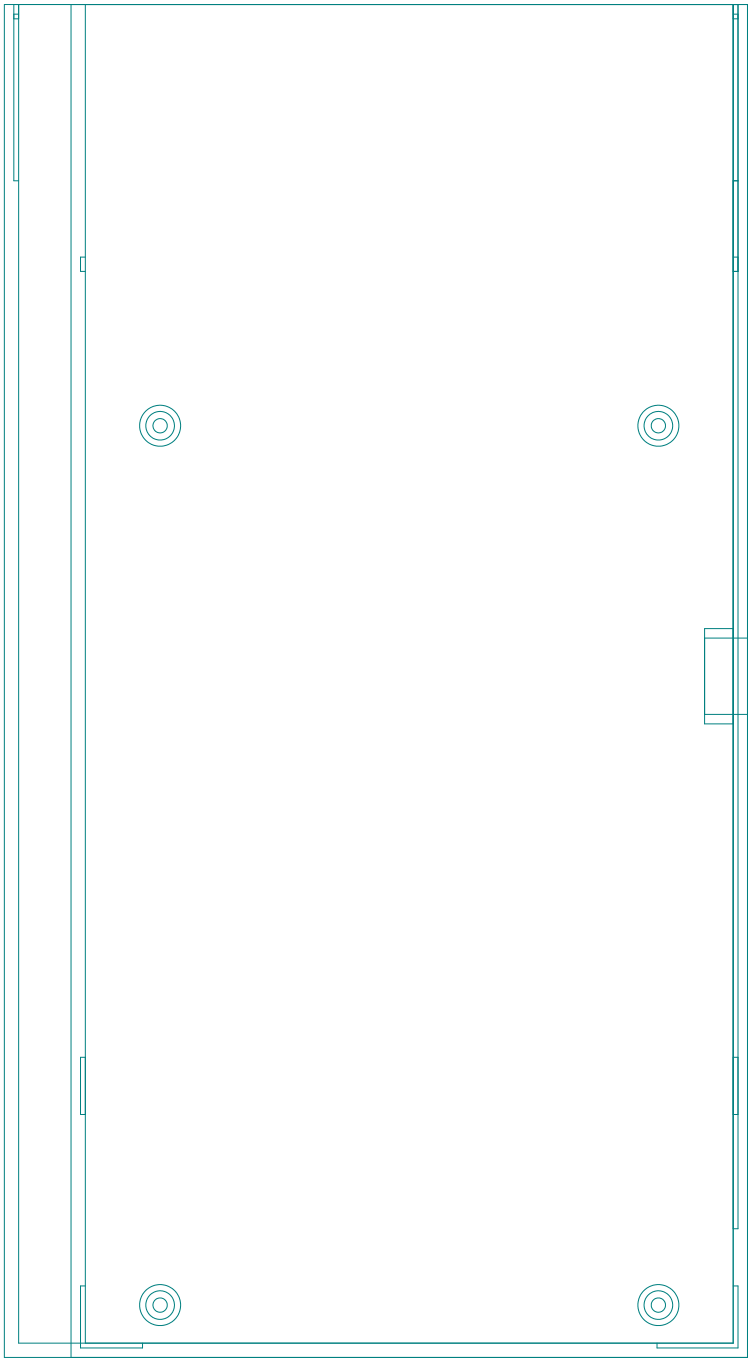


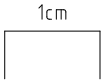


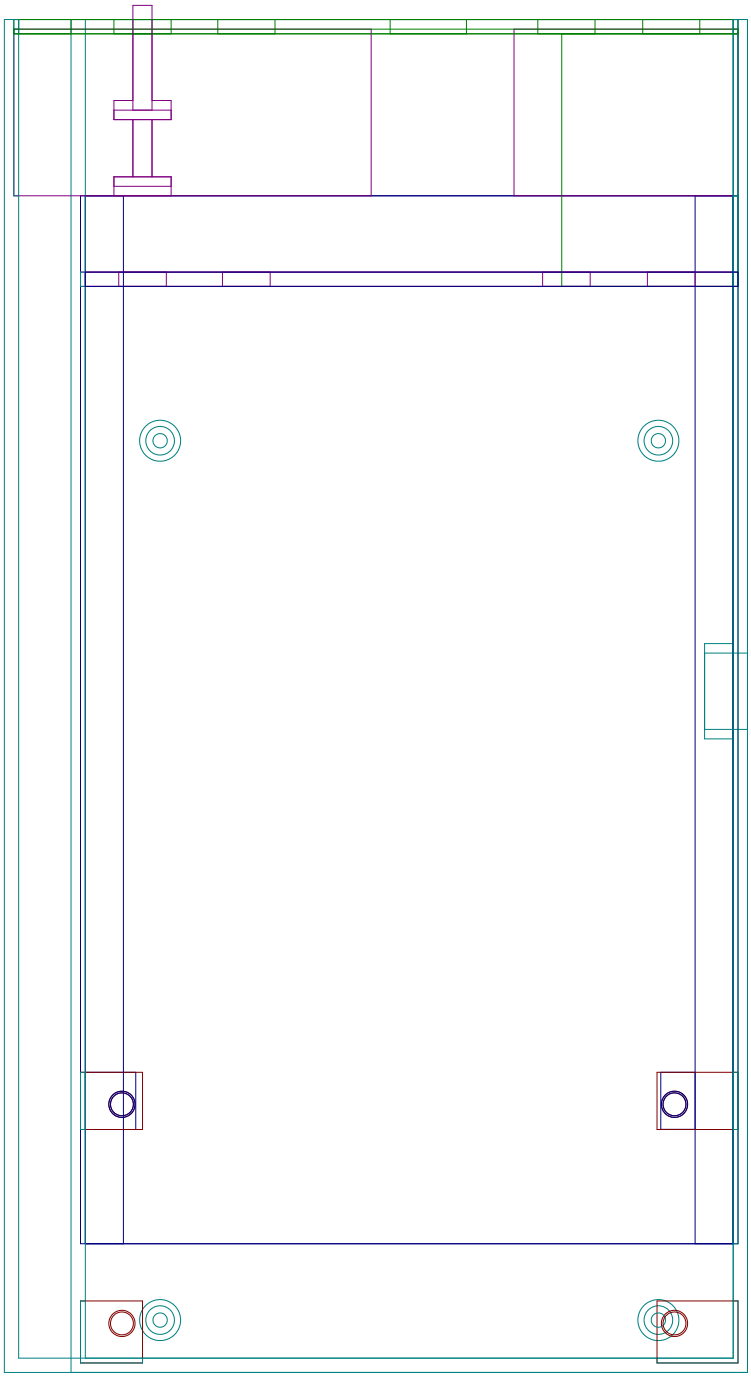
1cm

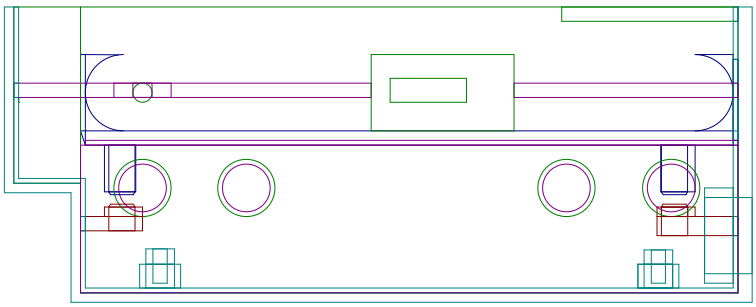
3.2 Frame

With the layout cadded out I set out to design some fancy frames to hold everything in. What I did is I had an outer frame that enveloped everything, and internal frames slotted into that that held all the components in place. You can see the design for that below:




1cm

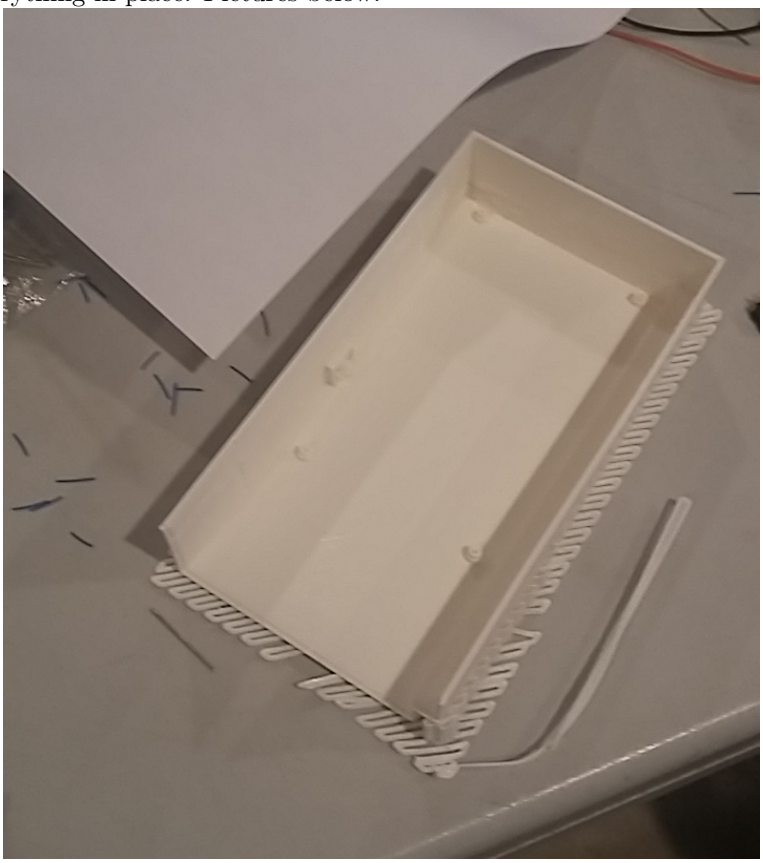


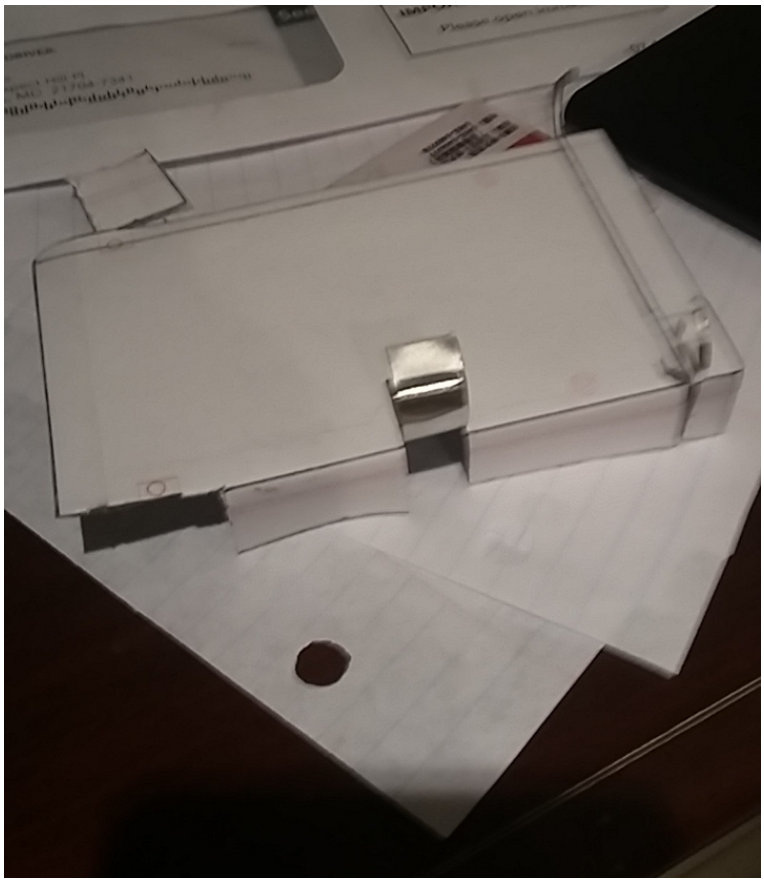


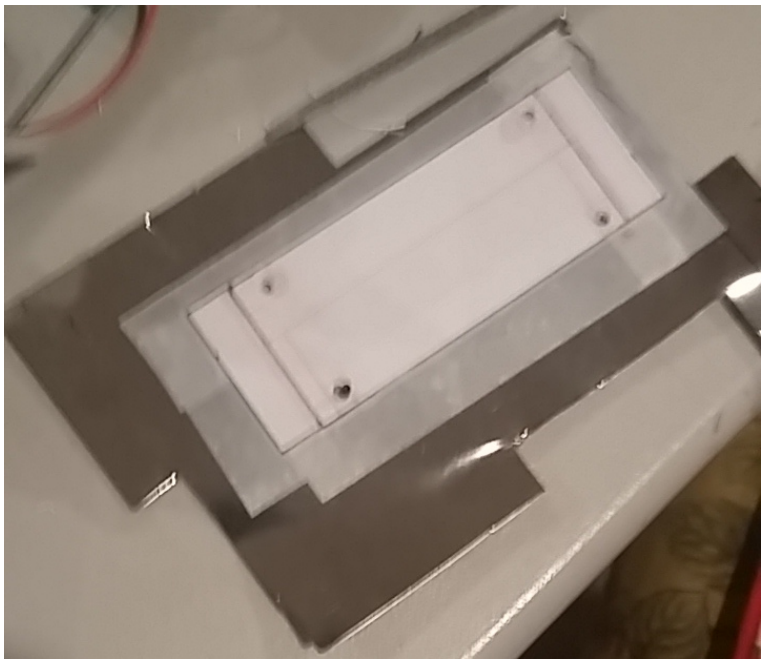
1cm

3.3 Frame Construction

I was able to model and print out the external frame, but realized that since I was using a public printer it was unfeasible to print out the internal frames. I opted instead to make them out of tin, using the above designs as templates. The slots in the external frame didn't turn out that well, so I ended up holding the tin plates in with friction. The plates are insulated and so are the mounting holes for all the components so I should be fine. I should ground everything in the future just for safety though.I have yet to create a faceplate to secure everything in place. Pictures below.







4 Internal USB Cables

The connectors on a standard USB cable have this long length of rigid plastic that takes up a lot of space. If you actually open one up you will see that it's mostly plastic there to serve as stress relief. To free up space, I made some custom USB cables that were the exact size I needed them to be, and had really small connectors that didn't take up much space. What I did was I scrapped some USB male and female headers from some junk electronics I had. USB has 4 wires and shielding. I soldered 4 wire-wrap wires to the connectors, then potted the exposed wire with hot glue(Resin is Resin) so that they don't cross over and short. I then used scotch-tape to secure the unexposed wires that made up the length of the cable. I wrapped aluminum foil around that as shielding, making sure to have a wire ground it to the metal parts of the USB connectors. Finally I insulated the entire cable with scotch-tape again. Pictures shown below:

The Powerbrick cable was slightly different in that I used a thicker gauge wire, and a switch so that I could hard shutdown the Omnicom without unplugging

anything. 

5 The Custom "Switch Keyboard

First I had drew out a schematic for the keyboard in KiCAD, shown below. It's a really standard 3x10 matrix design. The only thing special about it is that I used the same PCB to reroute the Raspberry Pi's GPIO to the hdmi touchscreen, so that I could potentially use the touchscreen, and didn't have to run a seperate cable for power.

5.1 Alas, N-Roll Keyboard!

Keyboards have this property called "roll", which is the number of keys it can detect for sure to be pressed down with one scan. In order to have "n-roll", or as many keys as there are on the keyboard, every key has to have a diode next to it. I didn't have room for that because I was using through-hole diodes so I settled on having 10 diodes for the columns. This doesn't have to do

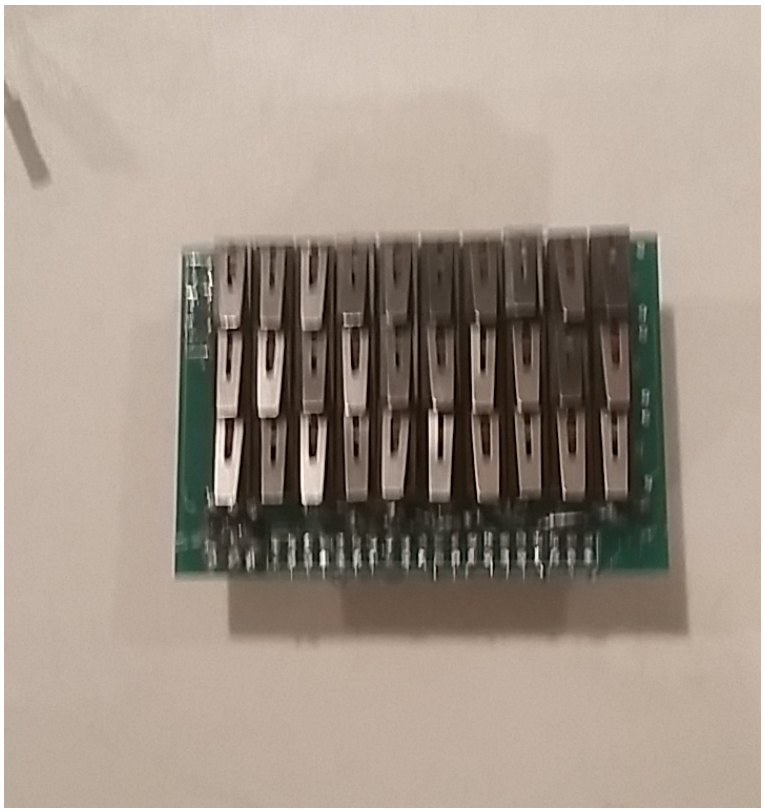
with roll but to protect against shorts. since the Raspberry Pi doesn't have tristate GPIO outputs, when you read a column, one column is set high and the others low. Without a diode there, if two switches on the same row are pressed, the high column pin could be shorted to one of the low column pins. These GPIOs are usually current limited so it's not a super serious issue if i don't have these. There are external pull-downs for the row pins as well, even though the Raspberry Pi has internal pull-up/pull-downs.

5.2 Custom PCB

I designed a PCB also in KiCAD and ordered it from PCBWay. Once I got it back I realized that I had made quite a few errors. The holes for the headerpins that connected to the Raspberry Pi GPIO were flipped, which meant I had to use wirewrap wires to connect the headerpin to the PCB using point to point soldering. I put in some supports for the exposed bit of wire and potted the entire mess which made it reasonably mechanically sound, but it's still a jank construction. I also realized that the holes for the dioles were too small, and the small pad around them meant I couldn't expand it. I ended up soldering the ends of the diodes to just one side of the pads without having them go through the hole. This is obviously not optimal, but alteast everything is staked.

5.3 PCB assembly problems

Then came the real folly, while testing I realized that I had soldered everything backwards, and because of my questionable soldering above it would've been a pain to fix it.I chose to just reverse the row and column pins so that the row pins were outputs and the column pins were inputs. This meant that I didn't have the protection of the diodes for shorts, and that I had to rely on the Raspberry Pi's internal pulldowns. I couldn't just make the keyboard active low either because the Raspberry Pi GPIO outputs weren't that good at sinking current.



5.4 Keyboard Layout

Obviously 30 keys is way less than a full keyboard, so what I did is i set aside 3 keys to switch the keyboard to different "modes", for all the keys on a standard keyboard, plus control a mouse. I also had a key set aside for sticky keys since you obviously couldn't do key-combos otherwise.

6 Some Configuration

As mentioned before, the Omnicom runs Linux, specifically the ArchLinux distro. I had already installed an Arch system on the Pi Zero I had so I'll spare the details. The main two things in software I had to work with were: configuring drivers for the SDR and LCD screen, and writing a kernel module to drive the keyboard. The LCD screen doesn't need a driver for the actual screen part, other than some settings in the Raspberry Pi's boot config that needed to be changed. However, I looked at the drivers for the touchscreen and found that they were proprietary. The touchscreen uses a really simple I2C interface so I'll just write my own later. I haven't gotten to the point where I have started using the LimeSDR, but I do know that everything is connected right and that it powers on as expected. LimeSDR has really good software support that's opensource, so I should be able to demo it soon.

7 Writing a Linux Kernel Module for the Keyboard

Obviously given that I built a custom keyboard using GPIOs I'm going to have to write a kernel module to drive it. After some looking I saw that I only needed to interface with two kernelspace APIs, the Linux Input Subsystem, and the the Linux GPIO interface. There are multiple versions of the Linux GPIO Interface, but I chose the legacy one because it was the simplest. Dumb decision I know, but I feel like that support's not going to dropped anytime soon. The code is below. There are obviously still some issues that I will describe below. The source code is too long to put in this document, but if you do want to take a look go to https://github.com/knolax/skey. I used git to transfer code from my laptop to the omnicom, so most of my commits weren't tested, and you'll see my weird use of commit messages as test logs, but digress. I have a thread that scans the keyboard, `skey_update_thread()`, which then reports the state of the keys to `processkey()`, which handles rising falling edge detection, the 3 keyboard mode I describe above, and stickykeys. `processkey()` then emmits the approrpiate hid events to the input subsystem. The keycode map is hard-coded in a header file also listed.

7.1 Issues

One issue I ran into was that the Linux GPIO interface, both new and old, did not have a way to set pin pullups/pulldowns. The "conventional" way to do this was through a device tree overlay, which declares system resources on a Raspberry Pi. The issue with this is that the Raspberry Pi's dtoverlay system was really sparsely documented and unreliable. In addition testing was a pain since the kernel I was running didn't support dynamic dtoverlay loading, so I had to reboot the Omnicom everytime. After a while I decided to just write a systemd unit file that used the `gpio` utility from WiringPi to set the pin pullups/pulldowns. This wasn't optimal because it added a userspace dependency to my driver. I did check how the `gpio` utility set pullups/pulldowns and found that it was using some really low level memory mapped controls. I wanted to stick to the "proper" kernel interfaces so I didn't try to replicate it in my driver.

8 The Future

Right now the Omnicom works well enough that I can develop code for the Omnicom on the Omnicom. I've attached a video of me playing tetris with the custom keyboard just for fun. Obviously I haven't set up the actual software that connects to the SDR yet so that's the next step. Right now I'm thinking of some cool demos I could do:

GSM cell tower spoofing

Ad-Hoc Wifi Network

Receiving Downlinks from the Iridium Network