Member-only story

Insulin Pumps, Decapped chips and Software Defined Radios

Pete Schwamb · Follow

17 min read · Apr 24, 2019

673 5

Reverse Engineering an Insulin Pump for DIY Closed Loop Therapy

Roughly 3 years ago, I heard about a website offering a bounty for something that was very close to my heart: reverse engineering communications to an insulin pump. My daughter was already using a system that I had helped to create called Loop, with a Medtronic pump that I had reverse engineered the RF comms for[1]. But the Medtronic pump required her to disconnect during her gymnastics for hours at a time. The tubeless design of this Omnipod pump sounded great, and I had all the tools to start working on the problem.

The Omnipod system consists of a small disposable pump called a pod, and a controlling unit called a PDM.

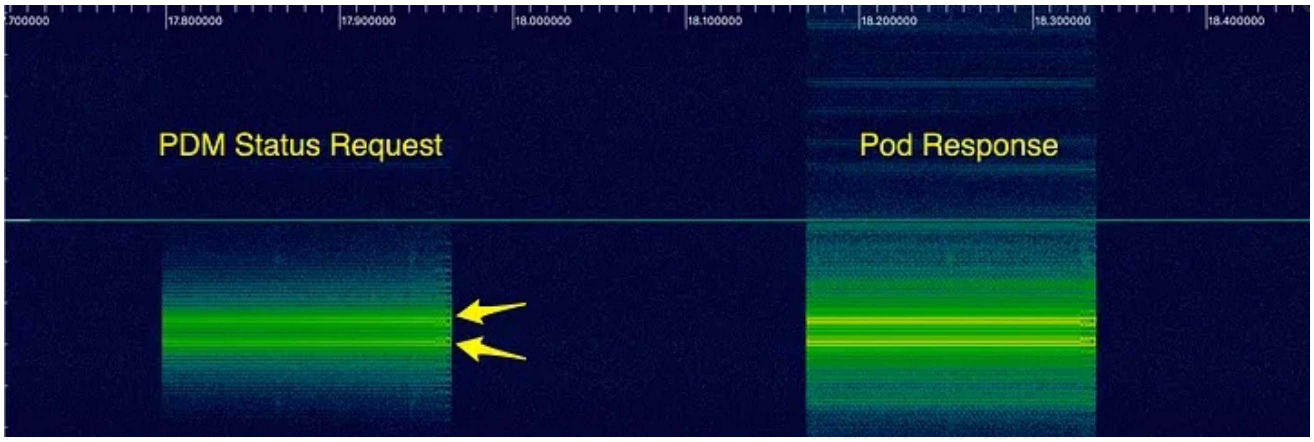
Because the PDM communicates with the pod using radio and the pod has no built in interface, it means the pod is entirely controllable over the radio. There was the potential to create a full integration with Loop using just a RileyLink, or a modified version of it.

James Wedding had put up the bounty, and it attracted a lot of attention, and ultimately the right people who would be needed to make progress.

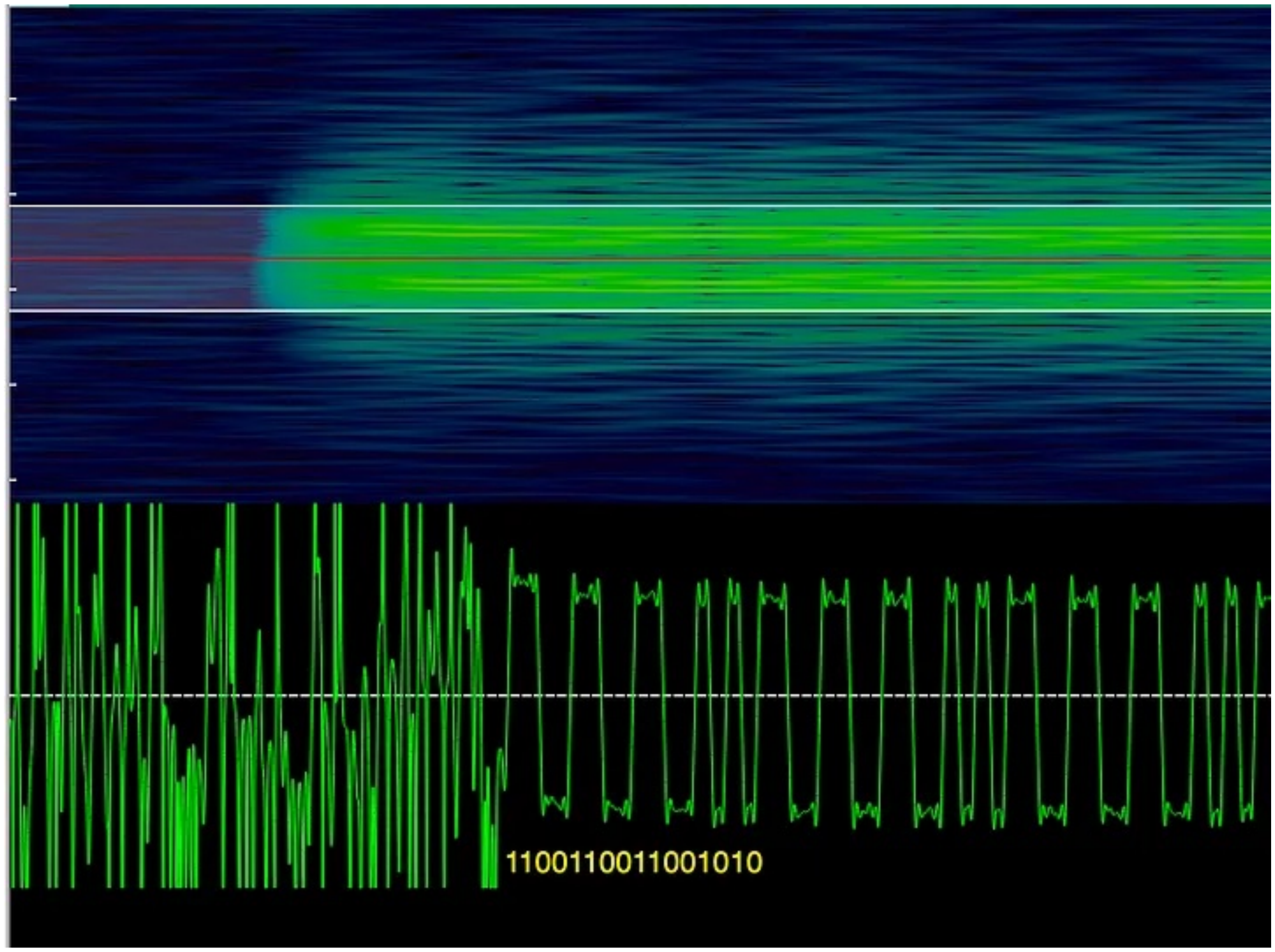
Software Defined Radio

SDRs are awesome tools; they make the hidden world of radio visible. There are all kinds of messages zipping by all the time, and these tools let you poke around, see the messages, and with some work, start decoding the little blips you see there. If you're looking for messages from a specific device, you need to know what general area to start looking in. That's where the FCC public filings come in handy.

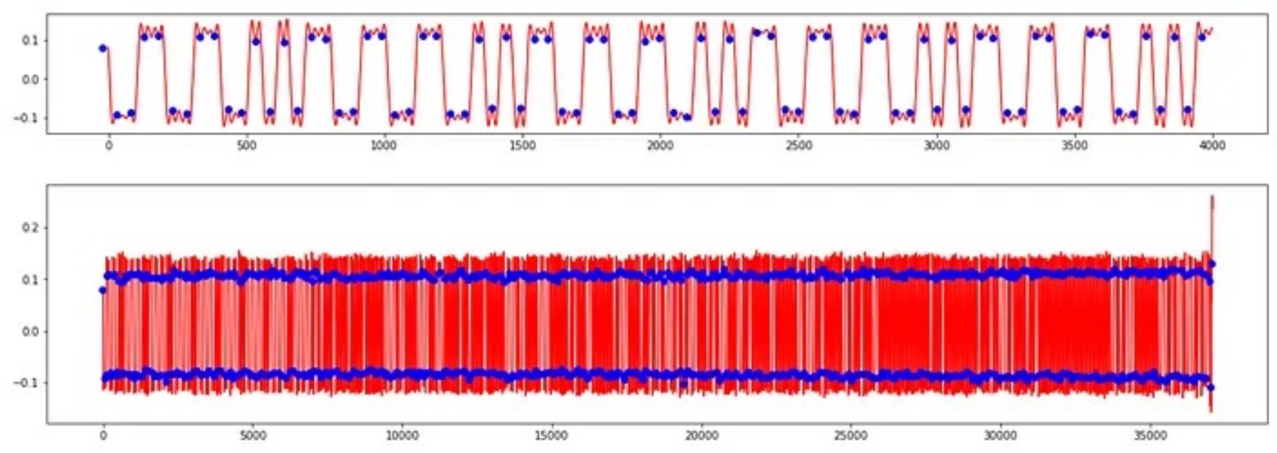
The FCC filing for the PDM, RBV-019, shows that the device transmits in the 433MHz range. Configuring the SDR software to listen in the 433MHz range while issuing a status from the PDM shows comms like this:



As I eventually learned, those two bright lines are indicative of a certain type of modulation called frequency shift keying, or FSK. This means that the frequency of the signal varies with the information being transmitted. A 1 bit is sent as a higher frequency (the top line) and a 0 bit is sent with a slightly lower frequency (the bottom line). And using the tool inspectrum we can have it analyze the data to more clearly show the 1’s and 0’s. Here’s a very zoomed in view of that first message above:



I ended up writing a python script to extract these bits so we could look at them as a whole packet.



Get bits for entire packet

```
(centers, bits) = resample(packet_samples, bit_center_offset, samples_per_bit)
bits = (bits > 0).astype(int)
print "".join(map(str, bits[:80]))
```

```
10011001100101011001100110010101100110011001010110011001100101011001100110010101
```

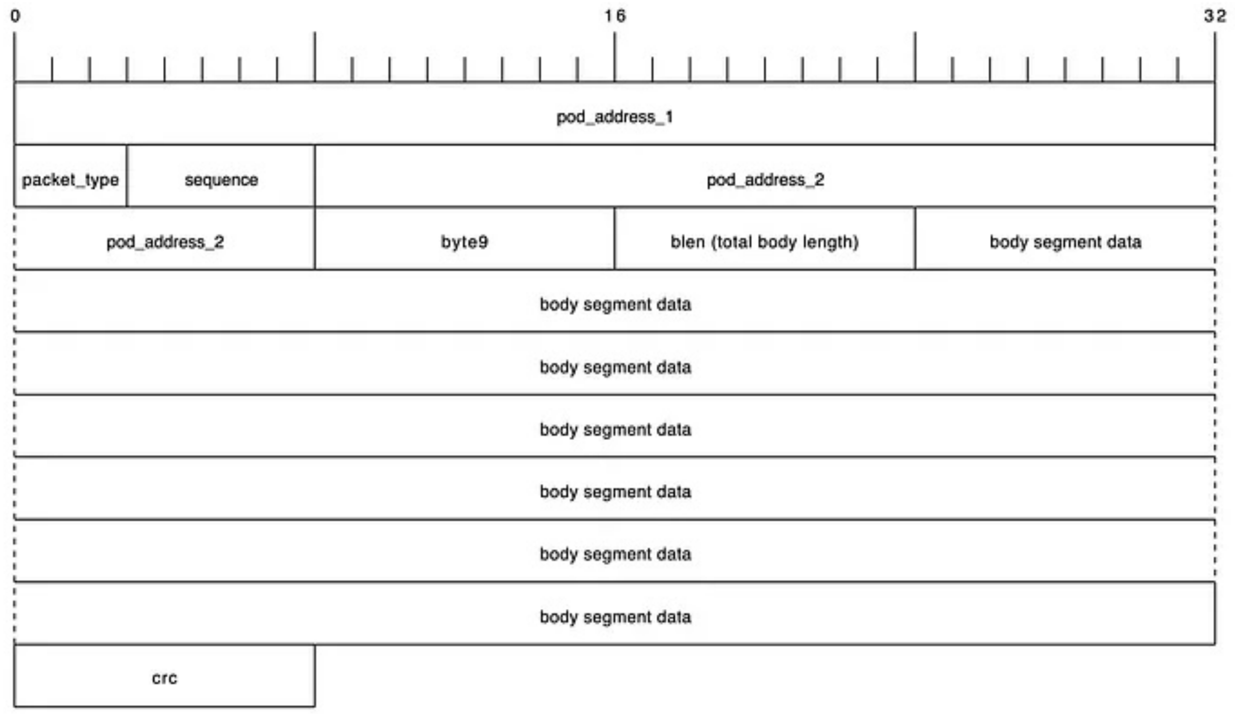
It turns out this repeating pattern was part of the preamble. To conserve energy, receivers often have a pattern of sleeping, and waking up periodically to check for a signal. The transmitter sends a preamble long enough to catch the receiver during one of the short listening periods. When the receiver hears the preamble, it stays awake until the real data shows up.

There was one more layer we had to get through before we were looking at actual packet data. You can't send your data over the radio exactly as the original bits, because the receiver uses the transitions to synchronize in time when to expect the next bit. If you have a long set of 1's or 0's, the receiver could get out of sync. So radio communications usually use an encoding to make sure there are enough transitions. The Omnipod communications use an encoding called Manchester encoding. Each bit is encoded by two bits. A 1 bit is encoded as 10, and a 0 bit is encoded as 01.

This all took a lot of hashing out and there were a lot of theories shared in the openomni slack, as we tried to make getting the raw bits repeatable. Mark Brighton, Dan Caron, and @larsonlr, had been having some success with using RFCat and a TI Stick to capture packets. Evariste Courjaud would eventually write a tool called rtlomni used an rtl-sdr stick to listen for packets and decode them that would prove to be very convenient and more reliable than the TI-Stick based methods.

Decoding Packets

Once we had the actual bits of the packet, we started to look into packet structure. Based on what bits were changing between different pods, and different commands, we pieced together an understanding that looked like this.



CRC8

Radio is a far from perfect transmission medium. There are many different sources of interference that can make the receiver hear a 1 when a 0 was sent, and vice versa. It's important to know when this has happened, so most protocols use a checksum of sorts, often something called a CRC. The receiver computes the CRC as it receives the data, and the last byte of the packet includes the CRC as the transmitter computed it. If they don't match, the receiver throws the packet away and waits for a retransmission.

The Omnipod protocol used a standard 8-bit CRC, so when we found this, we thought we were really close to having the messages figured out. Little did we know…

Messages, CRC16

Some messages are too big to fit into a single packet, so they are sent as multiple packets. We started to piece together the format of messages, and noticed another set of bits at the end of each message that looked like a 16-bit crc. But it was weird; 5 of the 16 bits were never set. We tried many different techniques to figure out how this was being set, but nothing worked.

This was the first big blocker; we could proceed to work out what the other bits in the messages were, but it would be little help to understand what was being sent, but not be able to generate new packets ourselves, and so progress slowed down.

Months passed with little progress, and finally in winter of 2016 a member of the group with the handle @lorelai posted that she had sucessfully dumped the firmware from the larger ARM-based chip on the PDM, and started the tedious process of disassembly: taking cpu instructions and turning them into human understandable code with semantic variable and function names. She did an amazing job figuring out what various methods were involved in sending data over the radio.

I was looking at one of the unnamed routines, and noticed it looked like a standard implementation of a table based CRC calculation. And the table had the values for a standard 16 bit CRC. I wrote my own implementation using the table, and it checked out like a normal CRC. Then I looked closely at how the function was written. A normal CRC implementation looks like this:

```
while (len--) {
  crc = (crc << 8) ^ crctable[((crc >> 8) ^ *c++)];
}
```

Theirs looked like this:

```
while (len--) {
  crc = (crc >> 8) ^ crctable[((crc >> 8) ^ *c++)];
}
```

Spot the difference? What should've been a bitwise left shift operator had somehow been coded as a right shift. This is a bug; there is no reason to cripple your own CRC algorithm, as it makes it less likely to catch corrupted messages.

We were up and running again! We started working on decoding messages, recording sessions from the PDM for delivering boluses, temp basals, suspends, etc…

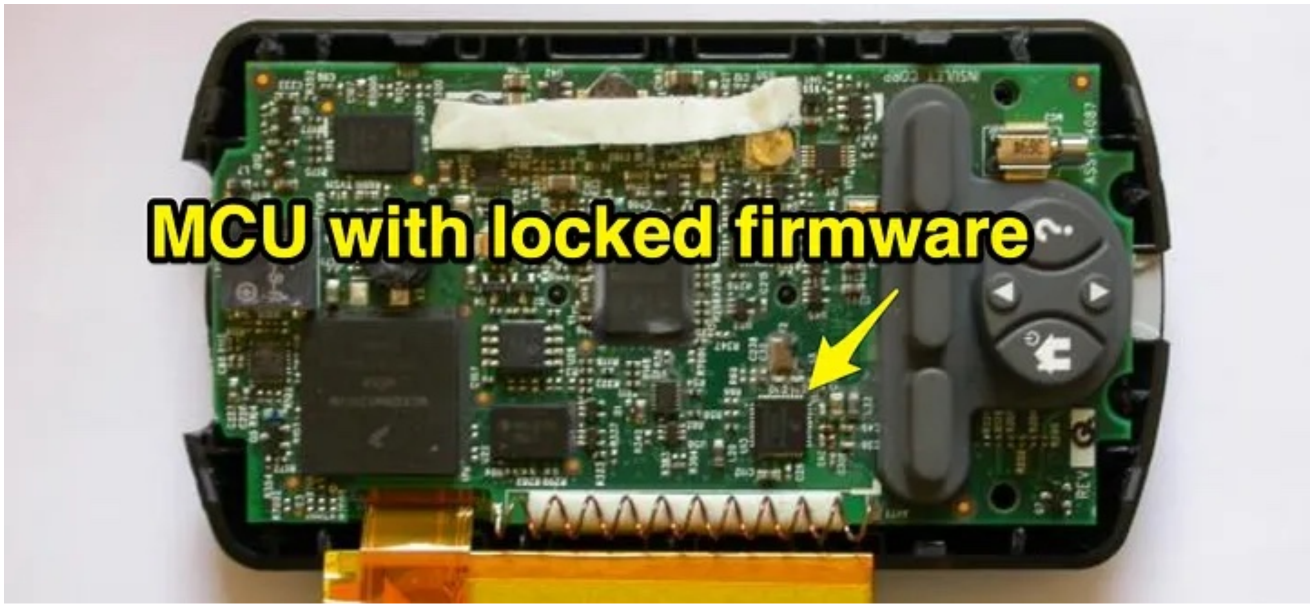
Nonce

All of the insulin delivery commands had a 4-byte chunk of data near the beginning of the message that looked like it might be some form of cryptography. Again, we tried many different ways of interpreting it, and analyzing it in the context of the messages it was sent in, but it wasn't a crc (we'd see the same 4 bytes on occasion even when the message data was different). And sometimes we'd see the pattern repeat. It looked maybe this was a bit of the protocol designed to prevent replay of data. Other protocols had features like this, called a nonce.

One possible route we had considered was to record a database of messages to replay for given commands. Even if the address of each pod was different, now that we knew how to generate the message crc, we could take a copy of the command we had seen before, put a new address on the message, and recompute the crc. Except this nonce was preventing us from using that strategy. For the next insulin delivery command, no matter what command was sent, the pod would only accept the next nonce in the sequence, and we didn't know how to generate the next nonce.

But hey! We have the decompiled PDM firmware now, we can just look there! So we pored over the PDM firmware, and tracked down the message generation in the code, and found where those four bytes should be. But instead of a method computing some cryptographic Nonce, we just found the four characters “INS.”. WTF?!?! Ok, somehow this area of the message must be being updated later in the pipeline.

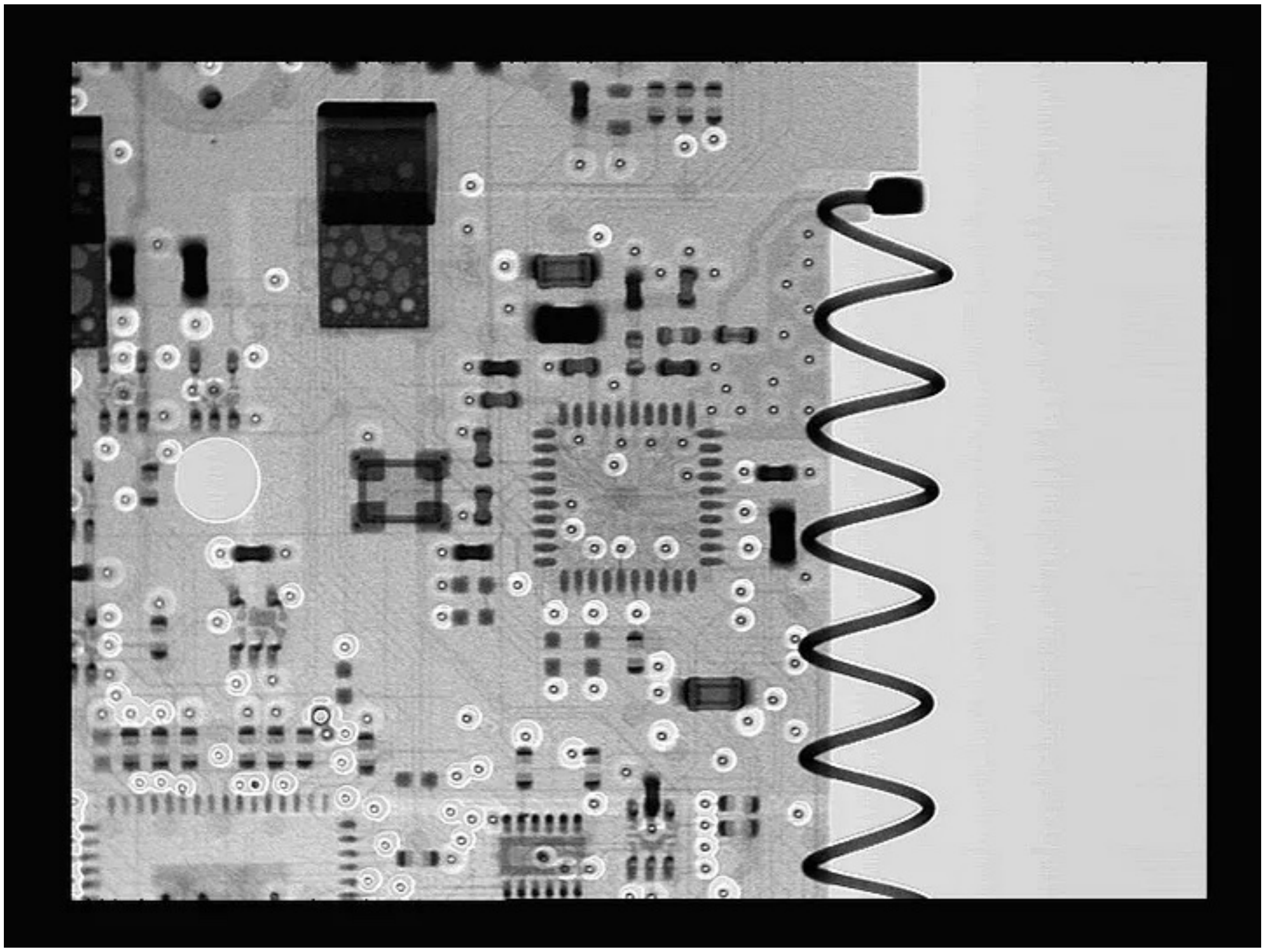
There was another chip on the PDM, closer to the radio. It was the same chip that was used in the Pods, a chip with an identifier of SC9S08ER48, something that wasn’t documented online, and was likely produced custom for Insulet. Maybe we could get the firmware off of that chip. Unfortunately, that chip was locked, which prevented dumping of firmware.



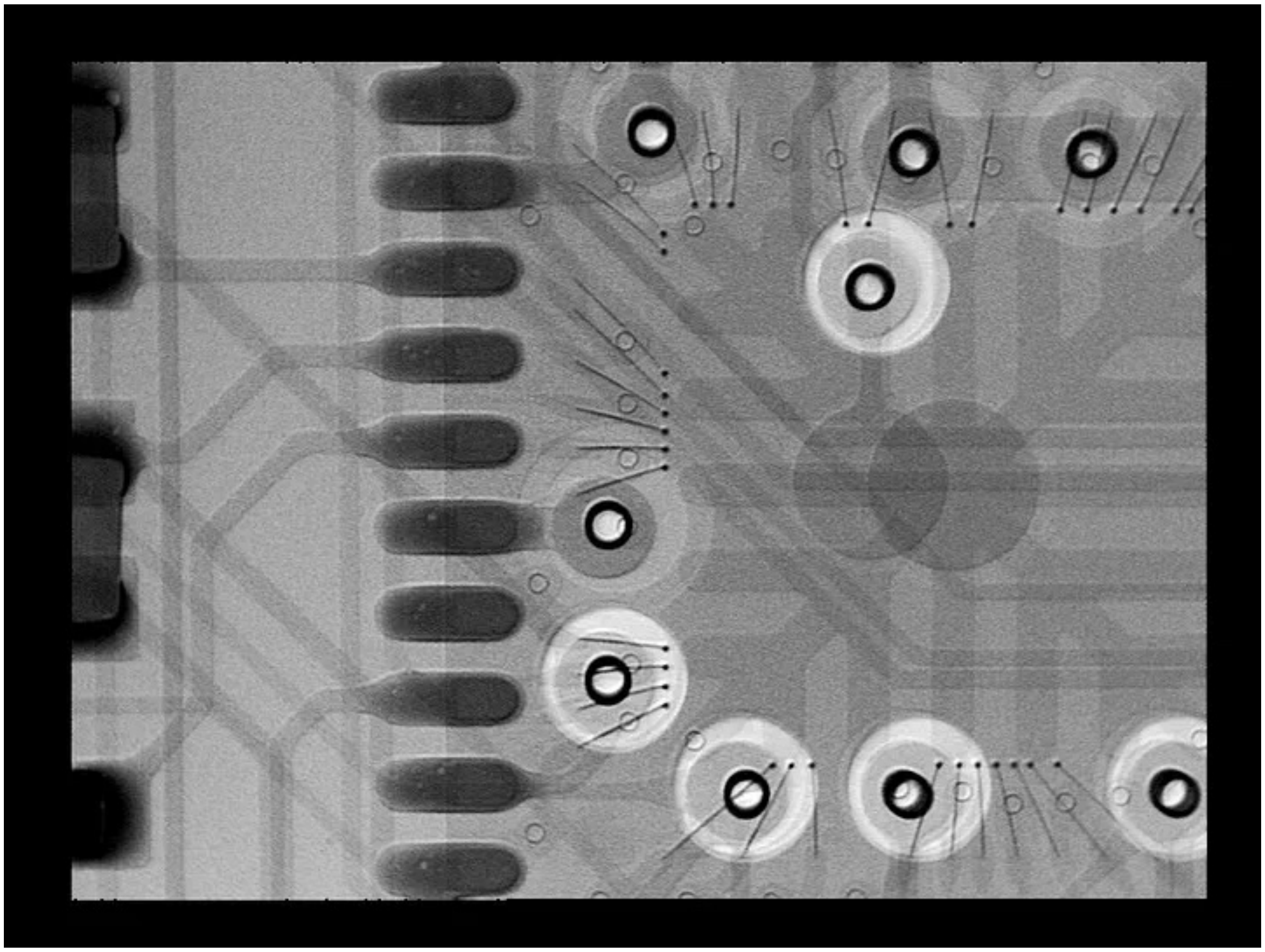
Again, progress slowed… This felt like a real dead end. We had thrown all our smarts at this nonce, and didn’t have any good leads on the math that was behind it. And the ER48 that (possibly) held the secrets was locked and it was hard to find any public details that might help us crack it.

X-Ray images

In trying to understand more about the ER48, some members of the slack community offered to try to get Xray images, which was really cool, but unfortunately didn’t open up any new avenues.



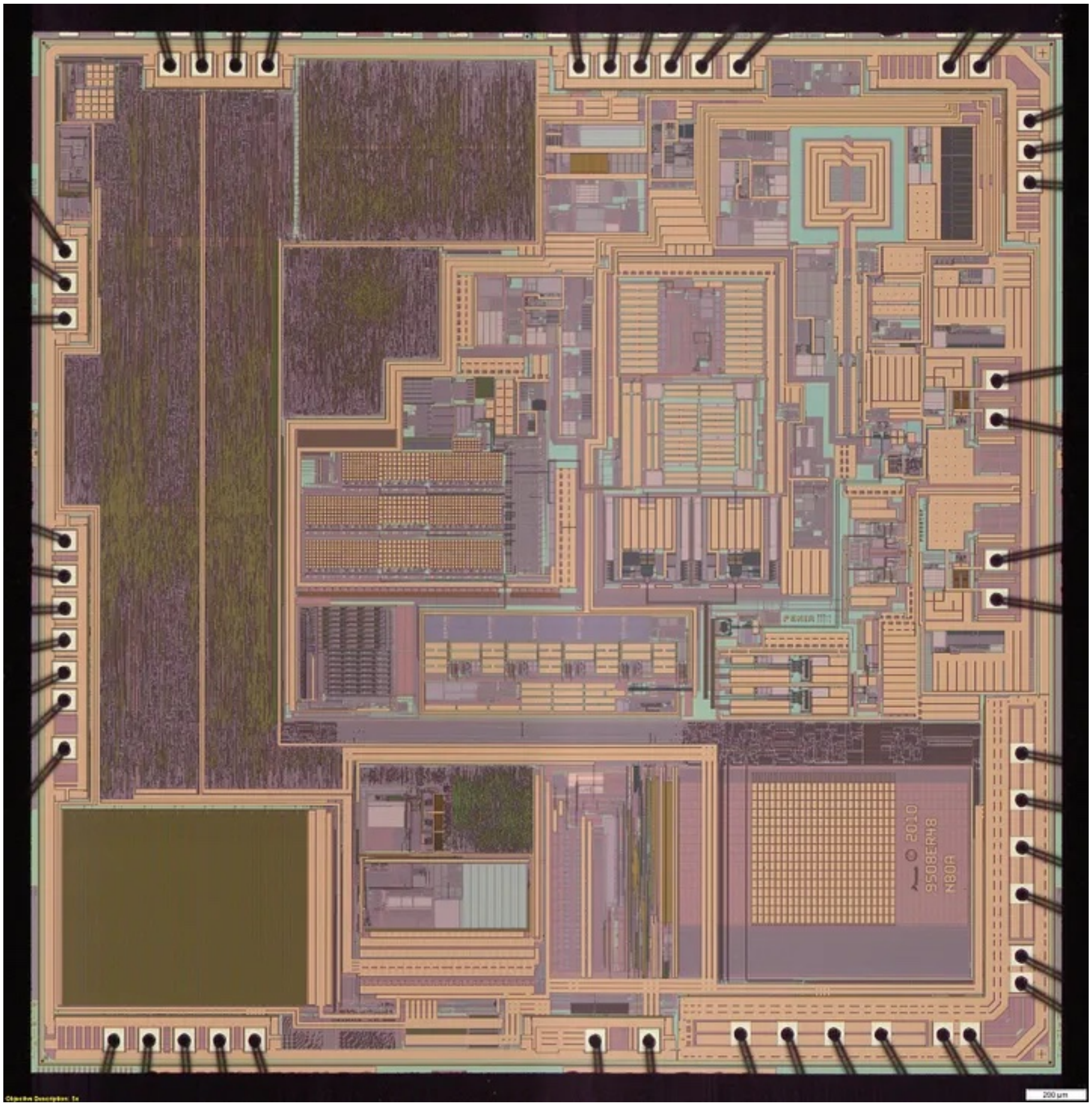
Overview Xray


Detail Xray

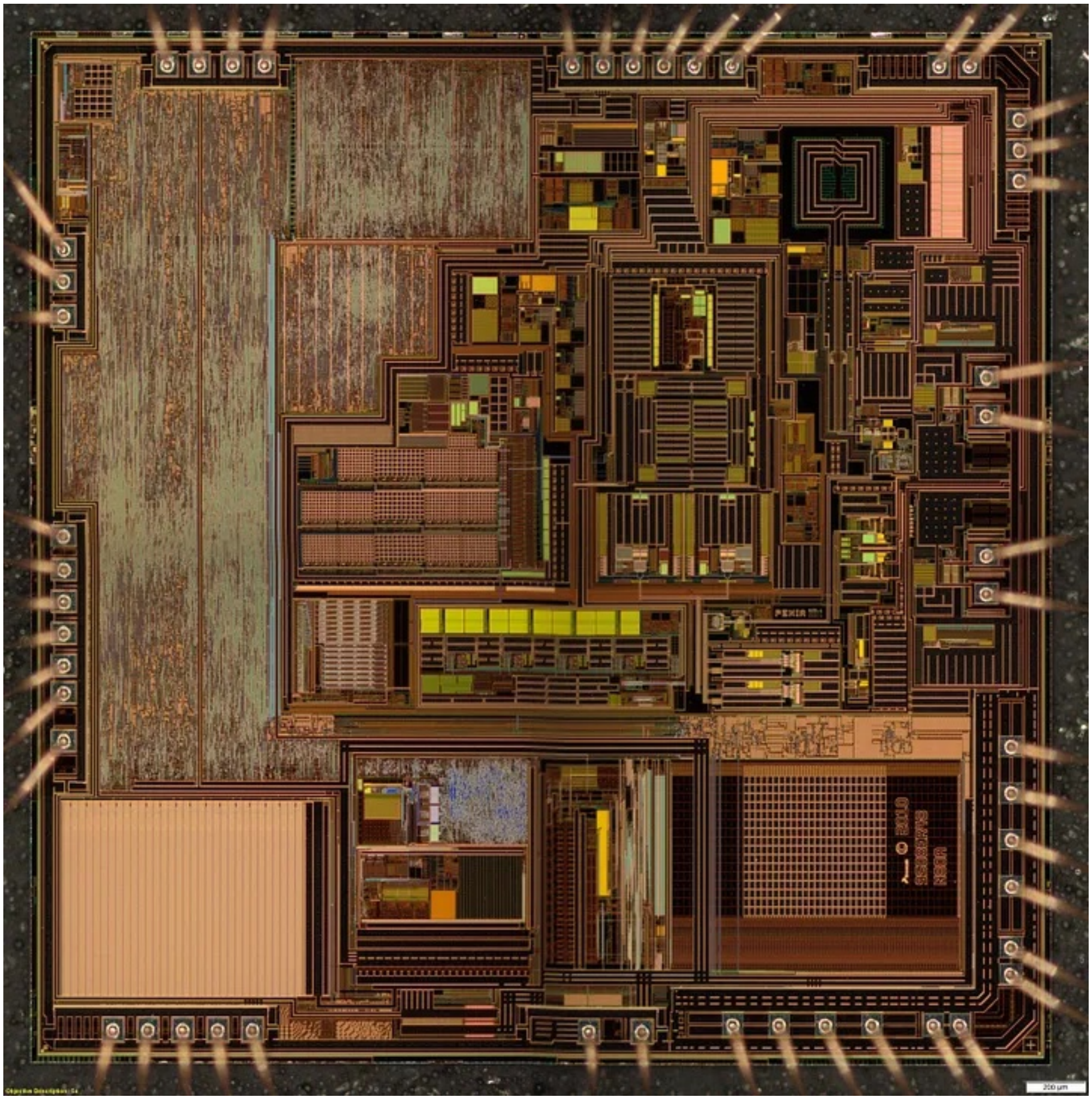
Decapping and Imaging

Dan Caron decided to reach out to a researcher, Dr. Sergei Skorobogatov, at the University of Cambridge in the UK that he had read about who had experience in extracting code from locked chips, and convinced him to take a look at our problem. Dr. Skorobogatov had led research in using SEM (Scanning Electron Microscopy) for chip reverse engineering and suggested it might be possible, but would be expensive, needing access to expensive equipment, and not guaranteed. Joe Moran, who had recently started using Loop after we met at the Fall 2016 Nightscout Hackathon dove in to helping out with this project, and arranged with a bay area company, Nanolab Technologies, to perform the decapping and imaging of the chips, and also graciously funded Nanolab and Dr Skorobogatov's work (and also his personal boxes of pods).

Dr. Skorobogatov had Nanolab perform a variety of imaging techniques to ascertain if the protection could be defeated with known non-invasive or semi-invasive methods. This resulted in a lot of images, some of them very beautiful. These are optical microscope pictures of the silicon die.

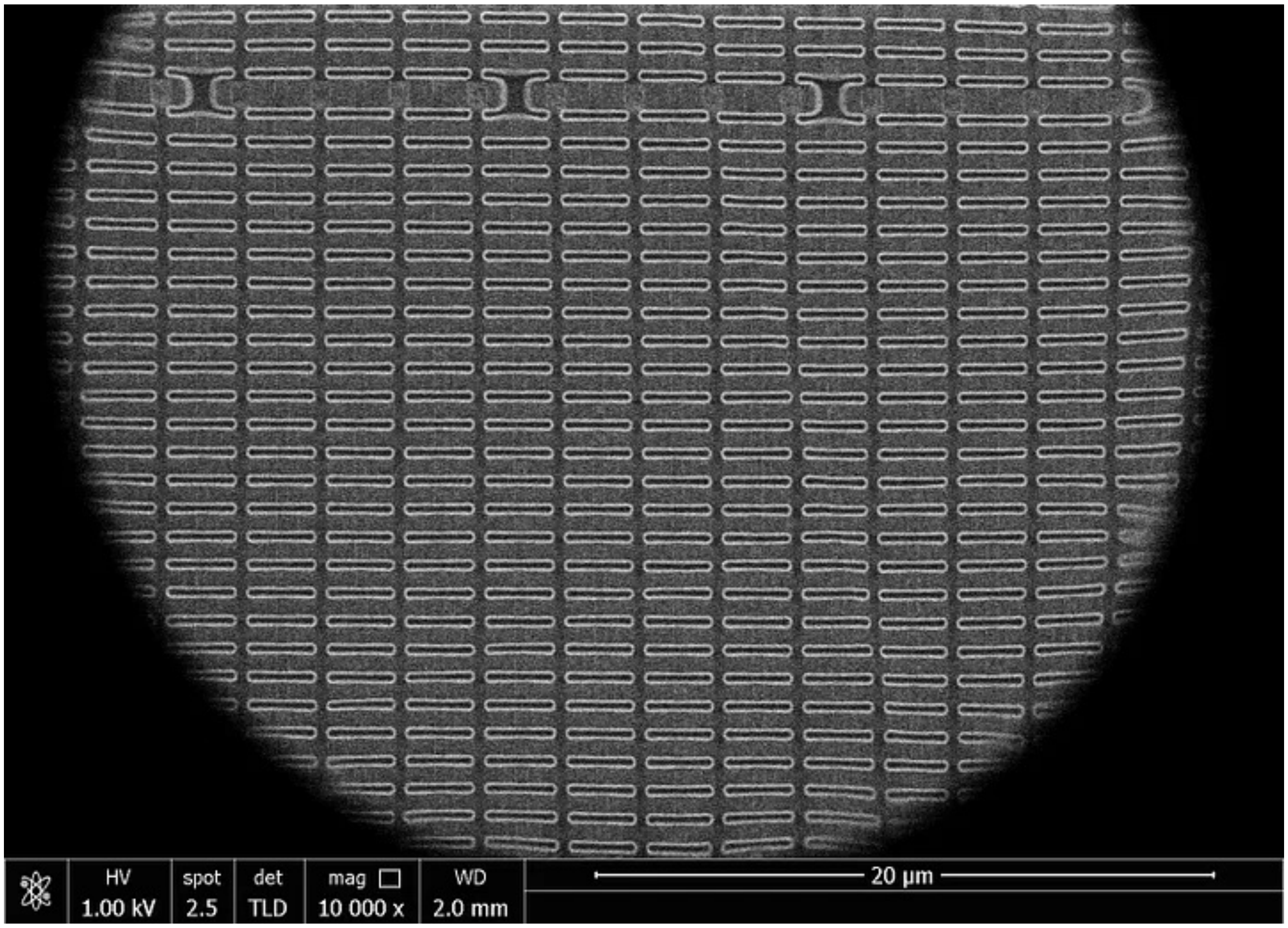


Optical microscope die overview



Optical microscope die overview

There were also images taken of specific areas of the die using scanning electron microscopy. Attempts were made with different voltages, different surface preparations, and different equipment.



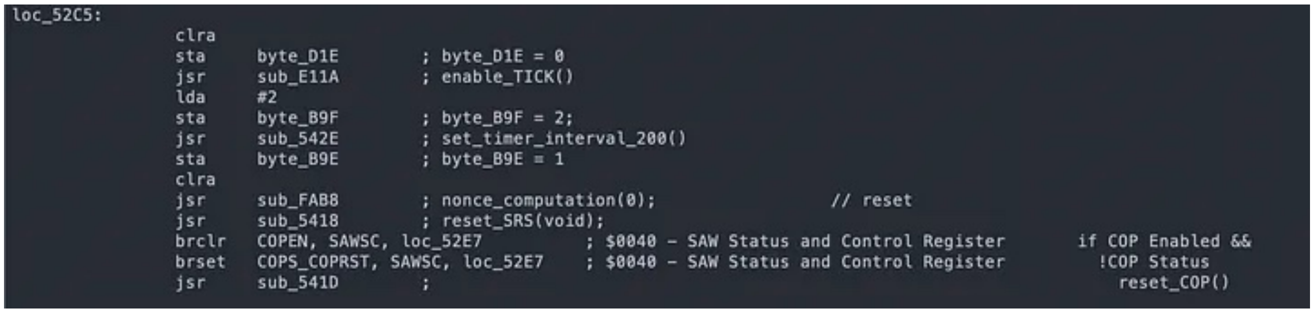
SEM Image of flash memory cells. Not resolving data.

Unfortunately, none of these produced images capable of resolving the actual contents of the flash.

Dr. Skorobogatov had one last resort method that was only to be used if these other techniques failed. It was a proprietary method that would have to be given the blessing to be used by the University. Dr. Skorobogatov did an initial test and confirmed that it was capable of resolving the data on this chip. But before we could proceed an NDA had to be agreed upon and so negotiations about who was to be the recipient of the extracted firmware were undertaken.

The Nightscout foundation would ultimately be the party signing the NDA and would take the responsibility to prevent unauthorized disclosure of the methods and results of the extraction.

The result of this arrangement and work was an incredible paper written by Dr Sergei Skorobogatov, and the raw firmware data. The initial extraction of the firmware had a significant number of errors, but was enough to get started on. Joe asked at the spring Nightscout Foundation Hack if anyone would be up for digging into the disassembly. Nobody raised their hands. Turning the cpu instructions into something understandable is painstaking work, and there are very few people who know how to do it. I tried digging into it, using cpu instruction documentation, but made very little progress, and grew frustrated. Others optimistically asked for request to the firmware with big expectations of fast progress, and then realized the scope and challenge of the task and quietly dropped off.



```
loc_52C5:
                clra
                sta     byte_D1E        ; byte_D1E = 0
                jsr     sub_E11A        ; enable_TICK()
                lda     #2
                sta     byte_B9F        ; byte_B9F = 2;
                jsr     sub_542E        ; set_timer_interval_200()
                sta     byte_B9E        ; byte_B9E = 1
                clra
                jsr     sub_FAB8        ; nonce_computation(0);                 // reset
                jsr     sub_5418        ; reset_SRS(void);
                brclr   COPEN, SAWSC, loc_52E7          ; $0040 - SAW Status and Control Register       if COP Enabled &&
                brset   COPS_COPRST, SAWSC, loc_52E7    ; $0040 - SAW Status and Control Register         !COP Status
                jsr     sub_541D        ;                                                                    reset_COP()
```

Example disassembly of SC908 instructions

It turns out Joe also has an extensive background working at the assembly level, and started taking on the onerous task himself. In July, Dr. Skorobogatov had completed a second extraction with much fewer errors. Over the summer Joe Moran worked tirelessly on mapping out the huge amount of cpu instructions, and slowly fitting them together into a larger picture of pod pseudocode.

Eventually, Ken Shirriff, an expert in hardware reverse engineering, would join the effort and sped up the process considerably. Together, Joe and Ken eventually mapped out enough code to find the function responsible for coding the nonce September of 2017.

RileyLink and Loop

We had been updating the openomni python scripts with our understanding of the comms, but now it was time to start focusing on RileyLink + iOS, so I started work on OmniKit, and firmware updates for the RileyLink. I believed we had the fundamentals of the protocol worked out, and the rest was just details. Again, completely underestimating how much more was ahead of us.

I had to write new firmware that would handle the pod modulation and encoding. I also had to rewrite the way the two chips on the RL talked to each other to handle 0's, as 0 was a special end of packet marker for Medtronic. Much in Loop needed to be reworked to support multiple Pumps, and new interfaces made to support pairing, deactivating, and handling faults. Thankfully Nate Racklyeft had set a solid foundation in Loop for this to happen.

Meanwhile, work on understanding the format of the commands continued and was documented thoroughly on the openomni wiki, which is the most comprehensive documentation on the pod protocol published publicly. Joe, Evariste, and Eelke Jager really did a massive amount of work decoding messages, and updating the pages over time. Various members of the slack channel have contributed captures of PDM to Pod communications to aid decoding efforts.

The decoding was fun work, with lots of small wins as each component of each command is deciphered, and I really enjoyed working on this part, and adding code to Loop to implement. In April of 2018 I shared on slack that I had “paired via iPhone + RL, primed, cannula insertion, basal schedule programmed, and then bolused 5U”.

The 2.0 RL firmware was finished in July of 2018, and new shipments started including it. It was hoped that these boards would be able to be used with Loop and Omnipod, but the existing 915 MHz antenna would prove to be too poor to do 433 MHz comms effectively.

Decoding and implementation was progressed well over the summer, and the Loop experience was coming together. Joe did an amazing thing in providing funding for me to quit my day job, and focus more on this project, and to eventually join the wonderful Tidepool team. There were of course more events happening outside of the omnipod RE story in the DIY and regulator spaces that I won’t cover, but it was a very interesting summer!

Screamers

As more functionality came online in the driver, I started hooking it up into Loop, including the ability to automatically adjust delivery by temp basals. At this point it was fairly common to get a screaming pod, which meant some of the pod’s internal checking had detected a situation where it decided there was a problem and it should not continue to deliver insulin.

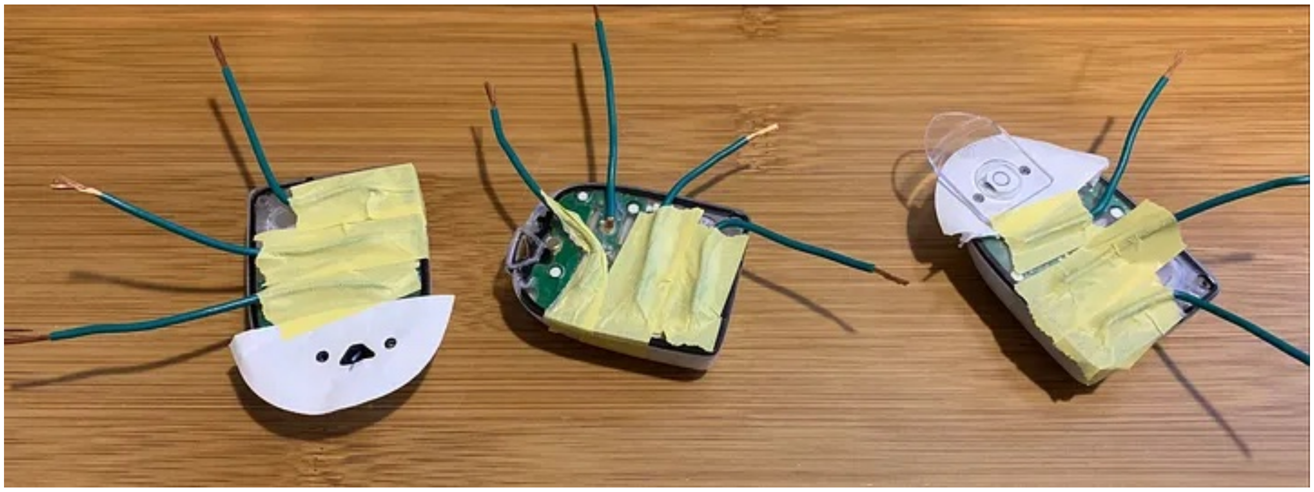
But it seemed like a workable problem, as we continued to find small discrepancies in what Loop was sending versus what the PDM would send if the command was delivered manually, and I assumed if we fixed all of those that the screaming would stop.

Looping!

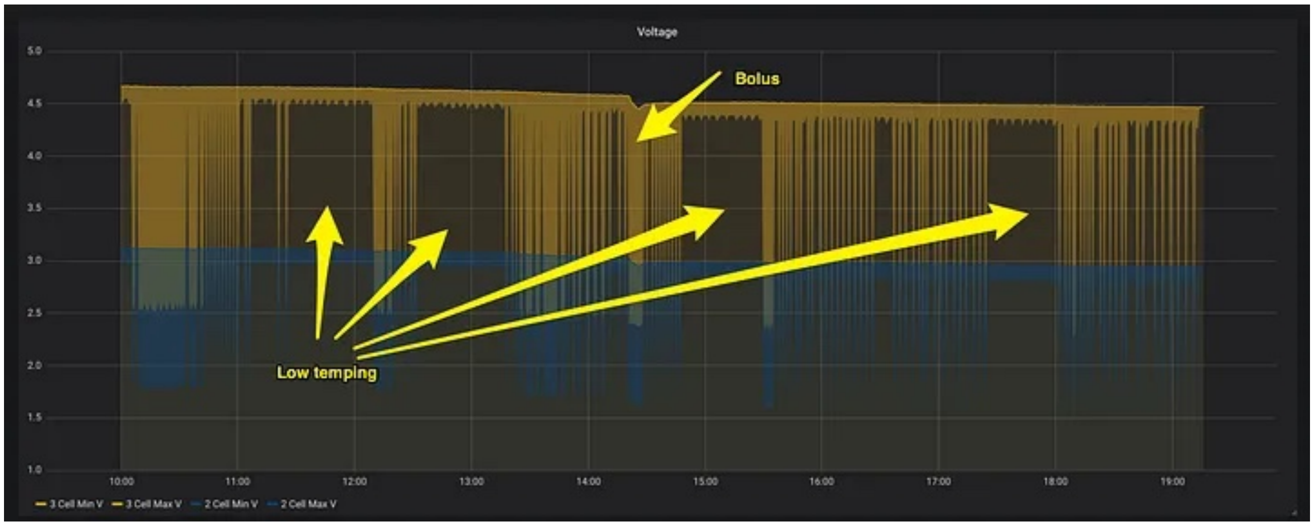
On October 3, 2018, Joe put a Loop controlled pod on himself and became the first Loop Omnipod user, but didn't tell me immediately as he knew I'd be worrying. When he did tell me, I still worried. We had seen the pod work, and understood the functionality, and the basic algorithm of Loop had been vetted for a long time, but still…

A month later, at the November 2018 Nightscout Hackathon, a few more adventurous souls opted try it out for themselves as well, and became part of a small private testing group that would grow to over 30 people before the branch was made public.

Unfortunately, we still had screamers often happening before the full 3 days of use were finished, and we had meticulously compared Loop's commands to PDM reference versions. Eelke was particularly helpful with this process, creating a script that could check commands against their reference versions automatically. I started to get worried that the increased demands on the battery for doing communication every five minutes had pushed the pod over the edge of what was sustainable.

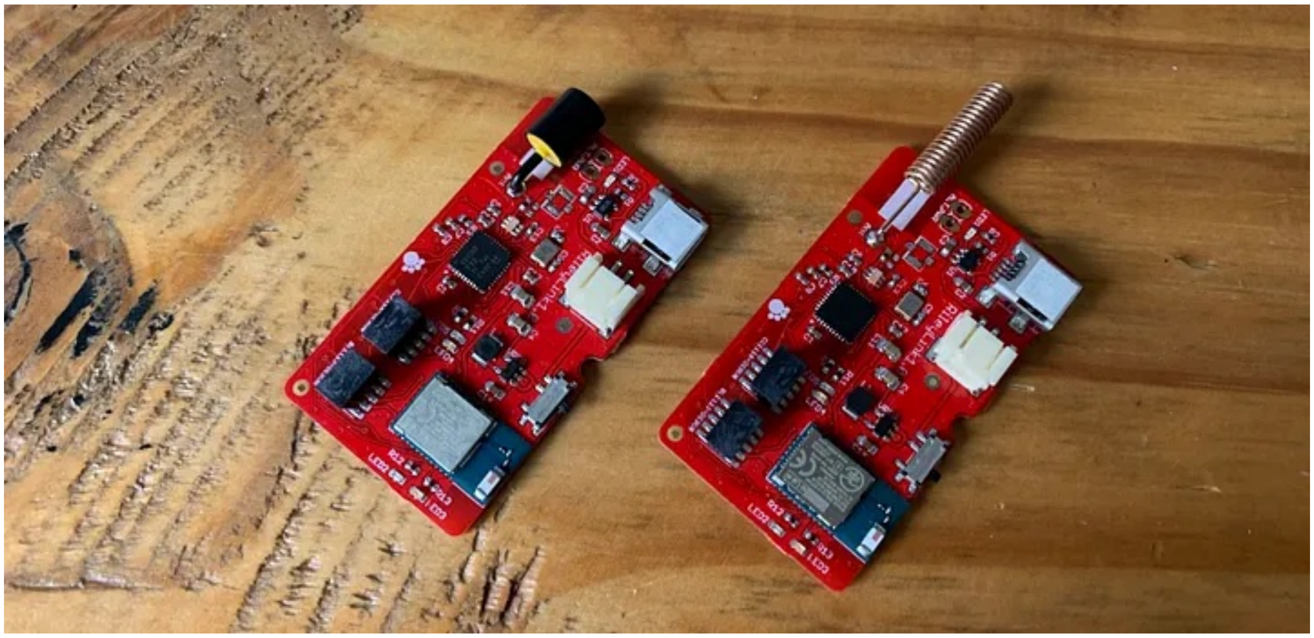

Pod voltage taps, drilled through back plastic, super-glued in place.

So I started measuring pod voltage with an arduino, recording the data and storing it a local database for visualization. Comparing PDM and Loop.



Long time scale view of pod voltage.

Unfortunately, this proved to be a dead end as well; using a PDM and bolusing large amounts, I could drive the pod to lower voltages than even Loop would do over the life of a looping pod, and wasn't able to make the pod scream. It seemed like voltage wasn't the issue, so there must be something else.



RileyLinks with 955 MHz (left) and coil 433 MHz (right) antennas

At one point I noticed that if a message exchange with the pod failed, the pod would sometimes be left in a state of trying to complete the exchange by re-sending packets over and over and over. Testers logs were also showing lots of failures, and so I started experimenting with antennas. Both of these problems should be improved with better comms. I had planned on trying different antennas and had ordered from random places on the internet, but hadn't had time to try them until it became a priority.

I had some flexible 433 MHz antennas that could be taped on the inside of the RL case. These would often have great performance in some scenarios, but not in others; it was too unreliable. When I got to the coil based one, it had good performance very consistently, and at very surprising ranges. Time for a new RileyLink case.

With the new antenna, and some optimizations to do fewer message exchanges while still allowing adjustments every 5 minutes, the frequency of screamers went down to a very low rate. Likely comparable to using a pod normally with a PDM. In the most recent 7,500 hours of live testing, 94% of pods completed without faults.

Testing and Documentation

The testing group grew slowly, which was a great way to continually get new eyes on the system and see what parts were confusing. These testers put up with a lot of screaming pods, and contributed in very big ways to making Loop with Omnipod work better. They did this primarily by sharing issue reports from each of their pods, whether successful or not.

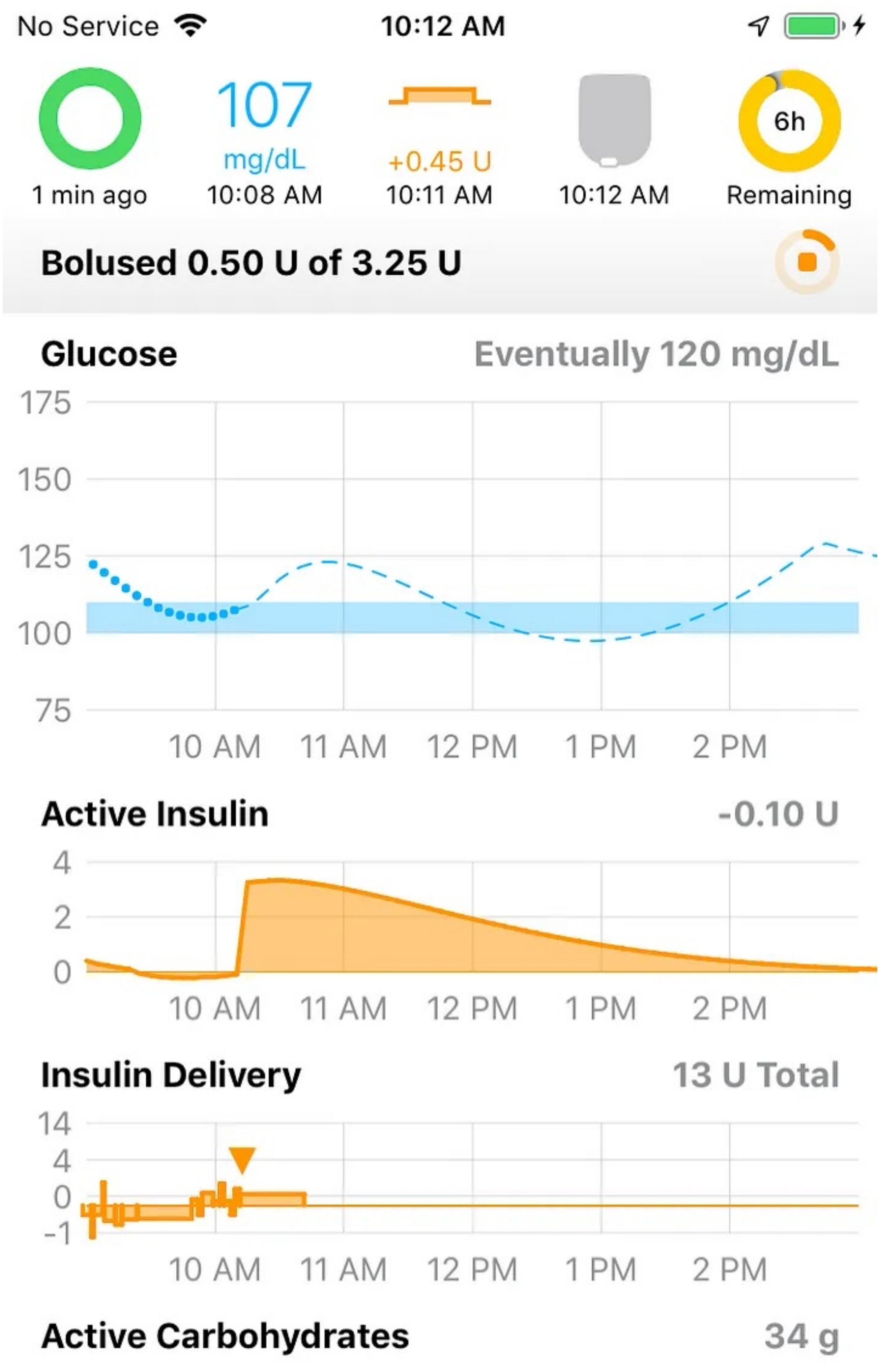
The issue reports have a message log in them that could be analyzed by the tool that Eelke built, giving us insight into whether we had any misformatted commands and also let us gather statistics around certain parts of Loop's interaction with the pods.

Marion Barker joined the testing group and added dedicated reporting, and additional statistics to the progress of testing, and we were able to use her stats of successful pods vs failures to have a high level view of progress.

Eventually Katie DiSimone joined the testing group, and started a large restructuring of loopdocs.org to provide documentation for using Loop with multiple devices. The anticipation around a version of Loop that worked with Omnipod was incredibly high, and without good documentation, it was certain that we'd be flooded with the same questions over and over.

New Features For Loop

Omnipod integration required rethinking of some interface elements, and adding new controls. The pod doesn't report battery, and there is little a user can do about a low battery if one were to somehow happen, so displaying a battery level widget didn't make sense. Also, without a UI on the pump, the user needs to be able to cancel a bolus quickly. The reservoir icon was a picture of a Medtronic reservoir, so we wanted to rethink that. Thank you to Paul Forgione for designing the pod logo that now shows reservoir level.



Bolused 0.50 U of 3.25 U

Glucose Eventually 120 mg/dL

Active Insulin -0.10 U

Insulin Delivery 13 U Total

Active Carbohydrates 34 g

Thanks

Thank you to all the people who help make this long road come to a place where we can share this with others, and realize the goal that we set out to do a long time ago. I know I didn't cover everyone, and everything that happened. That would be impossible in a single post, and for me, since I only have my experience through this. It's hard to imagine the total hours put into this. If you could add them all up, I'm sure it would be shocking. Not to mention the work that has gone into making the Omnipod itself, which I imagine dwarfs this effort. So thank you *all*. Also, many of those hours would otherwise have been spent with families. I really appreciate my wife and kids being understanding with the time I've spent on this, and want to thank them too.

Sidenotes

I have to mention Joakim Ornstedt, as one of the contributors to openomni decoding, and also the creator of what is probably actually the first looping integration with omnipod. He built a device that used optical character recognition (OCR) on the PDM to get data from the PDM, and wired in digital button presses to the PDM through another microcontroller. It's a hard to scale approach, but very clever, and bypasses a lot of the issues we had to deal with for an RE based solution. I really admired him for pulling this off, and getting looping in a tiny fraction of the time it took to make it work with Loop.

. . .

[1] Ben West decoded most of the main communication protocol for Medtronic pumps using a USB Carelink stick. I figured out the RF side of things, and did some additional work on the protocol.

. . .

Programming SEM Type 1 Diabetes Sdr Health

Written by Pete Schwamb

Follow