Reverse Engineering a Smartwatch

Benjamen Lim · Follow

Published in CSG @ GovTech · 9 min read · Jun 27, 2024

Some time ago, I was assigned a consignment of smart watches with geolocating capabilities that were being mothballed after a trial. I was determined to find some use for them and thus began my journey of reverse engineering a smartwatch!

In this article, I will share the reverse engineering process by first highlighting some initial observations on the watch surface and circuitry, before going into detail as to how I reprogrammed the smartwatch, and the final step of patching the firmware so it could be repurposed.

Initial Observations

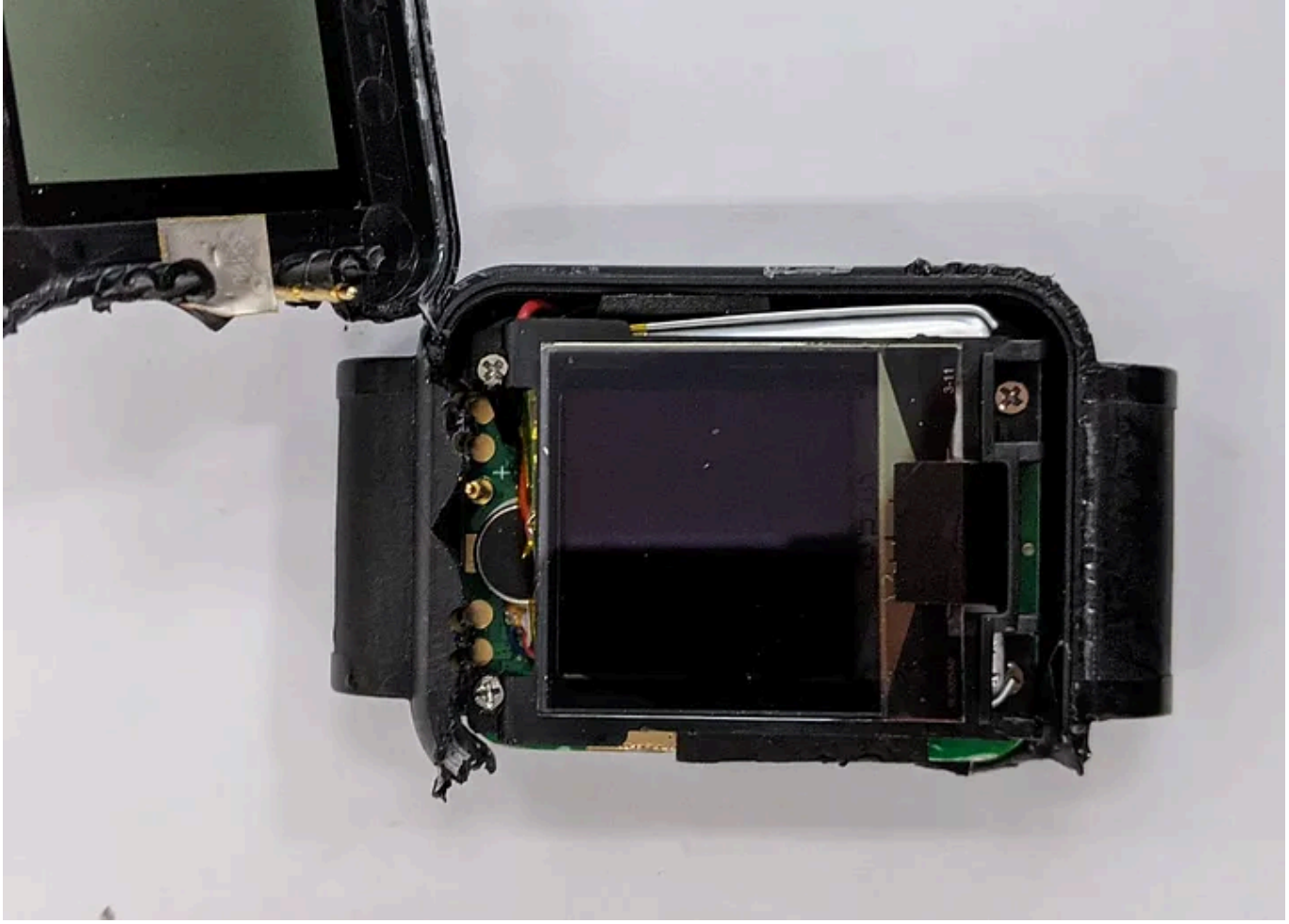
The watches as delivered were bare-bones and had a single page of instructions on how to charge and use them. Each box contained a single charger and a watch. There were no READMEs, websites, or developer portals. The watches themselves only had a single capacitive sensor on the surface that turned on the screen and allowed the user to view their heart rate, some debug information, and the time on different watch faces.

Watch faces

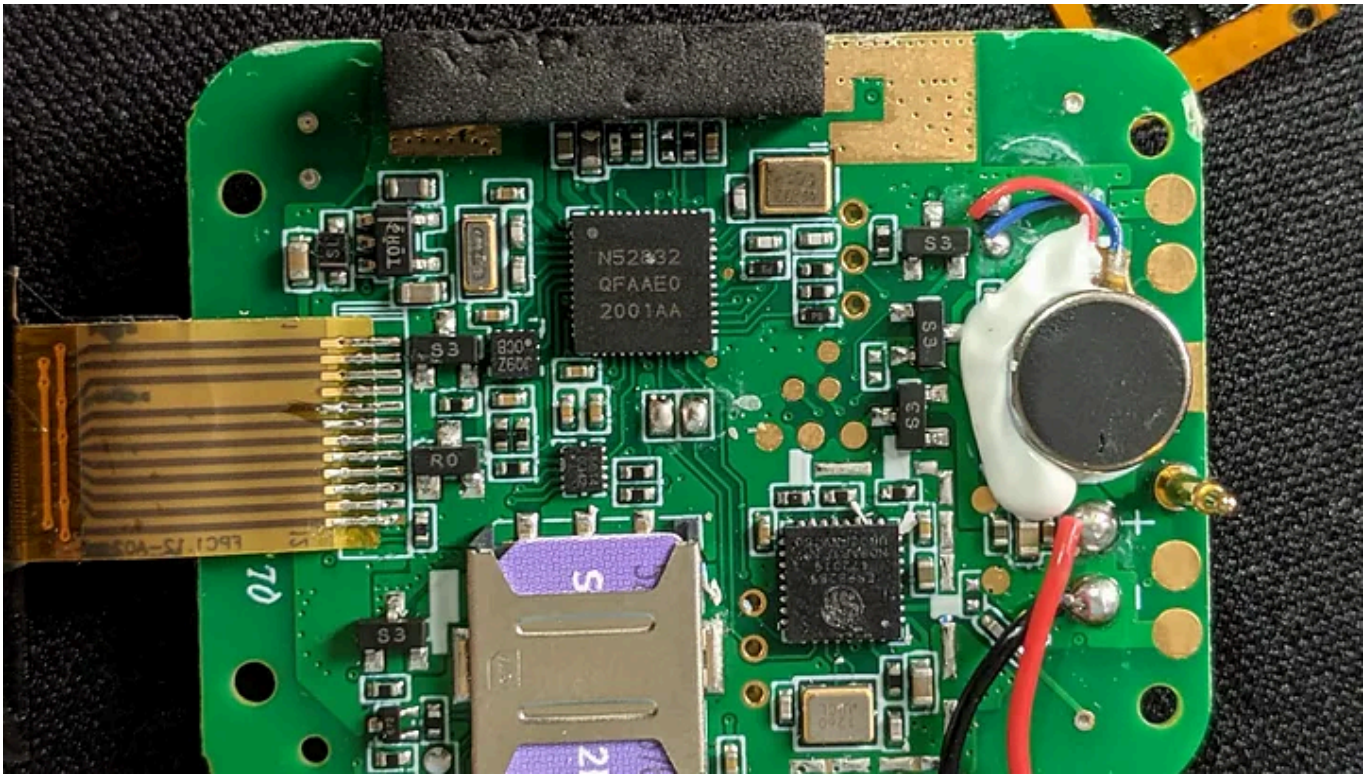
The debug information provided useful information for identifying the watch. I observed that the IP address on the debug watch face was common across different watches of the same model. My initial guess was that it referred to the IP address of the server that these watches were communicating with.

Charging the watch

The watch had an IP67 rating, i.e. they waterproof, so there was no way I could access these watches (with my limited tools) without damaging them. In the spirit of exploration, I took a pair of pilers to a watch, tearing it down to determine what lay within. It wasn't easy, but after an hour of careful prying and cutting, I managed to remove the outer casing of the watch.



Cracking the device open

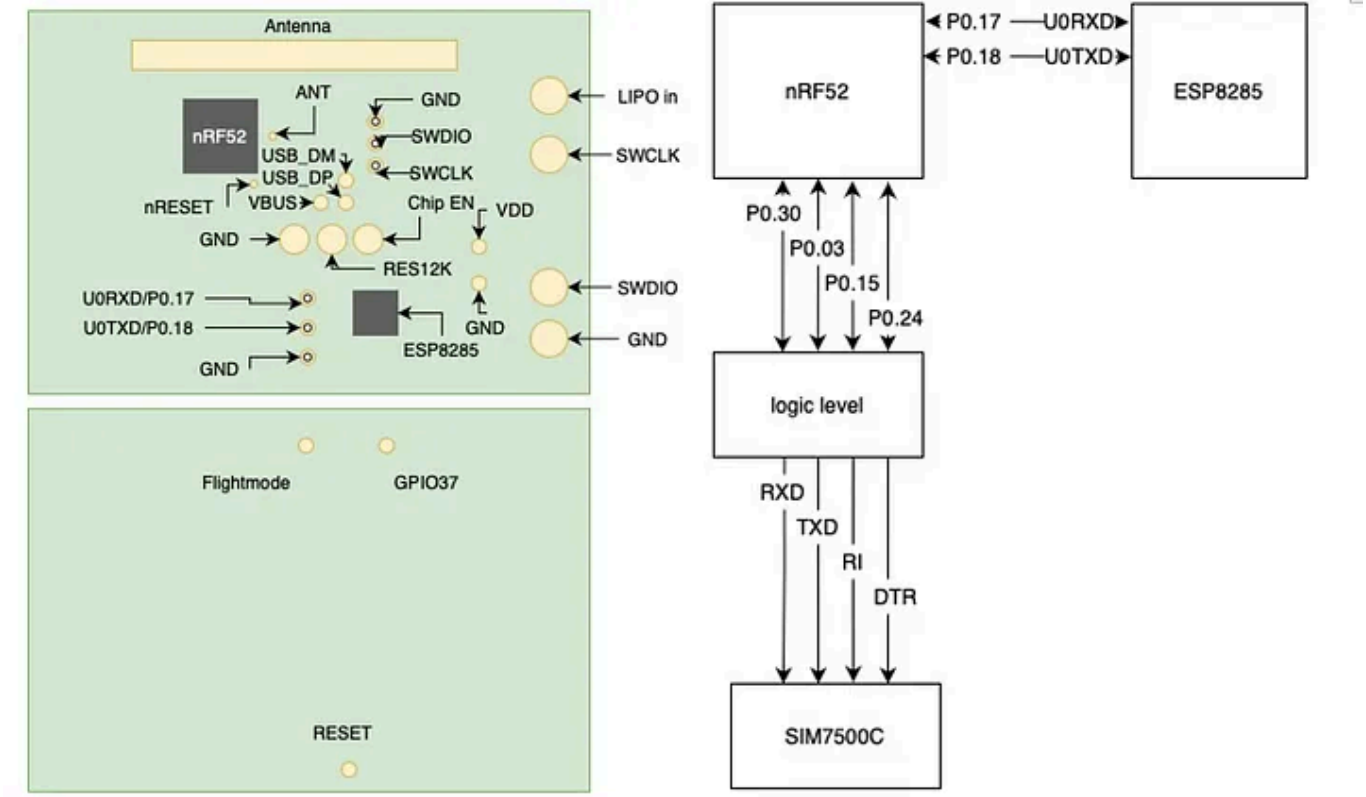


Viewing the innards

Inside was a single PCB with a LIPO battery. There were two antennas embedded in the case itself and were connected to u.FL connectors on the board.

Reverse Engineering the PCB

The SmartWatch was driven by a Bluetooth-enabled nRF52832 chip and had two other major ICs: a WiFi-capable ESP8285 and a cellular IC from SIMCOM. I was initially confused by the presence of the WiFi microcontoller, because there was no indication that the watch had WiFi capabilities. However, I found out later that it was used for urban triangulation. Because the GPS's accuracy is generally quite bad in an urban environment, an approximation of the location of the watch can be determined by using the WiFi AP and cellular data combined with historical GPS data. From the layout, the nRF52832 was the device's main IC, and used the WiFi chip to scan for local WiFi Access Points (APs). The nRF52832 also communicated with the SIMCOM device over UART and issued commands to communicate with the mobile network. Knowing that, I focused my efforts on I was looking for any UART or exposed programming pins on the nRF52832, as it was main IC and those connections are commonly used to interact with the microcontroller.



In this case, quite a few exposed UART and programming pins were on the board. One very interesting trace that I found was that the round gold contacts on the right side of the PCB were connected to SWDIO and SWCLK on the nRF52832 IC in addition to the connections that I expected for charging the battery of the watch. SWDIO and SWCLK are JTAG programming pins that are used to program the chip. There were also pogo pins, spring loaded electrical contacts, attached on the underside of the watch's front cover that would make contact with these pads when the watch was closed.

These pogo pins were exposed to the copper-plated contacts on the front of the watch's face, meaning that the programming pins for the main chip were exposed on the device's face. Having exposed programming pins was unusual because in general, the firmware for the device would be flashed into the PCB at the factory and would be subsequently updated over WiFi or Bluetooth. There is usually no need to expose the programming pins. This feature turned out to be a very helpful feature for later as it meant that I didn't need to open the watch to gain access to the firmware.

Needing to open the watch would have rendered useless my intention to repurpose the watch as evidenced by my attempt at opening the watch earlier: it wasn't something I was going to put back together anytime soon.

What was *more interesting* was that on the charging adapter, the pogo pins that connected to the SWCLK and SWDIO contacts on the watch's face were connected to D+ and D- pins on the microUSB port. The D+ and D- pins are usually used for data transfer. This meant that I didn't even need to build a custom programming platform to reprogram the watch: all I had to do was to splice a microUSB wire to expose the programming pins to any watch connected to the charger. Very nice.

Contacts

Splicing the cable

After I connected the watch to my JLink debugger through the charger, the first thing I noticed was that the watch was producing debug output using the JLink RTT viewer. Success! While being able to observe the debug output was very useful, however, as there was no input configured for the RTT module, so there was no way to send commands to the watch. However, the output confirmed my earlier assumptions about how the watch was connected internally.

UART communications being sent to the SIMCOM module from the nRF52832 IC

After a few exploratory attempts at sending commands over JLink, I decided to take a look at the firmware. With my JLink attached, I was able to dump the firmware using `nrfjprog` with the `--readcode` and `--readram` flags.

Raw firmware dump

The firmware was not read or write protected so I was able to obtain the full firmware dump. At this point I had two options for repurposing the watch: 1) I could reprogram it completely by flashing a new firmware into the device, or 2) I could patch the IP address and port in the firmware so that the watch would send data back to a server I control. Since reprogramming the watch was likely a huge undertaking, I decided to try patching the watch.

Enter the Ghidra

I needed to dump both the RAM and the flash to get a proper mapping of the functions and the variables in Ghidra. Since this is bare metal code from a Cortex M0+ device, I had to decompile the firmware as ARM little-endian and Ghidra was able to produce semi-readable pseudocode.

My objective at this point was to investigate the location of the IP address, which I believed was stored somewhere in the code. The watch sends data over a cellular connection to a server, the address of which was displayed on the watch's debug face. If I can update the IP address, I will be able to redirect the data that the watch was sending to a server I control.

However, my initial attempts to find a hardcoded string that corresponded to the IP address that I observed on the face of the watch were not successful.

My next attempt was to find the closest string that occurred in the debug output that I observed in the RTT viewer. In this case, it was `AT+CIPOPEN=0` which is the serial command sent from the nRF52832 to the SIMCOM module, instructing it to open a connection to an IP address which is specified as an argument. By finding this string, because it uses the IP address of the server that the watch has to connect to, I would be able to find the location in memory where the server's IP address is stored.

String found in memory

I was more successful at finding a format string. This match had a string pattern that matched that of an IP address. It was being called in a function that looked suspiciously like a `sprintf` function.

Reference to string being called in a sprintf function, which calls four other variables

The sprint function referenced an array that was stored in `DAT_20000887`, which corresponds to the Data RAM section of the nRF52832 datasheet as seen in the memory mapping below.

Memory map of nRF52832

Going to the RAM's location in memory, I found that it was being written to in two functions. Only the first location was of interest because it was where the hardcoded IP address was. The second location allows the IP address to be updated later over the air.

Jumping to the first function, I found these hardcoded variables hex values in memory that was written to the array as part of the main function, and these variables corresponded to the IP address that I observed on the watches.

Location in memory for patching

These locations in memory were the six bytes of data in the firmware that I needed to patch for the IP address and port.

6 bytes to patch, 4 bytes for IP, 2 bytes for port

One small wrinkle that emerged was that the original port number was 38899, which was saved as two bytes: 0x0c and 0x68. The compiled program used a `movn` instruction, which applied a Boolean `not` operation to the hardcoded values before they were put into RAM. Technically, the instruction could be patched to remove the `not` operation, but I was trying to minimise the number of bytes to change, therefore, I added an additional operation when converting the desired port number to the appropriate hex values.

Reprogramming the Smartwatch

With this knowledge, I was able to write a simple script that patched the firmware with any IP address and port that I wanted. The script also updated the checksum of each patched line to ensure that the firmware matched the expected format, and once the patching was complete, I flashed the firmware to the watch using Nordic Semiconductor's Programmer utility program.

Firmware patching script

Firmware Flashing

I am pleased to say that it worked! By updating the firmware to communicate with the server, we were able to get data from the phone and process it.

Patched firmware

That concludes the reverse engineering portion of this project. There was a lot of learning for me as I came in with limited knowledge of reverse engineering ARM devices. However, I had a good idea of how I would have developed the device and what it was capable of, so that significantly helped to narrow down what I was looking for.

An interesting takeaway was how the programming interface was exposed to the USB port which I have not observed in other watches. Not having

any read-write protection on the firmware is also uncommon as most other IoT devices tend to enforce firmware protections once they are in production to prevent trivial cloning of the firmware.

Nevertheless, it was great fun, and being able to revive and reuse electronics that would have either sat in a warehouse for a long time or thrown away gave this project additional meaning.

Cybersecurity IoT Csg Reverse Engineering Hardware

Written by Benjamen Lim

Follow

102 Followers · Writer for CSG @ GovTech

IoT and Applications Engineer