ESP32 Open MAC

Reverse engineering the ESP32 Wi-Fi hardware

Jasper Devreker

esp32-open-mac.be

2024-09-06

The ESP32

- low cost Wi-Fi/Bluetooth microcontroller ($\sim$ €2)
- dual core
- 520 KB RAM
- FreeRTOS (real-time operating system)
- very popular with makers/engineers

Figure: ESP32 on a dev board

A bit of history

- made by Espressif
- ESP8266, released in 2014
- started out as badly documented Wi-Fi module
- got picked up by makers for its price/potential
- 10 years later, more than 1 billion chips have been produced
- almost the entire SDK is open source

Almost the entire SDK open-source?

- except for the libraries to control the Wi-Fi and Bluetooth peripherals
- provided as compiled libraries
- these abstract away the hardware and Wi-Fi stack
- example: `esp_wifi_set_mode(WIFI_MODE_STA)`

Open-source, but the source is not open

- Binary blobs are “open-source”
- as in, the blobs themselves are licensed under the Apache 2.0 license
- . . . not entirely to the spirit of open-source, but useful for us
- this grants us explicit permission to reverse engineer the code

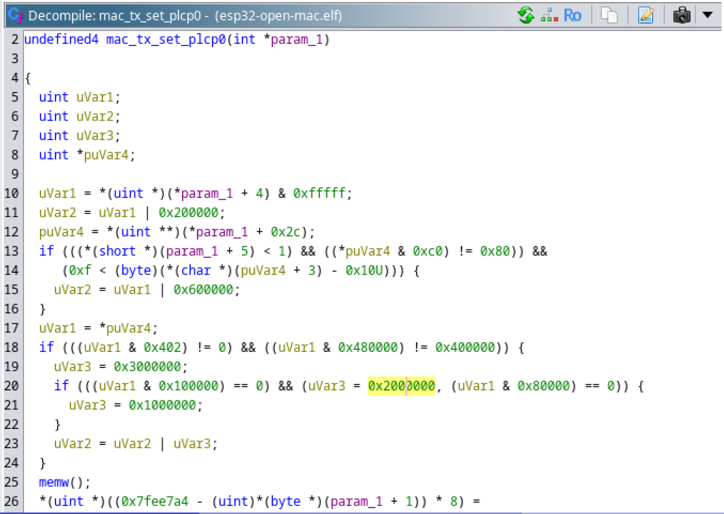
Why?

- originally: proper 802.11s mesh networking
- auditability
- “make hardware do things it was not designed for”

Table of Contents

Static analysis

- Ghidra
- now has mainline support for Xtensa (thanks Austin!)
- Fortunately, Espressif did not strip function names



```
undefined4 mac_tx_set_plcp0(int *param_1)

{
  uint uVar1;
  uint uVar2;
  uint uVar3;
  uint *puVar4;

  uVar1 = *(uint *)(*param_1 + 4) & 0xfffff;
  uVar2 = uVar1 | 0x200000;
  puVar4 = *(uint **)(*param_1 + 0x2c);
  if (((*(short *)(param_1 + 5) < 1) && ((*puVar4 & 0xc0) != 0x80)) &&
     (0xf < (byte)(*(char *)(puVar4 + 3) - 0x10U))) {
    uVar2 = uVar1 | 0x600000;
  }
  uVar1 = *puVar4;
  if (((uVar1 & 0x402) != 0) && ((uVar1 & 0x480000) != 0x400000)) {
    uVar3 = 0x3000000;
    if (((uVar1 & 0x100000) == 0) && (uVar3 = 0x2000000, (uVar1 & 0x80000) == 0)) {
      uVar3 = 0x1000000;
    }
    uVar2 = uVar2 | uVar3;
  }
  memw();
  *(uint *)((0x7fee7a4 - (uint)*(byte *)(param_1 + 1)) * 8) =
```

Dynamic analysis on real hardware

- JTAG to add breakpoints/inspect memory
- Wi-Fi dongle in monitor mode to receive all packets
- problem: lots of other Wi-Fi networks nearby

Building an affordable faraday cage

- needs to have data passthrough
- ... but powerline filter to let power in is very expensive
- solution: use fiber optics + big lead-acid battery
- `https://esp32-open-mac.be/posts/0003-faraday-cage/`
- at least 70 dB of attenuation @ 2.4GHz

Figure: Faraday cage

Dynamic analysis in emulator

- Espressif already has QEMU fork for their HW
- added support for Wi-Fi peripheral based on assumptions from static reversing
- added "execution tracing": a stacktrace is saved on every wifi peripheral access

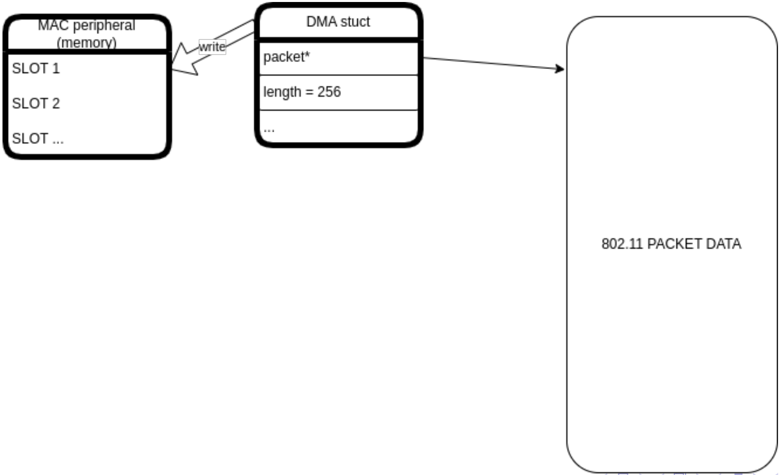
Table of Contents

Taking over from the proprietary implementation

- let proprietary code handle entire HW initialisation
- replace interrupt with our own
- this eliminates the proprietary FreeRTOS task

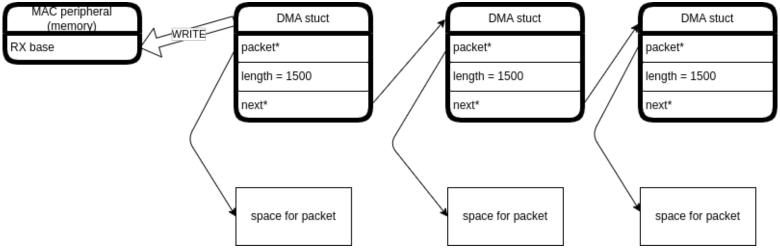
Reverse engineered TX

- DMA
- 5 TX slots
- interrupt when slot is finished with status



Reverse engineered RX

- also DMA
- chain of DMA slots
- interrupt when slot is filled in
- recycle buffers after RX



RX/TX to connect to access point

- connecting to open access point is surprisingly easy

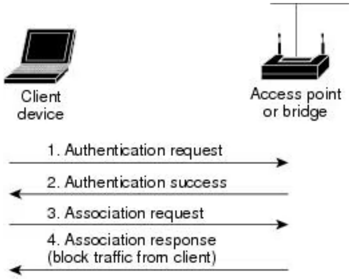

Client device
Access point or bridge
1. Authentication request
2. Authentication success
3. Association request
4. Association response (block traffic from client)

Figure: Authenticate/associate flow

Sending data to access point

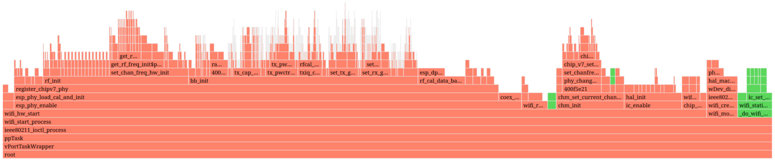
- pretend we are an ethernet card (ESP-NETIF, uses LwIP under the hood)
- put packets into data frame
- we can ping the network card with only open source code running

```
64 bytes from 10.0.0.233: icmp_seq=1149 ttl=255 time=16.9 ms
64 bytes from 10.0.0.233: icmp_seq=1150 ttl=255 time=24.4 ms
64 bytes from 10.0.0.233: icmp_seq=1151 ttl=255 time=19.1 ms
64 bytes from 10.0.0.233: icmp_seq=1151 ttl=255 time=21.5 ms (DUP!)
64 bytes from 10.0.0.233: icmp_seq=1152 ttl=255 time=13.1 ms
```

Table of Contents

Eliminate the blobs

- "we can ping the network card with only open source code running" is a bit misleading
- proprietary code is still needed to init HW
- calibration, . . .
- very complex code



Eliminate the blobs

- current way: top-down replacement of init function
- for now: keep blob to do radio calibration, only handle packets
- maybe in the future: mechanical translation to C with rev.ng

Further HW reverse enineering

- 802.11n
- AMPDU, HT40, QoS
- hardware acceleration for WPA cryptography
- change channel, TX power

Adding an 802.11 MAC stack

- we can now send packets, but what packets do we send?
- needed for more complex actions (scanning, joining WPA2 AP, . . .)
- PoC stack worked, but writing a whole stack is tedious
- . . . but that's exactly what we're doing!
- in Rust!
- ~~maybe use FreeBSD's net80211 stack~~

Porting to other ESP32 variants

- there are RISC-V versions of the ESP32
- preliminary: the Wi-Fi peripheral seems to be very similar

Bluetooth

- will likely be as much work as getting Wi-Fi to work

Thanks to!

- contributors: Simon, Antonio, Austin, Ishwar
- my day job: Dekimo Gent
- my hackerspace: Zeus WPI

Questions?

- `https://esp32-open-mac.be`