Liberating Wi-Fi on the ESP32

Jasper Devreker - redfast00
Simon Neuenhausen - Frostie314159

Who we are

Jasper Devreker

(redfast00)

Embedded & security engineer

Simon Neuenhausen

(Frostie314159)

Reverse engineer and Rust developer

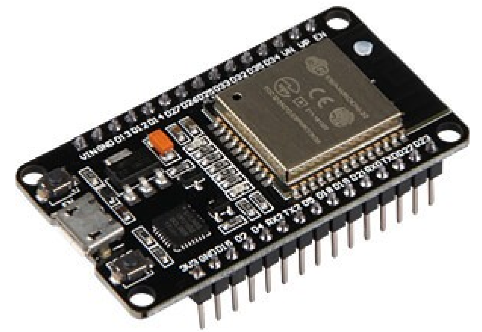
DECT: 8041

Why?

- Hardware can do more than what the proprietary library allows
  - 802.11s mesh networking
  - Apple Wireless Direct Link
  - Nintendo DSi PictoChat
- Security auditability

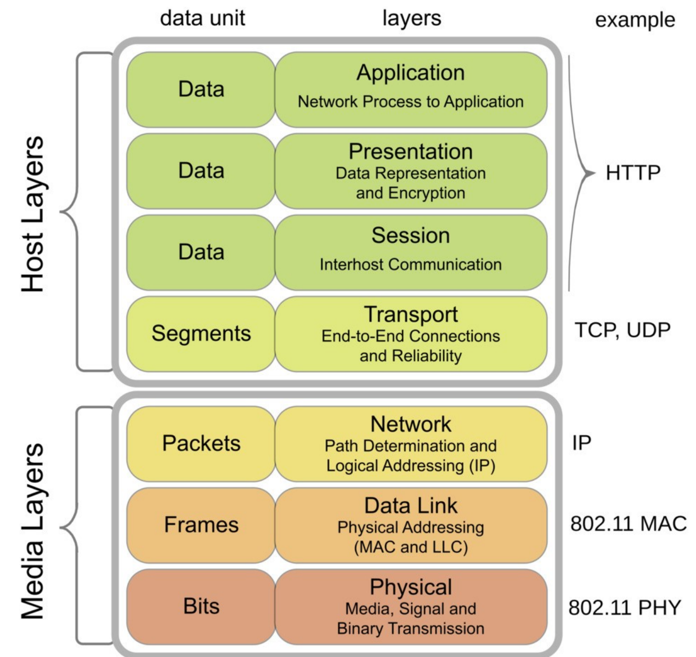
What is the ESP32?

- Low cost Wi-Fi/Bluetooth microcontroller (∼ €2)
- Dual core, 520 **KB** RAM
- More than 1 billion sold
- Almost the entire SDK is open source

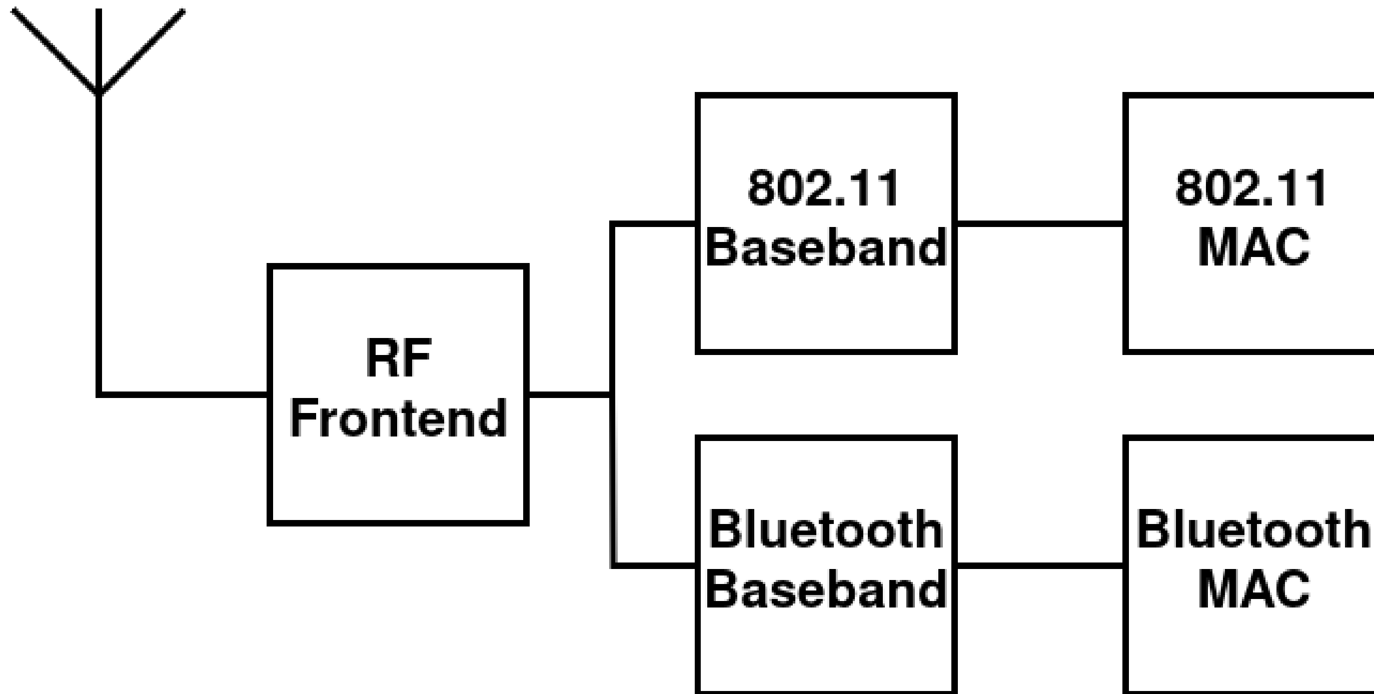


What is Wi-Fi

- Marketing term for the WLAN technology specified in IEEE 802.11
- Operates on Frames
- Defines Layer 1 & 2



How Wi-Fi works on the ESP32



The current situation

- Closed source Wi-Fi stack
- Espressif ships binaries licensed under Apache
- API exposed in public header files
- Public API is well documented
- Wi-Fi hardware was licensed from Riviera Waves

Reverse engineering

- Static vs dynamic
- On hardware vs in emulator

Intro to hardware reversing

- Interaction with Wi-Fi peripheral happens via MMIO
- There is a large undocumented hole in the memory map → mostly related to Wi-Fi and BT

Static analysis

- Ghidra now has mainline Xtensa support
- Espressif did not strip function names

Decompile: mac_tx_set_plcp0 - (esp32-open-mac.elf)

```
 2 undefined4 mac_tx_set_plcp0(int *param_1)
 3
 4 {
 5   uint uVar1;
 6   uint uVar2;
 7   uint uVar3;
 8   uint *puVar4;
 9
10   uVar1 = *(uint *)(*param_1 + 4) & 0xfffff;
11   uVar2 = uVar1 | 0x200000;
12   puVar4 = *(uint **)(*param_1 + 0x2c);
13   if (((*(short *)(param_1 + 5) < 1) && ((*puVar4 & 0xc0) != 0x80)) &&
14      (0xf < (byte)(*(char *)(puVar4 + 3) - 0x10U))) {
15     uVar2 = uVar1 | 0x600000;
16   }
17   uVar1 = *puVar4;
18   if (((uVar1 & 0x402) != 0) && ((uVar1 & 0x480000) != 0x400000)) {
19     uVar3 = 0x3000000;
20     if (((uVar1 & 0x100000) == 0) && (uVar3 = 0x2000000, (uVar1 & 0x80000) == 0)) {
21       uVar3 = 0x1000000;
22     }
23     uVar2 = uVar2 | uVar3;
24   }
25   memw();
26   *(uint *)((0x7fee7a4 - (uint)*(byte *)(param_1 + 1)) * 8) =
27        uVar2 | (*puVar4 >> 8 & 1) << 0x1b | (*puVar4 >> 9 & 1) << 0x1c;
28   memw();
29   return 0;
30 }
```

Dynamic analysis on real HW

- Use a JTAG debugger to set breakpoints (2)
- Wi-Fi dongle in monitor mode to receive packets
- Problem: lots of other networks nearby

Faraday cage

- Data passthrough via fiber
- No power passthrough, but battery
- at least 70 dB of attenuation @ 2.4GHz

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

Tradeoffs of each method

	Static analysis	Debugging HW	Emulating HW
Breakpoints	N/A	2	infinite
Guaranteed correct	Yes	Yes	No
Used for	Finding exact details	Verifying assumptions	Finding general direction

Reverse engineering results

- The hardware does a lot for us
  - Transmit packet
  - Automatically send ACK to received packets
  - Hardware cryptography, just tell it the keys
  - Receive packets into memory

Transmitting packets

- Write packet content to memory
- Write metadata to hardware registers (rate, length, …)
- Write address of packet to hardware register
- Set TX bit on slot
- Wait for interrupt

Receiving packets

- Write a linked list with each node pointing to a buffer to receive packets in
- Every time a packet is received, an interrupt is called
- Recycle the buffers!
- But what about ACKs? 10 µs of time is not a lot of time

Receive filters

- Avoid handling every packet in software
- Instead, only process packets we are interested in
- Hardware allows filtering based on RA / BSSID
- Automatically sends ACK to TA of matched packet

Cryptography acceleration

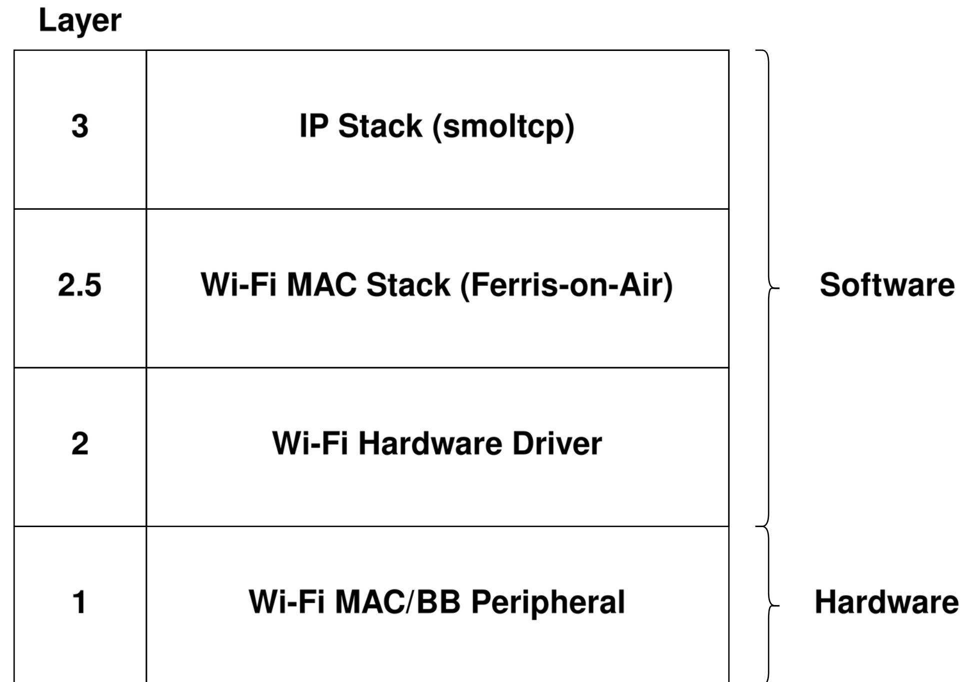
- With WPA, every packet needs to be encrypted/decrypted
- Hardware does this for us; set the key, algo and MAC address in one of the key slots
- When transmitting, tell the hardware what key slot index to use to encrypt + set the Protected bit
- When receiving, it will automatically decrypt based on the MAC address

Now we need a MAC stack

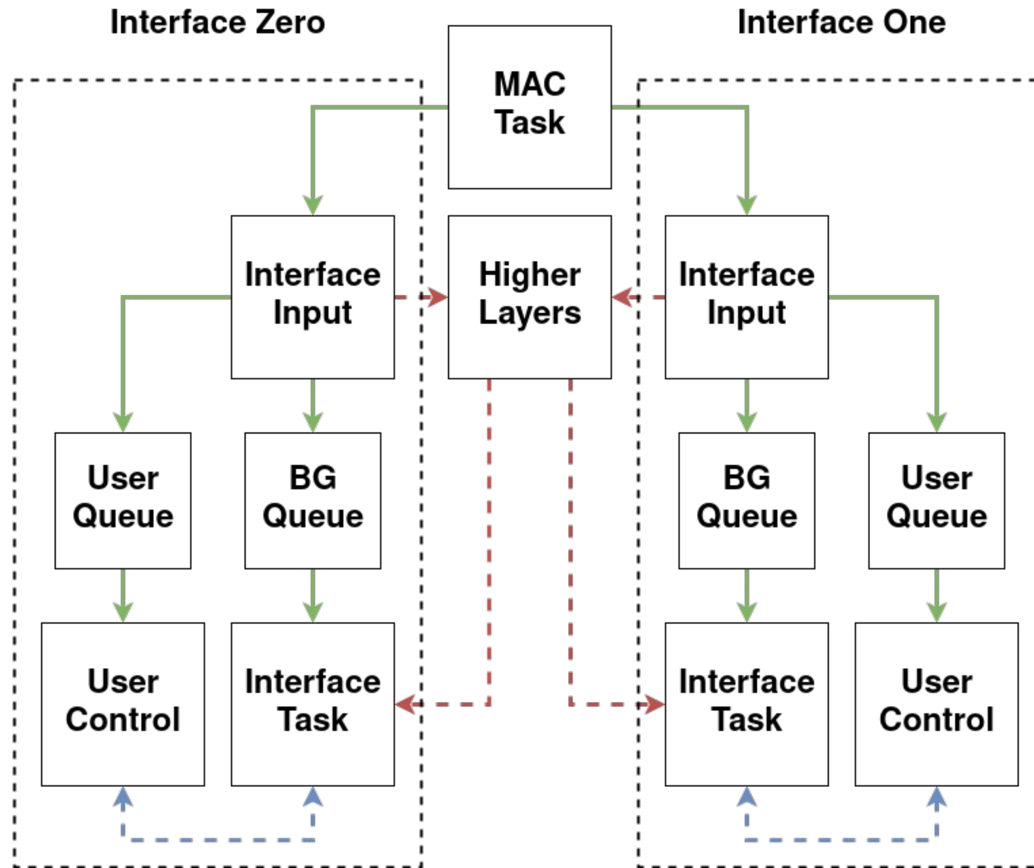
Ferris-on-Air

- Asynchronous IEEE 802.11 stack written in Rust
- Open Source
- Currently only supports the ESP32
- Uses modular interface design
- STA interface supports basic features

FoA in the OSI model



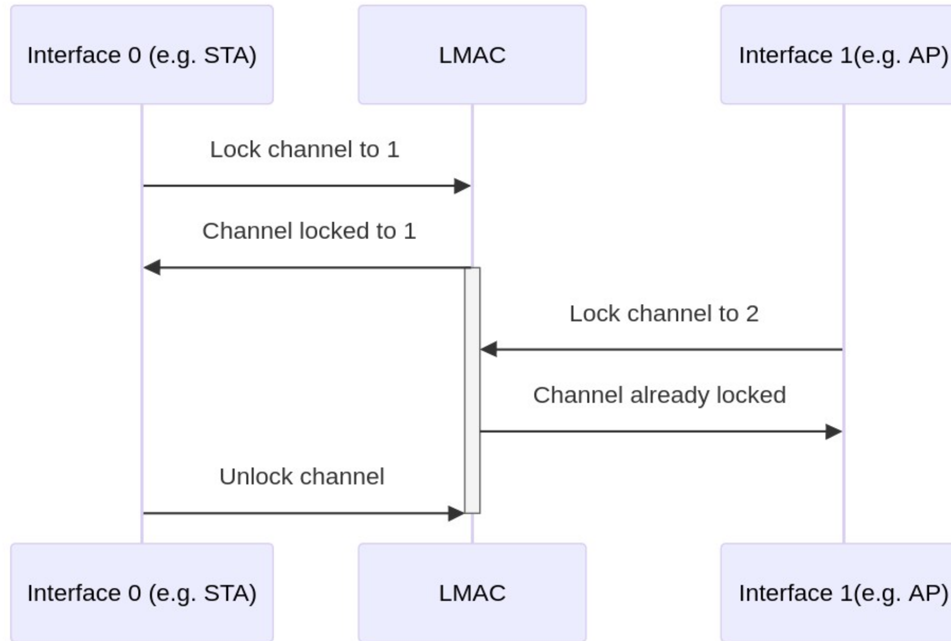
FoA's Architecture



The Lower MAC

- Divides access to the medium among the interfaces
- Controls the channel
- Thin layer between hardware driver and upper MAC

Channel Locking



What the STA interface can do

- Scanning
- Connecting
- Disconnecting
- User Data TX & RX

What it can’t do (yet)

- Rate selection
- Encryption
- Broadcast protection
- Power save
- 40 MHz Operation
- Target Wake Time
- Fine Timing Measurement
- WPA-Enterprise
- AMSDU
- AMPDU
- QoS
- Long Range mode
- Channel State Information

Future Work

- Implement missing features
- AP mode
- AWDL (AirDrop, Airplay)
- Mesh operation
- Indoor Navigation (maybe for c3nav?)

Thanks to these people

- Zeus WPI
- Austin Conn
- Everyone at the fNordeingang
- The embassy and esp-hal projects
- Espressif

Thanks for listening!

Questions?