Networking

Lecture 8

Networking

- OSI Network Stack
- Wi-Fi
- TCP/IP
- Raspberry Pi W
- Protocols

OSI Network Stack

Open Standard for Intercommunication

Bibliography

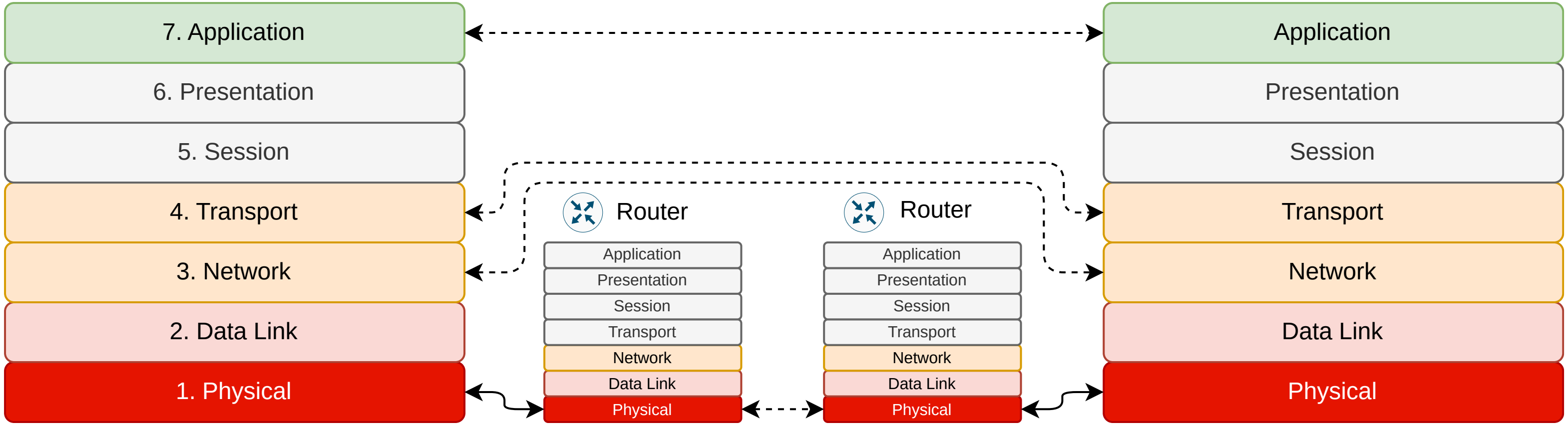
for this section

Andrew Tanenbaum, *Computer Networks (5th edition)*

- Chapter 1 - *Introduction*
  - Subchapter 1.1 - *Uses of Computer Networks*
  - Subchapter 1.2 - *Network Hardware*
  - Subchapter 1.3 - *Network Software*
  - Subchapter 1.4.1 - *The OSI Reference Model*

Standardized Interfaces

7 layers, each one communicates with its counterpart



- **L1** *hardware*, sends and receives data on the physical media
- **L2** *hardware* and *driver* sends and receives data from a device that it is directly connected to
- **L3** *driver* sends and receives data from devices not directly connected to using L2 from device to device
- **L4** *driver* connects the applications to the networking stack
- **L5/L6** not used
- **L7** is the *application*

Wi-Fi

Wireless Network

Bibliography

for this section

Andrew Tanenbaum, *Computer Networks (5th edition)*

- Chapter 1 - *Introduction*
  - Subchapter 1.5.3 - *Wireless LANs: 802.11*

Wi-Fi

- Wireless Network
- *L2* (Data Link) Protocol
- Devices
  - **AP** - Access Point
    - acts as a hub or switch
    - handles authentication
  - **Device** - The device that connects to the network
- Frequencies
  - 2.4 GHz
  - 5 GHz

Wireless Network Connection

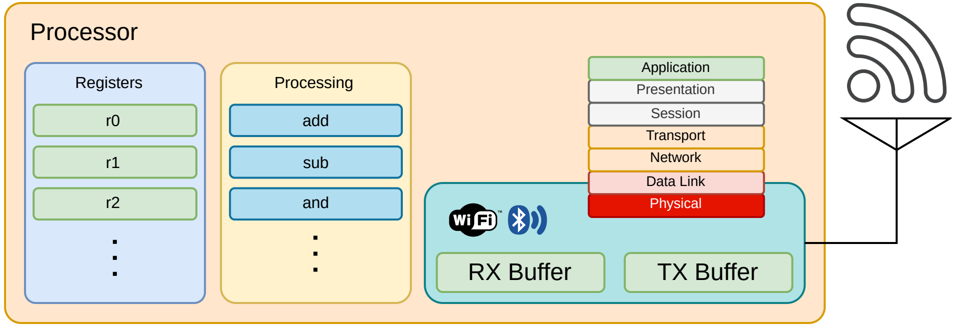
security

- **Open** - everyone receives all the communication
- **WEP** - all data is encrypted with the same key, everyone who knows the keys can read the data
- **WPA 1/2/3 (Personal)** - each **device** has a different encryption key shared with the **AP**
  - the **device** authenticates with the **AP** by using the network *passkey*
  - the **device** and the **AP** exchange a symmetric encryption key
- **WPA 1/2/3 Enterprise** - each **device** has a different encryption key shared with the **AP**
  - the **AP** provides a certificate to the **device** proving its authenticity
  - the **device** authenticates using username and password or a private key
  - the **device** and the **AP** exchange a symmetric encryption key

Integrated Network Device

the network device is integrated into the MCU

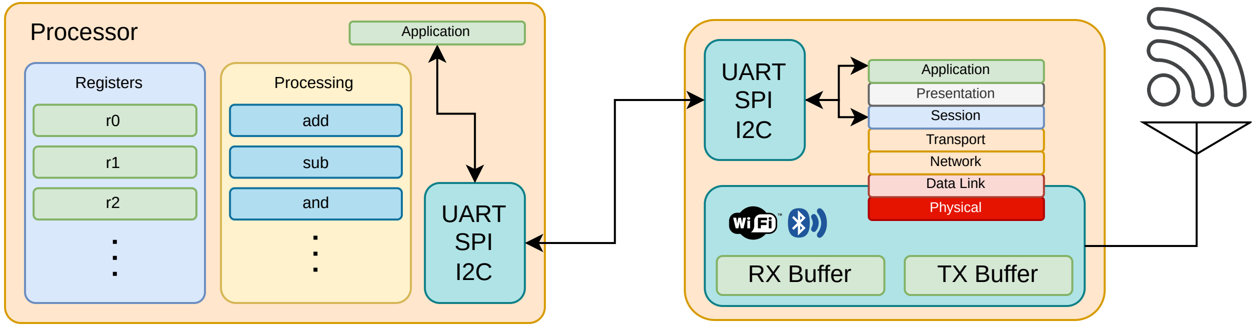
- a radio peripheral
  - knows how to emit and receive in 2.4 and 5 GHz
  - is controlled by software
  - can generate signals for Wi-Fi, BLE, 802.15.4, 6LoPAN, Thread
- it knows how to transmit and receive buffers (*L1*)
- some devices know *L2*



Discrete Network Device

the network device is connected into the MCU

- the MCU is connected to an external Wi-Fi/BLE device
- transport over UART, SPI or I2C
- most devices knows
  - *L3* - provides *socket*
  - *L4* - provides TCP/UDP *sockets*
  - *L7* - provides application functions (usually *HTTP* and *MQTT*)



TCP/IP Stack

Transport Control Protocol over Internet Protocol

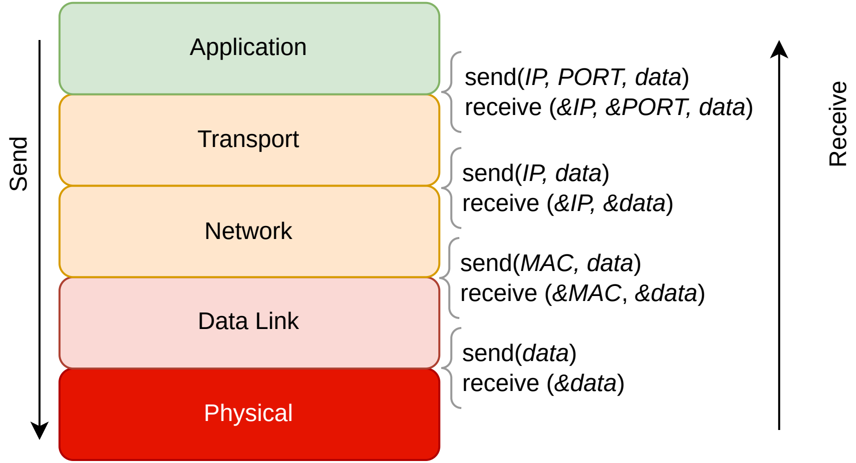
Bibliography

for this section

Andrew Tanenbaum, *Computer networks (5th edition)*

- Chapter 1 - *Introduction*
  - Subchapter 1.4.2 - *The TCP/IP Reference Model*

TCP/IP Stack



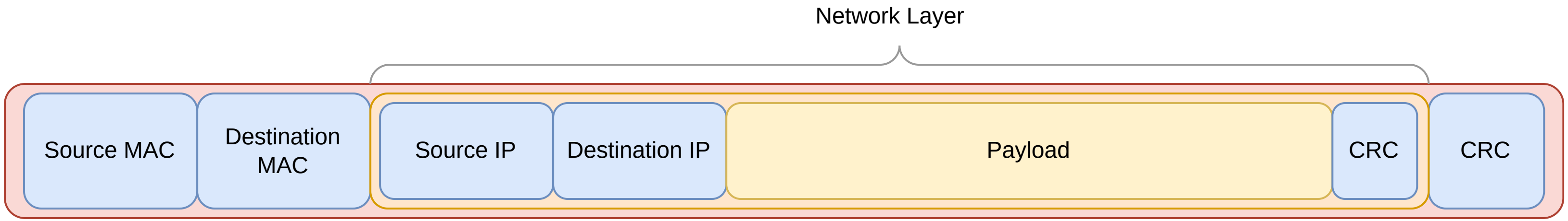
*the initial TCP/IP stack did not make any difference between the *Physical* and the *Data Link* layers

Data Link Layer

- very similar for Ethernet and Wi-Fi (HDLC)
- uses *Media Access Control* (*MAC*) addresses
- sends and receives *frames* from other devices directly connected to the same network

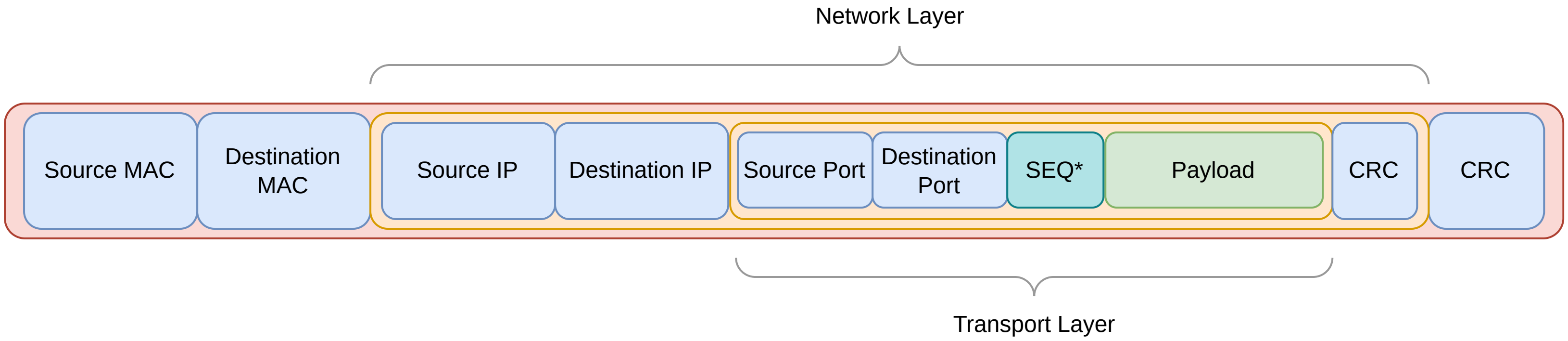
Network Layer

- Internet Protocol
- uses *Internet Protocol* (IP) addresses
  - *IPv4* - 32 bits
  - *IPv6* - 128 bits
- sends and receives *packets* from other devices remotely



Transport Layer

- Two protocols
  - *Transport Control Protocol* (*TCP*) - stream of data, makes sure it gets to the destination
  - *User Datagram Protocol* (*UDP*) - *fire and forget*, best effort do deliver the packet
- uses *Ports* to identify the destination and source application
- sends and receives *packets*



Raspberry Pi Pico W

Bibliography

for this section

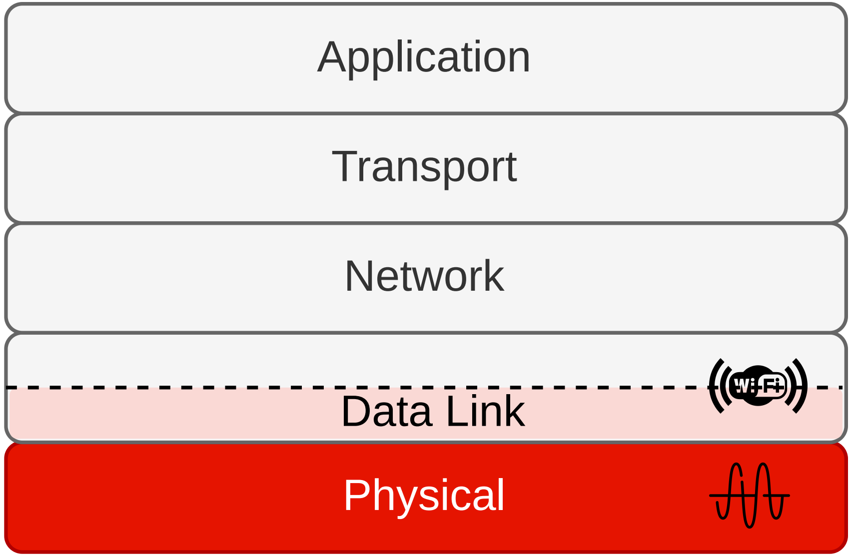
Andrew Tanenbaum, *Computer networks (5th edition)*

- Chapter 7 - *Application Layer*
  - Subchapter 7.1 - *DNS - Domain Name System*

Raspberry Pi Pico W

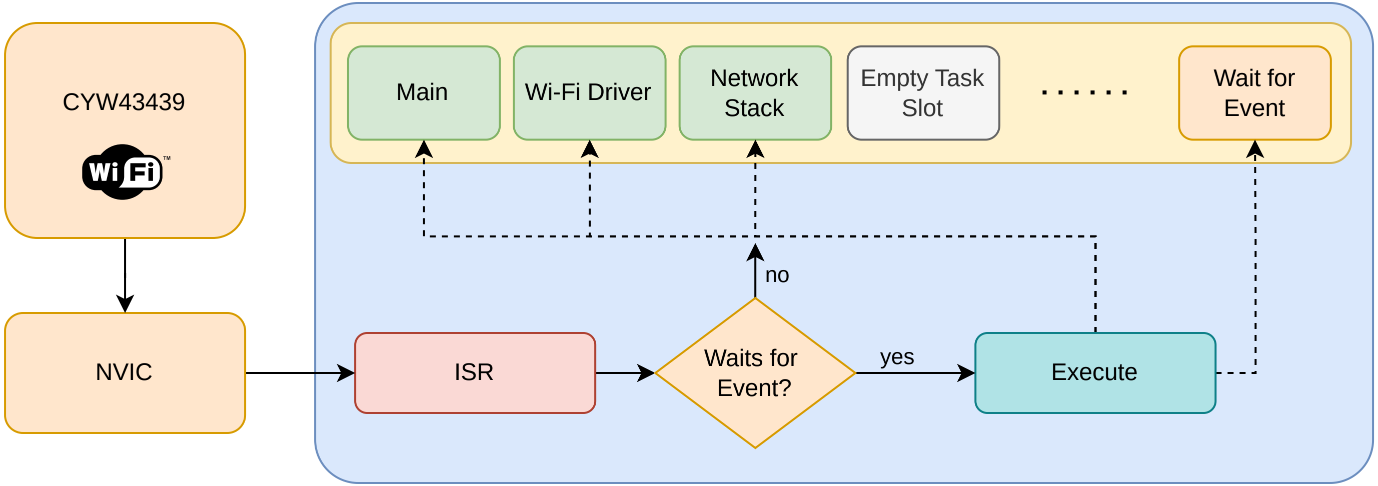
uses a discrete Wi-Fi chip

- Wi-Fi and BLE provided by **CYW43439** made by *Infineon*
- connected over SPI/PIO
- Wi-Fi 4 (802.11n), 2.4 GHz
  - WPA 3
  - SoftAP (4 clients)
  - Device
- BLE 5.2
  - Central
  - Peripheral
  - Bluetooth Classic
- Provides *L2* - allows sending of *Ethernet* (*MAC*) frames



Tasks

tasks that run when using Wi-Fi



CYW43439 API

the embassy driver

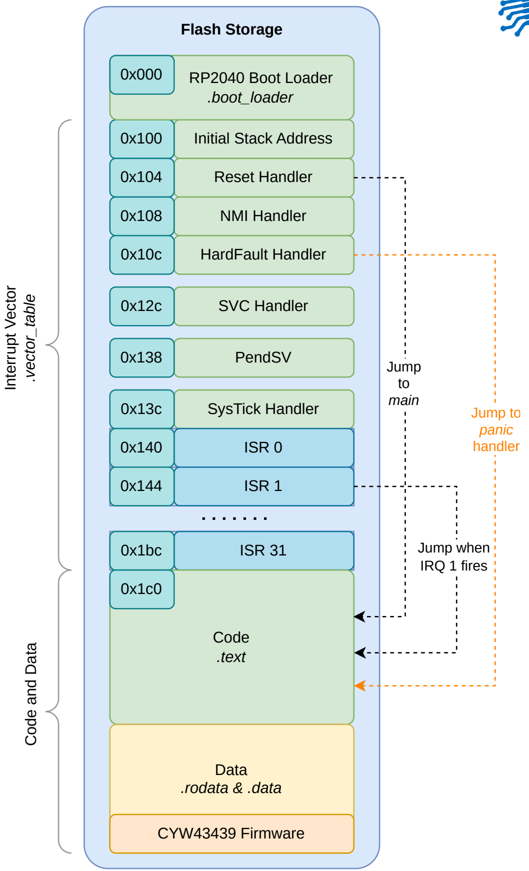
1. Load the *firmware* into the `.data` section.

```
let fw = include_bytes!("./cyw43439_firmware/43439A0.bin");
let clm = include_bytes!("./cyw43439_firmware/43439A0_clm.bin");
```

2. Use PIO0 as SPI device

```
bind_interrupts!(struct Irqs {
    PIO0_IRQ_0 => InterruptHandler<PIO0>;
});

let pwr = Output::new(p.PIN_23, Level::Low);
let cs = Output::new(p.PIN_25, Level::High);
let mut pio = Pio::new(p.PIO0, Irqs);
let spi = PioSpi::new(
    &mut pio.common, pio.sm0, pio.irq0,
    cs, p.PIN_24, p.PIN_29, p.DMA_CH0
);
```



* drawing is not at scale, code and data are significantly greater than the interrupt vector

CYW43439 API

the embassy driver

3. Write a task for the Wi-Fi driver

```
#[embassy_executor::task]
async fn wifi_task(runner: cyw43::Runner<'static, Output<'static>, PioSpi<'static, PIO0, 0, DMA_CH0>>) -> ! {
    runner.run().await
}
```

4. Start the driver

```
static STATE: StaticCell<cyw43::State> = StaticCell::new();
let state = STATE.init(cyw43::State::new());
let (_net_device, mut control, runner) = cyw43::new(state, pwr, spi, fw).await;
unwrap!(spawner.spawn(wifi_task(runner)));
```

5. Init the device

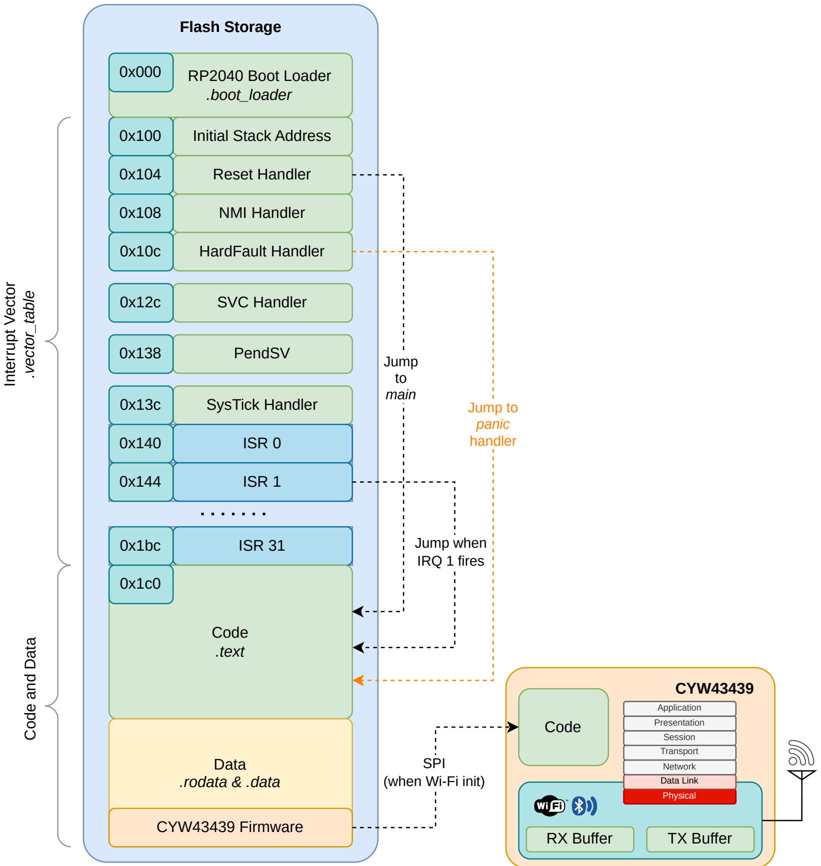
```
control.init(clm).await;
control
    .set_power_management(PowerManagementMode::PowerSave)
    .await;
```

Flash the firmware

write the firmware to the wifi device

The first action of the *wifi* task is to write the firmware from *.data* to the CY43439 chip.

```
#[embassy_executor::task]
async fn wifi_task(/* ... */) -> ! {
    runner.run().await
}
```



* drawing is not at scale, code and data are significantly greater than the interrupt vector

Wi-Fi AP Mode

Start an **AP** and allow other devices to connect.

Open Network (not a very good idea)

- network SSID
- channel number

```
control.start_ap_open("Network SSID", 5).await;
```

WPA network

- network SSID
- WPA password
- channel number

```
control.start_ap_wpa2("Network SSID", "WPA password", 5).await;
```

Wi-Fi Device Mode

Start an **device** and connect to a Wi-Fi network

Open Network (not a very good idea)

- network SSID

```
control.join_open("network SSID").await;
```

WPA network

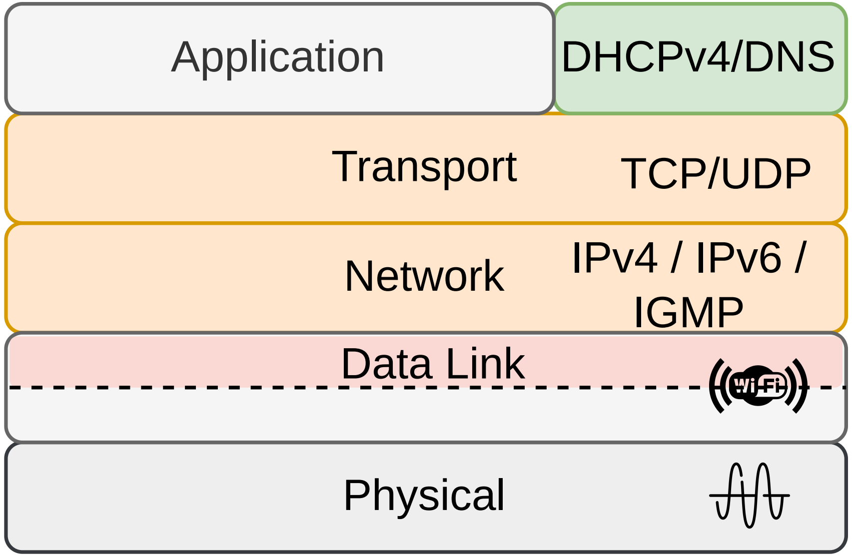
- network SSID
- network password

```
match control.join_wpa2("network ssid", "network password").await {
    Ok(_) => break,
    Err(err) => {
        info!("join failed with status={}", err.status);
    }
}
```

Embassy Net

a smol TCP/IP stack

- uses *smoltcp*, embedded (no_std) TCP/IP stack written in Rust
- *L3*: IPv4, IPv6, IGMPv4 (ping), 6LoWPAN
- *L4*: TCP and UDP
- *L7*: DHCPv4 and DNS



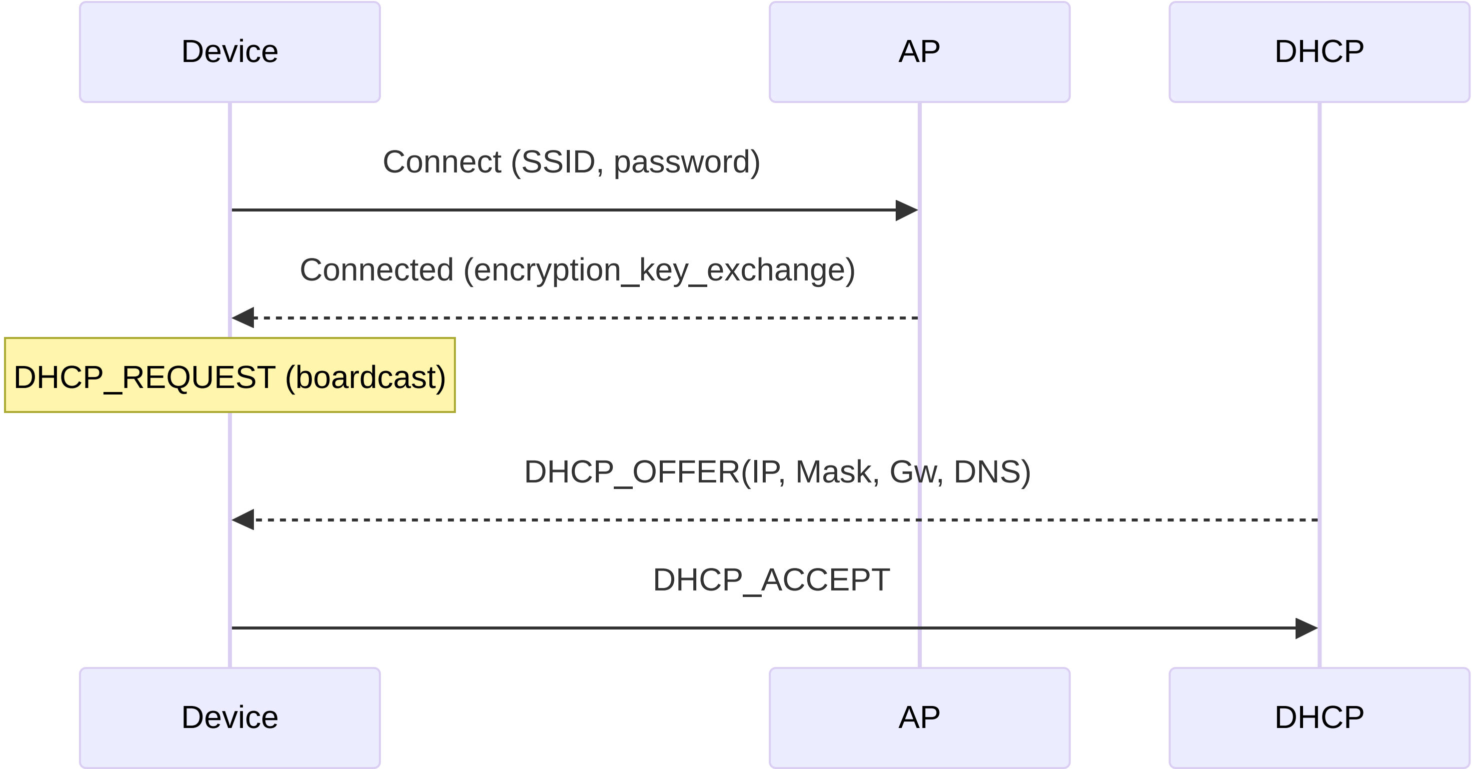
Embassy Net API

over smoltcp

1. Set how to obtain an IP address
   - self assigned
   - DHCP
2. Start the network stack
3. Use sockets to communicate

DHCP

Dynamic Host Control Protocol



Obtain a network address

self assigned or obtain one from a DHCP server

Self assigned

```rust
let config = embassy_net::Config::ipv4_static(embassy_net::StaticConfigV4 {
    address: Ipv4Cidr::new(Ipv4Address::new(192, 168, 69, 2), 24),
    dns_servers: vec![Ipv4Address::new(8, 8, 8, 8), Ipv4Address::new(1, 1, 1, 1)],
    gateway: Some(Ipv4Address::new(192, 168, 69, 1)),
});
```

Dynamic Host Control Protocol (DHCP)

```rust
let config = Config::dhcpv4(Default::default());

// start the network stack

// Wait for DHCP
info!("waiting for DHCP...");
while !stack.is_config_up() {
    Timer::after_millis(100).await;
}
```

Start the network stack

1. Write a network task

```
#[embassy_executor::task]
async fn net_task(stack: &'static Stack<cyw43::NetDriver<'static>>) -> ! {
    stack.run().await
}
```

2. Start the network stack

```
// chosen by fair dice roll. guaranteed to be random.
let seed = 0x0123_4567_89ab_cdef;

// Init network stack
static STACK: StaticCell<Stack<cyw43::NetDriver<'static>>> = StaticCell::new();
static RESOURCES: StaticCell<StackResources<2>> = StaticCell::new();
let stack = &*STACK.init(Stack::new(
    net_device,
    config,
    RESOURCES.init(StackResources::<2>::new()),
    seed,
));

unwrap!(spawner.spawn(net_task(stack)));
```

Query an IP address using DNS

IP address for a domain

- sockets use IP addresses
- to talk to a server, the IP of the server has to be obtained

```rust
let dns = DnsSocket::new(stack);
match dns.get_host_by_name("www.example.com", AddrType::IPv4) {
    Ok(ip) => info!("Ip is {:?}", ip),
    Err(e) => warn!("failed to retrieve address {:?}", e)
}
```

TCP Server Socket

listening for one single connection

smoltcp can only listen and accept one client

```
let mut rx_buffer = [0; 4096];
let mut tx_buffer = [0; 4096];

loop {
    let mut socket = TcpSocket::new(stack, &mut rx_buffer, &mut tx_buffer);
    socket.set_timeout(Some(Duration::from_secs(10)));

    info!("Listening on TCP:1234...");
    if let Err(e) = socket.accept(1234).await {
        warn!("accept error: {:?}", e);
        continue;
    }

    info!("Received connection from {:?}", socket.remote_endpoint());

    // handle the connection
}
```

TCP Client Socket

connecting to a server

```rust
let mut rx_buffer = [0; 4096];
let mut tx_buffer = [0; 4096];

loop {
    let mut socket = TcpSocket::new(stack, &mut rx_buffer, &mut tx_buffer);
    socket.set_timeout(Some(Duration::from_secs(10)));

    info!("Connecting to TCP 1.2.3.5:1234...");
    if let Err(e) = socket.connect(IpEndpoint::new(IpAddress::v4(1,2,3,5), 1234)).await {
        warn!("accept error: {:?}", e);
        continue;
    }

    info!("Connected to {:?}", socket.remote_endpoint());

    // handle the connection
}
```

Read from a TCP Socket

read bytes

```
let mut buf = [0u8; 4096];

let n = match socket.read(&mut buf).await {
    Ok(0) => {
        warn!("read EOF");
        break;
    }
    Ok(n) => n,
    Err(e) => {
        warn!("read error: {:?}", e);
        break;
    }
};

// display bytes as a UTF-8 string
info!("rxd {}", from_utf8(&buf[..n]).unwrap());
```

Write to a TCP Socket

write bytes

```rust
let buf = [0u8; 4096];

// write n bytes to buf

match socket.write_all(&buf[..n]).await {
    Ok(()) => {
        /* write was successful */
    }
    Err(e) => {
        warn!("write error: {:?}", e);
        break;
    }
};
```

Listen for UDP Packets

```
let mut rx_buffer = [0; 4096];
let mut rx_metdata_buffer = [PacketMetadata::EMPTY; 3];
let mut tx_buffer = [0; 4096];
let mut tx_metadata_buffer = [PacketMetadata::EMPTY; 3];

let mut buf = [0u8; 4096];

let mut socket = UdpSocket::new(
    stack, &mut rx_buffer, &mut rx_metadata_buffer, &mut tx_buffer, &mut tx_metadata_buffer
);

// listen for UDP packet on port 1234
if let Err(e) = socket.bind(1234) {
    warn!("bind error {:?}", e);
    break;
}

loop {
    match socket.recv_from(&mut buf) {
        Ok((n, endpoint)) => {
            info!("Received from {:?}: {:?}", endpoint, buf[..n]);
        }
    }
}
```

Send UDP Packets

```rust
let mut rx_buffer = [0; 4096];
let mut rx_metdata_buffer = [PacketMetadata::EMPTY; 3];
let mut tx_buffer = [0; 4096];
let mut tx_metadata_buffer = [PacketMetadata::EMPTY; 3];

let mut buf = [0u8; 4096];

let mut socket = UdpSocket::new(
    stack, &mut rx_buffer, &mut rx_metadata_buffer, &mut tx_buffer, &mut tx_metadata_buffer
);

info!("Sending to UDP 1.2.3.5:1234...");
match socket.send_to(&buf, IpEndpoint::new(IpAddress::v4(1,2,3,5), 1234)) {
    Ok(()) => {
        /* send successful */
    }
    Err(e) => {
        warn!("send error: {:?}", e);
    }
}
```

Protocols

Libraries

that provide protocols

- **MQTT** - *MQ Telemetry Transport*
  - publish/subscribe
  - minimq
- **CoAP** - *Constrained Application Protocol*
  - simplified binary HTTP
  - coap_lite

Conclusion

we talked about

- OSI Network Stack
- Wi-Fi
- TCP/IP
- Raspberry Pi W
- Protocols