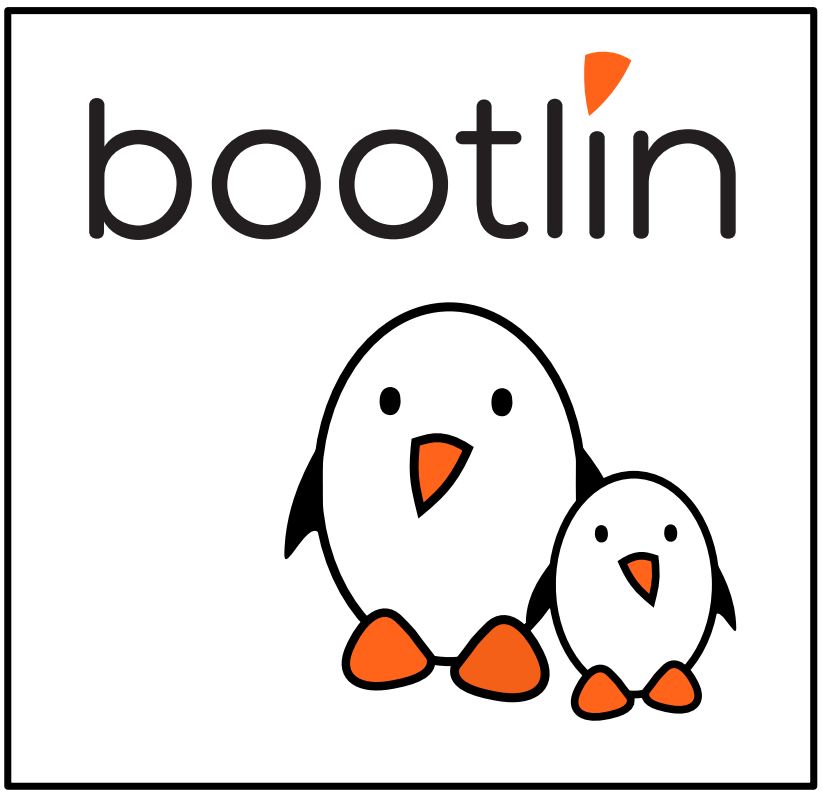
Unpacking the Linux WiFi stack: Writing and integrating wireless drivers

Alexis Lothoré
alexis.lothore@bootlin.com

Embedded Linux Conference Europe 2025

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!



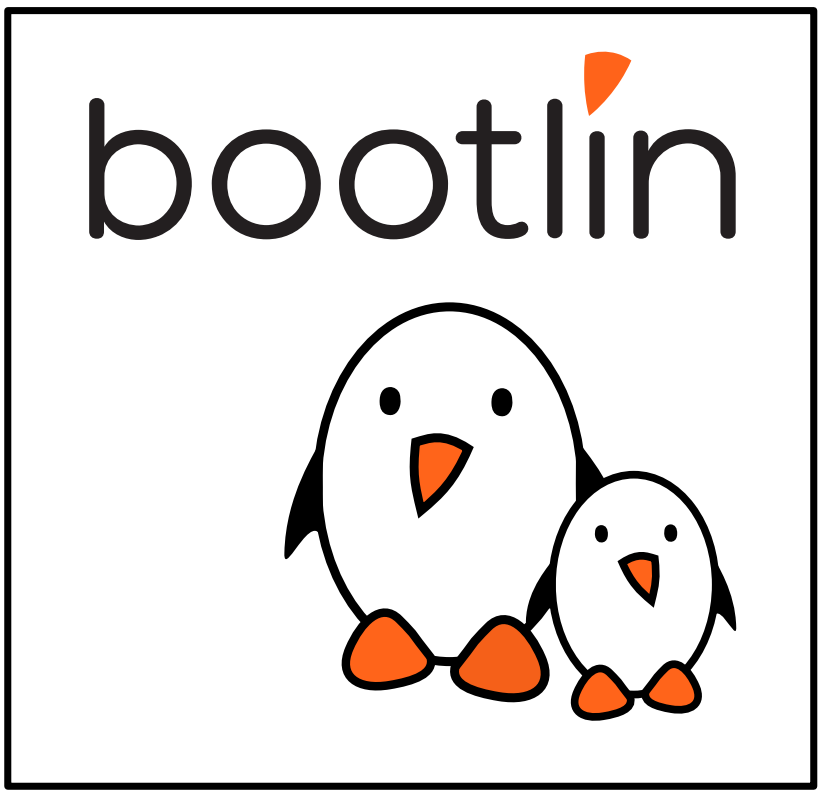
Alexis Lothoré

- Embedded Linux engineer and trainer at Bootlin since 2023
  - Expertise in Embedded Linux
  - Development, consulting and training
  - Strong open-source focus
- Working on embedded systems since 2016
- BSP, device drivers, networking, **wireless**, CI, eBPF
  - Microchip WILC1000
  - Microchip WILC3000
- Teaching training courses
- Living in Toulouse, south of France
- alexis.lothore@bootlin.com

Agenda

- 802.11 basics
- Linux wireless stack
- Implementing wireless drivers
- Userspace tools / testing
- Debugging

802.11 basics

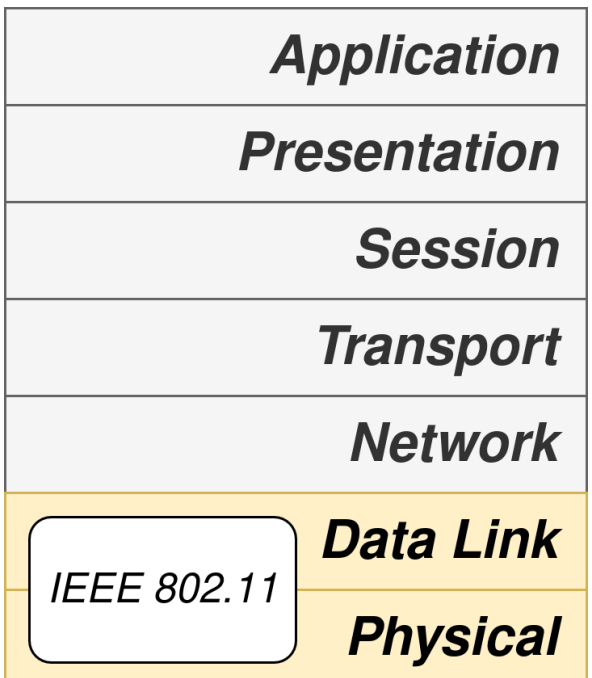


802.11

- "Wi-Fi" is a marketing name
- IEEE standard, first released in 1997, with many revisions and amendments:
  - 802.11b: 11Mb/s
  - 802.11a: 5Ghz band, 54Mb/s
  - 802.11g: Data Rate Extension in 2.4Ghz
  - 802.11n ("Wi-Fi 4"): MIMO, 72 Mb/s, 600 Mb/s
  - 802.11ac ("Wi-Fi 5"): Data Rate Extension in 5GHz, MU-MIMO
  - 802.11ax ("Wi-Fi 6/6E"): new 6GHz band, higher data rates
  - 802.11be ("Wi-Fi 7"): MLO, higher data rates
  - ...
- IEEE task group aims to release Wi-Fi 8 specification (802.11bn) in 2028

802.11 layers

- PHY layer
  - different modulations, depending on the version (mostly OFDM for modern devices)
  - different frequency operating bands (2.4GHz, 5GHz, 6GHz ...)
- MAC layer
  - CSMA-CA
  - 3 types of "frames": Management, Control, Data
  - scanning, authentication, association, network maintenance, security, power saving...

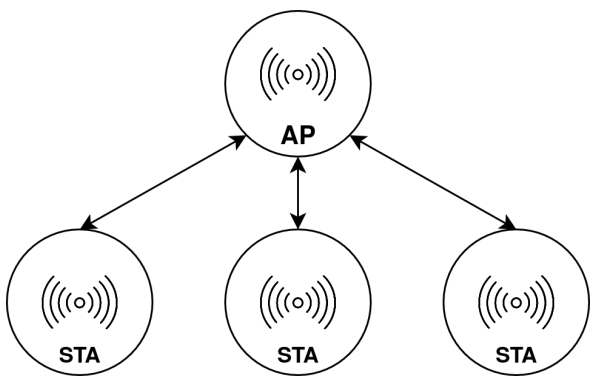


802.11 place in the OSI model

802.11 networks

- Infrastructure (*BSS*): one or many **stations** (STA) connected to a special station called **Access Point** (AP)

- Other types of networks available:
  - Ad Hoc (*IBSS*): no Access Point. Each station can only communicate with direct neighbors
  - Mesh (*802.11s*): multi-hop networks
  - ...



A simple infrastructure network

The hardware

- Wide variety of platforms:
  - Supporting different 802.11 standards
  - exposed through different buses: PCI, USB, SDIO, uart...
  - sometimes with multiple features: e.g. WLAN/BT
  - generally depends on a firmware to operate

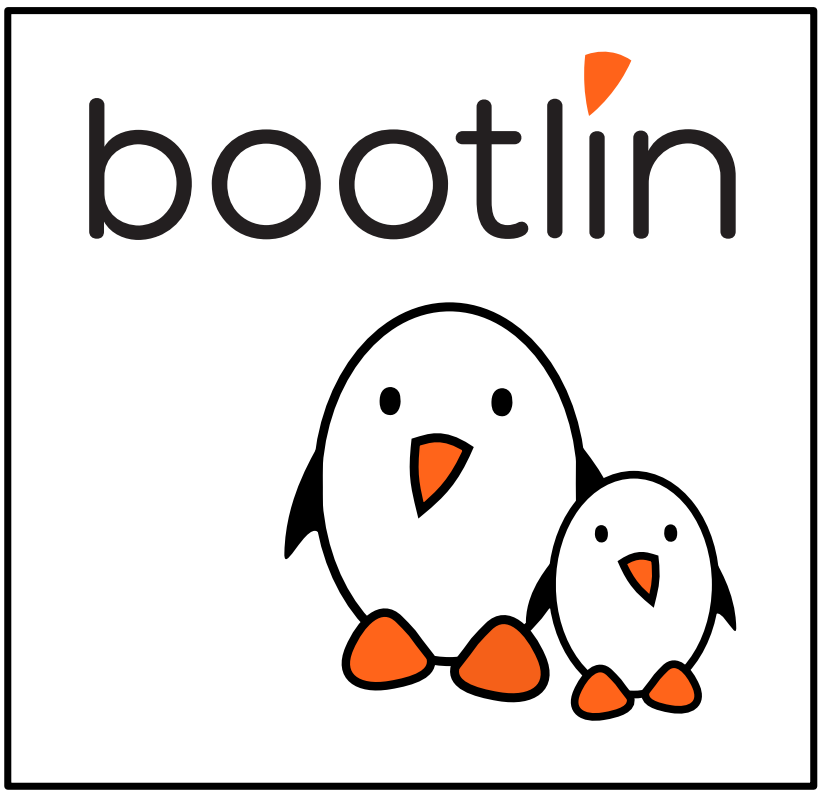
A TP-Link PCI card

A Dlink USB dongle

A Microchip module

- Identifying some exact hardware/upstream support is sometimes a challenge !
  - Ezurio LWB5+ -> Laird LWB5+ -> Infineon Airoc CYW43439
    - former Cypress -> former Broadcom -> brcmfmac

Linux Wireless Stack

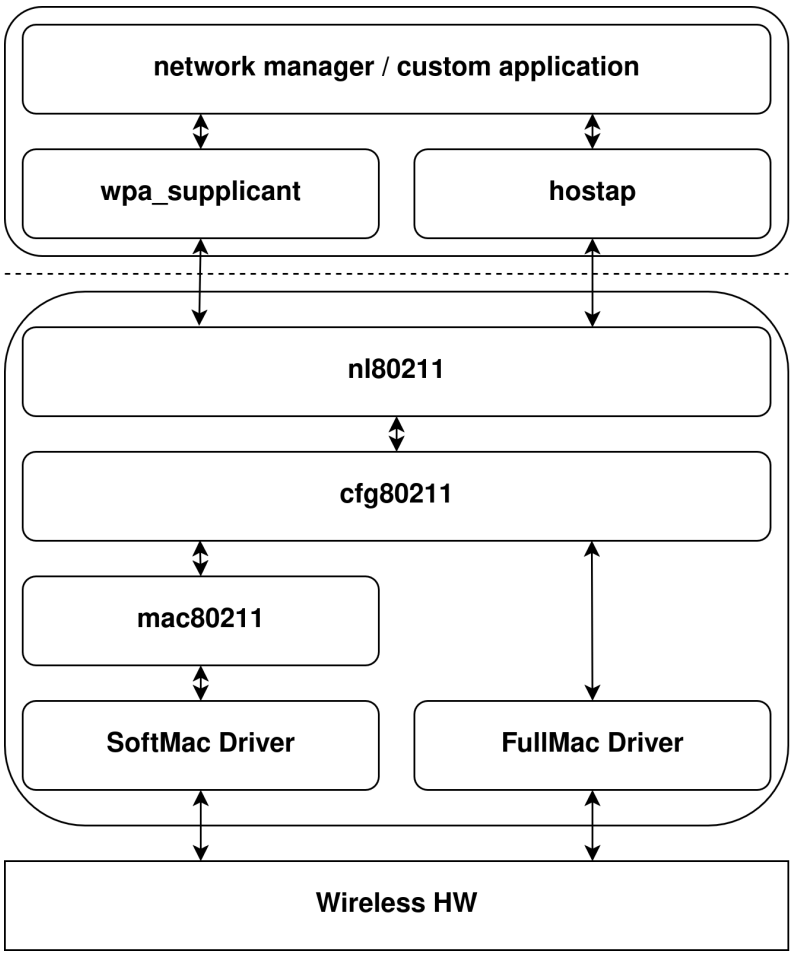


Full MAC / Soft MAC

Not all chips/drivers handle the same amount of features:

- **FullMAC** devices handle both PHY and MAC
  - The MAC layer is handled by the device firmware
  - [+] performance level may be higher
  - [−] hardware is more complex/expensive
  - [−] if there is a bug in the mac layer, it is harder to identify and fix
- **SoftMAC** devices only handle the PHY part, the MAC part is handled by the kernel:
  - [+] simpler/cheaper hardware
  - [+] all softmac devices benefit from 802.11 MAC layer improvements and fixes
  - [−] the overall wireless performance may be CPU bound.
- ~35 SoftMAC drivers and ~9 FullMAC drivers upstream
  - see `drivers/net/wireless`

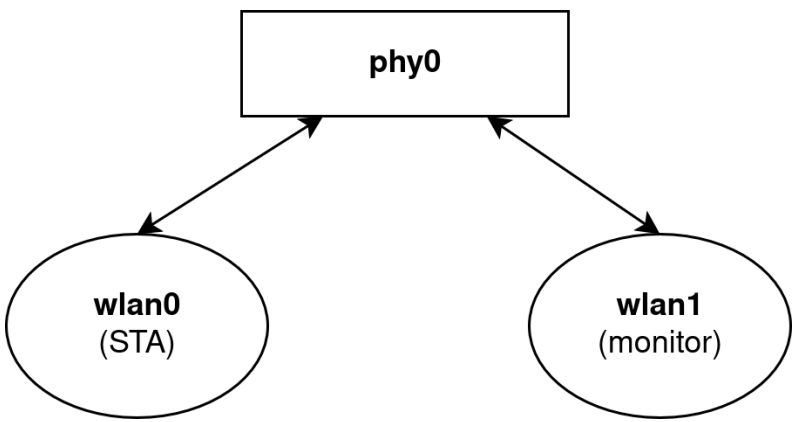
The wireless stack in Linux



- **cfg80211** is the core subsystem handling wireless configuration
  - interacts directly with fullmac drivers
  - or goes through **mac80211** for softmac drivers
- **mac80211** layer is a software implementation of IEEE80211 MAC
  - frames crafting and parsing
  - encryption/decryption
  - queues management
  - rate control
  - multiple state machines implementing IEEE 802.11 MAC
- userspace interacts with cfg80211 through **nl80211** command and events

The wireless phy and virtual interfaces

- The hardware is represented by a **wiphy**, on top of which multiple **virtual interfaces** (*VIFs*) can coexist concurrently
- concurrent VIF combinations depend on the driver, and on the hardware/firmware for FullMac
- you generally want/have at least one STA VIF automatically created
- additional VIFs can be created from userspace (e.g with `iw`)



A single wiphy with two VIFs

Implementing wireless drivers

The `struct wiphy`

- Represents the physical wireless device and its capabilities
- Drivers need to **allocate** and **configure** a `struct wiphy` for each wireless device:
  - supported modes ? (STA, AP, monitor, P2P...)
  - supported bands and channels ?
  - non-standard capabilities and constraints ?
  - supported concurrent interfaces combinations ?

```
enum nl80211_iftype {
  NL80211_IFTYPE_UNSPECIFIED,
  NL80211_IFTYPE_ADHOC,
  NL80211_IFTYPE_STATION,
  NL80211_IFTYPE_AP,
  NL80211_IFTYPE_AP_VLAN,
  NL80211_IFTYPE_WDS,
  NL80211_IFTYPE_MONITOR,
  NL80211_IFTYPE_MESH_POINT,
  NL80211_IFTYPE_P2P_CLIENT,
  NL80211_IFTYPE_P2P_GO,
  NL80211_IFTYPE_P2P_DEVICE,
  NL80211_IFTYPE_OCB,
  NL80211_IFTYPE_NAN,
  [...]
};
```

SoftMac drivers

SoftMAC drivers use the **mac80211** kernel API, exposed by `include/net/mac80211.h`

- `ieee80211_alloc_hw`:
  - consumes a `struct ieee80211_ops`
  - allocates a `struct ieee80211_hw`
- the `struct ieee80211_hw` contains members to be configured by the driver:
  - a `struct wiphy`
  - an additional *flags* fields describing driver capabilities or offloaded features
- `ieee80211_register_hw`: registers the wireless interface
  - automatically creates a STA net device (if relevant)

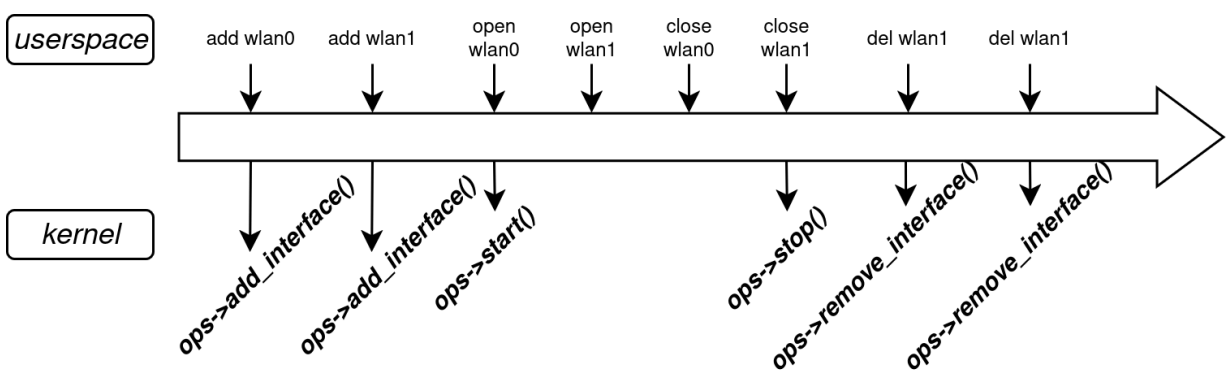
The `struct ieee80211_ops`

- Contains the basic driver ops that mac80211 layer will call
- Checked by *mac80211* at allocation time
- Plenty of ops, with a minimal mandatory set:

```
int   (*start)(struct ieee80211_hw *hw);
void  (*stop)(struct ieee80211_hw *hw, bool suspend);
int   (*add_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
void  (*remove_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
void  (*tx)(struct ieee80211_hw *hw, struct ieee80211_tx_control *control,
            struct sk_buff *skb);
void  (*wake_tx_queue)(struct ieee80211_hw *hw, struct ieee80211_txq *txq);
int   (*config)(struct ieee80211_hw *hw, u32 changed);
void  (*configure_filter)(struct ieee80211_hw *hw, unsigned int changed_flags,
                          unsigned int *total_flags, u64 multicast);
```

The `struct ieee80211_ops`

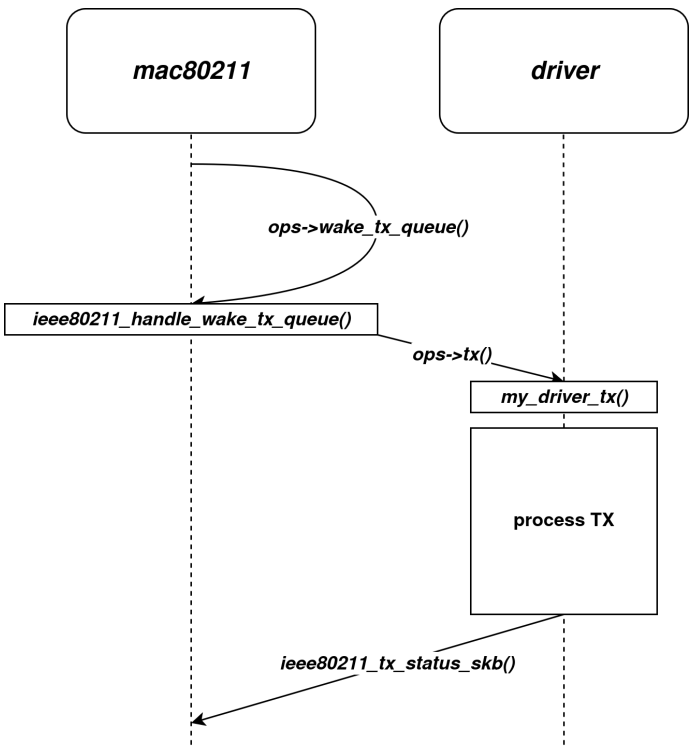
- `add_interface`/`remove_interface`
  - called when a virtual interface creation or deletion is requested
  - this is really about **vif** init and deinit
- `start`/`stop`
  - called before first vif is enabled / after last vif is disabled
  - this is really about **wireless hardware** init/deinit
  - you must perform all needed initialization needed to make the hardware able to run



mac80211 hw/vif management

The `struct ieee80211_ops`

- `tx`
  - mac80211 "pushes" a `struct sk_buff` to the driver
  - the driver must call `ieee80211_tx_status_skb` once TX is done, or `ieee80211_free_txskb` if status can't get deduced (e.g. TX failure)
- `wake_tx_queue`
  - notification to ask driver to "pull" packets from mac80211
    - use `ieee80211_tx_dequeue` to pull `struct sk_buff`
    - optionally, use `ieee80211_next_txq` to let mac80211 balance TX queues
  - drivers can use `ieee80211_handle_wake_tx_queue` as a default implementation: mac80211 will then call the `tx` ops



mac80211 driver simplified TX path

The `struct ieee80211_ops`

- `config`
  - called when hardware needs to be reconfigured: monitor flag set, device is now idle, channel change requested, etc
  - receives a `enum ieee80211_conf_changed` bitfield defining what should be reconfigured
  - also called on first interface being enabled
- `configure_filter`: ask driver/hw to configure the RX frame filter, ie additional frames to be passed to mac 80211
  - useful for example for the monitor mode to see frames that would otherwise be handled directly by the hardware/the driver

RX path

- multiple APIs to pass received frames to mac80211:
  - `ieee80211_rx`: the default RX callback, passes a single `struct sk_buff`
  - `ieee80211_rx_list`: passes a list of SKBs to mac80211, they are processed but not passed yet to the stack, drivers need to call `netif_receive_skb_list`
  - `ieee80211_rx_napi`: to be used if your driver using NAPI to handle RX
  - `ieee80211_rx_ni`: when RX is done in process context (e.g. workqueue)
  - `ieee80211_rx_irqsafe`: when RX is done in hard interrupt context
- Those callbacks expect a 802.11 header in front of passed SKBs

SoftMac drivers

```
struct ieee80211_ops my_ieee80211_ops {
  .start =              my_start,
  .stop =               my_stop,
  .add_interface =      my_add_interface,
  .remove_interface =   my_remove_interface,
  .config =             my_config,
  .tx =                 my_tx,
  .wake_tx_queue =      my_wake_tx_queue,
  .configure_filter =   my_configure_filter
};
```

```
int my_driver_probe(struct pci_device *pdev)
{
  struct ieee80211_hw *hw;
  [...]
  priv->rx_wq = alloc_workqueue("rx wq", WQ_BH, 0);
  INIT_WORK(&priv->rx_work, my_rx_work_handler);
  ret = request_irq(pdev->irq, irq_handler, IRQF_SHARED,
                    "my_device", NULL);
  [...]
  hw = ieee80211_alloc_hw(sizeof(struct my_priv),
                          my_ieee80211_ops);
  SET_IEEE80211_DEV(hw, &pdev->dev);
  SET_IEEE80211_PERM_ADDR(hw, mac_addr);
  ieee80211_hw_set(hw, HAS_RATE_CONTROL);
  ieee80211_hw_set(hw, SUPPORT_PS);
  ieee80211_hw_set(hw, SIGNAL_DBM);
  hw->wiphy->interface_modes = BIT(NL80211_IFTYPE_STATION) |
                               BIT(NL80211_IFTYPE_AP);
  hw->wiphy->bands[NL80211_BAND_2GHZ] = &my_supported_bands;
  [...]
  ret = ieee80211_register_hw(hw);
}
```

SoftMac drivers

```
static irqreturn_t irq_handler(int irq, void *arg)
{
  [...]
  queue_work(priv->rx_wq, priv->rx_work);
  [...]
}

static void my_rx_work_handler(struct work_struct *work)
{
  struct my_priv *priv = container_of(work, struct my_priv, rx_work);
  char buffer[MAX_RAW_DATA_LEN];
  struct sk_buff *skb;

  read_packet(priv, &buffer);
  // mac80211 expects a 802.11 header in front of the SKB
  skb = prepare_80211_skb(buffer)
  [...]
  ieee80211_rx(priv->hw, skb);
}
```

FullMac drivers

- the driver must handle many aspects that were managed by mac80211:
  - `struct ieee80211_ops` -> `struct cfg80211_ops`
  - `struct ieee80211_hw` -> bare `struct wiphy`
  - `ieee80211_alloc_hw` -> `struct wiphy_new`
  - the core manipulates VIFs through `struct wireless_dev`
- the driver must allocate and register a `struct net_device` to get a default interface
  - net devices and wireless dev are linked through the `ieee80211_ptr` field in the net device structure

FullMac drivers

- `struct cfg80211_ops` exposes a lot of ops (128 !), but no strict mandatory list
- As a starter, for STA:

```
struct wireless_dev * (*add_virtual_intf)(struct wiphy *wiphy, const char *name,
                                          unsigned char name_assign_type, enum nl80211_iftype type,
                                          struct vif_params *params);
int (*del_virtual_intf)(struct wiphy *wiphy, struct wireless_dev *wdev);
int (*scan)(struct wiphy *wiphy, struct cfg80211_scan_request *request);
int (*connect)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_connect_params *sme);
int (*disconnect)(struct wiphy *wiphy, struct net_device *dev, u16 reason_code);
int (*add_key)(struct wiphy *wiphy, struct net_device *netdev, int link_id, u8 key_index, bool pairwise,
               const u8 *mac_addr, struct key_params *params);
int (*del_key)(struct wiphy *wiphy, struct net_device *netdev, int link_id, u8 key_index, bool pairwise,
               const u8 *mac_addr);
int (*set_default_key)(struct wiphy *wiphy, struct net_device *netdev, int link_id, u8 key_index,
                       bool unicast, bool multicast);
```

FullMac drivers

- The driver needs to implement a `struct net_device_ops`

```
int             (*ndo_open)(struct net_device *dev);
int             (*ndo_stop)(struct net_device *dev);
netdev_tx_t     (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

- Received frames are passed to the network stack through `netif_rx` or `napi_gro_receive`
  - exception: userspace can register to specific 802.11 frames, those should be passed with `cfg80211_rx_mgmt`

Firmware management

- Wireless devices need some firmware to operate
- Generally published and hosted in the `linux-firmware` repository
- Drivers fetch the needed firmware through the `request_firmware` API
- Firmwares can be stored in different places:
  - in the root filesystem/an initramfs (*/lib/firmware*)
  - embedded directly in the kernel image (less common, and less convenient)
- Drivers are responsible of loading and starting the firmware

Security / key management

- the linux wireless stack only handles basic connection modes:
  - open
  - shared key (WEP)
- standard connection methods (WPA2, WPA3) are deferred to userspace (e.g. wpa_supplicant)
  - the supplicant then handles the authent/assoc/handshake state machine through nl80211 commands and events
- some features may be offloaded to the firmware (stated in `wiphy->ext_features`):
  - `NL80211_EXT_FEATURE_4WAY_HANDSHAKE_STA_PSK`: is able to handle the WPA handshake when set as STA
  - `NL80211_EXT_FEATURE_4WAY_HANDSHAKE_STA_1X`: is able to handle 802.1x handshake
  - `NL80211_EXT_FEATURE_4WAY_HANDSHAKE_AP_PSK`: is able to handle the WPA handshake when set as AP

Regulatory

- wireless devices must follow per-country regulatory rules:
  - usable radio bands
  - max TX power
  - misc constraints
- the official source of regulatory domains rules for linux is `wireless_regdb`

```
country NL: DFS-ETSI
  (2400 - 2483.5 @ 40), (100 mW)
  (5150 - 5250 @ 80), (200 mW), NO-OUTDOOR, AUTO-BW, wmmrule=ETSI
  (5250 - 5350 @ 80), (100 mW), NO-OUTDOOR, DFS, AUTO-BW, wmmrule=ETSI
  (5470 - 5725 @ 160), (500 mW), DFS, wmmrule=ETSI
  # short range devices (ETSI EN 300 440-1)
  (5725 - 5875 @ 80), (25 mW)
  # WiFi 6E
  (5945 - 6425 @ 320), (23), NO-OUTDOOR, wmmrule=ETSI
  # 60 GHz band channels 1-4 (ETSI EN 302 567)
  (57000 - 66000 @ 2160), (40)
```

Regulatory

- cfg80211 requests regulatory database when cfg80211 is initialized, through `reg_query_database`
  - expects a *regulatory.db* file installed in */lib/firmware*
- userspace can initiate a change in regulatory domain:

```
iw reg set NL
```

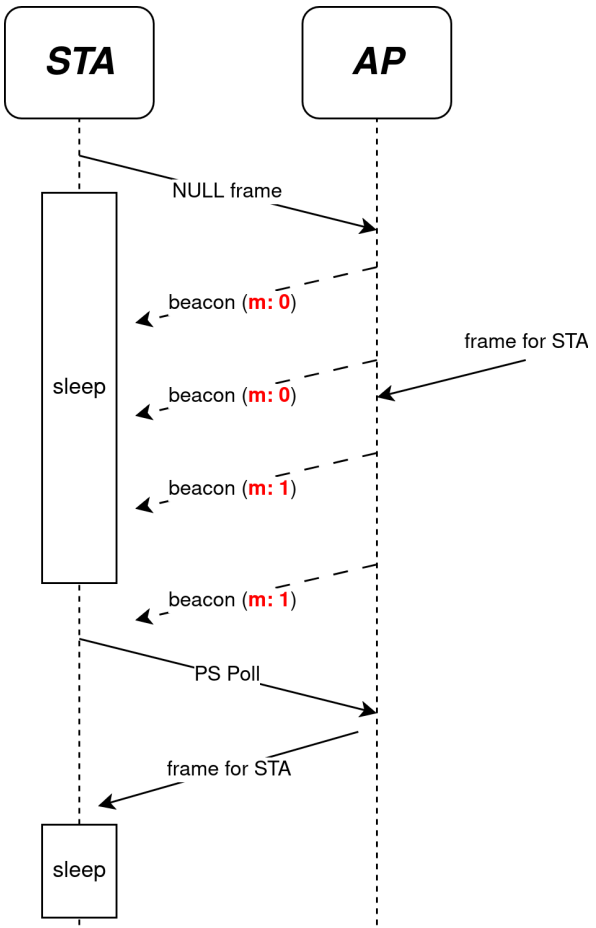
- kernel will then enforce the corresponding rules on each wiphy
- by default the kernel apply the "world" regulatory to new devices until a specific regulatory domain is set

Regulatory

- drivers can be notified about updates with the `wiphy->reg_notifier` callback
- drivers can hint the kernel with a specific domain by issuing a `regulatory_hint` call (e.g. for the initial regulatory configuration)
- drivers can also enforce specific regulatory management, described through flags in the `struct wiphy`:
  - `REGULATORY_CUSTOM_REG`
  - `REGULATORY_STRICT_REG`
  - `REGULATORY_WIPHY_SELF_MANAGED`
- General focus (for both drivers and wireless core): **it must be impossible for the final user to unknowingly fail to comply with local regulations**.

Power save

- Battery-powered station devices can periodically sleep
  - STA sends a NULL frame, AP reacts by:
    - putting any pending messages into a buffer
    - setting a “pending messages” bit in beacon frames
  - Periodically, STA will wake, read beacons, and send a PS-Poll frame to ask for the pending messages
  - When leaving power save mode, station sends a new NULL frame with the updated power save status



STA entering power-save

Power save

- **SoftMAC** drivers set the `IEEE80211_HW_SUPPORTS_PS` flag to let mac80211 know that they support power save.
  - The driver/hardware must:
    - handle the NULL frames
    - or ask the mac80211 to handle those with `IEEE8021_HW_PS_NULLFUNC_STACK`
  - `IEEE80211_HW_SUPPORTS_DYNAMIC_PS`: the hardware can handle dynamic powersave
- **FullMAC** drivers directly set the `set_power_mgmt` callback in `struct cfg80211_ops`

Power save

- Userspace can toggle power save:

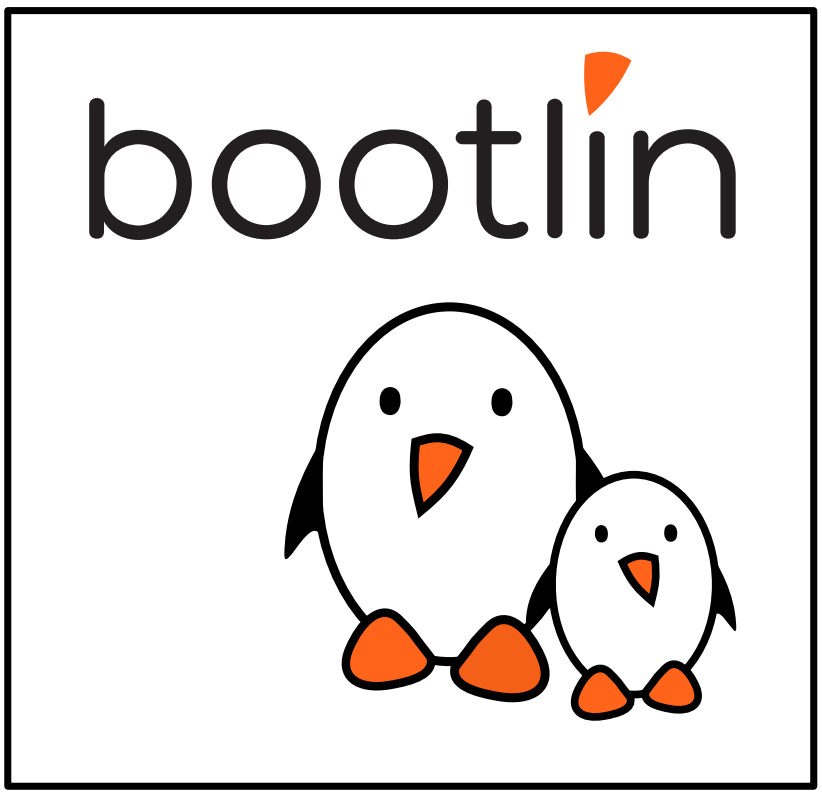
```
iw dev wlan set power_save on
```

- Caution: power save may be enabled by default !
  - if driver has set `WIPHY_FLAG_PS_ON_BY_DEFAULT`
  - if the kernel is built with `CONFIG_CFG80211_DEFAULT_PS`
- It is worth disabling it in development/when debugging a specific issue

Design tips

- hardware should stay off “as long as possible”
  - firmware should not run right at probe time
- init/register order matters: immediately after `wiphy_register`/ `ieee80211_register_hw`, the kernel can start calling your ops
- if you must support different versions of your device (eg: revisions, or busses), make sure to keep the core code separated from the
  - create bus files
  - create revision-specific files
- don’t be scared by the amount of things to implement
  - start small, you do not need to implement all the features
  - implement stubs for basic ops, so you can learn when they are called

Userspace tools/testing



iw

- The default userspace tool to interact with wireless devices
- https://git.kernel.org/pub/scm/linux/kernel/git/jberg/iw.git
- Uses the nl80211 layer to interact with the kernel
- CLI interface with various subcommands

iw

- list known devices and properties:

```
$ iw phy
```

- add a virtual interface on a specific phy

```
$ iw phy phy0 interface add wlan1 type managed
```

- start a scan

```
$ iw dev wlan1 scan
```

iw

- connect to an AP (works only with open or WEP networks)

```
iw dev wlan1 connect my_ssid 6926366642517d785936792458
```

- get connection status

```
$ iw dev wlan1 link
```

- monitor kernel userspace wireless messages

```
iw event
```

wpa_supplicant

- `wpa_supplicant` is the standard supplicant for Linux, supporting a wide range of security standards and algorithms
- provides:
  - the `wpa_supplicant` daemon which interacts with the kernel with nl80211 messages
  - `wpa_cli` as a command line tool to control `wpa_supplicant`
- alternative: `iwd`

wpa_supplicant

- prepare wpa_supplicant configuration

```
ctrl_interface=/var/run/wpa_supplicant
update_config=1
```

- start wpa_supplicant

```
$ wpa_supplicant -Dnl80211 -iwlan1 -c wpa_supplicant.conf
```

- use wpa_cli to connect (here: a WPA2 network)

```
$ wpa_cli
> scan
> scan_results
> add_network
> set_network 1 ssid my_ssid
> set_network 1 psk 5up3r53cr3t
> enable_network 1
> status
```

Debugging

Logs and tracepoints

- mac80211 and cfg80211 expose a lot of debug logs

```
$ echo "module cfg80211 +p" > /proc/dynamic_debug/control
$ echo "module mac80211 +p" > /proc/dynamic_debug/control
```

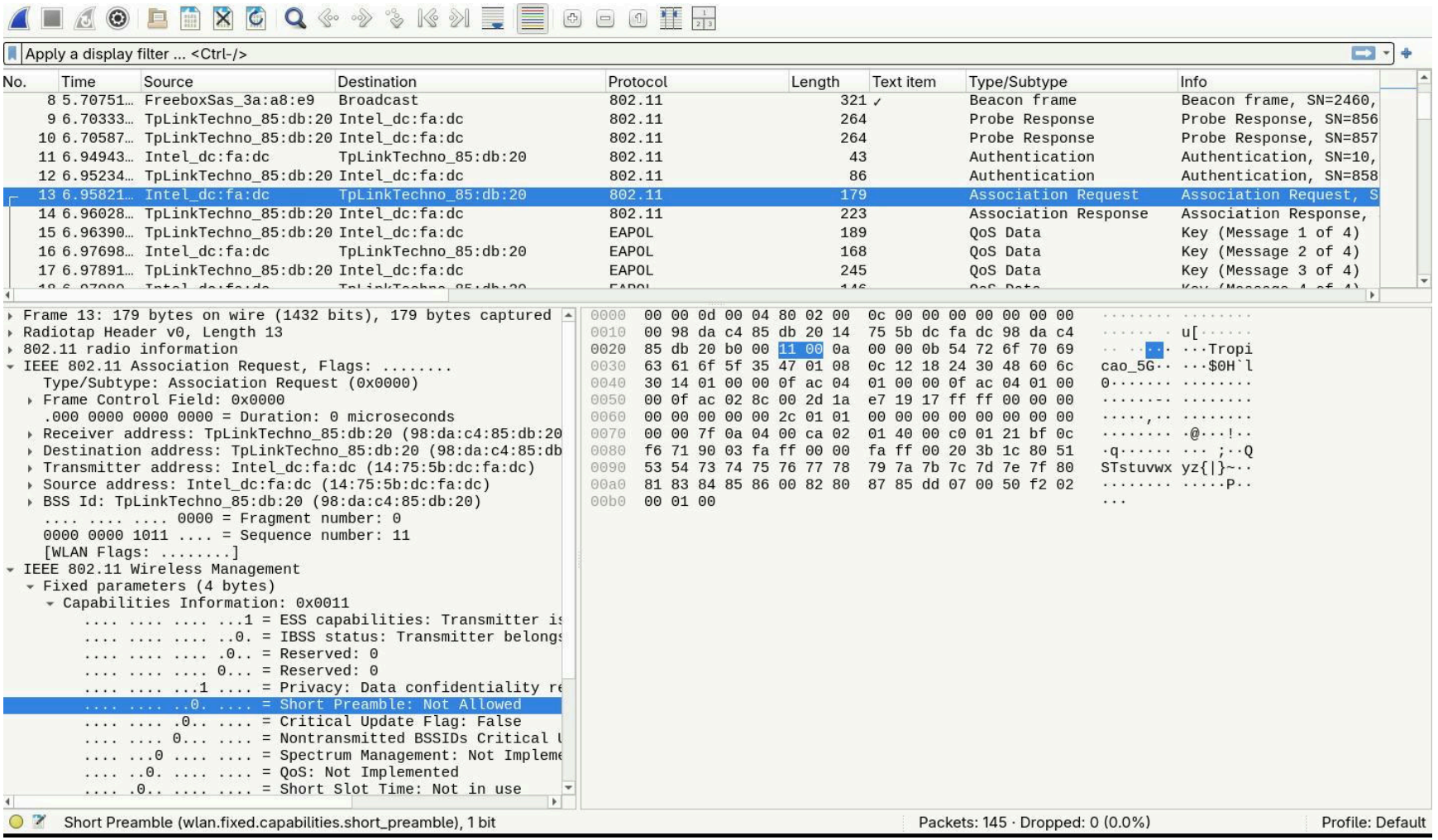
- they also come with plenty of tracepoints

```
$ trace-cmd record -e cfg80211 -e mac80211
  <C-c>
$ trace-cmd report
  wpa_supplicant-879   [005] ..... 21530.129640: drv_return_int:       phy6 - 0
  wpa_supplicant-879   [005] ..... 21530.129644: drv_vif_cfg_changed:  phy6 vif:wlp0s20f3(2) changed:0x4000
  wpa_supplicant-879   [005] ..... 21530.129647: drv_return_void:      phy6
  wpa_supplicant-879   [005] ..... 21530.129652: drv_link_info_changed: phy6 vif:wlp0s20f3(2) link_id:0, changed:0xe0
  wpa_supplicant-879   [005] ..... 21530.129980: drv_return_void:      phy6
  wpa_supplicant-879   [005] ..... 21530.129986: drv_sta_state:        phy6 vif:wlp0s20f3(2) sta:98:da:c4:85:db:20
state: 0->1
  wpa_supplicant-879   [005] ..... 21530.130494: drv_return_int:       phy6 - 0
  wpa_supplicant-879   [005] ..... 21530.130519: cfg80211_new_sta:     netdev:wlp0s20f3(13), 98:da:c4:85:db:20
  [...]
```

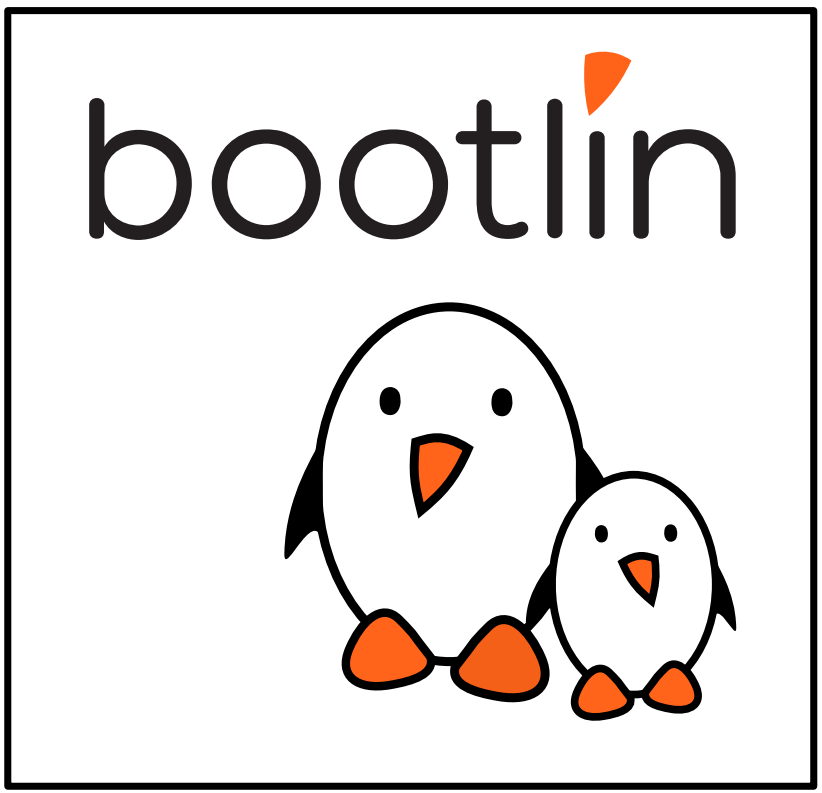
Tracing packets: monitor interface & tcpdump

```
$ iw phy phy0 interface add wlan1 type monitor
$ ip link set wlan1 up
$ iw dev wlan1 switch channel 40
$ tcpdump -i wlan1
23:08:37.396657 6.0 Mb/s [bit 15] Authentication (Open System)-1: Successful
23:08:37.399008 804764209us tsft 6.0 Mb/s 5200 MHz 11a -13dBm signal [bit 22] Authentication (Open System)-2:
23:08:37.400445 6.0 Mb/s [bit 15] Assoc Request (Tropicao_5G) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
23:08:37.402747 804767427us tsft 6.0 Mb/s 5200 MHz 11a -13dBm signal [bit 22] Assoc Response AID(2) :
PRIVACY : Successful
23:08:37.406459 804771545us tsft 6.0 Mb/s 5200 MHz 11a -13dBm signal [bit 22] EAPOL key (3) v2, len 95
23:08:37.415943 6.0 Mb/s [bit 15] EAPOL key (3) v1, len 117
23:08:37.417970 804783110us tsft 6.0 Mb/s 5200 MHz 11a -13dBm signal [bit 22] EAPOL key (3) v2, len 151
23:08:37.418584 6.0 Mb/s [bit 15] EAPOL key (3) v1, len 95
23:08:37.438645 804802933us tsft 6.0 Mb/s 5200 MHz 11a -16dBm signal [bit 22] Beacon (Tropicao_5G) [6.0* 9.0
12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS CH: 40, PRIVACY
23:08:37.456535 288410233326us tsft 6.0 Mb/s 5200 MHz 11a -13dBm signal [bit 22] Action (98:da:c4:85:db:20
(oui Unknown)): Reserved(21) Act#1
23:08:37.459150 [bit 15] Data IV:3aaaa Pad 0 KeyID 0
23:08:37.459610 [bit 15] Data IV:3aaaa Pad 0 KeyID 0
[...]
```

Tracing packets: wireshark



Some additional resources



Going further

- The official Linux Wireless documentation
- 802.11 standard (access to latest version is paying, but older versions are free)
- the Linux wireless mailing list
- the kernel documentation for cfg80211 and mac80211
- taking a look at the existing drivers

Thank you!

Questions?

Alexis Lothoré

alexis.lothore@bootlin.com

Slides under CC-BY-SA 3.0

https://bootlin.com/pub/conferences/