tags: `Linux`

Linux Wireless Stack

Before going deep into the topic, the first thing we need to know is that there are two types of WIFI devices when it comes to the network driver. This depends on where **IEEE802.11 MLME** is implemented. MLME stands for **MAC sublayer Management Entity** where the physical layer MAC state machines reside. MLME mainly deals with:

- Beacom
- Probe
- Authenticate
- Deauthenticate
- Associate
- Reassociate
- Disassociate
- Timing Synchronization Function (TSF)

If the MLME is implmeneted in a device hardware (or firmware), we call this device as **Full-MAC device**, othwerwise **Soft-MAC device**.

For **Soft-MAC device**, there's mac80211 which is a kernel module implmenting the 802.11 MAC layer. In this case cfg80211 will talk to mac80211 that will in turn use the hardware specific lower level driver. mac80211 registers itself with cfg80211 by using `cfg80211_ops`. The specific HW driver registers itself with mac80211 by using `ieee80211_ops` structure.

The MAC layer can be further divided into **upper MAC (UMAC)** and **lower MAC (LMAC)**, where the former refers to the management aspect of the MAC (e.g. probing, authentication and association), and the latter refers to the time critical operations of MAC such as ACK.

Most of time, the hardware, or WIFI adapters, deals with majority of PHY and **LMAC** operations, and the Linux wireless subsystem laryer largely handles the **UMAC**.

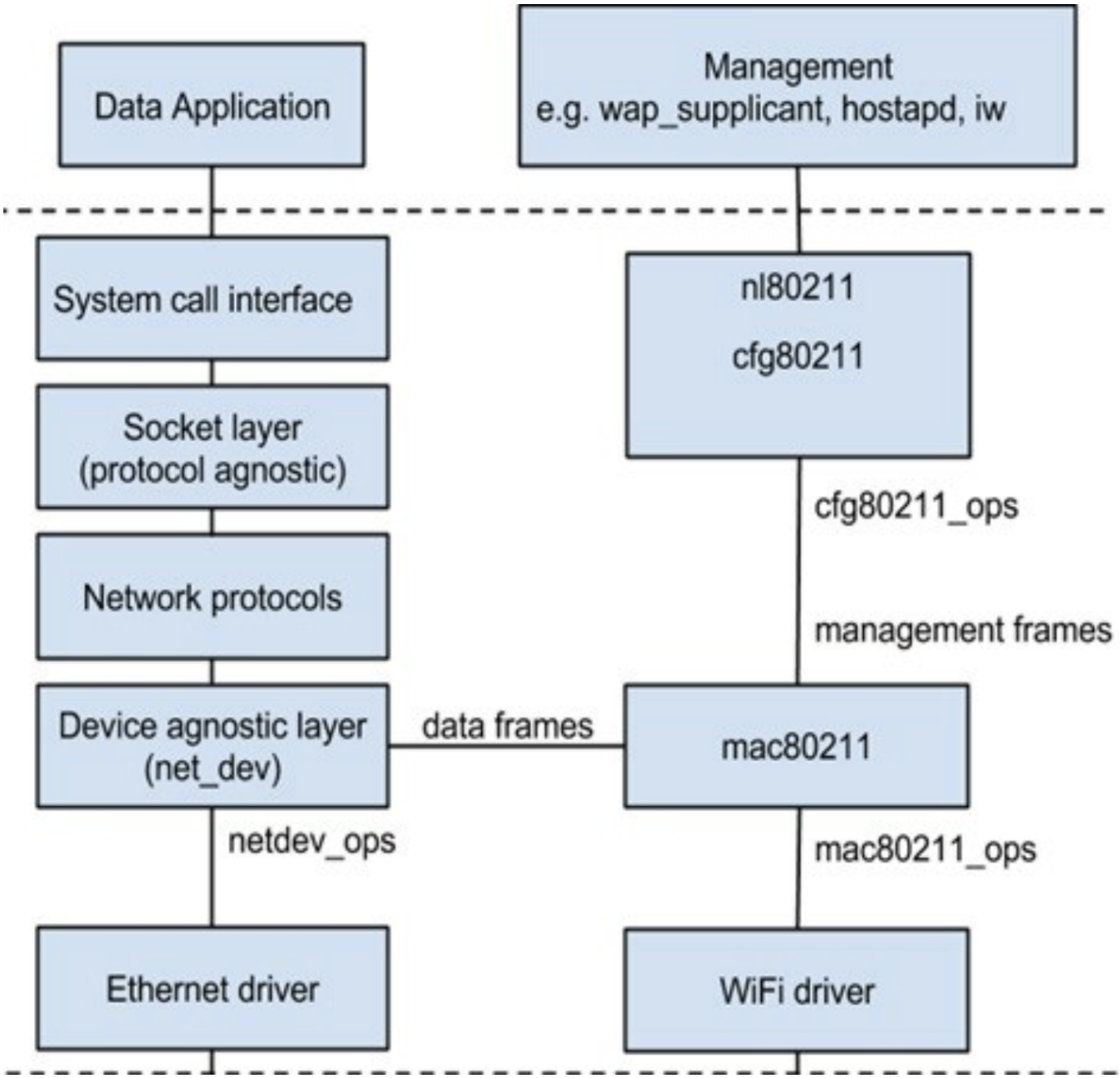
That's, **LMAC** drivers act as a bridge between the **UMAC** and the chipset (Firmware and HW). They do all device initialization, registration with OS, bus registration, Interrupts registration etc., through the services provided by the Linux kernel.

A well written wlan driver follows these conventions:

- Maintains a OS Independent Layer: Easy portability to different OSes.
- Maintains a **UMA** Independent Layer: Easy portability to different UMAC's: Proprietary, opensource, 3rd party etc.

- Bus Abstraction Layer: Maintains compatibility across different Physical Buses like PCI, PCIe, AHB, SDIO etc.

Linux Network Stack



Two major blocks of Linux wireless subsystem:

- MAC80211
- CFG80211

They help wireless driver to interface with rest of kernel and user space. It also provides management interface between kernel and user space using **nl80211**. Note that mac80211 driver API only supports **Soft-MAC Device**, while cfg80211 must be implmeneted by both **Soft-MAC Device** and **Full-MAC Device**.

Initialization

When the iwlmvm (Intel) module is loaded, it sets a series of callback functions to mac80211 `struct ieee80211_ops` by calling `iwl_op_mode_mvm_start()` to complete the initialization. `struct ieee80211_ops` maps driver implementation of the handlers to the common mac80211 API.

struct ieee80211_ops (https://elixir.bootlin.com/linux/latest/source/include/net/mac80211.h#L3241)

```
const struct ieee80211_ops iwl_mvm_hw_ops = {
        .tx = iwl_mvm_mac_tx,
        .wake_tx_queue = iwl_mvm_mac_wake_tx_queue,
        .ampdu_action = iwl_mvm_mac_ampdu_action,
        .get_antenna = iwl_mvm_op_get_antenna,
        .start = iwl_mvm_mac_start,
        .reconfig_complete = iwl_mvm_mac_reconfig_complete,
        .stop = iwl_mvm_mac_stop,
        .add_interface = iwl_mvm_mac_add_interface,
        .remove_interface = iwl_mvm_mac_remove_interface,
        .config = iwl_mvm_mac_config,

         .....

        .start_ap = iwl_mvm_start_ap_ibss,
        .stop_ap = iwl_mvm_stop_ap_ibss,
        .join_ibss = iwl_mvm_start_ap_ibss,
        .leave_ibss = iwl_mvm_stop_ap_ibss,
        .tx_last_beacon = iwl_mvm_tx_last_beacon,
        .set_tim = iwl_mvm_set_tim,
};
```

This structure will be passed into mac80211 subsystem via `ieee80211_alloc_hw()`, which allocates a new hardware device and a prvidate data area for iwlwifi.

Addtionally, in the `ieee80211_alloc_hw()` mac80211 will also register callback functions into `struct cfg80211_ops` and pass the structure by calling `wiphy_new_nm()`.

struct cfg80211_ops (https://elixir.bootlin.com/linux/latest/source/include/net/cfg80211.h#L3388)

```
const struct cfg80211_ops mac80211_config_ops = {
        .add_virtual_intf = ieee80211_add_iface,
        .del_virtual_intf = ieee80211_del_iface,
        .change_virtual_intf = ieee80211_change_iface,
        .start_p2p_device = ieee80211_start_p2p_device,
        .stop_p2p_device = ieee80211_stop_p2p_device,

        .....

#ifdef CONFIG_MAC80211_MESH
        .add_mpath = ieee80211_add_mpath,
        .del_mpath = ieee80211_del_mpath,
        .change_mpath = ieee80211_change_mpath,
        .get_mpath = ieee80211_get_mpath,
        .dump_mpath = ieee80211_dump_mpath,
        .get_mpp = ieee80211_get_mpp,
        .dump_mpp = ieee80211_dump_mpp,
        .update_mesh_config = ieee80211_update_mesh_config,
        .get_mesh_config = ieee80211_get_mesh_config,
        .join_mesh = ieee80211_join_mesh,
        .leave_mesh = ieee80211_leave_mesh,
#endif

        .....
};
```

Another important function needed to be called at inital stage is `ieee80211_register_hw()`. This function takes the `struct ieee80211_hw` that contains the configuration and hardware information for an 802.11 PHY to test if the information that developers fill to matches the hardware support. The Intel driver implements them in `iwl_mvm_mac_setup_register()`.

struct ieee80211_hw (https://elixir.bootlin.com/linux/latest/source/include/net/mac80211.h#L2386)

Tx & Rx

The data and management paths are split in mac80211.

- Data path
  - Corresponds to the IEEE 802.11 data frames
- Management Path
  - Corresponds to the IEEE 802.11 management frames.

Another frame is called **IEEE80211 control frames**, which is handled by hardware as most of them are time critical e.g. ACK.

Transimitting Data Packet

Brief flow:

User space Application -> Create Socket -> Bind it to interface (e.g.: Ethernet or Wifi) -> Put content into sk_buff & send it

1. Create Socket & system call

- Specify it's protocol family which will be used by kernel.
  - The `AF_` prefix stands for address family (IPv4 or IPv6) and the `PF_` prefix stands for protocol family (TCP,UDP...).
- This happens in data application block, and invokes a system call.

2. Socket layer (protocol agnostic)

The transmission first passes the socket layer, and an important structure here is **struct sk_buff (or skb)**.

- An important structure in transmission/reception process.
- Holds pointer for data buffer.
- Tracks data length.
- Provides good API support to transfer data among different layers of kernel such as header insertion and removal.

3. Network protocol layer

We then pass the **network protocol block**. The transmission is mapped to a networking protocol according to the protocol specified during socket creation, and the corresponding protocol will continue to handle the transmission of the packet.

4. Device agnostic layer

When packets arrives in **device agnostic layer**, which links various hardware devices like Ethernet and WiFi to different network protocols, the data inside **skb** is passed to a function called dev_queue_xmit (https://elixir.bootlin.com/linux/v5.8/source/net/core/dev.c#L4031). After tracing through the call from this function...

dev_queue_xmit -> dev_hard_start_xmit -> xmit_one->netdev_start_xmit -> __netdev_start_xmit

it eventually invokes **ops->ndo_start_xmit(skb, dev)**. This is exactly the API handler that Ethernet device drivers need to register. But in case of WIFI it's mac80211 that registers itself with netdev_ops `(net/mac80211/iface.c)`

```
static const struct net_device_ops ieee80211_dataif_ops = {
#if LINUX_VERSION_IS_LESS(4,10,0) && RHEL_RELEASE_CODE < RHEL_RELEASE_VERSION(7,6)
        .ndo_change_mtu = __change_mtu,
#endif

        .ndo_open               = ieee80211_open,
        .ndo_stop               = ieee80211_stop,
        .ndo_uninit             = ieee80211_uninit,
        .ndo_start_xmit                 = ieee80211_subif_start_xmit,
        .ndo_set_rx_mode        = ieee80211_set_multicast_list,
        .ndo_set_mac_address    = ieee80211_change_mac,
        .ndo_select_queue       = ieee80211_netdev_select_queue,
#if LINUX_VERSION_IS_GEQ(4,11,0) || RHEL_RELEASE_CODE >= RHEL_RELEASE_VERSION(7,6)
        .ndo_get_stats64        = ieee80211_get_stats64,
#else
        .ndo_get_stats64 = bp_ieee80211_get_stats64,
#endif
};
```

So mac80211 appears as net_device invokes transmit handler `ieee80211_subif_start_xmit` to transmit a packet. The result in below is the trace from that call:

ieee80211_subif_start_xmit -> ieee80211_xmit -> ieee80211_tx -> ieee80211_tx_frags -> drv_tx

We are now at the boundary between mac80211 and WiFi driver. The `drv_tx` is simply a wrapper that maps the transmission to the tx handler that WiFi device driver has registered:

```
static inline void drv_tx(struct ieee80211_local *local,
                          struct ieee80211_tx_control *control,
                          struct sk_buff *skb)
{
        local->ops->tx(&local->hw, control, skb);
}
```

The `local->ops->tx` was previously registered, and at this moment the wifi device driver will take over after this function invoked.

The driver may also need to feedback the transmit status (also via `struct ieee80211_tx_info`) to mac80211 by invoking `ieee80211_tx_status` , or one of its variants. This also marks the end of packet transmission.

Rx path

Usually when a device driver wants to receive data from hardware, it must register its interrupt handler to kernel to know when incoming packets arrvie. In ath10k, we can see the `ath10k_pci_interrupt_handler` will handle such as things.

```
static irqreturn_t ath10k_pci_interrupt_handler(int irq, void *arg)
{
    ....

        napi_schedule(&ar->napi);

        return IRQ_HANDLED;
}
```

There is nothing but calling a schedule to execute its handler later. NAPI stands for New API, which is used to invoke a function polling the data from queues based on the budget and quota. It is an importnat mechanism in Linux network for solving the resource starvation, however, it is out of this topic and will see it from another article. When NAPI starts to schedule the function that was registered previously, the driver starts dequeuing and delivering received data to `mac80211` after processing data from hardware.

```
static int ath10k_pci_napi_poll(struct napi_struct *ctx, int budget)
{
    .....

        ath10k_ce_per_engine_service_any(ar);

        done = ath10k_htt_txrx_compl_task(ar, budget);

    .....
}
```

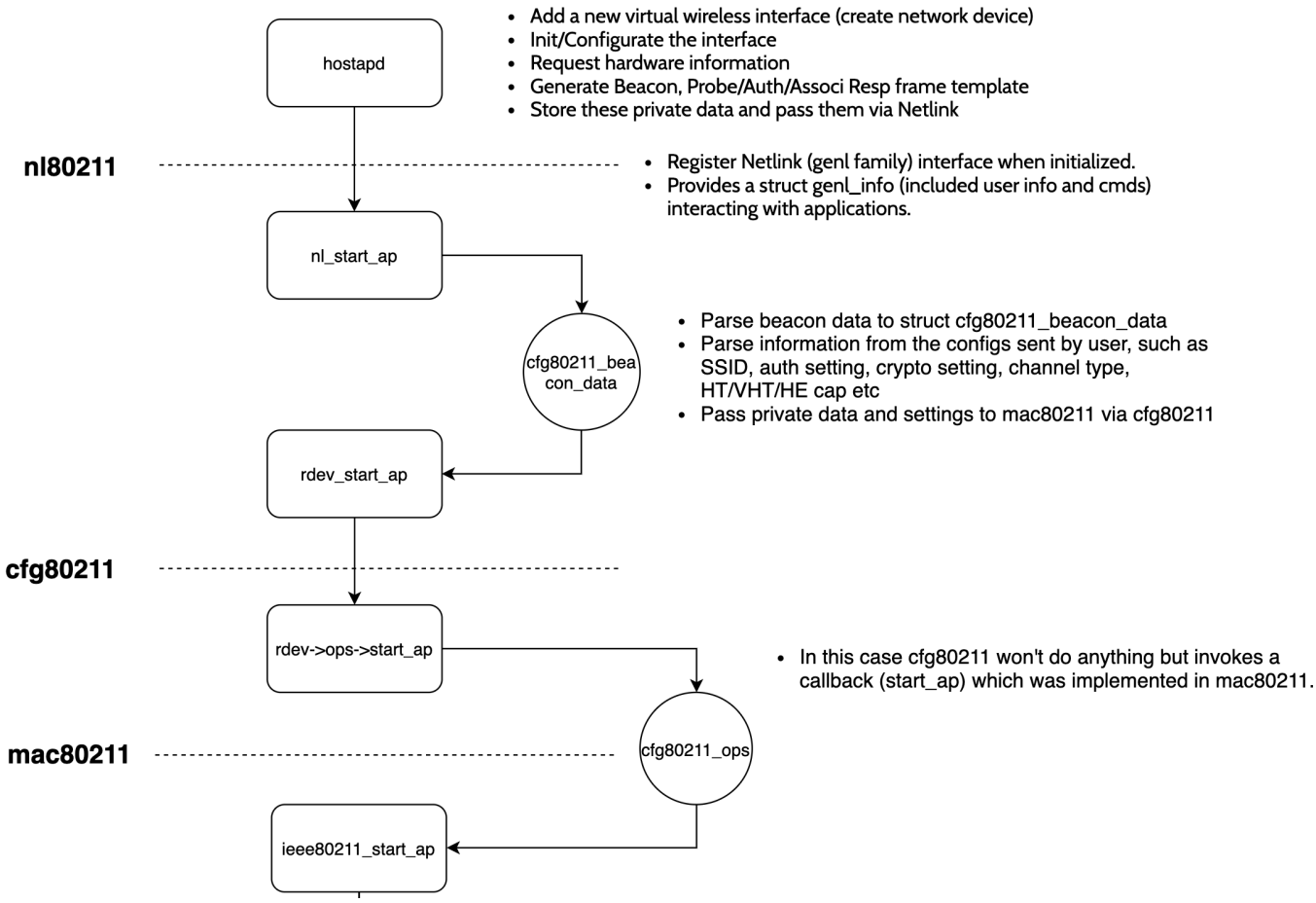
Before feeding data to be processed, it first get data from each `ce` engine where puts data from DMA to it private pipes. Eventually, the data inserted to `struct sk_buff` will be sent out to `mac80211` when `ath10k_pci_napi_poll` has been scheduled.

```
static void ath10k_process_rx(struct ath10k *ar, struct sk_buff *skb)
{
        struct ieee80211_rx_status *status;
        struct ieee80211_hdr *hdr = (struct ieee80211_hdr *)skb->data;
        char tid[32];

        status = IEEE80211_SKB_RXCB(skb);

    ....

        ieee80211_rx_napi(ar->hw, NULL, skb, &ar->napi);
}
```

Actually at this moment we know what MCS is used in this tranmission and what kinds of 802.11 protocol is involved and so on. All of the information will be pointed to `struct ieee80211_rx_status` and pass them by calling `ieee80211_rx_napi` to parse further mac frames later in `mac80211` .

Tx path

Beacon Tx path Part I

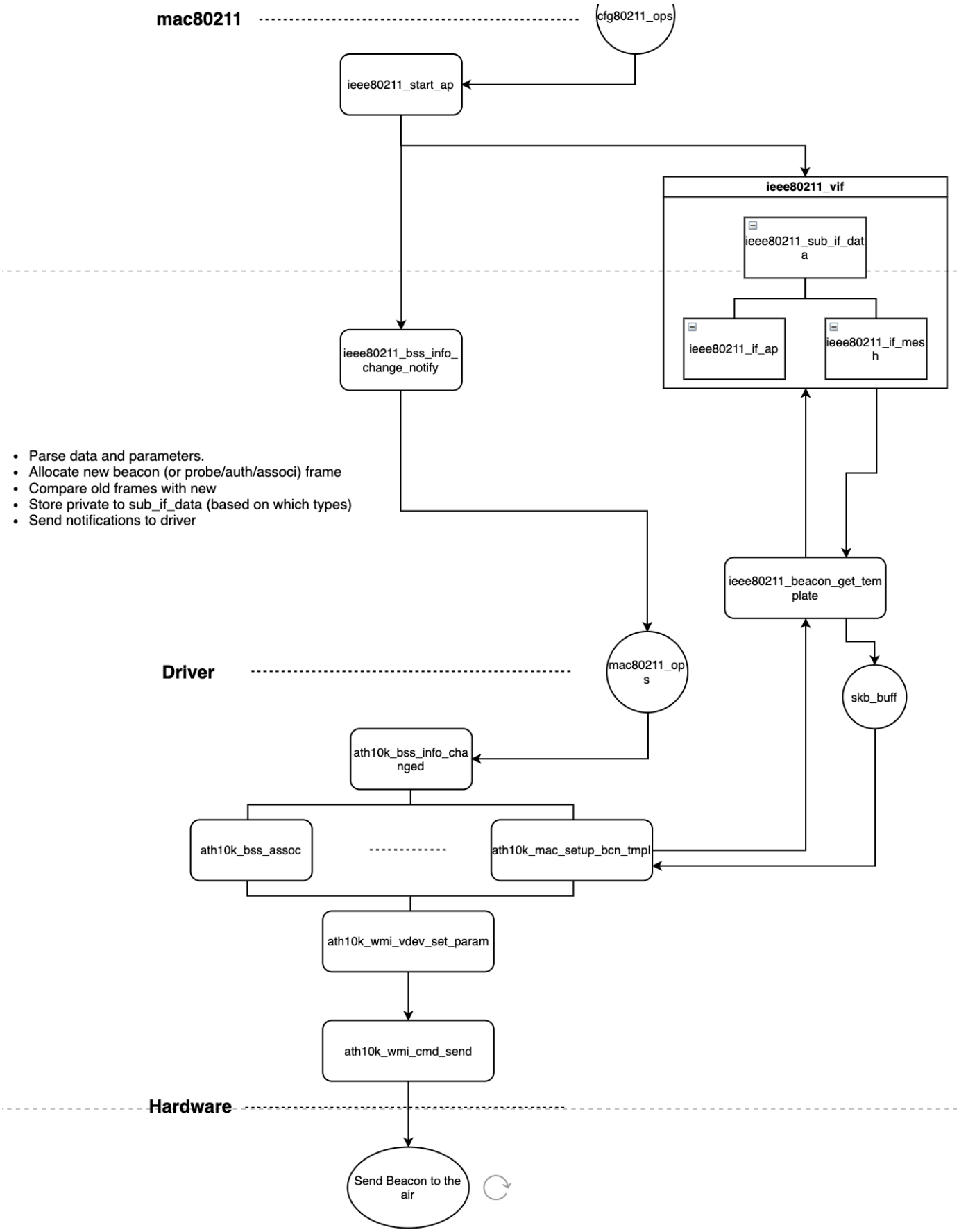


- **hostapd -> nl80211**
  - When the driver is initialized, a virtual interface will be created and a network device node is also registered with kernel by mac80211. Likewise, a area of private data for this interface is created as well.
  - When hostapd service starts, it sends messages to hardware through the interface for its capabilities.
  - hostapd then constructs a beacon frame template (probe/auth/associ resp built up as well) based on the capabilities and writes configurations into the structure by reading `hostapd.conf`
  - All parameters and templates are stored into `struct genl_info` , which will eventually be passed to **nl80221 layer** via Netlink mechnism.
- **nl80211 -> cfg80211**

- When the messages arrives at nl80211, it invokes a callback function according to the command which referecens to `strcut genl_ops` .
- The function `nl_start_ap` parses these messages into `struct cfg80211_ap_settings` , including beacon frame (associated with `struct cfg880211_beacon_data`) and other parameters on given `attributes` after validated with hardware.
- Finally, nl80211 hands over cfg80211 by invoking `rdev_start_ap`.

- **cfg80211 -> mac80211**
  - In this case cfg80211 won't do anything but invokes a function implemented in mac80211. (Actually `rdev_start_ap` calls `ops->start_ap` directly)
  - `.start_ap` is a part of function pointer in `struct cfg80211_ops` , which is implemented by `mac80211` explained earlier.

Beacon Tx path Part II



- **mac80211 <—> driver**
  - The function `ieee80211_start_ap` points out the area of private data by calling `IEEE80211_DEV_TO_SUB_IF` . One part of private areas stores actual data such as beacon frames and payloads, all of these are saved in `struct`

`ieee80211_sub_if_data` . Another private area stores all parameters in `struct ieee80211_bss_conf` .

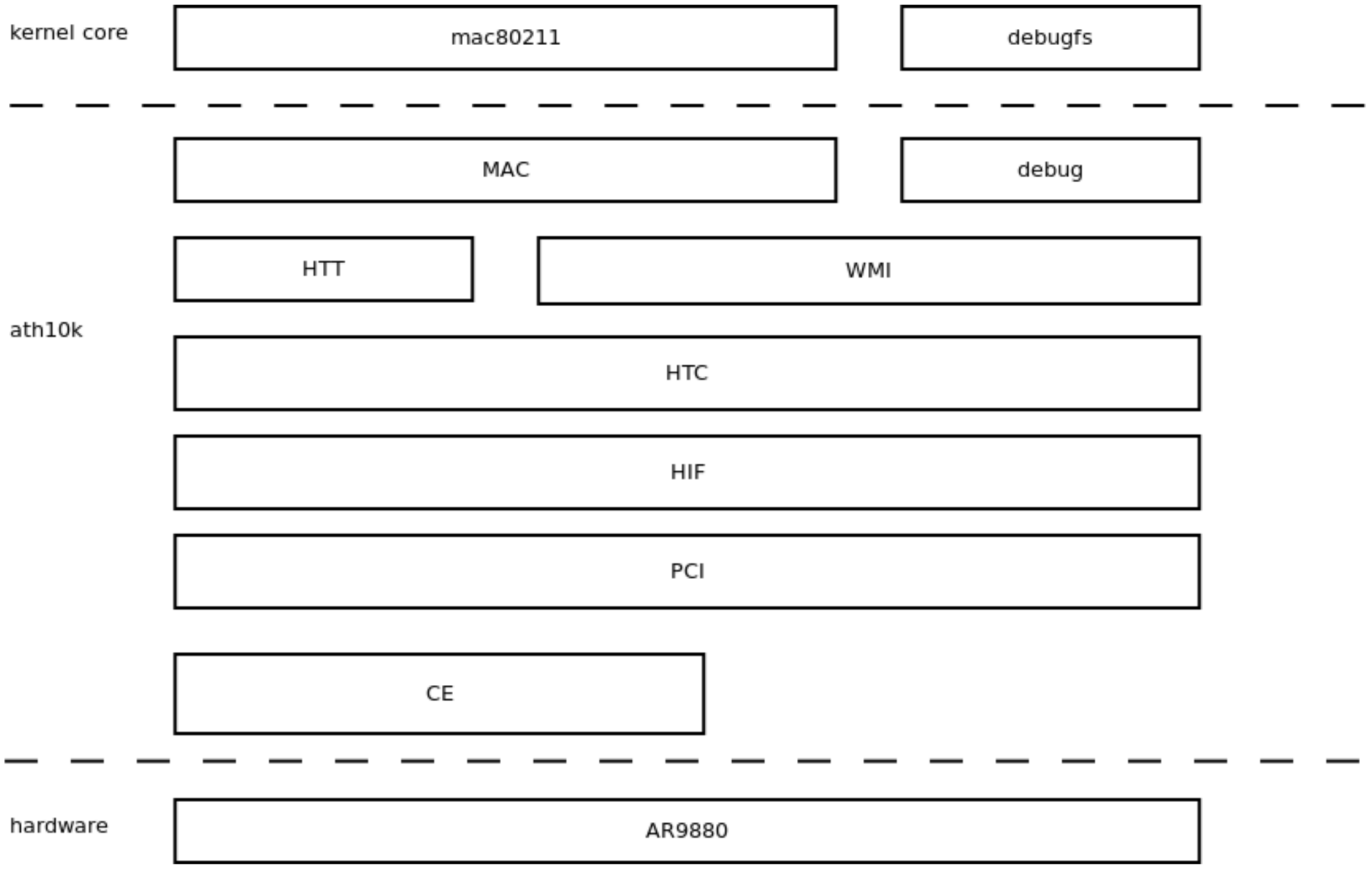
- `struct ieee80211_sub_if_data` is only visible for mac80211 and contains information about each virtual interface and sub-structures, such as `ieee80211_if_ap` , `ieee80211_if_mesh` and so on, while `struct ieee80211_vif` is passed to driver for those virtual interface that the driver knows about. They can be access by API like `vif_to_sdata` between two layers.
- The actual beacon frame will be constrcuted in the function `ieee80211_assign_beacon` . Once all data and parameters are prepared, the function that has been registered in `mac80211_ops` would be invoked to notify the driver that there is a event happening, in this case `.bss_info_changed` is called.

- **driver <–> hardware**
  - `atk10k_bss_info_changed` follows the flag passed by upper layer to set up its hardware status.
  - It fetches the beacon template by invoking `ieee80211_beacon_get_template` from `ath10k_mac_setup_bcn_tmpl` , where the frame with the TIM will be put into `skb_buff` and passed down to the driver.
  - Once all data and parameter was transmitted to the firmware successfully, the firmware will periodically send the beacn frame to the air until the hostapd service is terminated.

Note that the beacon frame template generated by hostapd is only for the static parts, while the dynamic parts like TimeStamp would be inserted by `mac80211` .

QCA6174 driver (ath10k)

ath10k driver is a wireless standard driver based on mac80221 arch in Linux kernel, which you can get the source code from mainstream or backport version. The structure of its driver is shown on below:



The following lists the each of component definitions.

- **MAC**
  - File: mac.c mac.h
  - This is the glue layer between mac80211 and ath10k lower levels. The interface to mac80211 is implemented through `ath10k_ops` . Data and management frames are sent to HTT, configuration commands to WMI.
- **Host-Target Transport (HTT)**
  - File: htt.c htt.h htt_rx.c htt_tx.c
  - The data path for ath10k. Sends frame descriptors to the firmware using HTC.
- **Wireless Module Interface (WMI)**
  - File: wmi.h wmi.c
  - Sends all sorts configuration commands to the firmware and receives configuration events from the firmware.
- **Host interconnect Framework (HIF)**
  - File: hif.h
  - Abstracts the access to different bus types. Currently only supports PCI, but it’s easy to add different bus types.

- **Debug**
  - File: debug.h debug.c
  - Component for various debug related to code. Currently only log messages and debugfs.
- **Tracing**
  - Files: trace.h trace.c
  - Provides tracing data (HTT/WMI packets) etc to userspace using Linux tracepoints. trace-cmd is the recommend tool to access the tracepoints.
- **PCI**
  - Files: pci.h pci.c ce.h ce.c
  - All PCI related code. Interface to HIF happens through `ath10k_pci_hif_ops` .
- **Copy Engine (CE)**
  - The firmware/ hardware has 8 rings for communication with host which is defined in `host_ce_config_wlan` .
  - Copy Engine provides abstraction for these ring buffers and calls each ring a pipe.
- **Bootloader Messaging Interface (BMI)**
  - Files: bmi.h bmi.c
  - Firmware upload and everything else which happens before firmware is booted.
- **Core**
  - Files: core.h core.c
  - Driver initialisation and firmware booting. Manages all ath10k components.

Intel AX200 driver

- **Transport Layer**
  - File: iwl_trans.h
  - This is a layer that deals with the HW directly, provides an abstraction of the underlying HW to the upper layer, and creates mechanismes to make the HW do something as well.
  - **Life cycle**

        - Transport helper function is called during the module initialization and registers the bus driver's ops with the transport's alloc function.
        - Refer to iwl_trans_alloc and iwl_pci_register_driver
        - Bus's probe calls to the transport layer's allocation that is bus specific.
        - This allocation functions will spawn the upper layer which will register mac80211.
        - `stop_device` will be called when finished or resetted.

- **Operational Mode (op_mode)**
    - File: iwl-op-mode.h
    - This is a layer that implements **mac80211's handlers**, which diff with fw APIs that will require different op modes.
    - It uses the **transport layer** to access the HW.
    - Life cycle
        - The driver layer (iwl-drv.c) chooses the op mode based on the capabilities advertised by the fw file.
        - The driver layer start the op mode through `ops->start` .
        - The op mode registers mac80211.
        - The op mode is governed by mac80211.
        - The driver layer stop the op mode.

References

- https://www.linkedin.com/pulse/wlan-drivers-code-walk-linux-bhanu-prakash-singh/ (https://www.linkedin.com/pulse/wlan-drivers-code-walk-linux-bhanu-prakash-singh/)
- https://www.linux.com/training-tutorials/linux-wireless-networking-short-walk/ (https://www.linux.com/training-tutorials/linux-wireless-networking-short-walk/)
- https://wireless.wiki.kernel.org/en/users/drivers/ath10k/architecture (https://wireless.wiki.kernel.org/en/users/drivers/ath10k/architecture)