Linux Wireless Networking: a short walk

By **Fred Chou** - March 15, 2015

How does Linux wireless interface with the kernel? How are packets transmitted / received?
When I started working on Linux wireless, I was lost in the immense code base, and I was looking for an introductory material that can answer high-level questions like above. After tracing through the source codes, I put down this summary in the hope that it can be helpful as an overview of how things work in Linux wireless networking.

Overview

To begin the walk, letâ€™s first have an overview of the architecture in Fig. 1, which should show the high-level blocks in Linux kernel networking, and how Linux wireless fits into the kernel.

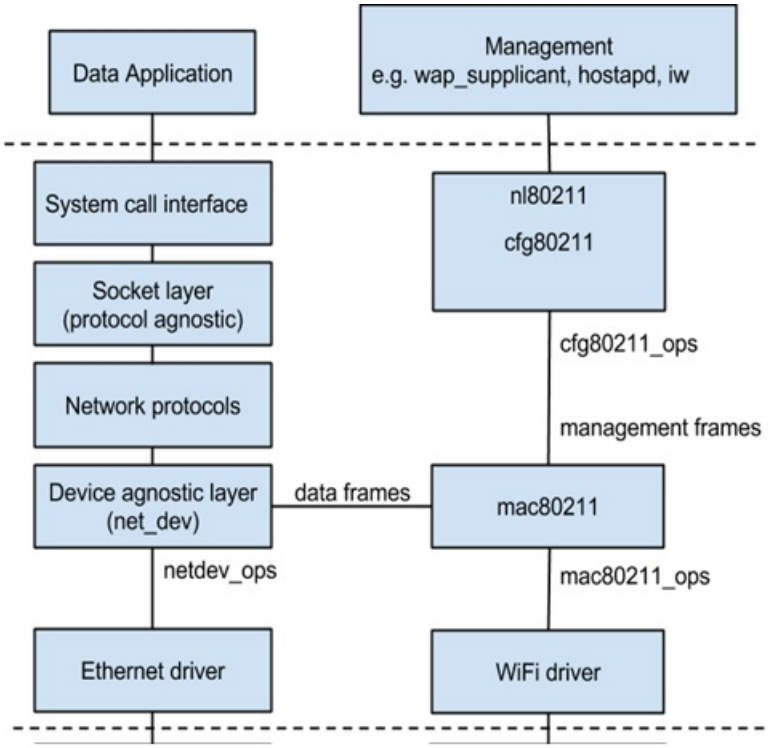

Fig. 1: Overview of Linux wireless networking architecture

The bulk of Fig. 1 shows the kernel space. The user space applications are running on top of it, and the hardware devices are at the bottom. On the left are Ethernet devices, and on the right are WiFi devices.

There are two types of WiFi devices, depending on where IEEE802.11 MLME is implemented. If it is implemented in device hardware, the device is a full MAC device. If it is implemented in software, the device is a soft MAC device. Most devices today are soft MAC devices.

Often, we can think of the Linux wireless subsystem to contain two major blocks: cfg80211 and mac80211, and they help the WiFi driver to interface with rest of the kernel and user space. In particular, cfg80211 provides configuration management services in the kernel. It also provides management interface between kernel and user space via nl80211. Both soft MAC and full MAC

devices need to work with cfg80211. Mac80211 is a driver API that supports only software MAC devices. Our focus here will be on soft MAC devices, as shown in Fig. 1.

The Linux wireless subsystem, together with WiFi devices, deals with the bottom two layers (MAC and PHY) of the OSI model. For a more refined breakdown, the MAC layer can be further divided into upper MAC and lower MAC, 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 the time, the hardware, such as WiFi adapters, deals with majority of PHY and lower MAC operations, and the Linux wireless subsystem largely handles the upper MAC.

Interface among Blocks

The blocks in Fig. 1 have clearly defined boundaries, and support transparency, which means what happens in one block should not affect others. For example, we may make some changes in the WiFi driver block (e.g. applying a patch, or even adding a new driver for new device), but such changes does not affect the mac80211 block, and we do not need to change the codes in mac80211. As another example, adding a new network protocol should ideally not change the codes in socket layer and device agnostic layer. Such transparency is often achieved through function pointers. An example of how a WiFi driver (rt73usb) interfaces with mac80211 is:

```
 static const struct ieee80211_opsrt73usb_mac80211_ops = {
.tx                    = rt2x00mac_tx,         .start                  =
rt2x00mac_start,         .stop                   = rt2x00mac_stop,
.add_interface         = rt2x00mac_add_interface,
.remove_interface      = rt2x00mac_remove_interface,
.config                = rt2x00mac_config,
.configure_filter      = rt2x00mac_configure_filter,
.set_tim               = rt2x00mac_set_tim,
.set_key               = rt2x00mac_set_key,
.sw_scan_start         = rt2x00mac_sw_scan_start,
.sw_scan_complete      = rt2x00mac_sw_scan_complete,
.get_stats             = rt2x00mac_get_stats,
.bss_info_changed      = rt2x00mac_bss_info_changed,
    .conf_tx               = rt73usb_conf_tx,
.get_tsf               = rt73usb_get_tsf,         .rfkill_poll
= rt2x00mac_rfkill_poll,         .flush                  =
rt2x00mac_flush,         .set_antenna            =
rt2x00mac_set_antenna,         .get_antenna            =
rt2x00mac_get_antenna,         .get_ringparam          =
rt2x00mac_get_ringparam,         .tx_frames_pending      =
rt2x00mac_tx_frames_pending,};
```

On the left are the API in the form of struct ieee80211_ops that mac80211 provides to the WiFi device drivers, and it is up to the drivers to implement these handlers. Of course, different devices will have different driver implementations. The struct ieee80211_ops keeps the mapping between driver implementation of the handlers to the common mac80211 API. During driver registration, these handlers are registered to the mac80211 (through ieee80211_alloc_hw), and mac80211 can then blindly invoke a handler without knowing its name and detailed implementations. The original struct

ieee80211_ops contains a long list of APIs, but not all of them are mandatory. For successful compilation, implementing the first seven APIs is sufficient, but for proper functioning, some more APIs may need to be implemented, as in the example above.

Data Path and Management Path

From Fig. 1, there are two major paths in the architecture: a data path, and a management path. The data path corresponds to the IEEE 802.11 data frames, and the management path corresponds to the IEEE 802.11 management frames. For IEEE802.11 control frames, as most of them are used for time-critical operations such as ACK, they are often handled by the hardware. An exception may be the PS-Poll frame, which can also be taken care of by mac80211. The data and management paths are split in mac80211.

How is data packet transmitted?

Here we focus on the data path for transmission.

Starting from user space application, often we create a socket, bind it to an interface (e.g. Ethernet or WiFi), put the content into socket buffer, and send it. In socket creation, we also specify its protocol family, which will be used by the kernel. This happens in the data application block in Fig. 1. Eventually this invokes a system call, and subsequent work happens in the kernel space.

The transmission first passes the socket layer, and an important structure here is struct sk_buff, or more commonly known as skb. A skb holds pointer to the data buffer and tracks the data length. It provides very good support and APIs to transfer our data among different layers in the kernel, such as header insertion/removal, and is used thoughout the packet transmission / reception process.

We then pass the network protocol block. There is not much to say about the networking protocols because there is too much to say about the networking protocols. The protocols are really not the focus here, and it suffices to know that 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.

Now the transmission lands itself at the device agnostic layer, which links various hardware devices like Ethernet and WiFi to different network protocols, transparently. The device agnostic layer is characterized by an important structure: struct net_device. This is how Ethernet device drivers interface with the kernel, as shown by the Ethernet driver block in Fig. 1. The interfacing is via struct net_device_ops, which has a long list of net_device operations. Particularly for transmit:

```
struct net_device_ops {

â€¦

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

```
â€¦

};
```

To send the packet, the skb is passed to a function called dev_queue_xmit. After tracing through the call, it eventually invokes ops->ndo_start_xmit(skb, dev). This is exactly the API handler that Ethernet device drivers need to register.

For WiFi devices, however, it is usually mac80211 (instead of the device drivers) that registers with netdev_ops. See net/mac80211/iface.c:

```
static const struct net_device_opsieee80211_dataif_ops =
{        .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_change_mtu         =
ieee80211_change_mtu,       .ndo_set_mac_address    =
ieee80211_change_mac,       .ndo_select_queue       =
ieee80211_netdev_select_queue,};
```

So mac80211 also appears as a net_device, and when a packet needs to be transmitted via WiFi, the corresponding transmit handler, ieee80211_subif_start_xmit, is invoked, and we are entering the mac80211 block. Below is the call trace for ieee80211_subif_start_xmits. Subsequently it invokes:

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

```
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);}
```

This is the end of mac80211, and device driver will take over.

As mentioned previously, through the mac80211 local->ops->tx, the registered handler in the device driver is invoked. Each driver has it unique implementation of the handler. Following the previous example in Interface among Blocks, the tx handler is rt2x00mac_tx, which first prepares the transmit descriptor normally including information such as frame length, ACK policy, RTS/CTS, retry limit, more fragments, and MCS, etc. Some information is passed down from mac80211 (e.g. in struct ieeee80211_tx_info, which in this case tells the device driver what to do), and the driver has to convert the information into a form that its underlying hardware can understand.Once the device specific transmit descriptor is done, the driver may condition the frame (e.g. adjust byte alignment), put the frame on a queue, and eventually send the frame (and its transmit descriptor) to hardware. As our

example is a USB WiFi adapter based on rt73usb, the frame is sent via the USB interface to the hardware, which further inserts the PHY header and other information and transmits the packet to the air. 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.

What about management path?

Theoretically we can transmit management frames the same way as in data path by constructing management frames in user space and send them via socket, but there are well-developed user space tools, notably wpa_supplicant and host_apd, that can do the work. Wpa_supplicant controls the wireless connection for client STAs such as scan, authentication, and association, whereas host_apd functions as AP. These user space tools uses netlink socket to communication with the kernel, and the corresponding handler in the kernel is nl80211 in cfg80211. These user space tools will invoke the send handlers in netlink library to send command (e.g. NL80211_CMD_TRIGGER_SCAN) to the kernel. In kernel space, the command is received by nl80211, which has the mapping between command (.cmd) and action (.doit) in static struct genl_ops nl80211_ops:

```
static const struct genl_opsnl80211_ops[] = {

â€¦

{

.cmd = NL80211_CMD_TRIGGER_SCAN,

.doit = nl80211_trigger_scan,

.policy = nl80211_policy,

.flags = GENL_ADMIN_PERM,

.internal_flags = NL80211_FLAG_NEED_WDEV_UP | NL80211_FLAG_NEED_RTNL,

},

â€¦

};
```

For the example of triggering scan, the scan request is passed from cfg80211 to mac80211 via the scan handler that mac80211 has registered with cfg80211 in struct cfg80211_ops:

```
const struct cfg80211_ops mac80211_config_ops = {
```

â€¦

.scan = ieee80211_scan,

â€¦

};

In mac80211, ieee80211_scan will take over the scanning process:

=>ieee80211_scan_state_send_probe

=>ieee80211_send_probe_req

=>ieee80211_tx_skb_tid_band
=>ieee80211_xmit
=>ieee80211_tx
=>ieee80211_tx_frags
=>drv_tx

How is received packet handled?

We are now traveling in the reverse direction for packet reception. For the moment we do not differentiate between data and management path.

When a packet is captured by the WiFi hardware over the air, the hardware may generate an interrupt to the kernel (e.g. as in most PCI interfaces), or the packet may be polled (e.g. for the case of USB interface). In the former case, the interrupt will lead to a receive interrupt handler, and for the latter, a receive callback handler is invoked.

It turns out that in these handlers the device driver does not do much with the received packet except for some sanity check, filling up the receive descriptor for mac80211, and then passing the packet to mac80211 for further processing (either directly or more commonly putting the packet on a receive queue).

Entry to mac80211 is via ieee80211_rx or one of its variants, which invokes various receive handlers in mac80211 (see in ieee80211_rx_handlers for the code). This is also where the data path and management path are divided.

If the received frame is of type data, it is translated into 802.3 frame (by __ieee80211_data_to8023) and is delivered to the networking stack via netif_receive_skb. From then on, the network protocol block will parse and decode the protocol header.

If the received frame is of type management, it is processed in ieee80211_sta_rx_queued_mgmt. Some management frames end in mac80211, and some are further passed up to cfg80211 and sent to user space management tools. For example authentication frames are further processed by cfg80211_rx_mlme_mgmt and sent to user space via nl80211_send_rx_auth, and association response frames are processed by cfg80211_rx_assoc_resp and sent to user space via nl80211_send_rx_assoc.

Concluding the Walk

The typical flow pattern of a WiFi driver contains three tasks: configuration, transmit path handling, and receive path handling. Take again the USB WiFi adapter for example, when the device is detected, the probe function is invoked. This could be where the configuration like registration of ieee80211_ops takes place:

First, ieee80211_alloc_hw allocates a struct ieee80211_hw, which represents the WiFi device. In particular, the following data structures are allocated:

- – struct wiphy: mainly used to describe WiFi hardware parameters like MAC address, interface modes and combinations, supported bands, and other hardware capabilities.
- – struct ieee80211_local: this is the driver visible part and is largely used by mac80211. The mapping of ieee80211_ops is linked to the struct ieee80211_local (const struct ieee80211_ops *ops of struct ieee80211_local). It can be accessed from ieee80211_hw by using container_of, or the API hw_to_local.
- – Private struct for device driver (void *priv in struct ieee80211_hw).

```
Registration of the hardware is completed through ieee80211_register_hw, after which other mac802
```

Hopefully, with this overview, tracing through the source codes would be easier.

References:

Linux wireless subsystem: https://wireless.wiki.kernel.org/en/developers/documentation

Fred Chou