FreeBSD WiFi Development

Part 2: Working on a Driver

BY TOM JONES

This is the second article in a series on hacking on WiFi in FreeBSD. In the first article, we introduced some terminology around WiFi/80211 networks, gave a crash course in a typical network architecture, and presented examples of using `ifconfig` and some wireless adapters to create station, host ap, and monitor mode WLAN interfaces. We also introduced the two distinct kernel layers that implement the WiFi subsystem — drivers and net80211.

Drivers Things like iwx, rtwn, and ath, which speak to wireless adapters over a physical bus such as USB or PCIe, usually via a firmware interface.

net80211 The abstract state machines required to join networks, send packets, and perform other complex operations are common across many drivers.

To enable flexibility in what hardware is required to provide, the net80211 layer can implement most parts of the IEEE 802.11 state machine itself. This architecture allows us to have a standard interface to integrate with the rest of the networking stack, abstracting away the specifics of what a card can support. It also enables purely software-based WLAN adapters, which can be very useful for creating test environments.

Network Adaptors can provide various levels of support from a FullMAC interface, where almost all of the processing is handled on the card directly to devices where all most all support is handled by the net80211 stack and the card just manages the radios. In a FullMAC card, the firmware exposes a configuration interface for the OS driver to use. All packet transmission, reception, and management operations, such as moving channel or scanning, are handled by the card's firmware. The bwfm Broadcom driver in OpenBSD and NetBSD is an example of a FullMAC driver.

Network Adaptors can provide various levels of support from a FullMAC interface.

Other cards need the net80211 stack to provide a variety of services to support the driver's operations. Some devices, like iwx, provide an interface to management operations such as scanning and joining networks, but most other operations are handled by net80211.

Older drivers have to implement more of the net80211 state machine themselves rather than reproducing a lot of very similar code.

All WiFi drivers exist at some point on this scale, ranging from firmware handling almost all the work to the OS managing most of the radio and transmission. The best way to see how this works practically is to look at a driver.

Let's start by looking at how a driver attaches and appears in the **net.wlan.devices** list, and then we will look into how a packet leaves the net80211 stack and heads toward the WiFi radio.

In this article, we will primarily focus on the **if_iwx** driver for a couple of reasons: I am very familiar with it, having brought the driver into the tree from the Future Crew source release, and as a new driver, there is still a lot to do in terms of low-hanging fruit.

All WiFi drivers exist at some point on this scale, ranging from firmware handling almost all the work to the OS managing most of the radio and transmission.

Connecting a driver to hardware

The life cycle of a driver is usually:

- probe
- attach <do some work>
- detach

Many drivers only ever detach when the machine is turned off. The probe and attach phases are where we need to start to add support for new hardware to an existing driver or when adding a new driver.

When a device is discovered by a bus, the bus will ask each registered driver if it can work with that hardware. Once all drivers have been queried, the bus will, in probe response order, ask the driver if it can **attach** to the device. The first device to attach **wins**.

At some point in the future, the driver will have to stop playing and go home. This can happen due to a bus error, if a device is removed, such as a USB, or if the system shuts down or restarts.

For each of these phases, a callback is registered with the bus. As an example here is the **pci_methods** struct from **if_iwx.c**

```
static device_method_t iwx_pci_methods[] = {
        /* Device interface */
        DEVMETHOD(device_probe,         iwx_probe),
        DEVMETHOD(device_attach,        iwx_attach),
        DEVMETHOD(device_detach,        iwx_detach),
        DEVMETHOD(device_suspend,       iwx_suspend),
        DEVMETHOD(device_resume,        iwx_resume),

        DEVMETHOD_END
};
```

if_iwx registers probe, attach, and detach methods, and suspend and resume methods. All of which are called when needed.

Probe

WiFi devices are typically made from a chipset and some supporting hardware. The chipset is made by a company such as Realtek or Intel, but the actual device is manufac-

tured around the chipset by another company. This arrangement means we get rtwn-based devices made by a company such as TP-Link. The company building a device around the chipset provides drivers and configuration data, resulting in a larger number of device IDs being supported by a smaller number of drivers.

This means that a very common first patch for a new FreeBSD contributor is to add a device ID for something not yet covered (my first patch was a flash chip ID in a MIPS router!).

Your first change in FreeBSD WiFi could be straightforward, buy a device you think should work and test it (follow the instructions from the first article in this series).

If no driver probes for the hardware, you can list out the USB or PCIe device IDs and use those to determine from other platforms which driver should support them.

Two recent changes I committed to FreeBSD for an external contributor were just this, adding device IDs for hardware supported by **`if_run`** and **`if_rum`**. The driver part for the run change was:

```
diff --git a/sys/dev/usb/wlan/if_run.c b/sys/dev/usb/wlan/if_run.c
index 00e005fd7d4d..97c790dd5b81 100644
--- a/sys/dev/usb/wlan/if_run.c
+++ b/sys/dev/usb/wlan/if_run.c
@@ -324,6 +324,7 @@ static const STRUCT_USB_HOST_ID run_devs[] = {
     RUN_DEV(SITECOMEU,          RT2870_3),
     RUN_DEV(SITECOMEU,          RT2870_4),
     RUN_DEV(SITECOMEU,          RT3070),
+    RUN_DEV(SITECOMEU,          RT3070_1),
     RUN_DEV(SITECOMEU,          RT3070_2),
     RUN_DEV(SITECOMEU,          RT3070_3),
     RUN_DEV(SITECOMEU,          RT3070_4),
```

Your first FreeBSD change could be as simple as adding a single line to the device IDs for a device. Once you have that, you need to add an entry to the relevant driver, test it, email me thj@freebsd.org a diff, and I will commit it.

Attaching to net80211

The state required for a WiFi driver is stored in an ieee80211com variable (usually called the ic) on the driver's softc.

A driver uses the IC to set flags for capabilities and override function pointers to hook or replace default functionality provided by the net80211 stack.

In the previous article in this series, I showed you how to create virtual WLAN interfaces (VAPs) on top of a driver using the **`ifconfig`** command. VAPs allow us to have multiple interfaces on top of a single card operating in different modes, sta, host ap, monitor, etc. The driver manages the availability of each of these modes using the **`ic_caps`** bit field.

The values in this field are set as part of the driver attach process. Here is an example from the **`iwx_attach`** function from the **`if_iwx`** driver:

```
...
ic->ic_softc = sc;
ic->ic_name = device_get_nameunit(sc->sc_dev);
ic->ic_phytype = IEEE80211_T_OFDM;  /* not only, but not used */
ic->ic_opmode = IEEE80211_M_STA;    /* default to BSS mode */
```

```
/* Set device capabilities. */
ic->ic_caps =
    IEEE80211_C_STA |
    IEEE80211_C_MONITOR |
    IEEE80211_C_WPA |            /* WPA/RSN */
    IEEE80211_C_WME |
    IEEE80211_C_PMGT |
    IEEE80211_C_SHSLOT |        /* short slot time supported */
    IEEE80211_C_SHPREAMBLE |    /* short preamble supported */
    IEEE80211_C_BGSCAN          /* capable of bg scanning */
...
```

attach from `if_iwx`

This snippet of code resides near the end of the attach method in `if_iwx`. The preceding attach code performs driver housekeeping state with independent tasks, discovering which PCIe device this is, and determining the exact Intel Wireless model of the card.

`if_iwx` supports the station mode (`IEEE80211_C_STA`) and monitor modes (`IEEE80211_C_MONITOR`); if the driver supported hosts AP mode (as rtwn does), it would have the additional `IEEE80211_C_HOSTAP` flag in its capability bit mask.

Beyond modes iwx supports: WPA encryption (`IEEE80211_C_WPA`), multimedia extensions for differential service (`IEEE80211_C_WME`), power management (`IEEE80211_C_PMGT`), short time slots (`IEEE80211_C_SHSLOT`), preambles (`IEEE80211_C_SHPREAMBLE`), and background scanning (`IEEE80211_C_BGSCAN`).

The complete list of capabilities lives in the ieee80211.h header files. The capabilities a driver can advertise depend on both hardware features and support in the driver. While a driver is in development, it might not yet implement features such as WPA offload, so just because a flag is missing in a driver, it doesn't mean the hardware feature is unavailable.

The second task performed by the driver attachment phase is to take over or implement net80211 functions, which is done through the `iwx_attach_hook` configuration callback. Here, the driver overrides function pointers for a lot of the features advertised by the `ic_caps` bit field.

First, `if_iwx` creates the channel map. For this card, the driver must ask the card's firmware to provide a set of supported channels.

```
iwx_init_channel_map(ic, IEEE80211_CHAN_MAX, &ic->ic_nchans,
        ic->ic_channels);

ieee80211_ifattach(ic);
ic->ic_vap_create = iwx_vap_create;
ic->ic_vap_delete = iwx_vap_delete;
ic->ic_raw_xmit = iwx_raw_xmit;
ic->ic_node_alloc = iwx_node_alloc;
ic->ic_scan_start = iwx_scan_start;
ic->ic_scan_end = iwx_scan_end;
ic->ic_update_mcast = iwx_update_mcast;
```

```
ic->ic_getradiocaps = iwx_init_channel_map;

ic->ic_set_channel = iwx_set_channel;
ic->ic_scan_curchan = iwx_scan_curchan;
ic->ic_scan_mindwell = iwx_scan_mindwell;
ic->ic_wme.wme_update = iwx_wme_update;
ic->ic_parent = iwx_parent;
ic->ic_transmit = iwx_transmit;

sc->sc_ampdu_rx_start = ic->ic_ampdu_rx_start;
ic->ic_ampdu_rx_start = iwx_ampdu_rx_start;
sc->sc_ampdu_rx_stop = ic->ic_ampdu_rx_stop;
ic->ic_ampdu_rx_stop = iwx_ampdu_rx_stop;

sc->sc_addba_request = ic->ic_addba_request;
ic->ic_addba_request = iwx_addba_request;
sc->sc_addba_response = ic->ic_addba_response;
ic->ic_addba_response = iwx_addba_response;

iwx_radiotap_attach(sc);
ieee80211_announce(ic);
```

Then the driver either replaces or intercepts calls that net80211 will make using the device's IC. Implementations are provided for **`ic_vap_create`** and **`ic_raw_xmit`**, but other calls, such as **`sc_ampdu_rx_start`** and **`stop`**, are intercepted.

Finally, the driver attaches to the radiotap subsystems, which allows raw packets to be fed to BPF and then announces the existence of the driver to the net80211 system.

The two **`ieee80211_`** calls in the attach methods are examples of our interface to the net80211 system. The first call attaches our driver to the net80211 subsystem (it is here that we get added to the list behind the **`net.wlan.devices`** sysctl). This makes the driver available for **`ifconfig`** to use.

The second call (**`ieee80211_announce`**) handles declaring that the device has been created; this is where we print the channel and feature support for the card.

Once the driver has attached to the net80211 subsystem, it will idle until external events cause it to move into an operating state. The next part of operating is handled by net80211, and it calls out to the hooked methods we overrode in the **`attach_hook`** callback.

Implementing station mode

In the first article, we created a station mode VAP for our first example. The command we ran was:

```
ifconfig wlan create wlandev iwx0
```

The **`wlan`** argument lets the system allocate a device number for us, and the **`iwx0`** tells the net80211 subsystem to use the device called **`iwx0`** to create this VAP.

This command is translated by **`ifconfig`** via a library to a **`net80211_ioctl`** call. The final result is net802211 calling the **`ic->ic_vap_create`** callback on our drivers **`ic`**. From above, you know that this is mapped to **`iwx_vap_create`**.

```
struct ieee80211vap *
iwx_vap_create(struct ieee80211com *ic, const char name[IFNAMSIZ], int unit,
    enum ieee80211_opmode opmode, int flags,
    const uint8_t bssid[IEEE80211_ADDR_LEN],
    const uint8_t mac[IEEE80211_ADDR_LEN])
{
        struct iwx_vap *ivp;
        struct ieee80211vap *vap;
        if (!TAILQ_EMPTY(&ic->ic_vaps))         /* only one at a time */
                return NULL;
ivp = malloc(sizeof(struct iwx_vap), M_80211_VAP, M_WAITOK | M_ZERO);
vap = &ivp->iv_vap;
ieee80211_vap_setup(ic, vap, name, unit, opmode, flags, bssid);
vap->iv_bmissthreshold = 10;            /* override default */
/* Override with driver methods. */
ivp->iv_newstate = vap->iv_newstate;
vap->iv_newstate = iwx_newstate;

ivp->id = IWX_DEFAULT_MACID;
ivp->color = IWX_DEFAULT_COLOR;

ivp->have_wme = TRUE;
ivp->ps_disabled = FALSE;

vap->iv_ampdu_rxmax = IEEE80211_HTCAP_MAXRXAMPDU_64K;
vap->iv_ampdu_density = IEEE80211_HTCAP_MPDUDENSITY_4;

/* h/w crypto support */
vap->iv_key_alloc = iwx_key_alloc;
vap->iv_key_delete = iwx_key_delete;
vap->iv_key_set = iwx_key_set;
vap->iv_key_update_begin = iwx_key_update_begin;
vap->iv_key_update_end = iwx_key_update_end;

ieee80211_ratectl_init(vap);
/* Complete setup. */
        ieee80211_vap_attach(vap, ieee80211_media_change,
            ieee80211_media_status, mac);
        ic->ic_opmode = opmode;

        return vap;
}
```

The `iwx_vap_create` performs some housekeeping to manage memory and establishes callbacks to be used by the net80211 system. For `iwx`, it establishes per-driver state (the `IWX_DEFAULT_MACID` and `IWX_DEFAULT_COLOR` values), which is used to coordinate with firmware about which station we use as a default.

For some functions that **iwx_vap_create** hooks into, we retain the default method and intercept calls to it. For instance, we override the **iv_newstate** callback and filter it through **iwx_newstate**.

The firmware for **iwx** manages a lot of state itself; one example is probing, where the hardware can be asked to send probes for networks across supported channels, and we aren't able to send these packets directly ourselves.

The iwx driver must hook the newstate methods to make requests to the firmware, updating its state machine. In this way, the net80211 and firmware state machines are kept in sync with host-level changes.

Sending packets

We have now covered enough of the driver that we can bring it up with **ifconfig** and ask the operating system to start sending packets.

When we are testing an interface, we might go through the following flow using ifconfig:

```
# ifconfig wlan0 ssid open-network up
```

These commands instruct **ifconfig** to bring up the interface and request that the net80211 stack join the open WiFi network **open-network**. It sets an address for the interface, but this doesn't lead to any packets on the wire (well, air).

Let's see what driver methods this series of commands translates into.

In our attach hook, we established two callbacks for the net80211 layer to use when it needs to send a packet: **ic_transmit** and **ic_raw_transmit**, and one to control the state of the interface (**ic_parent**).

```
ic->ic_raw_xmit = iwx_raw_xmit;
...
ic->ic_parent = iwx_parent;
ic->ic_transmit = iwx_transmit;
```

The **up** part of the **ifconfig** command eventually calls the **ic_parent** callback. For iwx, this is **iwx_parent**:

```
static void
iwx_parent(struct ieee80211com *ic)
{
        struct iwx_softc *sc = ic->ic_softc;
        IWX_LOCK(sc);

        if (sc->sc_flags & IWX_FLAG_HW_INITED) {
                iwx_stop(sc);
                sc->sc_flags &= ~IWX_FLAG_HW_INITED;
        } else {
                iwx_init(sc);
                ieee80211_start_all(ic);
        }
        IWX_UNLOCK(sc);
}
```

`iwx_parent` directly controls the hardware, calling a function to tear down all hardware state if we are running `iwx_stop`, or if we aren't running yet, asking the hardware to be initially configured with `iwx_init`. Once the hardware is ready, we then notify the net80211 stack that we are prepared to start with `ieee80211_start_all`.

The seemingly simple `ifconfig` action `up` results in a lot of hardware state being modified with the `iwx` driver. This contributes partially to "bringing the interface up and down" being a suggested magic fix to resolve network inconsistencies.

The second part of the `ifconfig` command results in the net80211 stack taking quite a few steps. By passing `ssid open-network` to `ifconfig`, we are asking the net80211 subsystem to discover and join a network called `open-network`.

The IEEE 802.11 process to join a network is made up of several steps:

- probe for a network
- authenticate to the network
- associate with the network

Each of these steps requires a device to send management frames. First, we need to discover the network we want to join; networks regularly beacon their presence (this is what fills the network list in your menu bar). This gives the operating system a list of networks to try. When a device wants to join a network, it sends out a probe request for the target network and waits for a probe response. This process facilitates the transfer of configuration parameters between the network and the host, indicating to the host that the network is truly available.

By passing ssid open-network to ifconfig, we are asking the net80211 subsystem to discover and join a network called open-network.

The next step involves authentication to the network, followed by association. At this point, we move into the `RUN` state and can start using the wireless interface like any other network device.

As the stack moves between each state, it triggers a call to the `iv_newstate` function, which for `iwx` is first intercepted by `iwx_newstate`. This allows the driver to control the sending of packets for state transitions. We need this in `iwx` because some of these transitions are handled by the device firmware rather than through direct packet transmission.

Rather than sending out probe requests directly, there is a firmware interface to trigger a scan of available networks. Once we have discovered the network and want to join it, we send a message to the firmware to add a station rather than sending out packets from the net80211 stack.

Not all management frames are sent by the firmware via an abstraction, and in those cases, the `iwx_raw_xmit` callback is used by the system. If you are debugging a driver and wondering why the transmit path isn't always hit, it could be management frames exiting the raw path.

Conclusion

In this article, we have looked at how a driver probes, attaches, and sends some first packets. By using an existing driver, we can cover a lot of ground in the driver quite quickly.

However, if you read through, you will see that `if_iwx.c` is a whopping 10,000 lines of code. That is more than we can address here.

This article, which has started to dig into hacking, has also glossed over many details. To join a network, we need to be able to both send and receive packets from the network interface.

If we don't get any packets, can we debug? What is offered by the system?

In Part 3 of the series, we will cover the built-in debugging features of the net80211 stack and how they hook into a driver for developer, testing, and troubleshooting.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.