FreeBSD WiFi Development

Part 1 – Experimenting with WiFi

BY TOM JONES

I have been working on WiFi in FreeBSD for the last 6 months through a project sponsored by the FreeBSD Foundation. The main result of this has been a port from OpenBSD of the `iwx` driver for Intel 80211ac/ax cards. Through this project, I have been exposed to most of the WiFi stack, and it has made clear to me that we need many more people working on WiFi and an easier route to starting to do development.

WiFi arrived 25 years ago and has become so core to our lives that people's first question when entering a new place is "Is there WiFi?"

FreeBSD lacks WiFi drivers. The last 5 years have seen the introduction of a linuxkpi layer with support for Wifi and by 2025 these drivers are starting to manage IEEE 802.11ac speeds in the hundreds of megabits. WiFi 7 devices on the market today do 2Gbit.

FreeBSD is behind, there is no argument. We need more people to work on WiFi. If you are interested in doing operating system development, I can think of nothing better than getting involved with improving the FreeBSD WiFi stack. It isn't going to be the easiest thing to do — but the thing about being behind is that we have a lot of open tasks of all difficulties and we need more contributors to catch up.

This series explains a path to working on WiFi in FreeBSD.. The raw nuts and bolts of building drivers and watching LLVM warm up your room are covered well elsewhere. In this first article of three, I will explain terms and demonstrate how to configure WLAN interfaces on FreeBSD for testing. This minimum setup is enough to start finding bugs. In following articles I will discuss what WiFi drivers do and how they interact with the larger `net80211` stack in FreeBSD.

WiFi arrived 25 years ago and has become so core to our lives that people's first question when entering a new place is "Is there WiFi?"

Terminology

Networking is about communication - communication is the effective transmission and accurate reception of a message. I've worked in developing and writing internet standards for a decade, and still, for communications experts we aren't the best at completing the communication story. From an IETF background, WiFi is that weird IEEE stuff concerned with making electrons dance.

To communicate well, we need to agree on what terms mean, which reminds me of a conversation in the University of Oslo Cafeteria many years ago with a German colleague.

Me: *I'm pretty sure it is why-phi you know, wireless fidelity, like high fidelity.*
Him: *No, it is wee-fee like hee-fee*

I am going to use a lot of acronyms and terms in these articles; it just can't be helped. Searching the Internet is your friend when it comes to figuring these out. Of the two I just used IETF (Internet Engineering Task Force) and IEEE (Institute of Electrical and Electronics Engineers) only IEEE will feature heavily here. Sadly throughout documentation and code there are variations of IEEE, net80211, and IEEE80211 repeatedly, there are a variety of terms used in the FreeBSD source code for WiFi infrastructure.

The WiFi Alliance (the people who sell certification) has made a mess between the standards (from IEEE) and the brand names. Using the standards is more correct, `IEEE80211n` (or `11n`, or `n`) is clearer for me than using WiFi 4. Either term might help you figure out what things are called, what a product does, or what someone is asking you about. FreeBSD is going to tend toward standard names, or even document revisions if we are lucky, rather than the marketing names.

The WiFi Alliance (the people who sell certification) has made a mess between the standards (from IEEE) and the brand names.

Getting up to speed with concepts in WiFi is also going to be as difficult as it is with names. I suggest some reading, Matthew Ghast's first book on WiFi (*802.11 Wireless Networks*) is a great introduction to all the main concepts at a suitable introductory level. Don't mind the age, the foundations are still the same, the numbers are just bigger, and the modulations are more complex.

WiFi	A collection of standards by the IEEE and marketing names by the WiFi alliance. Used colloquially it is "that thing your laptop uses to go on the internet" and that is good enough for us. Anyone correcting your terminology too intensely isn't your friend.
IEEE80211	IEEE 802.11 is the family of standards that defines WiFi.
net80211	Or the stack. The code in FreeBSD that implements the IEEE80211 state machine.
Band	A frequency range that a client or access point may use. (these are fractal 2.4GHz 'band', 2462 MHz band)
Channel	An RF frequency and parameters. Sometimes used interchangeably with 'band'
Station	Your device, other clients on a network (also a mode)
Access Point	The device running the network you connect to (also a mode)
Monitor	A mode which causes the network adapter to capture all packets on a band.

Network Adapter	The device with all the radios which enables you to do WiFi (also, a network card – though a USB stick isn't a card, might be a network interface, but best to avoid confusion between the hardware and the software model).
Driver	Code in FreeBSD that speaks to a network card either directly or via firmware
Firmware	Code that runs on the network card and does some of the work for you. We normally only mention it when it needs to be loaded by the OS or a driver.
MAC	The part of the 80211 machine which handles sharing the medium
Full MAC	Used in reference to drivers – full mac devices implement the MAC layer and much of the work that net80211 does can be skipped.
HT	High Throughput (also called 80211n)
VHT	Very High Throughput (also called 80211ac)
EHT	Extremely High Throughput (also called 80211ax)

There are also some core concepts you need to be familiar with for this article. I think, for most people, this is how their day-to-day network access works, but it is good to recognize that technologies some of us grew up with (me) or were introduced in our lifetimes (a bunch of people I've just made feel old), have existed for a decade longer that some readers have been alive.

The most common WiFi network found in people's homes is an access point (AP), which acts as the gateway to the larger Internet for stations (or clients). In many deployments, likely your home, the access point is a router that handles forwarding traffic from clients to the internet (maybe via a modem), assigning addresses, and a host of other network tasks.

To join a network, a station goes through a series of states while communicating with the access point. In summary, it will:

- scan
- probe
- receive beacons
- authenticate
- associate
- negotiate encryption keys
- acquire an IP address

All but the final steps here are WiFi-specific, in some ways they model you finding a cable, a port, and plugging your computer into the router. The final step happens at the IP layer, and we use the same tools as in wired networks.

All these phases and states are handled in the `net80211` stack and device drivers. Who does what depends on hardware support, some features are handled by firmware on the network adapter. When they can't be done in hardware, the `net80211` stack can implement most things itself, some features that use radio functionally can't be emulated in software.

As we do WiFi development on FreeBSD, we need to be aware of what the hardware and `net80211` layers are doing. Above is a short summary, but there are many other states and authentication modes. `IEEE80211` is approaching 30 and it has a lot of history.

Experimenting with WiFi on FreeBSD

Before we start reading code and making changes, we should discuss how to do management operations on WLAN adapters in FreeBSD.

The **net80211** stack provides an abstraction on top of network adapters which gives us virtual interfaces. Throughout the code, they are called VAPs (see the ieee80211_vap man page for a full description of their functionality).

This means that we first must create a WLAN interface from a device before we can use it.

This may seem clunky compared to the ease of management of an interface in OpenBSD (**ifconfig iwx0 up**), but it enables virtual functionality to be implemented on top of a single adapter. If the hardware supports it, you might be able to be an access point and a station at the same time, or a station and in monitor mode simultaneously.

Usually, this management is handled for you by some configuration in rc.conf, the installer adds lines like these:

```
wlans_iwlwifi0="wlan0"
ifconfig_wlan0="WPA SYNCDHCP"
```

The first line tells the **rc** system to create an interface from the **iwlwifi0** adapter called wlan0, the second line is a usual **ifconfig** line you would see for a wired interface.

Kernel development can be helped by controlling the creation of devices so let's first look at how we can manually create interfaces.

Manually Creating Interfaces

WLAN devices that are registered with **net80211** as devices can be listed by reading the **net.wlan.devices sysctl.**

```
$ sysctl net.wlan.devices
net.wlan.devices: iwx0 rtwn0
```

On my laptop, you can see I have an **iwx**-based card attached (**iwx0** on PCIe) and a **rtwn** card (**rtwn0** on USB).

From these devices, I can create interfaces using ifconfig like so:

```
# ifconfig wlan create wlandev iwx0
wlan0
# ifconfig wlan create wlandev rtwn0 wlanmode ap
wlan1
# ifconfig wlan create wlandev rtwn0 wlanmode monitor
wlan2
```

In the first example, we create **wlan0** without any arguments. Station mode is the default when creating a device without any wlanmode argument. WLAN devices in FreeBSD operate in one and only one mode set at creation time, the possible modes for any given device are governed by hardware and driver support.

The modes a driver supports are listed on the corresponding man page. If we compare **iwm**, **iwlwifi**, and **iwx** (which support some of the same hardware) we will see that iwm and iwlwifi at the time of writing can only operate in station mode, whereas **iwx** can operation in station and monitor mode. The Intel hardware that **iwx** supports can work as a host AP, but in a very limited way (only on the 2.4GHz band) and so support isn't yet implemented.

The **rtwn** driver supports station, adhoc, host AP, and monitor mode operation. The full list of modes supported by **net80211** is documented in the ifconfig(8) man page:

```
wlanmode mode
        Specify the operating mode for this cloned device.  mode is one
        of sta, ahdemo (or adhoc-demo), ibss (or adhoc), AP (or hostap),
        wds, tdma, mesh, and monitor.  The operating mode of a cloned
        interface cannot be changed.  The tdma mode is actually
        implemented as an adhoc-demo interface with special properties.
```

Right after configuration **ifconfig** will show us an interface with lots of information unpopulated:

```
$ ifconfig wlan0
wlan0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=0
        ether e4:5e:37:af:13:5b
        groups: wlan
        ssid "" channel 1 (2412 MHz 11b)
        regdomain FCC country US authmode OPEN privacy OFF txpower 30
        bmiss 10 scanvalid 60 bgscan bgscanintvl 300 bgscanidle 250
        roam:rssi 7 roam:rate 1 wme bintval 0
        parent interface: iwx0
        media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
        status: no carrier
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```

We have quite a few extra parameters in the **ifconfig** output compared to a wired device, let's look at some of these.

```
ssid "" channel 1 (2412 MHz 11b)
```

As we are a station, we have a nominated SSID (network name to connect to), or here nothing. We are on channel 1 with the frequency and the channel mode listed.

```
regdomain FCC country US authmode OPEN privacy OFF txpower 30
```

The regulatory domain and country have been set to a default but should update when we bring the interface up to match my regulator domain and country.

```
bmiss 10 scanvalid 60 bgscan bgscanintvl 300 bgscanidle 250
roam:rssi 7 roam:rate 1 wme bintval 0
```

We have some driver parameters that control scanning, moving between networks, and multimedia extensions (which are used for quality-of-service information not using your adapter to play MP3s).

```
parent interface: iwx0
media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
```

Finally, we have the parent interface (handy if you are doing a lot of WiFi) and the current media mode. This should normally be autoselect, but you might be forcing this to debug or get better throughput.

If we look at the VAPs created on the rtwn interfaces, they are similar but different, which is due to their different mode.

```
$ ifconfig wlan1
wlan1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=0
        ether 74:da:38:33:c0:62
        groups: wlan
        ssid "" channel 1 (2412 MHz 11b)
        regdomain FCC country US authmode OPEN privacy OFF txpower 30
        scanvalid 60 wme dtimperiod 1 -dfs bintval 0
        parent interface: rtwn0
        media: IEEE 802.11 Wireless Ethernet autoselect <hostap> (autoselect <hostap>)
        status: no carrier
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
$ ifconfig wlan2
wlan2: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=0
        ether 74:da:38:33:c0:62
        groups: wlan
        ssid "" channel 1 (2412 MHz 11b)
        regdomain FCC country US authmode OPEN privacy OFF txpower 30
        scanvalid 60 wme bintval 0
        parent interface: rtwn0
        media: IEEE 802.11 Wireless Ethernet autoselect <monitor> (autoselect <monitor>)
        status: no carrier
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```

When we are done, or if we accidentally created a device in the wrong mode, we could remove it using ifconfig:

```
# ifconfig wlan0 destroy
```

Using Interfaces

I lied earlier, well sort of. The FreeBSD WiFi stack has two main components, but a lot of WiFi state is driven by two userspace programs, **wpa_supplicant** and **hostapd**.

Station mode with `wpa_supplicant`

wpa_supplicant was originally a program for managing WPA (wireless protected access) encryption state for a device in station mode. It has grown to be a full userspace WiFi management interface. It is one of a few implementations and it is common to see **wpa_supplicant** used on Linux for wireless configuration.

hostapd is the host AP userspace daemon from the same project — it corresponds to **wpa_supplicant**, but rather than doing client tasks it does host tasks.

We can use ifconfig to manage a WLAN interface, the following command will bring up our station interface and configure it to join the "Test" SSID.

```
# ifconfig wlan0 ssid "Test" up
```

This allows a FreeBSD station to start trying to associate with an AP called "Test". The FreeBSD **net80211** stack doesn't directly support the WPA state machine, so for most networks you will need to use **wpa_supplicant**. The second line from our example rc.conf above handles starting **wpa_supplicant** on an interface (and enabling DHCP).

To manually run **wpa_supplicant** we need a configuration file with our networks, the **wpa_supplicant.conf** man page has some great examples, but a bare minimum configuration file looks like this:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=wheel

network={
        ssid="Open Network"
        key_mgmt=NONE
}
```

We can then start **wpa_supplicant** manually like so:

```
# wpa_supplicant -i wlan0 -c /etc/wpa_supplicant.conf -D bsd
```

The default invocation will run **wpa_supplicant** in the foreground and adding **-B** will run it in the background. The logged messages from **wpa_supplicant** are available via the control interface.

The **wpa_passphrase** can be used to add a WPA-protected network to the **wpa_supplicant** configuration file:

```
$ wpa_passphrase "Closed Network" superpassword
network={
        ssid="Closed Network"
        #psk="superpassword"
        psk=852c26a07d84c48e4bfeec71289214a39bcd9d881bc66aedf6a2d11372f59752
}
```

It only generates the required configuration to add to **wpa_supplicant.conf**, to not have to learn a lot of configuration syntax.

FreeBSD ships with **wpa_cli** to interface with **wpa_supplicant**. From **wpa_cli** we can list networks, connect (select), reconfigure, and disconnect. Here is an example session joining a WPA-protected network.

```
$ wpa_cli
wpa_cli v2.11
Copyright (c) 2004-2024, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.


Selected interface ‘wlan0’

Interactive mode
```

```
> list_networks
network id / ssid / bssid / flags
0       Open Network        [DISABLED]
1       Closed Network      [DISABLED]
> select_network 1
OK
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>Trying to associate with 20:05:b6:fa:13:f1 (SSID='Closed Network' freq=5180 MHz)
<3>Associated with 20:05:b6:fa:13:f1
<3>WPA: Key negotiation completed with 20:05:b6:fa:13:f1 [PTK=CCMP GTK=CCMP]
<3>CTRL-EVENT-CONNECTED - Connection to 20:05:b6:fa:13:f1 completed [id=0 id_str=]
disable_network disconnect
> disconnect
OK
<3>CTRL-EVENT-DISCONNECTED bssid=20:05:b6:fa:13:f1 reason=3 locally_generated=1
<3>CTRL-EVENT-DSCP-POLICY clear_all
```

AP mode with hostapd

hostapd is less used, and the ability to bring up your own AP in which you can do full kernel debugging and packet capture can be a very helpful debugging tool.

An example configuration for our example "Closed Network" might look like the following:

hostapd.conf:

```
ctrl_interface=/var/run/hostapd
ctrl_interface_group=wheel

interface=wlan1

# hw_mode=g
channel=8
ieee80211d=0
ieee80211n=0
wmm_enabled=0

# the name of the AP
ssid="Closed Network"
# 1=wpa, 2=wep, 3=both
auth_algs=1
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
wpa_passphrase="superpassword"
```

We can run hostapd in the foreground like so:

```
# hostapd hostapd.conf
```

With hostapd running we have most of an access point — the WiFi bit is easy! We also need to give **wlan1** an address and usually a process runs to provide dynamic addressing.

```
We can give the interface an IP address like normal:
# ifconfig wlan1 inet 192.168.2.1/24 up
```

For dynamic addressing, we need to install the dhcpd port and create a configuration file. A minimal configuration file can be found in the dhcpd.conf(5) man page and might look like this:

```
/usr/local/etc/dhcpd.conf:

subnet 192.168.2.0 netmask 255.255.255.0 {
    range 192.168.2.100 192.168.2.200
}
```

We can then enable and start dhcpd:

```
# service enable dhcpd
# service start dhcpd
```

Monitor mode

The final mode VAP we created in our example was a monitor mode VAP. While we can capture on VAPs in other modes, the interface is not run in promiscuous mode, the packets we receive are those that arrive with an address for our interface. Monitor mode allows us to receive all the packets on a channel (hardware support depending on different rates).

A simple proof of concept is to use **tcpdump** with the **-y** link level headers flag.

```
# tcpdump -L -i wlan2
Data link types for wlan0 (use option -y to set):
  EN10MB (Ethernet)
  IEEE802_11_RADIO (802.11 plus radiotap header)
```

I struggle to remember the order of the underscores in this variable, but tcpdump will show you the supported link-level headers for an interface type with the **-L** flag and the interface.

```
# sudo tcpdump -i wlan2 -y IEEE802_11_RADIO
tcpdump: data link type IEEE802_11_RADIO
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on wlan2, link-type IEEE802_11_RADIO (802.11 plus radiotap header), snapshot
length 262144 bytes
14:33:48.656430 3757399us tsft 1.0 Mb/s 2412 MHz 11g -76dBm signal -95dBm noise Data
IV:c1ed Pad 20 KeyID 1
14:33:50.657087 5759270us tsft 1.0 Mb/s 2412 MHz 11g -72dBm signal -95dBm noise
14:33:50.796280 5895802us tsft 1.0 Mb/s 2412 MHz 11g -76dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
14:33:53.151514 8251009us tsft 1.0 Mb/s 2412 MHz 11g -74dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
14:33:53.970729 9070213us tsft 1.0 Mb/s 2412 MHz 11g -74dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
```

```
14:34:10.183336 25285260us tsft 6.0 Mb/s 2437 MHz 11g -62dBm signal -95dBm noise Beacon
(a2-enc) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS CH: 6, PRIVACY
14:34:10.204045 25306099us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6
14:34:10.253438 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0* 9.0 12.0* 18.0 Mbit] IBSS CH: 6, PRIVACY
14:34:10.253441 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6
14:34:10.253445 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon
(HM-CM-$tte, HM-CM-$tte, kette) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6,
PRIVACY
14:34:10.285355 25387273us tsft 6.0 Mb/s 2437 MHz 11g -63dBm signal -95dBm noise Beacon
(a2-enc) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS CH: 6, PRIVACY
14:34:10.297973 25399988us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise Beacon
(HM-CM-$tte, HM-CM-$tte, adkette) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6,
PRIVACY
14:34:10.355834 25458704us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0* 9.0 12.0* 18.0 Mbit] IBSS CH: 6, PRIVACY
14:34:10.355836 25458704us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6
```

This **tcpdump** output is focused on the IEEE80211 radio frames. In-depth analysis will require a different tool such as Wireshark.

Testing Traffic

Now that we have all the pieces to build a purely FreeBSD AP station and debug the traffic from the air, we should talk about how to test things.

We can use our AP station and monitor mode to verify and investigate the packets that are sent during an association and sending traffic.

The first stage of testing is getting the station onto the network to the point where we can send pings to the AP. Using **wpa_supplicant** from the station we can select the network and request that **wpa_supplicant** associate for us. **wpa_supplicant** will print messages as it does this, and it will document the hand packets.

Once on the network, the station needs to request an IP address. If this doesn't happen automatically then running:

```
# dclient wlan0
```

will normally kick it to working. If you get a dhcpd lease (an address) then the next step is to try and ping the AP:

```
# ping 192.168.2.1
```

Any failures here are places to start debugging. You will likely have configuration issues (I can't promise the examples are perfect, but they are tested). Debugging why you can't connect isn't the most enjoyable WiFi development experience, but it is one we have all encountered.

Once we have an address, simple throughput tests are usually a good first metric. If you are testing patches from a branch, it might be enough to build a kernel with the patches and run a network throughput test. I like **iperf3** for this sort of testing, it allows you to test

throughput to a host on your network or the Internet. Traffic from `iperf` on a station to `iperf` on the app will give you a good idea of what throughput is possible with that combination of hardware.

If your system is stable enough and can run a web browser, I find testing against fast.com to be helpful too.

These options test very different limitations, the `iperf` test will give you an understanding of the throughput available to a station on your WiFi network and the fast.com test will tell you what is possible from your network to the internet. You may get a significantly lower number with fast.com. If it is higher than the `iperf` test, then something is funky.

You will see differences in throughputs between TCP and UDP tests. UDP can better saturate WiFi radio. Testing both is good, but for a quick measurement, TCP throughput is fine.

Ready for Work

This article has covered the background and terminology you need to start hacking on WiFi in FreeBSD, but we haven't managed to look at any code. Setting up a test network with the examples here is a core part of doing WiFi development and if you are eager and don't want to wait for Part 2, I am sure that if you try this network with enough hardware, you will start to find bugs.

In the next installment, we will look at the life cycle of a WiFi driver, the core functions it needs to have to do its work, and the interfaces to net80211 that can do a lot of the work from the driver.

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