

Getting started with WiFi development

Part 3: Debugging

The FreeBSD WiFi stack is amenable to interrogation.

Tom Jones

In a series on starting WiFi development, the debugging part easily could have been the first, second, and third sections and we might never have looked at driver code.

WiFi has all the features that make debugging difficult: It is a concurrent distributed state machine; temporal physical and random effects change how things behave; it integrates the black magic of microwave RF design; and the hardware is proprietary, undocumented and unintuitive to use.

With all that said, FreeBSD has high performance WiFi and many people have contributed to driver and stack support over the last two decades, mostly in their spare time.

In the end, people work on WiFi because many of the debugging stages can be very satisfying once you start to make progress.

In this article we will look at two approaches to debugging WiFi on FreeBSD for driver development: passive packet capture and system logging.

Packet capture gives you an external view of the system. You can run capture on a third computer to get an understanding of what is happening (or isn't) between your AP and station. It can give a ground truth on what data has crossed the network independent of driver effects.

System logging gives us an insight into what FreeBSD thinks is happening based on the packets it has seen. Packets and events are logged at many places and we can see how traffic causes changes in the state machine. We can see what happens to traffic which passed on the air interface but disappeared once it went to the FreeBSD machine.

First pass Debugging

First let us start with common issue-focused debugging .

I think for most people when it comes to debugging WiFi they are wondering why their station cannot join a network. In most cases failure to join the network is going to be due to something really straightforward to find, and it usually feels like a silly mistake.

If you get the PSK wrong for a `wpa` network, `wpa_supplicant` will suggest to you that is the problem:

```
<3>Trying to associate with 94:83:c4:58:ed:32 (SSID='o2-enc' freq=2437 MHz)
<3>Associated with 94:83:c4:58:ed:32
<3>CTRL-EVENT-DISCONNECTED bssid=94:83:c4:58:ed:32 reason=0
<3>WPA: 4-Way Handshake failed - pre-shared key may be incorrect
<3>CTRL-EVENT-SSID-TEMP-DISABLED id=8 ssid="o2-enc" auth_failures=1 duration=10
    reason=WRONG_KEY
<3>Added BSSID 94:83:c4:58:ed:32 into ignore list, ignoring for 10 seconds
<3>CTRL-EVENT-SCAN-RESULTS
```

This example is the output of the `wpa_cli` tool. It gives you a direct interface to `wpa_supplicant` logs and all.

Typing the wrong key can't be solved by any tool, but generating the correct entry can be solved by using `wpa_passphrase` to generate a new entry.

```
$ wpa_passphrase "example network" "excellentpsk"
network={
    ssid="example network"
    #psk="excellentpsk"
    psk=27231323cedae4b8036557f8dcd5f38bfb6e0ac9a5ab894a4ee5104f7998fc93
}
```

Other issues when trying to join the network might be due to range or the card not seeing beacons advertising the network's presence. In the past, when you only had one device, this was difficult to figure out. But now we almost always have another WiFi device that can verify that the network does in fact work here, suggesting that it is something up with the configuration on the FreeBSD device.

The second type of problems people want to debug are performance issues. Either the network doesn't reach to where you want it to, or there is a

configuration issue leading to less performance than you expected, based on using other systems on the same network.

This is much harder to figure out and fewer people are going to be able to help. You probably don't know what is inside your walls and no stranger on the Internet is going to be able to make a good guess at what is going wrong. They can however help by looking at the symptoms.

Thankfully it isn't just mystical radio issues that lower performance, and there is some configuration and investigation you can do to help improve network speeds `wpa_cli` is a great first place to look when trying to figure out which networks we can actually see:

```
> scan_results
bssid / frequency / signal level / flags / ssid
c8:e3:06:4d:1d:83      2437    -81    [RSN-SAE-CCMP] [MESH]    19c14
c8:e3:06:5d:3d:48      2437    -81    [WPA2-PSK-CCMP] [ESS]    Computer Network
18:e8:b9:c2:18:1a      2412    -87    [WPA2-PSK+SAE-CCMP] [ESS]    HomeWifi
a8:e3:06:5d:fd:47      2437    -81    [ESS]
20:12:34:c6:18:bd      2412    -87    [ESS]    Guest-Wifi
```

The `scan_results` command displays the current networks that `wpa_supplicant` knows about. It gets these from the driver it is bound to via the `net80211` subsystem. If you are trying to join a network but it can't be seen here, then you might be out of luck. Try moving closer to the AP if you can and gradually walk away until signal drops off.

If you are tracing a performance issue then your first stop should be looking at the receive signal strength indicator (RSSI). If the signal level is too low then you now know that the device isn't near enough to the access point. Signal level is reported in dBm. This will always be negative. As a relative measure, this means the closer it is to 1, the higher the strength. A very low just-detectable signal will be about ~87dBm, and a high signal might be

about 50dBm. Many protocol features are restricted to the signal strength being a certain level.

The other configuration issue that might be limiting performance can be very difficult to debug. WiFi uses radio spectrum in a variety of bands. In some places this is all free to use for WiFi devices, but in other parts of the world WiFi might be a secondary user of the spectrum. This blocks out some bands for use. And in some parts of the world weather radar detection must be performed in the 5GHz band.

The collection of bands and their availabilities are called a **regulatory domain**. If there is a mismatch between the regulatory domain of the station and the access point, you might find that the device is limited to using older WiFi standards.

A mismatch can occur for many reasons. If you use your ISP-provided wireless router, then it *should* be configured correctly for where you are. But you might have bought a device from a vendor who doesn't know to which regdomain it is going, or you might take a travel router with you. In that case you should verify matching regulatory domains between the station, the access point, and neighbouring access points. Yes, it is possible that someone else's access point is configured wrong, or differently and the firmware in your station's card decides it is in a different part of the world.

You can check the stations regdomain with `ifconfig`:

```
$ ifconfig wlan0
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=0
    ether 74:da:38:33:c0:62
    groups: wlan
    ssid "" channel 10 (2457 MHz 11g)
    regdomain FCC country US authmode WPA1+WPA2/802.11i privacy MIXED
    deftxkey UNDEF txpower 30 bmiss 7 scanvalid 60 protmode CTS wme
    roaming MANUAL bintval 0
    parent interface: rtwn0
```

```
media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
status: no carrier
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```

Out of the box FreeBSD has put my mostly unconfigured rtwn device into the US regdomain:

```
regdomain FCC country US authmode WPA1+WPA2/802.11i privacy MIXED
```

ifconfig can also be used to check the regdomain of access points it can see, though the output is quite difficult to read:

```
$ ifconfig -v wlan0 scan results
SSID/MESH ID          BSSID          CHAN RATE    S:N    INT CAPS
HomeWifi              18:e8:b9:c2:18:1a  1   54M  -87:-95   100
EPS  SSID<HomeWifi>  RATES<B2,B4,B11,B22,12,18,24,36> DSPARMS<1> COUNTRY<GB 1-
13,20> ERP<0x0> RSN<v1 mc:AES-CCMP uc:AES-CCMP km:8021X-PSK+?> XRATES<48,72,96,108>
BSSLOAD<sta count 3, chan load 56, aac 18> RRM_ENCAPS<460573d000000c> HTCAP<cap
0x1ac param 0x1b mcsset[0-15] extcap 0x0 txbf 0x0 antenna 0x0> HTINFO<ctl 1,
8,4,0,0 basicmcs[]> OVERLAP_BSS<4a0e14000a002c01c8-> EXTCAP<7f080500000200000040>
WME<qosinfo 0x0 BE[aifsn 3 cwmin 4 cwmax 10 txop 0] BK[aifsn 7 cwmin 4 cwmax 10
txop 0] VO[aifsn 2 cwmin 3 cwmax 4 txop 94] VI[aifsn 2 cwmin 2 cwmax 3 txop 47]>
ATH<0x7fff> VEN<dd3900156d00010100010217e5810618e8->
VEN<ddl68cfd0040000490000030209720100-> VEN<dd088cfd00101020100>
```

The COUNTRY element in the caps field at the end of the scan shows where the access point thinks it is:

```
COUNTRY<GB 1 -13,20>
```

The solution to this is to specify the regulatory domain and country with ifconfig:

```
$ ifconfig wlan0 list countries
Country codes:
DEBUG Debug          ZW Zimbabwe         YE Yemen            VN Viet Nam
VE Venezuela         UZ Uzbekistan       UY Uruguay          US United States
GB United Kingdom    AE United Arab Emi  UA Ukraine          TR Turkey
TN Tunisia           TT Tobago           TH Thailand         TW Taiwan
SY Syria             CH Switzerland      SE Sweden           LK Sri Lanka
ES Spain             ZA South Africa     SI Slovenia         SK Slovak Republic
SG Singapore         SA Saudi Arabia     RU Russia           RO Romania
QA Quatar            PR Puerto Rico      PT Portugal         PL Poland
PH Phillipines       PE Peru             PA Panama           PK Pakistan
OM Oman              NO Norway           NZ New Zealand      NL Netherlands
NP Nepal             MA Morocco         MC Monaco           MX Mexico
MT Malta             MY Malaysia         MK Macedonia        MO Macau
LU Luxemborg         LT Lithuania        LI Liechtenstein    LB Lebanon
LV Latvia            KW Kuwait           K2 Korea Republic2 KR Korea Republic
KP North Korea       KZ Kazakhstan       JO Jordan           J5 Japan5
J4 Japan4            J3 Japan3           J2 Japan2           J1 Japan1
JP Japan             JM Jamaica          IT Italy            IL Israel
IE Ireland           IR Iran             ID Indonesia        IN India
IS Iceland           HU Hungary          HK Hong Kong        HN Honduras
GT Guatemala         GR Greece           DE Germany          GE Georgia
F2 France2           FR France           FI Finland          EE Estonia
SV El Salvador       EG Egypt           EC Ecuador          DO Dominican Repub
DK Denmark           CZ Czech Republic  CY Cyprus           HR Croatia
CR Costa Rica        CO Colombia         CN China            CL Chile
CA Canada            BG Bulgaria         BN Brunei           BR Brazil
BO Bolivia           BZ Belize          BE Belgium          BY Belarus
```

BD Bangladesh	BH Bahrain	AZ Azerbaijan	AT Austria
AU Australia	AM Armenia	AR Argentina	DZ Algeria
AL Albania			

Regulatory domains:

XC900M	GZ901	XR9	SR9
NONE	ROW	TAIWAN	KOREA
APAC3	APAC2	APAC	ETSI3
ETSI2	ETSI	JAPAN	FCC4
FCC3	FCC	DEBUG	

```
$ sudo ifconfig wlan0 down
$ sudo ifconfig wlan0 country GB regdomain ETSI
```

Packet Capture

Generating a packet capture (or a pcap) can be a helpful debugging step when you want to understand the traffic which is going over the air and the WiFi interface. A wlan interface in FreeBSD can generate pcaps at two levels, as the logical Ethernet device similar to if you used a wired interface such as em and at the link layer giving you access to the 80211 packet types. The first is what you will get when an interface is configured as a station, but a wlan interface in monitor mode will give you more information about the underlying link.

Back in [part 1](#) of this series we created three different VAPs (virtual access points) for our example hardware, station, host AP, and monitor mode.

A monitor mode VAP puts the underlying hardware into a promiscuous mode where all packets are received and fed to bpf with the link headers intact . This gives more link information as well as all the packets that can be heard on a channel, rather than those directed towards the device or the broadcast address.

In the driver, all packets can be tagged with information about the radio environment that were received, this is put in a field called the radio tap header.

The radiotap header tells us useful metadata about the state of the WiFi radio when the packet was received, things such as the receive strength (RSSI), the WiFi channel and the modulation. These fields can be key in understanding why (or more often why not) something is working.

In monitor mode we get all the packets the radio can receive. This itself can lead to a lot of clues about what is happening in the local rf environment.

tcpdump

There are two tools to use when debugging WiFi using packet captures and `tcpdump` is key in getting information with minimal overhead. `tcpdump` is a standard packet capture tool and ships in FreeBSD base.

Normally `tcpdump` shows us packet information at the ip layer with a little introspection to look at what packets are doing. To get full advantage on a WiFi interface we need to ask `tcpdump` to include link layer headers and specifically `IEEE802_11_RADIO`.

```
# ifconfig wlan create wlandev rtwn0 wlanmode monitor
```

With WiFi headers included, we no longer have a view into the traffic but we get a lot of additional fields describing the radio environment and information about the mac layer of ieee80211.

```
$ sudo tcpdump -i wlan0 -y IEEE802_11_RADIO
tcpdump: data link type IEEE802_11_RADIO
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode listening
on wlan0, link-type IEEE802_11_RADIO (802.11 plus radiotap header), snapshot length
262144 bytes

10:27:07.724396 74143272us tsft 1.0 Mb/s 2437 MHz 11g -32dBm signal -95dBm noise
Beacon (o2-enc) [1.0* 2.0* 5.5* 11.0* 9.0 18.0 36.0 54.0 Mbit] ESS CH: 6, PRIVACY

10:27:07.938361 74362602us tsft 6.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:07.953755 74375625us tsft 6.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:07.958151 74382377us 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

10:27:07.973090 74394982us tsft 6.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:08.003137 74424986us tsft 6.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:08.007497 74432483us tsft 6.0 Mb/s 2437 MHz 11g -59dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:08.033133 74454986us tsft 6.0 Mb/s 2437 MHz 11g -61dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:08.037498 74462485us tsft 6.0 Mb/s 2437 MHz 11g -59dBm signal -95dBm noise
Clear-To-Send RA:e0:3e:44:08:34:e1 (oui Unknown)

10:27:08.063204 74486734us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise
Data IV:d43 Pad 20 KeyID 1
```

```
10:27:08.063206 74486734us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise
Data IV:d44 Pad 20 KeyID 1

10:27:08.063211 74486734us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise
Data IV:d45 Pad 20 KeyID 1

10:27:08.063211 74486734us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise
Data IV:d46 Pad 20 KeyID 1
```

When you run the above command, you are likely to be flooded with management (mgmt) traffic from local stations. To continue we need to think back to the WiFi session establishment process we discussed in previous articles.

In the [previous article](#) we talked about how a station joins a network, in summary it:

- probes for the network (probe request, response)
- authenticates (auth request, response)
- associates (assoc request, response)

Each of these stages is two different mgmt frames: a request and a response. But these are probably not even the frames you are being flooded with!

To speed up the network joining process and to give clients a list of networks to join, all access points 'beacon' their existence constantly. In a very busy environment beacons might use up a lot of the available airtime, that long list of available WiFi networks isn't really a good thing.

Beacon frames carry a lot of information about the network and its capabilities, we can filter to only see beacon frames in the `tcpdump` capture like so:

```
10:27:07.724396 74143272us tsft 1.0 Mb/s 2437 MHz 11g -32dBm signal -95dBm
noise Beacon (o2-enc) [1.0* 2.0* 5.5* 11.0* 9.0 18.0 36.0 54.0 Mbit] ESS CH:
6, PRIVACY
```

There is a lot more information in a beacon than `tcpdump` will show you on its own, to get more of the data carried by the beacon we need to switch to using `wireshark` or in this case its command line interface `tshark`.

First we can pull out a pcap using `tcpdump`:

```
$ sudo tcpdump -i wlan0 -y IEEE802_11_RADIO -w testcaputre.pcap -c 1000
```

We can feed our capture to `tshark` and ask it to show the first beacon packet for the test network we saw before:

```
$ tshark -V -r testcaputre.pcap -a "packets: 1" 'wlan.ssid == "o2-enc"'
```

This outputs nearly 900 lines of packet description for the 393 byte beacon frame, here is a small section showing the beacon frame and some radio information.

```
802.11 radio information
    PHY type: 802.11b (HR/DSSS) (4)
    Short preamble: False
    Data rate: 11.0 Mb/s
    Channel: 6
    Frequency: 2437MHz
    Signal strength (dBm): -66 dBm
    Noise level (dBm): -95 dBm
    Signal/noise ratio (dB): 29 dB
    TSF timestamp: 1853402918
    [Duration: 461µs]
        [Preamble: 192µs]
        [IFS: 15572µs]
        [Start: 1853402457µs]
        [End: 1853402918µs]
IEEE 802.11 Beacon frame, Flags: .....
```

Type/Subtype: Beacon frame (0x0008)

Frame Control Field: 0x8000

.... ..00 = Version: 0

.... 00.. = Type: Management frame (0)

1000 = Subtype: 8

Flags: 0x00

.... ..00 = DS status: Not leaving DS or network is operating
in AD-HOC mode (To DS: 0 From DS: 0) (0x0)

.... .0.. = More Fragments: This is the last fragment

.... 0... = Retry: Frame is not being retransmitted

...0 = PWR MGT: STA will stay up

..0. = More Data: No data buffered

.0.. = Protected flag: Data is not protected

0... = +HTC/Order flag: Not strictly ordered

.000 0000 0000 0000 = Duration: 0 microseconds

Receiver address: Broadcast (ff:ff:ff:ff:ff:ff)

.... ..1. = LG bit: Locally administered address
(this is NOT the factory default)

.... 1 = IG bit: Group address
(multicast/broadcast)

Destination address: Broadcast (ff:ff:ff:ff:ff:ff)

.... ..1. = LG bit: Locally administered address
(this is NOT the factory default)

.... 1 = IG bit: Group address
(multicast/broadcast)

Transmitter address: GLTechnologi_58:ed:32 (94:83:c4:58:ed:32)

.... ..0. = LG bit: Globally unique address
(factory default)

.... 0 = IG bit: Individual address (unicast)

Source address: GLTechnologi_58:ed:32 (94:83:c4:58:ed:32)

.... ..0. = LG bit: Globally unique address
(factory default)

.... 0 = IG bit: Individual address (unicast)

BSS Id: GLTechnologi_58:ed:32 (94:83:c4:58:ed:32)

.... ..0. = LG bit: Globally unique address

```
(factory default)
.... 0 .... = IG bit: Individual address (unicast)
.... 0000 = Fragment number: 0
1010 0010 1011 .... = Sequence number: 2603
[WLAN Flags: .....]
```

System Logs

The third source of information for debugging WiFi issues comes via system tools and logs. Most of the time the system aims to be relatively quiet. In the messages it prints, we normally only get those printed for exceptional conditions such as errors. Developers like to bake in the option of live system debugging and we can increase debugging levels and fill up the kernel message buffer with lots of juicy facts about the WiFi stack.

wldebug

```
/usr/sbin/wldebug
```

FreeBSD ships the `wldebug(8)` tool which gives a friendlier interface to configuring the verbosity of prints that come from the `net80211` stack.

```
# wldebug -i wlan1 scan+auth+assoc
```

Messages are printed directly to the kernel buffer which can be annoying when you want to track events as they come. We can display these newly printed messages with `dmesg`.

```
$ dmesg
```

```
wlan0: ieee80211_scanreq: vap 0xffffffff80139506000 iv_state 0x1 (SCAN) flags 0x20052  
duration 0x7fffffff mindwell 0 maxdwell 0 nssid 1
```

```
wlan0: ieee80211_check_scan: active scan, append, nojoin, once
```

```
wlan0: ieee80211_swscan_start_scan_locked: active scan, duration 2147483647  
mindwell 0 maxdwell 0, desired mode auto, append, nojoin, once
```

```
wlan0: scan set
```

```
1g, 6g, 11g, 7g, 2g, 3g, 4g, 5g, 8g, 9g, 10g
```

```
wlan0: dwell min 20ms max 200ms
```

```
wlan0: scan_curchan_task: loop start; scandone=0, scanstop=0, ss_iflags=0x2,  
ss_next=0, ss_last=11
```

```
wlan0: scan_curchan_task: chan 6n -> 1g [active, dwell min 20ms max 200ms]
```

```
wlan0: scan_curchan: calling; maxdwell=200
```

```
wlan0: scan_curchan_task: waiting
```

```
[22:e8:29:c7:48:1a] new probe_resp on chan 1 (bss chan 1) "Guest-Wifi" rssi 15
```

```
[22:e8:29:c7:48:1a] caps 0x1421 bintval 100 erp 0x100 country [GB 1-13,20]
```

```
wlan0: ieee80211_swscan_add_scan: chan 1g min dwell met (2167162969 >  
18446744071581747289)
```

```
wlan0: scan_mindwell: called
```

```
wlan0: scan_curchan_task: loop start; scandone=0, scanstop=0, ss_iflags=0x21,  
ss_next=1, ss_last=11
```

wlan0: scan_curchan_task: chan 1g -> 6g [active, dwell min 20ms max 200ms]

wlan0: scan_curchan: calling; maxdwell=200

wlan0: scan_curchan_task: waiting

[c8:e3:06:5d:fd:45] new probe_resp on chan 6 (bss chan 6)

0x48c3a47474652c2048c3a47474652c204661687261646b65747465 rssi 29

[c8:e3:06:5d:fd:45] caps 0x1431 bintval 100 erp 0x100 country [GBI 1-13,20]

[c8:e3:06:5d:fd:45] new probe_resp on chan 6 (bss chan 6)

0x48c3a47474652c2048c3a47474652c204661687261646b65747465 rssi 29

[c8:e3:06:5d:fd:45] caps 0x1431 bintval 100 erp 0x100 country [GBI 1-13,20]

[c8:e3:06:5d:fd:45] new probe_resp on chan 6 (bss chan 6)

0x48c3a47474652c2048c3a47474652c204661687261646b65747465 rssi 29

[c8:e3:06:5d:fd:45] caps 0x1431 bintval 100 erp 0x100 country [GBI 1-13,20]

wlan0: ieee80211_swscan_add_scan: chan 6g min dwell met (2167162992 >
18446744071581747312)

wlan0: scan_mindwell: called

[c8:e3:06:5d:fd:45] new probe_resp on chan 6 (bss chan 6)

0x48c3a47474652c2048c3a47474652c204661687261646b65747465

wlan0: scan_curchan_task: loop start; scandone=0, scanstop=0, ss_iflags=0x21,
ss_next=2, ss_last=11 rssi 29

wlan0: scan_curchan_task: chan 6g -> 11g [active, dwell min 20ms max 200ms]

[c8:e3:06:5d:fd:45] caps 0x1431 bintval 100 erp 0x100 country [GBI 1-13,20]

```
wlan0: scan_curchan: calling; maxdwell=200

wlan0: scan_curchan_task: waiting

wlan0: scan_curchan_task: loop start; scandone=0, scanstop=0, ss_iflags=0x20,
ss_next=3, ss_last=11

wlan0: scan_curchan_task: chan 11g -> 7g [active, dwell min 20ms max 200ms]

wlan0: scan_curchan: calling; maxdwell=200

wlan0: scan_curchan_task: waiting
```

These messages are controlled by a mask on a per device `sysctl`. We could set this manually, but `wldebug` gives us a nice interface for doing so. Once configured we get additional log messages on the system log, which we can recover using the `dmesg` command.

Each message which is printed is by the `IEEE80211_DPRINTF` kernel macro.

The final messages in our example come from `scan_curchan` and in the source they look like this:

```
IEEE80211_DPRINTF(vap, IEEE80211_MSG_SCAN,
    "%s: calling; maxdwell=%lu\n",
    __func__,
    maxdwell);

IEEE80211_DPRINTF(ss->ss_vap, IEEE80211_MSG_SCAN,
    "%s: loop start; scandone=%d, scanstop=%d, ss_iflags=0x%x, ss_next=%u,
    ss_last=%u\n",
    __func__,
    scandone,
```

```
scanstop,  
(uint32_t) ss_priv->ss_iflags,  
(uint32_t) ss->ss_next,  
(uint32_t) ss->ss_last);
```

wldebug enables you to take advantage of the debugging statements built into the kernel without having to do a rebuild or be overloaded with output messages all the time.

wlanstat

Collects stats using the SIOCG80211STATS ioctl. Inside net80211 these are stored in the vap iv_stats member which is an instance of ieee80211_stats a ~150 member struct tracking what has happened in the wlan stack.

Thankfully wlanstat is smart about what it prints and it will only display statistics with relevant information:

```
$ wlanstat  
1821          rx from wrong bssid  
2            rx discard mgt frames  
2127          rx beacon frames  
17629         rx element unknown  
53           rx frame chan mismatch  
5            active scans started  
18           ccmp crypto done in s/w  
2221         rx management frames  
2            rx action frames  
59           A-MSDU frames received  
3            A-MPDU frames discarded for out of range seqno  
5034         total data frames received  
5016         unicast data frames received  
18           multicast data frames received
```

6044	total data frames transmit
6044	unicast data frames sent
58.5M	current transmit rate
31.5	current rssi
-95	current noise floor (dBm)
-63.5	current signal (dBm)

From `wlanstat` we get some historic and current information. The current information at the end of the output shows us the signal strength, noise floor and current transmit rates. These will vary as we move and based on how much traffic we have sent recently.

The historic information tells us about the types of frames we have seen. One useful stat for debugging is the number of A-MSDU frames we have received. This number being low can be a great indicator as to why aren't able to achieve higher throughput rates.

ifconfig

`ifconfig` can tell you a lot of information about how the wlan interface is configured.

```
$ ifconfig -v wlan0
wlan0: 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) bssid 00:00:00:00:00:00
    regdomain FCC country US anywhere -ecm authmode OPEN -wps -tsn
    privacy OFF deftxkey UNDEF
    powersavemode OFF powersavesleep 100 txpower 30 txpowmax 50.0 -dotd
    rtsthreshold 2346 fragthreshold 2346 bmiss 7
    11b      ucast NONE      mgmt   1 Mb/s mcast  1 Mb/s maxretry 6
    11g      ucast NONE      mgmt   1 Mb/s mcast  1 Mb/s maxretry 6
```

```

llng      ucast NONE      mgmt  1 Mb/s mcast  1 Mb/s maxretry 6
scanvalid 60 -bgscan bgscanintvl 300 bgscanidle 250
roam:11b      rssi      7dBm rate  1 Mb/s
roam:11g      rssi      7dBm rate  5 Mb/s
roam:11ng     rssi      7dBm MCS   1
-pureg protmode CTS ht20 htcompat ampdu ampdulimit 64k
ampdudensity 16 amsdu shortgi htprotmode RTSCTS -puren -smgs -rifs
-stbc -ldpc -uapsd -vht wme -burst -dwds roaming AUTO bintval 0
AC_BE cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm ack
      cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm
AC_BK cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm ack
      cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm
AC_VI cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm ack
      cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm
AC_VO cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm ack
      cwmn  0 cwmn  0 aifs  0 txopLimit  0 -acm
parent interface: rtwn0
media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
status: no carrier
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
drivername: wlan0

```

The verbose `ifconfig` output tells us a lot of practical information about how the wlan interface is configured and the network and channel it is on (if it is on a network, in this example the ssid is blank as we aren't connected to anything).

The `ifconfig` output tells us parameters we can tune such as the tx power, country, and regdomain.

It also contains parameters we pick up from the network we are on.

The `AC_BE`, `AC_BK`, `AC_VI`, and `AC_VO` describe access class parameters advertised by the network in management frames.

Applying tools

There is a lot of information available from the FreeBSD WiFi stack to help you debug issues, but I think the reality for most people is that you will quickly be overwhelmed with technical details. In most cases, you will get more immediate value starting with higher level tools and working down. If you are having trouble connecting to a network, then running `wpa_cli` and taking note of the messages it reports will leave you in the best standing. If the underlying device is configured correctly and the Access Point is in range, then your problems are most likely to be addressed with advice from `wpa_supplicant`.

If you suspect that the `net80211` layer or device driver is the problem, then your next course of action will be the two-pronged approach of verifying the traffic on the air with packet captures and validating the behaviour of `net80211`. This takes more understanding of how WiFi works, but it can be a good place to get clues about what is wrong.

It is easy to jump at shadows and think that a field indicates an error that isn't there. My main piece of advice to take from this article and the series is how amenable to interrogation the FreeBSD WiFi stack is. If you want to debug a problem, then reaching out to the mailing lists is a great step. You might be asked to present debugging information such as some of the things discussed here.

This shouldn't be scary. It is quite straightforward to enable most of the available logging mechanisms and all of them will default to off again if you reboot.

The speeds and performance we can get with WiFi is incredible, it has grown from a 1Mbit/s link up to more than a gigabit and most of the time we get very consistent results. Our WiFi stack is always improving and with more users and testing it will get better.

Tom Jones is a FreeBSD committer interested in keeping the network stack fast.

Copyright © 2026 Held by Owner/Author. Publication Rights Licensed to The FreeBSD Journal. This work is licensed under a [**Creative Commons Attribution International 4.0 License**](#).

3980 Broadway Street
STE #103-107
Boulder CO 80304
1-720-207-5142
info@freebsd.foundation.org

