Porting FreeBSD Wi-Fi Stack to NetBSD

James Browning, Philip A. Nelson

1 Abstract

As the Wi-Fi family of protocols has continued to grow over the past decades, the NetBSD operating system has been somewhat lagging behind. A little more than a decade ago, FreeBSD introduced changes to make the Wi-Fi layer more practical, and FreeBSD has continued to update this layer to support newer Wi-Fi protocols. Because of the similarities in both FreeBSD's and NetBSD's source code, a project was undertaken by NetBSD to port this code from FreeBSD. The source code has been added to a branch of NetBSD, and it is in a buildable state. Several Wi-Fi drivers have been updated to use this new Wi-Fi API. Although project has made significant progress and continues to be worked on, it is not yet in a working state.

2 Background

2.1 Goal

NetBSD and FreeBSD are open source Unix-like operating systems derived from BSD[1][2]. Both operating systems are based on 4.4BSD, meaning they initially shared very similar code bases[3][1]. Over the decades the two projects have grown independently and their code bases have drifted further apart, especially in Wi-Fi support.

The goal of this project is to replace NetBSD's current Wi-Fi stack with the Wi-Fi stack of FreeBSD[4]. The Wi-Fi layer of these operating systems are both implemented in kernel space, so the changes will primarily revolve around adapting the FreeBSD Wi-Fi code to use NetBSD's kernel APIs. This is followed by porting NetBSD's wireless adapter drivers to use the new Wi-Fi API. One of the goals of this project is to make the Wi-Fi code into a shared code-base between FreeBSD and NetBSD, where system dependent parts of the code can be included using `#ifdef __NetBSD__` or `#ifdef __FreeBSD__` respectively, as well as kept in the `ieee80211_netbsd.c` and `ieee80211_freebsd.c` source files. This means that new additions to Wi-Fi can be worked on collaboratively by both communities.

2.2 Motivation

There are significant design differences between NetBSD and FreeBSD's Wi-Fi stack. The details of FreeBSD's Wi-Fi design are detailed in Sam Leffler's presentation "FreeBSD Wireless Networking"[5]. Mainly, FreeBSD's stack adds an extra layer of abstraction known as a "virtual access point" or vap. Multiple vaps can be cloned from a single Wi-Fi adapter, allowing for the network administrator to create multiple connections associated with a single Wi-Fi adapter. In contrast, NetBSD's old model only allows for a single interface to be created per Wi-Fi device. The vap design enables more flexibility. For example, it allows the administrator to set up multiple BSSs with a single adapter, each with their own security policy, or it can allow a radio to act as both a station and host simultaneously. The potential to create and combine multiple vaps allows for the creation of useful wireless network architectures.

For example a station vap and access point vap can be combined to create a wireless repeater, which will boost the signal of another access point.

FreeBSD's Wi-Fi layer also supports operating modes that were not supported in NetBSD's code. These modes include mesh (WMN) and distribution (WDS). WMN mode allows for multiple nodes (Wi-Fi adapters) to be chained together to extend network range. If one node goes down, the network can still function as long as there is still a path to the gateway. WDS mode is similar to WMN in that it allows for a wireless network to be extended by connecting access points, however it is based on a simpler model and is less dynamic/flexible than WMN. Both of these modes are very useful for creating robust wireless topologies, and it is a big step forward for NetBSD to support them.

In addition to the vap functionality and new operating modes, FreeBSD's Wi-Fi stack also supports more 802.11 protocols than NetBSD's stack. Namely, 802.11n[6]. The FreeBSD community is in the process of adding 802.11ac support[7], and will hopefully be adding 802.11ax support soon after. With a shared Wi-Fi code-base, both communities will be able to work collaboratively in adding support for 802.11ac and 802.11ax. Supporting these modes is significant because it will allow NetBSD to add driver support for the latest Wi-Fi hardware and achieve significantly higher wireless network speeds.

2.3 User Facing Differences

Since the FreeBSD 802.11 stack is significantly different than the NetBSD 802.11 stack, the way the system user creates and maintains Wi-Fi interfaces is different as well.
In NetBSD, configuring Wi-Fi is a bit simpler. When connecting an adapter, it will appear in the output of `ifconfig` (since there can only be a single interface per Wi-Fi adapter). `wpa_supplicant` is used to configure the adapter as a station, or `hostapd` can be used to configure the adapter as an access point.`ifconfig` is used to activate or deactivate the interface.

In FreeBSD, there will not initially be any interfaces for the connected Wi-Fi adapter when `ifconfig` is run. This is because the adapters themselves aren't considered interfaces, it's the vaps that are cloned from the adapters that act as the network interfaces. The list of connected Wi-Fi adapters can be obtained through "`sysctl net.wlan.devices`". With the name of the device, an interface can be cloned:

```
ifconfig create <interface_name> \
wlandev <adapter_name> \
wlanmode <opearting_mode>
```

Then the interface can be configured to use `wpa_supplicant` or `hostapd` through the `rc` startup daemon.

3 Initial Porting Effort

3.1 Initial Work

In the summer of 2018, the NetBSD foundation funded a project that started the work to port the FreeBSD stack to NetBSD. This was a single person project for that summer. The FreeBSD `ieee80211` code replaced the existing NetBSD code and the urtwn usb driver was used to facilitate the conversion. By the end of the summer, there was enough functionality in the conversion to allow a ssh session over an unencrypted connection. Progress on this project stalled at this point and was restarted in March of 2020. A number of other people have now been contributing toward this work.

3.2 Core Differences

Many changes were necessary to make the FreeBSD Wi-Fi code compatible with NetBSD's kernel. The core of these changes can be found in the `ieee80211_freebsd.c` and

ieee80211_netbsd.c of the respective operating systems. These files contain functions which are highly dependent on the kernel APIs. These functions include various functionality such as sysctl management, kernel privilege checking, task management, interface attachment and more.

NetBSD's workqueue framework does not have the same functionality as FreeBSD's taskqueue framework[8][9], so it had to be somewhat emulated in NetBSD for the new Wi-Fi code. A function was defined called ieee80211_runwork as well as a new struct called task. runwork takes in a task struct and then invokes whatever function is pointed to within that task. This allows for work to be dispatched to individual functions from a driver's global work queue (ic_tq). We also included the timeout functionality of taskqueue timeouts using NetBSD's callouts framework. This allows for us to schedule tasks to occur at specific intervals.

When management packets are passed up the network stack, this will potentially involve memory allocation or locking operations. In NetBSD these types of operations can only be done in a thread context (not in an interrupt). For this reason, a new NetBSD unique function was introduced called ieee80211_rx_enqueue. The driver should pass the received packet and the ieee80211com struct to this functio n, the function will then identify if the packet as a management packet or data packet. If it finds the packet is a management packet, it is placed in a queue to be processed later in a thread context, otherwise it simply passed up the stack.

```
void
ieee80211_rx_enqueue(struct ieee80211com
    *ic, struct mbuf *m, int rssi)
{
  struct ieee80211_frame *wh;
  struct ieee80211_node *ni;

  wh = mtod(m, struct ieee80211_frame *)
    ;
  ni = ieee80211_find_rxnode(ic, (struct
     ieee80211_frame_min *)wh);

  if (IEEE80211_IS_DATA(wh)) {
    /*
     * Just pass it up, it will be
    enqueued on the VAPs ifqueue
     */
    if (ni != NULL) {
      if (ni->ni_vap == NULL) {
        ieee80211_free_node(ni);
        return;
      }
      ieee80211_input(ni, m, rssi, 0);
      ieee80211_free_node(ni);
    } else {
      ieee80211_input_all(ic, m, rssi,
    0);
    }
  } else {
    /*
     * We might need to take "heavy"
    locks during
     * further processing (like the IC
    lock), and can
     * not do this from softint or
    callout context.
     */
    M_SETCTX(m, ic);
    m_append(m, sizeof(rssi), &rssi);
    IF_ENQUEUE(&ieee80211_rx_mgmt, m);
    taskqueue_enqueue(ic->ic_tq, &
    ieee80211_mgmt_input);
  }
}
```

Listing 1: ieee80211_rx_enqueue created by Martin Husemann

4 Converting a Driver

4.1 Intro

Since the new Wi-Fi layer is now in a buildable state for NetBSD, most of the remaining work is to convert the existing NetBSD drivers to test

the new layer. This process has been thoroughly documented by Martin Husemann[10], and this section is intended to elaborate on steps described in that guide. Understanding this process is very valuable for understanding the design differences between FreeBSD's Wi-Fi stack and NetBSD's stack.

4.2 Wi-Fi Data Structures & APIs

First, let's establish some important data structures in the Wi-Fi layer. The core data structure for any Wi-Fi driver is the struct `ieee80211com`. This contains essential information about the Wi-Fi adapter, such as the device name, synchronization locks, a vap list, a list of valid device channels, various call back functions, and much more. Virtually every function in the Wi-Fi API will take in the `iee80211com` struct. The `ieee80211vap` struct represents a specific vap that's been created for the device. `ieee80211_node` represents external Wi-Fi peers. What "peers" means will depend on the operating mode of the vap, for example, an AP will consider these nodes to be stations, whereas a station will register nodes of APs it has connected to.

The struct `ifnet` contains all the data for any network interface. It is part of NetBSD's network layer, not the Wi-Fi layer. Since network interfaces are now tied to vaps instead of the device itself, we must remove `ifnet` references from the driver attachment code. Many of the callback functions which previously took in the `ieee80211com` struct, will now take in a `ieee80211vap` instead. The vap struct contains a reference to the associated `ifnet` as well as a reference to the `ieee80211com`.

4.3 Attachment

The attachment phase is the phase in which the kernel has recognized a device which the driver supports, so now the driver must preform initialization of the device, and attach to the 802.11 layer. Before attaching to the 802.11 layer, the device must report which Wi-Fi capabilities it supports. This is reported by filling in the `ic_caps` field of the `ieee80211com` struct. This paper will not go over an exhaustive list of all the macros that are supported here, but some of the important capabilities the developer needs to consider are: operating modes (station, adhoc, ap, etc), 802.3 encapsulation, TDMA mode, A-MPDU/A-MSDU transmission, and encryption methods, to name a few.

During the attachment phase the device must setup the various callback functions required in the `ieee80211com` struct. Some of the callbacks are optional, meaning the 802.11 layer will fill-in a default callback. The callbacks that are required to be filled in are the ones which handle vap creation, vap deletion, channel setting, and starting/stopping of scanning. In order for the device to send packets out, the driver will need to fill in the transmit callback, as the default transmit just prints a message and drops the packet. There are many other callbacks which are technically optional, but probably necessary for the device to have full functionality. These callbacks include, reporting radio capabilities, transmitting management frames, transmitting raw packets, starting/stopping of A-MPDU reception, and more. While some of these callbacks are already present in NetBSD's existing Wi-Fi stack, the rest of these are new to NetBSD. The scope of this paper is not to describe every callback in detail, however it will cover the most important ones to get the device running.

4.4 Channels

The way the driver reports supported rates has changed. Previously the driver would populate the `ic_channels` field with the frequencies and appropriate flags for each channel supported by the device. Now it is done by defining a callback function to the `ic_getradiocaps` field. This function takes in two pointers,

nchans which points to an int which will represent the number of channels supported, the other is chans which points to an array of ieee80211_chanlist structs. The role of this function is to populate nchans (the number of channels) and channels (the list of supported channels). During attachment, this function needs to be used to populate ic_channels (passed in as chans) and ic_nchans (passed in as nchans) before calling ieee80211_ifattach.

```
void (*ic_getradiocaps)
(struct ieee80211com *ic, int maxchans,
    int *nchans, struct
    ieee80211_channel chans[]);
```

Listing 2: ic_getradiocaps callback

This can be accomplished by having the driver manually fill in the channel structs or by passing a numerical channel list to the Wi-Fi APIs and having them fill in the channel structs on behalf of the driver. The latter method is recommended and it will be suitable for the majority of cases. To accomplish this, the function must define an array that is IEEE80211_MODE_BYTES bytes long and populate that array with the supported protocols. This is done using the bit fields defined in ieee80211_ratectl.h. Then the array is passed to one of the ieee80211_add_channel_list* functions (there are several variations), along with the two pointers mentioned previously. The ieee80211_add_channel_list* function will look at which modes are supported based on the provided array, and then populate the values of the provided pointers. The driver must call this function during attachment, before calling ieee80211_ifattach, passing in ic_channels as the array to be populated.

There are a few variations of the ieee80211_add_channel_list* functions depending on what the device supports. ieee80211_add_channel_list_2ghz will populate the channel list using the appropriate flags for 2ghz functionality, and ieee80211_add_channel_list_5ghz will do the same for 5ghz channels. If the device supports both 2ghz and 5ghz, then ic_getradiocaps will need to call both of these functions with the appropriate numerical channel list for input. ieee80211_add_channel_list_default_2ghz can be used to support just the default 2ghz channels.

4.5 VAPs

As mentioned previously there is no longer have a single network interface for the Wi-Fi device, instead there is a network interface for each vap that is clone from the device. This means we must define two new callback functions: *_vap_create and *_vap_delete (* is the name of the driver). These will get assigned to ic_vap_create and ic_vap_delete respectively during attachment.

ic_vap_create takes in the ieee80211com struct, as well as other information needed to initialize the state of the vap, including the interface name, the operation mode, the associated bssid (if the vap was initialized with one), and the mac address. It will return a pointer to a new struct ieee80211vap which will contain the ifnet struct. The vap creation function needs to call ieee80211_vap_setup() which will initialize the ifnet struct inside of the ieee80211vap struct, as well as other fields. At this point the driver may override any callbacks for the vap or for the network interface, although in most cases this isn't necessary. ieee80211_vap_attach is called to attach the vap to the Wi-Fi stack.Now in the driver attachment function, *_vap_create() is registered as a callback in the struct ieee80211com. Additionally, certain fields such as ic_state from NetBSD's ieee80211com are not in FreeBSD's because they are per-vap and now exist in the ieee80211vap structs, so initialization of these fields will also need to be removed from the attachment function.

Something that must also be considered for the

vap creation function is the number of vaps supported, since this function is where the driver will enforce the vap limit. This will depend on the functionality of the hardware. For example, in FreeBSD's rtwn driver the vap creation function will return NULL if there are already 2 vaps since that is the number of MAC addresses supported by the hardware. The ath driver will only allow for a single station, but it will allow for multiple host access point vaps because the hardware contains a programmable bssid mask to accept varying numbers of destination MAC addresses.

`ic_vap_delete` is much simpler than the vap creation function, it should simply call `ieee80211_vap_detach` and then free the memory of the vap struct.

4.6 Receive and Transmit

In order to transmit packets or pass packets up the network stack (both of which will be needed for the device to actually be usable) the driver will need to assign some callbacks before attaching to the 802.11 stack, namely `ic_transmit` and `ic_raw_xmit`. This differs from the current method used by NetBSD, which is to define a call back to the network interface, `ic_ifp`. The new stack abstracts the network interface away from the driver, handling the interface callbacks for the driver, based on what is assigned to `ic_transmit` and `ic_raw_xmit`.

`ic_transmit` is called when the driver must first encapsulate the packet in an ethernet frame. `ic_raw_xmit` is called to transmit a raw packet (such as those injected through bpf). Typically whatever function the driver assigns to `ic_transmit` wraps a call to the function assigned to `ic_raw_xmit`.

The receive portion of the driver will look largely the same, since that path is usually is initiated by an interrupt generated by the device. The main change required here is the call to push the packet up the stack. NetBSD previously used a single call to `ieee80211_input` to pass all packets up the stack. Under the new design, the driver will need to check first if the packet is a destined for a specific vap, or if its being broadcast to all currently running vaps. In the first case it should pass the packet to `ieee80211_input`, and in the second it will pass the packet to `ieee80211_input_all`. The latter is a wrapper for the former, and will pass the packet to all of the vaps associated with the device. `ieee80211_rx_enqueue` will handle calling the appropriate input function, as well as enqueuing management packets (see the "Core Differences" section for more details on `ieee80211_rx_enqueue`).

4.7 Radiotap

The way radiotap is handled has changed as well. In NetBSD the driver was in charge of allocating a bpf listener and passing radiotap headers to the bpf layer. With the FreeBSD code, the bpf layer is abstracted away from the driver. Now the driver simply defines the radiotap header pointers `ic->ic_th` and `ic->ic_rh` which stand for "transmit header" and "receive header" respectively. When a packet is received, the driver will fill out the radiotap header and pass it to `ieee80211_radiotap_tx` which will pass the radiotap information to bpf. For receiving a packet, the radiotap header is simply filled out, and the 802.11 stack will hand it to bpf when the packet is passed up the stack. During attachment the driver must call `ieee80211_radiotap_attach` to hand the radiotap headers pointers to the 802.11 layer, this function will assign `ic->ic_th` and `ic->ic_rh` on behalf of the driver. It is suggested to implement radiotap support (assuming it is supported by the hardware) before testing the driver, as packet capturing is a very useful step in debugging a network driver.

5 Current State & Work to be Done

The new layer has been integrated in the source tree and is buildable, as are many of the Wi-Fi drivers. At this point, the drivers which have been converted (meaning they are compilable, but not necessarily fully working) include run(4), ipw(4), iwm(4), iwn(4), rtwn(4), and urtwn(4)[11]. The conversion for athn(4) is in progress. Currently, most of these devices result in either a dead lock or kernel panic when attempting to associate with an access point. Most operating modes have yet to be fully enabled, and attempts to use them will generally result in a kernel panic. When attempting to create an access point using either run(4) or athn(4), a kernel panic will occur briefly after activating the interface. Overall this project has a long ways to go before being merged with NetBSD's trunk, but progress is continuing steadily.

In terms of work to be done, the majority of the wireless adapter drivers still need to be ported to adapt to the new Wi-Fi stack. The process of porting a driver is mostly mechanical, and requires little engineering. However, it can be a time consuming process. Converting more drivers will also allow for tests on a wider range of hardware with the new layer, which will allow for testing of various hardware dependent features such as 5GHz frequencies or 802.11n.

Besides converting new drivers, the previously converted drivers also need to be completed. This will involve debugging and identifying the source of panics and dead locks for each device, preferably starting with station mode. After station mode is working, then other modes can be debugged such as AP mode or mesh mode.

Because NetBSD is an open source project, anyone may help contribute. If you are interested in contributing I would suggest reading the sources cited in this paper to get more familiar with the details of the project. From there you can reach out the NetBSD mailing list expressing interest in this port. This project is highly parallelizable, so more developers are always welcome, the more people working on this project the sooner it can be merged with trunk.

References

[1] About netbsd. [Online]. Available: https://www.netbsd.org/about/

[2] D. G. Baio. (2021, June) About freebsd. [Online]. Available: https://www.freebsd.org/about/

[3] M. J. Karels, J. Quarterman, M. K. McKusick, and K. Bostic, *The Design and Implementation of the 4.4BSD Operating System*, 1996.

[4] M. Husemann, "Wifi renewal restarted," April 2020. [Online]. Available: https://blog.netbsd.org/tnf/entry/wifi_renewal_restarted

[5] S. Leffler. (2005) Freebsd wireless networking. [Online]. Available: https://www.bsdcan.org/2005/papers/FreeBSDWirelessNetwokringSupport.pdf

[6] "Wireless networking," 2022. [Online]. Available: https://wiki.freebsd.org/WiFi

[7] "802.11ac (wi-fi 5) todo," 2021. [Online]. Available: https://wiki.freebsd.org/WiFi/80211ac

[8] D. Rabson, *FreeBSD Kernel Developer's Manual - TASKQUEUE(9)*, September 2021. [Online]. Available:

https://www.freebsd.org/cgi/man.cgi?query=taskqueueapropos=0sektion=0manpath=FreeBSD+13.1-RELEASE

[9] *NetBSD Kernel Developer's Manual - WORKQUEUE(9)*, December 2017. [Online]. Available: https://man.netbsd.org/NetBSD-9.2/workqueue.9

[10] M. Husemann. (2022, September) Converting drivers to the new wifi stack. [Online]. Available: https://wiki.netbsd.org/Converting_drivers_to_the_new_wifi_stack

[11] ——. (2022) Driver state matrix. [Online]. Available: https://wiki.netbsd.org/Driver_state_matrix