SEE TEXT ONLY

TRACING `ifconfig` Commands from Userspace to Device Driver

I am currently working on expanding FreeBSD's `rtwn(4)` wireless device driver. I have the basics down, such as initialization, powering on and off, loading the firmware, etc., and am now trying to fill in specific `ifconfig(8)` methods. This requires an in-depth knowledge of how `ifconfig(8)` commands are ultimately delivered to the driver. I could not find concise documentation that outlines each stage of the process, so I wrote one!

BY FARHAN KHAN

This article is specific to FreeBSD 12.0-CURRENT, but it should apply to any future version and other operating systems that utilize `net80211(4)`, such as OpenBSD, NetBSD, DragonFlyBSD, and illumos. I hope it serves to help the FreeBSD community continue to develop WiFi and other device drivers. This is not an exhaustive guide, but it should provide you with the basic order of operations. In this example, I walk through changing the channel on your WiFi card and placing it in monitor mode as follows:

```
# ifconfig wlan0 channel 6 wlanmode monitor
```

High-level Summary

FreeBSD's `ifconfig(8)` utilizes the `lib80211(3)` userspace library which functions as an API to populate kernel data structures and issue the `ioctl(2)` syscall. The kernel receives the `ioctl(2)` syscall in a new thread, interprets the structure, and routes the command to the appropriate stack. In our case this is `net80211(4)`. The kernel then creates a new queued task and terminates the thread. Later on, a different kernel thread receives the queued task and runs the associated `net80211(4)` handler, which immediately delivers execution to the device driver.

To summarize again:

Userspace: `ifconfig(8)` binary
Userspace: `lib80211(3)` library
Kernel: `net80211(4)` code
Kernel: `taskqueue(9)` to the driver
Kernel: Device Driver

Let's begin!

Userspace: `ifconfig(8)` + `lib80211(3)` library

Early in `ifconfig(8)`, it opens a `SOCK_DGRAM` socket in `/usr/src/sbin/ifconfig/ifconfig.c` as follows:

```
s = socket(AF_LOCAL, SOCK_DGRAM, 0).
```

This socket functions as the interface for userspace to kernel communication. Rather than tracing from the `if-else` maze in `main()`, I grepped for the string "channel", and found it in `ieee80211_cmd[]`defined at the end of `/usr/src/sbin/ifconfig/ifieee80211.c`.

This table enumerates all `ieee80211` `ifconfig(8)` commands. The "channel" command is defined as follows:

```
DEF_CMD_ARG ("channel" set80211channel).
```

Note the second argument. I looked up `DEF_CMD_ARG` and found that it was a pre-processor macro that defines what function is run when the user sends `ifconfig(8)` a command. A quick grep search shows `set80211channel` is defined in `/usr/src/sbin/ifconfig/ifieee80211.c`. The parameters are fairly easy to identify: `val` is the new channel number (1 through 14) and `s` is the socket we opened earlier. This executes `ifconfig(8)`'s `set80211` function whose sole purpose is to cleanly transfer execution into the `lib80211(3)` library.

`lib80211(3)` is an 802.11 wireless network management library to formally communicate with the kernel. It is worth noting that neither OpenBSD nor NetBSD has this library and instead opt to communicate directly to the kernel. As mentioned, `ifconfig(8)`'s `set80211` function calls `lib80211_set80211` located in `/usr/src/lib/lib80211/lib80211_ioctl.c`. The `lib80211_set80211` function populates an `ieee80211reqdata` structure, used for user-to-kernel `ieee80211` communication. In the example below, this is the `ireq` variable, which contains the WiFi interface name and intended channel.

The library then calls the ioctl(2), as follows:

```
ioctl(s, SIOCS80211, &ireq).
```

This runs the syscall to formally enter kernel-space execution. In essence, `ifconfig(8)` is nothing more than a fancy `ioctl(2)` controller. You could write your own interface configuration tool that directly calls the `ioctl(2)` syscall and get the same result. Now on to the kernel!

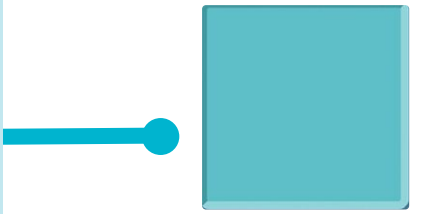
Kernel Command Routing to `net80211(4)`

There are two brief explanations before we proceed: First, at a high level, the BSD kernel operates like an IP router in that it routes execution through the kernel, populating relevant data values along the way until the execution reaches its destination handling functions. The following explanation shows how the kernel will identify the syscall type, determine that it is for an interface card, determine the type of interface card, and finally queue a task for future execution.

Second, the BSD kernel utilizes a common pattern of using template methods that call a series of function pointers. The exact function pointers are conditionally populated, allowing the code to maintain a consistent structure while the exact implementation may differ. It works very well, but can make tracing execution paths difficult if you are just reading the code straight through. When I had trouble, I typically used illumos's OpenGrok (http://src.illumos.org/source/) or dtrace(1).

Let's take a brief detour into dtrace(1). Solaris's dynamic tracing tool is imported to FreeBSD and used to monitor a kernel or process in real time. It is useful in understanding what the operating system is doing and saves you the trouble of using printf(3)-style debugging. I used dtrace(1) in writing this guide to identify what the kernel was executing, function arguments, and the stack trace at any given moment. For example, if I wanted to monitor the ifioctl function, I might run this:

```
# dtrace -n '
> fbt:kernel:ifioctl:entry {
> self->cmd = args[1];
> stack(10);
> }
> fbt:kernel:ifioctl:return {
> printf("ifioctl(cmd=%x) = %x", self->cmd, arg1);
> exit(0);
> } '
```



This dtrace(1) one-line command sets up handlers for `ifioctl`'s entry and return probes. On entry, dtrace(1) records the value of the second argument `cmd`, and displays the last 10 elements of the stack. On return, it displays the function argument and return value. I used variations of this basic command template throughout my research, especially when I was confused in tracing the code or could not identify a function's arguments.

The first non-assembly function is the amd64-specific syscall handler, `amd64_syscall`, that receives a new thread structure and identifies the type as a syscall. In our case, it is for an `ioctl(2)`, so amd64_syscall calls `sys_ioctl` located in `/usr/src/sys/kern/sys_generic.c`.

On FreeBSD, `sys_ioctl` performs input validation and formats the data it receives. It then calls `kern_ioctl`, which determines what type of file descriptor the `ioctl(2)` is working with, the capabilities for the socket, and assigns the function pointer `fo_ioctl` accordingly. (NetBSD and OpenBSD do not have `kern_ioctl`. For them, `sys_ioctl` directly calls `fo_ioctl`.) Our file descriptor corresponds to an interface, so FreeBSD assigns `fo_ioctl` as a function pointer to `ifioctl`, which handles interface-layer `ioctl(2)` calls. This function is located in `/usr/src/sys/net/if.c`.

The function `ifioctl` is responsible for all sorts of interfaces: Ethernet, WiFi, epair(4), etc. `ifioctl` starts with a switch-condition based on the `cmd` argument. This checks if the command can be handled by

net80211(4) without needing to jump into the driver, such as creating a clone interface or updating the MTU. A quick dtrace(2) probe reveals that the cmd argument is SIOCS80211, which fails to meet any switch-conditions, so execution jumps to the bottom. The function continues and calls ifp->if_ioctl, which, in the case of WiFi, is a function pointer to ieee80211_ioctl, located in /usr/src/sys/net80211/ieee80211_ioctl.c.

ieee80211_ioctl contains another switch-case. With cmd set to SIOCS80211, execution matches the associated case and calls ieee80211_ioctl_set80211, located in /usr/src/sys/net80211/ieee80211_ioctl.c.

ieee80211_ioctl_set80211 has yet another switch-case with a few dozen conditions. The ireq->i_type was set to IEEE80211_IOC_CHANNEL by lib80211(3) so it will match the associated case and execute ieee80211_ioctl_setchannel. The gist of this function is to determine if the input channel is valid, or if the kernel needs to set any other values. It concludes by calling setcurchan, which does two things. First, it determines the validity of the channel and if any additional values must be set. Second, it runs ieee80211_runtask, which makes the final thread-level call to taskqueue_enqueue.

The Kernel: Task Execution

taskqueue_enqueue is not an ieee80211(9) function, but it's worth a brief review. In a nutshell, the taskqueue(9) framework allows you to defer code execution into the future. For example, if you want to delay execution for 3 seconds, running the kernel equivalent of sleep(3) would cause the entire CPU core to halt for 3 seconds. This is unacceptable. Instead, taskqueue(9) allows you to specify a function that the kernel will execute at a later time.

In our channel change example, the scheduled function is the net80211(4) function update_channel, located in /usr/src/sys/net80211/ieee80211_proto.c. When taskqueue(9) reaches our enqueued task, it will first initiate the update_channel handler to receive the task and immediately hand over execution to the driver code pointed to by ic_set_channel.

To summarize, up to this point, the kernel has routed the command to the network stack, which routed to the WiFi-specific stack, where it was scheduled as a task for future execution. When taskqueue(9) reaches the task, it immediately jumps to the driver-specific code. At last, we entered the driver!

The Driver

From here on, the code is driver-specific and I will not get into the implementation details, as each device has its own unique channel changing process. I am currently working on rtwn(9), which is located in /usr/src/sys/dev/rtwn. NetBSD and OpenBSD separate USB and PCI drivers, so the same driver is located in /usr/src/sys/dev/usb/if_urtwn.c and /usr/src/sys/dev/pci/if_rtwn.c, respectively.

Operating systems need a standard way to communicate with device drivers. Typically, the driver provides a structure containing a series of function pointers to driver-specific code, and the kernel uses this as an entry point into the driver code. In the case of WiFi, this structure is ieee80211com, located in /usr/src/sys/net80211/ieee80211_var.h. By convention, all BSD-derived systems use the variable name ic to handle ieee80211(9) methods.

In our case, we are changing the channel, so the operating system will call ic->ic_set_channel, which is a pointer to the driver's channel changing function. For rtwn(9), this is rtwn_set_channel, which itself is a function pointer to r92c_set_chan, r92e_set_chan or r12a_set_chan, depending on which specific device you are using.

The specifics of rtwn(9) are outside of the scope of this article, but it is worth discussing how the driver communicates to the hardware. The softc structure is a struct that maintains the device's runtime variables, states, and method implementations. By convention, each driver's softc instance is called sc. You might wonder why you need yet another method function pointer when ieee80211com provides that. This is because ieee80211com's methods point to command handlers, not necessarily to device routines. Device drivers may have their own internal methods that are not part of ieee80211com. Also, the softcstructure can handle minor variations between device versions. rtwn(9)'s softc struct is

called `rtwn_softc` and is located in `/usr/src/sys/dev/rtwn/if_rtwnvar.h`.

How does a driver send data to the driver? `rtwn(9)` uses the `rtwn_write_[1|2|4]` and `rtwn_read_[1|2|4]` methods to actually send or receive a byte, word, or double-word. `rtwn_read_1` is a pointer to the `sc_read_1` method. The driver assigns the `sc_read` class of functions at initialization to either the `rtwn_usb_read_*` and `rtwn_usb_write_*` methods or `rtwn_pci_read_*` and `rtwn_pci_write_*`. The aforementioned classes of functions are abstractions to the PCI and USB buses. In the case of PCI, these function calls will eventually call `bus_space_read_*` and `bus_space_write_*`, which are part of the PCI subsystem. In the case of USB, the driver will call `usbd_do_request_flags`, which is part of the USB subsystem. A well-written driver should abstract these bus-specific layers and provide you with clean read and write methods for various data sizes. As an aside, FreeBSD is long overdue for an SDIO stack, and this is a major impediment for the Raspberry Pi, Chromebooks, and other embedded devices. But I digress…

As an example, the driver uses the following line to enable hardware:

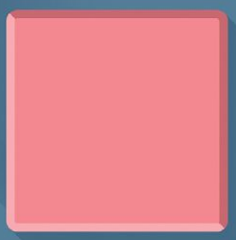
```
interrupts.rtwn_write_4(sc, R92C_HIMR, R92C_INT_ENABLE).
```

This will write the value `R92C_INT_ENABLE` to the `R92C_HIMR` device register.

Conclusion

To summarize this journey, the `ifconfig(8)` opens a socket and passes it to the `lib80211(3)` library. `lib80211(3)` sends a `userspace-to-kernel` command structure to the kernel with an `ioctl(2)` syscall. The syscall triggers the kernel to run a new kernel thread. From here, the kernel determines that the `ioctl(2)` command corresponds to a network card, specifies the type as a WiFi card, and then identifies the exact command type. The `ieee80211(9)` tells `taskqueue` to create a new task to change the WiFi channel, and then terminates. Later on, the `taskqueue(9)` runs the `ieee80211(9)` task handler that transfers execution to the driver. The driver communicates to the hardware using the PCI or USB buses to change the WiFi channel.

In my opinion, FreeBSD is technically superior to Linux, but is still lacking in several critical areas, and among those is hardware support. I hope this article prompts the FreeBSD community to continue to produce high-quality, faster device drivers. ●



FARHAN KHAN is a 2012 graduate from George Mason University in Applied Information Technology Security. He currently works as a Senior Security Engineer for McAfee Security. He considers himself an IPv6 evangelist, avid programmer, and hopes to contribute to the FreeBSD Project.