Getting started with

OpenBSD device driver development

Stefan Sperling <stsp@openbsd.org>

EuroBSDcon 2017

Clarifications (1/2)

Courtesy of Jonathan Gray (jsg@):

- We don't want source code to firmware
- We want to be able to redistribute firmware freely
- Vendor drivers are often poorly written, we need to be able to write our own
- We want documentation on the hardware interfaces
- Source code is not documentation

Clarifications (2/2)

I would like to add:

- Device driver code runs in our kernel
- Firmware code runs on a peripheral device, not in our kernel
- When we say "Binary Blobs" we are talking about closed-source device drivers, not firmware
- Firmware often runs with high privileges; use at your own risk
- If you do not want closed-source firmware, then don't buy and use hardware which needs it
- We are software developers, we do not build hardware
- We will not sign NDAs
- Please talk to hardware vendors and advocate for change; the current situation is very bad for free software

This talk is for ...

- somewhat competent C programmers
- who use OpenBSD
- and would like to use a device which lacks support
- so have an itch to scratch (enhance or write a driver)
- but are not sure where to start
- perhaps are not sure how the development process works
- and will research all further details themselves

The idea for this talk came about during conversations I had at EuroBSDcon 2016. Common question: "How do I add support for my laptop's wifi device?"

Itches I have scratched...

- piixpcib(4): ported some asm from NetBSD
- rtsx(4): Realtek SD host controller driver
- pms(4): support for Elantech touchpads
- rtwn(4): PCIe version of urtwn(4)
- run(4): device support ported from FreeBSD
- ulpt(4): firmware loading for HP LaserJet
- iwm(4): de-facto maintainer (original itch by phessler@)
- iwn(4): 802.11n support; bug fixes
- ral(4): bug fixes
- athn(4): 802.11n support for my access point
- xhci(4): support for isochronous transfers (with mpi@)

I am drawing on my own experience. The information provided here is neither authoritative nor exhaustive. Do your own research!

Process topics

Getting in the right frame of mind

You will make many, many mistakes. Everyone does.
There will be a lot of:

- unknowns
- surprises
- setbacks

You will overcome these with:

- motivation
- endurance
- patience

Can you let a stuck project sit for a week and come back to it?
Or will you abandon your device driver project when it gets stuck?

Pick a reasonable target

You may want your laptop's Broadcom wifi to work now, but it won't happen. Start with something small as your first driver project and work your way up.

- Extend device support of an existing driver
  - Many drivers lack support for newest models of their devices
  - Sometimes just adding a new PCI/USB device ID is enough
- Target a relatively simple device
  - Sensors – see sensordev_attach(8)
  - Touchpads
  - Serial port adaptors
  - Time receivers
  - Real time clock sources
  - Many ARM boards still lack drivers for some components

Research existing device drivers

Find drivers to read in `/usr/src/sys/dev` and with `apropos(1)`.
Many types of drivers exist, such as:

- sensors: asms(4), km(4), sdtemp(4), utrh(4), ...
- gpio: bytgpio(4), macgpio(4), skgpio(4), ...
- buses: pci(4), usb(4), sdmmc(4), sunxi(4), ...
- disk: sd(4), wd(4), rd(4), mpi(4), ...
- network: em(4), vr(4), ix(4), athn(4), iwn(4), iwm(4), ...
- input: kbd(4), pms(4), ws(4), joystick(4), ...
- display: wsfb(4), efifb(4), vga(4), inteldrm(4), ...
- pseudo: tun(4), vlan(4), bio(4), softraid(4), ...
- virtual: virtio(4), pvbus(4), vbus(4), vmx(4), ...

If possible, base your new driver on an existing one.

Stages of driver development

- Make kernel attach your device
  - to an existing driver if possible (and skip most steps below)
  - to your fresh bare skeleton driver
- Register a stub function as interrupt handler
- Device initialization (poking at registers with dark magic)
  - hack Linux driver to print all register reads and writes
  - sift through walls of hex values; implement same init sequence
- Perhaps load some firmware?
- Get your first interrupt (it's alive!) and crash or hang
- Fix a bug
- Crash or hang
- Fix a bug
- Hardware reports some error status
- Fix a bug
- Crash or hang
- Rinse and repeat
- It suddenly works!

Tips for your development environment

You will regularly crash the kernel. Use at least two machines:

- one for editing code, reading docs, and compiling
- one for rebooting, running your modified kernel, and crash

Tracking changes with `cvs diff` becomes impossible quickly.
Use local source code version control, such as:

- `https://github.com/openbsd` for git users
- hg patch queues inside a CVS checkout
- fresh git/hg/fossil repo inside a subtree of a CVS checkout

Follow OpenBSD development

Be aware of what's going on within OpenBSD

- Subscribe to:
  - tech@openbsd.org
  - source-changes@openbsd.org
- Update all your machines and diffs to -current before embarking on another stretch of work on your project
- Read all diffs sent and/or committed by developers working within the vicinity of your driver
- Ask technical questions to these developers

Licensing

Follow the project's copyright policy:
`http://www.openbsd.org/policy.html`

- New code should be ISC-licensed
  - `/usr/share/misc/license.template`
- Honour the rights of authors
  - Many authors do not want their code copied into OpenBSD
  - This includes GPL and other incompatible open source licences
- Know your rights
  - Reverse-engineering from source or binary form for interoperability is allowed in many jurisdictions, e.g. "An objective of this exception is to make it possible to connect all components of a computer system, including those of different manufacturers, so that they can work together" (EU Directive 2009/24)

Finding information you need

Hardware without documentation is a dead brick.
Only use sources of information everyone can access:

- published or leaked documents (search the web)
- source code from other systems (Linux and FreeBSD)
- reverse-engineered information on binary drivers

Do not sign an NDA. An NDA gives you privileged information which your fellow developers cannot access. This makes effective peer review of your code impossible and breaks our community's process. **DO NOT SIGN AN NDA**

Finding new friends

Contact open source developers working on the same or similar kinds of devices. Many of them work for hardware vendors. They might help out if you get stuck.[1] Find email addresses in:

- copyright statements
- git log
- mailing list archives

Be respectful! Do not ask questions which require tedious work to answer. Contact them once your driver is almost working and demonstrate one specific problem you are stuck with. Send your source code files along.

[1] rtsx(4) and iwm(4) would still be stuck without such help

Submitting diffs

Send your working driver diff to tech@openbsd.org
Some people may respond in private, some prefer to Cc the list.
Respect their choice and reply accordingly.
Avoid common pitfalls:

- Diffs should be against -current CVS
- Diffs should be mailed inline. No attachments.
- Do not send mangled diffs. Mail the diff to yourself and apply it with patch(1) before mailing tech@ for the first time.

If you don't get any feedback, follow up to on your own thread after one week.

Extra things which need testing

Avoid regressions. If the driver already supports other devices:

- Try to obtain these devices cheaply and test them yourself
- Ask tech@ for tests of your finished diff with specific devices

Do not break snapshots!

- Run your diff through ‘make release’ – see release(8)
- Test a bsd.rd install or upgrade with your diff applied

If your diff breaks the release build and upgrade cycle Theo cannot get work done and will be rightfully annoyed at you.

Maintenance

Your project is not done once your driver is in the tree

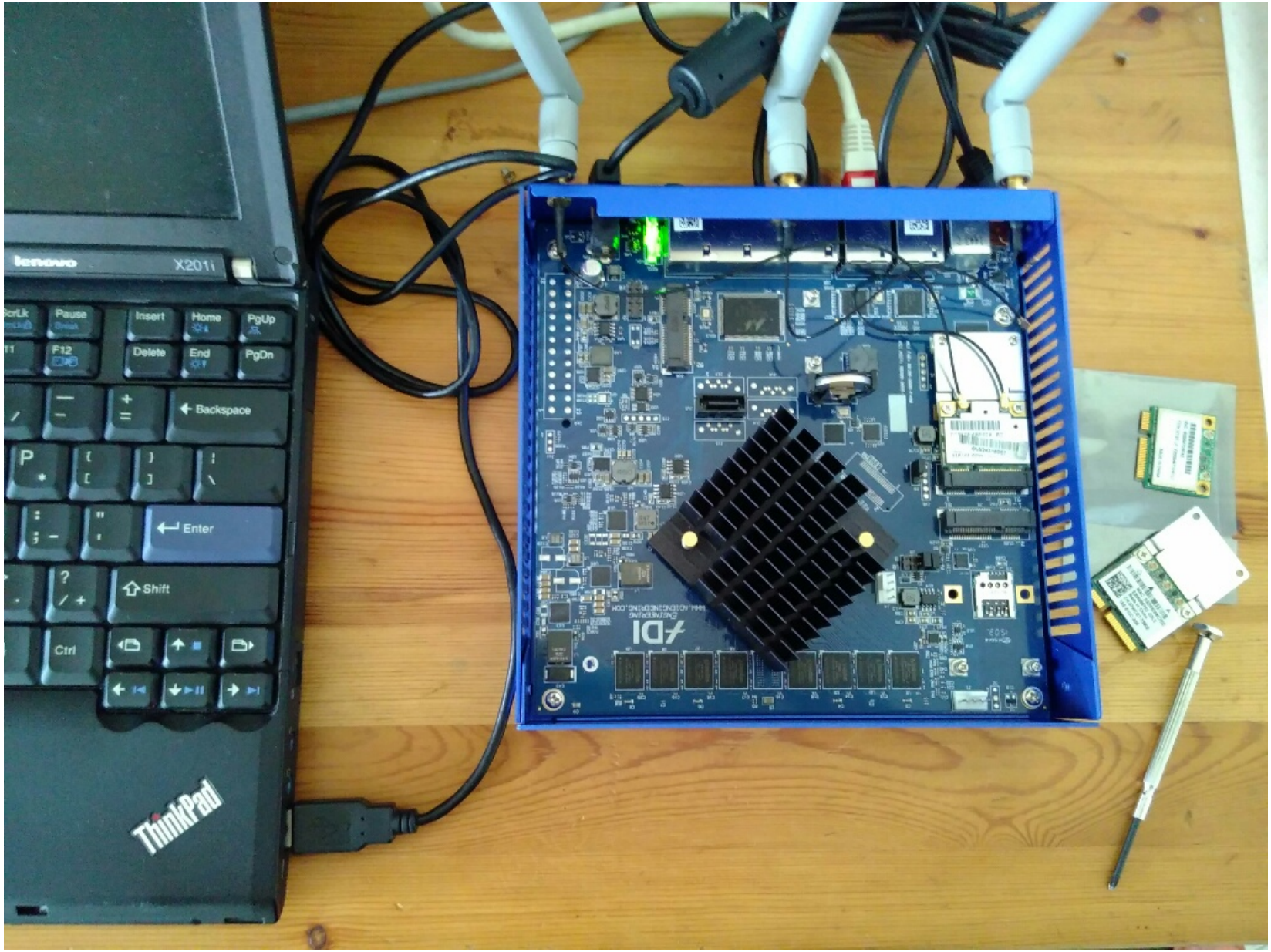
- Stay around and respond to questions and other feedback
- Be prepared for your diff being backed out in case of problems
  - You may not be able to fix every bug right away
  - After backout, take your time to make things right
- Try hard to reproduce bug reports locally
  - Some bugs only affect particular machines
  - Ask machine owners for help
- Some people in the community file useless bug reports
  - Keep asking for more information until problem is reproducible
  - If they don't provide it, it's not your problem (avoids burnout)

Coding style

Make sure your driver code looks like every other driver in the tree. Consistency is important.

- follow style(9)
- fixed-width integers in new code should use C99 type names
  - e.g. use `uint8_t` rather than `u_int8_t`
- prefix all symbols with driver name (e.g. `rtwn_`)
- no `static` functions in the kernel
- use very few .c files per directory
- use `*var.h` header for driver-specific declarations
  - function declarations
  - structs for software state
- use `*reg.h` header for device-specific definitions
  - register offsets
  - `__packed` structs for descriptors and commands

Technical topics



Device framework

Devices are organized in a device tree displayed by dmesg(8).
Device tree is managed by autoconf(9) framework.
The device tree contains one instance of your driver per device.
Drivers implement the following callbacks in `struct cfattach`:

- ca_match(): match device
  - hardware-specific port probing, e.g. in pms(4)
  - PCI vendor/product ID
  - USB vendor/product ID
- ca_attach(): initialize driver state
- ca_detach(): free resources and reset driver state
- ca_activate(): used for suspend/resume

Driver software state data structure

Each instance of the driver gets its own global software state called the *softc*. Declare it in your driver's `*var.h` header file and add any flags and variables which will be shared between functions. The cn_attach() function will receive a pointer to an allocated softc. Example for a fictional device driver xyz(4):

```
struct xyz_softc {
        struct device   sc_dev;
        bus_dma_tag_t   sc_dmat;
        bus_dmamap_t    sc_dmap;
        uint32_t        sc_flags;
#define XYZ_F_HAS_RNG   0x01 /* supports random numbers */
#define XYZ_F_JUMBO     0x02 /* supports jumbo frames */
};
```

The first softc struct member must be of type 'struct device'. This will be pre-populated with information such as the driver instance's name string (`sc_dev.drv_xname`), i.e. "xyz0", "xyz1", etc.

Adding new device IDs

The first step is to make your device known to the kernel.
Use pcidump(8) or usbdevs(8) to find your device's ID.
To add a new PCI ID:

- go to `/usr/src/sys/dev/pci/`
- edit `pcidevs` file
- run `make` to regenerate the PCI device list
- rebuild the kernel

Works accordingly for USB.

Device Registers

A "device register" is a small unit of memory inside the peripheral device. Each register has an address and a specific purpose. Its contents control or report some behavioural aspect of the device. Generally, there are:

- config registers (configure some aspect of device)
- control registers (tell device to do something)
- status registers (information provided by device)

Registers may be 8, 16, 32, or 64 bits wide.
Individual bits in registers may be

- read-only / write-only / read-write
- clear-on-read / clear-on-write

Register byte order can be big or little endian. Converting from/to host byte order is the driver's responsibility.

Device Register Documentation

Register documentation is a programmer's device user manual.
Knowing the "register map" is a pre-requisite for writing a driver.
It is documented in:

1. any half-decent data sheet
   - often secret or only available under NDA :-(
   - sometimes, leaked copies can be found on the web
2. source code of other open source device drivers
   - derived from documentation or reverse-engineered
   - comments sometimes document behaviour of the device

Device Register I/O

Identify device registers and their bits via macros.
See `sys/dev/ic/*reg.h` for examples.
Access device registers via:

- bus_space(9) read/write abstraction API
  - memory-mapped I/O (e.g. PCI devices)
  - port-mapped I/O (e.g. legacy x86 I/O devices)
- device- or bus-specific transactions
  - USB devices generally accept commands to read/write registers
  - device may provide some registers via firmware commands only
  - device may provide indirect access to some registers only
    - write the address of register X to register R1
    - now read the current value of register X from register R2

Polling

Some device registers must be polled, usually during initialization of device, before interrupts are active.

- write a register to request an operation
- keep reading a register until its value indicates success

  Implemented as a *for* or *while* loop:

  read register
  test its value; exit loop if successful
  busy-wait with DELAY(9)
  Eventually, after some amount of loop iterations, give up

Polling blocks the entire system. Don't overuse it.
Example: rtwn_fw_reset() in `sys/dev/ic/rtwn.c`

DMA (Direct Memory Access)

Using register I/O for large data transfers wastes CPU time.
With DMA we instruct the device to perform a memory read/write transfer autonomously and asynchronously.
bus_dma(9) API will allocate a block of memory below the 4GB boundary and:

1. map it into kernel virtual address (KVA) space
2. configure the IOMMU, if present, to allow device access to this block of memory

DMA buffer is usually filled/consumed by layers above the driver.
Any such access to the buffer is via the *virtual* address.
Driver needs to:

- tell device about *physical* address and size, e.g. via registers
- start the DMA transaction, then sleep or return
- handle interrupt at end of transaction

See bus_dma(9) for pseudo code and further information.

Sleeping

Drivers often need to wait for some operation to finish, e.g.

- waiting for firmware to load
- waiting for data read/write to complete
- waiting for a firmware command to complete

Use tsleep(9) to "go to sleep" and wait. Sleeping requires a *process context*. The current process[2] will be suspended and the scheduler will select another process to run.
Use wakeup(9) to wake the sleeping thread (i.e. mark it runnable for the scheduler). Usually done from *interrupt context* where completion of the pending operation is signalled by the device.

[2] kernel threads are processes, too

Interrupts

Interrupts allow communication from device to driver.

- Driver provides a function pointer to bus code

  See pci_intr_establish(9) and usbd_setup_xfer(9)

- This function is called when device signals an interrupt
- Function reads status from device and acts on events

  "I have loaded my firmware"
  "I have received a packet"
  "I am done writing data"
  "I have encountered a fatal error"

- The function may call wakeup(9) to wake a sleeping process
- The function cannot sleep!
  - No tsleep(9)
  - No malloc(9) without M_NOWAIT

Example: rtsx_intr() in `sys/dev/ic/rtsx.c`

Timeouts and Tasks

Timeouts and tasks can both create driver 'threads'.
Schedule a timeout if:

- an action needs to occur after some time has elapsed
  e.g. reset device if a command times out
- an action needs to occur periodically
  e.g. device needs regular calibration while operating

Schedule a task if:

- an interrupt needs to trigger code that must sleep
  e.g. must run another command and wait for response
- a dedicated thread of execution is required to handle events
  e.g. SD card insertion/removal thread

See timeout_add(9) and task_add(9).
The USB stack has its own task API; see usb_add_task(9).

Configuration from userspace

Configuration requests from userspace arrive via ioctl(2).

- Triggered by ifconfig(8), bioctl(8), audioctl(1), ...
- Driver's ioctl handler runs before the upper layers
  - Perform configuration change
  - Perhaps pass request to upper layers

    ieee80211_ioctl(9), ifioctl(), ...
  - Indicate success or failure

Driver's ioctl handler can sleep since it runs in the context of the process which made the ioctl system call.
The ioctl handler can trigger at any time and race interrupts, timeouts, and tasks. Software bugs thrive in such conditions!

SPL (System Priority Level)

The priority level of an interrupt is set when the interrupt handler is registered. Several levels are defined, see spl(9).
Drivers use the SPL to block interrupts from their device and any other interrupts of the same or lower priority.
Drivers commonly use:

- spltty: block interrupts from terminal devices
- splnet: block interrupts from network interfaces
- splbio: block interrupts from mass storage devices
- splsoftnet: block packet processing in network stack

ALways raise the SPL before accessing data structures which are also accessed by an interrupt handler.
Afterwards, restore the previous level with splx(9).

Refcounting tasks and timeouts

The reference-counting API can be used to keep track of tasks:

- Each task must call refcnt_rele_wake(9) before exiting
- Call refcnt_init(9) while no tasks are running
- Call refcnt_take(9) before task_add(9)
- If task_add(9) returns zero, call refcnt_rele_wake(9)
- If task_del(9) returns non-zero, call refcnt_rele(9)

All running tasks are now reference-counted.
To stop all tasks, e.g. in the ioctl handler:

- Use task_del(9) to remove all pending tasks from the queue
  If task_del(9) returns non-zero, call refcnt_rele(9)
- Call refcnt_finalize(9) to sleep until all running tasks are done
  The last task exiting will wake refcnt_finalize(9)

This also works with timeouts.

Firmware

Use loadfirmware(8) to load firmware files from `/etc/firmware`.
You need a process context for this.
Pay attention to the licence of firmware files!

- If firmware is distributable without restrictions it can go into the base system; see `/usr/src/sys/dev/microcode/`
- If distribution is restricted[3], firmware will be banished to ports
  - Create or modify a firmware port; see `/usr/ports/sysutils/firmware/`
  - fw_update(8) will install firmware on systems which need it

If distribution rights are unclear, email the hardware vendor and ask. Keep an archived copy of any replies for future reference.

[3] Including any restrictions of your rights, such as the right to modify or reverse-engineer

Suspend/Resume

If you don't implement suspend/resume, some laptops might not suspend/resume anymore. This will be treated as a regression.
Implement a config_activate(9) handler. It will be called with:

- DVACT_QUIESCE – suspend is imminent; you can still sleep
- DVACT_SUSPEND – suspending; you cannot sleep

  save register state to memory here and/or stop device
- DVACT_RESUME – resuming; you cannot sleep

  restore register state from memory here and/or restart device
- DVACT_WAKEUP – resuming; you can sleep[4]

Upper layers are already paused when your handler runs.
USB devices will be detached upon suspend.
Examples: rtsx_activate() in `sys/dev/ic/rtsx.c`
iwn_activate() in `sys/dev/pci/if_iwn.c`

[4] sleeps in the acpi thread

Debugging

Debugging drivers

You will earn more experience in debugging than you ever wanted. Expect vastly more time spent on debugging than writing code.

- If possible, get a serial console
- Learn how to use ddb(4)

  boot **bsd.gdb**[5] to get line numbers in ddb(4)!
- printf(9) adds latency which can hide race condition bugs
- frequent printf(9) calls will lock up your machine
- You can break into ddb to set your driver's print debug level:

```
ddb> set iwm_debug = 2
ddb> continue
```

- Can't find the problem? Try harder...
- Found the problem? Here's another...

[5] can be found at `compile/GENERIC.MP/obj/bsd.gdb`

Tracking down bugs

The bugs will usually be in your code and obvious in hindsight.
I have spent hours tracking down in my code:

- uninitialized variables; undefined behaviour
- logic errors; code does not do what was intended
- bad length calculations; off-by-ones
- typos and copy-paste errors
- use after free
- race conditions
- bits written to registers inverted
- undocumented register bits that need to be set

Search starts out based on erratic hardware behaviour. A good strategy is to stop thinking and look. Ignore things you believe you already know. Find and observe facts.

Some example bugs (1/2)

Obvious in hindsight...

- rtwn(4)
  - iqcal code wrote garbage stack memory to registers due to incorrectly scoped struct variable (caught by clang -O2)
  - 8192CE baseband initialization values got an extra copy-pasted line during refactoring merge with urtwn(4); radio was deaf
  - must wait for firmware to boot or accessing PCI config space registers will hang entire machine (such quirky hardware)
- rtsx(4)
  - asked hardware to transfer 0 bytes per sector instead of 512
  - asked hardware to read when we wanted to write and vice versa

Some example bugs (2/2)

- xhci(4)
  - isochronous transfer used remaining length of N-1 descriptors where remaining length of N descriptors was needed; the last descriptor which has the "interrupt on completion" flag was not processed – hence no interrupt
- iwm(4)
  - did not set a bit which makes 8260 device send one interrupt per received frame; bit definition had already been deleted from Linux code and was hard to find (their driver doesn't need it)
  - asked hardware to send ACK frames at fast (a.k.a. fragile) data rates; the AP missed most ACKs and retried excessively

See also other people's presentations

Other people's presentations (1/2)

`https://www.openbsd.org/events.html`

- Mike Belopuhov

  Implementation of Xen PVHVM Drivers in OpenBSD (2016)
  OpenBSD Kernel Architecture, Network Stack (2008)

- Vladimir Kirillov

  OpenBSD Kernel Internals: The Hitchhiker's Guide. (2009)

- Jason L Wright

  Hardware Is Wrong, or "They can Fix It In Software". (2008)

- Theo de Raadt

  Hardware Documentation (2007)

- Jonathan Gray

  Driver architecture and implementation in OpenBSD (2006)

Other people's presentations (2/2)

- Constantine A. Murenin

  OpenBSD Hardware Sensors Framework (2009)

- Claudio Jeker

  OpenBSD Network Stack Internals (2008)

- Marc Balmer

  Support for Radio Clocks in OpenBSD (2007)
  Support for Time Signal Station Receivers and GPS in OpenBSD (2006)

- David Gwynne and Marco Peereboom

  bio and sensors in OpenBSD (2006)

- Niall O'Higgins and Uwe Stuehler

  Embedded OpenBSD (2005)

Further reading

- OpenBSD drivers for similar devices
- Drivers for your device on other open source systems
- Man page recommendations:

  style(9), autoconf(9), pci_intr_establish(9), pci_make_tag(9), usbd_transfer(9), usbd_open_pipe(9), usbd_close_pipe(9), spl(9), tsleep(9), wakeup(9), usbd_ref_wait(9), refcnt_init(9), timeout(9), task_add(9), usb_add_task(9), loadfirmware(9), kern(9), dohooks(9), copy(9), bpf_mtap(9), ieee80211(9)
- Driver development books for other systems[6] can help where concepts translate well. However, other systems tend to be more complicated...

[6] There is none specific to OpenBSD :(

Thank you for listening!

Questions?