In the Kernel Trenches: Mastering Ethernet Drivers on Linux

Maxime Chevallier
maxime.chevallier@bootlin.com

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Maxime Chevallier

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
- Living near **Toulouse**, France

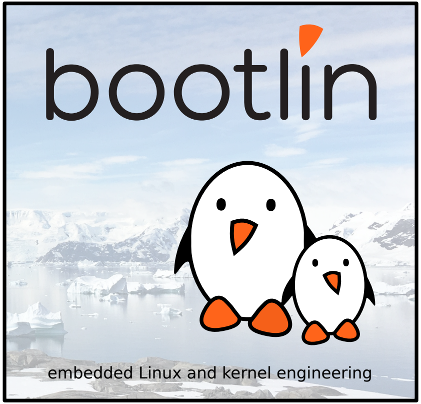
What is this all about ?

- Take a look at what Ethernet Drivers do
- What are they in charge of ?
- Which kernel subsystems and frameworks to they interact with ?
- Focus on drivers found on Embedded Systems
  - Not the same constraints as a High-Speed Datacented Networking driver
  - What we will see still applies for these drivers :)

Ethernet Controller

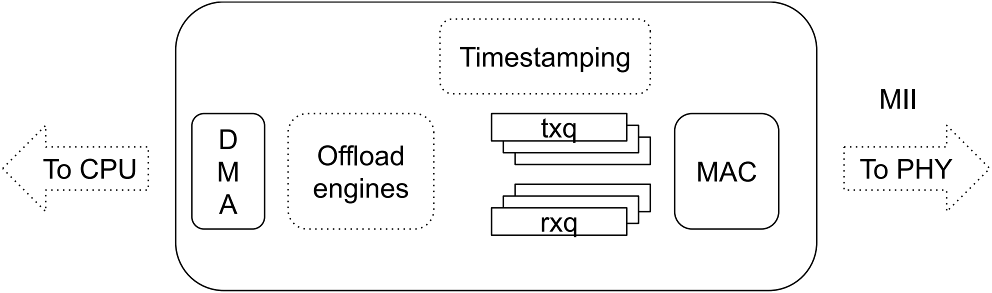
Maxime Chevallier
maxime.chevallier@bootlin.com

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!



Ethernet controller - Inside look

- MII interface to PHY
- MAC : 802.3 operations (SoF, collision management, flow-control, Idle word, IPG)
- Queues and DMA
- Internal engines : Timestamping, Filtering, Parsing, Encryption, Switching...



struct net_device

- Represents a network interface
- Backbone of the driver
- The `.probe()` function of the driver usually registers via `netdev_register()`
- `net_device`s are network **interfaces**, visible with `ip link show`
- Each `net_device` has it's unique `ifindex` within it's namespace (`struct net`)
- netdevs can be part of a hierarchy : `lower` and `upper` devices
- Allocated through `(devm_)alloc_etherdev_mqs(priv_size, txqs, rxqs)`
- Driver-specific data retrieved using `netdev_priv(dev)`
- User-visible right after `register_netdev()`

struct net_device_ops

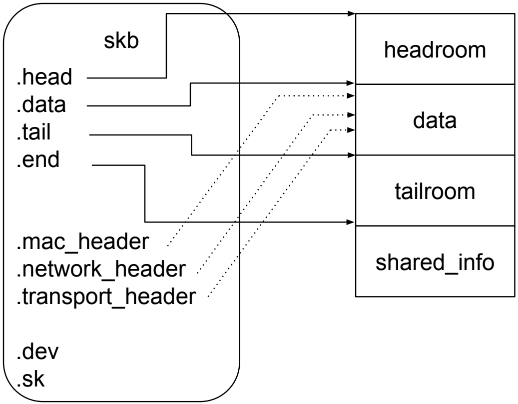
- Callbacks that the driver exposes to the net core
- Referred to as "NDOs"
- Some are on the data path, some on the control path
- One is mandatory :
  - `.ndo_start_xmit()`, to transmit data
- Others might be required depending on the exposed `features`
- Specified at init time, **before registration** :
  - `netdev->netdev_ops = &my_netdev_ops;`

Data Path

Maxime Chevallier
maxime.chevallier@bootlin.com

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

struct sk_buff



- `struct sk_buff` (**sock**et **buff**er)
- By convention, pointers to such objects are very often named `skb`
- Represents a Packet through it's traversal of the kernel networking stack
- Created by the Ethernet Driver on RX (`build_skb(data, frag_size)`)
- Consumed on TX (`kfree_skb`, `dev_kfree_skb_any` and similar)
- Can be a simple packet, or a fragmented packet
- Contains a data section (payload + headers) and metadata

Page Pool

- Designed to optimize buffer allocation for DMA transfers
- Maintain a pool of memory pages that stays mapped for the device
- Allow buffer recycling : `skb_mark_for_recycle()`
- Prerequisite for XDP, but can be used as-is
- Not mandatory, but useful for better performances !
- Documentation available at https://docs.kernel.org/networking/page_pool.html

If Page Pool isn't used, manual DMA mapping/unmapping for RX/TX must be done.

TX path

- `.ndo_start_xmit()` is called by the core, passing an `skb` as a parameter
- The driver will create and enqueue DMA descriptors
- The driver must take care of sending each fragments and segments
- If supported, the `tx queue` on which to enqueue the frame must be retrieved with `skb_get_queue_mapping(skb)`
- Controllers usually raise an interrupt when a packet has been transmitted
- The driver reports how many bytes were sent, for BQL (Bufferbloat)
  - `netdev(_tx)_send_queue` upon enqueueing
  - `netdev(_tx)_completed_queue` upon completion
- The `skb` can be released once it's been sent

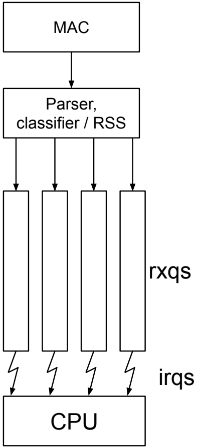
~~*New API*~~ **NAPI** means **NAPI**

- Process RX packets in budgeted `poll` loops after a first packet gets received
- **NAPI Instances** are registered through `netif_napi_add()`, passing a `poll` callback
  1. The fist RX packet raises an interrupt
  2. Driver calls `napi_schedule()` and keeps interrupts masked
  3. The `poll` callback of a driver is called, with a budget of N packets to process at most
  4. Once N or all packets are processed, the interrupt is re-enabled
- Runs in softirq context, can be switched to threads
- Also works for TX, for processing TX completions
- There can be multiple NAPI instances (e.g. one per queue)
- https://docs.kernel.org/networking/napi.html

Timestamping

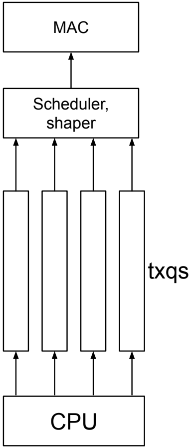
- Ethernet controllers can have precise timestamping units
- Configured through :
  - `.ndo_hwtstamp_get()` and `.ndo_hwtstamp_set()` *(new)*
  - `SIOCGHWTSTAMP` ioctl *(legacy)*
- Upon RX, the driver grabs the timestamp from the controller
  - Sets it in the skb's `struct skb_shared_hwtstamps`
  - Retrieved using `skb_hwtstamps()`
- Upon TX, the driver checks `skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP`
  - If timestamping is possible, set `skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;`
  - call `skb_tx_timestamp()` in any case as close to the SKB being sent
  - call `skb_tstamp_tx(skb)` when the timestamp is available
- The PHY might also timestamp the packet.
- See https://www.kernel.org/doc/html/next/networking/timestamping.html

Queues - RX



- It's common to have more than one queue per direction
- **RX queues**, often called `rxq`
  - Ingress traffic is **steered** towards different queues
  - `rxq` can then be assigned dedicated `irq`
  - `per-queue interrupt` can be pinned per-CPU
  - Needs some hardware packet parsing support
  - Spread traffic across queues based on a hash : `RSS`
  - Steer individual flows towards dedicated queues : `tc`, `rxfnc`

Queues - TX



- **TX queues**, often called `txq`
  - Egress traffic enqueued on several queues
  - `XPS` : eXpress Packet Steering, one queue per CPU
  - `mqprio` : Queues are mapped to priorities
  - Can then be used for TSN and Time-aware scheduling
  - Can be used for QoS (DCB VLAN priorisation)
- `skb_get_queue_mapping()` to retrieve the queue index for an `skb`
  - Used in `.ndo_start_xmit()`
- https://docs.kernel.org/networking/scaling.html

XDP

- XDP allows running eBPF programs directly at the driver level
- Useful for filtering, redirecting, analyzing traffic
- XDP is driver-dependent, and requires the driver to use `page_pool`
- eBPF program gets loaded with `.ndo_bpf`
- If a program is loaded, run it in the `NAPI poll` loop
- Dedicated NDO for xmit : `.ndo_xdp_xmit`
- Documentation at https://docs.cilium.io/en/latest/bpf/progtypes/#xdp
- Reference driver : https://elixir.bootlin.com/linux/latest/source/drivers/net/ethernet/marvell/mvneta.c

Control Plane

Maxime Chevallier
maxime.chevallier@bootlin.com

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Control Plane

- Ethernet controllers are highly configurable
- Stats reporting at various level (per-queue, MAC, PHY, internal engines)
- Offload configuration : Vlan filters, classification, checksumming
- Ethernet configuration : MTU, Link speed, Flow control
- Some run under `rtnl_lock`
  - Serializes network configuration
  - Some ops such as `ethtool_ops` must be called under `rtnl_lock()`
  - Unlock with `rtnl_unlock()`
  - Use `ASSERT_RTNL()` if your code relies on it being held by the caller
- Might still need to be fast !

Entry-points

- NDOs
  - netdev features, TC, MTU configuration, etc.
- Other sets of ops in `net_device`
  - `ethtool_ops`, `macsec_ops`, ...
  - Usually set prior to calling `register_netdev()`
- Notifiers
  - Registered hooks : `register_netdevice_notifiers`
  - e.g. switchdev relies on `register_switchdev_notifiers`
  - Driver decides which notification is relevant for it
- Ioctls
  - Timestamping (moved to NDO)
  - PHY control (handled by `phylib` and `phylink`)
  - Being gradually replaced
- Registered anciliary functions
  - `struct phylink_ops`
  - `struct mii_bus`
  - `struct ptp_clock_info`

NDOs

- Currently 92 defined ops
- Start and Stop the interface
  - `ip link set eth0 up/down` => `.ndo_open()` / `.ndo_close()` called
- Gather stats : `.ndo_get_stats64`
- Set RX mode (e.g. promisc mode) : `.ndo_set_rx_mode()`
- Specific features : VFs, Briding, FCoE, VLAN filtering...

netdev features

- features represents hardware offload capabilities
  - Checksumming, Scatter-gather, segmentation, filtering (mac / vlan)
  - see ethtool -k <iface>
  - attributes of struct net_device
- Drivers set netdev.hw_features at init, and can also set netdev.features
  - features : The current active features
  - hw_features : Features that can be changed (*hw != hardware*)
- Users but also the core might want to change the enabled features
  - Child devices might require some features to be disabled
- .ndo_fix_features() filters incompatible feature sets for the driver
- .ndo_set_features() applies the new feature set
- https://docs.kernel.org/networking/netdev-features.html

ethtool

- API to report and control ethernet-specific parameters
- These settings and parameters are accessible with `ethtool`
- Uses a legacy `ioctl` interface, superseded by `netlink`
- Uses a dedicated set of ops : `struct ethtool_ops`
- `netdev->ethtool_ops` are set before driver registration
- All `ethtool_ops` are optional, and must run under `rtnl_lock`
- Around 70 different ops
- Userspace API : https://docs.kernel.org/networking/ethtool-netlink.html

struct ethtool_ops

- Fine-grained Hardware statistics gathering
- Link parameters : Supported modes, flow-control, status, Link-partner, speed, aneg...
- Flow classification : Hardware steering to queues, filtering
- RSS : Indirection table(s) configuration, key configuration
- Channels configuration : Map queues to Interrupts
- EEE, FEC, Interrupt Coalescing, WoL, SFP modules, Self-tests, Register dumps...
- see `struct ethtool_ops` definition

More ethtool ops exists for PHY devices

TC

- Traffic Control : shaping, policing, scheduling, dropping
- Some of the TC operations can be offloaded to Hardware :
- `.ndo_setup_tc(netdev, type, *data)` is the main entry-point
- `type == TC_SETUP_QDISC_MQPRIO`
  - Setup the hardware queue prio mapping
  - Setup per-queue rate-limit
- `type == TC_SETUP_QDISC_TAPRIO`
  - Time Aware per-queue scheduling
- `type == TC_SETUP_QDISC_CLSFLOWER`
  - Flow steering, assigning different RX flows to dedicated queues
- `type == TC_SETUP_QDISC_CBS`
  - Shaping: Limiting transmit speed
- Documentation : see the code...

switchdev

- When the interface is part of an internal switch
- The entrypoints are based on `notifiers` :
  - `register_switchdev_notifier`
  - `register_switchdev_blocking_notifier`
- The same driver handles multiple ports (one `net_device` per port)
- Each port should work as a standalone interface at init
- `bridging` operations are then offloaded
- `FDB`, `MDB`, `VLAN` additions and removals are configured into hardware tables
- see https://docs.kernel.org/networking/switchdev.html

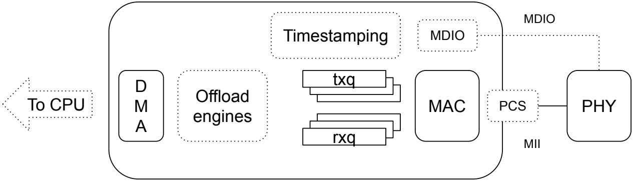
Precision Time Protocol

- For timestamping, Ethernet devices might have internal clocks
- These clocks can be synchronized using Precision Time Protocol
- `linuxptp` provides userspace tools that implements PTP
- `struct ptp_clock` represents such a clock (PHC)
- Dedicated ops for clock configuration : `struct ptp_clock_info`
  - `.adjfine()` to adjust the frequency
  - `.adjtime()` to adjust the time
  - `.get/settime64` to set the time
- Clock is registered through `ptp_clock_register`
- **timestamping** settings and **clock** settings are separated
  - Timestamping through `.ndo_hwtstamp_get/set`
  - Clock through `ptp_clock_info`'s ops
- https://docs.kernel.org/driver-api/ptp.html

Other ops

- `macsec_ops`
  - For MACSec (802.1AE) offloading
- `xfrmdev_ops`
  - For IPSec offloading
- `tlsdev_ops`
  - For TLS offloading
- `dcbnl_ops`
  - For DataCenter Bridging offload

PHY



- Ethernet PHYs are in charge of handling the Layer 1 aspects of a transfer
- There exists dedicated chips or IP blocks, which have dedicated drivers
- Ethernet drivers need to attach to a PHY, and notify the PHY layer when :
  - The PHY should start
  - The PHY should stop
  - A few more specific operations such as suspend/resume
- The PHY layer will notify the Ethernet driver when the link changes

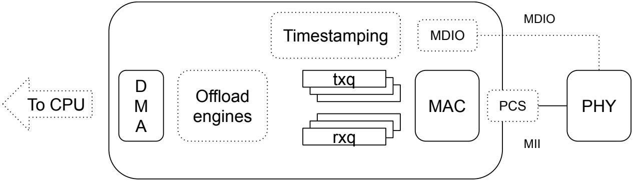
phylib

- Framework to write PHY drivers and maintain a PHY's internal state
- Out of the scope of this talk
- Can be interacted with from Ethernet drivers
- `struct phy_device` represents an Ethernet PHY
- The Ethernet driver is in charge of registering the PHY and attaching to it
- In some cases, the Ethernet driver will also act as a MDIO bus driver
  - Needs to register a `struct mii_bus`
- see https://www.kernel.org/doc/html/next/networking/phy.html
- `phylink` is now preferred, instead of manually dealing with the PHY

phylink

- `phylink` handles the link between the Ethernet controller and the PHY
- The `MII` (MAC to PHY) link can sometimes need dynamic reconfiguration
- `phylink` manages the PHY, using `phylib` and deals with it's registration.
- `phylink` supports **SFP** cages connection
- The Ethernet driver shall provide `phylink_mac_ops`
  - `.mac_link_up()` and `.mac_link_down()` called depending on the link state
    - Established link settings can be configured
    - `speed`, `duplex`, `pause` settings
  - `.mac_config` called when the link changes:
    - `phy_interface_t` modification
    - Autoneg mode : From PHY, Inband or Fixed
- `phylink_create(cfg, fwnode, interface, ops);` then `phylink_of_phy_connect(pl, dn, flags);`
- see https://www.kernel.org/doc/html/next/networking/sfp-phylink.html

PCS



- Some Ethernet Controllers can have one or more Physical Coding Sublayer blocks
- PCS can also sometimes be handled through an external driver
- **phylink** supports dedicated PCS control through `struct phylink_pcs`
- The `phylink_pcs` is either locally crafted, or use dedicated drivers
  - Retrieved from the device-tree through `pcs-handle`
- The Ethernet driver indicates to phylink which PCS to use
  - `.mac_select_pcs` phylink ops

Coding style

- Comments must be :

```
/* I am a comment
 * using netdev-style format
 * this is beautiful
 */
```

- and not :

```
/*
 * I am not a comment
 * using netdev-style format
 */
```

- Use reverse christmas-tree (RCS) declaration
- local variables declaration from longest to shortest *if possible*:

```
struct net_device *dev;
struct sk_buff *skb;
unsigned int rxq;
int err;
```

Contributing

- Send your patches to the `netdev@vger.kernel.org` list (lore archive)
- Two git trees are maintained :
  - net-next : For new features
    - https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git
  - net : For fixes
    - https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git
- net-next closes during each merge window (no patches accepted)
  - Quick status check here : https://patchwork.hopto.org/net-next.html
- Indicate the tree (net or net-next) your patches target in the subject :
  - `git format-patch --subject-prefix='PATCH net-next' ...`
- Fast-paced development, but high-volume list
- Help with reviews can't hurt :)
- https://www.kernel.org/doc/html/next/process/maintainer-netdev.html

Questions? Suggestions? Comments?

Maxime Chevallier

maxime.chevallier@bootlin.com

Slides under CC-BY-SA 3.0

https://bootlin.com/pub/conferences/2024/elc/mastering-ethernet-drivers-linux.pdf