ChatGPT

Linux Network Device Drivers (Updated for Kernel 6.x)

Introduction

Linux network interfaces are a distinct class of devices that do not appear as files in `/dev` – instead, they exist in their own namespace and are accessed via sockets. Unlike block drivers (which only respond to kernel requests to read/write data), network drivers must handle asynchronous inbound data (packets arriving from the outside) [1]. Thus, the network driver API is designed for an event-driven model where the device "pushes" incoming packets to the kernel. Network drivers also support various administrative operations (setting MAC addresses, adjusting transmission parameters, reporting traffic statistics, etc.) beyond simple read/write [2]. The kernel's networking subsystem is largely protocol-independent, meaning drivers can be written for Ethernet, Wi-Fi, etc., without being tied to a specific network layer protocol [3]. In this guide, we'll walk through the architecture of Linux network drivers as of kernel 6.x, covering the lifecycle of a network interface, key data structures, the transmit and receive data paths, interrupt handling with NAPI (New API) polling, and integration with user-space tools. We will also highlight updates since older kernels (e.g., Linux 2.6) – such as the use of `net_device_ops`, 64-bit statistics, and modern features like multiqueue and XDP – to bring the material up to date. Finally, we'll propose a hands-on project: implementing a basic PCIe NIC driver for a QEMU-emulated Intel e1000 device, outlining the major steps and resources for that endeavor.

Registering a Network Device (Connecting to the Kernel)

To make a network interface known to the kernel, a driver must allocate and register a `struct net_device` for it. In modern kernels, this is typically done with helpers like `alloc_netdev()` or `alloc_etherdev()`, which allocate and initialize a net_device structure (including space for driver-private data) [4] [5]. For example, Ethernet drivers often use `alloc_etherdev(sizeof(priv))`, which calls `ether_setup` to fill in standard Ethernet defaults (such as MTU = 1500, hardware header length = 14, etc.) [6] [7]. After allocation, the driver sets the device's fields: a unique name (e.g. "eth%d" for dynamic numbering), the hardware address (`dev_addr`), and the pointer to its operations (`dev->netdev_ops`). In kernels 6.x, the `net_device` no longer embeds function pointers for ops directly; instead it has a `netdev_ops` pointer to a table of operations (callbacks) defined by the driver [8]. The driver fills a `static const struct net_device_ops` with its implementation of required methods (open, stop, start_xmit, etc.) and assigns `dev->netdev_ops` to point to it before registration [9]. It may also assign `dev->header_ops` for link-layer header handling if needed (Ethernet drivers usually use the default `eth_header` set by ether_setup) and set flags for device features (e.g., checksum offload capabilities, described later). Once setup is complete, the driver calls `register_netdev(dev)`, which adds the interface to the kernel's global list and makes it visible to system networking code [10] [11]. After registration, the interface will appear in `/sys/class/net/` and commands like `ip link` or `ifconfig` can interact with it.

The `struct net_device` – This core structure represents the network interface within the kernel. It contains identifying information and numerous fields controlling interface behavior. Key fields include:
- **Name and Index:** `dev->name` (e.g. `"eth0"`) and `dev->ifindex` (a unique numeric identifier). These are set during registration.
- **Hardware Address:** `dev->dev_addr` (and length `addr_len`), which holds the MAC address for hardware interfaces [12]. The driver must populate this (often reading from device EEPROM) before registering. There's also a `broadcast` address (all 0xFF for Ethernet) set by ether_setup [13].
- **MTU:** `dev->mtu` is the Maximum Transfer Unit – e.g. 1500 bytes for standard Ethernet [14]. Drivers can override the default if hardware needs a different MTU, and should implement an .ndo_change_mtu method if MTU changes are allowed.
- **Flags:** `dev->flags` is a bitmask of interface flags (prefixed `IFF_` in `<linux/if.h>`). Some are set by the kernel (e.g. `IFF_UP` when the interface is running), others by the driver to indicate capabilities. For example, `IFF_NOARP` for devices that don't use ARP, `IFF_PROMISC` (promiscuous mode), `IFF_MULTICAST` if the device supports multicast filtering, etc. [15] [16]. When certain flags change (promiscuous or all-multicast), the kernel will invoke the driver's callback to update hardware filtering (discussed under "Multicast" below).
- **Interface Type:** `dev->type` identifies the link layer protocol (e.g. `ARPHRD_ETHER` for Ethernet) [17]. This and related fields are set by helper setup functions (ether_setup, etc.) for common device types.
- **Queues:** `dev->tx_queue_len` sets the length of the software transmit queue (number of packets that can be queued when interface is congested) [18]. Default is 1000 for Ethernet; drivers may adjust (for instance, a low-throughput link like a parallel port ("plip") sets 10 to save memory [18]). Modern devices can have multiple transmit queues – in which case `dev->num_tx_queues` and associated structures manage each queue. By default (single-queue), there is one transmit queue; multi-queue is discussed later.
- **Device Resources:** `dev->irq` and `dev->base_addr` historically store the interrupt line and I/O base address (for devices with legacy I/O ports) [19] [20]. On modern PCIe NICs, these may be informational (the actual MSI/MSI-X vectors aren't stored here, and MMIO base address is handled via ioremap of PCI BARs, not directly in `net_device`). Still, `ifconfig` displays the IRQ and base_addr for reference [21].
- **Operations Pointers:** Most important are `dev->netdev_ops` (the table of driver callbacks) [8] and `dev->ethtool_ops` (for driver-specific `ethtool` commands) [22]. Also, `dev->header_ops` can point to functions for building link-layer headers if custom (Ethernet drivers typically use the default).
- **Stats:** `dev->stats` is a struct `net_device_stats` that holds traffic counters (packets/bytes sent/received, errors, drops, etc.). In modern usage, drivers often maintain stats in their private data and use 64-bit counters, exposing them via `.ndo_get_stats64` or `.ndo_get_stats` callback. However, `dev->stats` is still present for quick access to basic 32-bit counters and is used for `/proc/net/dev` output [23].
- **Private Data:** In legacy (2.4) kernels, net_device had a `void *priv`; now, private data is allocated *alongside* the net_device structure. Drivers obtain it with `netdev_priv(dev)` which returns a pointer to the memory region trailing the net_device [24] [25]. This region (of size specified to `alloc_netdev`) is where the driver stores per-interface state (e.g. rings, stats, locks, etc.). By placing it adjacent, cache efficiency is improved. The `net_device` itself is large, but defaults are set so drivers often only need to adjust a few fields for basic operation [26].

Overall, `struct net_device` is quite large and complex, but helper functions and defaults (especially for Ethernet) save the driver writer from filling out every field [4]. Typically, after calling `alloc_etherdev`, a driver only needs to set up the MAC address, any device-specific flags or feature flags, and the operations pointers [27]. The kernel's network core will handle common behaviors based on these settings.

Network Device Operations (net_device_ops)

Each network interface driver implements a set of callback functions (methods) that the kernel calls to operate the device. In kernel 6.x, these are defined in a `net_device_ops` struct and assigned to `dev->netdev_ops` as described. The most essential methods are:

- **ndo_open** – Called when the network interface is enabled (e.g. `ifconfig up` or `ip link set <dev> up`). This corresponds to "starting" the device. The driver should allocate hardware resources (DMA buffers, descriptor rings), activate the hardware (e.g. enable receiver/transmitter in device registers), and register for interrupts. If using NAPI, this is where `netif_napi_add()` is called to initialize the poll mechanism, and where the driver typically **enables the transmit queue** via `netif_start_queue()` [28]. For example, if the device needs an IRQ line, the open function would call `request_irq()` here. It should also set the device's carrier state: usually `netif_carrier_on(dev)` to indicate link is up (or off if no link until cable plugged in). On success, ndo_open returns 0; on failure (e.g., hardware missing), it returns a negative error code and the interface stays down.

- **ndo_stop** – Called when an interface is deactivated (`ifconfig down`). It should reverse the ndo_open steps: disable the hardware from receiving or sending, stop and purge transmit queues, and free any allocated resources. For instance, the driver should mask or disable device interrupts, unregister the interrupt handler, and free DMA buffers. If NAPI is used, call `netif_napi_del()` to remove the poll handler. The transmit queue is typically stopped here as well (with `netif_stop_queue()`) [29] in case it wasn't already. After ndo_stop, the interface is considered down (kernel clears IFF_UP flag).

- **ndo_start_xmit** – This is the packet transmission function, invoked by the kernel when a packet is ready to send. The function signature is `netdev_tx_t ndo_start_xmit(struct sk_buff *skb, struct net_device *dev)`. The driver is given a pointer to a socket buffer (sk_buff) which holds the fully constructed packet (including all protocol headers and the link-layer header) [30]. The driver's job is to map this buffer for DMA (if not already accessible to device), queue it on the hardware transmit ring, and notify the device that a new packet is available. Let's break down a typical transmit flow:

- **Buffer Mapping:** If the NIC uses DMA, the skb's data must be accessible via a DMA address. Drivers often call `dma_map_single(dev, skb->data, skb->len, DMA_TO_DEVICE)` to get a DMA address (unless the device supports scatter-gather and the skb has fragments – then each fragment might be mapped). The driver may also ensure the data is aligned or copied to bounce buffers if required by hardware. Many NICs support scatter-gather and can transmit multi-fragment skbs, in which case the driver will populate multiple descriptors for one packet.

- **Descriptor Setup:** The driver finds the next free transmit descriptor in its ring (usually tracked by a "tail index"). It fills in the descriptor with the DMA address and length of the buffer. It also sets control bits as needed – for example, a descriptor might have flags for "end of packet" (EOP), "report status" (tell NIC to generate an interrupt when done with this packet), checksum offload commands, etc., depending on NIC. In modern Intel NIC descriptors, for instance, there are fields for command and status bits.

- **Update Ring Pointers:** The driver updates the ring's tail index (often by writing to a MMIO register on the NIC). For many devices, writing the tail register tells the NIC that new descriptors up to that index are ready to transmit [31] [32]. The NIC will then fetch the descriptors and start sending the packets out on the wire.
- **Stopping the Queue:** If the transmit ring is now full (or near full) after adding this packet, the driver should stop the software queue to throttle the upper layers. It calls `netif_stop_queue(dev)` to signal to the kernel that it should not hand more packets to ndo_start_xmit for now [33]. This prevents race conditions with multiple CPUs – `netif_stop_queue` also ensures that no other CPU is currently running the xmit function when it returns [34]. The kernel will not invoke ndo_start_xmit again until the driver re-enables the queue.
- **Return Code:** The ndo_start_xmit function returns `NETDEV_TX_OK` on success. If the driver finds that it cannot send the packet (e.g., no free descriptor), it can return `NETDEV_TX_BUSY` *without* freeing the skb – this signals the stack that it should retry later. In practice, drivers avoid returning BUSY by stopping the queue when full; a BUSY return is generally only used in race conditions (e.g., two CPUs tried to transmit simultaneously and one filled the ring first).

The actual transmission is asynchronous – ndo_start_xmit only enqueues the packet. The skb's memory should not be touched after ndo_start_xmit returns; ownership passes to the driver/NIC. Once the NIC transmits the packet, the driver will clean up (usually in a separate "TX completion" step, often done in interrupt context or in a NAPI poll). At cleanup, the driver unmaps the DMA and frees the skb (via `dev_kfree_skb()` or similar). Drivers often keep an array of skb pointers parallel to the descriptor ring to remember which skb corresponds to which descriptor, so they can free them when the NIC reports completion.

- **ndo_tx_timeout** – This method is called by the kernel if a packet transmission "hangs". The kernel networking layer starts a watchdog timer whenever a packet is queued for transmit. If too much time passes (by default, 5 seconds) without the driver freeing/transmitting packets (i.e., the transmit ring seems stuck and `netif_queue_stopped()` remains true), the kernel presumes the NIC or driver is wedged and invokes the `.ndo_tx_timeout` handler [35]. The driver's tx_timeout should print a warning and initiate recovery: typically, it will reset the hardware or reinitialize the transmit ring to get things moving again [36] [37]. For example, it might record an error counter, then simulate a hardware interrupt to kick off the recovery or directly call the same routine used in an ISR to clean the ring [37]. After a reset, it should call `netif_wake_queue()` to resume traffic. A simple tx_timeout handler from our sample driver might log the timeout (including `jiffies` to see latency) and then call the same code as an interrupt service routine to clean and restart transmission [38].

- **ndo_get_stats64 / ndo_get_stats** – Used to retrieve network statistics. In older drivers, `.get_stats` returned a pointer to a static `net_device_stats` struct (which typically lived in the driver's private data) [39]. In modern usage, `.ndo_get_stats64` is preferred, which fills a `rtnl_link_stats64` structure with 64-bit counters for each stat. The kernel calls this when users request stats (e.g., `ifconfig`, `ip -s link`, or reading `/proc/net/dev`). The driver should populate all relevant fields (tx_packets, rx_packets, tx_bytes, rx_bytes, tx_errors, rx_errors, etc.). It must be safe to call at any time, even when the interface is down [40], so the stats need to persist as long as the net_device exists. A typical implementation simply copies from the driver's internal counters. For example, our driver's `ndo_get_stats64` might do: `stats->tx_packets = priv->tx_packets; stats->tx_bytes = priv->tx_bytes; …` etc., under proper locking or using

atomic64 (or u64_stats_sync) to avoid tearing on 32-bit. Many drivers accumulate stats in `netdev->stats` for legacy reasons, updating them on the fly; if so, returning a pointer to that or copying it is enough. In any case, providing accurate stats is important for user visibility. Common counters and their meanings are: packets and bytes transmitted/received, errors (packet failed to send or bad packets received), dropped (packets the driver had to drop due to lack of resources), multicast (count of multicast frames received) [41], collisions (on half-duplex Ethernet), etc.

- **ndo_set_rx_mode** – This handler is called when the multicast or promiscuous mode configuration of the device changes. In older kernels this was `.set_multicast_list` [42], but now the more general name is `.ndo_set_rx_mode`. The kernel calls it whenever `dev->flags` related to IFF_PROMISC or IFF_ALLMULTI change, or when the list of multicast addresses to filter is updated [43] [44]. The driver must program the hardware's receive filter accordingly. For example, if `IFF_PROMISC` is set, the NIC should be put into promiscuous mode (accept all packets). If promiscuous is off, but `IFF_ALLMULTI` is set, the NIC can still filter unicast but should accept all multicast frames. If neither is set, then the NIC should filter multicast frames against the specific addresses in the device's multicast list (accessible via `dev->mc_list` and count `dev->mc_count` [45]). Depending on hardware, implementing this might involve writing a set of hash filter registers or a multicast table. Simpler devices that cannot filter specific addresses might have to enable all multicast when any multicast is requested. The driver indicates multicast capability by setting IFF_MULTICAST in dev->flags; if not, the kernel knows the device either receives none or all multicast by nature [15]. In summary, `ndo_set_rx_mode` is responsible for ensuring the hardware's packet filters (unicast, multicast, broadcast) match the current configuration requested by the network stack [46].

- **ndo_set_mac_address** – Allows changing the device's physical address (MAC). If supported by hardware (most NICs allow it), the driver implements this to program the new address into the device's registers and update `dev->dev_addr`. The kernel ensures the address passed is valid (e.g., not multicast) before calling this. For instance, on Intel e1000, this would involve writing to the Receive Address Register 0 (RAL0/RAH0) with the new MAC [47] [48].

- **ndo_change_mtu** – If the NIC supports a configurable MTU (e.g., jumbo frames), the driver implements this to reconfigure the device with the new MTU. This might involve allocating bigger receive buffers or programming a register. If an MTU is requested that the hardware cannot support, the driver should reject it with an error. If not implemented, the kernel assumes the MTU is fixed (aside from the standard 1500->9000 range possibly allowed by core for Ethernet if driver sets `dev->max_mtu`).

- **ndo_do_ioctl** – Legacy hook to handle any device-specific ioctl calls (via `socket(AF_INET, SOCK_DGRAM,0)` interface ioctls). This is rarely needed in modern drivers because most control is done via netlink and ethtool. Historically used for things like setting media (e.g., forcing 100Mb/1Gb) before ethtool existed, or for custom ioctls. In 6.x, you'll often leave this NULL unless implementing something unusual.

These are the most common callbacks. Additional ones exist (for advanced features): e.g. `.ndo_validate_addr` (to validate a manually set MAC), `.ndo_set_tx_maxrate` (to rate-limit egress per queue), `.ndo_bpf` (for attaching XDP BPF programs), etc. For a basic Ethernet driver, implementing

open, stop, start_xmit, tx_timeout, set_rx_mode, get_stats64, set_mac (optional), change_mtu (optional), and maybe ndo_do_ioctl is sufficient. The device's `netdev_ops` structure would be filled accordingly, as seen in an Intel driver example [9]:

```
static const struct net_device_ops igb_netdev_ops = {
    .ndo_open            = igb_open,
    .ndo_stop            = igb_close,
    .ndo_start_xmit      = igb_xmit_frame,
    .ndo_get_stats64     = igb_get_stats64,
    .ndo_set_rx_mode     = igb_set_rx_mode,
    .ndo_set_mac_address = igb_set_mac,
    .ndo_change_mtu      = igb_change_mtu,
    .ndo_do_ioctl        = igb_ioctl,
    .ndo_tx_timeout      = igb_tx_timeout
    // ... (others if needed)
};
```

The net_device_ops is registered by assigning `netdev->netdev_ops = &igb_netdev_ops` in the probe function [49] [50]. With these hooks in place, the kernel will call into the driver to perform operations as users configure the network interface or as packets flow through.

Opening and Closing the Interface

Bringing the interface up (e.g., `ifconfig eth0 up` or `ip link set eth0 up`) will trigger the kernel to invoke the driver's `.ndo_open`. Under the hood, this goes through the generic code that sets the IFF_UP flag and calls the driver. In response, the driver should prepare the hardware: allocate descriptor rings and buffers, initialize DMA settings, enable interrupts and NAPI polling, and start the device operation. For example, if our driver manages a PCI NIC, in ndo_open we would:

- Enable the RX and TX rings: allocate a block of memory for each ring if not already (using `dma_alloc_coherent` for descriptors), and allocate packet buffers for RX descriptors (using `netdev_alloc_skb` for each, or page fragments). Then, write the ring base addresses and lengths into the device registers, and set the initial Head and Tail indices. (On Intel NICs, for instance, the driver writes the physical address of the descriptor list to RDBAL/RDBAH for RX and TDBAL/TDBAH for TX, and the size to RDLEN/TDLEN [51] [52]. It also ensures the head and tail registers are initialized to 0 [53].)
- Configure device settings: program the NIC's receive control register (promiscuity, multicast filter, buffer sizes) and transmit control register. For example, set the "enable" bits in those control regs (e.g., `TCTL.EN` and `RCTL.EN` on Intel hardware) [54] [55], enable features like automatic padding of short frames (TCTL.PSP) [54], or disable loopback mode, etc., as appropriate.
- Enable interrupts: if using MSI/MSI-X, set those up (often done in probe, but enabling happens here). Register the ISR with `request_irq()`. The device might have an "interrupt mask" register – driver should unmask relevant interrupt events (e.g. link change, RX complete, TX complete) by writing to it [56]. With NAPI, many drivers initially only enable RX-related interrupts and possibly link-change, and not TX interrupts (to reduce overhead) [56].

- Start the NAPI polling: call `napi_enable()` for each NAPI context now that the device is about to run (netif_napi_add is done earlier, but NAPI is typically enabled during open) [57] [58].
- Indicate the link state: if the NIC has an auto-negotiation process, the driver might initially set carrier off and then later turn it on when link is achieved. Some drivers query the PHY status here and call `netif_carrier_on(dev)` if link is already up (or leave it off and let an interrupt update it).
- Finally, enable traffic: invoke `netif_start_queue(dev)` to allow the kernel to start queuing packets for transmission [59] [60]. This marks the device as ready to accept outgoing packets by clearing the software queue stop flag. At this point, the device is "up" and network stack can transmit through it.

Bringing the interface down will call `.ndo_stop`, which should do roughly the opposite:

- Stop the transmit queue (`netif_stop_queue(dev)`) to block new transmissions from upper layers [60].
- Disable NAPI: call `napi_disable()` (ensuring no poll is currently running) and `netif_napi_del()` if removing it completely.
- Quiesce the hardware: turn off RX/TX units (e.g., clear RCTL.EN/TCTL.EN bits or device-specific enable bits). Possibly issue a software reset to the NIC if needed to stop DMA.
- Mask and disable interrupts, and free the IRQ (`free_irq()`). If MSI was enabled, also disable MSI/MSI-X (via `pci_disable_msi` or `pci_disable_msix`) after freeing IRQ [61].
- Reclaim pending buffers: clean out the TX ring, free any skbs that were not yet freed (in case of device shut down mid-transfer), and unmap their DMA. Reclaim RX buffers – since interface is going down, typically we free all RX skbs or memory buffers.
- Free descriptor rings if allocated here (some drivers allocate rings once at probe and keep them until remove; others allocate at open and free at stop to save memory when interface is down). Free any other resources like DMA pools.
- Other cleanup: cancel any driver timers or workqueues (e.g., if the driver scheduled a watchdog work to check link, stop it).
- Mark link state as down (netif_carrier_off) if not already, just for consistency.

After ndo_stop, the device should be inert: no interrupts, no DMA, and no references to any active skbs. The interface can be unregistered or re-opened later. Notably, the kernel might call ndo_stop if the module is being removed or if a fatal error occurred. The driver must handle being called even if ndo_open failed halfway (so it should check what's allocated before freeing). Proper open/close sequences ensure resource leaks are avoided.

ifconfig, ip, and sysfs integration: Calling `ifconfig` or `ip link` ultimately triggers these open/stop calls via the kernel's netlink or ioctl handlers. For instance, `ifconfig eth0 up` sets the IFF_UP flag and calls ndo_open [62]. Similarly, setting the interface flags (PROMISC, ALLMULTI) will call ndo_set_rx_mode. Changing MTU via `ip link set mtu X` calls ndo_change_mtu. Setting the MAC via `ip link set address <addr>` calls ndo_set_mac_address. These user commands use the netlink API (rtnetlink) in modern Linux, which is translated by the kernel into the driver calls. Legacy ifconfig uses ioctl `SIOCSIFFLAGS`, `SIOCSIFADDR`, etc., but the end effect is the same. The driver doesn't interact with ifconfig/ip directly; it just implements the hooks.

The interface's presence in `/sys/class/net/` provides a sysfs view: for example, `/sys/class/net/eth0/address` shows the MAC, `/sys/class/net/eth0/operstate` shows "up" or "down", `/sys/`

`class/net/eth0/statistics/` contains files for each stat counter (rx_packets, tx_bytes, etc.). The kernel handles these, drawing from the net_device stats and state. The driver's job is mainly to update stats and state correctly so that these reflect reality.

Also, when the interface is registered, Linux creates a symlink from the device's sysfs entry to the underlying hardware device (e.g., a PCI device). This is done via `SET_NETDEV_DEV(netdev, &pdev->dev)` in the PCI probe, associating the netdev with its parent device in sysfs [63]. That way, `ls -l /sys/class/net/eth0/device` points to the PCI bus device (helpful for identifying which NIC corresponds to which interface). Conversely, `pci_set_drvdata` links the netdev to the pci_dev's driver data [64] so the driver can retrieve its net_device from the PCI device context easily.

Packet Transmission Path (Socket to Wire)

When an application sends data through a socket (say a UDP packet or an HTTP request over TCP), the data travels through the kernel's network stack and eventually reaches the network driver's transmit function. Here's an overview of the packet transmit path in modern Linux:

1. **Socket Layer to Network Stack:** The application write triggers the transport layer (TCP/UDP) to create a packet. This includes adding transport headers (TCP/UDP) and handing the packet to the IP layer, which adds an IP header (and does routing). Finally, the packet reaches the link layer (Ethernet). At each step, the kernel builds the packet in an `sk_buff` (socket buffer). By the time it's ready to send on a given interface, the skb contains the complete frame: Ethernet header (destination MAC, source MAC, ethertype), followed by IP header, TCP/UDP header, and payload. The skb also has meta-info like the protocol type, checksum status, etc.

2. **Queuing Disciplines:** The packet is passed to the Traffic Control (QoS) layer. Unless configured otherwise, it goes into the default queue (pfifo_fast or fq_codel by default in newer kernels). This layer can reorder or drop packets based on policy. Ultimately, it invokes `dev_queue_xmit(skb)` which hands the skb to the device's queue. The net_device has one or more software queues (`struct netdev_queue`). If the queue is stopped (e.g., driver previously called netif_stop_queue), the packet will be queued in the Qdisc and not given to the driver yet. If the queue is awake, the skb is dequeued and the driver's `.ndo_start_xmit` is called to actually transmit it. Note that on SMP, multiple CPUs may be handling different flows: the Qdisc layer ensures only one CPU at a time calls the driver xmit for a given queue (by a spinlock or lockless algorithm depending on Qdisc). For multiqueue devices, flows get hashed to different queues (by skb_get_tx_queue or XPS settings), and each queue can be processed in parallel on different CPUs.

3. **Driver Start_xmit (Hardware Submit):** As detailed earlier, the driver will map the skb data for DMA, fill a descriptor, and notify the NIC hardware. For example, with an Intel NIC, it sets up the descriptor with the buffer's address and length and then writes to the TDT (Transmit Descriptor Tail) register the index of the new tail [31] [65]. The NIC sees that tail moved and fetches the new descriptor(s). The driver might also take this opportunity to update some stats (e.g., `priv->stats.tx_bytes += skb->len` and `tx_packets++`) and set a timestamp for watchdog (`dev_trans_start(dev)` is updated internally to current jiffies whenever a packet is transmitted).

4. **NIC DMA and Transmission:** The NIC, as bus master, reads the descriptor, obtains the packet data via DMA from host memory, and sends it out on the wire (serializing into Ethernet frames, including preamble, CRC, etc.). This happens asynchronously to the CPU. The NIC will typically set a "descriptor done" bit or increment a head pointer when the transmission is completed. It may also generate an interrupt to notify completion, though many drivers disable per-packet TX interrupts for efficiency, relying instead on either polling or batch interrupts. For example, the Intel NIC has a setting to interrupt when the TX ring is half empty or on a timer, rather than every packet.

5. **Transmit Completion Handling:** Eventually, the driver needs to reclaim the resources for sent packets. This typically occurs either in the interrupt handler or NAPI poll. The NIC might raise a TX complete interrupt or, if using NAPI for both RX and TX, the driver might check the TX ring for completed descriptors during the NAPI poll cycle. In any case, the driver finds descriptors that the NIC has finished with (often by comparing the NIC's "head" index register with a stored index of last cleaned descriptor). For each completed descriptor, the driver will free the associated skb and unmap its DMA. After freeing, it can reclaim that descriptor slot for future packets. If the driver had previously stopped the transmit queue due to ring full, this is where it would call `netif_wake_queue(dev)` to resume the queue [33]. For instance, once a certain number of descriptors become free, the driver clears the stop condition. The networking core will then allow new packets to be dequeued to the driver.

6. **Error Handling:** If the NIC encountered an error sending a packet (e.g., excessive collisions in half-duplex, or FIFO underrun), it might set an error status in the descriptor or raise a different interrupt. The driver should handle these by updating `tx_errors` or more specific counters (e.g., `collisions` if applicable) [41]. The skb should still be freed. If the error is severe (like the transmitter hung), the driver might invoke a reset and rely on the watchdog as well.

From the perspective of the rest of the system, the packet is "out" once the driver's xmit returns successfully. The skb's lifecycle ends when the driver frees it on TX completion. Upper layers may be notified of completion for certain protocols (e.g., TCP will infer that a packet was sent when ACKs return, not directly from TX completion). But one important thing the driver does on TX completion is potentially notify the stack if the device was stalled: i.e., on `netif_wake_queue()` the stack will immediately try to send any queued packets in the qdisc. Modern kernels handle this efficiently with per-CPU NAPI context or softirqs.

Concurrency considerations: The network stack can call `.ndo_start_xmit` from multiple CPUs for different queues (on multiqueue devices). For single-queue devices, by default a qdisc like fq_codel will still only invoke one at a time. However, an SMP machine could theoretically call xmit concurrently if a lock isn't held – to guard, older drivers used a spinlock around the TX ring (or set `dev->xmit_lock` / `dev->flags` with IFF_XMIT_BUSY in older kernels). In Linux 6.x, the core network code serializes calls to ndo_start_xmit per device queue, so many simple drivers don't need their own TX lock for a single queue. For multiqueue, typically each queue has its own lock or uses lockless rings with atomic operations. Our example driver can likely get away without additional locking on TX if it uses proper memory barriers for ring index updates (the kernel's netdev infrastructure might set `dev->priv_flags |= IFF_XMIT_DST_RELEASE` or others to indicate lockless operation, but that's an advanced detail). The main point is the driver must be careful updating ring indices and descriptors in memory that may be accessed by the device at the same time – usually accomplished via proper ordering (e.g., write descriptor contents then write tail pointer register as the last step, possibly using `wmb()` memory barrier to ensure descriptors are visible before the device DMAs).

Packet Reception Path (Wire to Socket)

Receiving data is more complex than transmission because the driver must coordinate with the NIC to hand off buffers for incoming packets and process them, often under high throughput. Here's the general receive workflow:

1. **Ring Buffer Setup:** During interface open (ndo_open), the driver allocates a set of buffers for the NIC to DMA incoming packets into. Typically, a ring of receive descriptors is created in DMA-coherent memory, and each descriptor contains a pointer to a buffer (and perhaps some control fields). For example, our driver might allocate 256 receive buffers, each ~2048 bytes (enough for standard MTU). It then maps each buffer's address to a DMA address and places that into the descriptor, setting an "ownership" flag to indicate to NIC that the descriptor is ready. Finally, it writes the ring base and size to NIC registers (RDBAL, RDBAH, RDLEN) and sets the NIC's "Receive Descriptor Tail" (RDT) register to the index of the last descriptor plus one (which tells the NIC the range of descriptors it can use) [66] [67]. The NIC maintains a "head" pointer which it advances as it fills descriptors with received packets [68] [69]. Initially, head == tail (meaning ring is empty but ready).

2. **Packet Arrival and DMA:** When a packet arrives on the wire, the NIC's hardware parses the Ethernet frame (checking CRC, etc.) and, upon determining the packet is for this interface (or in promiscuous mode, or it's broadcast/multicast that passes filter), it will DMA the frame into the next available buffer in the ring. The NIC then writes status and length information into the descriptor (or into a status array) indicating that descriptor now holds a packet of X bytes. It advances the "head" index to the next descriptor (marking one packet received). If the NIC supports interrupt moderation or NAPI, it may hold off interrupting immediately – it might wait for a batch of packets or a timer. Otherwise, it will trigger an interrupt to the CPU to signal a packet is ready.

3. **Interrupt Handling (Scheduling NAPI):** When the NIC triggers an interrupt for receive, the driver's ISR (interrupt service routine) runs. In a non-NAPI driver, the ISR would immediately begin processing packets (which is costly to do at interrupt context). In modern drivers with NAPI, the ISR's job is typically to *schedule a softirq-based poll* and disable further interrupts from that NIC. The driver will call `napi_schedule_irqoff(&napi_struct)` (or `napi_schedule()` after having already masked the device interrupt) [70] [71]. This flags the NAPI context as ready to run. It also writes to the NIC to stop additional RX interrupts – since NAPI will poll for remaining packets, we don't want an interrupt for each one. For example, on Intel NIC you'd mask the RX interrupt bit in the IMS register. The ISR then returns, leaving the heavy lifting to be done in softirq context. (If NAPI is not used, the ISR would loop through receive descriptors directly, which can saturate the CPU with interrupts under load – the original problem NAPI solves [72].)

4. **NAPI Polling (Receive Processing):** Shortly after `napi_schedule()`, the network subsystem will invoke the driver's registered poll function, `ndo_poll` (associated with the `struct napi_struct`). This happens in the context of a software interrupt (NET_RX_SOFTIRQ), often handled by the `ksoftirqd` thread if packets are continuous [73]. The poll function is handed a `budget` (max number of packets it can process in one go). Our driver's poll function will iterate over the receive ring, starting from the last "cleaned" index up to where the NIC's head pointer is (i.e., all descriptors that have been filled by the NIC and not yet processed by software). For each packet descriptor:

5. It **retrieves the packet length** and status from the descriptor. Suppose the NIC wrote that descriptor i has `length = N` bytes and status "done".
6. It then obtains the corresponding buffer. Often the driver stored an `skb` for each descriptor ahead of time. Some drivers pre-allocate sk_buffs and attach the DMA buffer to skb->data; others allocate plain memory buffers and only construct an skb now. A common approach is: allocate skbs upfront with `netdev_alloc_skb()`, which gives an skb with a DMA-able data area (GFP_ATOMIC, from the NIC's memory region). Put the skb pointers in an array indexed by descriptor. So now, to process descriptor i, the driver takes `skb = rx_skb_ring[i]`.
7. Set the skb's `tail` and `len` to indicate it contains the N-byte packet (e.g., `skb_put(skb, N)` to adjust length). If the NIC included the Ethernet FCS (Frame Check Sequence) in the buffer, the driver might need to trim it off (some hardware do, some automatically strip the CRC – typically modern Ethernet NICs strip the FCS by default, controlled by a bit in RCTL, e.g., RCTL.SECRC). The driver also checks if the packet is error-free. If NIC indicated a CRC error or length error in the descriptor status, the driver should increment `rx_errors` and `rx_crc_errors` (for instance) and drop this packet (don't pass it up). Usually such packets are rare unless cable issues.
8. For a good packet, the driver sets up the skb's metadata: call `skb->protocol = eth_type_trans(skb, dev)` which parses the Ethernet header and sets `skb->protocol` (e.g., ETH_P_IP) and also strips the Ethernet header from the skb data (so the payload starts at network layer) [74] [75]. It also determines if the packet is unicast, multicast, or broadcast and sets `skb->pkt_type` accordingly (though if the NIC filters properly, most received will be unicast to us or broadcasts).
9. Handle checksum offload: Many NICs can verify checksums for TCP/UDP. If the NIC indicates the packet's checksum is OK (or that it did hardware checksum), the driver can set `skb->ip_summed = CHECKSUM_UNNECESSARY` [76], meaning upper layers don't need to recompute the checksum. If the NIC provides partial checksum info, there are other flags (e.g., CHECKSUM_COMPLETE). If no offload, `skb->ip_summed = CHECKSUM_NONE` (kernel will compute later). In our example, since snull is a virtual driver, they set CHECKSUM_UNNECESSARY because there's no corruption chance in memory transfer [77].
10. Finally, the driver passes the packet to the kernel networking stack. In NAPI context, this is done with `netif_receive_skb(skb)` (or the newer `napi_gro_receive()` if supporting GRO – generic receive offload). `netif_receive_skb` will take the packet through the protocol dispatch: it finds the appropriate handler for skb->protocol (IP, ARP, IPv6, etc.), then passes it up to the socket or kernel consumer. This function returns quickly (it queues packet to upper layers, maybe performs GRO aggregation if possible).
11. The driver then **replenishes** the RX buffer: that is, allocate a new skb or buffer to replace the one just handed off. It cannot reuse the same buffer if we gave ownership to the networking stack (in our approach, we handed the skb upward). So allocate a fresh skb (`new_skb = netdev_alloc_skb(dev, buffer_length)`). If successful, map its data for DMA (or use build_skb with a page fragment), put the DMA address into descriptor i, and store `rx_skb_ring[i] = new_skb`. Then clear the descriptor status and mark it as owned by NIC. If allocation fails (out of memory), the driver will not refill now – leaving that slot empty. It should record `rx_dropped++` and hope to refill later; in the meantime, the NIC ring has effectively shrunk by one.
12. Increase the count of processed packets. If the count reaches the budget (say 64), the poll function should stop even if there are more packets, and return the number processed, so that the kernel will reschedule it again (to avoid monopolizing CPU).

This loop continues until either we hit the budget or the NIC's head index (no more new packets). If all packets were processed (head == our index), the poll is done. The driver calls `napi_complete_done(&napi, processed_count)` (or `napi_complete()` in older API) to mark the NAPI cycle finished [70] [78]. At this point, the driver should **re-enable interrupts** for receive on the NIC, since we're going out of polling mode. Typically, napi_complete will call the driver's `enable_irq` or unmask function via a callback, or the driver does it explicitly after napi_complete. For example, in Intel e1000, after napi complete you write to IMS register to unmask RX interrupt again [79] [78]. If there are leftover packets (i.e., we hit the budget and NIC still has more data), we do not complete – we will return `processed_count` which equals budget, and the NAPI infrastructure will keep the poll active and call us again immediately. This is the **NAPI mechanism**: process packets in batches up to a max count, and repeat if necessary, otherwise exit and resume interrupts. It's efficient because under high load, interrupts stay disabled and we just poll; under low load, we handle a packet or few and quickly re-enable interrupts for responsiveness.

Each iteration also updates stats: e.g. `rx_packets++`, `rx_bytes += skb->len`. If any errors, update those counters.

1. **Upper Layer Delivery:** Once `netif_receive_skb(skb)` is called, the packet is handed to the network stack. If it's IP, it goes to IP input, then maybe to a TCP socket receive queue, etc. Eventually the user-space application `recv()` will copy data from the socket buffer. All that is outside the driver's scope. The driver's role in receiving ended when it gave the skb to `netif_receive_skb`. However, the driver's efficiency and correctness in this stage are critical – any delay or error can drop packets.

2. **Repeat:** The NIC will keep filling buffers as packets come. The driver, via NAPI polling, will continuously harvest them. When no packets are coming, the poll won't be scheduled, and the NIC interrupts are armed to wake it when something arrives (with some possible delay due to interrupt moderation).

Receive Interrupt Mitigation (NAPI): We've implicitly described NAPI (New API) polling above. To summarize its benefits: traditionally, each incoming packet would trigger an interrupt and the driver would handle one packet per interrupt, leading to "livestock" problem under high PPS (packets-per-second). NAPI addresses this by **turning off interrupts while processing a batch of packets**, and processing as many as available (up to a limit) in one go [72]. The steps for NAPI-enabled drivers are:

- In the driver's probe or open, initialize NAPI via `netif_napi_add(dev, &priv->napi, my_poll, weight)` [80] [81]. The weight is the budget max (commonly 64 as a default) [80].
- In the ISR, call `napi_schedule()` and disable device IRQs [82] [71].
- In `my_poll(struct napi_struct *napi, int budget)`, do the packet processing loop as above. If packets < budget and all done, call `napi_complete_done()` and re-enable IRQs [70] [78]. If `processed == budget`, stop without completing; the kernel will keep polling.
- Ensure that when interface is downed or driver removed, you call `netif_napi_del()`.

NAPI drastically reduces interrupt load – instead of an interrupt per packet, you might get one interrupt for, say, 64 packets (or for a burst arriving close together). If packets keep arriving, eventually the NIC stops interrupting and relies on the polling. Only when the traffic falls off and the poll cycle completes do

interrupts get re-armed, so the next new packet will generate an interrupt to start polling again [83] [84]. This hybrid approach gives low latency under light load (each packet triggers a prompt interrupt & processing) and high throughput under heavy load (packets processed in batches, minimizing context switches).

Most Linux drivers in 6.x implement NAPI (it's essentially mandatory for high-speed interfaces). The older interface (with `dev->poll` pointer and `dev->weight` in net_device) has been replaced by the separate napi_struct as described. One device can even have multiple napi_structs (e.g. one per queue for multiqueue NICs) for parallel receive on multiple CPUs [85] [86].

For completeness, if NAPI were *not* used, the driver's ISR would do: disable further interrupts, loop receiving packets (calling `netif_rx(skb)` for each to queue them to backlog), then re-enable interrupts. This "bottom half" processing was done via `netif_rx` which queues to a per-CPU backlog and raises NET_RX_SOFTIRQ. It's similar in effect to NAPI but less efficient because every packet still caused an interrupt and overhead. NAPI integrates the idea of polling to coalesce this work.

Interrupt Handling and Softirqs

Network drivers utilize Linux's **softirq** mechanism for deferring work outside of hard interrupt context. The primary softirq for networking is NET_RX_SOFTIRQ, which handles packet receive processing as we saw. When the NIC interrupts, the driver's top-half (hard IRQ handler) should execute quickly – typically just acknowledging the interrupt and scheduling NAPI [71] [87]. The heavy packet processing is then done in the softirq (which may run on the same CPU shortly after the IRQ, or be punted to ksoftirqd). This design prevents long periods with interrupts disabled and improves scalability.

Additionally, there is a NET_TX_SOFTIRQ for transmission – though drivers usually do not explicitly deal with it. The NET_TX_SOFTIRQ is raised by the networking core to handle dequeuing packets from Qdisc to call the driver's xmit. In old kernels, if a driver returns NETDEV_TX_BUSY or if the qdisc has more packets, the softirq ensures the dev->xmit is called in a loop. The distinction between process context and softirq for transmit is subtle: if you send packets from a user process, the first packet may be sent in that process context, but if the driver or qdisc defers any work, NET_TX_SOFTIRQ can handle further transmission asynchronously. In general, as a driver writer, one just needs to be aware that `.ndo_start_xmit` might be called in softirq context (so sleeping is not allowed in xmit, and you must use spinlocks if needed). But since xmit is supposed to be non-blocking, this is fine.

Tasklets were once used for network drivers (a form of bottom-half similar to softirq). Historically, drivers without NAPI might schedule a tasklet in the ISR to handle packet receive outside the hardirq. With NAPI's introduction, tasklets are largely obsolete for NIC drivers – NAPI does the job in a controlled way (and tasklets themselves are built on softirqs). So, in modern code, you won't see drivers scheduling tasklets for RX; they rely on NAPI. You might still see tasklets for other purposes (some WLAN drivers use tasklets for some offload handling, etc.), but not typically for Ethernet RX/TX.

Softirq and thread context: If the system is overwhelmed, the ksoftirqd kernel threads will handle the NET_RX_SOFTIRQ processing in process context (this prevents live-lock if packets come in faster than they can be processed; the softirqs are then queued and handled by a schedulable thread to yield CPU

occasionally). As a driver developer, just ensure your poll routine is efficient and returns when it should (after `budget` or when done) so that ksoftirqd can manage load.

Multicast and Promiscuous Mode

Ethernet supports multicast addresses – frames destined to a group of stations (not all, unlike broadcast). A NIC typically has a limited ability to filter multicast addresses. The networking stack keeps track of which multicast groups each interface is subscribed to (e.g., if any socket joined an IPv4 multicast address, the corresponding Ethernet MAC is added to the device's filter list). The driver doesn't need to know why (that's handled by IGMP/MLD in the network layer); it just gets an updated list of MAC addresses to accept. The `.ndo_set_rx_mode` method, as discussed, is the hook for this. Let's outline driver behavior:

- If `dev->flags & IFF_PROMISC` is set, the driver should enable promiscuous mode on the NIC (receive all packets, regardless of dest MAC). This mode is used by packet sniffers or bridge devices. The NIC usually has a register bit for this (e.g., set RCTL.UPE and RCTL.MPE on Intel to accept all uni/multicast). The driver should also program the hardware to not filter anything. Promiscuous mode can produce a lot of unwanted traffic, but the kernel handles dropping what isn't needed at higher layers.

- Else if `IFF_ALLMULTI` is set (and not promisc), the driver enables reception of all multicast frames (but can still filter unicast by MAC). This is a fallback when the multicast list is too large for hardware filters. For example, if a NIC can only filter 64 MACs and the system joins 100 multicast groups, the kernel will set IFF_ALLMULTI and expect the driver to just accept all multicast frames (the assumption is that multicast traffic volume is usually low) [88] [43] .

- If neither promisc nor allmulti, the driver should program the multicast filter to accept the specific addresses in `dev->mc_list`. The `mc_list` is a list of `dev_mc_list` entries (MAC addresses) and `dev->mc_count` the count [89] . On update, the driver can either use perfect filtering (if NIC has exact-match slots) or a hash filter (many NICs have a hash filter for multicast). For example, the Intel e1000 has a 256-bit hash table (Multicast Table Array – MTA). The driver computes a hash of each MAC and sets the corresponding bit in MTA for each address to accept [90] . NICs that have a set number of perfect filter slots might use those for the first few addresses, and if too many, resort to allmulti. Simpler devices might not support filtering at all – in which case the driver would have cleared IFF_MULTICAST flag at init (so kernel knows it can't do selective filtering) [91] [92] , and then the device will either always receive all multicast or none. Typically, those devices would choose all or none by setting IFF_ALLMULTI when needed.

- The `.ndo_set_rx_mode` should also handle toggling the hardware's broadcast filter, if any. But normally, all NICs always receive broadcast (Ethernet broadcast MAC ff:ff:ff:ff:ff:ff) by default, so not much to do – broadcast frames are counted separately (`stats.multicast` counts multicast frames received [93] , broadcast are counted in `rx_packets` but not in `multicast`).

It's important for the driver to implement multicast filtering correctly for performance – on a busy LAN with many multicast streams, you don't want the CPU to receive everything if it only needs a few groups. But on the other hand, if a device has no filtering, it's acceptable (the kernel just expects that and will set flags accordingly). Setting IFF_MULTICAST in dev->flags advertises capability; if not set, kernel will avoid using

multicast on that dev (only use broadcast or fallback to promisc for multicast). Most modern NICs support multicast filtering, so drivers set IFF_MULTICAST [15].

Memory and DMA Considerations

Network drivers are responsible for managing DMA mappings and memory buffers efficiently:

- **Descriptor rings** are typically allocated with `dma_alloc_coherent()` (or pci_alloc_consistent()) so that the device and CPU see the descriptor memory synchronously (coherent memory) [94]. These ring allocations should be aligned as required by the NIC (often 16-byte or 128-byte alignment) [95]. The size of the ring is a multiple of the descriptor size; drivers often choose ring lengths that are powers of two for convenience (though not strictly required by hardware, it can simplify index wrap calculations).

- **Packet buffers** (for RX) can be allocated in various ways. The simplest is to allocate an `sk_buff` for each incoming packet and use its data area for DMA. The `netdev_alloc_skb(dev, length)` helper allocates an skb with a data buffer that is appropriately aligned for DMA and on the node/zone memory preferred by the device. Many drivers allocate skbs of size slightly larger than MTU (to accommodate headroom or append room). For example, for a 1500 MTU Ethernet, allocate 1536 or 2048 bytes to have a power-of-two size. The driver maps the skb->data for DMA (using `dma_map_single` with `DMA_FROMDEVICE`) and stores that DMA address in the RX descriptor. On packet receive, as described, the skb is used and then a new one allocated.

Some high-performance drivers use **page recycling**: they allocate pages or part of a page for each buffer (like 2k out of a 4k page) and try to reuse them to reduce allocations. Linux has infrastructure like the Page Pool API (new in recent kernels) to manage recycling of RX buffers efficiently, especially with XDP (which might drop packets early, allowing immediate reuse of buffers). But for our walkthrough, per-packet skb allocation is fine.

- **DMA Mapping**: For TX, the driver uses `dma_map_single` or `dma_map_page` on the skb's data (and fragments if any) before handing to device. After the packet is sent and the interrupt signals completion, the driver must call `dma_unmap_single` to release the mapping (unless the device is coherent and mapping was not needed due to consistent memory – but generally we treat data buffers as streaming DMA). The mapping APIs handle things like bounce buffering or addressing limitations. The driver should set the DMA mask appropriately in probe (e.g., `dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(64))` if the device can do 64-bit DMA) [96] [97]. If a device can only do 32-bit DMA, the driver sets a 32-bit mask, and the kernel will ensure memory allocated is below 4GB (or bounce buffer if needed). This is indicated by the NETIF_F_HIGHDMA flag historically – drivers set NETIF_F_HIGHDMA to tell the core it can handle high memory DMA [98] (on modern PCI, almost always true).

Coherent vs streaming: descriptor rings are coherent (consistent memory), meaning the CPU can read/write them without explicit sync and the device can access anytime. Packet data buffers are streaming: mapped just long enough for device to use, then unmapped.

- **Memory barriers:** On some architectures, writing to memory (like descriptors) might not immediately be visible to the device due to caching or posting. Drivers often use `wmb()` (write

memory barrier) before telling the NIC a new descriptor is ready (like before writing the tail register). Similarly, after device writes to descriptors (for RX), the driver may need `rmb()` (read memory barrier) before reading the data to ensure it sees all updated fields. The x86 architecture is strongly ordered for writes to WC memory? But generally, drivers use the appropriate barriers provided by the kernel primitives or iommu mapping functions.

- **SKB Data Alignment:** The network stack prefers the IP header to be 2-byte aligned in memory (some arch require this for performance). `netdev_alloc_skb` ensures 2-byte alignment by default. Some drivers further ensure 4 or 8 byte alignment of payload for DMA reasons. If an incoming packet size is odd, the driver might realign or copy – but most modern NICs can DMA to arbitrary addresses as long as buffer is contiguous.

- **Buffer Size and Truncation:** If a packet larger than expected arrives (e.g. baby jumbo frame) and the buffer isn't large enough, NIC may truncate it and set an error. The driver then counts `rx_length_errors`. Usually, if MTU is properly set and we allocate MTU-sized buffers, this shouldn't happen except for malicious oversize frames or runts.

- **Packet Splitting:** Some advanced NICs can split packets (e.g., header in one buffer, payload in another for cache efficiency) or support scatter-Gather on RX. Our simple driver likely won't use that. We assume one buffer per packet.

- **Offloads and Features:** If our device supports offloads like checksum or TSO (TCP segmentation offload), the driver should set the feature flags in `dev->features`. For example, NETIF_F_HW_CSUM for checksum offload [99] [100], NETIF_F_TSO for TCP segmentation, NETIF_F_SG for scatter-gather (ability to send from fragmented skb without linearizing). The kernel will then give skbs with `ip_summed = CHECKSUM_PARTIAL` for hardware checksumming – the driver needs to fill descriptor fields so NIC will compute the checksum (commonly by setting a context descriptor or flags indicating where in the packet the checksums start – details in device datasheet). For TSO, the kernel will pass a large skb (with gso_size set and segments described) and expects the NIC to segment. The driver again prepares special descriptors to instruct NIC. Implementing these is beyond a basic driver walkthrough, but one should be aware of them. If the driver does not set these feature flags, the kernel will perform these tasks in software (for example, if NETIF_F_HW_CSUM is not set, the IP stack will compute the checksum in software and mark the skb as CHECKSUM_NONE, so the driver just sends it normally) [101] [100]. Our hypothetical driver might simply disable offloads at first (clear any default flags like NETIF_F_CSUM) to keep things simple, or support a basic checksum offload if the hardware mandates it.

- **Jumbo Frames:** If supporting MTUs > 1500, the driver must allocate larger buffers (e.g., 4KB or 9KB) and possibly use multiple descriptors per packet if hardware doesn't support huge single descriptors. Many NICs handle jumbo in one descriptor assuming the buffer is large enough. The driver sets `dev->max_mtu` accordingly and handles mtu change by reallocating buffers.

Monitoring and Statistics

As previously covered, drivers maintain statistics for management and debugging. Key points:

- The `struct net_device_stats` (or the 64-bit equivalent) fields include: `rx_packets`, `tx_packets` (successful packet counts), `rx_bytes`, `tx_bytes` (octet counts), `rx_errors`, `tx_errors` (various errors), `rx_dropped`, `tx_dropped` (packets discarded due to no resources or other local reason), `multicast` (number of RX multicast frames) [41] [102], `collisions` (Ethernet collisions on transmit) [41], and more specific ones like `rx_length_errors`, `rx_crc_errors`, `rx_fifo_errors`, `rx_missed_errors`, etc., which drivers may use if the hardware provides such counts.

- Many NICs have hardware counters (especially in managed switches or high-end NICs) for things like CRC errors, alignment errors, missed packets (buffer overflow), etc. A driver can query these registers and include them in stats. Simpler drivers accumulate stats in software (e.g., increment rx_errors when dropping a packet due to no skb).

- The `.ndo_get_stats64` method should fill in these stats. The kernel will merge them when reporting to users. In modern kernel, `/proc/net/dev` is populated from these stats, and `ip -s link` shows the same. The driver should ensure that reading stats is thread-safe with respect to updates. This often means using atomic or per-CPU counters. A common technique is `u64_stats_sync`: drivers maintain stats per CPU to avoid locking on every packet, and then in get_stats64 use a seqlock to aggregate them safely. For our educational driver, a simple spinlock around increments or using atomic64_t for counters is fine (performance is not a big concern in an example).

- **ethtool statistics:** The `ethtool -S eth0` command can retrieve extended stats from driver. If we implement `.get_ethtool_stats` and define stat names, we could expose more details (like specific error breakdowns, or NIC internal stats). This is optional.

- **Link state:** Drivers should also handle link up/down events by calling `netif_carrier_on/off`. Many NICs have an interrupt for Link Status Change (LSC). In the ISR, when LSC bit is set, driver queries the PHY or device registers to see if link is up (speed/duplex). If link went down, call `netif_carrier_off(dev)`, which informs the stack (so that e.g. `ifconfig` shows NO-CARRIER and upper-layer protocols know the link is down, stopping retransmits perhaps). If link came up, call `netif_carrier_on(dev)`. Link speed, duplex, auto-negotiation, etc., are usually reported via ethtool (the driver implements `.ndo_get_link_ksettings` to return those). In our driver, we could ignore link state for simplicity (or assume always up if loopback). But a real NIC driver would do this.

- **Errors and debugging:** Many drivers include a debug counter or use kernel's `netdev_warn`/`netdev_dbg` macros to log unusual events (like TX timeout, RX overflow). For example, if the NIC says "RX FIFO overrun", you might increment rx_over_errors and possibly log a warning that the device couldn't keep up (meaning increase ring size or something).

The statistics help the user and system software understand network performance. For example, `rx_dropped` increasing might indicate driver is running out of skbs (potential tuning issue). Collisions (if any) indicate a hub or half-duplex link. `tx_errors` might indicate a malfunction if nonzero regularly.

Advanced Features in Modern NIC Drivers (XDP, Multiqueue, etc.)

Linux 6.x drivers often support features beyond basic send/receive. While these are optional in the scope of a simple driver, it's worth mentioning contextually:

- **Multiqueue:** Most modern NICs have multiple transmit and receive queues, allowing parallel processing on multicore systems. Our driver can declare `dev->num_tx_queues = N` and allocate N ring buffers for TX, similarly N for RX with separate `struct napi_struct` for each. The kernel will then invoke `.ndo_start_xmit` with a queue index (or actually, it'll call the same xmit but we use `skb_get_queue_mapping(skb)` to pick the right ring). We also use `netif_tx_stop_queue(txq)` and `netif_tx_wake_queue(txq)` per queue. Implementing full multiqueue requires careful coordination (including how interrupts are mapped – often one per RX queue, and TX completions are handled on the same CPU that handled that queue's RX for locality). For simplicity, one could stick to a single queue. But being aware: in Linux, the default is usually at least queues equal to number of CPU cores for high-end NICs.

- **RSS (Receive Side Scaling):** This is related – NICs can spread incoming flows across multiple RX queues by hashing packet headers. Each queue has its own interrupt (affinitized to separate CPU). This significantly improves throughput by parallelizing receive processing. A driver enabling RSS will set up multiple RX rings and NAPI instances, and program the NIC's indirection table for hashing. The kernel doesn't need to know much except that you call `netif_napi_add` multiple times and handle each queue separately.

- **XDP (eXpress Data Path):** XDP is a feature where a BPF program can be attached at the driver level to intercept packets at the earliest point (before the kernel allocates an sk_buff). If implementing XDP, the driver needs to allocate packet buffers from a special memory pool (often using `xdp_buff` or `page_pool`) and have code in the RX poll to invoke the BPF program. The BPF program can decide to DROP, TX (redirect) the packet, or pass it up to the stack. XDP is used for high-performance packet filtering/forwarding (millions of packets per second). Supporting XDP in a driver means implementing `.ndo_bpf` to set up the program and modifying the RX loop to handle XDP result. It's quite advanced; for our study, we mention that modern drivers like Intel's have XDP support, but we won't implement it here.

- **TSO, LRO, GRO:** TCP Segmentation Offload (TSO) allows the stack to send one huge skb (e.g. 64KB of data) and NIC will split into MSS-sized segments, saving CPU. Our driver would simply see an skb with `skb_is_gso(skb)` true and a gso_size; it would then break it into descriptors with proper flags (or if NIC requires, create a special context descriptor telling NIC to do segmentation). Checksum offload goes hand-in-hand because for each segmented packet NIC will also do checksums. Large Receive Offload (LRO) is mostly deprecated at driver level in favor of GRO (Generic Receive Offload) which is done in software in the stack. Drivers just provide the ability to coalesce if needed, but usually GRO in networking stack handles it after driver posts packets via `napi_gro_receive`.

- **Energy Efficient Ethernet, Wake-on-LAN, etc.:** Many NICs support WoL (wake on LAN) or EEE (802.3az low power idle). These are controlled via ethtool. A driver implementing them would respond to ethtool commands to enable WoL patterns or set EEE timers. For a teaching driver, we can ignore this.

- **Hardware timestamps:** Some NICs can timestamp packets (for PTP - precise time protocol). If implementing, driver would extract timestamps and use kernel time sync APIs.

In summary, a production driver has many bells and whistles; but the core concepts remain: initialize hardware, manage rings and buffers, implement open/close, transmit, and receive (often with NAPI), and integrate with OS networking features.

Walkthrough Summary

We have covered how a network device driver interacts with the Linux 6.x kernel: from registration and `net_device` setup, through packet transmit and receive flows, to interrupt handling and NAPI polling, and support routines like stats and multicast filtering. We updated the old "snull" example concepts to current kernel APIs (using `net_device_ops`, NAPI, etc.) and touched on new features that didn't exist in the 2.6 era (like XDP or multiqueue). With this foundation, one should be ready to understand or write a simple network driver.

To solidify this understanding, let's outline a practical project: implementing a driver for a real (emulated) NIC.

Project: Implementing a PCIe NIC Driver in QEMU (Intel e1000e)

Problem Statement: Write a Linux kernel driver for the Intel 82540EM/82545EM Gigabit Ethernet card – commonly emulated as `e1000` in QEMU. The goal is to create a minimal but functional driver that can send and receive packets on this NIC when running a Linux VM under QEMU. This exercise will require implementing all the concepts discussed: device initialization, register setup, descriptor ring management, interrupts, and packet processing. We will not use any existing e1000e code; instead, we'll use the device's documentation and reference the QEMU device model to understand hardware behavior.

Key responsibilities and steps:

1. **PCI Device Detection and Setup:** The Intel 82540EM NIC is a PCI device with Vendor ID 0x8086 and Device ID (for 82540EM) 0x100E (and related IDs for variants) [103]. Begin by writing a PCI driver (`struct pci_driver`) that matches this device ID (using `MODULE_DEVICE_TABLE`). Implement the `.probe` function to initialize the device. In probe:
2. Enable the PCI device (e.g., `pci_enable_device_mem(pdev)`) [96] and set DMA addressing capability with `dma_set_mask_and_coherent` (this card is 64-bit capable, so use 64-bit mask) [97] [104].
3. Request and ioremap the device's MMIO registers. The e1000 device's registers are memory-mapped (typically BAR0). Use `pci_request_regions` then `pci_iomap` to get an `void __iomem *hw_addr` pointer.

4. Register the net_device: allocate with `alloc_etherdev(sizeof(struct my_priv))`. Set `SET_NETDEV_DEV(netdev, &pdev->dev)` to associate with PCI device for sysfs [63], and `pci_set_drvdata(pdev, netdev)` to retrieve netdev in remove [64].
5. Fill in `netdev->netdev_ops` with your functions (ndo_open, ndo_stop, ndo_start_xmit, ndo_tx_timeout, etc.) and `netdev->ethtool_ops` if implementing any ethtool hooks (not strictly required to get basic functionality).
6. Read the device's factory MAC address from hardware. On e1000, the NIC's EEPROM contains the MAC. The device regs at offset 0x5400 (RAL) and 0x5404 (RAH) hold the MAC (split into low 32 bits and high 16 bits, with a valid bit) [47] [105]. Alternatively, the e1000 driver uses EEPROM reads via the management interface, but for simplicity, you can try reading those registers after a reset. Set `netdev->dev_addr` to the MAC. If uncertain, you can also generate a random MAC for testing.
7. Initialize driver private data (allocate descriptor rings, etc. but actual memory allocation of rings can wait until ndo_open). For instance, set `priv->pdev = pdev`, `priv->hw_addr = ioaddr` for use by other functions.

8. Finally, register the network device with `register_netdev(netdev)`. If this fails, cleanup. If success, your interface (say `eth1` in the VM) is now known to the system.

9. **Device Initialization (ndo_open):** When the interface is brought up, perform the hardware init:

10. **Reset the NIC:** The e1000 NIC should be reset to a known state. This can be done by writing to the Device Control register (CTRL). For instance, writing CTRL.RST (bit 26) triggers a device reset. After reset, many registers are at defaults (the datasheet notes which ones). You must wait a bit for the reset to complete (polling the CTRL.RST bit to clear).
11. **Configure Receive (RX):** Allocate a contiguous memory block for the RX descriptor ring (e.g. 128 descriptors * 16 bytes each = 2048 bytes, aligned to 16 bytes). Use `dma_alloc_coherent` to get `priv->rx_desc` (virt) and `priv->rx_desc_dma` (dma addr). Program the NIC registers: RDBAL = lower 32 bits of `rx_desc_dma`, RDBAH = upper 32 bits [106] [107], and RDLEN = ring size in bytes (128 * 16 = 2048, which must be 128-byte aligned as required [108]). Initialize the RX descriptor structs: for each descriptor, allocate an skb of, say, 2048 bytes (`priv->rx_buf[i] = netdev_alloc_skb(...)`), map it (pci_map_single) and put the DMA address in descriptor[i]. Set the descriptor status field to 0 (owner = NIC). The e1000 uses a slightly complex descriptor format (64 bits for buffer addr, 16 for length, 8 for csum, 8 for status, etc.), but essentially: buffer address goes in, length can remain 0 until filled by NIC, and you ensure the DD bit (descriptor done) is cleared.
12. Program the Receive Control (RCTL) register: set RCTL.EN = 0 (leave disabled until everything ready), set RCTL.LPE (if supporting long packets, probably not for now), RCTL.BAM = 1 (accept broadcast), RCTL.SECRC = 1 (strip CRC), RCTL.BSIZE to 2048 (there are bits to select buffer size, e.g. BSIZE=1024 or BSIZE=2048 via a combination of bits, the 82540 datasheet details RCTL bits) [66] [67]. Also, if using multiqueue or such, disable for simplicity. We'll accept all unicast to our MAC by default (the MAC is set via RAL0/RAH0), and all broadcast; multicast filtering can be set by RCTL.MPE=1 (to accept all multicasts) or using the MTA, but initially we can enable all multicasts to keep it simple (or none if not needed).
13. **Configure Transmit (TX):** Allocate a DMA-coherent block for TX descriptors (e.g., 128 descriptors * 16 bytes). Program TDBAL/TDBAH with its address [109] [110], and TDLEN = size (e.g., 2048 bytes for 128 descriptors) [111] [112]. Set TX head and tail pointers to 0 (NIC will do it on reset, but writing 0 to

TDH and TDT is good practice) [53] . Also initialize your software tracker for the ring (e.g., priv->tx_index = 0 for next desc to use, priv->clean_index = 0 for next to clean).

14. Program the Transmit Control (TCTL) register: set TCTL.EN=0 (disable until ready), TCTL.PSP=1 (pad short packets) [54] . Also set collision related fields: Collission threshold (CT) = 0x10, Collision distance (if applicable). For gigabit full-duplex, collisions not an issue, but the register still requires some defaults (the Intel Linux driver sets `TCTL_CT=15h, TCTL_COLD=0x40 for gig, 0x200 for 10/100). We can copy typical values from the datasheet or known driver defaults. Also set TIPG (inter-packet gap) registers to recommended values (e.g., 0x0060200A for 1000Mb, found in Intel docs). These magic values can be pulled from the Intel manual or existing driver. They ensure proper spacing between packets.
15. **Interrupt Setup:** The e1000 has an IMS (Interrupt Mask Set) register. Enable relevant interrupts by writing to IMS: definitely RX-related (e.g., IMS bit 7 = RXT0, meaning packet received or RX timer), maybe Link Status Change (bit 2 = LSC) [56] . Optionally TX (bit 0 = TXDW for TX descriptor written back, bit 1 = TXQE for TX queue empty). The Intel driver often enables LSC and RX, and either polls TX or uses TX Interrupt Delay (there's a TIDV register to moderate TX interrupts). We can enable TX interrupts for simplicity so we know when to free TX buffers. Write IMS with bits for LSC, RXO (RX overrun), RXT0.
16. Register the interrupt handler: call `request_irq(pdev->irq, e1000_isr, IRQF_SHARED, "e1000drv", netdev)`. (82540 uses legacy INTx by default in QEMU, and can use MSI if enabled, but to keep it simple use the shared INTx line.) In probe, we should have done `pci_enable_msi` if we wanted MSI – we can skip that for now.
17. Final steps: set the receive address registers for our MAC. There are 16 receive address registers (RAL/RAH pairs). Write RAL0 with the low 32 bits of MAC and RAH0 with high 16 bits and set RAH0.VALID bit to 1 [113] [114] . Also set up the Multicast Table Array (MTA) to 0 (no multicast addresses initially) [90] .
18. Enable the RX and TX units: write RCTL.EN=1 to start receive [115] , and TCTL.EN=1 to start transmit [54] . Now the NIC is actively listening and able to send.
19. Enable NAPI context(s) and then allow the stack to send: call `napi_enable(&priv->napi)` (if NAPI was added) [116] [58] , and `netif_start_queue(netdev)` [59] .
20. Return 0 from ndo_open. The interface is now up.
21. **Packet Transmission (ndo_start_xmit):** When the kernel has a packet to send, it will call your `e1000_start_xmit(skb, netdev)`. In this function:
22. Calculate the index in the TX ring to use (priv->tx_index). Check if the ring is full: one way is to keep a count of used descriptors or check `(tx_index + 1) % ring_size == priv->clean_index` (the last cleaned descriptor index). If full, you should return `NETDEV_TX_BUSY` (and not consume the skb). However, a better approach is to stop the queue ahead of time. We can monitor free slots and if dropping below a threshold, call `netif_stop_queue`.
23. Assuming space available, map the skb data for DMA: `dma_addr = dma_map_single(&pdev->dev, skb->data, skb_headlen(skb), DMA_TO_DEVICE)`. Also, handle fragments: e1000 can do scatter-gather with multiple descriptors or a single descriptor per fragment if using advanced mode. To simplify, you may choose to linearize small skbs (or ensure the stack does not send fragmented skbs by not setting NETIF_F_SG). But let's say we support simple SG: If `skb_shinfo(skb)->nr_frags > 0`, you'd have to use multiple descriptors. The 82540 has two descriptor formats (legacy and extended); the extended can handle a single packet spread over

multiple descriptors by marking only the last one with EOP (end-of-packet). For simplicity, we might just linearize skb if fragments exist: e.g., call `skb_copy_expand` to make it one piece, freeing the old. That's inefficient but acceptable for learning. Alternatively, set `dev->features &= ~NETIF_F_SG` to have kernel do it.

24. Fill the descriptor at tx_index: put the DMA address in lower 64 bits. Set length = skb->len. Set command bits: for legacy descriptors, "EOP" (end of packet) for the last descriptor of this packet (here always true if one descriptor per packet), "IFCS" to instruct NIC to append the FCS (CRC), "RS" (report status) maybe for every Nth packet or last in ring to trigger TX interrupt (we could set RS for the last descriptor of each packet to get timely TX completion interrupts, or be more selective). Also set "DEXT" if using extended mode (but let's assume legacy mode for now). The datasheet has specifics: in legacy mode, lower bits of cmd include EOP, IFCS, RS, and status bits are written back by NIC.
25. Advance `priv->tx_index = (tx_index + 1) % ring_size`. If this new index equals priv->clean_index, it means ring is now full, so call `netif_stop_queue(netdev)` to stop further xmit until some complete.
26. Write the index of the new tail to the TDT register (or rather, write the new TDT value which is equal to the new tx_index, because TDT points *beyond* the last filled descriptor) [31] [65] . On e1000, writing TDT actually gives the index of the first *free* descriptor (so NIC will process up to TDT-1). So if we just queued descriptor N, we write TDT = N+1.
27. Store the skb in a sw array (priv->tx_skb[tx_index]) so we can free it later when that descriptor is completed.
28. Return `NETDEV_TX_OK`.
29. **Interrupt Handler:** Implement `e1000_isr(int irq, void *data)`. This will be invoked when an interrupt occurs (likely shared, so you should check it's really your device's interrupt).
30. Read the Interrupt Cause Register (ICR). On e1000, ICR bits mirror those in IMS that were enabled [56] . Check which events occurred: e.g., ICR bit 7 = RXT0 (receive threshold or timer), bit 0 = TXDW (transmit descriptor written back), bit 2 = LSC (link status change).
31. For link change: read the Status register or specific PHY registers to determine new link state. Update `netdev_link` state accordingly. If link down, set `netif_carrier_off(netdev)`; if up, set `netif_carrier_on(netdev)` and possibly print link speed/duplex (which can be read from the Device Status register's speed/duplex bits).
32. For receive: schedule NAPI poll. Call `napi_schedule(&priv->napi)` (since we added NAPI in open). Also (very important) disable further RX interrupts to defer to polling. On e1000, one approach is to not re-enable RX interrupts until napi_complete. Alternatively, some use the RDTR (receive delay timer) to throttle. Simpler: mask the bits in IMC (Interrupt Mask Clear) register for RX events. Intel drivers often write to IMS to disable or use a flag in the RCTL to not generate interrupts temporarily. But simply calling napi_schedule might be enough if the stack knows to not re-enter (the NAPI is marked in progress). Because e1000 uses a single interrupt for multiple events, we can't fully disable it without also losing link change or other interrupts – unless we use the EI (interrupt throttle) registers. For simplicity, a common trick: in ISR, if we schedule NAPI, we return IRQ_HANDLED and do *not* re-enable those interrupts; the NAPI poll will re-enable them by writing to IMS on completion [70] [78] . We can maintain a flag like priv->napi_in_progress. But since e1000's design is known, we can follow what the real driver does: it has an "interrupt throttling rate" and uses multiple vector with MSI-X in newer chips. However, with legacy, e1000 uses RX interrupt delay

timer by default (so it doesn't interrupt every packet). For our driver, simplest is: on RX event, disable interrupts by writing 0xFFFF to IMC (which masks all) and rely on NAPI poll to later re-enable. Because we likely enabled only a subset in IMS, writing IMC for those bits should suffice.

33. For transmit complete: we can handle TX clean here or in NAPI poll. If NAPI is being used, one strategy is to also do TX reclaim in the poll routine (this is what many drivers do, to avoid needing TX interrupts at all – they either disable TX interrupts entirely or only use them as a fallback). But if we enabled TXDW, we'll get an interrupt when at least one transmit is done (and if RS bit was set in a descriptor). We could clean a few in the ISR itself, but doing so in hardirq is not great if it involves kfree of skbs. Instead, we might use schedule NAPI for TX as well: we can signal that TX cleanup needed and handle it in poll. Or, simply handle it in the same NAPI poll as RX because TX reclaim is quick. For simplicity, we can do TX reclaim in the poll function without using TX interrupts at all – by periodically checking the ring. But to keep it simple: enable TX interrupts with moderate use. Since we are scheduling NAPI on RX or general, we can also schedule it on TX interrupt. The napi poll can check both rings. So we might: if (icr & (RX | TX)) { napi_schedule(...); } so one poll deals with both. Or create separate napi for TX, but that's overkill. So possibly, just schedule NAPI for either RX or TX events. The poll will know which to process by checking descriptors.
34. Acknowledge the interrupts we handled by writing the bits back to ICR (some devices require writing 1s to clear). Actually on e1000, reading ICR clears the bits that were set. So just ensure to read ICR at top.

35. Return IRQ_HANDLED if it was our device (ICR was non-zero), IRQ_NONE otherwise (for shared line if not our interrupt).

36. **Polling (NAPI) Function:** Implement `e1000_poll(struct napi_struct *napi, int budget)`. This will run in softirq context to handle receive (and possibly transmit completions). Pseudocode:

```
int work_done = 0;
// Handle RX:
while (work_done < budget) {
    if (priv->rx_head == NIC_HEAD_INDEX_REGISTER_VALUE)
        break; // no more received packets
    // get descriptor at priv->rx_head
    dma_rmb(); // ensure we see updated descriptor
    if (!(desc.status & DD)) break; // NIC not done, shouldn't happen if
head reg advanced
    len = desc.length;
    if (desc.status & EOP == 0) {
        // jumbo frame spanning multiple desc (we didn't configure, skip)
    }
    skb = priv->rx_skb_ring[rx_head];
    skb_put(skb, len);
    skb->protocol = eth_type_trans(skb, netdev);
    skb->ip_summed = (desc.status & IPCS && desc.status & TCPCS && !
(desc.errors & checksum_error))
                     ? CHECKSUM_UNNECESSARY : CHECKSUM_NONE;
    priv->stats.rx_packets++;
```

```
    priv->stats.rx_bytes += len;
    netif_receive_skb(skb);
    // allocate new skb for this slot
    new_skb = netdev_alloc_skb(netdev, 2048);
    if(new_skb) {
        dma_addr = dma_map_single(&pdev->dev, new_skb->data, 2048,
DMA_FROMDEVICE);
        desc.buffer_addr = dma_addr;
        desc.status = 0;
        priv->rx_skb_ring[rx_head] = new_skb;
    } else {
        // allocation failed, leave descriptor as is (or mark to not reuse,
meaning drop in future)
        priv->stats.rx_dropped++;
        // Could reuse old buffer by not freeing skb, but that complicates
things
    }
    rx_head = (rx_head + 1) % RX_RING_SIZE;
    priv->rx_head = rx_head;
    work_done++;
}
// Update NIC tail register to tell it buffers are free:
if (work_done > 0) {
    writel(priv->rx_head == 0 ? RX_RING_SIZE-1 : priv->rx_head - 1, RDT);
    // Actually, for e1000, you write index of last available descriptor.
    // Simpler: just regularly top it up: always keep RDT = last index of
ring (the NIC uses internal H/W head).
    // Or maintain a "tail" pointer that trails head by some safe margin.
}
// Handle TX reclaim:
while (priv->clean_index != priv->tx_index) {
    // get descriptor at priv->clean_index
    if (!(desc.status & DD))
break; // descriptor not done yet (stop if NIC hasn't marked done)
    // Free associated skb
    dev_kfree_skb(priv->tx_skb_ring[clean_index]);
    priv->tx_skb_ring[clean_index] = NULL;
    priv->stats.tx_packets++;
    priv->stats.tx_bytes += desc.length;
    clean_index = (clean_index + 1) % TX_RING_SIZE;
    priv->clean_index = clean_index;
}
// If we cleaned some and the queue was stopped, wake it if there's room
now:
if (priv->clean_index != priv->tx_index && netif_queue_stopped(netdev)) {
    netif_wake_queue(netdev);
}
// Complete NAPI if done:
```

```
if (work_done < budget) {
    napi_complete_done(napi, work_done);
    // Re-enable interrupts:
    writel(IMS bits for RX/TX we want, IMS);
}
return work_done;
```

The above is conceptual. For e1000 specifically, the NIC uses separate head and tail for RX and TX. We can get NIC's current RX head via register RDH. But more directly, e1000 has an internal mechanism: you have to periodically update RDT (tail) to let NIC know how many free desc. Simpler: leave RDT at ring_size-1 always, which means all descriptors are always available (Intel recommends not doing that to avoid NIC running off end without wrapping properly, but in simulation it might be fine). A better approach: at init, set RDT = ring_size - 1 (meaning index of last descriptor). NIC will then use descriptors 0..last. When you refill descriptors, you might not need to update RDT unless you withheld some descriptors intentionally. Actually, on e1000, after init, you keep RDT ahead of RDH by some margin. Because if RDH catches up to RDT, NIC thinks ring is empty and stops. So indeed, one typically sets RDT to (last index) at start to indicate entire ring is ready. Then NIC moves RDH as it receives. You then have to update RDT as you refill. If we always keep ring full, we can just re-post each buffer immediately and write RDT to that new index. But since we handle one by one here, probably easier to batch update RDT after cleaning a bunch. Perhaps track how many were cleaned and do RDT = (old RDT + cleaned) % ring_size.

Anyway, after processing, we re-enable interrupts by writing to IMS the bits we disabled. For example, `writel(IMS_RXT0 | IMS_RXO | IMS_TXDW | IMS_LSC, IMS_REG);` to re-enable. Actually, because we disabled all in IMC, we re-enable all we want.

1. **Timeout Handler:** If `.ndo_tx_timeout` is triggered, it means something's stuck (no TX complete for too long). In our driver, we can implement `e1000_tx_timeout(netdev)` to log an error and perform a reset. For simplicity, we can schedule a reset work or just call our open->stop->open sequence to reinit the hardware. E.g.:

```
netdev_warn(netdev, "Transmit timeout, resetting device\n");
netif_stop_queue(netdev);
// maybe increment a counter
// Reset NIC:
writel(CTRL_RST, CTRL_REG);
mdelay(1);
// Free pending TX skbs:
while (priv->clean_index != priv->tx_index) {
    dev_kfree_skb(priv->tx_skb_ring[priv->clean_index]);
    priv->clean_index = (priv->clean_index+1) % TX_RING_SIZE;
}
priv->tx_index = priv->clean_index = 0;
// Re-init descriptors and registers:
```

```
e1000_configure_tx_rx(priv); // similar to open
netif_wake_queue(netdev);
```

The idea is to get the hardware and software back in sync.

2. **Cleanup (ndo_stop and remove):** Ensure that in `ndo_stop` we disable interrupts, free all skbs in RX ring (after unmapping, though if we used consistent DMA for descriptors and single mapping for each packet and we recycle them it's fine; if not recycled, unmap and free). Free TX skbs that remain. Free descriptor memory via dma_free_coherent. Free any other allocations. In `pci_remove`, unregister netdev and free it.

3. **Testing and Validation:** After writing this driver, compile it for the kernel in the QEMU VM (could be done by building as module). Boot the VM with `-device e1000,netdev=net0,...` to use the e1000 card. Once the driver is loaded (via modprobe or built-in), bring the interface up: `ip link set dev ethX up` (it might be eth0 if no other NIC). Assign IP: `ip addr add 192.168.0.2/24 dev ethX` and try to ping another host or the gateway. Use `dmesg` to see your driver's prints (you should add debug prints at least for link up, tx timeouts, etc.). Use `ethtool -S ethX` if you implemented anything or at least `ifconfig ethX` to see RX/TX counters increment as you pass traffic.

Resources to consult: The Intel 8254x Software Developer's Manual [117] [106] is essential – it details every register and bit. For example, it describes how to initialize the transmit and receive units (which we summarized from it) including the exact sequence of writing TDBAL, TDLEN, TDH/TDT, TCTL bits, etc. It also explains the descriptor formats and flags. The Linux e1000e driver (in `drivers/net/ethernet/intel/e1000e/`) can be referenced as a blueprint for handling edge cases (though it is more complex due to supporting many chip versions and features). QEMU's e1000 device model (in QEMU source `hw/net/e1000.c`) can also provide insight – since it emulates the hardware, reading it might clarify what the device expects. However, the QEMU model might be complex C code; the Intel manual is more straightforward about device behavior.

By working through this project, you will exercise registering a PCI network driver, interacting with hardware registers via MMIO, managing DMA memory, and hooking into the Linux networking stack. It consolidates the concepts from this chapter in a tangible way. Don't be discouraged by the detailed hardware specifics – start with basic transmission and reception, then incrementally handle more (like link status or offloads) as needed. Testing in QEMU is very convenient because if something goes wrong, it won't crash your host, only the VM. Use tools like `tcpdump` on the host or VM to verify packets, and enable debugging prints in your driver to trace execution. With persistence, you'll have a working e1000 driver module that can talk on the network, demonstrating a full implementation of a Linux network device driver on kernel 6.x.

---

1 2 3 4 6 7 10 11 12 13 14 15 16 17 18 19 20 21 26 27 28 30 33 34 35 36 37 38 39 40 41 42 43 44 45 46 59 62 74 75 76 77 88 89 91 92 93 98 99 100 101 102 ch17.pdf
file://file_000000005b40722fa21af6a1b9698c60

5 24 25 29 60 61 63 64 Netdev - HackMD
https://hackmd.io/@octobersky/HyGqYAKcB

8 22 23 net_device struct fast path usage breakdown — The Linux Kernel documentation
https://docs.kernel.org/networking/net_cachelines/net_device.html

9 49 50 57 58 70 71 72 73 78 79 80 81 82 83 84 85 86 87 96 97 103 104 116 Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog
https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/

31 32 47 48 51 52 53 54 55 56 65 66 67 68 69 90 94 95 105 106 107 108 109 110 111 112 113 114 115 117 8254x Family of Gigabit Ethernet Controllers Software Developer’s Manual
https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf