Linux Networking Drivers

Based on the Intel igb Driver

From Ring Buffers to AF_XDP Zero-Copy

A Practical Guide for Developers New to the Kernel

February 2026

Contents

Chapter 1

Introduction

1.1 What This Book Covers

This book is a comprehensive guide to understanding Linux networking drivers, using Intel's **igb** driver as the primary case study. The igb driver manages Intel's 1 Gigabit Ethernet controllers — specifically the 82575, 82576, 82580, I350, I354, and I210/I211 chipsets — and serves as an excellent example because it implements virtually every feature a modern networking driver needs: DMA ring buffers, NAPI polling, MSI-X interrupts, SR-IOV virtualization, hardware timestamping, and most recently XDP (eXpress Data Path) and AF_XDP zero-copy support.

We chose the igb driver for several reasons. First, it is mature and well-understood, having been in the kernel since version 2.6.25. Second, Intel's 1GbE controllers are among the most commonly deployed Ethernet adapters in servers, embedded systems, and networking appliances worldwide. Third, the driver is clean enough to learn from while still being complex enough to demonstrate real-world patterns. Finally, recent additions of XDP and AF_XDP zero-copy support (landed in Linux 6.14) make it especially relevant for anyone interested in high-performance packet processing.

1.2 Who This Book Is For

This book targets software engineers who have some programming experience (ideally in C) and a basic understanding of operating systems, but who have never worked inside the Linux kernel. If you know what a system call is, can read C code, and understand the basics of how a computer's memory works, you have all the prerequisites you need.

By the end of this book, you will understand how a packet travels from the wire to userspace, how the igb driver manages hardware resources, and how XDP and AF_XDP allow you to process packets at speeds approaching line rate without ever touching the traditional network stack.

1.3 The igb Driver at a Glance

The igb driver lives in the Linux kernel source tree at:

```
drivers/net/ethernet/intel/igb/
```

It consists of several key source files:

File	Purpose
`igb_main.c`	Core driver logic: probe, open, close, transmit, receive, NAPI poll, XDP
`igb.h`	Main header: data structures (`igb_adapter`, `igb_ring`, `igb_q_vector`)
`igb_ethtool.c`	ethtool interface: statistics, ring params, self-test, diagnostics
`igb_ptp.c`	IEEE 1588 Precision Time Protocol / hardware clock support
`igb_xsk.c`	AF_XDP zero-copy support (added in Linux 6.14)
`e1000_hw.h`	Hardware register definitions and constants
`e1000_82575.c/h`	Hardware-abstraction layer for 82575/82576/82580
`e1000_i210.c/h`	Hardware-specific code for I210/I211 controllers
`e1000_mac.c/h`	Common MAC operations (link, flow control)
`e1000_phy.c/h`	PHY (physical layer) management

1.4 Supported Hardware

The igb driver supports a range of Intel Gigabit Ethernet controllers. Each generation brought new features:

Controller	PCIe Gen	Max Queues	Key Features
82575	Gen1 x1	3 RX / 3 TX	Basic multi-queue, MSI-X
82576	Gen2 x4	16 RX / 16 TX	SR-IOV (up to 7 VFs), VMDq
82580	Gen2 x4	8 RX / 8 TX	DCA, improved timestamp
I350	Gen2 x4	8 RX / 8 TX	Enhanced SR-IOV, EEE, 4-port
I354	Gen2 x1	8 RX / 8 TX	Quad-port, SFP support
I210	Gen2 x1	4 RX / 4 TX	PTP (ns precision), AVB, CBS/LaunchTime
I211	Gen2 x1	2 RX / 2 TX	Consumer variant of I210

> **NOTE:** The I210 and I211 are particularly popular in embedded and IoT applications due to their precise hardware timestamping and Audio/Video Bridging (AVB) capabilities.

1.5 Key Timeline of igb XDP Development

Date	Kernel	Milestone
Sep 2020	5.10	Initial XDP support (XDP_PASS, XDP_DROP, XDP_TX, XDP_REDIRECT)
Oct 2020	5.10	Follow-up: TX timeout fix, VLAN double header, metadata support
2023–2024	6.x	AF_XDP zero-copy patches developed and reviewed (8 revisions)
Jan 2025	6.14	AF_XDP zero-copy Rx and Tx support merged into net-next

Chapter 2

Linux Kernel Networking Fundamentals

2.1 The Network Stack in 60 Seconds

When a packet arrives at a network interface card (NIC), it goes through several stages before reaching a userspace application:

1. Packet arrives at the NIC from the physical medium (copper, fiber).
2. The NIC copies the packet into a pre-allocated region of system RAM via DMA (Direct Memory Access).
3. The NIC raises a hardware interrupt to notify the CPU.
4. The interrupt handler in the driver schedules a NAPI poll (software interrupt).
5. The NAPI poll function harvests packets from the DMA ring buffer.
6. Each packet is wrapped in a socket buffer (`sk_buff`) and passed to the protocol stack.
7. The protocol stack (IP, TCP/UDP) processes headers and delivers data to sockets.
8. The application reads data from the socket via system calls (`read`/`recv`/`recvmsg`).

XDP inserts itself between steps 2 and 5 — it runs a BPF program on the raw packet data before any `sk_buff` is allocated, which is why it can be so fast.

2.2 PCI and PCIe: How the NIC Talks to the CPU

Every Intel Gigabit Ethernet controller communicates with the host system over PCI Express (PCIe). Understanding PCIe is fundamental to understanding NIC drivers because it governs how the driver discovers, configures, and communicates with the hardware.

2.2.1 PCI Device Discovery

When the system boots, the PCI subsystem enumerates all devices on the bus. Each device is identified by a Vendor ID and Device ID pair. The igb driver registers itself with the PCI subsystem by providing a table of supported device IDs:

```
static const struct pci_device_id igb_pci_tbl[] = {
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_82575EB_COPPER), board_82575 },
```

```
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_COPPER),    board_82575 },
    { PCI_VDEVICE(INTEL, E1000_DEV_ID_I211_COPPER),    board_82575 },
    /* ... more entries ... */
};
```

When the PCI subsystem finds a device matching one of these IDs, it calls the driver's `probe` function.

2.2.2 Base Address Registers (BARs)

PCIe devices expose memory regions through Base Address Registers (BARs). The igb driver maps BAR 0 to access the NIC's control and status registers (CSRs). All communication between the driver and hardware happens through reading and writing these memory-mapped registers.

2.2.3 DMA — Direct Memory Access

DMA is the mechanism that allows the NIC to read and write system RAM without CPU involvement. This is crucial for performance: instead of the CPU copying every byte of every packet, the NIC writes packet data directly into pre-allocated buffers in RAM.

The driver's job is to:

- Allocate memory pages for packet buffers.
- Map those pages for DMA access (translate virtual to physical addresses).
- Tell the NIC where these buffers are by writing DMA addresses into descriptor rings.
- After the NIC writes packet data, unmap/sync the DMA region so the CPU sees correct data.

> **WARNING:** DMA addresses are not the same as physical addresses on systems with an IOMMU. The IOMMU translates DMA addresses, providing isolation (important for SR-IOV) and allowing 32-bit devices to access memory above 4 GB.

2.3 Interrupts: MSI-X, MSI, and Legacy

Interrupts are how the NIC tells the CPU that something happened. There are three interrupt delivery mechanisms:

Type	Description	Advantage
MSI-X	Multiple vectors, each with own IRQ	Per-queue interrupts, CPU affinity control
MSI	Single message-signaled interrupt	No sharing, cleaner than legacy
Legacy (INTx)	Shared, level-triggered pin	Universally supported, but slowest

The igb driver prefers MSI-X because it allows each queue (or queue pair) to have its own interrupt vector directed to a specific CPU core. The driver falls back to MSI, then legacy, if MSI-X is not available.

2.4 NAPI: The Polling Revolution

NAPI (New API) is a Linux kernel mechanism that replaces the old "one interrupt per packet" model with a hybrid interrupt-driven/polling approach. It is the single most important optimization in the Linux networking stack.

2.4.1 The Problem with Pure Interrupts

If a NIC generates one interrupt for every received packet, a system under heavy load (say, 1 million packets per second) would need to handle 1 million interrupts per second. Each interrupt involves context switching, saving/restoring registers, and other overhead. The system would spend all its time handling interrupts and no time processing packets. This condition is called "receive livelock."

2.4.2 How NAPI Works

NAPI solves this with a two-phase approach:

1. **Interrupt phase:** When the first packet arrives, the NIC raises an interrupt. The interrupt handler disables further interrupts for that queue and schedules a NAPI poll.
2. **Polling phase:** The softirq subsystem calls the driver's poll function, which processes packets in a loop up to a "budget" (default 64 packets per invocation).
3. If more packets remain, the poll is rescheduled. If the ring is empty, the driver re-enables interrupts and exits polling mode.

In the igb driver, the key functions are:

```
igb_msix_ring()    /* Interrupt handler: calls napi_schedule() */
igb_poll()         /* NAPI poll: calls igb_clean_tx_irq() + igb_clean_rx_irq() */
igb_clean_rx_irq() /* Harvests packets from the RX ring */
```

> **TIP:** Under sustained high traffic, NAPI may never re-enable interrupts because the ring buffer is never empty. This is ideal — the system processes packets at maximum rate with zero interrupt overhead. This is the same principle behind busy-polling in AF_XDP.

2.5 Socket Buffers (sk_buff)

The `sk_buff` (usually abbreviated "skb") is the fundamental data structure for packets in the Linux kernel. It contains a pointer to packet data, header offsets, metadata (device, timestamp, checksum info, protocol), and linked list pointers for queuing.

Allocating and managing `sk_buff`s is a significant source of overhead. This is one key reason XDP is faster — it operates on raw `xdp_buff` structures that are much lighter.

Chapter 3

igb Driver Architecture

3.1 Driver Lifecycle

A kernel driver follows a well-defined lifecycle:

1. **Module Load:** `igb_init_module()` registers the driver with the PCI subsystem.
2. **Device Probe:** For each matching PCI device, `igb_probe()` allocates the adapter structure, maps BAR registers, resets the hardware, and registers the network device.
3. **Interface Up:** `igb_open()` allocates ring buffers, sets up DMA, registers interrupts, and starts the hardware.
4. **Data Path:** Packets are received via NAPI polling and transmitted via `igb_xmit_frame()`.
5. **Interface Down:** `igb_close()` stops the hardware, frees rings and interrupts.
6. **Device Remove:** `igb_remove()` tears down everything allocated in probe.
7. **Module Unload:** `igb_exit_module()` unregisters from PCI.

3.2 Key Data Structures

3.2.1 igb_adapter

The `igb_adapter` structure is the driver's "brain" — it holds all per-device state:

```
struct igb_adapter {
    struct net_device    *netdev;      /* The Linux network device     */
    struct pci_dev       *pdev;        /* The PCI device               */
    struct e1000_hw      hw;           /* Hardware abstraction layer   */
    struct bpf_prog      *xdp_prog;   /* Attached XDP program         */

    struct igb_ring      *tx_ring[16]; /* Transmit ring pointers       */
    struct igb_ring      *rx_ring[16]; /* Receive ring pointers        */
    struct igb_q_vector  *q_vector[8]; /* Queue vector array           */

    unsigned int         num_tx_queues;
    unsigned int         num_rx_queues;
    unsigned int         num_q_vectors;
    /* ... hundreds more fields ... */
};
```

3.2.2 igb_ring

Each transmit or receive queue is represented by an `igb_ring`:

```
struct igb_ring {
    struct igb_q_vector  *q_vector;   /* Parent interrupt vector      */
    struct net_device    *netdev;
    struct device        *dev;         /* For DMA operations          */
    struct bpf_prog      *xdp_prog;   /* Per-ring XDP program pointer */
    struct xsk_buff_pool *xsk_pool;   /* AF_XDP zero-copy pool        */

    union {
        struct igb_tx_buffer *tx_buffer_info;
        struct igb_rx_buffer *rx_buffer_info;
    };

    void                 *desc;        /* Descriptor ring (DMA-coherent) */
    dma_addr_t           dma;          /* Physical addr of desc ring   */
    unsigned int         size;         /* Size of desc ring in bytes   */

    u16                  count;        /* Number of descriptors        */
    u16                  next_to_use;  /* Driver writes here           */
    u16                  next_to_clean;/* Driver reads/frees here      */
    u8 __iomem           *tail;        /* MMIO addr of tail register   */
};
```

3.2.3 igb_q_vector

A queue vector ties together an interrupt, one or more rings, and NAPI polling:

```
struct igb_q_vector {
    struct igb_adapter   *adapter;
    struct igb_ring_container rx, tx; /* Associated rings             */
    struct napi_struct   napi;        /* NAPI instance                */
    u16                  itr_val;     /* Interrupt throttle rate      */
};
```

3.3 The Descriptor Ring

The descriptor ring is the primary communication mechanism between the driver and the NIC hardware. It is a circular buffer of fixed-size descriptors (16 bytes each) allocated in DMA-coherent memory.

3.3.1 How It Works

Each descriptor contains a DMA address pointing to a packet buffer in system RAM, plus metadata (packet length, status bits, checksum offloads). Two pointers track the ring state:

- **next_to_use (NTU):** Where the driver places the next available buffer (RX) or packet (TX). The driver advances this pointer and writes it to the hardware tail register.
- **next_to_clean (NTC):** Where the driver looks for completed operations. The driver reads the Descriptor Done (DD) status bit.

The hardware maintains its own head pointer (readable via MMIO). The space between head and tail represents descriptors owned by the hardware.

3.3.2 RX Descriptor Format

Intel uses "Advanced Receive Descriptors" with two formats:

```
/* Read format (driver -> hardware) */
+-----------------------------------------------+
| Buffer Address [63:0]                         |  8 bytes
+-----------------------------------------------+
| Header Buffer Address [63:0]                  |  8 bytes
+-----------------------------------------------+

/* Writeback format (hardware -> driver) */
+-----------------------------------------------+
| Packet Checksum | IP ID | Status/Error        |  8 bytes
+-----------------------------------------------+
| VLAN | Error | Status | Checksum | Length      |  8 bytes
+-----------------------------------------------+
```

3.3.3 TX Descriptor Format

```
/* Advanced Transmit Descriptor */
+-----------------------------------------------+
| Buffer Address [63:0]                         |  8 bytes
+-----------------------------------------------+
| PAYLEN | PORTS | CC | IDX | STA | DCMD | ... |  8 bytes
+-----------------------------------------------+
```

The command (DCMD) field tells the hardware what offloads to perform: checksum insertion, TSO, VLAN tag insertion, etc.

3.4 Buffer Management and Page Recycling

One of the most performance-critical aspects of the igb driver is how it manages memory for packet data:

1. **Pre-allocation:** During ring setup, the driver allocates one page per RX descriptor and maps it for DMA.
2. **Split pages:** Each 4 KB page can hold multiple packet buffers (e.g., two 2 KB buffers). The `page_offset` field tracks which half is in use.
3. **Page recycling:** After processing, the driver checks if the page can be reused via a `pagecnt_bias` mechanism, avoiding expensive atomic refcount operations.
4. **The flip:** If reusable, the driver "flips" to the other half of the page, serving multiple receive operations from a single allocation.

> **TIP:** Page recycling is a critical optimization. Without it, the driver would need to call the page allocator for every received packet, a major bottleneck at high packet rates.

Chapter 4

The Receive Path in Detail

4.1 From Wire to Ring Buffer

When a packet arrives at the NIC, the following sequence occurs entirely in hardware:

1. The MAC (Media Access Controller) verifies the Ethernet frame CRC.
2. If RSS is enabled, the NIC computes a hash and selects a receive queue.
3. The NIC reads the next available RX descriptor from the descriptor ring.
4. Using the buffer address, the NIC DMAs the packet data into system RAM.
5. The NIC writes back the descriptor with status information (length, checksum, VLAN).
6. The NIC updates its internal head pointer.
7. If interrupt moderation has expired, the NIC raises an MSI-X interrupt.

4.2 igb_clean_rx_irq: The Heart of the Receive Path

This function is called from `igb_poll()` and does the actual work:

```
static int igb_clean_rx_irq(struct igb_q_vector *q_vector, int budget)
{
    struct igb_ring *rx_ring = q_vector->rx.ring;
    u16 cleaned_count = igb_desc_unused(rx_ring);
    struct sk_buff *skb = rx_ring->skb;

    while (likely(total_packets < budget)) {
        union e1000_adv_rx_desc *rx_desc;
        struct igb_rx_buffer *rx_buffer;

        /* 1. Get the next descriptor */
        rx_desc = IGB_RX_DESC(rx_ring, rx_ring->next_to_clean);
        if (!igb_test_staterr(rx_desc, E1000_RXD_STAT_DD))
            break;  /* No more completed descriptors */

        /* 2. Memory barrier to ensure fresh data */
        dma_rmb();

        /* 3. Get the buffer and unmap DMA */
        rx_buffer = igb_get_rx_buffer(rx_ring, size, &rx_buf_pgcnt);
```

```
        /* 4. *** XDP RUNS HERE *** (if program loaded) */
        if (xdp_prog) {
            xdp_result = igb_run_xdp(adapter, rx_ring, &xdp, xdp_prog);
            if (xdp_result != IGB_XDP_PASS) continue;
        }

        /* 5. Build an sk_buff from the data */
        skb = igb_construct_skb(rx_ring, rx_buffer, &xdp, timestamp);

        /* 6. Process headers, VLAN, checksum */
        igb_process_skb_fields(rx_ring, rx_desc, skb);

        /* 7. Send up the stack via GRO */
        napi_gro_receive(&q_vector->napi, skb);
    }

    /* 8. Refill the ring with fresh buffers */
    if (cleaned_count)
        igb_alloc_rx_buffers(rx_ring, cleaned_count);

    return total_packets;
}
```

Notice step 4: this is where XDP hooks into the receive path. If an XDP program is attached, it runs on the raw packet data before any `sk_buff` is allocated. If the program returns `XDP_DROP`, `XDP_TX`, or `XDP_REDIRECT`, the allocation in step 5 is completely skipped.

4.3 GRO: Generic Receive Offload

Before packets reach the protocol stack, they pass through GRO (Generic Receive Offload). GRO merges small packets from the same TCP flow into larger ones, reducing per-packet overhead in the upper layers. This is the software equivalent of hardware LRO, but safer because it preserves packet boundaries.

4.4 Interrupt Moderation

The igb driver implements adaptive interrupt throttling (ITR) to balance latency and throughput. You can override this with ethtool:

```
ethtool -C ethX rx-usecs 50     # Set RX interrupt coalesce to 50us
ethtool -C ethX adaptive-rx on  # Enable adaptive mode
```

Chapter 5

The Transmit Path

5.1 From Socket to Wire

When an application calls `send()`, the packet traverses the stack in reverse:

1. The protocol layer builds the packet and calls the device's transmit function.
2. The qdisc (queuing discipline) enqueues the packet.
3. `dev_hard_start_xmit()` calls `igb_xmit_frame()`.
4. `igb_xmit_frame()` maps the sk_buff data for DMA and writes transmit descriptors.
5. The driver writes the tail register to tell the hardware new descriptors are available.
6. The hardware reads descriptors, fetches packet data via DMA, and transmits.
7. After transmission, the hardware writes back the descriptor with a Done status.
8. In the next NAPI poll, `igb_clean_tx_irq()` frees the transmitted buffers.

5.2 TX Descriptor Setup

For each packet, the driver creates one or more TX descriptors. The first descriptor ("context descriptor") may contain offload parameters (TSO, checksum). A key optimization is that the driver maps the existing sk_buff's data pages for DMA rather than copying — zero-copy transmit from the driver's perspective.

5.3 Hardware Offloads

Offload	Description
TX Checksum	Hardware computes and inserts L3/L4 checksums
TSO	Hardware splits large TCP segments into MTU-sized frames
VLAN Insertion	Hardware inserts VLAN tags on egress
IEEE 1588 Timestamping	Hardware records precise TX timestamps (I210)
LaunchTime / CBS	Time-based scheduling of frames (I210, for AVB/TSN)

Chapter 6

XDP (eXpress Data Path) in igb

6.1 What is XDP?

XDP is a high-performance, programmable packet processing framework built on eBPF. It allows you to attach a small program to the NIC driver that runs for every received packet, before the kernel allocates an `sk_buff` or does any protocol processing.

Verdict	Action	Use Case
`XDP_PASS`	Continue to normal stack	Selective filtering
`XDP_DROP`	Drop immediately	DDoS mitigation, firewalling
`XDP_TX`	Transmit back out same NIC	Reflection, load balancers
`XDP_REDIRECT`	Forward to another NIC/CPU/AF_XDP	Forwarding, AF_XDP
`XDP_ABORTED`	Error case, drop with trace	Debugging

On igb hardware, the original XDP patches demonstrated:

Path	Performance
Normal stack routing	382 Kpps
XDP Redirect (`xdp_fwd`)	1.48 Mpps
XDP Drop	1.48 Mpps

Tested on an Intel Atom C2338 at 1.74 GHz with two I211 NICs — achieving line rate forwarding at 1 Gbps on a very low-power CPU.

6.2 How XDP Was Added to igb

XDP support was contributed by Sven Auhagen (Voleatech) in September 2020, closely following the ixgbe driver's implementation. The patch went through 6 revisions.

6.2.1 Shared TX Queues

Unlike ixgbe (which has enough hardware queues to dedicate separate ones for XDP), igb has limited TX queues. XDP must share TX queues with the normal stack, requiring locking:

```
static inline struct igb_ring *igb_xdp_tx_queue_mapping(
        struct igb_adapter *adapter)
{
    unsigned int r_idx = smp_processor_id();
    if (r_idx >= adapter->num_tx_queues)
        r_idx = r_idx % adapter->num_tx_queues;
    return adapter->tx_ring[r_idx];
}
```

6.2.2 Buffer Layout Changes

When XDP is active, the driver adds `XDP_PACKET_HEADROOM` (256 bytes) before each packet buffer. This allows XDP programs to prepend headers using `bpf_xdp_adjust_head()` without buffer reallocation.

6.2.3 The igb_run_xdp Function

This is the core XDP dispatch function:

```
static int igb_run_xdp(struct igb_adapter *adapter,
                       struct igb_ring *rx_ring,
                       struct xdp_buff *xdp,
                       struct bpf_prog *xdp_prog)
{
    u32 act = bpf_prog_run_xdp(xdp_prog, xdp);

    switch (act) {
    case XDP_PASS:
        return IGB_XDP_PASS;
    case XDP_TX:
        igb_xdp_xmit_back(adapter, xdp);
        return IGB_XDP_TX;
    case XDP_REDIRECT:
        xdp_do_redirect(adapter->netdev, xdp, xdp_prog);
        return IGB_XDP_REDIRECT;
    default: /* XDP_ABORTED or XDP_DROP */
        return IGB_XDP_CONSUMED;
    }
}
```

6.3 XDP Limitations on igb

- **MTU restriction:** Max frame must fit in a single page (~3.5 KB with headroom). No jumbo frames with XDP.
- **TX queue sharing:** XDP shares TX queues with the stack, requiring locking that adds overhead.
- **No multi-buffer XDP:** igb does not support XDP for fragmented/multi-buffer packets.

- **TX timeout (fixed):** Early versions could trigger transmit queue timeout. Fixed by setting `trans_start = jiffies` in the XDP transmit path.

6.4 Loading an XDP Program on igb

```
# Compile an XDP program
clang -O2 -target bpf -c xdp_prog.c -o xdp_prog.o

# Load in driver mode (native XDP)
ip link set dev eth0 xdpdrv obj xdp_prog.o sec xdp

# Verify it's loaded
ip link show eth0   # Should show: xdp/id:XX

# Remove the program
ip link set dev eth0 xdp off
```

> **NOTE:** When an XDP program is loaded or unloaded, the igb driver resets the interface to reconfigure buffer layouts. This briefly interrupts traffic.

Chapter 7

AF_XDP Zero-Copy Support

7.1 What is AF_XDP?

AF_XDP is a socket type (address family) that provides a high-performance path from the NIC directly to userspace. Introduced in Linux 4.18, when combined with XDP it allows userspace applications to send and receive raw packets with minimal kernel overhead through shared memory rings.

7.2 AF_XDP Architecture

An AF_XDP socket uses four shared rings plus a UMEM (user-space memory area):

Component	Direction	Purpose
FILL Ring	User → Kernel	User provides empty buffer addresses for RX
RX Ring	Kernel → User	Kernel delivers received packet descriptors
TX Ring	User → Kernel	User submits packets for transmission
COMPLETION Ring	Kernel → User	Kernel confirms transmitted packets
UMEM	Shared	Pre-registered memory region for all packet data

The UMEM is a contiguous memory region divided into fixed-size "chunks" (typically 4 KB, matching page size). All packet data is stored in UMEM chunks, and the rings pass indices/addresses referencing these chunks.

7.3 Copy Mode vs Zero-Copy Mode

7.3.1 Copy Mode (Generic)

In copy mode, the kernel copies packet data from the driver's DMA buffers into the UMEM. This works with any XDP-capable driver but adds latency and CPU overhead. Performance is typically 2–3× better than regular sockets but far from the hardware limit.

7.3.2 Zero-Copy Mode (Driver-Specific)

In zero-copy mode, the driver performs DMA directly into/from the UMEM buffers. There is no data copy at all — the packet data the NIC writes via DMA is the exact same memory the

application reads. This requires explicit support from each driver.

For igb, zero-copy support was added in Linux 6.14 by Sriram Yagnaraman (Linutronix) and Kurt Kanzenbach. The patches went through 8 revisions over nearly two years (2023–2025).

7.4 igb AF_XDP Zero-Copy Implementation

The zero-copy implementation lives in `igb_xsk.c` and modifies the core receive and transmit paths.

7.4.1 Key Data Structures

The `igb_ring` gains a new field:

```
struct xsk_buff_pool *xsk_pool;  /* Non-NULL when ZC is active */
```

7.4.2 RX Zero-Copy Path

1. **Fill:** Instead of allocating pages, the driver calls `xsk_buff_alloc()` to get buffers from the user's FILL ring.
2. **Receive:** DMA writes directly into these UMEM buffers. No copy needed.
3. **Deliver:** The driver writes packet descriptors to the RX ring.
4. **Recycle:** The application returns buffer addresses via the FILL ring.

7.4.3 TX Zero-Copy Path

1. The application writes packet data into UMEM buffers and posts descriptors to the TX ring.
2. The driver reads TX ring descriptors and maps corresponding UMEM addresses for DMA.
3. Hardware reads packet data directly from UMEM and transmits.
4. The driver posts completed buffer addresses to the COMPLETION ring.

7.5 Performance Numbers

Benchmarks from QEMU-emulated 82576 NIC (real hardware would be significantly faster):

Benchmark	XDP-SKB	XDP-DRV	XDP-DRV (ZC)
rxdrop (64B)	200 Kpps	210 Kpps	310 Kpps
txpush (64B)	1 Kpps	1 Kpps	410 Kpps
l2fwd (64B)	0.9 Kpps	1 Kpps	160 Kpps

WARNING: These numbers were from a QEMU emulator, not real hardware. On actual I210/I350 hardware, absolute numbers would be significantly higher.

7.6 Known Issues and Fixes During Development

The AF_XDP zero-copy patches went through 8 revisions. Key issues found and fixed:

- **TX unit hang (v2→v3):** Fixed by properly setting `time_stamp` on `tx_buffer_info`.
- **Uninitialized nb_buffs (v2→v3):** Variable not initialized, causing undefined behavior. Reported by Simon Horman.
- **READ/WRITE_ONCE for xsk_pool (v5→v6):** Added atomic access macros to prevent torn reads.
- **synchronize_net() missing (v5→v6):** Needed when disabling zero-copy to ensure no NAPI poll uses stale pointer.
- **napi_id for busy polling (v4→v5):** Registration was missing the napi_id, which broke busy polling.
- **olinfo_status in TX (v4→v5):** TX descriptor field not set in zero-copy path; frames were not transmitted on real hardware.

7.7 Using AF_XDP with igb

```
# 1. Steer traffic to a specific queue
ethtool -N eth0 rx-flow-hash udp4 fn
ethtool -N eth0 flow-type udp4 src-port 4242 action 0

# 2. Run AF_XDP in zero-copy mode
./xdpsock -i eth0 -q 0 -r -N -z  # Force zero-copy

# 3. Best performance with busy polling
echo 2 | sudo tee /sys/class/net/eth0/napi_defer_hard_irqs
echo 200000 | sudo tee /sys/class/net/eth0/gro_flush_timeout
```

> **NOTE:** If zero-copy is not supported, AF_XDP silently falls back to copy mode. Use the `XDP_ZEROCOPY` bind flag to force zero-copy and get an error if unavailable.

Chapter 8

SR-IOV and Virtualization

8.1 What is SR-IOV?

Single Root I/O Virtualization (SR-IOV) is a PCI specification that allows a single physical NIC to present itself as multiple virtual NICs. Each Virtual Function (VF) can be assigned directly to a VM, providing near-native performance. The igb driver supports SR-IOV on 82576 and I350 controllers (up to 7 VFs each).

8.2 PF and VF Architecture

- **Physical Function (PF):** The main driver (igb) that manages physical hardware and controls VF configuration.
- **Virtual Function (VF):** A lightweight function with its own queues, interrupts, and register space. Each VF appears as a separate PCI device managed by the `igbvf` driver.

The PF driver is responsible for creating/destroying VFs, setting MAC addresses and VLAN filters, rate limiting VF traffic, handling mailbox messages, and anti-spoofing enforcement.

8.3 Enabling SR-IOV

```
# Enable 4 VFs on the PCI device
echo 4 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs

# Configure a VF MAC address
ip link set eth0 vf 0 mac 00:11:22:33:44:55
```

> **WARNING:** Do not unload the PF driver (igb) while VFs are assigned to guests. This can cause system instability.

Chapter 9

Performance Tuning and Troubleshooting

9.1 ethtool: Your Best Friend

```
ethtool -i eth0              # Driver info
ethtool -g eth0              # Ring sizes
ethtool -G eth0 rx 4096 tx 4096  # Increase rings
ethtool -l eth0              # Queue count
ethtool -L eth0 combined 4   # Set queues
ethtool -k eth0              # Offload settings
ethtool -S eth0              # Statistics (very detailed)
ethtool -c eth0              # Interrupt coalescing
```

9.2 IRQ Affinity

For maximum performance, pin each queue's interrupt to a specific CPU core:

```
cat /proc/interrupts | grep eth0
echo 1 > /proc/irq/<IRQ_NUM>/smp_affinity  # Pin to CPU 0
```

> **TIP:** For latency-sensitive workloads, disable irqbalance and manually pin IRQs. For throughput, irqbalance usually does a good job.

9.3 Common Problems

9.3.1 NETDEV WATCHDOG: Transmit Queue Timeout

This is one of the most commonly reported igb issues. The kernel's watchdog detects that a TX queue hasn't transmitted within a timeout period and resets the adapter. Common causes:

- PCIe link instability (especially with Thunderbolt docks)
- Misconfigured interrupt affinity leading to starvation
- XDP programs that consume all TX descriptors without cleaning
- Hardware issues (check dmesg for PCIe errors)

9.3.2 Hardware Initialization Failure (-5)

Some I210 devices fail with "Hardware Initialization Failure" error code -5 (PHY init error). Typically caused by EEPROM/NVM corruption or firmware incompatibility. Updating the NIC firmware usually resolves this.

9.3.3 XDP MTU Limitations

When loading an XDP program, the driver checks if the MTU fits within a single page buffer. If jumbo frames are enabled (MTU >∼3500), XDP will refuse to load. Solution: reduce MTU to 1500 first.

9.4 Monitoring with bpftrace

```
bpftrace -e 'kprobe:igb_poll { @polls = count(); }'
bpftrace -e 'kprobe:igb_alloc_rx_buffers { @refills = count(); }'
cat /proc/net/softnet_stat
```

Chapter 10

Practical XDP and AF_XDP Development

10.1 Writing Your First XDP Program

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, __u64);
} pkt_count SEC(".maps");

SEC("xdp")
int xdp_counter(struct xdp_md *ctx)
{
    __u32 key = 0;
    __u64 *count = bpf_map_lookup_elem(&pkt_count, &key);
    if (count)
        (*count)++;
    return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

10.2 XDP Forwarding Between igb Interfaces

A common use case is forwarding packets between two igb interfaces at XDP speed. The kernel's `xdp_fwd` sample program does this using `bpf_redirect_map()` with a DEVMAP (device map). This was used to benchmark the 1.48 Mpps figure on the Atom CPU.

10.3 AF_XDP Application Architecture

A typical AF_XDP application follows this pattern:

1. Create an AF_XDP socket and allocate a UMEM region.

2. Configure the FILL, RX, TX, and COMPLETION rings via `setsockopt`/`mmap`.
3. Load an XDP program that steers traffic to the socket (via an XSKMAP).
4. Fill the FILL ring with buffer addresses.
5. Enter a poll loop: check RX ring, process packets, post TX, check COMPLETION ring.
6. Return processed buffers to the FILL ring for reuse.

For Rust developers, the **aya** crate provides a safe AF_XDP API. For C, the kernel's `libxdp` and `libbpf` libraries are the standard choice.

10.4 Relevance to Userspace Packet Processing

If you're building high-performance packet processing applications (SD-WAN routers, VPN gateways, firewalls):

- igb shares TX queues — minimize `XDP_TX`, prefer `XDP_REDIRECT` to another interface.
- AF_XDP zero-copy requires Linux 6.14+ for igb.
- 1.48 Mpps on an Atom CPU proves line-rate 1GbE forwarding on very low-power hardware.
- For 10/40/100 GbE, ixgbe/i40e/ice follow the same patterns but with dedicated XDP TX queues.

Chapter 11

Comparing igb with Other Intel Drivers

11.1 The Intel Driver Family

Driver	Speed	Controllers	XDP	AF_XDP ZC
e1000e	1 GbE	I217–I219 (PCH)	No	No
igb	1 GbE	82575–I211	Yes (5.10+)	Yes (6.14+)
igc	2.5 GbE	I225, I226	Yes	Yes (5.14+)
ixgbe	10 GbE	82599, X540, X550	Yes	Yes
i40e	10–40 GbE	X710, XL710, XXV710	Yes	Yes
ice	10–100 GbE	E800 series	Yes	Yes

11.2 Key Architectural Differences

Feature	igb (1GbE)	ixgbe (10GbE)	ice (100GbE)
TX queues for XDP	Shared (locked)	Dedicated	Dedicated
Max queues	8–16	64–128	256+
AF_XDP ZC since	6.14	4.18	5.5
Multi-buffer XDP	No	Yes	Yes
HW timestamping	Yes (I210)	Limited	Yes

Appendix A

Glossary

Term	Definition
AF_XDP	Address Family for XDP sockets, providing high-performance raw packet access
BAR	Base Address Register — PCIe memory region exposed by a device
BPF/eBPF	Extended Berkeley Packet Filter — in-kernel virtual machine for safe programs
CBS	Credit-Based Shaper — TSN traffic shaping algorithm (I210)
DCA	Direct Cache Access — optimization to warm CPU cache for incoming data
DD	Descriptor Done — status bit set by hardware when a descriptor is complete
DMA	Direct Memory Access — hardware reading/writing RAM without CPU
GRO	Generic Receive Offload — software merging of small packets
ITR	Interrupt Throttle Rate — minimum interval between interrupts
MSI-X	Message Signaled Interrupts Extended — per-queue interrupt vectors
NAPI	New API — interrupt/polling hybrid for efficient packet processing
NTU/NTC	Next To Use / Next To Clean — ring buffer pointers
PTP	Precision Time Protocol — IEEE 1588, nanosecond clock synchronization
RSS	Receive Side Scaling — hardware distribution of packets across queues
sk_buff	Socket buffer — the kernel's packet data structure
SR-IOV	Single Root I/O Virtualization — hardware NIC virtualization
TSN	Time-Sensitive Networking — IEEE 802.1 standards for deterministic networking
TSO	TCP Segmentation Offload — hardware splits large TCP segments
UMEM	User Memory — pre-registered memory region for AF_XDP
VF	Virtual Function — virtual NIC in SR-IOV
XDP	eXpress Data Path — programmable fast-path packet processing
XSK	XDP Socket — another name for AF_XDP socket

Appendix B

Essential Kernel Config Options

```
CONFIG_IGB=m              # igb driver (module)
CONFIG_BPF=y              # BPF subsystem
CONFIG_BPF_SYSCALL=y      # BPF system call
CONFIG_XDP_SOCKETS=y      # AF_XDP support
CONFIG_BPF_JIT=y          # JIT compiler for BPF (huge perf boost)
CONFIG_NET_ACT_BPF=m      # TC BPF actions
CONFIG_PTP_1588_CLOCK=y   # PTP support (for I210 timestamping)
CONFIG_VFIO=m             # For SR-IOV passthrough to VMs
```

Appendix C

Key Source Files Quick Reference

File Path	What to Look For
igb/igb_main.c	Everything: probe, open, tx, rx, NAPI, XDP, net-dev_ops
igb/igb.h	All data structures: igb_adapter, igb_ring, igb_q_vector
igb/igb_xsk.c	AF_XDP zero-copy: igb_clean_rx_irq_zc, igb_xmit_zc
igb/igb_ethtool.c	Diagnostics: ring config, stats, self-test
igb/igb_ptp.c	Hardware timestamping and PTP clock
net/xdp/xsk.c	Core AF_XDP socket implementation
net/core/xdp.c	XDP core: redirect, memory management
kernel/bpf/verifier.c	BPF verifier (validates XDP programs)
samples/bpf/xdpsock_user.c	Reference AF_XDP application

Appendix D

Further Reading

- Linux kernel documentation: `Documentation/networking/device_drivers/ethernet/intel/igb.rst`
- AF_XDP documentation: `Documentation/networking/af_xdp.rst`
- NAPI documentation: `Documentation/networking/napi.rst`
- XDP tutorial: `https://github.com/xdp-project/xdp-tutorial`
- PackageCloud blog: "Monitoring and Tuning the Linux Networking Stack" (uses igb as example)
- Intel I350 datasheet (hardware register reference)
- Original XDP patch: commit `9cbc948b5a20`
- AF_XDP zero-copy patches: search `https://lore.kernel.org` for "igb zero copy"