Understanding Linux Network Device Driver Code

Julia Lawall
Inria/LIP6
February 19, 2015

Overview

A network device driver connects the OS, and hence applications, to a network device.

This talk:

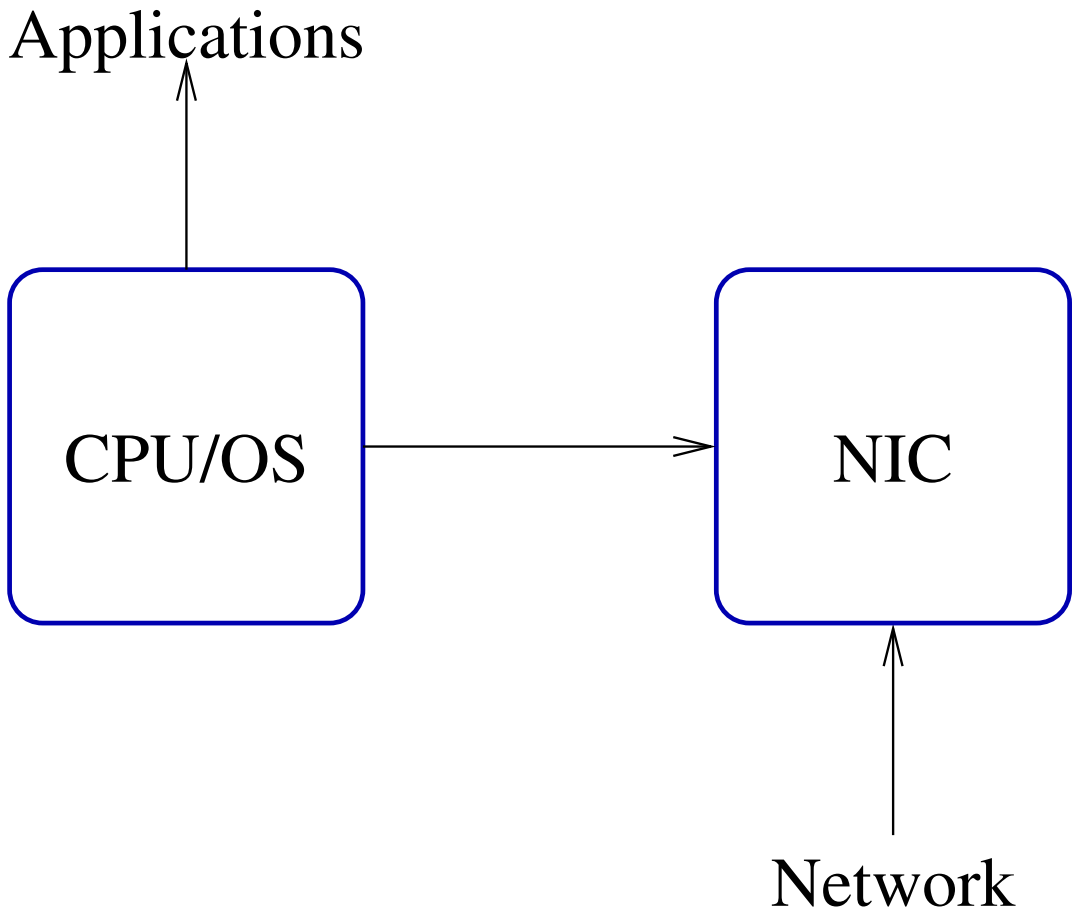
- How does the OS interact with a network device?
- How has the code changed as network speeds have gotten faster?
- Focus on packet reception.

All code from Linux 3.15, released June 2014.

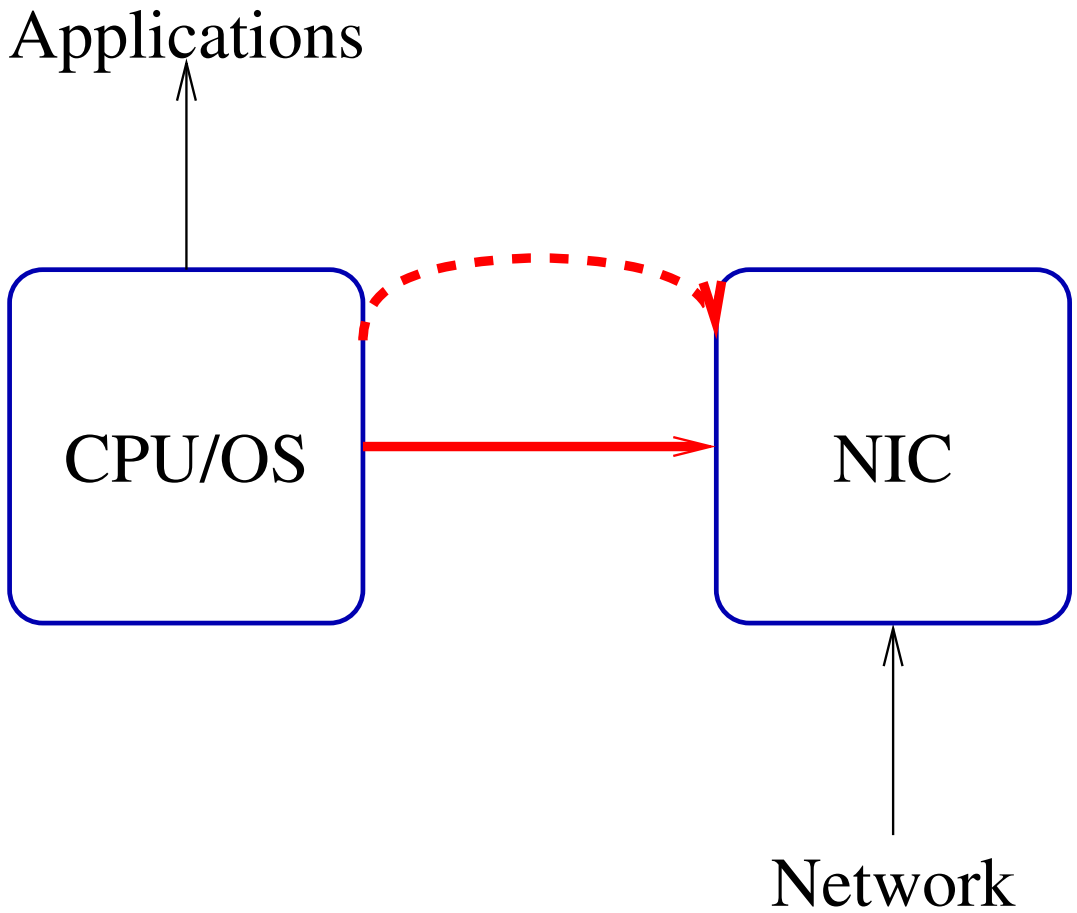
The role of a network driver (packet reception)

- Packets arrive from the network.
- They are received by a Network Interface Controller (NIC).
  - Specialized in receiving packets from networks.
  - May do some preprocessing, eg filtering.
- CPU/OS retrieves packets from the NIC.
  - May do some more processing, eg filtering.
- CPU/OS sends packets up the network stack.

The role of a network driver, in pictures



Our focus



Our starting point - 3c509.c

- Driver for the 3Com EtherLink III Series Ethercard
- drivers/net/3c509.c exists in Linux 1.0.
- First written in 1993, by Donald Becker
- Updates in 2002 and 2008.
- Device available on Amazon for $10 - $39.95.

Actions of the driver

1. Check for packets.
   - The OS has no idea when packets arrive.
   - The device informs the OS via an interrupt.
2. Make packets available in memory.
3. Send packet up the network stack.

- Check for all sorts of error conditions along the way.

The 3c509 interrupt handler

```
/* The EL3 interrupt handler. */
static irqreturn_t el3_interrupt(int irq, void *dev_id) {
  struct net_device *dev = dev_id;
  ...
  spin_lock(&lp->lock);
  ioaddr = dev->base_addr;
  ...
  while ((status = inw(ioaddr + EL3_STATUS)) & (IntLatch | RxComplete | StatsFull)) {
    if (status & RxComplete)
      el3_rx(dev);
    if (status & TxAvailable) { ... }
    if (status & (AdapterFailure | RxEarly | StatsFull | TxComplete)) {
      /* Handle all uncommon interrupts. */
      ...
    }
    ...
    outw(AckIntr | IntReq | IntLatch, ioaddr + EL3_CMD); /* Ack IRQ */
  }
  ...
  spin_unlock(&lp->lock);
  return IRQ_HANDLED;
}
```

The 3c509 interrupt handler

```
/* The EL3 interrupt handler. */
static irqreturn_t el3_interrupt(int irq, void *dev_id) {
  struct net_device *dev = dev_id;
  ...
  spin_lock(&lp->lock);
  ioaddr = dev->base_addr;
  ...
  while ((status = inw(ioaddr + EL3_STATUS)) & (IntLatch | RxComplete | StatsFull)) {
    if (status & RxComplete)
      el3_rx(dev);
    if (status & TxAvailable) { ... }
    if (status & (AdapterFailure | RxEarly | StatsFull | TxComplete)) {
      /* Handle all uncommon interrupts. */
      ...
    }
    ...
    outw(AckIntr | IntReq | IntLatch, ioaddr + EL3_CMD); /* Ack IRQ */
  }
  ...
  spin_unlock(&lp->lock);
  return IRQ_HANDLED;
}
```

The 3c509 interrupt handler

```
/* The EL3 interrupt handler. */
static irqreturn_t el3_interrupt(int irq, void *dev_id) {
  struct net_device *dev = dev_id;
  ...
  spin_lock(&lp->lock);
  ioaddr = dev->base_addr;
  ...
  while ((status = inw(ioaddr + EL3_STATUS)) & (IntLatch | RxComplete | StatsFull)) {
    if (status & RxComplete)
      el3_rx(dev);
    if (status & TxAvailable) { ... }
    if (status & (AdapterFailure | RxEarly | StatsFull | TxComplete)) {
      /* Handle all uncommon interrupts. */
      ...
    }
    ...
    outw(AckIntr | IntReq | IntLatch, ioaddr + EL3_CMD); /* Ack IRQ */
  }
  ...
  spin_unlock(&lp->lock);
  return IRQ_HANDLED;
}
```

el3_rx

```
static int el3_rx(struct net_device *dev) {
  int ioaddr = dev->base_addr;
  short rx_status;
  while ((rx_status = inw(ioaddr + RX_STATUS)) > 0) {
    if (rx_status & 0x4000) { /* Error, update stats. */
      ...
    } else {
      short pkt_len = rx_status & 0x7ff;
      struct sk_buff *skb;
      skb = netdev_alloc_skb(dev, pkt_len + 5);
      if (skb != NULL) {
        ...
        continue;
      }
      outw(RxDiscard, ioaddr + EL3_CMD);
      dev->stats.rx_dropped++;
    }
    inw(ioaddr + EL3_STATUS); /* Delay to discard packet. */
    while (inw(ioaddr + EL3_STATUS) & 0x1000) ;
  }
  return 0;
}
```

el3_rx

```
static int el3_rx(struct net_device *dev) {
  int ioaddr = dev->base_addr;
  short rx_status;
  while ((rx_status = inw(ioaddr + RX_STATUS)) > 0) {
    if (rx_status & 0x4000) { /* Error, update stats. */
      ...
    } else {
      short pkt_len = rx_status & 0x7ff;
      struct sk_buff *skb;
      skb = netdev_alloc_skb(dev, pkt_len + 5);
      if (skb != NULL) {
        ...
        continue;
      }
      outw(RxDiscard, ioaddr + EL3_CMD);
      dev->stats.rx_dropped++;
    }
    inw(ioaddr + EL3_STATUS); /* Delay to discard packet. */
    while (inw(ioaddr + EL3_STATUS) & 0x1000) ;
  }
  return 0;
}
```

el3_rx

```
static int el3_rx(struct net_device *dev) {
  int ioaddr = dev->base_addr;
  short rx_status;
  while ((rx_status = inw(ioaddr + RX_STATUS)) > 0) {
    if (rx_status & 0x4000) { /* Error, update stats. */
      ...
    } else {
      short pkt_len = rx_status & 0x7ff;
      struct sk_buff *skb;
      skb = netdev_alloc_skb(dev, pkt_len + 5);
      if (skb != NULL) {
        ...
        continue;
      }
      outw(RxDiscard, ioaddr + EL3_CMD);
      dev->stats.rx_dropped++;
    }
    inw(ioaddr + EL3_STATUS); /* Delay to discard packet. */
    while (inw(ioaddr + EL3_STATUS) & 0x1000) ;
  }
  return 0;
}
```

Packet processing

```
/* Align IP on 16 byte */
skb_reserve(skb, 2);
/* 'skb->data' points to the start of sk_buff data area */
insl(ioaddr + RX_FIFO, skb_put(skb,pkt_len), (pkt_len + 3) >> 2);
/* Pop top Rx packet */
outw(RxDiscard, ioaddr + EL3_CMD);
skb->protocol = eth_type_trans(skb,dev);
netif_rx(skb);
dev->stats.rx_bytes += pkt_len;
dev->stats.rx_packets++;
```

insl

From include/asm-generic/io.h:

```
static inline void
insl(unsigned long addr, void *buffer, int count) {
  if (count) {
    u32 *buf = buffer;
    do {
      u32 x = __raw_readl(addr + PCI_IOBASE);
      *buf++ = x;
    } while (--count);
  }
}
```

Assessment

Concepts:

- On interrupt, OS reads multiple packets, if available.
- OS communicates with the device via registers.
- Limited by device buffer size.
- More interrupts may come soon, depending on network speed.

Implementation:

- Specific instructions for accessing device memory (`inw`, `outw`, etc.).
- Key APIs:
  - `netdev_alloc_skb`
  - `skb_reserve/put`
  - `netif_rx`

Assessment

Concepts:

- On interrupt, OS reads multiple packets, if available.
- OS communicates with the device via registers.
- Limited by device buffer size.
- More interrupts may come soon, depending on network speed.

Implementation:

- Specific instructions for accessing device memory (`inw`, `outw`, etc.).
- Key APIs:
  - `netdev_alloc_skb`
  - `skb_reserve/put`
  - `netif_rx`

How to scale up?

Aeroflex Gaisler GRETH 10/100/1G Ethernet (2005 –)

```
static irqreturn_t greth_interrupt(int irq, void *dev_id) {
  struct net_device *dev = dev_id;
  struct greth_private *greth;
  u32 status, ctrl;
  irqreturn_t retval = IRQ_NONE;

  ...
  /* Get the interrupt events that caused us to be here. */
  status = GRETH_REGLOAD(greth->regs->status);
  ...
  /* Handle rx and tx interrupts through poll */
  if (((status & (GRETH_INT_RE | GRETH_INT_RX)) && (ctrl & GRETH_RXI)) ||
      ((status & (GRETH_INT_TE | GRETH_INT_TX)) && (ctrl & GRETH_TXI))) {
    retval = IRQ_HANDLED;

    /* Disable interrupts and schedule poll() */
    greth_disable_irqs(greth);
    napi_schedule(&greth->napi);
  }
  ...
  return retval;
}
```

Aeroflex Gaisler GRETH 10/100/1G Ethernet (2005 –)

```
static irqreturn_t greth_interrupt(int irq, void *dev_id) {
  struct net_device *dev = dev_id;
  struct greth_private *greth;
  u32 status, ctrl;
  irqreturn_t retval = IRQ_NONE;

  ...
  /* Get the interrupt events that caused us to be here. */
  status = GRETH_REGLOAD(greth->regs->status);
  ...
  /* Handle rx and tx interrupts through poll */
  if (((status & (GRETH_INT_RE | GRETH_INT_RX)) && (ctrl & GRETH_RXI)) ||
      ((status & (GRETH_INT_TE | GRETH_INT_TX)) && (ctrl & GRETH_TXI))) {
    retval = IRQ_HANDLED;

    /* Disable interrupts and schedule poll() */
    greth_disable_irqs(greth);
    napi_schedule(&greth->napi);
  }
  ...
  return retval;
}
```

Issues

Problem:

- Modern devices process packets faster.
- Interrupting the CPU for each packet degrades CPU performance.
- Processing lots of packets in an interrupt handler delays other things.

Issues

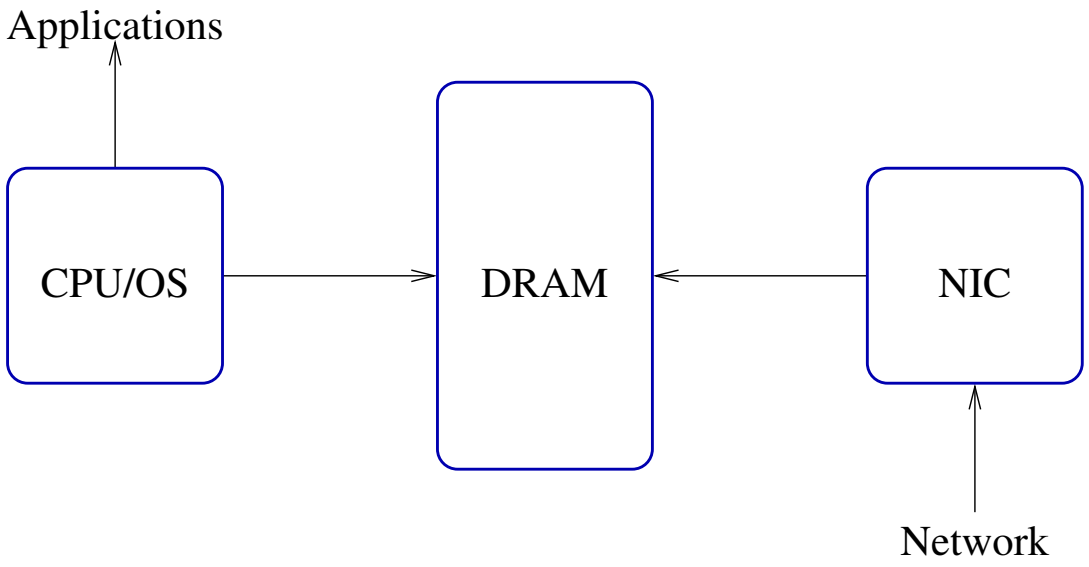
Problem:

- Modern devices process packets faster.
- Interrupting the CPU for each packet degrades CPU performance.
- Processing lots of packets in an interrupt handler delays other things.

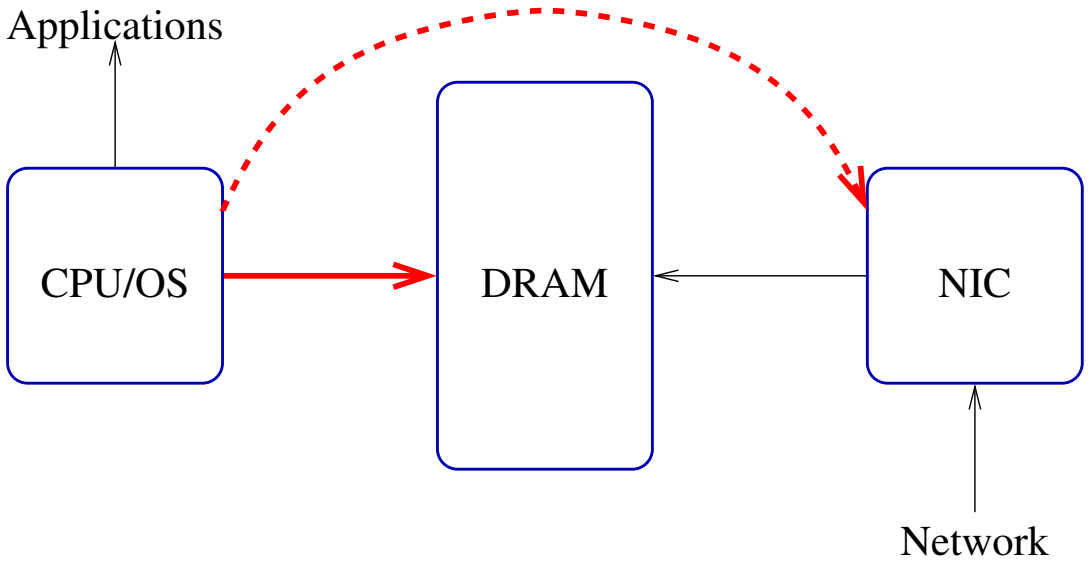
Solution: NAPI (New API)

- Packet processing via polling.
- Minimal requirements on the device:
  - Disabling interrupts.
  - Ring buffer, in main memory

Ring buffer



Ring buffer



Greth NAPI poll function

```
static int greth_poll(struct napi_struct *napi, int budget) {
  ... /* various declarations */
restart_txrx_poll:
  ... /* network transmission code */
  if (greth->gbit_mac) {
    work_done += greth_rx_gbit(greth->netdev, budget - work_done);
  } else {
    work_done += greth_rx(greth->netdev, budget - work_done);
  }
  if (work_done < budget) {
    spin_lock_irqsave(&greth->devlock, flags);
    ... /* interrupt processing */
    if (GRETH_REGLOAD(greth->regs->status) & mask) {
      GRETH_REGSAVE(greth->regs->control, ctrl);
      spin_unlock_irqrestore(&greth->devlock, flags);
      goto restart_txrx_poll;
    } else {
      __napi_complete(napi);
      spin_unlock_irqrestore(&greth->devlock, flags);
    }
  }
  return work_done;
}
```

Greth NAPI poll function

```
static int greth_poll(struct napi_struct *napi, int budget) {
  ... /* various declarations */
restart_txrx_poll:
  ... /* network transmission code */
  if (greth->gbit_mac) {
    work_done += greth_rx_gbit(greth->netdev, budget - work_done);
  } else {
    work_done += greth_rx(greth->netdev, budget - work_done);
  }
  if (work_done < budget) {
    spin_lock_irqsave(&greth->devlock, flags);
    ... /* interrupt processing */
    if (GRETH_REGLOAD(greth->regs->status) & mask) {
      GRETH_REGSAVE(greth->regs->control, ctrl);
      spin_unlock_irqrestore(&greth->devlock, flags);
      goto restart_txrx_poll;
    } else {
      __napi_complete(napi);
      spin_unlock_irqrestore(&greth->devlock, flags);
    }
  }
  return work_done;
}
```

Greth NAPI poll function

```
static int greth_poll(struct napi_struct *napi, int budget) {
  ... /* various declarations */
restart_txrx_poll:
  ... /* network transmission code */
  if (greth->gbit_mac) {
    work_done += greth_rx_gbit(greth->netdev, budget - work_done);
  } else {
    work_done += greth_rx(greth->netdev, budget - work_done);
  }
  if (work_done < budget) {
    spin_lock_irqsave(&greth->devlock, flags);
    ... /* interrupt processing */
    if (GRETH_REGLOAD(greth->regs->status) & mask) {
      GRETH_REGSAVE(greth->regs->control, ctrl);
      spin_unlock_irqrestore(&greth->devlock, flags);
      goto restart_txrx_poll;
    } else {
      __napi_complete(napi);
      spin_unlock_irqrestore(&greth->devlock, flags);
    }
  }
  return work_done;
}
```

Megabit ethernet (10, 100)

```
static int greth_rx(struct net_device *dev, int limit) {
  ... /* decls and get private structure */
  for (count = 0; count < limit; ++count) {
    bdp = greth->rx_bd_base + greth->rx_cur;
    GRETH_REGSAVE(greth->regs->status, GRETH_INT_RE | GRETH_INT_RX);
    status = greth_read_bd(&bdp->stat);
    dma_addr = greth_read_bd(&bdp->addr);
    bad = 0;
    ... /* error checking, may set bad to 1 */
    if (unlikely(bad)) dev->stats.rx_errors++;
    else { ... } /* receive packet */
    status = GRETH_BD_EN | GRETH_BD_IE;
    ... /* adjust status */
    greth_write_bd(&bdp->stat, status);
    dma_sync_single_for_device(greth->dev, dma_addr, MAX_FRAME_SIZE,
                               DMA_FROM_DEVICE);
    spin_lock_irqsave(&greth->devlock, flags); /* save from XMIT */
    greth_enable_rx(greth);
    spin_unlock_irqrestore(&greth->devlock, flags);
    greth->rx_cur = NEXT_RX(greth->rx_cur);
  }
  return count;
}
```

Megabit ethernet (10, 100)

```
static int greth_rx(struct net_device *dev, int limit) {
  ... /* decls and get private structure */
  for (count = 0; count < limit; ++count) {
    bdp = greth->rx_bd_base + greth->rx_cur;
    GRETH_REGSAVE(greth->regs->status, GRETH_INT_RE | GRETH_INT_RX);
    status = greth_read_bd(&bdp->stat);
    dma_addr = greth_read_bd(&bdp->addr);
    bad = 0;
    ... /* error checking, may set bad to 1 */
    if (unlikely(bad)) dev->stats.rx_errors++;
    else { ... } /* receive packet */
    status = GRETH_BD_EN | GRETH_BD_IE;
    ... /* adjust status */
    greth_write_bd(&bdp->stat, status);
    dma_sync_single_for_device(greth->dev, dma_addr, MAX_FRAME_SIZE,
                               DMA_FROM_DEVICE);
    spin_lock_irqsave(&greth->devlock, flags); /* save from XMIT */
    greth_enable_rx(greth);
    spin_unlock_irqrestore(&greth->devlock, flags);
    greth->rx_cur = NEXT_RX(greth->rx_cur);
  }
  return count;
}
```

Megabit ethernet (10, 100)

```
static int greth_rx(struct net_device *dev, int limit) {
  ... /* decls and get private structure */
  for (count = 0; count < limit; ++count) {
    bdp = greth->rx_bd_base + greth->rx_cur;
    GRETH_REGSAVE(greth->regs->status, GRETH_INT_RE | GRETH_INT_RX);
    status = greth_read_bd(&bdp->stat);
    dma_addr = greth_read_bd(&bdp->addr);
    bad = 0;
    ... /* error checking, may set bad to 1 */
    if (unlikely(bad)) dev->stats.rx_errors++;
    else { ... } /* receive packet */
    status = GRETH_BD_EN | GRETH_BD_IE;
    ... /* adjust status */
    greth_write_bd(&bdp->stat, status);
    dma_sync_single_for_device(greth->dev, dma_addr, MAX_FRAME_SIZE,
                               DMA_FROM_DEVICE);
    spin_lock_irqsave(&greth->devlock, flags); /* save from XMIT */
    greth_enable_rx(greth);
    spin_unlock_irqrestore(&greth->devlock, flags);
    greth->rx_cur = NEXT_RX(greth->rx_cur);
  }
  return count;
}
```

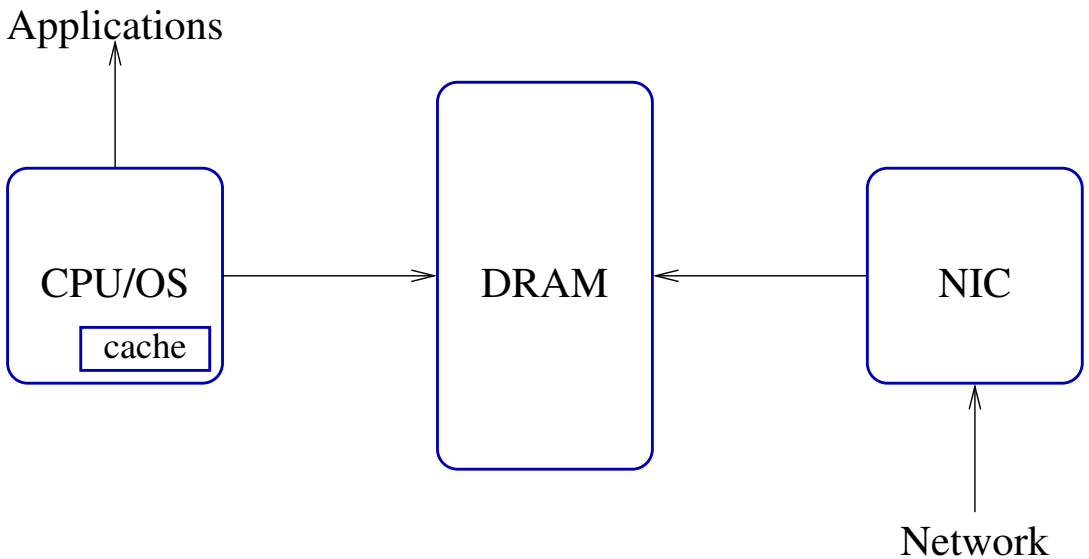
Megabit ethernet (10, 100)

```
if (unlikely(bad))
  dev->stats.rx_errors++;
else {
  pkt_len = status & GRETH_BD_LEN;
  skb = netdev_alloc_skb(dev, pkt_len + NET_IP_ALIGN);
  if (unlikely(skb == NULL))
    dev->stats.rx_dropped++;
  else {
    skb_reserve(skb, NET_IP_ALIGN);
    dma_sync_single_for_cpu(greth->dev, dma_addr, pkt_len, DMA_FROM_DEVICE);
    memcpy(skb_put(skb, pkt_len), phys_to_virt(dma_addr), pkt_len);
    skb->protocol = eth_type_trans(skb, dev);
    dev->stats.rx_bytes += pkt_len;
    dev->stats.rx_packets++;
    netif_receive_skb(skb);
  }
}
```

Megabit ethernet (10, 100)

```
if (unlikely(bad))
  dev->stats.rx_errors++;
else {
  pkt_len = status & GRETH_BD_LEN;
  skb = netdev_alloc_skb(dev, pkt_len + NET_IP_ALIGN);
  if (unlikely(skb == NULL))
    dev->stats.rx_dropped++;
  else {
    skb_reserve(skb, NET_IP_ALIGN);
    dma_sync_single_for_cpu(greth->dev, dma_addr, pkt_len, DMA_FROM_DEVICE);
    memcpy(skb_put(skb, pkt_len), phys_to_virt(dma_addr), pkt_len);
    skb->protocol = eth_type_trans(skb, dev);
    dev->stats.rx_bytes += pkt_len;
    dev->stats.rx_packets++;
    netif_receive_skb(skb);
  }
}
```

Architecture, with cache



Megabit ethernet (10, 100)

```
if (unlikely(bad))
  dev->stats.rx_errors++;
else {
  pkt_len = status & GRETH_BD_LEN;
  skb = netdev_alloc_skb(dev, pkt_len + NET_IP_ALIGN);
  if (unlikely(skb == NULL))
    dev->stats.rx_dropped++;
  else {
    skb_reserve(skb, NET_IP_ALIGN);
    dma_sync_single_for_cpu(greth->dev, dma_addr, pkt_len, DMA_FROM_DEVICE);
    memcpy(skb_put(skb, pkt_len), phys_to_virt(dma_addr), pkt_len);
    skb->protocol = eth_type_trans(skb, dev);
    dev->stats.rx_bytes += pkt_len;
    dev->stats.rx_packets++;
    netif_receive_skb(skb);
  }
}
```

Assessment

Concepts:

- Fixed buffers of maximum packet size are placed in memory during initialization.
- Synchronization needed to allow CPU to read, and to allow device to write again.
- Packet data is copied from fixed buffer to skb.
- Up to a quota of buffers are picked up.

Implementation:

- Key APIs:
  - `napi_schedule`
  - `dma_sync_single_for_cpu/device`
  - `netif_receive_skb`

Assessment

Concepts:

- Fixed buffers of maximum packet size are placed in memory during initialization.
- Synchronization needed to allow CPU to read, and to allow device to write again.
- Packet data is copied from fixed buffer to skb.
- Up to a quota of buffers are picked up.

Implementation:

- Key APIs:
  - `napi_schedule`
  - `dma_sync_single_for_cpu/device`
  - `netif_receive_skb`

Copying packets is expensive

Packet copying

Megabit case

- Ring buffer is allocated once in memory.
- Device copies into it, CPU/OS copies out of it.

Eliminating copies

- Device copies into packet-sized buffers.
- CPU/OS picks up a buffer, sends it upward, and drops in a new one.
- Copying remains, but not in CPU time.

Gigabit ethernet

```
static int greth_rx_gbit(struct net_device *dev, int limit) {
  ... /* decls and get private structure */
  for (count = 0; count < limit; ++count) {
    bdp = greth->rx_bd_base + greth->rx_cur;
    skb = greth->rx_skbuff[greth->rx_cur];
    GRETH_REGSAVE(greth->regs->status, GRETH_INT_RE | GRETH_INT_RX);
    status = greth_read_bd(&bdp->stat);
    bad = 0;
    ... /* error checking, may set bad to 1 */
    if (!bad && ...) { ... } /* receive packet */
    else { ... }
    status = GRETH_BD_EN | GRETH_BD_IE;
    ... /* adjust status */
    greth_write_bd(&bdp->stat, status);
    spin_lock_irqsave(&greth->devlock, flags);
    greth_enable_rx(greth);
    spin_unlock_irqrestore(&greth->devlock, flags);
    greth->rx_cur = NEXT_RX(greth->rx_cur);
  }
  return count;
}
```

Gigabit ethernet

```
static int greth_rx_gbit(struct net_device *dev, int limit) {
  ... /* decls and get private structure */
  for (count = 0; count < limit; ++count) {
    bdp = greth->rx_bd_base + greth->rx_cur;
    skb = greth->rx_skbuff[greth->rx_cur];
    GRETH_REGSAVE(greth->regs->status, GRETH_INT_RE | GRETH_INT_RX);
    status = greth_read_bd(&bdp->stat);
    bad = 0;
    ... /* error checking, may set bad to 1 */
    if (!bad && ...) { ... } /* receive packet */
    else { ... }
    status = GRETH_BD_EN | GRETH_BD_IE;
    ... /* adjust status */
    greth_write_bd(&bdp->stat, status);
    spin_lock_irqsave(&greth->devlock, flags);
    greth_enable_rx(greth);
    spin_unlock_irqrestore(&greth->devlock, flags);
    greth->rx_cur = NEXT_RX(greth->rx_cur);
  }
  return count;
}
```

Gigabit ethernet

```
/* Allocate new skb to replace current, not needed if the current skb can be reused */
if (!bad && (newskb=netdev_alloc_skb(dev, MAX_FRAME_SIZE + NET_IP_ALIGN))) {
    skb_reserve(newskb, NET_IP_ALIGN);
    dma_addr = dma_map_single(greth->dev, newskb->data,
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
    if (!dma_mapping_error(greth->dev, dma_addr)) { /* Process the incoming frame. */
        pkt_len = status & GRETH_BD_LEN;
        dma_unmap_single(greth->dev, greth_read_bd(&bdp->addr),
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
        skb_put(skb, pkt_len);
        ... /* checksum */
        skb->protocol = eth_type_trans(skb, dev);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += pkt_len;
        netif_receive_skb(skb);
        greth->rx_skbuff[greth->rx_cur] = newskb;
        greth_write_bd(&bdp->addr, dma_addr);
    } else {
        dev_kfree_skb(newskb);
        dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
    }
} else
    dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
```

Gigabit ethernet

```
/* Allocate new skb to replace current, not needed if the current skb can be reused */
if (!bad && (newskb=netdev_alloc_skb(dev, MAX_FRAME_SIZE + NET_IP_ALIGN))) {
    skb_reserve(newskb, NET_IP_ALIGN);
    dma_addr = dma_map_single(greth->dev, newskb->data,
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
    if (!dma_mapping_error(greth->dev, dma_addr)) { /* Process the incoming frame. */
        pkt_len = status & GRETH_BD_LEN;
        dma_unmap_single(greth->dev, greth_read_bd(&bdp->addr),
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
        skb_put(skb, pkt_len);
        ... /* checksum */
        skb->protocol = eth_type_trans(skb, dev);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += pkt_len;
        netif_receive_skb(skb);
        greth->rx_skbuff[greth->rx_cur] = newskb;
        greth_write_bd(&bdp->addr, dma_addr);
    } else {
        dev_kfree_skb(newskb);
        dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
    }
} else
    dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
```

Gigabit ethernet

```
/* Allocate new skb to replace current, not needed if the current skb can be reused */
if (!bad && (newskb=netdev_alloc_skb(dev, MAX_FRAME_SIZE + NET_IP_ALIGN))) {
    skb_reserve(newskb, NET_IP_ALIGN);
    dma_addr = dma_map_single(greth->dev, newskb->data,
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
    if (!dma_mapping_error(greth->dev, dma_addr)) { /* Process the incoming frame. */
        pkt_len = status & GRETH_BD_LEN;
        dma_unmap_single(greth->dev, greth_read_bd(&bdp->addr),
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
        skb_put(skb, pkt_len);
        ... /* checksum */
        skb->protocol = eth_type_trans(skb, dev);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += pkt_len;
        netif_receive_skb(skb);
        greth->rx_skbuff[greth->rx_cur] = newskb;
        greth_write_bd(&bdp->addr, dma_addr);
    } else {
        dev_kfree_skb(newskb);
        dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
    }
} else
    dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
```

Gigabit ethernet

```
/* Allocate new skb to replace current, not needed if the current skb can be reused */
if (!bad && (newskb=netdev_alloc_skb(dev, MAX_FRAME_SIZE + NET_IP_ALIGN))) {
    skb_reserve(newskb, NET_IP_ALIGN);
    dma_addr = dma_map_single(greth->dev, newskb->data,
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
    if (!dma_mapping_error(greth->dev, dma_addr)) { /* Process the incoming frame. */
        pkt_len = status & GRETH_BD_LEN;
        dma_unmap_single(greth->dev, greth_read_bd(&bdp->addr),
                              MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
        skb_put(skb, pkt_len);
        ... /* checksum */
        skb->protocol = eth_type_trans(skb, dev);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += pkt_len;
        netif_receive_skb(skb);
        greth->rx_skbuff[greth->rx_cur] = newskb;
        greth_write_bd(&bdp->addr, dma_addr);
    } else {
        dev_kfree_skb(newskb);
        dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
    }
} else
    dev->stats.rx_dropped++; /* reusing current skb, so it is a drop */
```

Assessment

Concepts:

- Each iteration:
  - Makes a new skb.
  - Picks up and sends an old skb.
  - Replaces the old skb with the new one.
- Each skb has two names:
  - CPU name: `newskb`, `skb`
  - Device name: `dma_addr`, `greth_read_bd(&bdp>addr)`

Implementation:

- Key APIs:
  - `dma_map_single`
  - `dma_unmap_single`

Assessment

Concepts:

- Each iteration:
  - Makes a new skb.
  - Picks up and sends an old skb.
  - Replaces the old skb with the new one.
- Each skb has two names:
  - CPU name: `newskb`, `skb`
  - Device name: `dma_addr`, `greth_read_bd(&bdp>addr)`

Implementation:

- Key APIs:
  - `dma_map_single`
  - `dma_unmap_single`

Cache effects of `netdev_alloc_skb`?

Copying vs. non-copying case

Copying:

```
skb = netdev_alloc_skb(dev, pkt_len + NET_IP_ALIGN);
memcpy(skb_put(skb, pkt_len), phys_to_virt(dma_addr), pkt_len);
netif_receive_skb(skb);
```

Non-copying:

```
newskb=netdev_alloc_skb(dev, MAX_FRAME_SIZE + NET_IP_ALIGN);
dma_addr = dma_map_single(greth->dev, newskb->data,
                          MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
dma_unmap_single(greth->dev, greth_read_bd(&bdp->addr),
                          MAX_FRAME_SIZE + NET_IP_ALIGN, DMA_FROM_DEVICE);
netif_receive_skb(skb);
```

Cache issue

Two parts to an skb

- Metadata: initialized by `netdev_alloc_skb`
- Packet data: initialized by `memcpy`/device.

Cache effects

- Metadata consumed by upper layers after reception.
- Non-copying case initializes this metadata far in advance.
- But the device doesn't need the metadata for anything.

Cache issue

Two parts to an skb

- Metadata: initialized by `netdev_alloc_skb`
- Packet data: initialized by `memcpy`/device.

Cache effects

- Metadata consumed by upper layers after reception.
- Non-copying case initializes this metadata far in advance.
- But the device doesn't need the metadata for anything.

build_skb (2011)

- Initialize metadata of a previously allocated buffer.

hip04_rx_poll (added 14.01.2015)

```
while (cnt && !last) {
    buf = priv->rx_buf[priv->rx_head];
    skb = build_skb(buf, priv->rx_buf_size);
    if (unlikely(!skb))
      net_dbg_ratelimited("build skb failed\n");

    dma_unmap_single(&ndev->dev, priv->rx_phys[priv->rx_head],
          RX_BUF_SIZE, DMA_FROM_DEVICE);

    ... /* send skb onward */

    buf = netdev_alloc_frag(priv->rx_buf_size);
    if (!buf)
      goto done;
    phys = dma_map_single(&ndev->dev, buf, RX_BUF_SIZE, DMA_FROM_DEVICE);
    if (dma_mapping_error(&ndev->dev, phys))
      goto done;
    priv->rx_buf[priv->rx_head] = buf;
    priv->rx_phys[priv->rx_head] = phys;
    ...
  }
```

hip04_rx_poll (added 14.01.2015)

```
while (cnt && !last) {
    buf = priv->rx_buf[priv->rx_head];
    skb = build_skb(buf, priv->rx_buf_size);
    if (unlikely(!skb))
      net_dbg_ratelimited("build skb failed\n");

    dma_unmap_single(&ndev->dev, priv->rx_phys[priv->rx_head],
        RX_BUF_SIZE, DMA_FROM_DEVICE);

    ... /* send skb onward */

    buf = netdev_alloc_frag(priv->rx_buf_size);
    if (!buf)
      goto done;
    phys = dma_map_single(&ndev->dev, buf, RX_BUF_SIZE, DMA_FROM_DEVICE);
    if (dma_mapping_error(&ndev->dev, phys))
      goto done;
    priv->rx_buf[priv->rx_head] = buf;
    priv->rx_phys[priv->rx_head] = phys;
    ...
  }
```

hip04_rx_poll (added 14.01.2015)

```
while (cnt && !last) {
    buf = priv->rx_buf[priv->rx_head];
    skb = build_skb(buf, priv->rx_buf_size);
    if (unlikely(!skb))
      net_dbg_ratelimited("build skb failed\n");

    dma_unmap_single(&ndev->dev, priv->rx_phys[priv->rx_head],
          RX_BUF_SIZE, DMA_FROM_DEVICE);

    ... /* send skb onward */

    buf = netdev_alloc_frag(priv->rx_buf_size);
    if (!buf)
      goto done;
    phys = dma_map_single(&ndev->dev, buf, RX_BUF_SIZE, DMA_FROM_DEVICE);
    if (dma_mapping_error(&ndev->dev, phys))
      goto done;
    priv->rx_buf[priv->rx_head] = buf;
    priv->rx_phys[priv->rx_head] = phys;
    ...
  }
```

Summary and issues

Summary:

- Networking code has evolved with networking technology.
- Many generations coexist in the Linux kernel.

Issues:

- Many variations, more or less complex.
- How to isolate similarities and essential differences.
- Tools needed to infer code properties and rationalize the code.