Network drivers

Team Emertxe

Topics

- Linux Network stack and Driver
- Interrupt
- Memory access
- RX/TX
- SKBuffs and NAPI
- Mellanox Adapters Programmer’s Reference Manual

EMERTXE

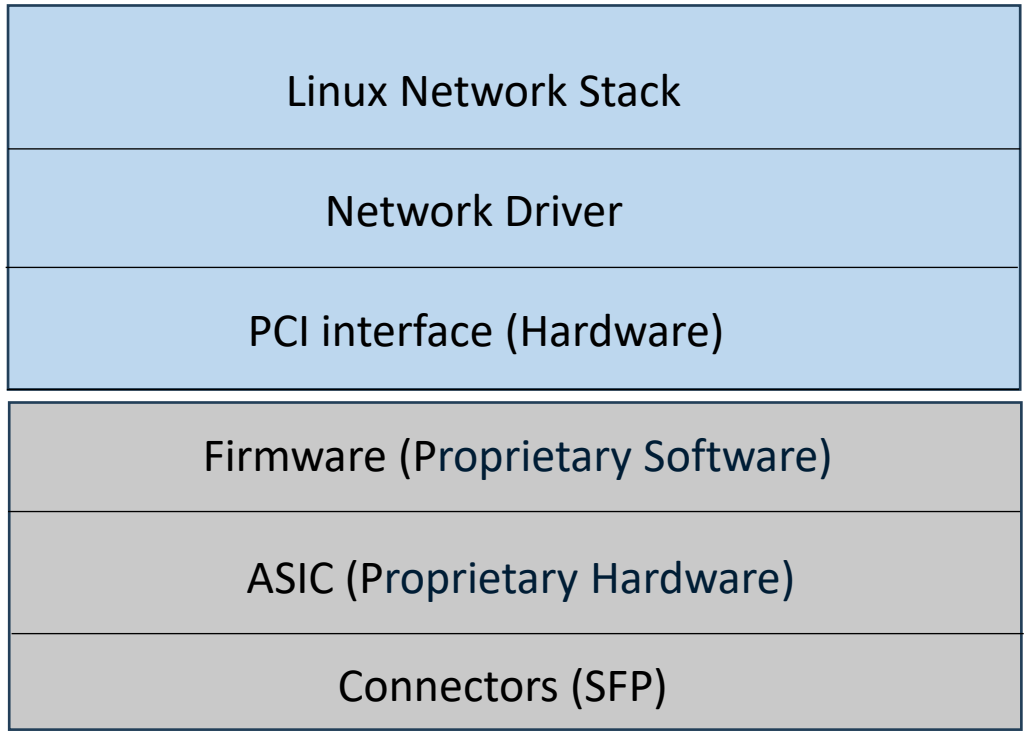
PCI Driver Registration

- int pci_register_driver(struct pci_driver *drv);
- int pci_unregister_driver(struct pci_driver *drv);
- struct pci_driver
  - const char *name
  - const struct pci_dev_id *id_table;
    - PCI_DEVICE(vendor, device);
    - PCI_DEVICE_CLASS(dev_class, dev_class_mask);
  - int (*probe)(pci_dev, id_table);
  - void (*remove)(pci_dev);

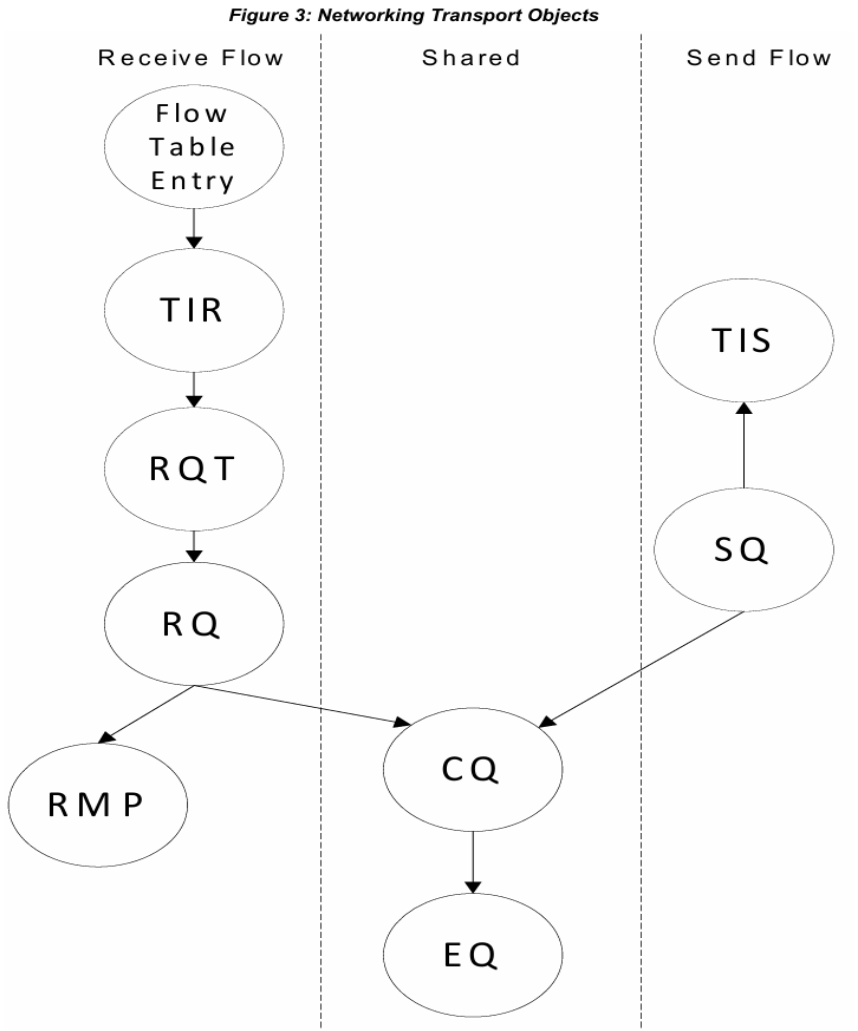
The 'probe' Function

```
int probe(struct pci_dev *d, struct pci_dev_id *id)
{
   /* Initialize the PCI Device */
   ...
   /* Enable the PCI Device */
   pci_enable_device(d);

   ...
   return 0; /* Claimed. Negative for not Claimed */
}
```

Network interface Overview



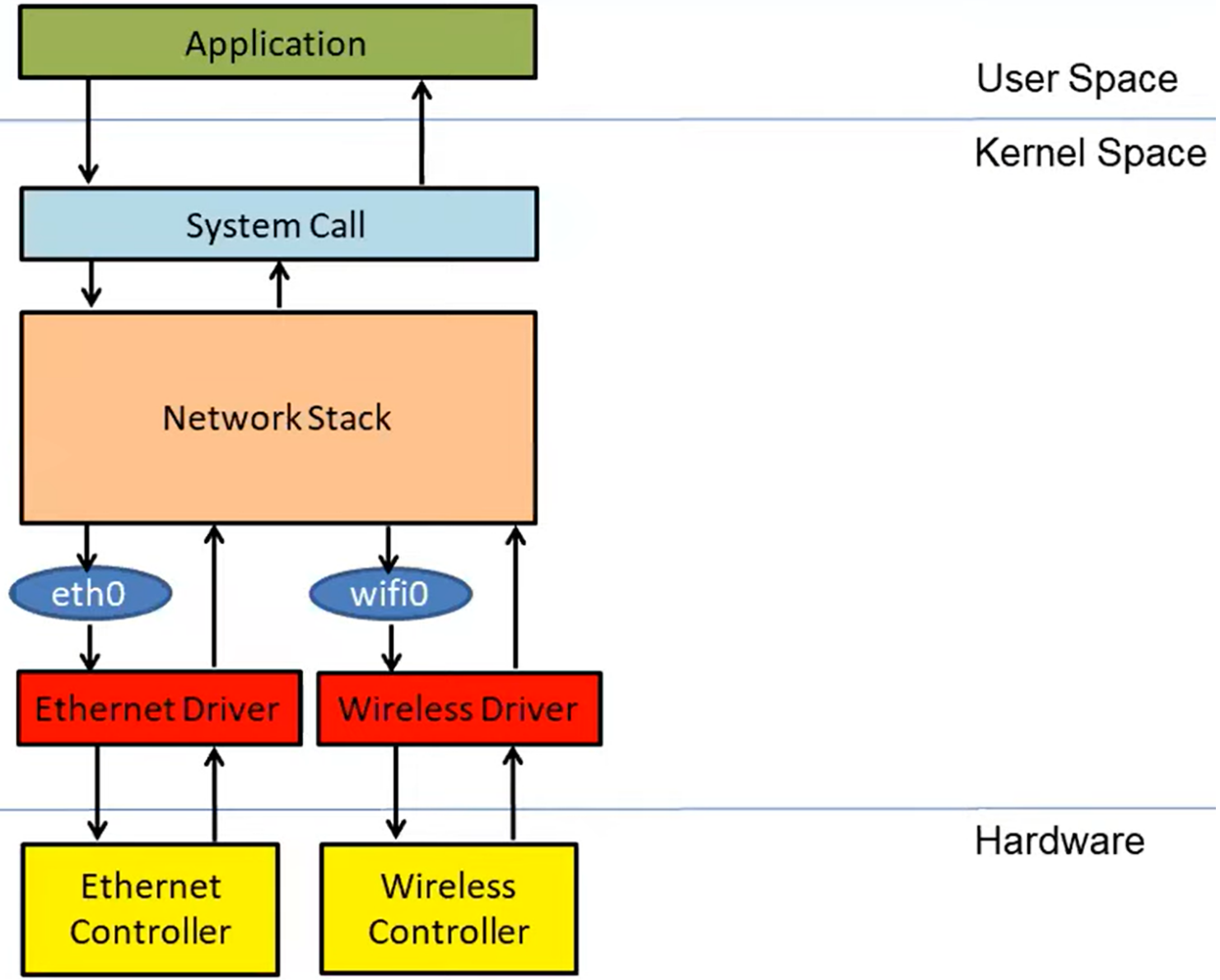
Network interface Overview -Firmware



Network interface Overview

- PCI : Probe, BAR, MSIX
- IEEE 803 standards
- Linux kernel concept: IRQ, threads, memory, etc.
- Linux network stack: sk_buff, net_device_ops, ethtool
- Tools: RSS settings, Wireshark, iPerf, etc.

Network Interface in Network Stack



Linux Network Driver

Device registration with network stack

```
/**
 *          register_netdev      - register a network device
 *          @dev: device to register
 *
 *          Take a completed network device structure and add it to the kernel
 *          interfaces. A %NETDEV_REGISTER message is sent to the netdev notifier
 *          chain. 0 is returned on success. A negative errno code is returned
 *          on a failure to set up the device, or if the name is a duplicate.
 *
 *          This is a wrapper around register_netdevice that takes the rtnl semaphore
 *          and expands the device name if you passed a format string to
 *          alloc_netdev.
 */
int register_netdev(struct net_device *dev)
```

Linux Network Driver

Driver registration

```
static int __init dummy_init_module(void)
{
        int i, err = 0;

        down_write(&pernet_ops_rwsem);
        rtnl_lock();
        err = __rtnl_link_register(&dummy_link_ops);
        if (err < 0)
                goto out;

        for (i = 0; i < numdummies && !err; i++) {
                err = dummy_init_one();
                cond_resched();
        }
        if (err < 0)
                __rtnl_link_unregister(&dummy_link_ops);

out:
        rtnl_unlock();
        up_write(&pernet_ops_rwsem);

        return err;
}
```

Linux Network Driver

```
static int __init dummy_init_one(void)
{
        struct net_device *dev_dummy;
        int err;

        dev_dummy = alloc_netdev(0, "dummy%d", NET_NAME_ENUM, dummy_setup);
        if (!dev_dummy)
                return -ENOMEM;

        dev_dummy->rtnl_link_ops = &dummy_link_ops;
        err = register_netdevice(dev_dummy);
        if (err < 0)
                goto err;
        return 0;

err:
        free_netdev(dev_dummy);
        return err;
}
```

EMERTXE

Linux Network Driver

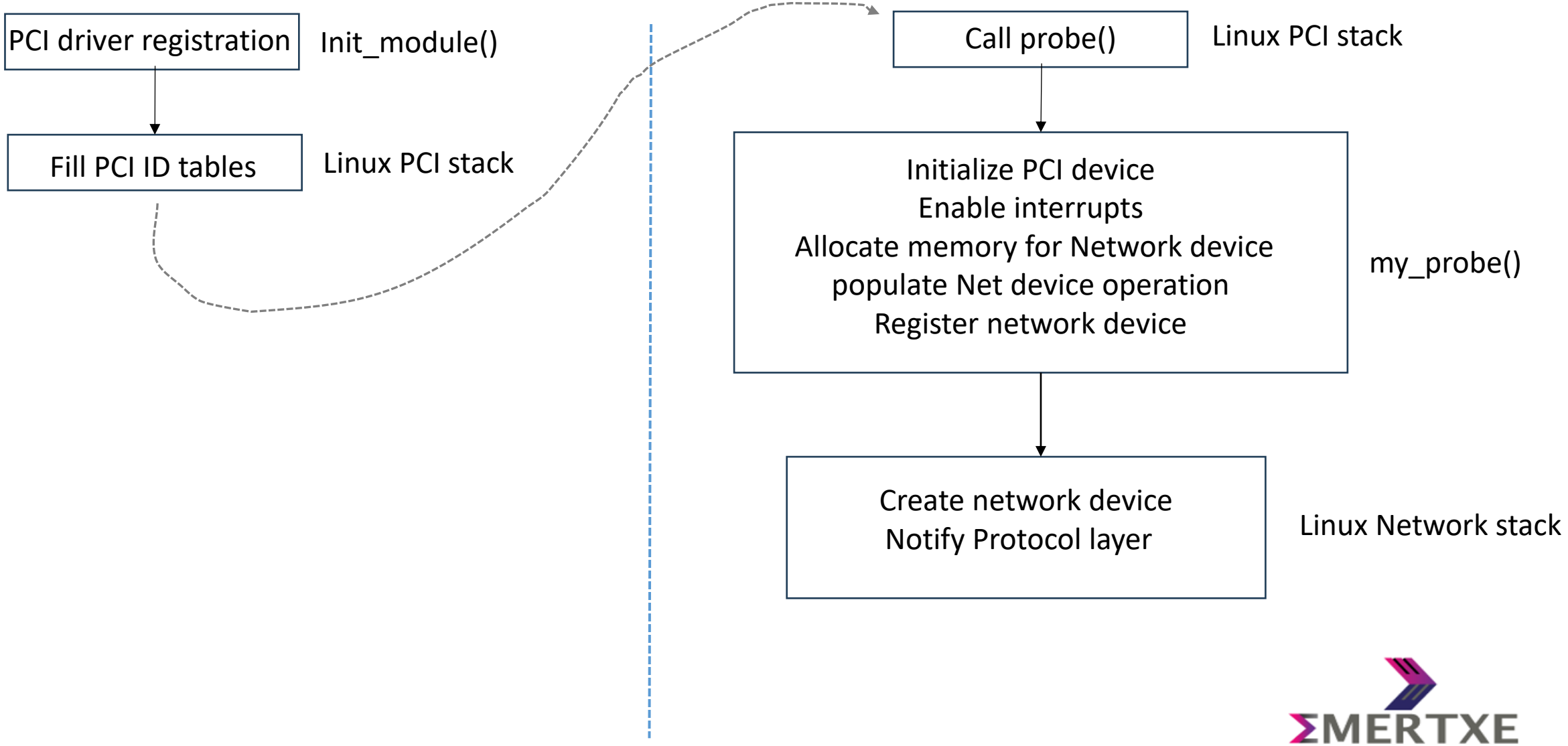
```c
static void dummy_setup(struct net_device *dev)
{
        ether_setup(dev);

        /* Initialize the device structure. */
        dev->netdev_ops = &dummy_netdev_ops;
        dev->ethtool_ops = &dummy_ethtool_ops;
        dev->needs_free_netdev = true;

        /* Fill in device structure with ethernet-generic values. */
        dev->flags |= IFF_NOARP;
        dev->flags &= ~IFF_MULTICAST;
        dev->priv_flags |= IFF_LIVE_ADDR_CHANGE | IFF_NO_QUEUE;
        dev->lltx = true;
        dev->features   |= NETIF_F_SG | NETIF_F_FRAGLIST;
        dev->features   |= NETIF_F_GSO_SOFTWARE;
        dev->features   |= NETIF_F_HW_CSUM | NETIF_F_HIGHDMA;
        dev->features   |= NETIF_F_GSO_ENCAP_ALL;
        dev->hw_features |= dev->features;
        dev->hw_enc_features |= dev->features;
        eth_hw_addr_random(dev);

        dev->min_mtu = 0;
        dev->max_mtu = 0;
}
```

Linux Network Driver

Device registration with network stack

```
static const struct net_device_ops dummy_netdev_ops = {
        .ndo_init               = dummy_dev_init,
        .ndo_start_xmit         = dummy_xmit,
        .ndo_validate_addr      = eth_validate_addr,
        .ndo_set_rx_mode        = set_multicast_list,
        .ndo_set_mac_address    = eth_mac_addr,
        .ndo_get_stats64        = dummy_get_stats64,
        .ndo_change_carrier     = dummy_change_carrier,
};
```

Linux Network Driver



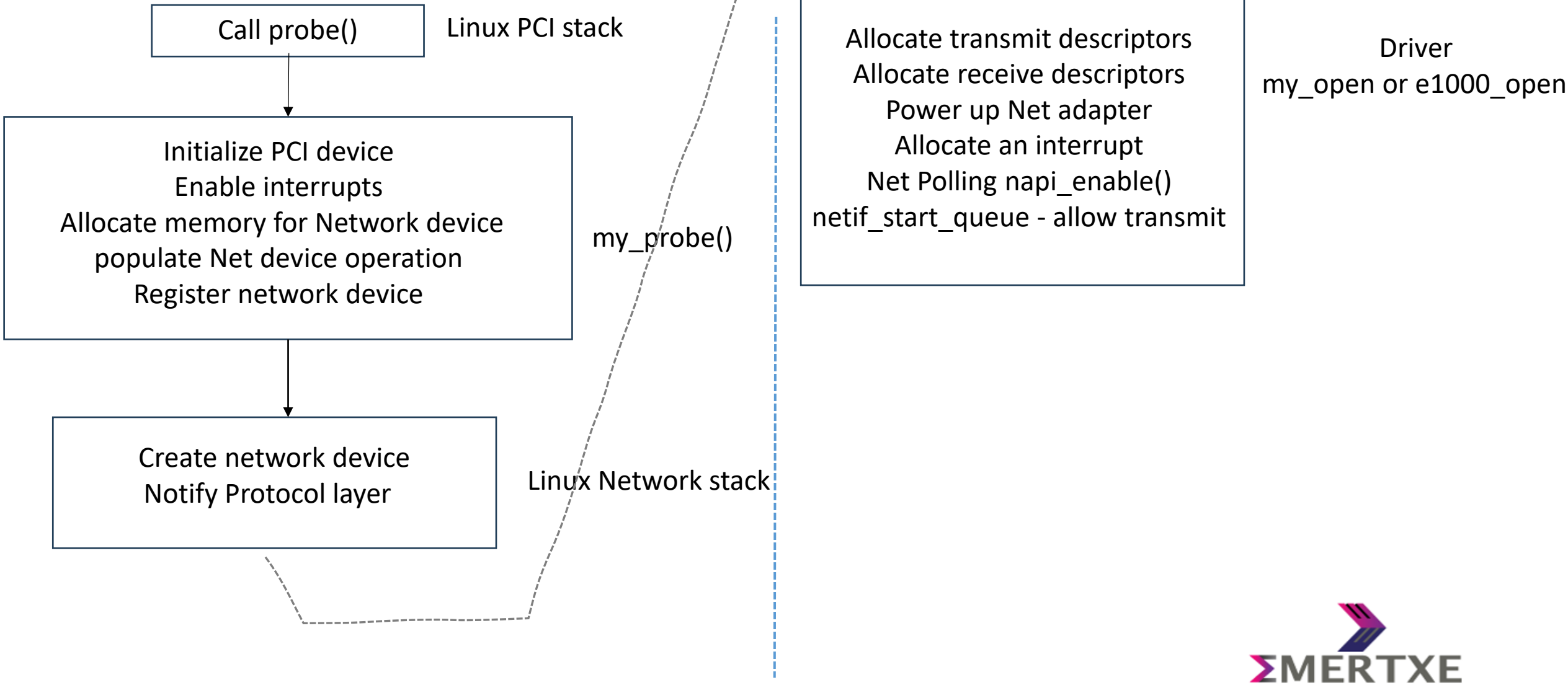
EMERTXE

Linux Network Driver

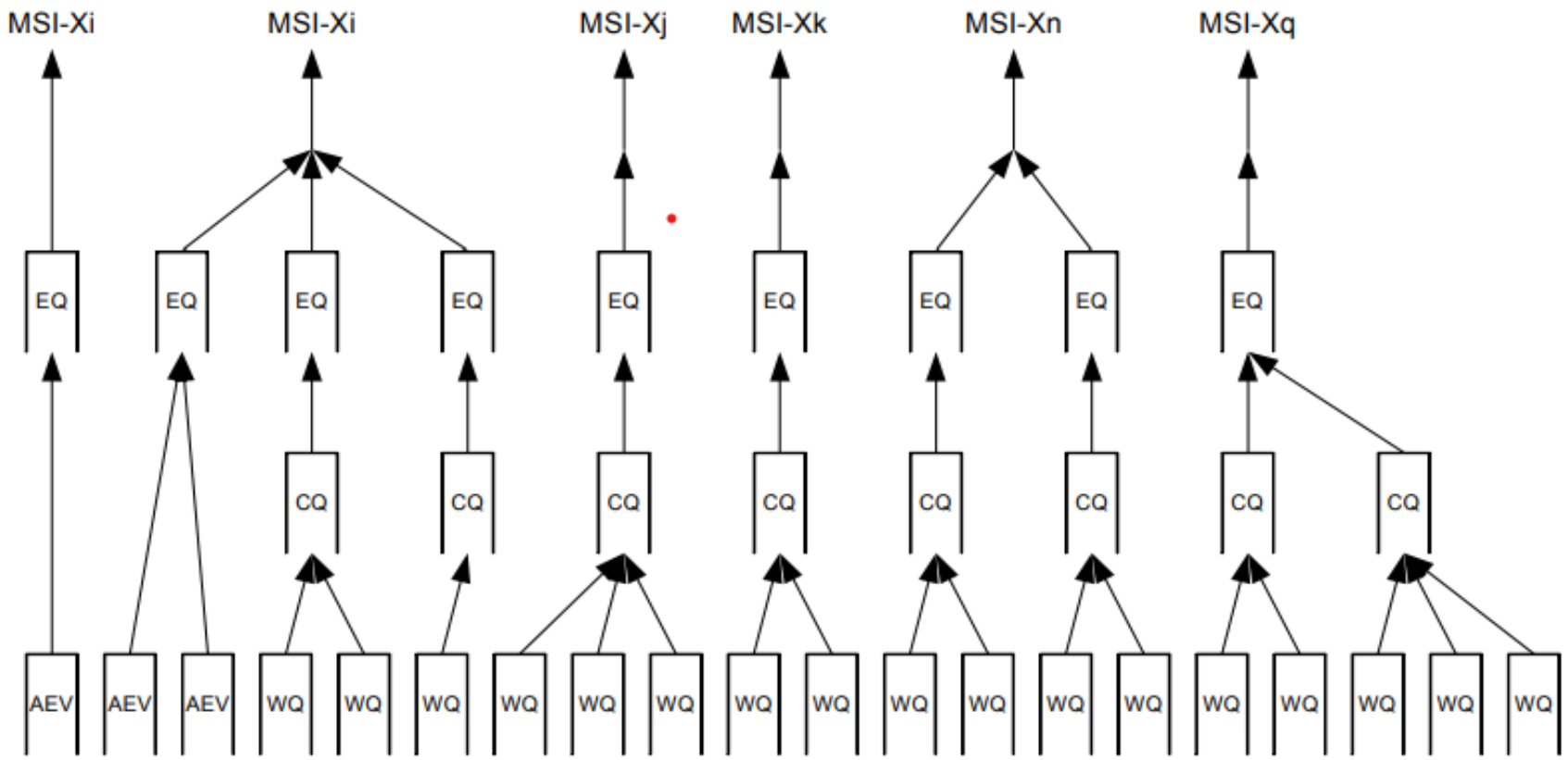
Network device operations

```
824  static const struct net_device_ops e1000_netdev_ops = {
825          .ndo_open               = e1000_open,
826          .ndo_stop               = e1000_close,
827          .ndo_start_xmit         = e1000_xmit_frame,
828          .ndo_set_rx_mode        = e1000_set_rx_mode,
829          .ndo_set_mac_address    = e1000_set_mac,
830          .ndo_tx_timeout         = e1000_tx_timeout,
831          .ndo_change_mtu         = e1000_change_mtu,
832          .ndo_eth_ioctl          = e1000_ioctl,
833          .ndo_validate_addr      = eth_validate_addr,
834          .ndo_vlan_rx_add_vid    = e1000_vlan_rx_add_vid,
835          .ndo_vlan_rx_kill_vid   = e1000_vlan_rx_kill_vid,
836  #ifdef CONFIG_NET_POLL_CONTROLLER
837          .ndo_poll_controller    = e1000_netpoll,
838  #endif
839          .ndo_fix_features       = e1000_fix_features,
840          .ndo_set_features       = e1000_set_features,
841  };
```

Linux Network Driver



PCI MSIX



Linux Network Driver

interrupt

- request_irq()
- request_threaded_irq()
- free_irq()
- disable_irq()
- enable_irq()

Linux Network Driver

interrupt

```
250  static int e1000_request_irq(struct e1000_adapter *adapter)
251  {
252          struct net_device *netdev = adapter->netdev;
253          irq_handler_t handler = e1000_intr;
254          int irq_flags = IRQF_SHARED;
255          int err;
256
257          err = request_irq(adapter->pdev->irq, handler, irq_flags, netdev->name,
258                            netdev);
259          if (err) {
260                  e_err(probe, "Unable to allocate interrupt Error: %d\n", err);
261          }
262
263          return err;
264  }
265
```

Linux Network Driver

Memory Access

- What is the Memory type [Port mapped or Memory mapped]
- Size of memory access [Register read(bytes or words) or DMA]

Port mapped (IO mapped)

Include asm/io.h

readb(), writeb(), readw(), writew() & readl(), writel()

Memory mapped

Normal memory access

Linux Network Driver

Memory Access

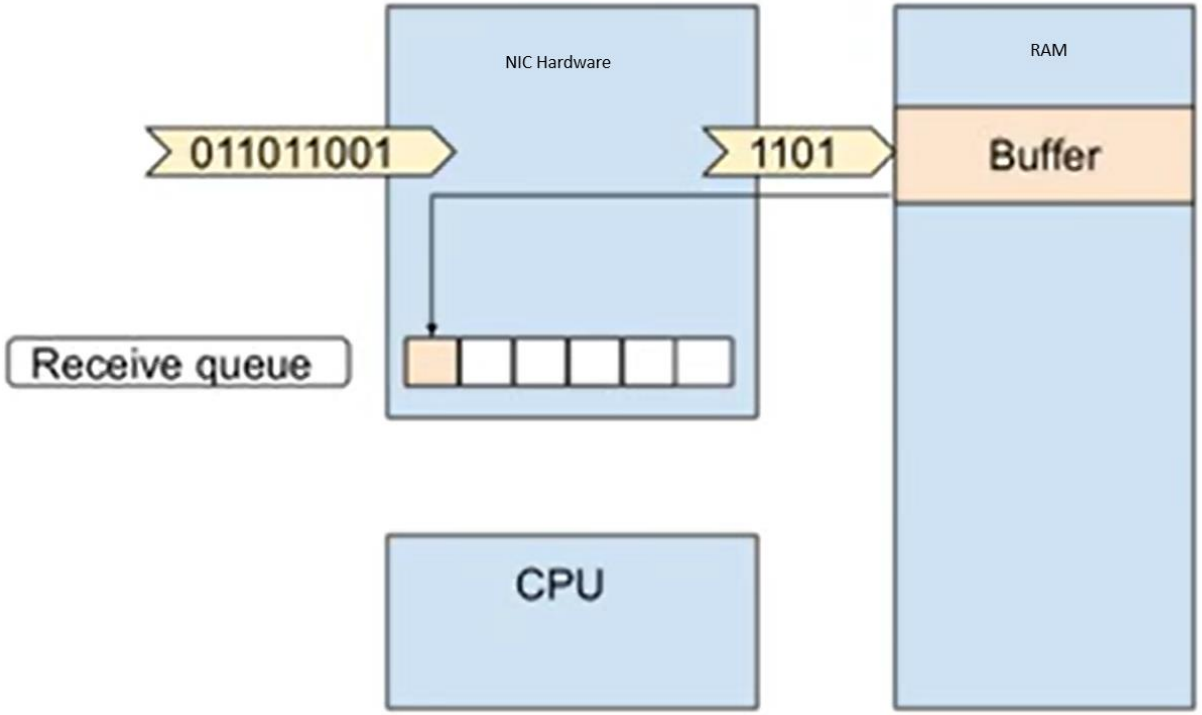
Port mapped (IO mapped)

Include asm/io.h

readb(), writeb(), readw(), writew() & readl(), writel()

```
131  void __ew32(struct e1000_hw *hw, unsigned long reg, u32 val)
132  {
133          if (hw->adapter->flags2 & FLAG2_PCIM2PCI_ARBITER_WA)
134                  __ew32_prepare(hw);
135
136          writel(val, hw->hw_addr + reg);
137  }
```

Network interface Overview

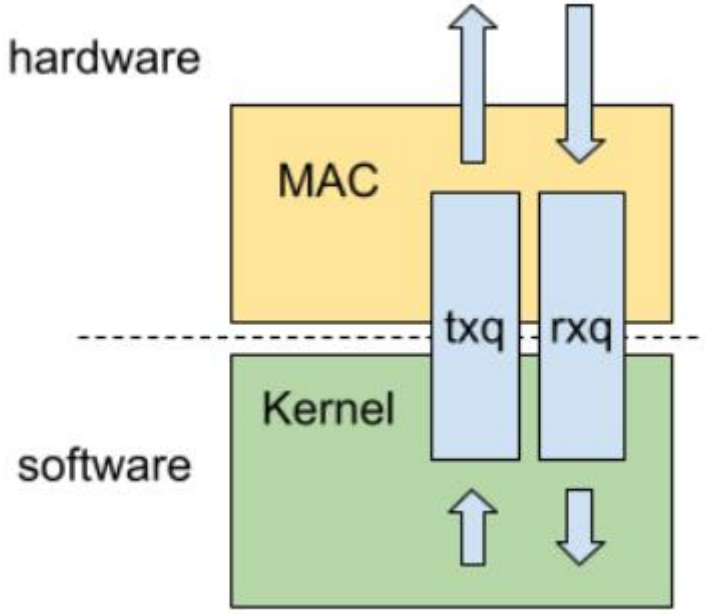


- The MAC received data and writes it to RAM using DMA
- A descriptor is created
- It's address is put in a queue

EMERTXE

Linux Network Driver

RX and TX



Linux Network Driver

RX/TX

```
1362
1363        /* allocate transmit descriptors */
1364        err = e1000_setup_all_tx_resources(adapter);
1365        if (err)
1366                goto err_setup_tx;
1367
1368        /* allocate receive descriptors */
1369        err = e1000_setup_all_rx_resources(adapter);
1370        if (err)
1371                goto err_setup_rx;
1372
```

Linux Network Driver

Struct skbuff

```
* DOC: Basic sk_buff geometry
*
* struct sk_buff itself is a metadata structure and does not hold any pac
* data. All the data is held in associated buffers.
*
* &sk_buff.head points to the main "head" buffer. The head buffer is divi
* into two parts:
*
*  - data buffer, containing headers and sometimes payload;
*    this is the part of the skb operated on by the common helpers
*    such as skb_put() or skb_pull();
*  - shared info (struct skb_shared_info) which holds an array of pointer
*    to read-only data in the (page, offset, length) format.
*
* Optionally &skb_shared_info.frag_list may point to another skb.
*
* Basic diagram may look like this::
*
*                                ---------------
*                               | sk_buff       |
*                                ---------------
*    ,---------------------------  + head
*   /          ,-----------------  + data
*  /          /      ,-----------  + tail
* |          |      |            , + end
* |          |      |           |
* v          v      v           v
*  -----------------------------------------------
* | headroom | data |  tailroom | skb_shared_info |
*  -----------------------------------------------
*                               + [page frag]
*                               + [page frag]
*                               + [page frag]
*                               + [page frag]       ---------
*                               + frag_list    --> | sk_buff |
*                                                  ---------
```

EMERTXE

Linux Network Driver

NAPI

```
/**
 *      napi_enable - enable NAPI scheduling
 *      @n: NAPI context
 *
 * Resume NAPI from being scheduled on this context.
 * Must be paired with napi_disable.
 */
void napi_enable(struct napi_struct *n)
```

```
/**
 * netif_napi_add() - initialize a NAPI context
 * @dev:  network device
 * @napi: NAPI context
 * @poll: polling function
 *
 * netif_napi_add() must be used to initialize a NAPI context prior to cal
 * *any* of the other NAPI-related functions.
 */
static inline void
netif_napi_add(struct net_device *dev, struct napi_struct *napi,
               int (*poll)(struct napi_struct *, int))
{
        netif_napi_add_weight(dev, napi, poll, NAPI_POLL_WEIGHT);
}
```

EMERTXE