Aricent®

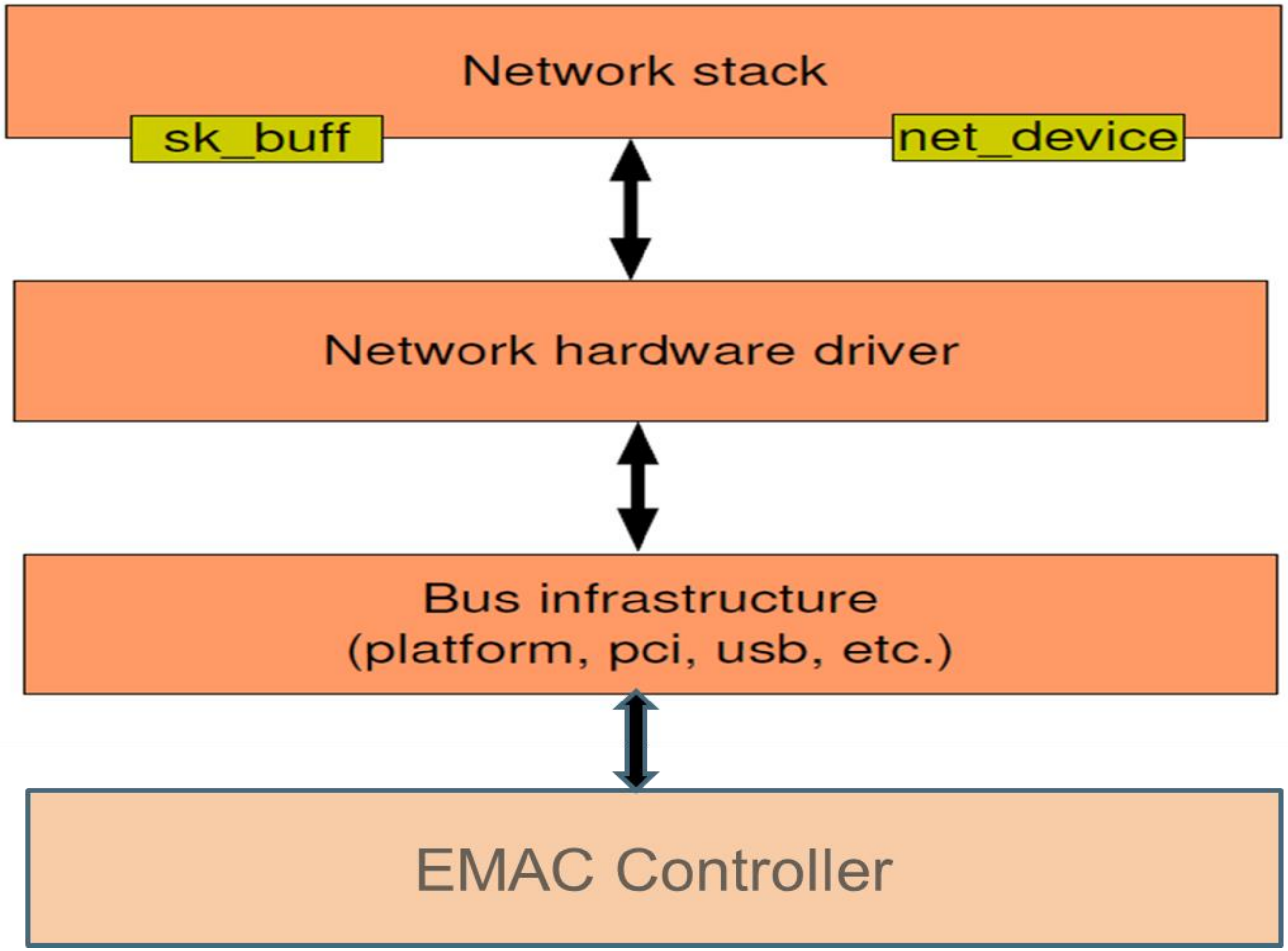
Linux Network Device Driver

Contents

Network Device Drivers

- Network device drivers receive and transmit data packets on hardware interface that connect to external systems.
- It provides a uniform interface that network protocols can access.
- These drives provides an abstraction to kernel network stack from the hardware implementation of the Ethernet controller themselves, whose implementation hides the underlying layer-1 and layer-2 protocols.
- Callback routines of a network driver must be registered with the kernel to allow the kernel to execute them when data arrives on a network port or when the kernel needs to send data out on a network port.
- Network drivers do not have device files under /dev.

Linux Network stack and driver Architecture overview



sk_buff structure

- The struct sk_buff is the structure representing a network packet
- Designed to easily support encapsulation/de-capsulation of data through the protocol layers
- In addition to the data itself, an sk_buff maintains
  - **head**, the start of the packet
  - **data**, the start of the packet payload
  - **tail**, the end of the packet payload
  - **end**, the end of the packet
  - **len**, the amount of data of the packet
- These fields are updated when the packet goes through the protocol layers.

Registration of EMAC Controller with Kernel through DTSI

- A entry for EMAC controller in DTSI files to describe the device and its address map .
- It also describes virtual IRQs corresponding to HW interrupt line for EMAC controller.
- Various fields are included in entry for EMAC controller in DTSI files
    - compatible = “controller name”
    - reg = <address map>
    - interrupts = <0 1 ..>
    - interrupt-map = <>
- DTSI compiler compile the dtsi file to create an image which are linked with kernel image by Bootloader.
- When entry for EMAC controller is encountered then kernel registers EMAC controller.
- Kernel allocates a struct platform_device with EMAC controller and populate the structure from the fields in EMAC controller entry in DTSI files

Registration of Network Device Drivers

- With the driver model, network device drivers are registered in the module-initialization function.
- The driver registers as a PCI driver, a platform driver etc. at initialization time, and the driver model infrastructure will call a probe() method when a device handled by driver is detected.
  - ✓ When driver registers as a PCI driver then it implies that the IP device which is driving by driver will communicate over PCI bus.
  - ✓ When driver registers as a platform driver then it implies that the IP device which is driving by driver will communicate over platform bus.
- In the module initialization function, register the platform driver with platform_driver_register() to register itself as platform driver.
- In the module cleanup function, platform_driver_unregister() is called to unregister driver as platform driver.
- Define a platform_driver structure and set remove and probe members so that they point to two new functions with the proper prototype, and define the driver members to the following substructure. It is important to set the compatible field of “of_match_table” pointer to “controller name”

```
.driver = {
.name = “macb”,
.owner = THIS_MODULE,
.of_match_table = dwceq_of_match
}
```

Initialization Function for EMAC Controller : Probe function

- As mentioned before, during loading kernel registers EMAC controller and get its information from DTSI image and populate it in struct platform_device correspond to controller.

- When network device driver is registered with kernel as platform driver using platform_driver_register() then kernel checks for compatible field for the driver (defined during platform driver registration under driver subfield in struct platform_driver) with compatible field for EMAC controller(defined in entry for EMAC controller in DTSI).

- When both compatible fields matches then kernel gives the ownership to EMAC controller to driver by calling probe function which are registered by network device driver.

- While calling probe function, kernel passes the pointer of struct platform_device which is populated by kernel during EMAC controller registration via DTSI.
  - Driver registers a network interface with kernel using **register_netdev()**
    - ✓ It creates and allocates a struct net_device structure and it is register with kernel using register_netdev().
    - ✓ Kernel add an entry for struct net_device structure in linked list. It is network interface list having all the network interfaces registered by network drivers and maintained by kernel.

Initialization Function for EMAC Controller : Probe function (Contd..)

- ➢ To access the I/O registers of the network card, driver needs to map them into memory, using the **ioremap()** function
- ➢ **platform_get_irq()** is called to get the IRQ number of the Ethernet controller.
  - ✓ Information for IRQ number is stored in platform_device structure which is used as an input by platform_get_irq() wrapper to get the IRQ number.
- ➢ **request_irq()** is called to register the IRQ number and interrupt handler.
- ➢ Enable the connection with PHY using **mdiobus_register()**

- Network driver maintain a private data structure to store:
  - ➢ The pointers for the structures like net_device, platform_device etc.
  - ➢ Various pointer fields to handle DMA channels or queues in EMAC controller.
  - ➢ Various fields capturing various fields like speed, duplexity, bus_id etc related to Ethernet PHY.

Network interface related structures and utility functions

a) struct net_device

- ➢ This structure represents a single network interface
- ➢ Allocation takes place with alloc_etherdev()
  - ✓ The size of private data must be passed as argument. The pointer to these private data can be read in net_device->priv
  - ✓ alloc_etherdev() is a specialization of alloc_netdev() for Ethernet interfaces
  - ✓ Allocation is done before network interface registration using register_netdev() in probe function.
- ➢ Registration with register_netdev()
- ➢ Un-registration with unregister_netdev()
- ➢ Liberation with free_netdev()

Network interface related structures and utility functions (Contd..)

b) struct net_device_ops

- The methods of a network interface. The most important ones:
  - ✓ ndo_open(), called when the network interface is up'ed
  - ✓ ndo_close(), called when the network interface is down'ed
  - ✓ ndo_start_xmit(), to start the transmission of a packet
- And others:
  - ✓ ndo_get_stats(), to get statistics
  - ✓ ndo_do_ioctl(), to implement device specific operations
  - ✓ ndo_set_rx_mode(), to select promiscuous, multicast, etc.
  - ✓ ndo_set_mac_address(), to set the MAC address
  - ✓ ndo_set_multicast_list(), to set multicast filters
- Set the netdev_ops field in the struct net_device structure to point to the struct net_device_ops structure.

Network interface related structures and utility functions (Contd..)

c) Utility functions

- netif_start_queue()
  - ✓ Tells the kernel that the driver is ready to send packets
- netif_stop_queue()
  - ✓ Tells the kernel to stop sending packets. Useful at driver cleanup of course, but also when all transmission buffers are full.
- netif_queue_stopped()
  - ✓ Tells whether the queue is currently stopped or not
- netif_wake_queue()
  - ✓ Wakeup a queue after a netif_stop_queue(). The kernel will resume sending packets

Network Operations

a) Transmission

- The driver implements the ndo_start_xmit() operation.
- The kernel calls this operation with a SKB as argument.
- The driver sets up DMA buffers and other hardware dependent mechanisms and starts the transmission.
  - ✓ Depending on the number of free DMA buffers available, the driver can also stop the queue with netif_stop_queue()
- When the packet has been sent, an interrupt is raised. The driver is responsible for
  - ✓ Acknowledging the interrupt
  - ✓ Freeing the used DMA buffers
  - ✓ Freeing the SKB with dev_kfree_skb_irq()
  - ✓ If the queue was stopped, start it again
- Returns NETDEV_TX_OK or NETDEV_TX_BUSY

Network Operations (Contd..)

b) Reception – Original mode

➢ Reception is notified by an interrupt. The interrupt handler should

- ✓ Allocate an SKB with dev_alloc_skb()
- ✓ Reserve the NET_IP_ALLIGN bytes offset with skb_reserve()
    - skb_reserve() is used Increase the headroom of empty buffer by reducing the tail room in struct sk_buff
    - CPUs often takes a performance hit when accessing unaligned memory locations. Since an ethernet header is 14 bytes network drivers often end up with the IP header at an unaligned offset. The IP header is required to be aligned by shifting the start of the packet by 2 bytes
- ✓ Copy the packet data from the DMA buffers to the SKB
    - skb_copy_to_linear_data() or skb_copy_to_linear_data_offset()
- ✓ Update the SKB pointers with skb_put()
    - skb_put() is used to update the SKB pointers after copying the payload.
- ✓ Update the skb->protocol field with eth_type_trans(skb,netdevice)
- ✓ Give the SKB to the kernel network stack with netif_rx(skb)

Network Operations (Contd..)

c) Reception – NAPI mode

- The original mode is nice and simple, but when the network traffic is high, the interrupt rate is high. The NAPI mode allows to switch to polled mode when the interrupt rate is too high.
- In the network interface private structure it is required to add a struct napi_struct for rx queues.
  - ✓ Stuct napi_struct is defined as structure for NAPI scheduling. Some important fields are :
    - **unsigned long state:** To maintain the state of poll
    - **int (*poll)(struct napi_struct *, int);** Pointer to function registered by driver responsible to copy data from DMA registers and passes the packets to network stack.
    - **struct net_device *dev;** This fields point to network interface for which it is used.
- At driver initialization, register the NAPI poll operation: netif_napi_add(dev, &bp->napi, macb_poll, 64);
  - ✓ dev is the network interface
  - ✓ &bp->napi is the struct napi_struct
  - ✓ macb_poll is the NAPI poll operation
  - ✓ 64 is the «weight» that represents the importance of the network interface. It limits the number of packets each interface can feed to the networking core in each polling cycle.
- In the interrupt handler, when a packet has been received:

```
if (napi_schedule_prep(&bp>napi)) {
  /* Disable reception interrupts */
  __napi_schedule(& bp->napi);
  }
```

Network Operations (Contd..)

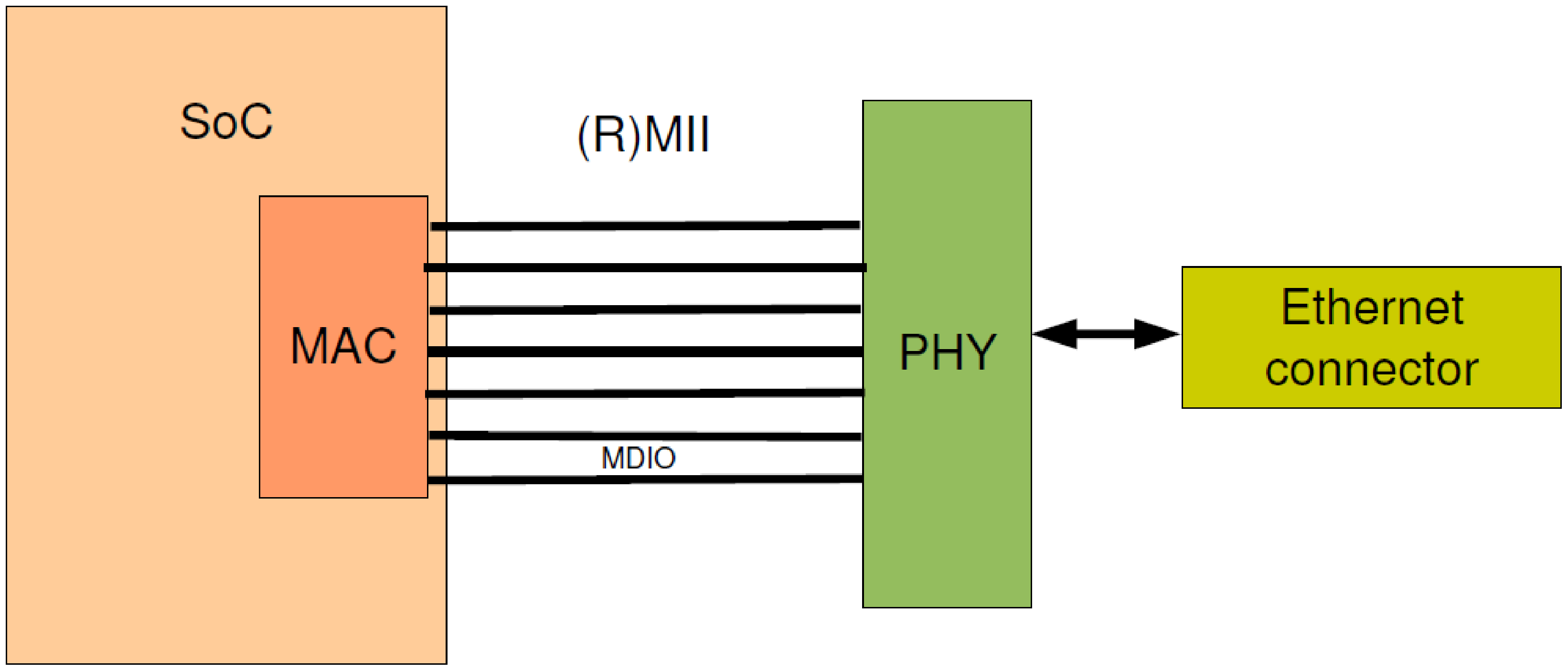
- Reception – NAPI mode (Contd..)
  - The kernel will call our poll() operation regularly
  - The poll() operation has the following prototype

    static int macb_poll(struct napi_struct *napi, int budget)
  - It must receive at most budget packets and push them to the network stack using netif_receive_skb().
  - If less than budget packets have been received, switch back to interrupt mode using napi_complete(& bp->napi) and re-enable interrupts
  - Must return the number of packets received

Communication with the PHY

- Usually, on embedded platforms, the SoC contains the Ethernet controller, that takes care of layer 2 (MAC) communication.
- An external PHY is responsible for layer 1 communication.
- There are many interfaces between MAC and PHY like MII/GMII/RMII/RGMII/SGMII. Commonly interfaces between MAC and PHY are RMII and RGMII interfaces
  - ✓ RMII = Reduced Media Independent Interface
  - ✓ RGMII = Reduced Gigabit Media Independent Interface
- This interface contains two wires used for the MDIO bus (Management Data Input/Output)
  - ✓ A PHY management interface, MDIO, used to read and write the control and status registers of the PHY which are used to configure each PHY before operation, and to monitor link status during operation
  - ✓ One wire is for **MDC clock**: driven by the MAC device to the PHY.
  - ✓ Another is for **MDIO data**: bidirectional, the PHY drives it to provide register data at the end of a read operation.
- The Ethernet driver needs to communicate with the PHY to get information about the link (up, down, speed, full or half duplex) and configure the MAC accordingly

Communication with the PHY (Contd..)



MDIO bus initialization

- The driver must create a MDIO bus structure that tells the PHY infrastructure how to communicate with the PHY.
- Allocate a MDIO bus structure
  - ✓ struct mii_bus *mii_bus = mdiobus_alloc();
- Fill the MDIO bus structure
  - ✓ mii_bus->name = “foo”
  - ✓ mii_bus->read = foo_mii_bus_read,
  - ✓ mii_bus->write = foo_mii_bus_write,
  - ✓ snprintf(mii_bus->id, MII_BUS_ID_SIZE, "%x", pdev->id);
  - ✓ mii_bus->parent = struct net_device *
- The foo_mii_bus_read() and foo_mii_bus_write() are operations to read and write a value to the MDIO bus. They are hardware specific and must be implemented by the driver.
- The mii_bus->irq[] array must be allocated and initialized. To use polling, set the values to PHY_POLL.
  - ✓ It is an array of interrupts, each PHY's interrupt at the index matching its address.
  - ✓ mii_bus->irq = kmalloc(sizeof(int)*PHY_MAX_ADDR, GFP_KERNEL);
  - ✓ for (i = 0; i < PHY_MAX_ADDR; i++)
    - ✓ Bp->mii_bus->irq[i] = PHY_POLL;
- Finally, register the MDIO bus. This will scan the bus for PHYs and fill the mii_bus->phy_map[] array with the result.
  - ✓ mdiobus_register(bp>mii_bus)

Connection to the PHY

- The mdiobus_register() function filled the mii_bus->phy_map[] array with struct phy_device * pointers
- The appropriate PHY (usually, only one is detected) must be selected
- Then, connecting to the PHY allows to register a callback that willbe called when the link changes :
  - ✓ int phy_connect_direct(struct net_device *dev,

        struct phy_device *phydev, void (*handler)(struct net_device *),

        u32 flags, phy_interface_t interface)
- Interface is usually PHY_INTERFACE_MODE_RMII or PHY_INTERFACE_MODE_RGMII

Handling link changes

- The callback that handle link changes should have the following prototype
  - ✓ void foo_handle_link_change(struct net_device *dev)
- It must check the duplex, speed and link fields of the struct phy_device structure, and update the Ethernet controller configuration accordingly
  - ✓ duplex is either DUPLEX_HALF or DUPLEX_FULL
  - ✓ speed is either SPEED_10, SPEED_100, SPEED_1000, SPEED_2500 or SPEED_10000
  - ✓ link is a Boolean
- After set up, the PHY driver doesn't operate. To poll regularly the PHY hardware, network device driver start it with
  - ✓ phy_start(phydev)
    - ✓ This will start the phy state machine in kernel and from there PHY registers will be accessed regularly via MDIO bus.
- And when the network interface is closed, the PHY must also be stopped, using
  - ✓ phy_stop(phydev)

Network Statistics

- The network driver is also responsible for keeping statistics up to date about the number of packets/bytes received/transmitted, the number of errors, of collisions, etc.
  - ✓ Collecting these information is left to the driver
- To expose these information, the driver must implement a get_stats() operation, with the following prototype
  - ✓ struct net_device_stats *foo_get_stats (struct net_device *dev);
- The net_device_stats structure must be filled with the driver. It contains fields such as rx_packets, tx_packets, rx_bytes, tx_bytes, rx_errors, tx_errors, rx_dropped, tx_dropped, multicast, collisions, etc.

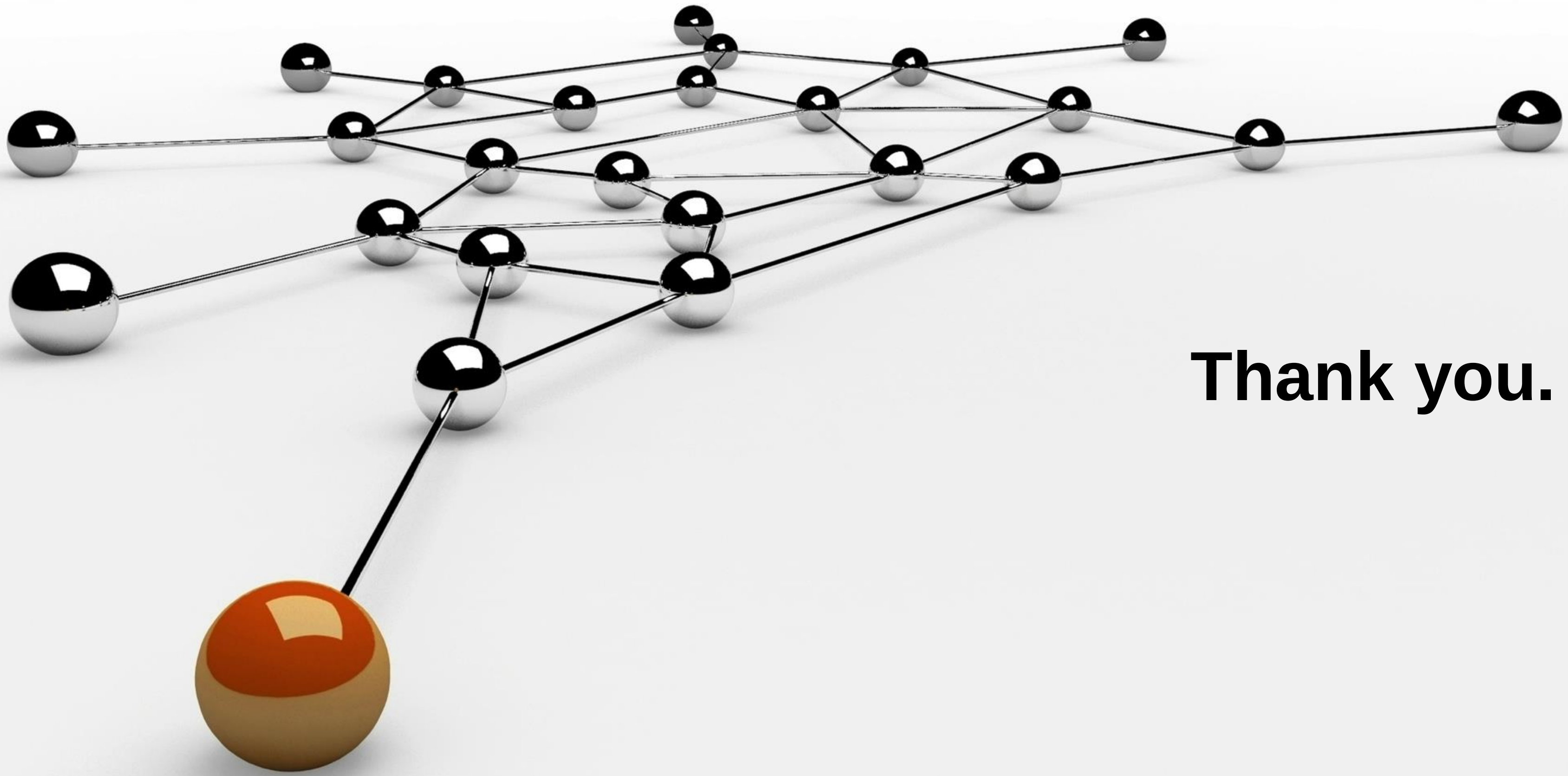
Ethtool

- **ethtool** is a userspace tool that allows to query low-level information from an Ethernet interface and to modify its configuration
- On the kernel side, at the driver level, a struct ethtool_ops structure can be declared and connected to the struct net_device using the ethtool_ops field.
- List of callback pointers under struct ethtool_ops to perform various operations:
  - ✓ get_settings()
  - ✓ set_settings()
  - ✓ get_drvinfo()
  - ✓ get_wol(),
  - ✓ set_wol()
  - ✓ get_link()
  - ✓ get_tso(),
  - ✓ set_tso(),
  - ✓ get_flags()
  - ✓ set_flags()
- Some of these operations can be implemented using the PHY interface (phy_ethtool_gset(), phy_ethtool_sset()) or using generic operations (ethtool_op_get_link() for example.

References

- «Essential Linux Device Drivers», chapter 15
- «Linux Device Drivers», chapter 17 (a little bit old)
- Documentation/networking/netdevices.txt
- Documentation/networking/phy.txt
- include/linux/netdevice.h, include/linux/ethtool.h, include/linux/phy.h,include/linux/sk_buff.h
- And of course, drivers/net/ for several examples of drivers
- Driver code templates in the kernel sources:
  - ✓ drivers/usb/usbskeleton.c
  - ✓ drivers/net/isaskeleton.c
  - ✓ drivers/net/pciskeleton.c
  - ✓ drivers/pci/hotplug/pcihp_skeleton.c

Engineering excellence. Sourced.



Thank you.

www.aricent.com