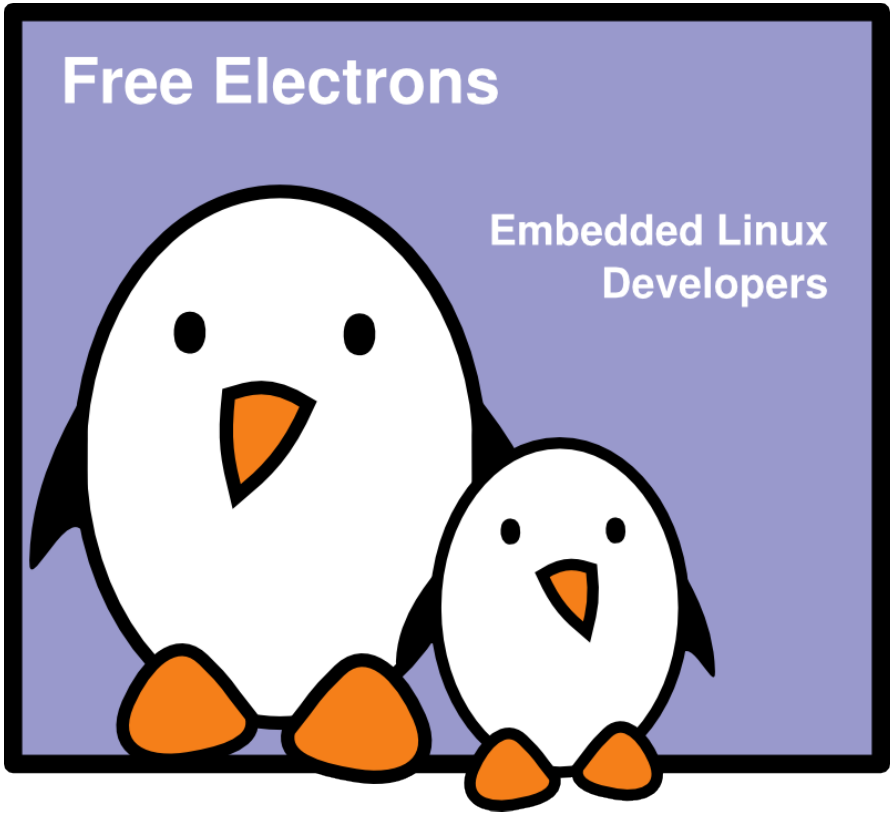
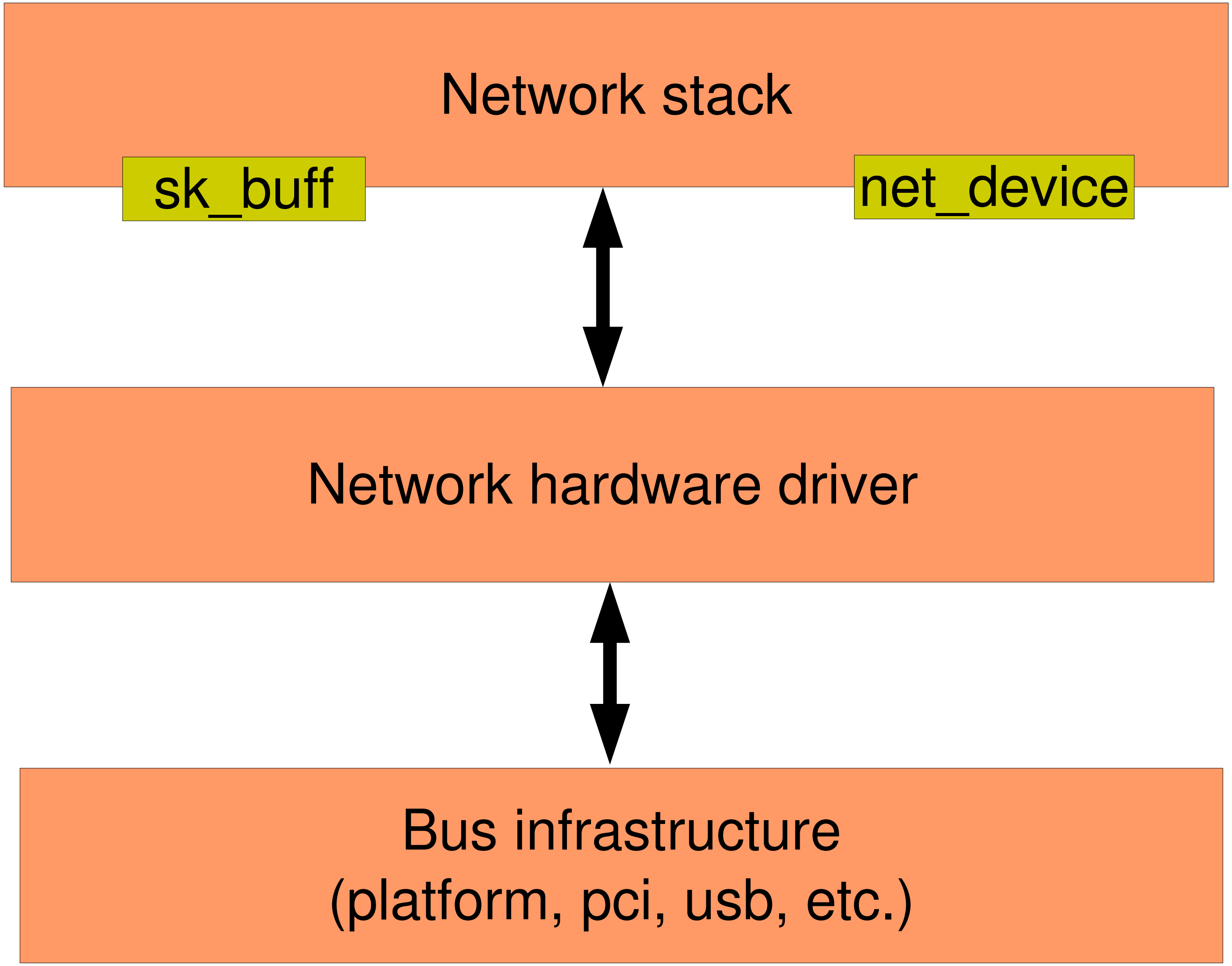
Network drivers

Thomas Petazzoni
Free Electrons



Latest update: Dec 24, 2009,
Document sources, updates and translations:
http://free-electrons.com/docs/network-drivers
Corrections, suggestions, contributions and translations are welcome!

Architecture



Network stack

sk_buff

net_device

Network hardware driver

Bus infrastructure
(platform, pci, usb, etc.)

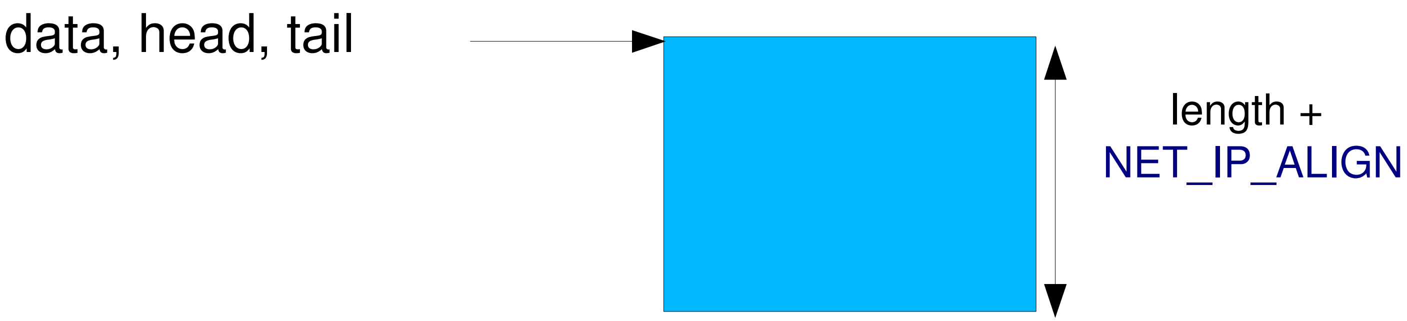
sk_buff

- The struct `sk_buff` is the structure representing a network packet
- Designed to easily support encapsulation/decapsulation 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

Allocating a SKB

- Function `dev_alloc_skb()` allows to allocate an SKB
- Can be called from an interrupt handler.
  Usually the case on reception.
- On Ethernet, the size allocated is usually the length of the packet + 2, so that the IP header is word-aligned (the Ethernet header is 14 bytes)

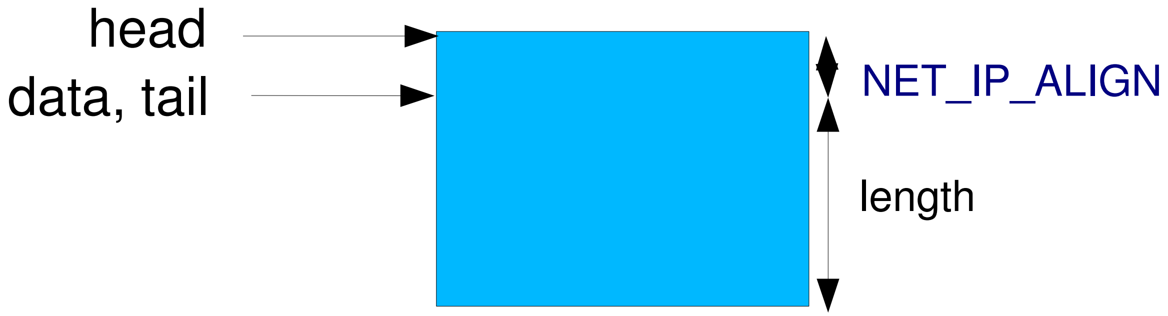
```
skb = dev_alloc_skb(length + NET_IP_ALIGN);
```



Reserving space in a SKB

- Need to skip `NET_IP_ALIGN` bytes at the beginning of the SKB
- Done with `skb_reserve()`

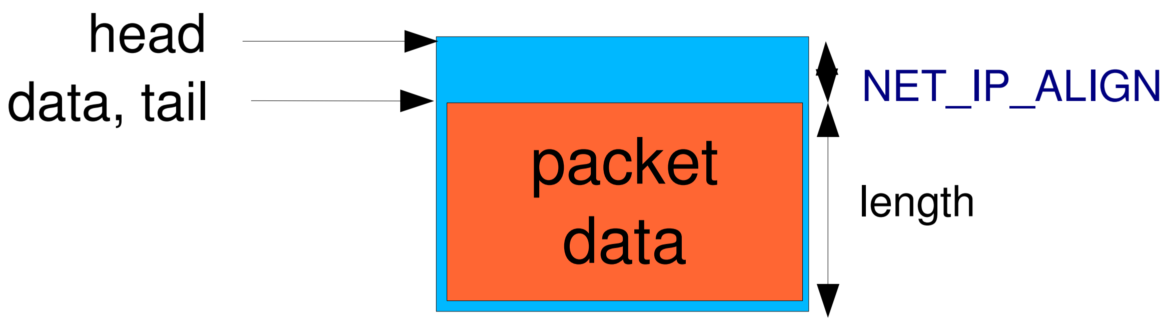
```
skb_reserve(skb, NET_IP_ALIGN);
```



Copy the received data

- The packet payload must be copied from the DMA buffer to the SKB, using
  - `static inline void` **`skb_copy_to_linear_data`**`(struct sk_buff *skb, const void *from, const unsigned int len);`
  - `static inline void` **`skb_copy_to_linear_data_offset`**`(struct sk_buff *skb, const int offset, const void *from, const unsigned int len);`

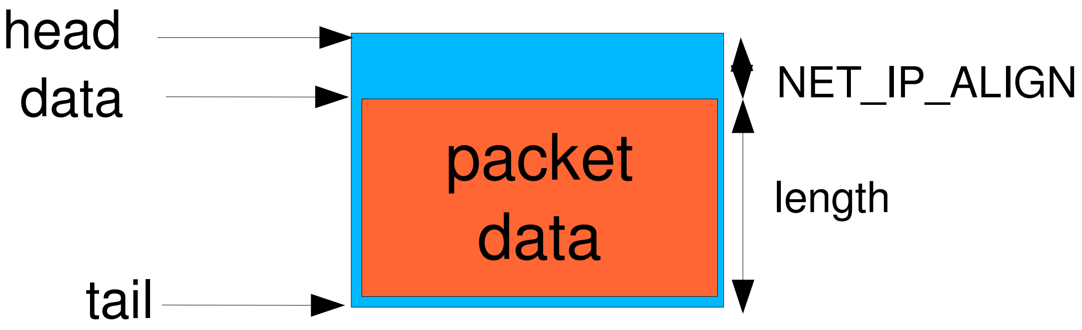
```
skb_copy_to_linear_data(skb, dmabuffer,
                  length);
```



Update pointers in SKB

- `skb_put()` is used to update the SKB pointers after copying the payload

```
skb_put(skb, length);
```



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
- Registration with `register_netdev()`
- Unregistration with `unregister_netdev()`
- Liberation with `free_netdev()`

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.

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()`
  - Wake-up a queue after a `netif_stop_queue()`. The kernel will resume sending packets

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`

Reception: original mode

- Reception is notified by an interrupt. The interrupt handler should
  - Allocate an SKB with `dev_alloc_skb()`
  - Reserve the 2 bytes offset with `skb_reserve()`
  - 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()`
  - Update the `skb->protocol` field with `eth_type_trans(skb, netdevice)`
  - Give the SKB to the kernel network stack with `netif_rx(skb)`

Reception: NAPI mode (1)

- 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, add a `struct napi_struct`
- 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. If this quota is not met, the driver will return back to interrupt mode. Don't send this quota to a value greater than the number of packets the interface can store.

Reception: NAPI mode (2)

- In the interrupt handler, when a packet has been received:

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

  - The kernel will call our `poll()` operation regularly
- The `poll()` operation has the following prototype

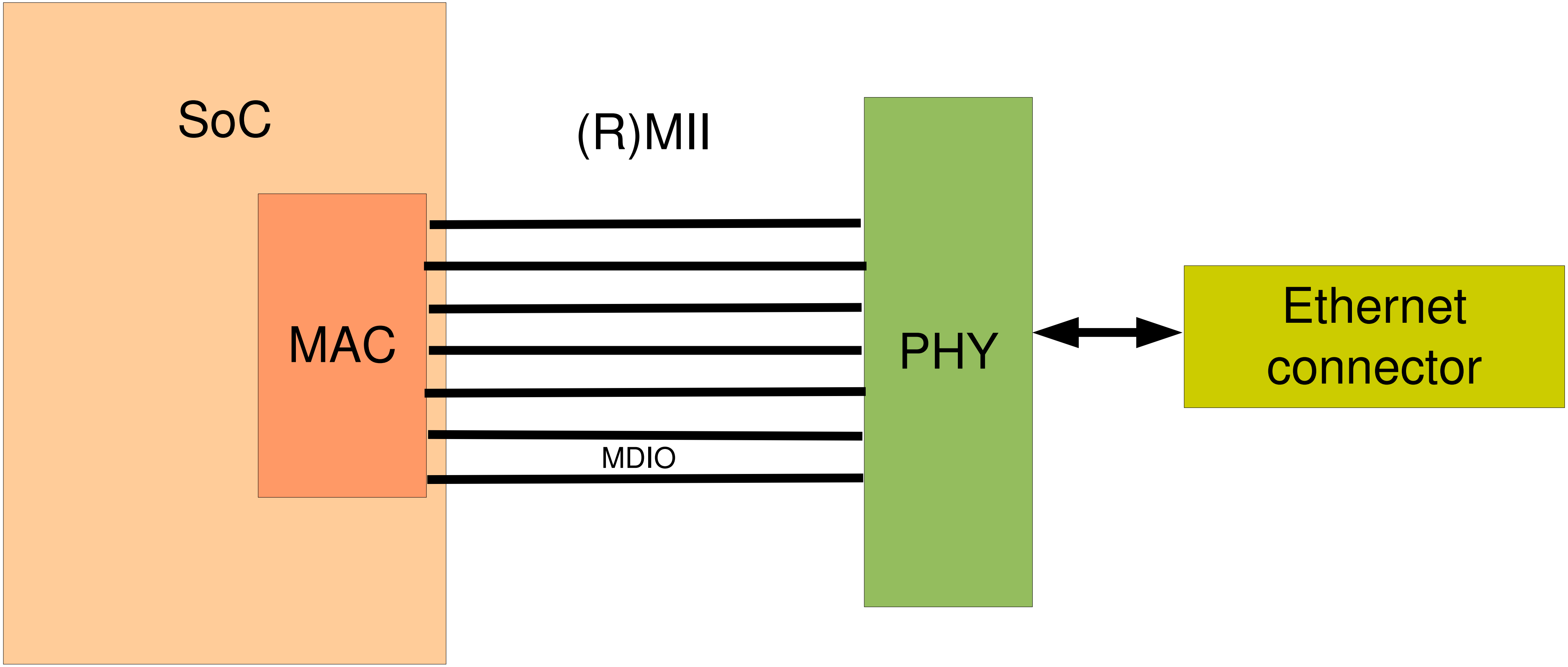
```
static int macb_poll(struct napi_struct *napi, int budget)
```

- 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 (1)

- 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.
- The MAC and the PHY are connected using a MII or RMII interface
  - MII = Media Independent Interface
  - RMII = Reduced Media Independent Interface
- This interface contains two wires used for the MDIO bus (Management Data Input/Output)
- 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 (2)



PHY in the kernel

- The kernel provides a framework that
  - Exposes an API to communicate with the PHY
  - Allows to implement PHY drivers
  - Implements a basic generic PHY driver that works with all PHY
- Implemented in `drivers/net/phy/`
- Documented in `Documentation/networking/phy.txt`

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.

MDIO bus initialization (2)

- The `->irq[]` array must be allocated and initialized. To use polling, set the values to `PHY_POLL`.

```
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 will be 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_MII` or `PHY_INTERFACE_MODE_RMII`

Updating MAC capabilities

- The MAC and the PHY might have different capabilities. Like a PHY handling Gigabit speed, but not the MAC
- The driver is responsible for updating `phydev->advertise` and `phydev->supported` to remove any PHY capability that the MAC doesn't support
- A typical solution for a 10/100 controller is
  - `phydev->supported &= PHY_BASIC_FEATURES`
  - `phydev->advertising = phydev->supported`

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

Starting and stopping the PHY

- After set up, the PHY driver doesn't operate. To make it poll regularly the PHY hardware, one must start it with

```
phy_start(phydev)
```

- And when the network is stopped, the PHY must also be stopped, using

```
phy_stop(phydev)
```

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 operations: `get_settings()`, `set_settings()`, `get_drvinfo()`, `get_wol()`, `set_wol()`, `get_link()`, `get_eeprom()`, `set_eeprom()`, `get_tso()`, `set_tso()`, `get_flags()`, `set_flags()`, etc.
- 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)

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 informations 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.

Power management

- To support suspend and resume, the network driver must implement the `suspend()` and `resume()` operations
- These operations are referenced by the `xxx_driver` structure corresponding to the bus on which the Ethernet controller is
- The `suspend()` operation should
  - Call `netif_device_detach()`
  - Do the hardware-dependent operations to suspend the devices (like disable the clocks)
- The `resume()` operation should
  - Do the hardware-dependent operations (like enable the clocks)
  - Call `netif_device_attach()`

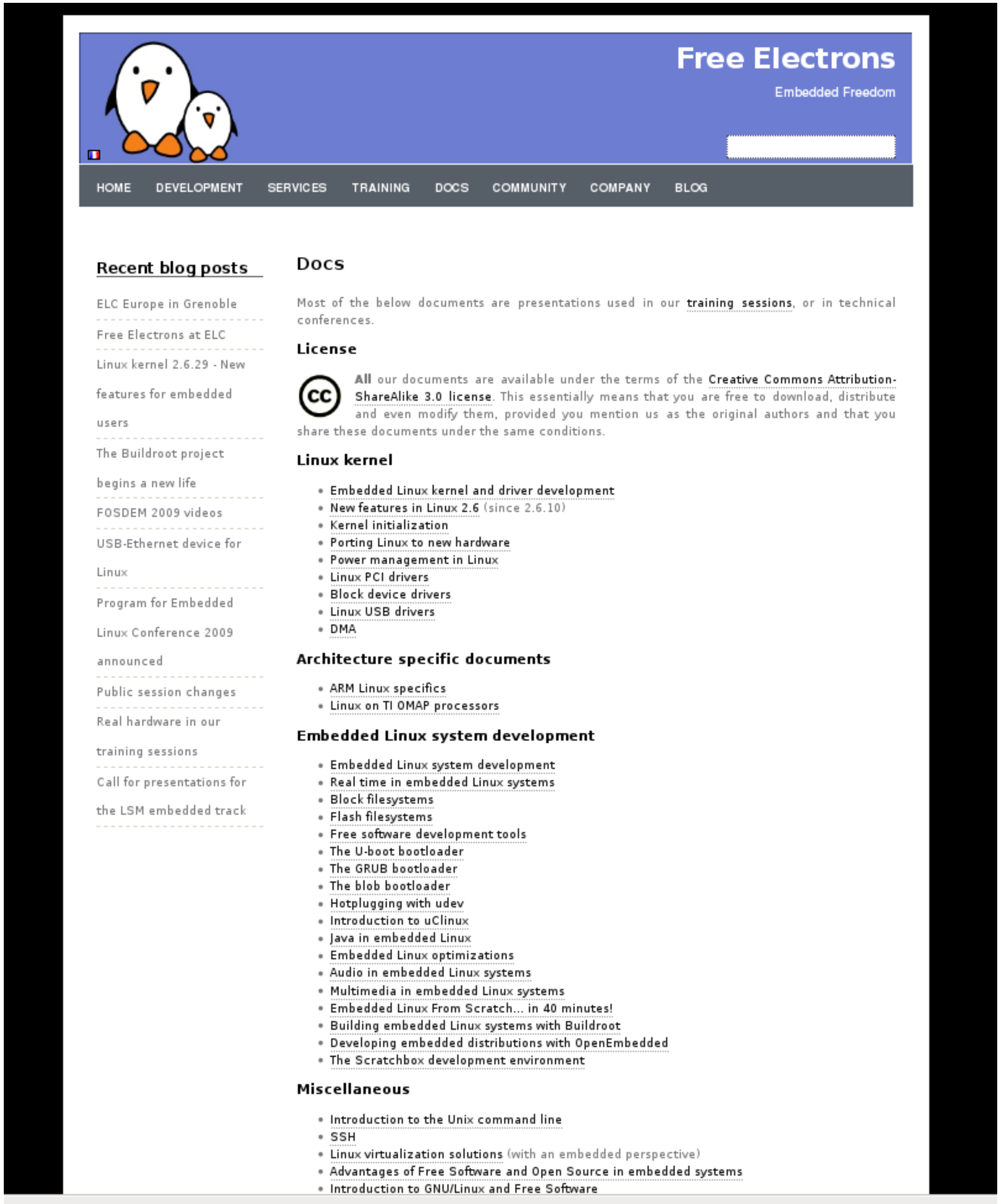
References

- *«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/usb-skeleton.c`
  `drivers/net/isa-skeleton.c`
  `drivers/net/pci-skeleton.c`
  `drivers/pci/hotplug/pcihp_skeleton.c`

Practical lab – Network drivers

- Implement a working network driver for the MACB Ethernet controller of the AT91SAM9263 CPU

Related documents



Recent blog posts

- ELC Europe in Grenoble
- Free Electrons at ELC
- Linux kernel 2.6.29 - New features for embedded users
- The Buildroot project begins a new life
- FOSDEM 2009 videos
- USB-Ethernet device for Linux
- Program for Embedded Linux Conference 2009 announced
- Public session changes
- Real hardware in our training sessions
- Call for presentations for the LSM embedded track

Docs

Most of the below documents are presentations used in our training sessions, or in technical conferences.

License

All our documents are available under the terms of the Creative Commons Attribution-ShareAlike 3.0 license. This essentially means that you are free to download, distribute and even modify them, provided you mention us as the original authors and that you share these documents under the same conditions.

Linux kernel

- Embedded Linux kernel and driver development
- New features in Linux 2.6 (since 2.6.10)
- Kernel initialization
- Porting Linux to new hardware
- Power management in Linux
- Linux PCI drivers
- Block device drivers
- Linux USB drivers
- DMA

Architecture specific documents

- ARM Linux specifics
- Linux on TI OMAP processors

Embedded Linux system development

- Embedded Linux system development
- Real time in embedded Linux systems
- Block filesystems
- Flash filesystems
- Free software development tools
- The U-boot bootloader
- The GRUB bootloader
- The blob bootloader
- Hotplugging with udev
- Introduction to uClinux
- Java in embedded Linux
- Embedded Linux optimizations
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems
- Embedded Linux From Scratch... in 40 minutes!
- Building embedded Linux systems with Buildroot
- Developing embedded distributions with OpenEmbedded
- The Scratchbox development environment

Miscellaneous

- Introduction to the Unix command line
- SSH
- Linux virtualization solutions (with an embedded perspective)
- Advantages of Free Software and Open Source in embedded systems
- Introduction to GNU/Linux and Free Software

All our technical presentations
on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization