# Kernel – Network device driver programming

Objective: Develop a network device driver for the AT91SAM9263 CPU from scratch.

## Warning

In this lab, we are going to re-implement a driver that already exists in the Linux kernel tree. Since the driver already exists, you could just copy the code, compile it, and get it to work in a few minutes. However, the purpose of this lab is to re-create this driver from scratch, taking the time to understand all the code and all the steps. So please play the game, and follow our adventure of creating a network driver from scratch !

## Setup

Go to the `/home/<user>/felabs/linux/networking` directory. It contains:

- `rootfs.jffs2`, the JFFS2 image of a root filesystem, containing the necessary tools to load and test the network device driver. Obviously, since we are going to re-develop the network driver, we cannot use NFS to mount our root filesystem during development!
- `module/`, containing a skeleton of a kernel module

The datasheet of the device is available at http://www.free-electrons.com/labs/at91sam9263-manual.pdf.

We'll need a special kernel for this lab because we need to remove the official network driver of the Calao board. Follow these steps to configure and compile the kernel:

- Grab the tarball of a recent kernel
- Modify the `Makefile` with `ARCH=arm` and adjust `CROSS_COMPILE` to your cross-compiler
- Run `make usb-a9263_defconfig` to load the default configuration for the Calao board
- Run `make xconfig` or `make menuconfig`, and in the configuration utility, go to «*Device Drivers*» → «*Network device support*» → «*10/100 Mbit/s devices*» and disable the «*Atmel MACB support*».

Now, boot the board in U-Boot, transfer and flash the kernel and root filesystem to the board, and adjust the `bootargs` parameter to mount the root filesystem from flash. In U-Boot:

- `nand erase 0 200000`
- `tftp 21000000 uImage`
- `nand write 21000000 0 200000`
- `nand erase 200000 400000`
- `tftp 21000000 rootfs.jffs2`
- `nand write 21000000 200000 ${filesize}`

- `setenv bootcmd nboot 21000000`
- `setenv autostart yes`
- `setenv bootargs mtdparts=atmel_nand:2m(kernel)ro,3m(rootfs)rw root=/dev/mtdblock1 rootfstype=jffs2`
- `saveenv`

Reboot your board, and see your kernel booting, mounting your root filesystem and starting the userspace applications.

### Testing the kernel module

Before we actually start developing the kernel driver, let's make sure we can properly compile and test a simple kernel module. The module/ directory in the current lab directory contains such a simple kernel module. Start by adjusting the `Makefile` so that it points to the location of your kernel sources, then compile the module.

As we cannot transfer the module through the network, we'll use the serial port to do so. Here's the procedure to do so:

- From Minicom, on the target, run the "`rz`" command (which is a shortcut for « receive zmodem »). A few strange characters will be displayed
- Then, again from Minicom, press "`Ctrl-a s`", which is the shortcut for the command to send files. A small dialog should pop-up to select the transfer protocol, select `zmodem`.
- A new dialog box prompting for the file to transfer will show up. Navigate to transfer the `.ko` file of your new kernel module.
- At the end of the transfer, the "`rz`" command on the target should be terminated. You can now insert and remove your module from the kernel as usual, with `insmod` and `rmmod`.

Note: make sure to remove the `.ko` file from the target before transferring a new version of the module. The "`rz`" command doesn't overwrite existing files.

### Registering a network interface

Obviously, the first and simplest step, is to register a network interface in the module initialization function, and to remove in the module cleanup function.

In the initialization function, use `alloc_etherdev()` to create a `net_device` structure, set its `netdev_ops` member to an empty `net_device_ops` structure, and register the interface using `register_netdev()`. In the cleanup function, use `unregister_netdev()` and `free_netdev()` to remove the interface and free the memory. If you load this module, a new `eth0` network interface should appear in the target system.

### Integration in the driver model

With the driver model, devices are not registered in the module initialization function. Rather, the module registers a PCI driver, a platform driver, a USB driver, etc. at initialization time, and the

driver model infrastructure will call a `probe()` method when a device handled by our driver is detected.

So, let's integrate our driver in this kernel framework:

- Define a `platform_driver` structure, set the `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:
```
.driver = {
 .name = “macb”,
 .owner = THIS_MODULE,
}
```
- In the module initialization function, remove the existing code register the platform driver with `platform_driver_register()`
- In the module cleanup function, remove the existing code and call `platform_driver_unregister()`.
- Implement the `netdrv_probe()` function. It must do the same as the previous initialization function (`alloc_etherdev()` and `register_netdev()`), but must also
  - Connect the network device (`struct net_device`) to the underlying platform device. This is done using `SET_NETDEV_DEV(dev, & pdev->dev)` where `dev` is the `struct net_device` representing the network interface, and `pdev` the platform device passed as argument to the `probe()` method
  - Set the platform device driver data pointer to the network device, using `platform_set_drvdata(pdev, dev)` Both these calls allows to have cross-references between the structure representing the network interface (high-level) and the structure representing the platform device (low-level)
- Implement the `netdrv_remove()` function. It must do the same as the previous module cleanup function. This function receives as argument a `platform_device` pointer, and not the `struct net_device` pointer. So how do we get the `struct net_device` pointer? By using the platform device driver data, that we set in the probe() function: `struct net_device *dev = platform_get_drvdata(pdev)` After unregistering the network device (`unregister_netdev()`), reset the platform device driver data using `platform_set_drvdata(pdev, NULL)` and free the network device structure (`free_netdev()`).

Once everything is implemented, compile your module, transfer it to the target and load it. Does the `eth0` interface appear as it used to do? It shouldn't! Let's see why in the next section.

### Enabling the platform device

A platform device is statically defined in the kernel source code, more precisely in the board definition file. In our case, the board definition file is `arch/arm/mach-at91/board-usb-a9263.c`. To initialize the Ethernet controller, this file calls

`ek_add_device_eth()` which is defined in `arch/arm/mach-at91/at91sam9263_devices.c` (many boards use the same AT91SAM9263 and therefore they share a lot of code).

This function `ek_add_device_eth()` only does something if `CONFIG_MACB` or `CONFIG_MACB_MODULE` are defined (these are defined when the official driver is compiled). Since we disabled the official driver to compile our own, the `ek_add_device_eth()` function is empty in our kernel, so that the `platform_device_register()` for the Ethernet controller is never called. As a workaround, change the line

```
#if defined(CONFIG_MACB) ||defined(CONFIG_MACB_MODULE)
```

by

```
#if 1
```

and then recompile and reflash the kernel on the board.

Of course, for a real driver integrated into the kernel tree, we would use the same mechanism as the one used for the official driver.

### Mapping the I/O registers

Obviously, to access the I/O registers of the network card, we need to map them into memory, using the `ioremap()` function. But prior to doing that, we are at a point in the development of our driver where we will need to store private information about the network device: for now, the virtual address at which the I/O registers have been remapped (later, other private informations will be stored).

So first, let's create a structure contain the private data, holding a single member, pointer to the I/O registers:

```
struct netdrv_device
{
        void __iomem *regs;
};
```

Then, in the `probe()` function, change the call to `alloc_etherdev(0)` to `alloc_etherdev(sizeof(struct netdrv_device))`. The kernel will automatically allocate the memory needed to store the private data. Then, at anytime in the driver code, you can use `netdev_priv()` on a `struct net_device` to get a pointer to the private structure. This area of memory will be automatically freed by `free_netdev()`, so no special change is required in the `remove()` function.

Now, let's do the mapping of the I/O registers itself. In the `probe()` function:

- Use `platform_get_resource()` to get the physical address of the I/O registers from the board definition file (see `arch/arm/mach-at91/at91sam9263_devices.c` for the list of resources for this device). This function returns a pointer to a `struct resource`, which contains two interesting fields:

`start` and `end` (both are physical addresses)

- Call `ioremap()` to remap the memory area returned by `platform_get_resource()`. The returned value, a virtual address, should be stored in the private structure that we allocated previously.

Of course, don't forget to do the error checking!

Finally, in the `remove()` function, call `iounmap()` at the proper location to remove the memory mapping.

Now, to make it easier to develop the rest of the driver, we'll add two more fields to our private `struct netdrv_device`:

- `struct net_device *dev;`
- `struct platform_device *pdev;`

And in the `probe()` function, we initialize them respectively to the `net_device` pointer and the `platform_device` pointer. This way, in the rest of the driver, we can just pass a `struct netdrv_device` pointer to sub-functions, and they will be able to access the other pieces of information.

### Registering the IRQ

Registering the IRQ is very similar to mapping the I/O registers. In the `probe()` function:

- Call `platform_get_irq()` to get the IRQ number of the Ethernet controller. It should be stored in the `irq` field of the `net_device` structure (as the `net_device` structure contains such a field, there's no point in storing the IRQ number in our private structure, as we did for the I/O registers virtual address).
- Call `request_irq()` to register this IRQ number. This will involve the creation of an interrupt handler. Just make it return `IRQ_NONE` for the moment.

In the `remove()` function, don't forget to unregister the IRQ using `free_irq()`.

### Configuring and enabling the clock

To configure the clock on the device, we'll first need some definitions of register address and values. So, take the AT91SAM9263 datasheet, chapter 41, about the EMAC Ethernet controller. More specifically, the part 41.5, describing all the registers, will be particularly useful in our case.

The clock configuration takes place in the *Network Configuration Register, EMAC_NCFG*, so let's do:

```
#define EMAC_NCFG 0x4
```

Four values for the clock divider are possible, let's add defines for them:

```
#define EMAC_CLK_DIV8  0
#define EMAC_CLK_DIV16 1
#define EMAC_CLK_DIV32 2
#define EMAC_CLK_DIV64 3
```

And the clock divider is defined at bit 10 and 11 is the NCFG register, so let's add a define for this:

```
#define EMAC_NFCG_CLK_DIV_SHIFT 10
```

Now, in the `probe()` function, we'll use the clock API of the kernel. Remember, the clock API is just `clk_get()/clk_put()`, `clk_enable()/clk_disable()` and `clk_get_rate()`. So, in the `probe()` function, we'll get and enable the "`macb_clk`" clock. The clocks are defined statically in `arch/arm/mach-at91/at91sam9263.c`. The `struct clk` pointer returned by `clk_get()` will be stored in the private structure `struct netdrv_device`.

Once the clock has been get and enabled, we need to adjust the divider of the Ethernet controller, according to the datasheet of the CPU:

```
clk_hz = clk_get_rate(priv->clk);
if (clk_hz <= 20000000)
  config = (EMAC_CLK_DIV8 << EMAC_NCFG_CLK_DIV_SHIFT);
else if (clk_hz <= 40000000)
  config = (EMAC_CLK_DIV16 << EMAC_NCFG_CLK_DIV_SHIFT);
else if (clk_hz <= 80000000)
  config = (EMAC_CLK_DIV32 << EMAC_NCFG_CLK_DIV_SHIFT);
else
  config = (EMAC_CLK_DIV64 << EMAC_NCFG_CLK_DIV_SHIFT);
 __raw_writel(config, priv->regs + EMAC_NCFG);
```

Of course, in the `remove()` function, don't forget to disable and put the clock.

### Get the MAC address

The next initialization step is to get the MAC address from the hardware, to tell the network stack about it. According to the datasheet, the MAC address can be read from two registers:

```
#define EMAC_SA1B 0x98
#define EMAC_SA1T 0x9C
```

The first one contains the low 4 bytes, the second one contains the top 2 bytes, forming the 6 bytes MAC address.

Write a function that:

- reads the MAC address (using `__raw_readl`)
- initialize a 6 bytes array with the MAC address
- test if this MAC address is valid using the `is_valid_ether_addr()` function provided by the kernel. If the address is valid, copy it to the `dev_addr` field of the `net_device` structure. If the address is not valid, generate a random network address into the same `dev_addr` field using the `random_ether_addr()` function, also provided by the kernel.

Now, in the `probe()` function, call your MAC address reading function. After returning from the function, you can add a `printk()` message to print the MAC address from the `dev_addr` field of the `net_device` structure. Compile and test your module to see if it works.

## Access to the PHY through the MDIO bus

The next step is to enable the connection between the Ethernet controller and the PHY. This takes place through the MDIO bus, for which the kernel provides a framework. The MDIO infrastructure will notify us of link state changes (cable connected or disconnected, full or half duplex, 10 or 100 Mbit/s, etc.). In this part, we'll just initialize the connection through this bus.

### MDIO bus initialization

First, add a `struct mii_bus` pointer to your private structure `netdrv_device`. Then, implement a `netdrv_mii_init()` function, that performs the following steps:

- Enable the management port at a hardware level (which is used to access the MDIO bus). This is done by setting the `MPE` bit in the `NCR` register. Add the necessary `#define` to your driver, and use `__raw_writel()` to enable this management port.
- Allocate the `struct mii_bus` structure using `mdiobus_alloc()`
- Initialize the different fields of the `mii_bus` structure
  - `name` could be set to the `"NETDRV_mii_bus"` string
  - `read` is a function pointer, so create an empty `netdrv_mdio_read()` function with the correct prototype. It will be used by the MDIO bus infrastructure to read data from the bus
  - `write` is similar, but for writing to the MDIO bus, so create an empty `netdrv_mdio_write()` function with the correct prototype
  - Initialize the `id` field using
    ```
    snprintf(mii_bus->id, MII_BUS_ID_SIZE, "%x",
                netdrvdev->pdev->id);
    ```
  - the `priv` pointer will be set so that it points to our `struct netdrv_device` structure. It will be very useful to get access to our private structure in the MDIO `read()` and `write()` functions we defined before
  - the `irq` field of the `mii_bus` structure must be allocated and initialized in the following way to tell the MDIO infrastructure that interrupts are not used between the PHY and the Ethernet controller:
    ```
    mii_bus->irq = kmalloc(sizeof(int)*PHY_MAX_ADDR,
                           GFP_KERNEL);
    for (i = 0; i < PHY_MAX_ADDR; i++)
          mii_bus->irq[i] = PHY_POLL;
    ```
- Finally, register the `mii_bus` structure using `mdiobus_register()`. The MDIO bus infrastructure will then ask the PHY for its identifier, and find if a suitable PHY driver is available (see `drivers/net/phy` for the available drivers). As in our case, no specific PHY driver exists, the generic PHY driver implemented in `drivers/net/phy/phy_device.c` will be used.

### MDIO bus access functions

Now, we have to implement the MDIO read and write functions. Reading and writing to the MDIO bus takes place through the Phy Maintenance Register (`EMAC_MAN`), while the `IDLE` bit of the Network Status Register (`EMAC_NSR`) tells us whether the MDIO bus is busy or not. So the read and write functions will be implemented as follows:

```
static int netdrv_mdio_read(struct mii_bus *bus, int mii_id,
                            int regnum)
{
       struct netdrv_device *netdrvdev = bus->priv;
       u32 out;

       out = (EMAC_MAN_SOF_VALUE  << EMAC_MAN_SOF_SHIFT)  |
             (EMAC_MAN_RW_READ    << EMAC_MAN_RW_SHIFT)   |
             (mii_id              << EMAC_MAN_PHYA_SHIFT) |
             (regnum              << EMAC_MAN_REGA_SHIFT) |
             (EMAC_MAN_CODE_VALUE << EMAC_MAN_CODE_SHIFT);


      __raw_writel(out, netdrvdev->regs + EMAC_MAN);

       while(! (__raw_readl(netdrvdev->regs + EMAC_NSR) &
             (1 << EMAC_NSR_IDLE_SHIFT))
             cpu_relax();

      return __raw_readl(netdrvdev->regs + EMAC_MAN) & 0xFFFF;
}

static int netdrv_mdio_write(struct mii_bus *bus, int mii_id,
                             int regnum, u16 value)
{
       struct netdrv_device *netdrvdev = bus->priv;
       u32 out;

       out = (EMAC_MAN_SOF_VALUE  << EMAC_MAN_SOF_SHIFT)  |
             (EMAC_MAN_RW_WRITE   << EMAC_MAN_RW_SHIFT)   |
             (mii_id              << EMAC_MAN_PHYA_SHIFT) |
             (regnum              << EMAC_MAN_REGA_SHIFT) |
             (EMAC_MAN_CODE_VALUE << EMAC_MAN_CODE_SHIFT) |
             (value               &  0xFFFF);

       __raw_writel(out, netdrvdev->regs + EMAC_MAN);

       while(! (__raw_readl(netdrvdev->regs + EMAC_NSR) &
             (1 << EMAC_NSR_IDLE_SHIFT))
               cpu_relax();

       return 0;
}
```

Of course, you'll have to create all the definitions for the different registers, according to the AT91SAM9263 specifications.

### Main initialization

Finally, we have to use this new mechanism from the `probe()` function of our driver. We'll first enable the clock and configure whether we're using a RMII or a MII connection with the PHY (through the EMAC_USRIO register), and then call our `netdrv_mii_init()` function.

The selection between RMII or MII is done based on *platform data*. These are data attached to a platform device, that are completely specific to a given device. It allows the board definition file to transmit detailed and custom information about the device to the driver. In our case, the platform data is transmitted in the form of a `eth_platform_data` structure, defined in `arch/arm/mach-at91/board-usb-a9263.c`.

To get these platform data, we'll do the following in the `probe()` function (where `pdev` is the pointer to the `platform_device` structure):

```
struct eth_platform_data *pdata;
pdata = pdev->dev.platform_data;
```

Now, we'll set bit `CLKEN` of register `EMAC_USRIO`, and optionally set the `RMII` bit if the `is_rmii` field of the platform data is true. Refer to the AT91SAM9263 datasheet for the registers and bits values, and use `__raw_writel()` to write to the `EMAC_USRIO` register.

Finally, call the `netdrv_mii_init()` function.

### Connecting the PHY and getting link change notifications

Now that the MDIO bus is initialized, we'll be able to actually connect the PHY. This will allow us to register a callback that will get called when something changes: link goes up or down, switching from half to full duplex, speed changing from 10 to 100 Mbit/s, etc.

First, let's add a `struct phy_device` to our private data structure. We'll also add fields to store the current speed and duplex status:

```
struct phy_device *phydev;
unsigned int speed;
unsigned int duplex;
```

This will point to the PHY we're using. Then, we'll implement a `netdrv_mii_probe()` function. The first step is to scan the detected PHYs to get the `phy_device` of our PHY:

```
for (phy_addr = 0; phy_addr < PHY_MAX_ADDR; phy_addr++) {
      if (netdrvdev->mii_bus->phy_map[phy_addr]) {
            phydev = netdrvdev->mii_bus->phy_map[phy_addr];
            break;
      }
}

if (! phydev)
      return -1;
```

Now, we will connect the PHY to our Ethernet controller using `phy_connect()`, and set pass a `netdrv_handle_link_change()` callback that will be called when the link status changes.

```
pdata = netdrvdev->pdev->dev.platform_data;
if (pdata && pdata->is_rmii) {
      phydev = phy_connect(netdrvdev->dev,
                           dev_name(&phydev->dev),
                           &netdrv_handle_link_change,
                           0, PHY_INTERFACE_MODE_RMII);
```

```
} else {
      phydev = phy_connect(netdrvdev->dev,
                           dev_name(&phydev->dev),
                           &netdrv_handle_link_change,
                           0, PHY_INTERFACE_MODE_MII);
}
if (! phydev)
      return -1;
```

Finally, we will set the list of supported and advertised features of our PHY to the basic features, and initialize the `phydev`, `speed` and `duplex` fields of our private structure to sane values:

```
phydev->supported &= PHY_BASIC_FEATURES;
phydev->advertising = phydev->supported;
priv->speed = 0;
priv->duplex = -1;
priv->phydev = phydev;
```

Our function is now done. Don't forget to call it from `netdrv_mii_init()` !

The last step is to implement the `netdrv_handle_link_change()` callback. This function will look at the `phydev->link`, `phydev->speed` and `phydev->duplex` values, and update accordingly the `FD` (Full-Duplex) and `SPD` (Speed) bits of the Network Configuration Register (`NCFGR`).

First case to handle, when the link is up, we check if `phydev->speed` and `phydev->duplex` are different from the one we saved in our private structure. If yes, then we update the `NCFGR` register, and save the new values in our private structure:

```
      if (phydev->link) {
            if ((priv->speed  != phydev->speed) ||
                (priv->duplex != phydev->duplex)) {
                  u32 reg;

                  reg = __raw_readl(priv->regs  + EMAC_NCFGR);

                  reg &= ~((1 << EMAC_NCFGR_SPD_SHIFT) |
                           (1 << EMAC_NCFGR_FD_SHIFT));
                  if (phydev->duplex)
                        reg |= (1 << EMAC_NCFGR_FD_SHIFT);
                  if (phydev->speed == SPEED_100)
                        reg |= (1 << EMAC_NCFGR_SPD_SHIFT);
                  __raw_writel(reg, priv->regs + EMAC_NCFGR);


                  priv->speed = phydev->speed;
                  priv->duplex = phydev->duplex;
            }
      }
```

The next case to handle is when the link goes down. Here we simply reset the speed and duplex field of our private data structures, so that next time the link goes up, they have sane default values:

```
      else {
            priv->speed  = 0;
            priv->duplex = -1;
      }
```

## DMA buffers allocation, initialization and cleanup

We'll continue our work on the network driver by writing three auxiliary functions that we will use later:

- `netdrv_alloc_consistent()`, to allocate the DMA buffers
- `netdrv_free_consistent()`, to free the DMA buffers
- `netdrv_init_rings()`, to initialize the DMA rings

First, have a read of section 41.3.2 of the AT91SAM9263 datasheet. It explains how DMA works with the Ethernet controller.

Basically, we need two rings of DMA buffer descriptors, one for the reception buffers and one for the transmission buffers. These descriptors are 8 bytes long, with 4 bytes for the address of the DMA buffer, and 4 bytes for various control flags. So let's define a structure for these descriptors:

```
struct dma_desc {
        u32     addr;
        u32     ctrl;
};
```

For the reception, we also need to allocate the DMA buffers themselves. According to the datasheet, their size is 128 bytes, therefore we define

```
#define RX_BUFFER_SIZE          128
```

We will arbitrarily decide that our reception ring contains 512 DMA buffers (and descriptors !), so let's define

```
#define RX_RING_SIZE            512
```

Therefore, the memory size to allocate for the reception DMA descriptors is

```
#define RX_RING_BYTES (sizeof(struct dma_desc) * RX_RING_SIZE)
```

Now, for the transmission, the buffers will be allocated by the kernel, since there are filled by userspace applications with the payload. For transmission, we will have 128 DMA descriptors, so let's define that and compute the amount of memory needed to store these descriptors:

```
#define TX_RING_SIZE   128
#define TX_RING_BYTES  (sizeof(struct dma_desc) * TX_RING_SIZE)
```

In addition to the DMA descriptors required by the hardware, we will also need to keep track of which packet is being transmitted through a given DMA descriptor, and where it is mapped in DMA memory. So, we define another structure, `struct ring_info`, which is not hardware-related, and is only used internally by our driver. We will later allocate `TX_RING_SIZE` elements of this structure:

```
struct ring_info {
        struct sk_buff          *skb;
        dma_addr_t              mapping;
};
```

The struct `sk_buffer` is a pointer to the packet being transmitted, while the `dma_addr_t` is the DMA address at which the packet contents were mapped prior to the beginning of the transmission.

We also need a few additional fields in our private structure, struct netdrv_device:

- `struct dma_descs *rx_ring`, which will contain the ring of reception DMA descriptors
- `void *rx_buffers`, which will contain the reception buffers themselves
- `struct dma_desc *tx_ring`, which will contain the ring of transmission DMA descriptors
- `struct ring_info *tx_skb`, which will contain the array of struct ring_info used to keep track of transmission DMA descriptors
- `dma_addr_t rx_ring_dma`, the DMA address of the reception ring (DMA addresses might be different from CPU addresses)
- `dma_addr_t tx_ring_dma`, the DMA address of the transmission ring
- `dma_addr_t rx_buffers_dma`, the DMA address of the reception buffers
- three unsigned integers, `rx_tail`, `tx_head` and `tx_tail`, that will be used to keep track of the consumption of the two rings of DMA descriptors

Now that the data structures are in place, let's create the allocation function, `netdrv_alloc_consistent()`. We will do four allocations:

1. Allocation of the array of `struct ring_info`, which can be done with normal memory allocation (`kmalloc`) since these informations are not going to be used by the Ethernet controller
2. Allocation of the reception DMA descriptors. Since they are shared with the Ethernet hardware, they must be allocated in a coherent way with `dma_alloc_coherent()`.
3. Allocation of the transmission DMA descriptors. Same as the reception DMA descriptors.
4. Allocation of the reception buffers. Same as the reception DMA descriptors.

Write the `netdrv_alloc_consistent()` function, and make it fill the `tx_skb`, `tx_ring`, `rx_ring`, `rx_buffers`, `rx_ring_dma`, `tx_ring_dma`, `rx_buffers_dma` members of our private structure `netdrv_device`. Make sure you get the error handling correct.

Similarly, write the `netdrv_free_consistent()` function that does the opposite, using `kfree()` and `dma_free_coherent()`.

Finally, we'll write a `netdrv_init_rings()` function to initialize the two rings, according to the datasheet specification.

For the reception ring, we'll initialize each descriptor with the address of the corresponding reception buffer. The last descriptor will have the WRAP bit of the addr field set, to indicate it is the last descriptor:

```
addr = priv->rx_buffers_dma;
for (i = 0; i < RX_RING_SIZE; i++) {
      priv->rx_ring[i].addr = addr;
```

```
      priv->rx_ring[i].ctrl = 0;
      addr += RX_BUFFER_SIZE;
}
priv->rx_ring[RX_RING_SIZE - 1].addr |=
             (1 << EMAC_DMA_RX_WRAP_SHIFT);
```

For the transmission ring, we'll initialize all addresses to zero (since we don't yet have packets to transmit!), and we will set the USED bit in the `ctrl` field to indicate that these descriptors are owned by the CPU and not the Ethernet controller. Similarly to reception descriptors, the last transmission descriptor will have its WRAP bit set to indicate it's the last. Be careful, in reception descriptors, this bit is part of the `addr` field while for transmission descriptors, it is part of the `ctrl` field.

```
for (i = 0; i < TX_RING_SIZE; i++) {
      priv->tx_ring[i].addr = 0;
      priv->tx_ring[i].ctrl = (1 << EMAC_DMA_TX_USED_SHIFT);
}
priv->tx_ring[TX_RING_SIZE - 1].ctrl |=
      (1 << EMAC_DMA_TX_WRAP_SHIFT);
```

Finally, reset the `tx_head`, `tx_tail` and `rx_tail` fields:

```
priv->rx_tail = priv->tx_head = priv->tx_tail = 0;
```

### Hardware reset and initialization

Obviously, to do hardware initialization, we need a set of register addresses definitions:

- The transmit status register (TSR)
  `#define EMAC_TSR 0x14`
- The receive buffer queue pointer (RBQP)
  `#define EMAC_RBQP 0x18`
- The transmit buffer queue pointer (TBQP)
  `#define EMAC_TBQP 0x1C`
- The reception status register (RSR)
  `#define EMAC_RSR 0x20`
- The interrupt status register (ISR)
  `#define EMAC_ISR 0x24`
- The interrupt enable register (IER)
  `#define EMAC_IER 0x28`
- The interrupt disable register (IDR)
  `#define EMAC_IDR 0x2C`

In addition to these, bit definitions are needed:

- For the Network Configuration Register (NCR), we need the bits to enable transmission and reception
  ```
  #define EMAC_NCR_RE_SHIFT 2
  #define EMAC_NCR_TE_SHIFT 3
  ```
- For the interrupt enable register (IER), we need the bits to enable interrupts on transmission and reception completion
  ```
  #define EMAC_IER_RCOMP_SHIFT    1
  #define EMAC_IER_TCOMP_SHIFT    7
  ```

Now, write a `netdrv_reset_hw()` function that:

- Set all bits to one in the Transmit Status Register
- Set all bits to one in the Reception Status Registered
- Set all bits to one in the Interrupt Disable Register
- Read the Interrupt Status Register to clear any pending interrupt

And write a `netdrv_init_hw()` function that:

- Calls `netdrv_reset_hw()`
- Sets the correct values in the Transmit Buffer Queue Pointer and Reception Buffer Queue Pointer registers
- Enable reception and transmission in the Network Configuration Register
- Enable the transmission and reception completion interrupts in the Interrupt Enable register

Of course, these functions will be used later.

### Implement open and close operations

These operations are respectively called when the network interface is enabled and disabled, for example using `ifconfig` from userspace.

First, create two empty functions, `netdrv_open()` and `netdrv_close()`. Both functions return an integer value and take as argument a `struct net_device` pointer. Then, register these operations in the `net_device_ops` structure previously created, under the `ndo_open` and `ndo_close` fields:

```
        .ndo_open               = netdrv_open,
        .ndo_stop               = netdrv_close,
```

Now, let's implement these functions. In the `netdrv_open()` function, we need to:

- Allocate the DMA buffers using `netdrv_alloc_consistent()`
- Initialize the DMA buffers using `netdrv_init_rings()`
- Initialize the hardware using `netdrv_init_hw()`
- Start the PHY using `phy_start()` on the PHY device that we've previously stored in our private data structure. This `phy_start()` function will start polling the PHY regularly to detect link changes
- Call `netif_start_queue()` to tell the kernel that our interface is ready to operate packets

Symmetrically, in the `netdrv_close()` function:

- Call `netif_stop_queue()` to tell the kernel that our interface no longer accepts packets
- Stop the PHY using `phy_stop()`
- Reset the hardware using `netdrv_reset_hw()` so that interupts are disabled, etc.
- Free the DMA buffers using `netdrv_free_consistent()`

## Introduce locking

Until now, our driver does not implement proper locking, which might lead to incorrect concurrent access to shared resources. Therefore, we must implement locking. In this driver, a single spinlock will be used, since the concurrent accesses that must be prevented occur between the interrupt handler and the process context code.

Therefore, add a `spinlock_t` structure to our private data structure, and initialize this spinlock with `spin_lock_init()` in the `probe()` method.

Then, we must use this spinlock in:

- `netdrv_handle_link_change()`, with `spin_lock_irqsave()` and `spin_unlock_irqrestore()` to prevent concurrency between the execution of this function and the interrupt handler
- `netdrv_close()`, again with `spin_lock_irqsave()` and `spin_lock_irqrestore()` to prevent concurrency between interrupts and the operation of stopping the network interface. This must be done after stopping the queue and the PHY.

## Implement transmission

### Definitions

Before implementing the transmission function themselves, let's start by adding the usual definitions:

- The TSTART bit of the Network Configuration Register, used to start the transmission of the packets stored in the Transmission Queue
  ```
  #define EMAC_NCR_TSTART_SHIFT 9
  ```
- The transmission completion bit of the Transmit Status Register
  ```
  #define EMAC_TSR_COMP_SHIFT 5
  ```
- The bit of the transmission DMA descriptor that tells if the current descriptor is the last buffer of the current frame. In our case, this bit will be set of all transmission DMA descriptors since we will always send a packet in a single DMA buffer
  ```
  #define EMAC_DMA_TX_LAST_SHIFT 15
  ```
- A macro that tells how many DMA buffers are currently available (free) in the queue
  ```
  #define TX_BUFFS_AVAIL(priv) \
  (((priv)->tx_tail <= (priv)->tx_head) ?                \
  (priv)->tx_tail + TX_RING_SIZE - 1 - (priv)->tx_head: \
  (priv)->tx_tail - (priv)->tx_head - 1)
  ```
- A macro that given an index in the queue of DMA transmission buffers, returns the index of the next one
  ```
  #define NEXT_TX(n) (((n) + 1) & (TX_RING_SIZE - 1))
  ```

### The transmission entry point

The entry point of our driver for the transmission of packets is the

`int ndo_start_xmit(struct sk_buff *, struct net_device *)` operation. So, create an empty `netdrv_start_xmit()` function and register it in the `net_device_ops` structure.

The code of the netdrv_start_xmit() function will manipulate the queue of DMA buffer descriptors and this queue will also be modified by the interrupt handler. Therefore, locking must be used. As the `start_xmit()` function is guaranteed never to be called from an interrupt handler, we can directly use `spin_lock_irq()` and `spin_unlock_irq()`.

Once the lock is taken, the first thing to check is if we have at least one remaining DMA buffer descriptor available to send the packet (using the `TX_BUFFS_AVAIL` macro) If not, this is really a problem since we are supposed to manage this queue and tell the kernel to stop sending packets when the queue is full. Therefore, if this happens, stop the queue with `netif_stop_queue()`, release the spinlock and return 1 (which the kernel will interpret as an error).

If we have at least one DMA buffer descriptor available, the next available is the one pointed by `tx_head` in our private data structure.

The next step is to map the packet so that it can be send through DMA (we are using « streaming DMA »). It takes place using the `dma_map_single()` function, which takes as argument a struct device pointer (can be found from our private data structure), the memory area to be mapped (the pointer to the packet data is `skb->data`), the length (`skb->length`) and the direction of the DMA transfer (in our case a transmission to the device, so `DMA_TO_DEVICE`). The function returns a DMA address, of the type dma_addr_t. This is the address we must give to our device.

Then, update our internal `tx_skb` array with the DMA address and the pointer to the SKB. This will be useful at the completion of the transmission.

Now, let's compute the value of the ctrl field of the DMA buffer descriptor:

- It must contain the length of the data to transmit, skb->len
- The EMAC_DMA_TX_LAST_SHIFT bit must be set, as all our packets are sent through a single buffer
- If the buffer we're using is the last one of the queue (tx_head is equal to TX_RING_SIZE - 1), then the EMAC_DMA_TX_WRAP_SHIFT bit must be set

Then, initialize the `addr` field of the DMA buffer descriptor with the DMA address, and the `ctrl` field with the value computed previously. To prevent the reordering of these writes with the write that will start the transmission, add a write memory barrier after the setup of the DMA buffer descriptor.

Then, update the `tx_head` to the next available transmission buffer so that further calls to `start_xmit()` will use another buffer (hint: use the `NEXT_TX` macro).

Finally, start the transmission by setting the `EMAC_NCR_TSTART_SHIFT` bit of the Network Configuration Register. Be careful not to change the value of other bits in this register !

Before the end of the function, we must tell the kernel if we still have DMA buffer descriptors available to accept new packets. Test the number of DMA buffer descriptors available, and call `netif_stop_queue()` if needed.

### Transmission completion

The completion of the transmission will of course be notified by an interrupt. So, when an interrupt is raised, we will check if it's due to a transmission completion, and if so, we will unmap the DMA buffer, mark it as available, and potentially signal the kernel that we are ready again to send more packets.

So, the first part takes place in the interrupt handler, `netdrv_interrupt()`. First, we need to test if the interrupt really originates from our device. To do so, read the EMAC_ISR (Interrupt Status Register), and if it's 0 (no interrupt pending), then simply return `IRQ_NONE` to the kernel.

Otherwise, take our spinlock, so that the execution of the code of our interrupt handler is protected against concurrent access. Using the spin_lock() and spin_unlock() variant is sufficient, since our interrupt is already guaranteed to be disabled.

Then, we have to loop until the EMAC_ISR register is 0. This register gets reset to 0 when it's read, so there's no need to reset bits manually in it. However, this also mean that you must save and use the value of the register as it was in the first test at the beginning of the interrupt handler.

In the loop, test if the bit `EMAC_IER_TCOMP_SHIFT` bit is set, which notifies a transmission completion. If so, call a new `netdrv_tx()` function that will take care of finishing the transmission process.

Now, let's implement the netdrv_tx() function. This function should:

- Verify in the Transmit Status Register that a transmission completion occurred. To do so, one must
  - Read the `EMAC_TSR` register
  - Write the read value into the `EMAC_TSR` register to clear the bits (according to the controller specification, writing with a bit set actually clears the bit in the register)
  - Test if the `EMAC_TSR_COMP_SHIFT` bit is set, and if not, return
- Test all DMA buffer descriptors (in a loop), for the tail (pointed by `tx_tail`) to the head (pointed by `tx_head`). Remember to use `NEXT_TX()` to compute the index of the next DMA buffer descriptor in the queue. For each descriptor, we will:
  - Use a read memory barrier to make sure that what we will actually read is what has been set by the device into the DMA descriptor
  - Test the `EMAC_DMA_TX_USED_SHIFT` bit. If it isn't set, then we have to stop the loop over the DMA descriptors, since it means that we reached a DMA descriptor whose transmission hasn't been completed by the controller
  - Unmap the SKB using `dma_unmap_single()`

- Free the SKB using `dev_kfree_skb_irq()`
- Reset the SKB pointer in our private `tx_skb[]` array.

- At the end of the loop, update tx_tail so that it points to the DMA descriptor to be analyzed at the next transmission completion interrupt.
- Finally, if the queue was stopped (which can be tested using netif_queue_stopped()) and if we have enough transmit buffers available (say 32), then tell then kernel to start sending packets again using netif_wake_queue().

With this transmission infrastructure in place, your system should be able to send packets. You can test with Wireshark on your host PC, and try to ping the host PC from the target. Ping will not work of course (due to the lack of reception support), but Wireshark should see ARP requests coming from the target.

## Implement reception

The last (but not least!) part of our driver is obviously to implement the reception support.

The reception of packets is notified through an interrupt, so in the interrupt handler, we'll add a call to a netdrv_rx() function. This function will go through the list of DMA descriptors, and find the ranges of DMA descriptors that correspond to a packet. For each of these ranges, a netdrv_rx_frame() function will be called to handle the reception of a packet. Here, we have a difference between transmission and reception: on the transmission side, eack packet is completely sent through a single DMA buffer and descriptor, while on the reception side, DMA buffers are limited to 128 bytes, so multiple reception DMA buffers are usually needed to store the contents of a network packet.

### Definitions

As usual, additional definitions are needed:

- Bit definitions for the DMA reception descriptors
  - `#define EMAC_DMA_RX_USED_SHIFT 0`
    This bit is set to one in the address field of the DMA descriptor by the device when the DMA buffer has been filled with data
  - `#define EMAC_DMA_RX_SOF_SHIFT 14`
    This bit is set to one in the control field of the DMA descriptor by the device when the data in this DMA buffer is the beginning of a network packet (SOF stands for Start of Frame)
  - `#define EMAC_DMA_RX_EOF_SHIFT 15`
    This bit is set to one in the control field of the DMA descriptor by the device when the data in this DMA buffer is the end of a network packet (EOF stands for End of Frame)
- As the Ethernet header is 14 bytes in size and for performance reasons, it's better to have the IP header word-aligned, many Ethernet drivers allocates two additional bytes in each packet and shift by two bytes the Ethernet header. So

we define a constant `RX_OFFSET`:
`#define RX_OFFSET 2`

- A macro `NEXT_RX()`, implemented just like `NEXT_TX()` except that it wraps at `RX_RING_SIZE` instead of `TX_RING_SIZE`.

### In the interrupt handler

The work in the interrupt handler is simple: after the test for the transmission completion, add a similar test on the `EMAC_IER_RCOMP_SHIFT` bit, and if this bit is set, call `netdrv_rx()`.

### The netdrv_rx() function

In this function, loop over the DMA descriptors starting at the reception tail (`rx_tail`), and so the following things:

- Call `rmb()` to make sure that what you're reading from the DMA descriptors will be correct
- If the `EMAC_DMA_RX_USED_SHIFT` bit in the address field of the current DMA descriptor isn't set, then we have reached the last received DMA buffer, and we can break out of the loop
- If the `EMAC_DMA_RX_SOF_SHIFT` bit in the control field of the current DMA descriptor is set, then the current DMA buffer is the beginning of a new packet. Store the index of the current DMA descriptor in a variable, so that we remember what is the first DMA buffer of the current packet
- If the `EMAC_DMA_RX_EOF_SHIFT` bit in the control field of the current DMA descriptor is set, then the current DMA buffer is the end of the current packet. So now, we have the index of the beginning of the packet (saved previously) and the index of the end of the packet. With these two informations, we call our `netdrv_rx_frame()` function to handle the reception of a complete packet

After the loop, remember to update the `rx_tail` properly.

### The netdrv_rx_frame() function

This function is in charge of allocating an SKB from the network stack, filling it with the packet data, and submitting the SKB for analysis to the network stack.

At the beginning of the function, let's compute the length of the received packet. This length is store in the 11 low-order bits of the control field of the last DMA descriptor containing the packet.

Then, we ask the kernel to allocate a SKB for us, using `dev_alloc_skb()`. This function takes a length as argument, which must be the length of the received packet plus the `RX_OFFSET` used to make sure the IP header will be word-aligned. Of course, check that the allocation was successful. If it isn't, the packet is simply dropped, but don't forget to mark the DMA descriptors of the packet as unused by clearing the `EMAC_DMA_RX_USED_SHIFT` bit of the address field.

Then, we need to set-up the SKB:

- tell the kernel that the first two bytes of the packets are to be ignored using `skb_reserve(skb, RX_OFFSET)`.

- Tell the kernel that our device didn't do any verification of the packet checksums (some devices do this directly in hardware). This is done using `skb->ip_summed = CHECKSUM_NONE`.
- Tell the kernel how much data we will put in our SKB using `skb_put()` with the packet length as argument

Once our SKB is setup, let's go through the different DMA buffers that contain the data of our packet and handle them in a loop that does:

- Computes the length of the data available in the current DMA buffer. Usually it's the size of the buffer, RX_BUFFER_SIZE, except for the last one !
- Copy data from the DMA buffer to the SKB, using `skb_copy_to_linear_data_offset()`. The arguments of this function are: the SKB pointer, the offset in the SKB at which the data should be copied, the location from which the data should be taken, and the length of the data to copy
- Clear the EMAC_DMA_RX_USED_SHIFT bit in the DMA descriptor, to mark the corresponding DMA buffer as available again for future receptions

At the end of the function, we must compute the protocol of the packet that has been received and store it in the SKB: `skb->protocol = eth_type_trans(skb, priv->dev)`.

And finally, submit the received packet to the kernel using `netif_rx()`.

Now, your driver should be working, and network traffic should go back and forth between the target and the rest of the world. Congratulations!

### Improvements

Compared to the official driver for this Ethernet controller as available in the kernel, our driver lacks a few features:

- No support for NAPI, which allows to limit the interrupt rate when the network traffic increases significantly
- No support for the ethtool API, which allows userspace applications to get informations about the status of the link and to configure a few settings
- No support for the statistics (packets received/sent, bytes received/sent, errors, etc.)
- No support for promiscuous mode and for the multicast filters
- No proper management of errors communicated by the Ethernet controller