Kernel frameworks for device drivers

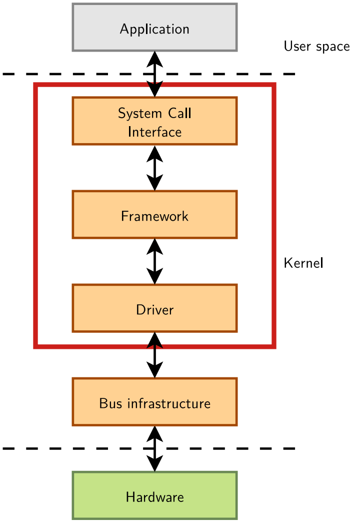
Corrections, suggestions, contributions and translations are welcome!

Kernel and Device Drivers

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features to user space applications.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *kernel frameworks*, while the *bus infrastructure* was covered earlier in this training.



User space vision of devices

Types of devices

Under Linux, there are essentially three types of devices:

- **Network devices**. They are represented as network interfaces, visible in user space using `ip a`
- **Block devices**. They are used to provide user space applications access to raw storage devices (hard disks, USB keys). They are visible to the applications as *device files* in `/dev`.
- **Character devices**. They are used to provide user space applications access to all other types of devices (input, sound, graphics, serial, etc.). They are also visible to the applications as *device files* in `/dev`.

→ Most devices are *character devices*, so we will study these in more details.

Major and minor numbers

- Within the kernel, all block and character devices are identified using a *major* and a *minor* number.
- The *major number* typically indicates the family of the device.
- The *minor number* allows drivers to distinguish the various devices they manage.
- Most major and minor numbers are statically allocated, and identical across all Linux systems.
- They are defined in `admin-guide/devices`.

Devices: everything is a file

- A very important UNIX design decision was to represent most *system objects* as files
- It allows applications to manipulate all *system objects* with the normal file API (`open`, `read`, `write`, `close`, etc.)
- So, devices had to be represented as files to the applications
- This is done through a special artifact called a **device file**
- It is a special type of file, that associates a file name visible to user space applications to the triplet *(type, major, minor)* that the kernel understands
- All *device files* are by convention stored in the `/dev` directory

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda /dev/sda1 /dev/sda2 /dev/sdc1 /dev/zero
brw-rw---- 1 root disk    8,  0 2011-05-27 08:56 /dev/sda
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
brw-rw---- 1 root disk    8, 32 2011-05-27 08:56 /dev/sdc
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a serial port

```c
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Creating device files

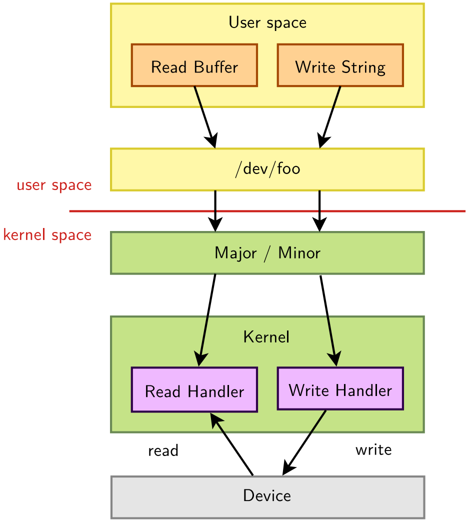
- Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needed root privileges
  - Coherency between device files and devices handled by the kernel was left to the system developer
- The `devtmpfs` virtual filesystem can be mounted on `/dev` and contains all the devices registered to kernel frameworks. The `CONFIG_DEVTMPFS_MOUNT` kernel configuration option makes the kernel mount it automatically at boot time, except when booting on an initramfs.
- `devtmpfs` can be supplemented by userspace tools like `udev` or `mdev` to adjust permission/ownership, load kernel modules automatically and create symbolic links to devices.

Character drivers

A character driver in the kernel

- From the point of view of an application, a *character device* is essentially a **file**.
- The driver of a character device must therefore implement **operations** that let applications think the device is a file: `open`, `close`, `read`, `write`, etc.
- In order to achieve this, a character driver must implement the operations described in the `struct file_operations` structure and register them.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

From user space to the kernel: character devices



File operations

Here are the most important operations for a character driver, from the definition of `struct file_operations`:

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

Many more operations exist. All of them are optional.

open() and release()

- `int foo_open(struct inode *i, struct file *f)`
  - Called when user space opens the device file.
  - **Only implement this function when you do something special with the device at `open()` time.**
  - `struct inode` is a structure that uniquely represents a file in the filesystem (be it a regular file, a directory, a symbolic link, a character or block device)
  - `struct file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains information like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the `file` structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user space closes the file.
  - **Only implement this function when you do something special with the device at `close()` time.**

read() and write()

- `ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `read()` system call on the device.
  - Must read data from the device, write at most `sz` bytes to the user space buffer `buf`, and update the current position in the file `off`. `f` is a pointer to the same file structure that was passed in the `open()` operation
  - Must return the number of bytes read.
    `0` is usually interpreted by userspace as the end of the file.
  - On UNIX, `read()` operations typically block when there isn't enough data to read from the device
- `ssize_t foo_write(struct file *f, const char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `write()` system call on the device
  - The opposite of `read`, must read at most `sz` bytes from `buf`, write it to the device, update `off` and return the number of bytes written.

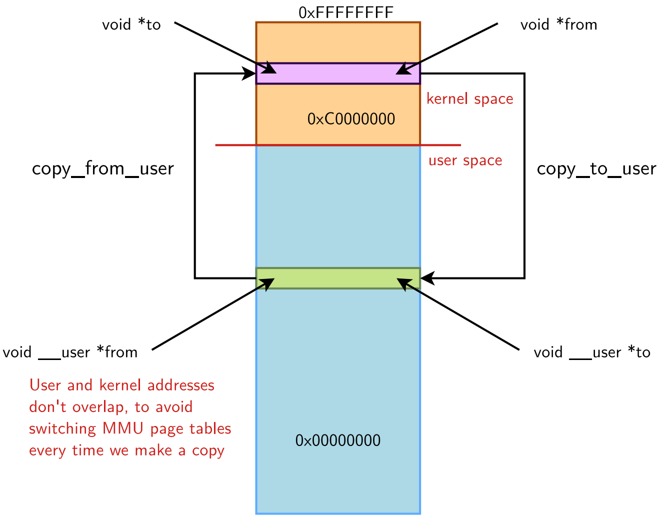
Exchanging data with user space 1/3

- Kernel code isn't allowed to directly access user space memory, using `memcpy()` or direct pointer dereferencing
  - User pointer dereferencing is disabled by default to make it harder to exploit vulnerabilities.
  - If the address passed by the application was invalid, the kernel could segfault.
  - **Never** trust user space. A malicious application could pass a kernel address which you could overwrite with device data (`read` case), or which you could dump to the device (`write` case).
  - Doing so does not work on some architectures anyway.
- To keep the kernel code portable, secure, and have proper error handling, your driver must use special kernel functions to exchange data with user space.

Exchanging data with user space 2/3

- A single value
  - `get_user(v, p);`
    - The kernel variable `v` gets the value pointed by the user space pointer `p`
  - `put_user(v, p);`
    - The value pointed by the user space pointer `p` is set to the contents of the kernel variable `v`.
- A buffer
  - `unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);`
  - `unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);`
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

Exchanging data with user space 3/3



User and kernel addresses don't overlap, to avoid switching MMU page tables every time we make a copy

Zero copy access to user memory

- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).
- *Zero copy* options are possible:
  - `mmap()` system call to allow user space to directly access memory mapped I/O space. See our `mmap()` chapter.
  - `get_user_pages()` and related functions to get a mapping to user pages without having to copy them.

unlocked_ioctl()

- `long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`
  - Associated to the `ioctl()` system call.
  - Called unlocked because it didn't hold the Big Kernel Lock (gone now).
  - Allows to extend the driver capabilities beyond the limited read/write API.
  - For example: changing the speed of a serial port, setting video output format, querying a device serial number... Used extensively in the V4L2 (video) and ALSA (sound) driver frameworks.
  - `cmd` is a number identifying the operation to perform.
    See `driver-api/ioctl` for the recommended way of choosing `cmd` numbers.
  - `arg` is the optional argument passed as third argument of the `ioctl()` system call. Can be an integer, an address, etc.
  - The semantic of `cmd` and `arg` is driver-specific.

ioctl() example: kernel side

```c
#include <linux/phantom.h>

static long phantom_ioctl(struct file *file, unsigned int cmd,
    unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    default:
        return -ENOTTY;
    }

    return 0;
}
```

Selected excerpt from drivers/misc/phantom.c

ioctl() Example: Application Side

```c
#include <linux/phantom.h>

int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, &reg);
    assert(ret == 0);

    return 0;
}
```

Allocating a major / minor number

Before registering a character device, it is necessary to obtain a major and minor number:

- A major / minor number is stored in an opaque type `dev_t` (currently a `u32`). It should not be accessed directly! Instead:
  - Create a `dev_t` using `MKDEV()`.
  - Retrieve the major / minor number with `MAJOR()` and `MINOR()`.
- Some major / minor numbers are statically allocated. In this case, `register_chrdev_region()` is used. See the list of registered numbers in `admin-guide/devices`.
- It is recommended to allocate a `dev_t` dynamically instead:
  - Allocate one (or several) number(s) using `alloc_chrdev_region()`.
  - Release it using `unregister_chrdev_region()`.

Registering a character device

The registration of a character device is done using a `struct cdev`:

- Either it can be dynamically allocated with `cdev_alloc()`.
- Or, if the structure is static, it should be initialized using `cdev_init()`.
- In both cases, a pointer to the `struct file_operations` is stored inside `struct cdev`.

Finally the registration is performed using `cdev_add()`. When the character device is no longer needed, use `cdev_del()` to delete it.

Registering a character device: example

```
static dev_t first;

static int __init dax_attach(void)
{
    ...
    if (alloc_chrdev_region(&first, 0, 1, DAX_NAME) < 0) {
        dax_err("alloc_chrdev_region failed");
        ret = -ENXIO;
        goto done;
    }
    cdev_init(&c_dev, &dax_fops);
    if (cdev_add(&c_dev, first, 1) == -1) {
        dax_err("cdev_add failed");
        ret = -ENXIO;
        goto cdev_error;
    }
    ...
}
```

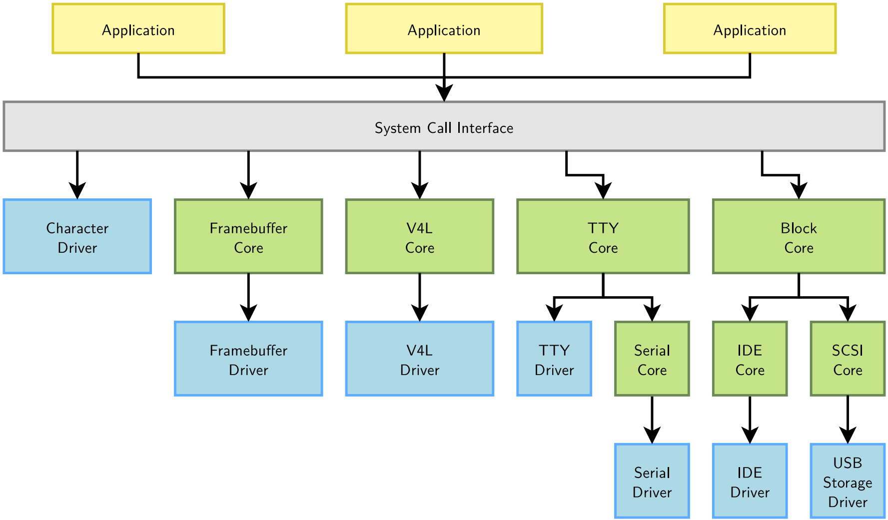
Selected excerpt from `drivers/sbus/char/oradax.c`

The concept of kernel frameworks

Beyond character drivers: kernel frameworks

- Many device drivers are not implemented directly as character drivers
- They are implemented under a *framework*, specific to a given device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From user space, they are still seen as character devices by the applications
  - The framework allows to provide a coherent user space interface (`ioctl`, etc.) for every type of device, regardless of the driver

Example: Some Kernel Frameworks



Example: Framebuffer Framework

- Kernel option CONFIG_FB
  - menuconfig FB
    - tristate "Support for frame buffer devices"
- Implemented in C files in drivers/video/fbdev/core/
- Defines the user/kernel API
  - include/uapi/linux/fb.h (constants and structures)
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
  - struct fb_ops
  - include/linux/fb.h

Framebuffer driver operations

Here are the operations a framebuffer driver can or must implement, and define them in a `struct fb_ops` structure (excerpt from `drivers/video/fbdev/skeletonfb.c`)

```
static struct fb_ops xxxfb_ops = {
    .owner = THIS_MODULE,
    .fb_open = xxxfb_open,
    .fb_read = xxxfb_read,
    .fb_write = xxxfb_write,
    .fb_release = xxxfb_release,
    .fb_check_var = xxxfb_check_var,
    .fb_set_par = xxxfb_set_par,
    .fb_setcolreg = xxxfb_setcolreg,
    .fb_blank = xxxfb_blank,
    .fb_pan_display = xxxfb_pan_display,
    .fb_fillrect = xxxfb_fillrect,             /* Needed !!! */
    .fb_copyarea = xxxfb_copyarea,             /* Needed !!! */
    .fb_imageblit = xxxfb_imageblit,           /* Needed !!! */
    .fb_cursor = xxxfb_cursor,                 /* Optional !!! */
    .fb_rotate = xxxfb_rotate,
    .fb_sync = xxxfb_sync,
    .fb_ioctl = xxxfb_ioctl,
    .fb_mmap = xxxfb_mmap,
};
```

Framebuffer driver code

- In the probe() function, registration of the framebuffer device and operations

```
static int xxxfb_probe (struct pci_dev *dev, const struct pci_device_id *ent)
{
    struct fb_info *info;
    [...]
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    [...]
    info->fbops = &xxxfb_ops;
    [...]
    if (register_framebuffer(info) < 0)
        return -EINVAL;
    [...]
}
```

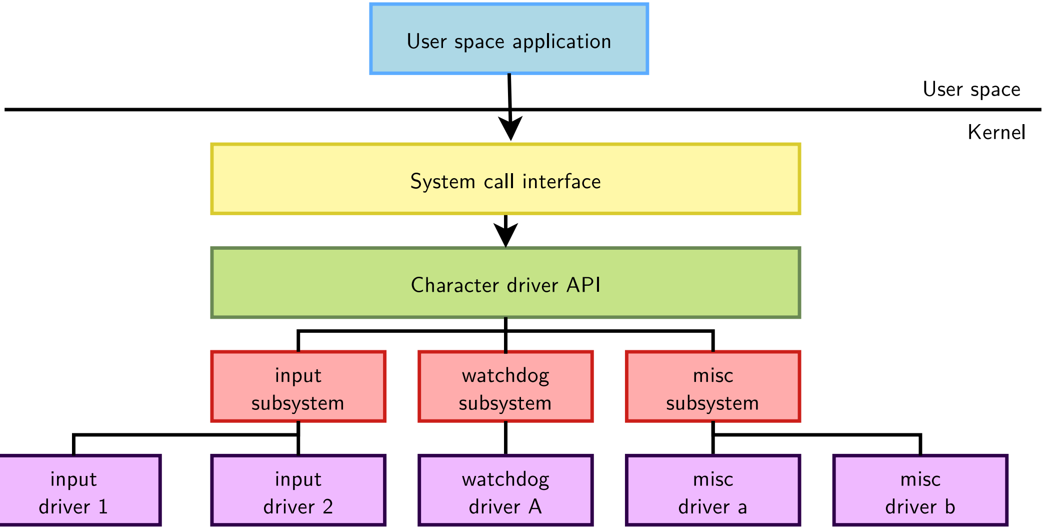
- register_framebuffer() will create a new character device in *devtmpfs* that can be used by user space applications with the generic framebuffer API.

The misc subsystem

Why a *misc* subsystem?

- The kernel offers a large number of **frameworks** covering a wide range of device types: input, network, video, audio, etc.
  - These frameworks allow to factorize common functionality between drivers and offer a consistent API to user space applications.
- However, there are some devices that **really do not fit in any of the existing frameworks**.
  - Highly customized devices implemented in a FPGA, or other weird devices for which implementing a complete framework is not useful.
- The drivers for such devices could be implemented directly as raw *character drivers* (with `cdev_init()` and `cdev_add()`).
- But there is a subsystem that makes this work a little bit easier: the **misc subsystem**.
  - It is really only a **thin layer** above the *character driver* API.
  - Another advantage is that devices are integrated in the Device Model (device files appearing in *devtmpfs*, which you don't have with raw character devices).

Misc subsystem diagram



Misc subsystem API (1/2)

- The misc subsystem API mainly provides two functions, to register and unregister **a single** *misc device*:
  - `int misc_register(struct miscdevice * misc);`
  - `void misc_deregister(struct miscdevice *misc);`
- A *misc device* is described by a `struct miscdevice` structure:

```
struct miscdevice  {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        umode_t mode;
};
```

Misc subsystem API (2/2)

The main fields to be filled in `struct miscdevice` are:

- `minor`, the minor number for the device, or `MISC_DYNAMIC_MINOR` to get a minor number automatically assigned.
- `name`, name of the device, which will be used to create the device node if *devtmpfs* is used.
- `fops`, pointer to the same `struct file_operations` structure that is used for raw character drivers, describing which functions implement the *read*, *write*, *ioctl*, etc. operations.
- `parent`, pointer to the `struct device` of the underlying "physical" device (platform device, I2C device, etc.)

User space API for misc devices

- *misc devices* are regular character devices
- The operations they support in user space depends on the operations the kernel driver implements:
  - The `open()` and `close()` system calls to open/close the device.
  - The `read()` and `write()` system calls to read/write to/from the device.
  - The `ioctl()` system call to call some driver-specific operations.

Example of a misc device

```c
static const struct file_operations adi_fops = {
    .read         = adi_read,
    .write        = adi_write,
    ...
};
static struct miscdevice adi_miscdev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = KBUILD_MODNAME,
    .fops = &adi_fops,
};
static int __init adi_init(void)
{
    return misc_register(&adi_miscdev);
}
static void __exit adi_exit(void)
{
    misc_deregister(&adi_miscdev);
}
module_init(adi_init);
module_exit(adi_exit);
```

Selected excerpt from `drivers/char/adi.c`

Driver data structures and links

Driver-specific Data Structure

- Each *framework* defines a structure that a device driver must register to be recognized as a device in this framework
  - `struct uart_port` for serial ports, `struct net_device` for network devices, `struct fb_info` for framebuffers, etc.
- In addition to this structure, the driver usually needs to store additional information about each device
- This is typically done
  - By subclassing the appropriate framework structure
  - By storing a reference to the appropriate framework structure
  - Or by including your information in the framework structure

Driver-specific Data Structure Examples 1/2

- i.MX serial driver: `struct imx_port` is a subclass of `struct uart_port`

```
struct imx_port {
    struct uart_port port;
    struct timer_list timer;
    unsigned int old_status;
    int txirq, rxirq, rtsirq;
    unsigned int have_rtscts:1;
    [...]
};
```

- ds1305 RTC driver: `struct ds1305` has a reference to `struct rtc_device`

```
struct ds1305 {
    struct spi_device       *spi;
    struct rtc_device       *rtc;
    [...]
};
```

Driver-specific Data Structure Examples 2/2

- rtl8150 network driver: `struct rtl8150` has a reference to `struct net_device` and is allocated within that framework structure.

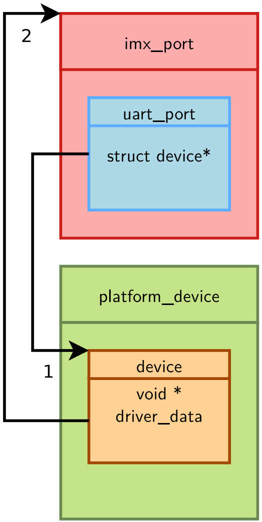
```
struct rtl8150 {
    unsigned long flags;
    struct usb_device *udev;
    struct tasklet_struct tl;
    struct net_device *netdev;
    [...]
};
```

Links between structures 1/4

- The framework structure typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
  - It's the relationship between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- The device structure also contains a `void *` pointer that the driver can freely use.
  - It's often used to link back the device to the higher-level structure from the framework.
  - It allows, for example, from the `struct platform_device` structure, to find the structure describing the logical device

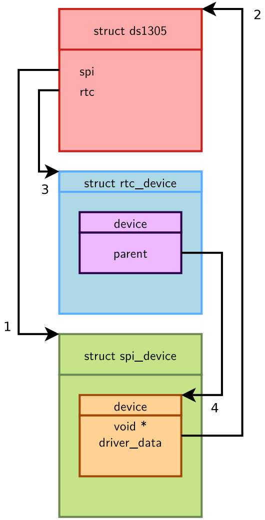
Links between structures 2/4

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport; /* per device structure */
    [...]
    sport = devm_kzalloc(&pdev->dev, sizeof(*sport), GFP_KERNEL);
    [...]
    /* setup the link between uart_port and the struct
     * device inside the platform_device */
    sport->port.dev = &pdev->dev;                          // Arrow 1
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, sport);                     // Arrow 2
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrieve the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    [...]
    uart_remove_one_port(&imx_reg, &sport->port);
    [...]
}
```



Links between structures 3/4

```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305                   *ds1305;

    [...]

    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
            return -ENOMEM;
    ds1305->spi = spi;                     // Arrow 1
    spi_set_drvdata(spi, ds1305);          // Arrow 2

    [...]

    ds1305->rtc = devm_rtc_allocate_device(&spi->dev);
                                           // Arrows 3 and 4

    [...]
}

static int ds1305_remove(struct spi_device *spi)
{
    struct ds1305 *ds1305 = spi_get_drvdata(spi);

    [...]
}
```



Links between structures 4/4

```c
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    [...]

    dev->udev = udev;      // Arrow 1
    dev->netdev = netdev;  // Arrow 2

    [...]

    usb_set_intfdata(intf, dev);        // Arrow 3
    SET_NETDEV_DEV(netdev, &intf->dev); // Arrow 4

    [...]
}

static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

    [...]
}
```