Linux Kernel architecture for device drivers

Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com

Thomas Petazzoni, Free-Electrons

- Free Electrons is a company specialized in Embedded Linux. It offers
  - **development services and consulting**: board support package development, kernel and driver development, embedded Linux system integration
  - **training**: device driver development in the Linux kernel, embedded Linux system development
- Thomas Petazzoni
  - Embedded Linux engineer and trainer at Free Electrons since January 2008
  - Currently works on OMAP Power Management for TI
  - Major contributor to Buildroot, a simple and fast embedded Linux build system
  - Also developer of MapOSMatic (talk on Friday!)

Agenda

- Userspace vision: different types of devices
- Implementation of basic character drivers
- Kernel “frameworks” for device drivers
  - General concept
  - Example of the framebuffer and serial ports frameworks
- The device model
  - General concept
  - Focus on an USB network driver
  - Platform drivers

Different types of devices

Userspace sees three main types of devices:

1. **Character devices** is the most common type of devices. Initially for devices implementing streams of bytes, it is now used for a wide range of devices: serial ports, framebuffers, video capture devices, sound devices, input devices, I2C and SPI gateways, etc.
2. **Block devices** for storage devices like hard disks, CD-ROM drives, USB keys, SD/MMC cards, etc.
3. **Network devices** for wired or wireless interfaces, PPP connections and others

Accessing the devices

- *Network devices* are accessed through **network-specific APIs and tools** (socket API of the standard C library, tools such as `ifconfig`, `route`, etc.)
- *Block and character devices* are represented for userspace applications as files than can be manipulated using the traditional file API (`open()`, `read()`, `write()`, `close()`, etc.)
  - Special file types for block and character devices, associating a name with a couple *(major, minor)*
  - The kernel only cares about the *(type, major, minor)*, which is the unique identifier of the device
  - Special files traditionaly located in `/dev`, created by `mknod`, either manually or automatically by *udev*

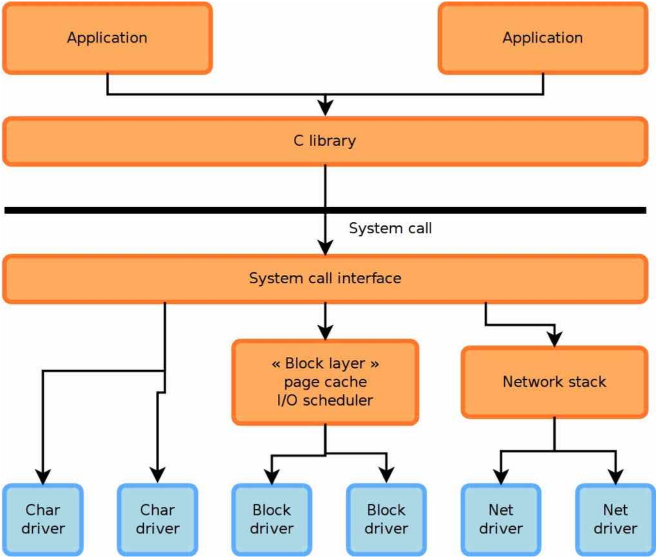
Inside the kernel

Device drivers must register themselves to the core kernel and implement a set of operations specific to their type:

- *Character drivers* must instantiate and register a `cdev` structure and implement `file_operations`
- *Block drivers* must instantiate and register a `gendisk` structure and implement `block_device_operations` and a special `make_request` function
- *Network drivers* must instantiate and register a `net_device` structure and implement `net_device_ops`

In this presentation, we will first focus on character devices as an example of device drivers.

General architecture



File operations

The file operations are generic to all types of files: regular files, directories, character devices, block devices, etc.

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

Character driver skeleton

Implement the `read()` and `write()` operations, and instantiate the `file_operations` structure.

```
static ssize_t demo_write(struct file *f, const char __user *buf,
                          size_t len, loff_t *off)
{
        [...]
}

static ssize_t demo_read(struct file *f, char __user *buf,
                         size_t len, loff_t *off)
{
        [...]
}

static struct file_operations demo_fops =
{
        .owner = THIS_MODULE,
        .read = acme_read,
        .write = acme_write
};
```

Character driver skeleton

Register and unregister the driver to the kernel using register_chrdev_region/unregister_chrdev_region and cdev_add/cdev_del.

```
static dev_t demo_dev = MKDEV(202,128);
static struct cdev demo_cdev;

static int __init demo_init(void)
{
    register_chrdev_region(demo_dev, 1, \demo");
    cdev_init(&demo_cdev, &demo_fops);
    cdev_add(&demo_cdev, demo_dev, demo_count);
}

static void __exit demo_exit(void)
{
    cdev_del(&demo_cdev);
    unregister_chrdev_region(demo_dev, 1);
    iounmap(demo_buf);
}

module_init(demo_init);
module_exit(demo_exit);
```

Driver usage in userspace

1. Making it accessible to userspace application by creating a device node: `mknod /dev/demo c 202 128`
2. Using normal the normal file API :

```
fd = open("/dev/demo", O_RDWR);

ret = read(fd, buf, bufsize);

ret = write(fd, buf, bufsize);
```

From the syscall to your driver

In fs/read_write.c

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
                fput_light(file, fput_needed);
        }

        return ret;
}
```

From the syscall to your driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_READ))
                return -EBADF;
        if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
                return -EINVAL;
        if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
                return -EFAULT;

        ret = rw_verify_area(READ, file, pos, count);
        if (ret >= 0) {
                count = ret;
                if (file->f_op->read)
                        ret = file->f_op->read(file, buf, count, pos);
                else
                        ret = do_sync_read(file, buf, count, pos);
                if (ret > 0) {
                        fsnotify_access(file->f_path.dentry);
                        add_rchar(current, ret);
                }
                inc_syscr(current);
        }

        return ret;
}
```

ioctl mechanism

- The file_operations set of operations, while being sufficient for regular files, isn't sufficient as an API to the wide range of character and block devices
- Device-specific operations such as changing the speed of a serial port, setting the volume on a soundcard, configuring video-related parameters on a framebuffer are not handled by the file_operations
- One of the operations, ioctl() allows to extend the capabilities of a driver with driver-specific operations
- In userspace: `int ioctl(int d, int request, ...);`
  - d, the file descriptor
  - request, a driver-specific integer identifying the operation
  - ..., zero or one argument.
- In kernel space: `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);`

ioctl example, kernel side

Implement the demo_ioctl() operation and reference it in the file_operations structure:

```
static int demo_ioctl(struct inode *inode,
      struct file *file,
      unsigned int cmd,
      unsigned long arg)

{
        char __user *argp = (char __user *)arg;

        switch (cmd) {

                case DEMO_CMD1:
                          /* Something */
                          return 0;

                default:
                          return -ENOTTY;
        }
}

static const struct file_operations demo_fops =
{
        [...]
        .ioctl = demo_ioctl,
        [...]
};
```

ioctl example, userspace side

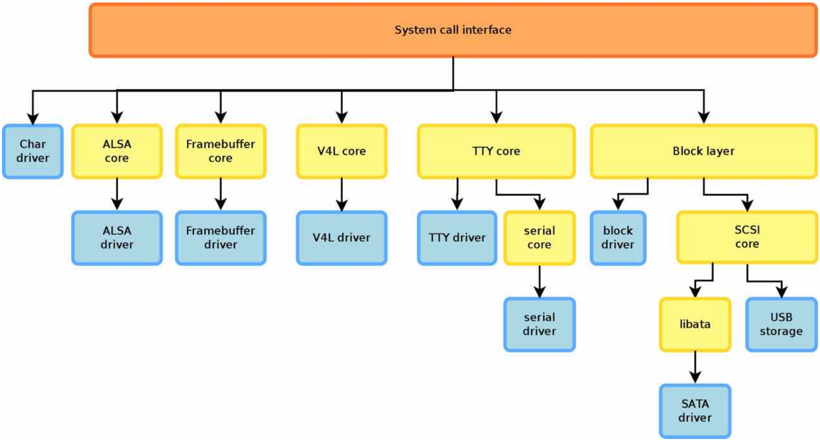
Use the ioctl() system call.

```
int fd, val;

fd = open("/dev/demo", O_RDWR);

ioctl(fd, DEMO_CMD1, & val);
```

Kernel framework

- Most device drivers are not directly implemented as character devices or block devices
- They are implemented under a *framework*, specific to a device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From userspace, they are still seen as normal character devices
  - The framework allows to provide a coherent userspace interface (ioctl numbering and semantic, etc.) for every type of device, regardless of the driver

Example of frameworks



Example of the framebuffer framework

- Kernel option CONFIG_FB
- Implemented in drivers/video/
  - fb.c, fbmem.c, fbmon.c, fbcmap.c, fbsysfs.c, modedb.c, fbcvt.c
- Implements a single character driver (through file_operations), registers the major number and allocates minors, defines and implements the user/kernel API
  - First part of include/linux/fb.h
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
  - struct fb_ops
  - Second part of include/linux/fb.h

The framebuffer driver

- Must implement some or all operations defined in `struct fb_ops`. Those operations are framebuffer-specific.
  - `xxx_open()`, `xxx_read()`, `xxx_write()`, `xxx_release()`, `xxx_checkvar()`, `xxx_setpar()`, `xxx_setcolreg()`, `xxx_blank()`, `xxx_pan_display()`, `xxx_fillrect()`, `xxx_copyarea()`, `xxx_imageblit()`, `xxx_cursor()`, `xxx_rotate()`, `xxx_sync()`, `xxx_get_caps()`, etc.
- Must allocate a `fb_info` structure with `framebuffer_alloc()`, set the `->fbops` field to the operation structure, and register the framebuffer device with `register_framebuffer()`

Skeleton example

```
static int xxx_open(struct fb_info *info, int user) {}
static int xxx_release(struct fb_info *info, int user) {}
static int xxx_check_var(struct fb_var_screeninfo *var, struct fb_info *info) {}
static int xxx_set_par(struct fb_info *info) {}

static struct fb_ops xxx_ops = {
        .owner          = THIS_MODULE,
        .fb_open        = xxxfb_open,
        .fb_release     = xxxfb_release,
        .fb_check_var   = xxxfb_check_var,
        .fb_set_par     = xxxfb_set_par,
        [...]
};

init()
{
    struct fb_info *info;
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    info->fbops = &xxxfb_ops;
    [...]
    register_framebuffer(info);
}
```

Other example of framework: serial driver

1. The driver registers a single `uart_driver` structure, that contains a few informations such as major, starting minor, number of supported serial ports, etc.
   - Functions `uart_register_driver()` and `uart_unregister_driver()`
2. For each serial port detected, the driver registers a `uart_port` structure, which points to a `uart_ops` structure and contains other informations about the serial port
   - Functions `uart_add_one_port()` and `uart_remove_one_port()`
3. The driver implements some or all of the methods in the `uart_ops` structure
   - `tx_empty()`, `set_mctrl()`, `get_mctrl()`, `stop_tx()`, `start_tx()`, `send_xchar()`, `stop_rx()`, `enable_ms()`, `break_ctl()`, `startup()`, `shutdown()`, `flush_buffer()`, `set_termios()`, etc.
   - All these methods receive as argument at least a `uart_port` structure, the device on which the method applies. It is similar to the `this` pointer in object-oriented languages

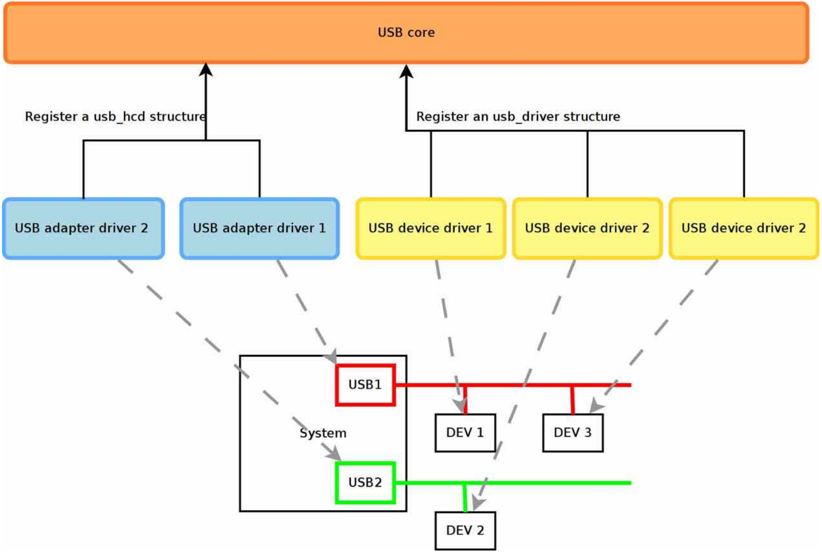
Device and driver model

- One of the features that came with the 2.6 kernel is a *unified device and driver model*
- Instead of different ad-hoc mechanisms in each subsystem, the device model unifies the vision of the devices, drivers, their organization and relationships
- Allows to minimize code duplication, provide common facilities, more coherency in the code organization
- Defines base structure types: `struct device`, `struct driver`, `struct bus_type`
- Is visible in userspace through the `sysfs` filesystem, traditionnaly mounted under `/sys`

Bus driver

- Core element of the device model
- A single bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- This driver is responsibles for
  - Registering the bus type (`bus_type` structure)
  - Allow the registration of adapter/interface drivers (USB controllers, I2C controllers, SPI controllers). These are the hardware devices capable of detecting and providing access to the devices connected to the bus
  - Allow the registration of device drivers (USB devices, I2C devices, SPI devices). These are the hardware devices connected to the different buses.
  - Matching the device drivers against the detected devices

Adapter, bus and device drivers



Example of device driver

To illustrate how drivers are implemented to work with the device model, we will use an USB network adapter driver. We will therefore limit ourselves to *device drivers* and won't cover *adapter drivers*.

Device identifiers

- Defines the set of devices that this driver can manage, so that the USB core knows which devices this driver can handle.
- The `MODULE_DEVICE_TABLE` macro allows `depmod` to extract at compile the relation between device identifiers and drivers, so that drivers can be loaded automatically by *udev*. See `/lib/modules/$(uname -r)/modules.{alias,usbmap}`.

```
static struct usb_device_id rtl8150_table[] = {
        {USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150)},
        {USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX)},
        {USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR)},
        {USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX)},
        {USB_DEVICE(VENDOR_ID_OQO, PRODUCT_ID_RTL8150)},
        {USB_DEVICE(VENDOR_ID_ZYXEL, PRODUCT_ID_PRESTIGE)},
        {}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instantation of `usb_driver`

- Instantiates the `usb_driver` structure. This structure is a specialization of `struct driver` defined by the driver model. We have an example of *inheritance* here.

```
static struct usb_driver rtl8150_driver = {
        .name =         "rtl8150",
        .probe =        rtl8150_probe,
        .disconnect =   rtl8150_disconnect,
        .id_table =     rtl8150_table,
        .suspend =      rtl8150_suspend,
        .resume =       rtl8150_resume
};
```

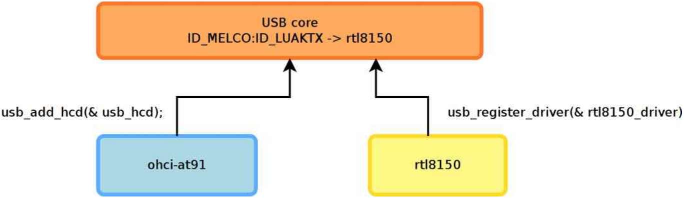
Registration of the driver

When the driver is loaded and unloaded, it simply registers and unregisters itself as an USB device driver.

```
static int __init usb_rtl8150_init(void)
{
        return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
        usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

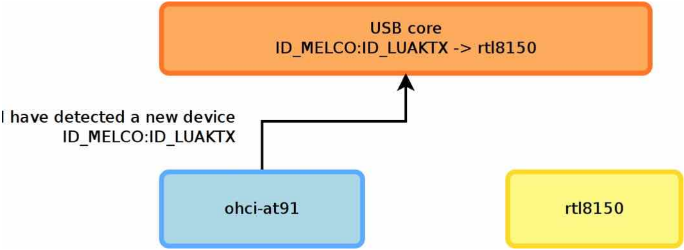
Probe call sequence (1/3)

At boot time, the USB device driver registers itself to the generic BUS infrastructure



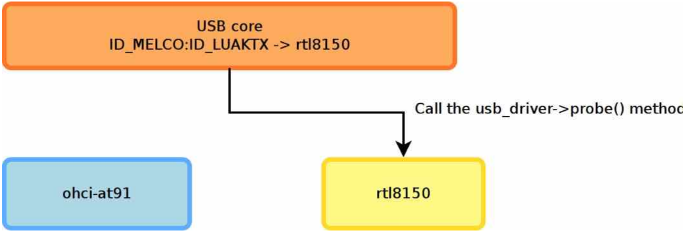
Probe call sequence (2/3)

When a bus adapter driver detects a device, it notifies the generic USB bus infrastructure



Probe call sequence (3/3)

The generic USB bus infrastructure knows which driver is capable of handling the detected device. It calls the `probe()` method of that driver



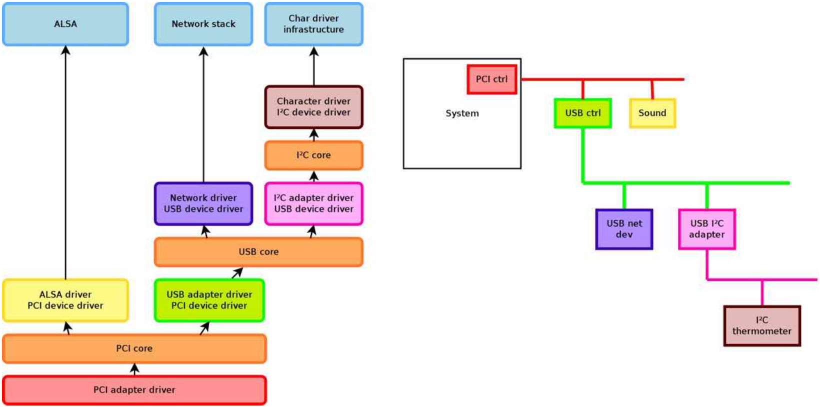
Probe method

- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`pci_dev`, `usb_interface`, etc.)
- This function is responsible for
  - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupts numbers and other device-specific information.
  - Registering the device to the proper kernel framework, for example the network infrastructure.

rtl8150_probe

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

        netdev = alloc_etherdev(sizeof(rtl8150_t));

        dev = netdev_priv(netdev);
        tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
        spin_lock_init(&dev->rx_pool_lock);

        netdev->netdev_ops = &rtl8150_netdev_ops;
        alloc_all_urbs(dev);

        usb_set_intfdata(intf, dev);
        SET_NETDEV_DEV(netdev, &intf->dev);
        register_netdev(netdev);

        return 0;
}
```

The model is recursive



Platform drivers

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- However, we still want the devices to be part of the device model.
- The solution to this is the platform driver / platform device infrastructure.
- The platform devices are the devices that are directly connected to the CPU, without any kind of bus.

Initialization of a platform driver

Example of the *iMX* serial port driver, in `drivers/serial/imx.c`. The driver instantiates a `platform_driver` structure:

```
static struct platform_driver serial_imx_driver = {
        .probe          = serial_imx_probe,
        .remove         = serial_imx_remove,
        .driver         = {
                .name   = "imx-uart",
                .owner  = THIS_MODULE,
        },
};
```

And registers/unregisters it at init/cleanup:

```
static int __init imx_serial_init(void)
{
        platform_driver_register(&serial_imx_driver);
}
static void __ext imx_serial_cleanup(void)
{
        platform_driver_unregister(&serial_imx_driver);
}
```

Instantiation of a platform device

As platform devices cannot be detected dynamically, they are statically defined:

- by direct instantiation of `platform_device` structures, as done on ARM
- by using a *device tree*, as done on PowerPC

Example on ARM, where the instantiation is done in the board specific code (`arch/arm/mach-imx/mx1ads.c`)

```
static struct platform_device imx_uart1_device = {
        .name           = "imx-uart",
        .id             = 0,
        .num_resources  = ARRAY_SIZE(imx_uart1_resources),
        .resource       = imx_uart1_resources,
        .dev = {
                .platform_data = &uart_pdata,
        }
};
```

The matching between a device and the driver is simply done using the name.

Registration of platform devices

The device is part of a list:

```
static struct platform_device *devices[] __initdata = {
        &cs89x0_device,
        &imx_uart1_device,
        &imx_uart2_device,
};
```

And the list of devices is added to the system during the board initialization

```
static void __init mx1ads_init(void)
{
        [...]
        platform_add_devices(devices, ARRAY_SIZE(devices));
        [...]
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
        [...]
        .init_machine   = mx1ads_init,
MACHINE_END
```

The *resource* mechanism

- Each device managed by a particular driver typically uses different hardware resources: different addresses for the I/O registers, different DMA channel, different IRQ line, etc.
- These informations can be represented using the kernel `struct resource`, and an array of *resources* is associated to a *platform device* definition.

```
static struct resource imx_uart1_resources[] = {
        [0] = {
                .start  = 0x00206000,
                .end    = 0x002060FF,
                .flags  = IORESOURCE_MEM,
        },
        [1] = {
                .start  = (UART1_MINT_RX),
                .end    = (UART1_MINT_RX),
                .flags  = IORESOURCE_IRQ,
        },
};
```

The `platform_data` mechanism

- In addition to the well-defined *resources*, some driver require driver-specific configuration for each platform device
- These can be specified using the `platform_data` field of the `struct device`
- As it is a `void *` pointer, it can be used to pass any type of data to the driver
- In the case of the *iMX* driver, the platform data is a `struct imxuart_platform_data` structure, referenced from the `platform_device` structure

```
static struct imxuart_platform_data uart_pdata = {
        .flags = IMXUART_HAVE_RTSCTS,
};
```

Driver-specific data structure

- Typically, device drivers *subclass* the type-specific data structure that they must instantiate to register their device to the upper layer framework
- For example, serial drivers subclass `uart_port`, network drivers subclass `netdev`, framebuffer drivers subclass `fb_info`
- This *inheritance* is done by aggregation or by reference

```
struct imx_port {
        struct uart_port        port;
        struct timer_list       timer;
        unsigned int            old_status;
        int                     txirq,rxirq,rtsirq;
        unsigned int            have_rtscts:1;
        unsigned int            use_irda:1;
        unsigned int            irda_inv_rx:1;
        unsigned int            irda_inv_tx:1;
        unsigned short          trcv_delay; /* transceiver delay */
        struct clk              *clk;
};
```

probe() method for platform devices

- Just like the usual probe() methods, it receives the platform_device pointer, uses different utility functions to find the corresponding resources, and registers the device to the corresponding upper layer.

```
static int serial_imx_probe(struct platform_device *pdev)
{
        struct imx_port *sport;
        struct imxuart_platform_data *pdata;
        void __iomem *base;
        struct resource *res;

        sport = kzalloc(sizeof(*sport), GFP_KERNEL);
        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        base = ioremap(res->start, PAGE_SIZE);

        sport->port.dev = &pdev->dev;
        sport->port.mapbase = res->start;
        sport->port.membase = base;
        sport->port.type = PORT_IMX,
        sport->port.iotype = UPIO_MEM;
        sport->port.irq = platform_get_irq(pdev, 0);
        sport->rxirq = platform_get_irq(pdev, 0);
        sport->txirq = platform_get_irq(pdev, 1);
        sport->rtsirq = platform_get_irq(pdev, 2);

        [...]
```

probe() method for platform devices

```
        sport->port.fifosize = 32;
        sport->port.ops = &imx_pops;

        sport->clk = clk_get(&pdev->dev, "uart");
        clk_enable(sport->clk);
        sport->port.uartclk = clk_get_rate(sport->clk);

        imx_ports[pdev->id] = sport;

        pdata = pdev->dev.platform_data;
        if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
                sport->have_rtscts = 1;

        ret = uart_add_one_port(&imx_reg, &sport->port);
        if (ret)
                goto deinit;
        platform_set_drvdata(pdev, &sport->port);

        return 0;
}
```

Other non-dynamic busses

- In addition to the special *platform* bus, there are some other busses that do not support dynamic enumeration and identification of devices. For example: I2C and SPI.
- For these busses, a list of devices connected to the bus is hardcoded into the board-specific informations and is registered using `i2c_register_board_info()` or `spi_register_board_info()`. The binding between the device is also done using a string identifier.

```
static struct i2c_board_info pcm038_i2c_devices[] = {
	{	I2C_BOARD_INFO("at24", 0x52),
		.platform_data = &board_eeprom,  },
	{	I2C_BOARD_INFO("pcf8563", 0x51), },
	{	I2C_BOARD_INFO("lm75", 0x4a),    }
};

static void __init pcm038_init(void) {
	[...]
	i2c_register_board_info(0, pcm038_i2c_devices,
				ARRAY_SIZE(pcm038_i2c_devices));
	[...]
}
```

Typical organization of a driver

A driver typically

- Defines a **driver-specific data structure** to keep track of per-device state, this structure often subclass the type-specific structure for this type of device
- Implements a set of **helper functions**, interrupt handlers, etc.
- Implements some or all of the **operations**, as specified by the framework in which the device will be subscribed
- Instantiate the **operation table**
- Defines a `probe()` **method** that allocates the "state" structure, initializes the device and registers it to the upper layer framework. Similarly defines a corresponding `remove()` method
- Instantiate a `SOMEBUS_driver` **structure** that references the `probe()` and `remove()` methods and give the bus infrastructure some way of binding a device to this driver (by name, by identifier, etc.)
- In the **driver initialization function**, register as a device driver to the bus-specific infrastructure. In the **driver cleanup function**, unregister from the bus-specific infrastructure.

Conclusion

- The Linux kernel now has a coherent and uniform model to organize busses, drivers and devices. This model, and the Linux kernel in general, uses some concept of object-oriented programming to structure the code.
- The organization of device drivers has been greatly simplified and unified by using this model. Functionalities such as *udev* have been made possible using this unified model.

Questions ?