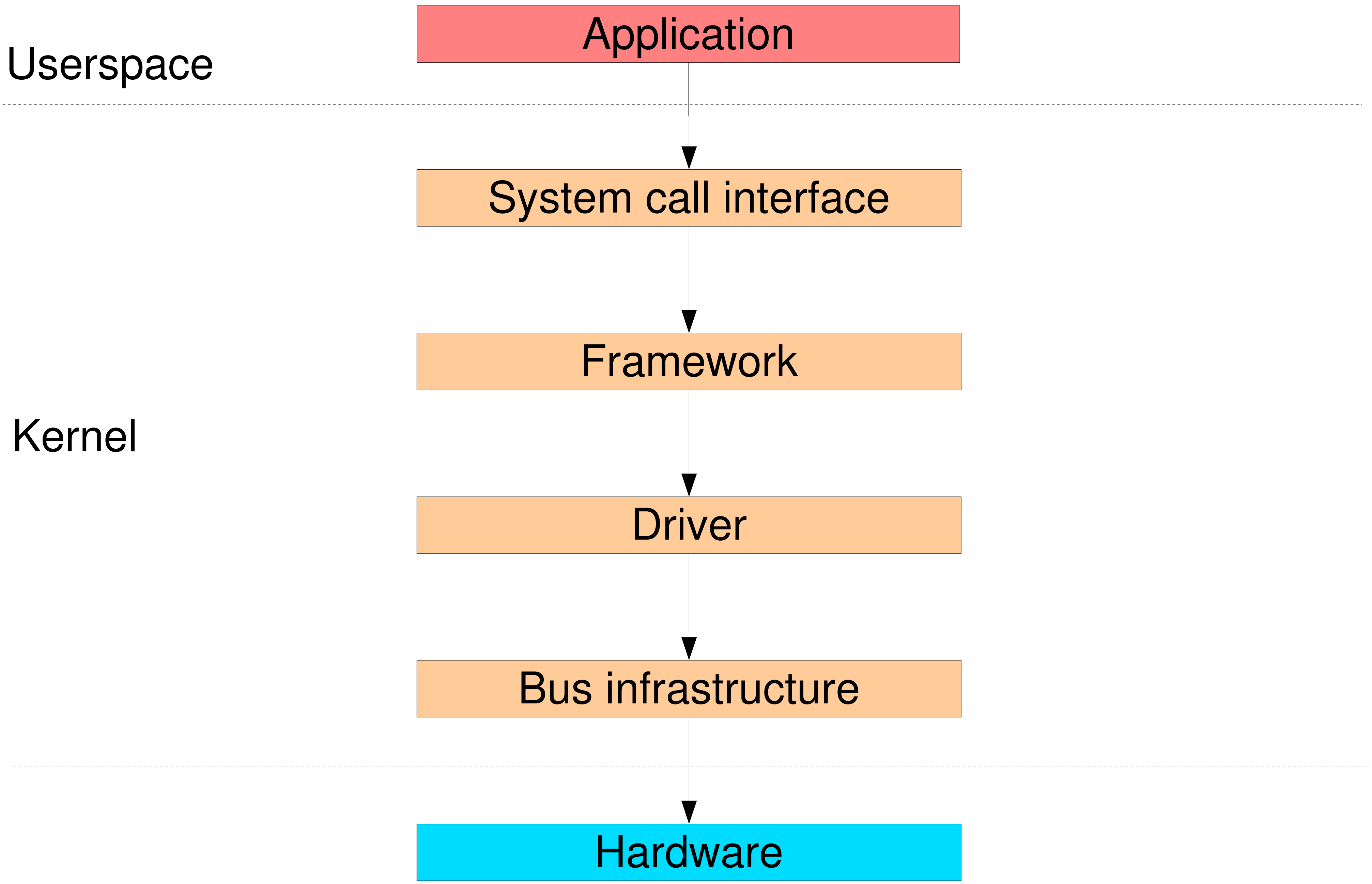
Driver development

Kernel architecture for device drivers

Kernel and device drivers

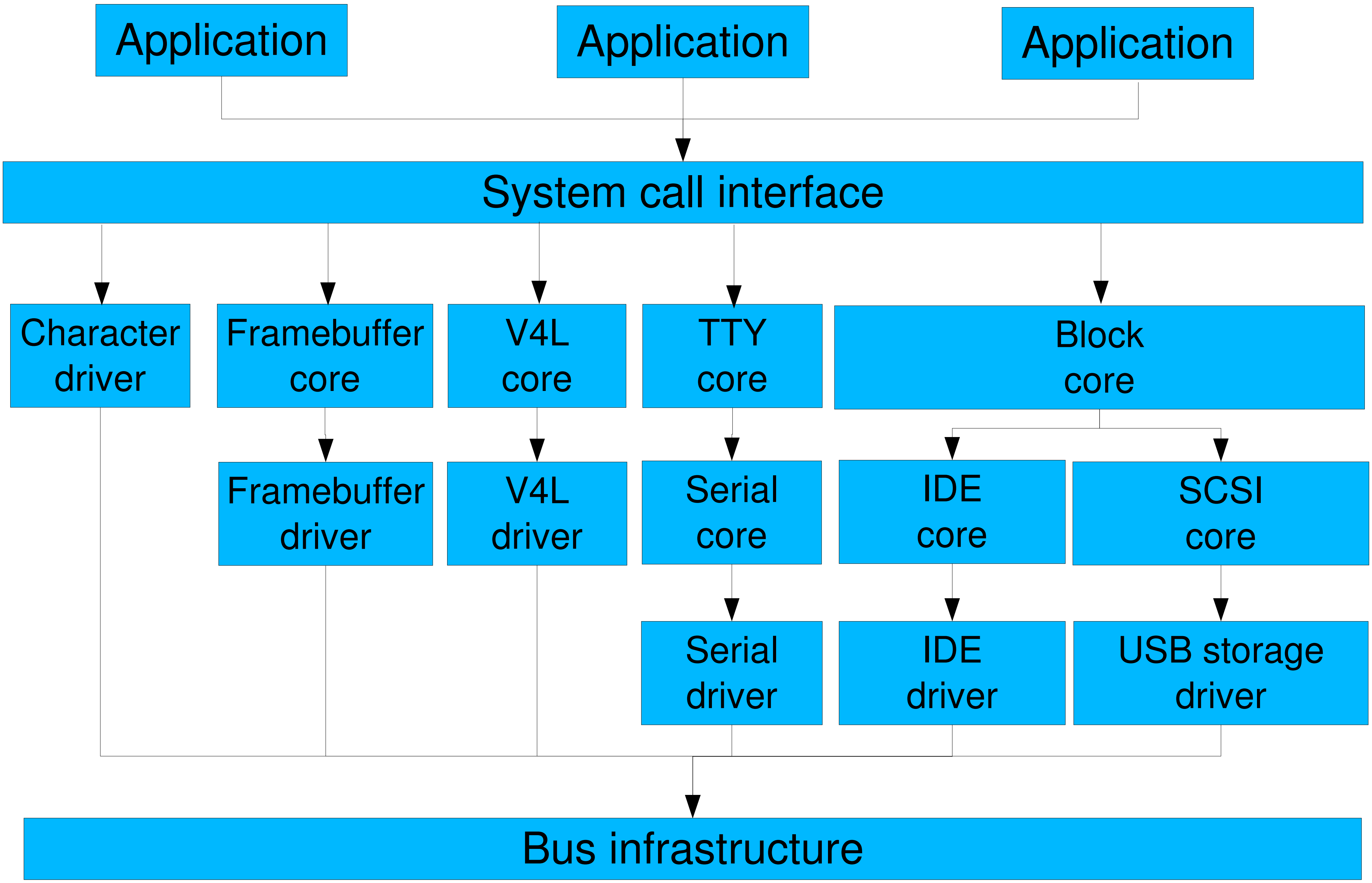


Kernel and device drivers

- 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 userspace, they are still seen as character devices by the applications
  - The framework allows to provide a coherent userspace interface (ioctl, etc.) for every type of device, regardless of the driver
- The device drivers rely on the « bus infrastructure » to enumerate the devices and communicate with them.

Kernel frameworks

« Frameworks »


Application
Application
Application
System call interface
Character driver
Framebuffer core
V4L core
TTY core
Block core
Framebuffer driver
V4L driver
Serial core
IDE core
SCSI core
Serial driver
IDE driver
USB storage driver
Bus infrastructure

Example: framebuffer framework

- Kernel option `CONFIG_FB`

```
menuconfig FB
        tristate "Support for frame buffer devices"
```

- Implemented in `drivers/video/`
  - `fb.c, fbmem.c, fbmon.c, fbcmap.c, fbsysfs.c, modedb.c, fbcvt.c`
- Implements a single character driver and defines 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` (in `#ifdef __KERNEL__`)

Framebuffer driver skeleton

- Skeleton driver in `drivers/video/skeletonfb.c`
- Implements the set of framebuffer specific operations defined by the `struct fb_ops` structure
  - `xxxfb_open()`
  - `xxxfb_read()`
  - `xxxfb_write()`
  - `xxxfb_release()`
  - `xxxfb_checkvar()`
  - `xxxfb_setpar()`
  - `xxxfb_setcolreg()`
  - `xxxfb_blank()`
  - `xxxfb_pan_display()`
  - `xxxfb_fillrect()`
  - `xxxfb_copyarea()`
  - `xxxfb_imageblit()`
  - `xxxfb_cursor()`
  - `xxxfb_rotate()`
  - `xxxfb_sync()`
  - `xxxfb_ioctl()`
  - `xxxfb_mmap()`

Framebuffer driver skeleton

- After the implementation of the operations, definition of a `struct fb_ops` structure

```
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 skeleton

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

```
static int __devinit 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 the character device that can be used by userspace application with the generic framebuffer API

Device Model and Bus Infrastructure

Unified device model

- The 2.6 kernel included a significant new feature: a unified device model
- Instead of having different ad-hoc mechanisms in the various subsystems, the device model unifies the description of the devices and their topology
- Minimizing code duplication
- Common facilities (reference counting, event notification, power management, etc.)
- Enumerate the devices, view their interconnections, link the devices to their buses and drivers, categorize them by classes.

Bus drivers

- The first component of the device model is the bus driver
- One bus driver for each type of bus: USB, PCI, SPI, MMC, ISA, etc.
- It is responsible for
  - Registering the bus type
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able of detecting the connected devices
  - Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
  - Matching the device drivers against the devices detected by the adapter drivers.

List of device identifiers

- Depending on the bus type, the method for binding a device to a driver is different. For many buses, it is based on unique identifiers
- The device driver defines a table with the list of device identifiers it is able to manage :

```
static const struct pci_device_id rhine_pci_tbl[] = {
        { 0x1106, 0x3043, PCI_ANY_ID, PCI_ANY_ID, },    /* VT86C100A */
        { 0x1106, 0x3053, PCI_ANY_ID, PCI_ANY_ID, },    /* VT6105M */
        { }     /* terminate list */
};
MODULE_DEVICE_TABLE(pci, rhine_pci_tbl);
```

Code on this slide and on the next slides are taken from the via-rhine driver in `drivers/net/via-rhine.c`

Defining the driver

- The device driver defines a driver structure, usually specialized by the bus infrastructure (`pci_driver`, `usb_driver`, etc.)
- The structure points to: the device table, a probe function, called when a device is detected and various other callbacks

```
static struct pci_driver rhine_driver = {
        .name           = DRV_NAME,
        .id_table       = rhine_pci_tbl,
        .probe          = rhine_init_one,
        .remove         = __devexit_p(rhine_remove_one),
#ifdef CONFIG_PM
        .suspend        = rhine_suspend,
        .resume         = rhine_resume,
#endif /* CONFIG_PM */
        .shutdown =     rhine_shutdown,
};
```

Registering the driver

- In the module initialization function, the driver is registered to the bus infrastructure, in order to let the bus know that the driver is available to handle devices.

```
static int __init rhine_init(void)
{
        [...]
        return pci_register_driver(&rhine_driver);
}
static void __exit rhine_cleanup(void)
{
        pci_unregister_driver(&rhine_driver);
}
```

- If a new PCI device matches one of the identifiers of the table, the `probe()` method of the PCI driver will get called.

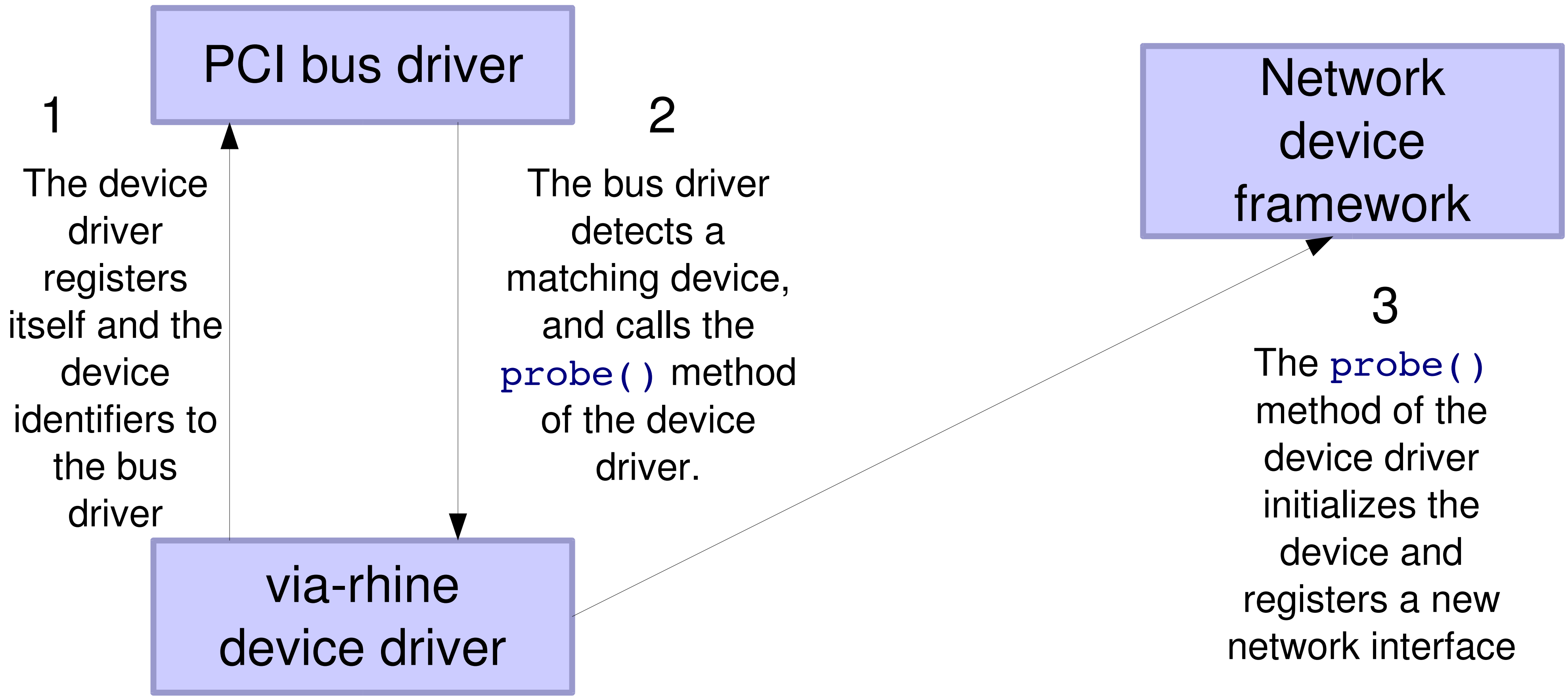
Probe method

- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`pci_dev`, `usb_device`, 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.

Device driver (5)

```
static int __devinit rhine_init_one(struct pci_dev *pdev,
                                    const struct pci_device_id *ent)
{
        struct net_device *dev;
       [...]
        rc = pci_enable_device(pdev);
       [...]
        pioaddr = pci_resource_start(pdev, 0);
        memaddr = pci_resource_start(pdev, 1);
       [...]
        dev = alloc_etherdev(sizeof(struct rhine_private));
       [...]
        SET_NETDEV_DEV(dev, &pdev->dev);
       [...]
        rc = pci_request_regions(pdev, DRV_NAME);
       [...]
        ioaddr = pci_iomap(pdev, bar, io_size);
       [...]
        rc = register_netdev(dev);
       [...]
}
```

Global architecture



sysfs

- The bus, device, drivers, etc. structures are internal to the kernel
- The `sysfs` virtual filesystem offers a mechanism to export such information to userspace
- Used for example by `udev` to provide automatic module loading, firmware loading, device file creation, etc.
- `sysfs` is usually mounted in `/sys`
- `/sys/bus/` contains the list of buses
- `/sys/devices/` contains the list of devices
- `/sys/class` enumerates devices by class (`net`, `input`, `block`...), whatever the bus they are connected to. Very useful!
- Take your time to explore `/sys` on your workstation.

Platform devices

- 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.

Implementation of the platform driver

- The driver implements a platform_driver structure (example taken from `drivers/serial/imx.c`)

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

- And registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void)
{
        [...]
        ret = platform_driver_register(&serial_imx_driver);
        [...]
}
```

Platform device instantiation (1)

- In the board-specific code, the platform devices are instantiated (`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 match between the device and the driver is made using the name. It must be unique amongst drivers !

Platform device instantiation (2)

- 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 board initialization

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

I/O resources

- Each platform device is associated with a set of I/O resources, referenced in the `platform_device` structure

```
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,
        },
};
```

- It allows the driver to be independent of the I/O addresses, IRQ numbers ! See `imx_uart2_device` for another device using the same platform driver.

Inside the platform driver

- When a platform_device is added to the system using `platform_add_device()`, the `probe()` method of the platform driver gets called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources :

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

sysfs tools

- http://linux-diag.sourceforge.net/Sysfsutils.html
- libsysfs - The library's purpose is to provide a consistent and stable interface for querying system device information exposed through sysfs.
- systool - A utility built upon libsysfs that lists devices by bus, class, and topology.

References

- Kernel documentation
  `Documentation/driver-model/`
  `Documentation/filesystems/sysfs.txt`
- Linux 2.6 Device Model
  http://www.bravegnu.org/device-model/device-model.html
- Linux Device Drivers, chapter 14 «The Linux Device Model»
  http://lwn.net/images/pdf/LDD3/ch14.pdf
- The kernel source code
  Full of examples of other drivers!

Framework and bus infrastructure

- A typical driver will
  - Be registered inside a framework
  - Rely on a bus infrastructure and the device model
- Example with the iMX serial driver, `drivers/serial/imx.c`
- At module initialization time, the driver registers itself both to the UART framework and to the platform bus infrastructure

```
static int __init imx_serial_init(void)
{
        ret = uart_register_driver(&imx_reg);
        [...]
        ret = platform_driver_register(&serial_imx_driver);
        [...]
        return 0;
}
```

iMX serial driver

Definition of the iMX UART driver

```
static struct uart_driver imx_reg = {
        .owner              = THIS_MODULE,
        .driver_name        = DRIVER_NAME,
        .dev_name           = DEV_NAME,
        .major              = SERIAL_IMX_MAJOR,
        .minor              = MINOR_START,
        .nr                 = ARRAY_SIZE(imx_ports),
        .cons               = IMX_CONSOLE,
};
```

Definition of the iMX platform driver

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

iMX serial driver

When the platform device is instantiated, the `probe()` method is called

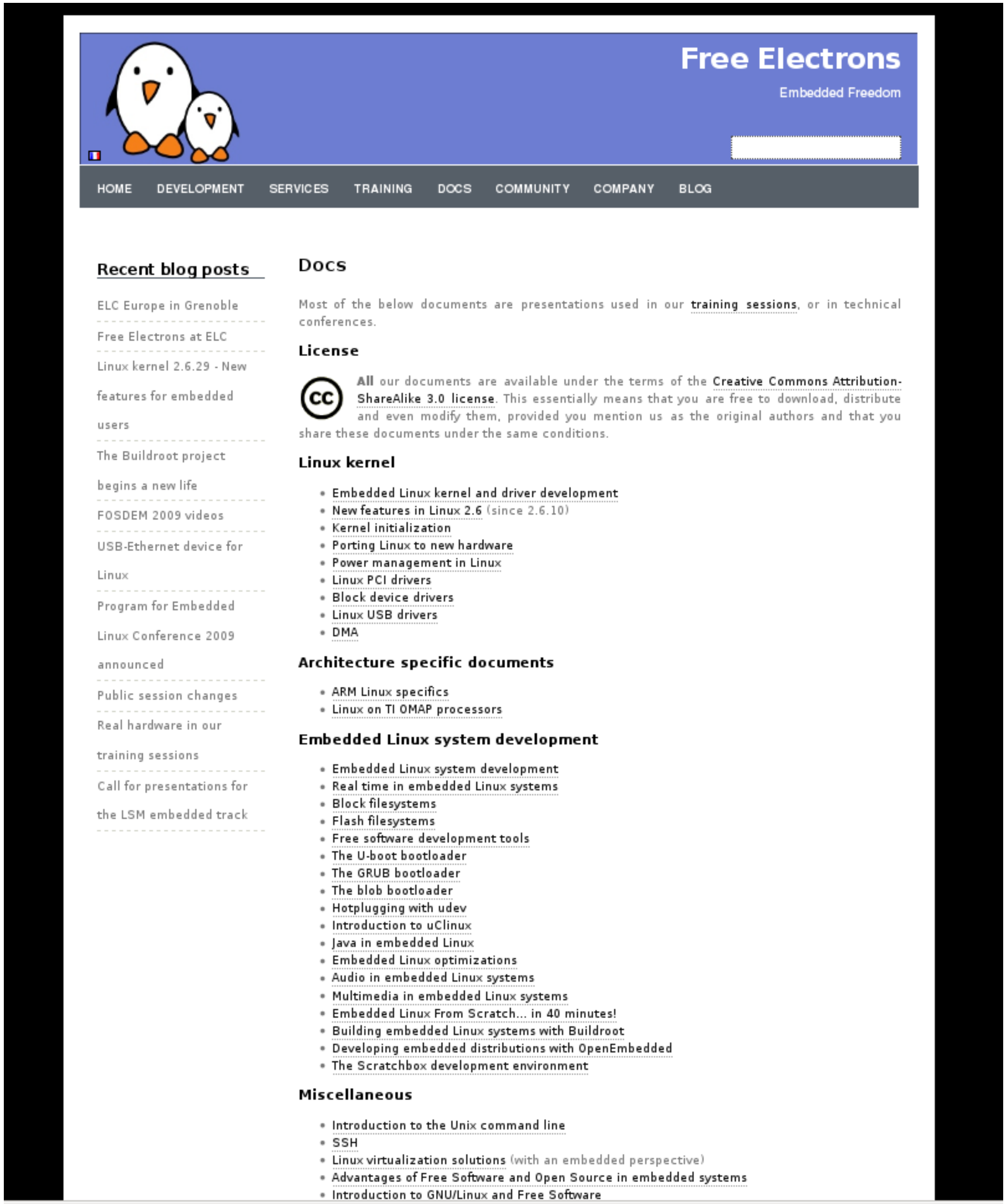
```
static int serial_imx_probe(struct platform_device *pdev)
{
        struct imx_port *sport;

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

        /* sport initialization */
        sport->port.irq = platform_get_irq(pdev, 0);
        sport->port.ops = &imx_pops;

        sport->clk = clk_get(&pdev->dev, "uart_clk");
        clk_enable(sport->clk);

        uart_add_one_port(&imx_reg, &sport->port);
}
```

iMX serial driver

The operation structure `uart_ops` is associated to each port. The operations are implemented in the iMX driver

```
static struct uart_ops imx_pops = {
        .tx_empty       = imx_tx_empty,
        .set_mctrl      = imx_set_mctrl,
        .get_mctrl      = imx_get_mctrl,
        .stop_tx        = imx_stop_tx,
        .start_tx       = imx_start_tx,
        .stop_rx        = imx_stop_rx,
        .enable_ms      = imx_enable_ms,
        .break_ctl      = imx_break_ctl,
        .startup        = imx_startup,
        .shutdown       = imx_shutdown,
        .set_termios    = imx_set_termios,
        .type           = imx_type,
        .release_port   = imx_release_port,
        .request_port   = imx_request_port,
        .config_port    = imx_config_port,
        .verify_port    = imx_verify_port,
};
```

Related documents


Free Electrons
Embedded Freedom

HOME DEVELOPMENT SERVICES TRAINING DOCS COMMUNITY COMPANY BLOG

Recent blog posts

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

Docs

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

License

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

Linux kernel

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

Architecture specific documents

- ARM Linux specifics
- Linux on TI OMAP processors

Embedded Linux system development

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

Miscellaneous

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

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

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

How to help

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

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

Free Electrons

Our services

Custom Development

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

Consulting and technical support

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

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

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