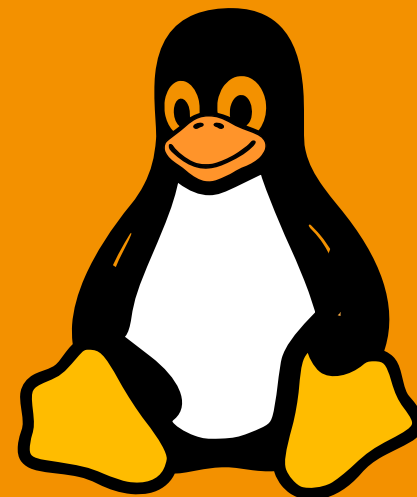




Prof. Dr. Florian Künzner

Technical University of Applied Sciences Rosenheim, Computer Science

OS 16 – Drivers

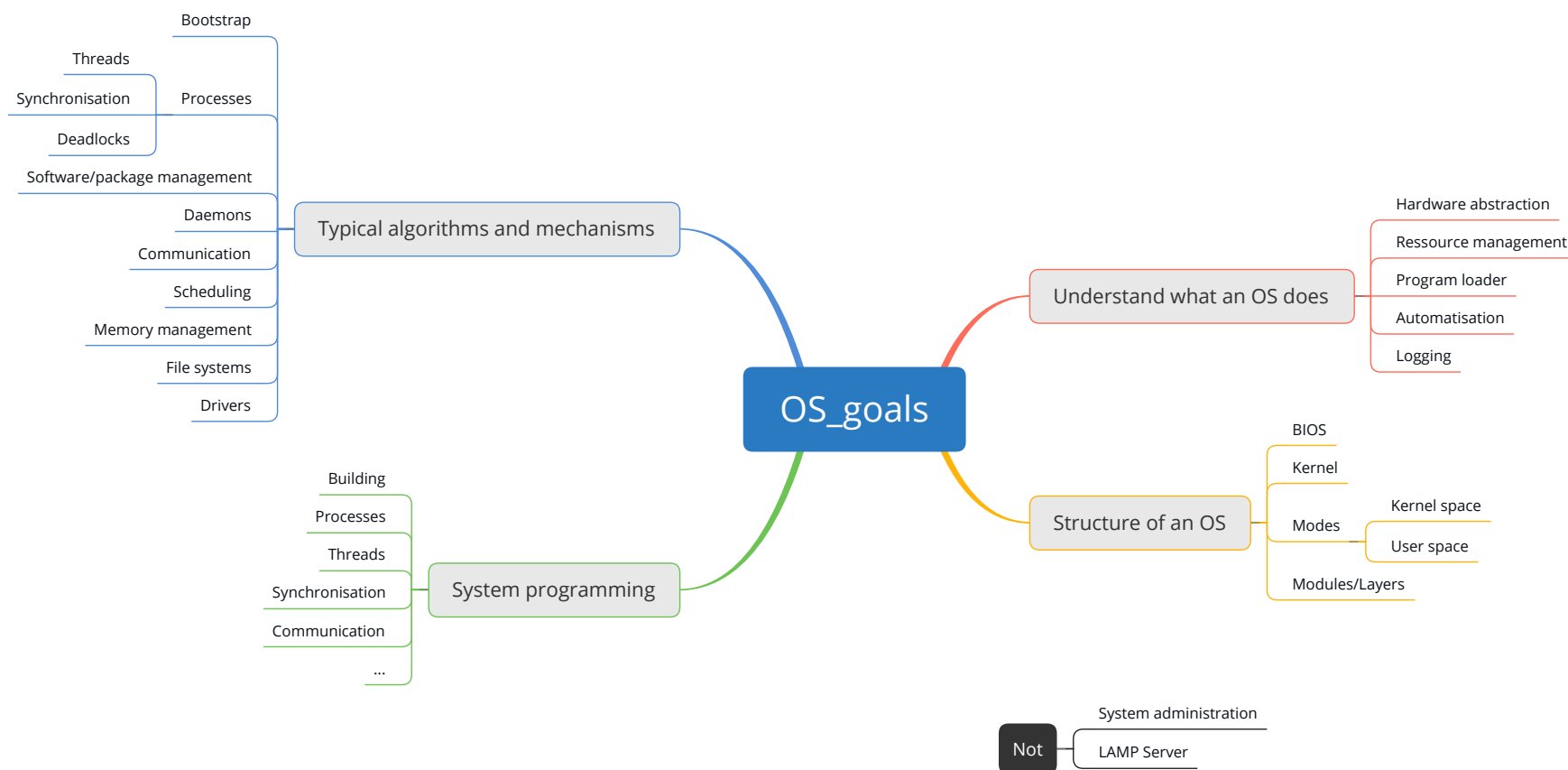


source: icons.png.com

The lecture is based on the work and the documents of Prof. Dr. Ludwig Frank



Goal



Goal

OS::Drivers

- Design goals
- Structure of I/O systems
- Linux device files
- Kernel modules
- Linux device driver development

Intro

What is a **device driver**?

Intro

A **device driver** is a piece of software that **controls the interaction** with an connected, built-in, or virtual device.

Static vs dynamic driver

A **static** driver is **built-in** into the kernel.

A **dynamic** driver can be **loaded and unloaded at runtime**.

Device classes

- Character device
- Block device
- USB
- Network
- Bluetooth
- FireWire (IEEE1394)
- SCSI
- IrDA (Infrared Data Association)
- Cardbus and PCMCIA
- Parallelport
- I2C
- ...

Design goals

User programs should be device independent

The type of the device or the position on the bus shouldn't be visible to the user application.

- Different hardware vendors
- Different device types: hard drive, SSD, USB stick, network drive, ...

Design goals

Uniform device names

Devices are in the /dev folder and should be independent of a hardware connection.

Example 1:

```
1 cat file > /dev/lp1 #sends file to printer on /dev/lp1
```

Example 2:

```
8 int main(void){
9     int fd = open("/dev/device", O_RDWR, 0666);
10
11     char buffer[LEN] = {"Hello\n"};
12     ssize_t bytes_written = write(fd, buffer, LEN);
13     ssize_t bytes_read    = read(fd, buffer, LEN);
14
15     close(fd);
16
17     return EXIT_SUCCESS;
18 }
```

Design goals

Error handling

- Localisation and report of errors
- Error logging
- Correction of transient errors (e.g. read again)
- Automatic notification of the maintenance service

Design goals

Support for synchronous and asynchronous data transfer

Synchronous

Process waits until the I/O operation completes.

Asynchronous

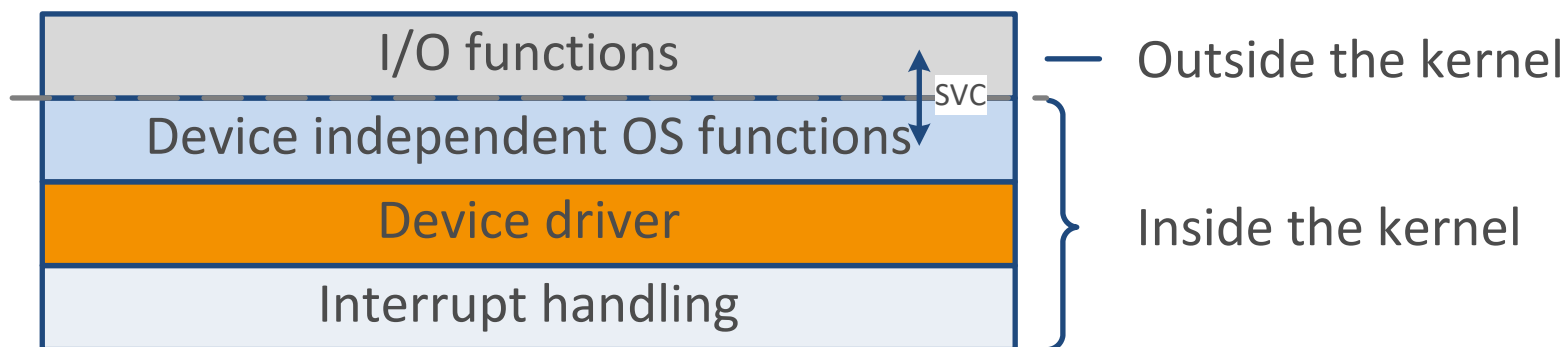
Process can proceed while the OS transfers the data.

Design goals

Device management

- Exclusive or shared access
- Locking mechanism (`fcntl()`)
- Deadlock monitoring (e.g. banker's algorithm)

Structure of I/O systems



Device independent OS functions

- Naming of device files (e.g. `/dev/fd0`)
- Access control
- Caching
- Locking mechanism
- Device allocation management
- Error handling
- Logging

Device driver

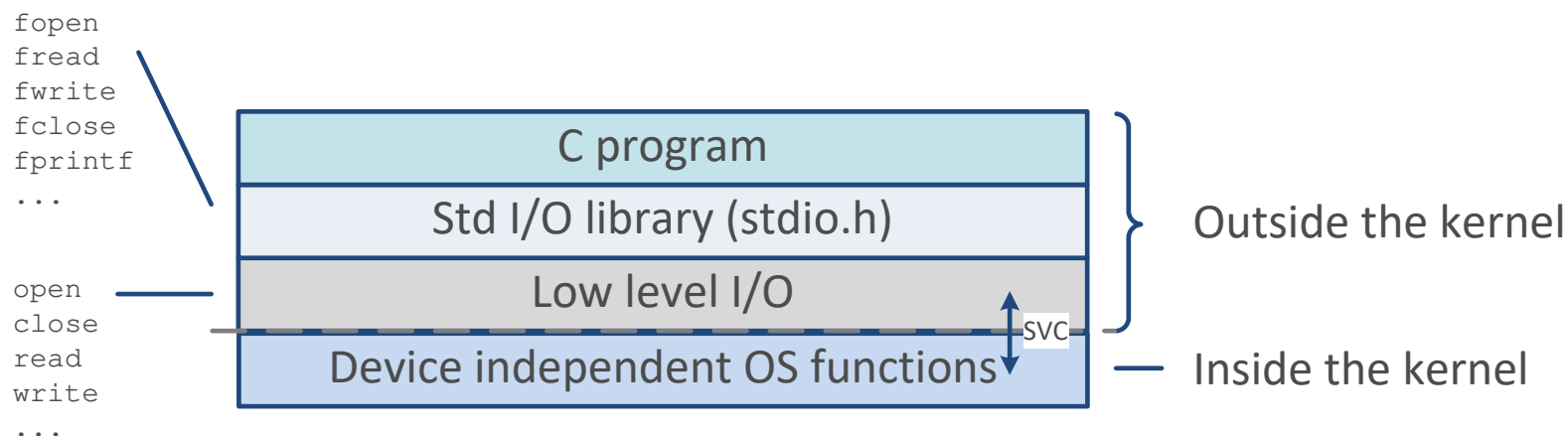
- Knows the HW properties of the devices
- Provides a HW independent interface

Interrupt handling

- Handling of interrupts
- Assignment to the waiting process



I/O layers

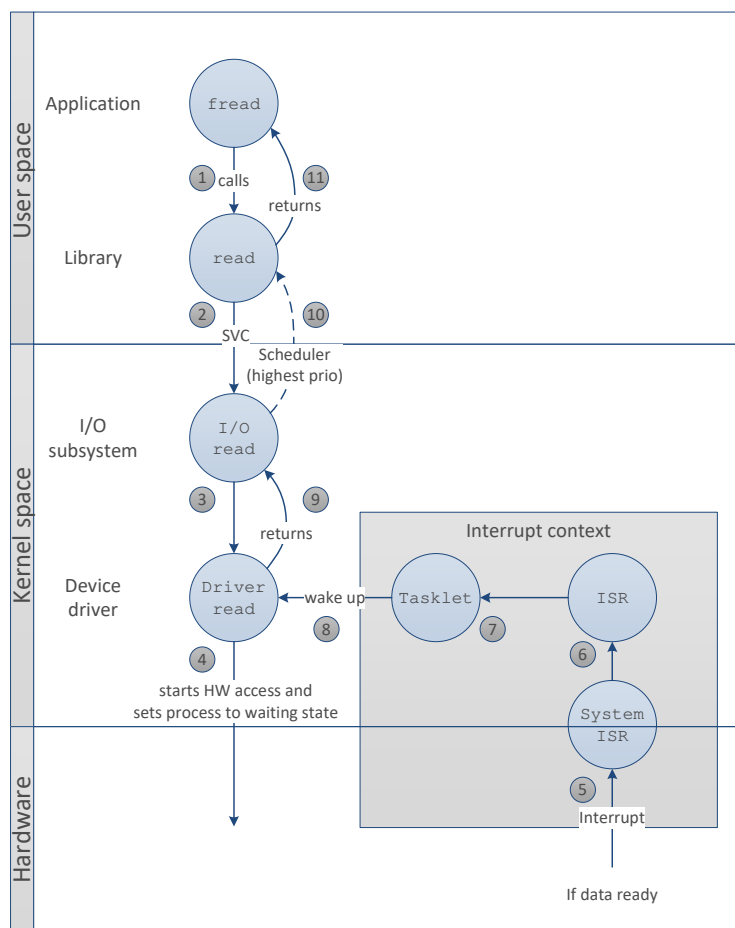


Functions of stdio.h

- Buffering
- Formatting
- Format conversion
- Higher level of abstraction (easier to use)



Example operation: read data



- 1 Application calls `fread()`
- 2 `read()` starts a `SVC` (supervisor call)
- 3 The HW independent I/O read calls the driver
- 4 The driver starts the read on the HW and sets the calling process into waiting state.
- 5 The HW triggers an interrupt if the data is ready. The kernel first handles the interrupt in its system `ISR` (interrupt service routine)
- 6 The system `ISR` calls the `ISR` of the driver
- 7 The `ISR` of the driver starts a tasklet (its like a thread in the kernel but more lightweight)
- 8 The tasklet wakes up the driver
- 9 The driver read function continuous: fetches and copies the data into user space.
- 10 The I/O read function continuous and the `SVC` is now finished. If the process has the highest prio it can run on the CPU.
- 11 The data is processed in `fread()` and finally returned to the application.

Questions?

All right?



Question?



and use **chat**

or

speak *after* I
ask you to

Linux device files

- One **device file** per device
- **Device examples:** mouse, harddisk, keyboard, display, terminal, ...
- **Device file examples:** /dev/sda, /dev/tty0, /dev/cdrom
- Talk to device → `open()` / `close()` / `read()` / `write()` with device file
- Application uses the device file to communicate with a device
- Device files only have some meta data (inode) and does not require disk space.

Device types

Device type	File type	Example	Description
character device	c	Terminal, printers, ...	Sending and receiving single characters (bytes).
block device	b	Harddisks, USB, CD, ...	Sending and receiving of entire data blocks.

Create a Linux device file

Create a device file

```
1 mknod /dev/device_file TYPE MAJOR MINOR
2
3 #TYPE   = c (character) or b (block)
4 #MAJOR  = device type and the corresponding driver
5 #MINOR  = device within the driver (major_device_number)
```

Delete a device file

```
1 #delete device file
2 rm /dev/device_file
```

Inode device file example

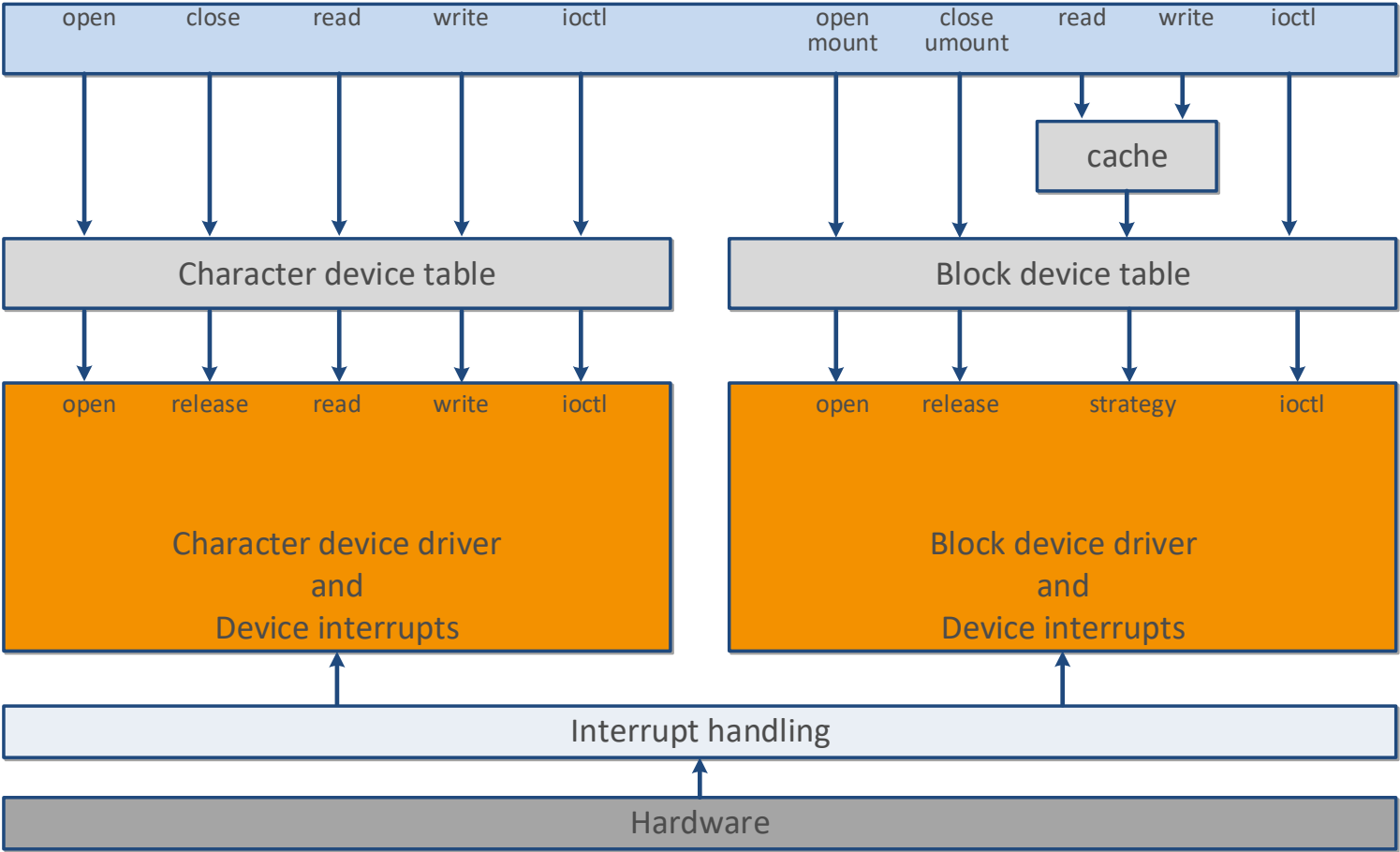
An inode on a device files contains the device type (c/b), the **major**, and the **minor** number.

```

1 ls -l -i /dev
2     type                                major  minor
3 328 brw-rw---- 1 root disk 259,    1    Jan 14 22:37 nvme0n1p1
4 329 brw-rw---- 1 root disk 259,    2    Jan 14 22:37 nvme0n1p2
5 330 brw-rw---- 1 root disk 259,    3    Jan 14 22:37 nvme0n1p3
6 ...
7  6 crw-rw-rw- 1 root root  1,     3    Dez 27 10:42 null
8 ...
9 15 crw--w---- 1 root tty   4,     0    Dez 27 10:42 tty0
10 20 crw--w---- 1 root tty   4,     1    Dez 27 10:42 tty1
11 ...

```

Integration of the device drivers



Linux device driver tables

Character device table*

MAJOR	open	release	read	write	ioctl	Example
1	ram_open	ram_release	ram_read	ram_write	ram_ioctl	Memory
3	ser_open	ser_release	ser_read	ser_write	ser_ioctl	Serial
4	tty_open	tty_release	tty_read	tty_write	tty_ioctl	Terminal
...						

Block device table*

MAJOR	open	release	ioctl	...	Example
1	hd_open	hd_release	hd_ioctl	hd_...	Harddisk
2	fd_open	fd_release	fd_ioctl	fd_...	Floppy disk
...					

* Simplified visualisation of device table with sample data.

List of MAJOR device numbers: <https://www.kernel.org/doc/html/latest/admin-guide/devices.html>

Questions?

All right?



Question?



and use **chat**

or

speak *after* I
ask you to

Kernel modules

A **kernel module** is a piece of compiled code that is dynamically loaded into the kernel.

- Filetype: *.ko (kernel object)
- Place: /lib/modules btw. /lib/modules/\$(uname -r)/
- Executed in the context of the kernel (kernel space).
- Can be loaded and unloaded at runtime (without reboot).



Kernel module: C template

```
1 #include <linux/module.h>    //needed by all modules
2 #include <linux/kernel.h>    //needed for KERN_INFO
3 #include <linux/init.h>      //needed for the macros
4
5 //this is called when the module is loaded
6 static int __init exmod_init(void) {
7     printk(KERN_INFO "Hello, exmod\n");
8     //register functions and initialise the module
9     return 0;
10 }
11
12 //this is called when the module is unloaded
13 static void __exit exmod_exit(void) {
14     printk(KERN_INFO "Goodbye, exmod\n");
15     //unregister functions and cleanup the module
16 }
17
18 module_init(exmod_init); //register the exmod_init to be called on load
19 module_exit(exmod_exit); //register the exmod_exit to be called on unload
20
21 //module meta data
22 MODULE_LICENSE("GPL");
23 MODULE_AUTHOR("Florian Künzner");
24 MODULE_DESCRIPTION("Exmod is an example module");
25 MODULE_VERSION("1.0");
```




Kernel modules: Linux kernel API

Inside the kernel you can **only** use the **Linux kernel API**. The `glibc` is not available there.

Linux kernel API examples:

Function	Description
<code>printk</code>	Print into the kernel log <code>/var/log/kern.log</code> .
<code>strcpy</code> , <code>strncpy</code> , ...	String functions from <code>include/linux/string.h</code>
<code>kmalloc/kfree</code>	Allocates and frees memory in the kernel space.

...

Linux kernel API https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_api.html#linux-kernel-api

Compile a kernel module

Makefile

```
1 obj-m += module_name.o
2
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

On shell:

```
dev@dev ~/module$ make
```

Kernel Makefile doc: <https://elixir.bootlin.com/linux/latest/source/Documentation/kbuild/makefiles.rst>



Kernel modules: Linux commands

Command

Description

`lsmod`

List the loaded modules (`/proc/modules`).

`modinfo module`

Show information about a module.

`insmod module.ko`

Load a `module.ko` file into the kernel.

`modprobe module`

Load a module from `/lib/module/$(uname -r)/` into the kernel and handle dependencies. Requires an up to date `modules.dep`.

`depmod`

Update `/lib/module/$(uname -r)/modules.dep`

`rmmod module`

Unload a module

`modprobe -r module`

Unload a module and handle dependencies.

Questions?

All right? \Rightarrow 

Question? \Rightarrow  and use **chat**

or

speak *after* I
ask you to

Device driver development

The developer has to implement

- a kernel module
- register itself at the I/O management in the OS
- handle application requests

Functions that have to be implemented

Functions to load the module into the kernel

- `module_init()` → `driver_init()`
- `module_exit()` → `driver_exit()`

Functions triggered through applications

- `open()` → `driver_open()/close()` → `driver_release()`
- `read()` → `driver_read()/write()` → `driver_write()`
- ...

Functions triggered through OS or hardware

- ISR (Interrupt Service Routine)
- Kernel threads
- Tasklet



Device driver: C template

```
6 #define MAJOR_NR 236
7 static int __init driver_init(void) {           //called on modprobe
8     register_chrdev(MAJOR_NR, "driver" , &fops);
9     //...
10    return 0;
11 }
12 static void __exit driver_exit(void) {          //called on rmmod
13     unregister_chrdev(MAJOR_NR ,NAME);
14 }
15 module_init(driver_init);
16 module_exit(driver_exit);
17
18 static struct file_operations fops = {
19     .open      = driver_open,
20     .release   = driver_release,
21     .read      = driver_read,
22     .write     = driver_write,
23 };
24 static int driver_open(...){                   //called on open()
25     bool ok = try_module_get(THIS_MODULE);    //increment driver usage
26     //...
27 }
28 static int driver_release(...){                //called on close()
29     module_put(THIS_MODULE);                  //decrement driver usage
30     //...
31 }
32 static ssize_t driver_read(...){               //called on read()
33     //...
34 }
35 static ssize_t driver_write(...){              //called on write()
36     //...
37 }
38 //module meta data...
```



Low level function usage

User space program:

```
1 #include <stdlib.h> //EXIT_SUCCESS
2 #include <fcntl.h> //flags O_RDWR, ...
3 #include <unistd.h> //open, close, read, write...
4 #include <stdio.h> //printf
5
6 #define LEN 10
7
8 int main(void){
9     int fd = open("/dev/device", O_RDWR, 0666);
10
11     char buffer[LEN] = {"Hello\n"};
12     ssize_t bytes_written = write(fd, buffer, LEN);
13     ssize_t bytes_read = read(fd, buffer, LEN);
14
15     close(fd);
16
17     return EXIT_SUCCESS;
18 }
```


Summary

Summary

- Design goals
- Structure of I/O systems
- Linux device files
- Kernel modules
- Linux device driver development

The end

Good luck in the exams.

We see us again in:

- “Rechnerarchitektur”
- Maybe in one of my FWPMs:
 - BSTG: Benutzerschnittstellen für technische Geräte (summer sem.)
 - SL: Scripting languages (winter sem.)
 - MP: Microcontroller Programming (winter sem.)