INTERRUPT

Et **interrupt** (afbrud) er et signal til CPU'en om, at **noget vigtigt er sket**, som kræver øjeblikkelig opmærksomhed.

🔧 Kort forklaret:

Et interrupt **afbryder** det program, CPU'en kører på det tidspunkt, for at håndtere en hændelse — f.eks. når:

- En tast bliver trykket ned på tastaturet
- En sensor sender data
- En netværkspakke ankommer
- En timer udløber

📌 Hvorfor bruges interrupts?

De gør systemet **hurtigere og mere effektivt**, fordi CPU'en ikke skal **vente aktivt** på, at noget sker (dette kaldes *polling*). Den reagerer kun, når det er nødvendigt.

🧠 Eksempel fra virkeligheden:

Forestil dig, at du sidder og arbejder, men har din telefon på lydløs.

- Hvis du **tjekker telefonen hvert minut** for beskeder = **polling** (ineffektivt).
- Hvis telefonen **vibrerer når der er besked** = **interrupt** (effektivt og responsivt).

🧩 I Linux/kernel-verdenen:

- En **interrupt handler** (`interrupt handler`) er en funktion, der kører automatisk, når en bestemt hændelse sker.
- Eksempel: Når en GPIO-pin ændrer tilstand, kan et interrupt aktivere en bestemt funktion i din driver.

PROC

`/proc` er en **virtuel filsystem-mappe** i Linux, som bruges til at vise information om systemet og kørende processer. Det er ikke en rigtig mappe på disken – den genereres **dynamisk i RAM** af kernel, når du tilgår den.

VISER KUN – LAGRER IKKE

Fil/Mappe	Forklaring
`/proc/cpuinfo`	Info om CPU'er (model, kerner, frekvens osv.)
`/proc/meminfo`	RAM-brug og hukommelsesstatus
`/proc/uptime`	Hvor længe systemet har været oppe
`/proc/loadavg`	System load over tid
`/proc/interrupts`	Aktive hardware interrupts og deres tællere
`/proc/[PID]/`	En mappe for hver aktiv proces (PID = Process ID)
`/proc/version`	Kernel-version og build-information

✅ Dit svar er helt korrekt:

> På et Linux-system kan man se hvilke **interrupts** der er i brug i filen:
> `/proc/interrupts`

📂 Hvad viser `/proc/interrupts`?

- En **liste over aktive hardware-interrupts**
- Antal gange hver interrupt er udløst
- Hvilken **CPU** der håndterer dem
- Hvilken **driver** eller enhed der er tilknyttet

Eksempelindhold:

yaml — Kopiér — Rediger

```
           CPU0       CPU1
  1:          12          0  timer
 23:        1445        256  gpio-irq
```

❌ Forkerte valgmuligheder:

Sti	Hvorfor forkert?
`/sys/class/interrupts`	Findes ikke – der er ingen sådan klasse
`/dev/`	Indeholder device nodes, ikke statusfiler
`/sys/firmware/devicetree/interrupts`	Device Tree-data, ikke runtime-info

🧪 Example Use Cases

- Monitor processes: `ps`, `top`, `htop` read from `/proc`
- Debug memory usage: `cat /proc/meminfo`
- Check CPU layout: `cat /proc/cpuinfo`
- Inspect network interfaces: `cat /proc/net/dev`
- Trace open file descriptors: `ls -l /proc/[PID]/fd`

SYS

`/sys` er et **virtuelt filsystem** i Linux (ligesom `/proc`), men det bruges primært til at **interagere med kernelens devices og drivere** – altså **hardwarekonfiguration og status**.

Det hedder også `sysfs`, og det giver dig adgang til struktureret information om f.eks. USB-enheder, diske strømstyring, kernel-moduler, og meget mere.

📂 Hvad findes i `/sys`?

Sti	Forklaring
`/sys/class/`	"Virtuelle" enhedsklasser – netværk, blokenheder, strøm, lyd osv.
`/sys/block/`	Information om block devices (harddiske, SSD'er, USB-drev)
`/sys/bus/`	Alle hardwarebusser som PCI, USB, I2C osv.
`/sys/devices/`	Rå hardwarestruktur – alle fysiske og virtuelle devices
`/sys/module/`	Kernelmoduler der er i brug
`/sys/firmware/`	Firmware-oplysninger (BIOS, ACPI, m.m.)
`/sys/power/`	Strømstyring – suspend, hibernate osv.
`/sys/kernel/`	Kernelinfo og visse kontrolpunkter (f.eks. kexec, debug)
`/sys/dev/`	Enheder organiseret efter deres major/minor device numbers

BOOT

🧠 Hvad er `/boot` i Linux?

`/boot` er en **systemmappe**, der indeholder de **filer der skal bruges ved opstart af systemet (boot)** – altså før operativsystemet kører helt.

Det er typisk et **egentligt partitioneret område** på disken (f.eks. `/dev/mmcblk0p1` på Raspberry Pi), og det er ofte i **FAT32-format**, så bootloadere og firmware kan læse det.

📂 Indhold i `/boot`

På en Raspberry Pi kan du typisk finde:

Fil/mappe	Beskrivelse
`config.txt`	📄 Konfigurationsfil for firmware og overlays
`cmdline.txt`	📄 Kernelparametre ved boot
`kernel.img` / `kernel7.img`	🧠 Linux-kernen (armv6/armv7/armv8)
`start.elf`, `bootcode.bin`	🚀 GPU-firmware og bootloader
`overlays/`	📂 `.dtbo`-filer (Device Tree overlays)
`initrd.img-*`	🧰 Initial ramdisk (bruges af kernel ved boot – ikke altid på RPi)

Question 4 (Mandatory) (5 points) ✓ *Saved*

I hvilken folder skal jeg lægge en device tree blob overlay fil (.dtbo) for at den kan indlæses automatisk under boot?

- ◯ /lib/modules
- ◯ /sys/firmware/devicetree/
- ◉ /boot/overlays/
- ◯ /etc/devicetreeoverlays/

Der står øverst at config.txt står for firmware og **overlays**

✅ Dit svar er 100 % korrekt!

> `/boot/overlays/` er den rigtige mappe til `.dtbo` (Device Tree Blob Overlay) filer på f.eks. Raspberry Pi.

📂 Hvad sker der i `/boot/overlays/`?

- Når du lægger en `.dtbo` her og tilføjer den til `config.txt`:

```txt
dtoverlay=myoverlay
```

... så vil den automatisk blive **indlæst ved boot** og ændre device tree'en med dine hardware-indstillinger.

❌ Forkerte svar – kort forklaring:

Sti	Hvorfor forkert?
`/lib/modules/`	Bruges til kernelmoduler, ikke overlays
`/sys/firmware/devicetree/`	Er **readonly runtime view**, ikke til input
`/etc/devicetreeoverlays/`	Ikke en standard-mappe i Linux

🧠 Hvorfor?

Når du bruger Raspberry Pi og vil indlæse et device tree overlay under boot (via `config.txt`), skal `.dtbo`-filer placeres i `/boot/overlays/`. Denne mappe ligger under `/boot`, som er den relevante mountede partition under boot.

Eksempel i `config.txt`:

txt Kopiér Rediger

```
dtoverlay=my_overlay
```

Hvis `my_overlay.dtbo` ligger i `/boot/overlays/`, vil den blive automatisk indlæst.

TL;DR:

- **Folder:** `/boot/overlays/`
- **Bruges af**: Raspberry Pi firmware ved opstart
- **Defineres i:** `/boot/config.txt`

❌ `<source_tree>/arch/arm/boot`

- **Forkert fordi:**
  Dette er en **build-mappe** i Linux kernelens **kildekode**. Den bruges under **kompilering** af kernel og device tree blobs (`.dtb`), men den har intet at gøre med **boot-processen på en færdig Raspberry Pi**.
- **Kort sagt:** Bruges kun under kernelbygning, ikke til runtime-boot.

❌ `<root_fs>/sys/firmware/devicetree/`

- **Forkert fordi:**
  Denne mappe indeholder **den aktuelt indlæste device tree** i **runtime**, som en **read-only visning** (via sysfs).
- Du **kan ikke lægge filer her**, og systemet læser ikke overlay-filer herfra ved boot.
- **Kort sagt:** Visning af allerede-indlæst tree – ikke en placering for overlay-filer.

❌ `<root_fs>/var/lib/`

- **Forkert fordi:**
  Denne mappe bruges typisk af **pakkehåndtering**, **tjenestestyring** (f.eks. systemd) og lignende til at gemme **state** og metadata – **intet med device trees at gøre**.
- **Kort sagt:** Ikke relateret til device tree eller boot-processen.

✅ `<root_fs>/boot/` – Hvorfor den er rigtig:

- **Fordi** Raspberry Pi's firmware (før Linux kører) **læser config.txt** og **loader overlay-filer fra** `/boot/overlays/`.

GIT KOMMANDOER

🔁 Hurtig opsummering af Git-kommandoerne:

Kommando	Hvad den gør
`git add`	Tilføjer ændrede filer til **staging area**
`git commit`	Gemmer ændringer i det **lokale repository**
`git push`	Sender commits til **remote repository** ✅
`git clone`	Henter hele projektet fra et remote repository (starter et nyt repo lokalt)

DRIVER OPRETTELSE

✅ Korrekt rækkefølge bør være:

1. `alloc_chrdev_region()`
   → Allokerer major/minor numre til din karakterenhed
2. `platform_driver_register()`
   → Registrerer platform-driveren, hvilket fører til et kald til `probe()`
   (herfra kaldes resten af logikken)
3. `of_gpio_count()` *(typisk i `probe()`)*
   → Aflæser GPIO-konfiguration fra device tree (`dtoverlay`)
4. `device_create()`
   → Opretter en `/dev/`-node til brug fra userspace

Question 5 (Mandatory) (5 points) ✓ *Saved*

Du har indlæst en gpio platform driver med insmod og det tilhørende overlay med dtoverlay. Hvad er rækkefølgen af funktioner kaldt i din driver?

4	device_create()
3	of_gpio_count()
1	alloc_chardev_region()
2	platform_driver_register()

INCLUDE

🧩 Hvad betyder **API**?

API står for **Application Programming Interface**.
Det er en **samling af funktioner, datastrukturer og regler**, som et program eller en del af et system stiller til rådighed for andre programmer, så de kan bruge det **uden at kende de indre detaljer**.

📎 Kort sagt:

- `include/` = **definerer API'et** i kernelens verden.
- Det er **ikke implementering**, men det, som moduler og drivere **inkluderer og bruger**.

📁 Struktur i `include/` (fra f.eks. `~/sources/rpi-5.4.83/include/`):

Mappe	Indhold
`include/linux/`	Kernelinterne headers (meget brugt af drivere)
`include/uapi/`	Headers der må bruges fra userspace
`include/asm/`	Arkitektur-specifikke headers
`include/net/`	Netværksrelaterede headers
`include/media/`	Video/lyd/media-subsystem
`include/drm/`	GPU/graphics (Direct Rendering Manager)
`include/trace/`	Tracepoint-definitioner til debug og performance

!!!FINDES ALTID UNDER SOURCE TREE

SE NÆSTE SIDE

✅ Dit svar er helt korrekt:

> Den primære kilde til `request_irq()` flag-navne er:
> `<source tree>/include/linux/interrupt.h`

📂 Hvorfor `interrupt.h`?

Denne header indeholder alle definitioner for interrupt flags såsom:

c — Kopiér — Rediger

```c
#define IRQF_SHARED     0x00000080
#define IRQF_TRIGGER_RISING  0x00000001
#define IRQF_TRIGGER_FALLING 0x00000002
#define IRQF_ONESHOT    0x00002000
```

Du kan finde disse i f.eks.:

bash — Kopiér — Rediger

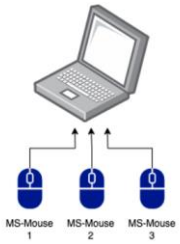
```bash
linux/include/linux/interrupt.h
```

❌ Forkerte svar:

Valg	Hvorfor forkert?
`/proc/interrupts`	Viser kun aktive interrupts – ikke flag-definitioner
`Google`	Ikke en kilde du kan stole teknisk 100 % på
`<device tree>/interrupts`	Device Tree definerer interrupt-numre, ikke `request_irq` flags

Major og Minor Numre

Givet følgende system:



Hvor mange major og minor numre skal MS-Mouse driveren bruge?

- ○ 3 major og 3 minor numre
- ○ 3 major og 1 minor numre
- ◉ 1 major og 3 minor numre
- ○ 1 major og 1 minor numre

📖 Forklaring:

I Linux bruges **device numre** (major og minor) til at identificere enheder:

- **Major-nummer**: Hvilken **driver** der håndterer enheden.
- **Minor-nummer**: Hvilken **individuel enhed** der håndteres af driveren.

🖱️ I eksemplet:

Du har **3 identiske mus (MS-Mouse)**, og de bruger **samme driver**.

- De skal derfor **dele samme major-nummer** (én driver).
- Hver mus skal have **sit eget minor-nummer** for at blive identificeret unikt.

🔢 Derfor:

- **1 major** (fælles driver)
- **3 minor** (én pr. mus)

- Major = 1 pr. driver
- Minor = 1 pr. enhed under den driver

Så hvis man havde 3 mus og 1 tastatur, så ville det være 2 major og 4 minor

DEV

`/dev` er en **systemmappe i Linux**, som indeholder **filer der repræsenterer enheder (devices)** – altså alt fra harddiske til mus, tastatur, lydkort og terminaler. Disse kaldes **device files**.

De er **interfaces til hardware** eller virtuelle enheder, så programmer kan bruge dem som almindelige filer – læse, skrive, åbne osv.

Filnavn	Forklaring
`/dev/sda`	Første harddisk (typisk hele disken)
`/dev/sda1`	Første partition på sda
`/dev/tty`	Aktiv terminal
`/dev/null`	Sort hul – alt du skriver her bliver droppet
`/dev/random`	Tilfældigt genererede data (bruges til kryptering m.m.)
`/dev/zero`	Uendelig strøm af nuller (`\0`)
`/dev/loop0`	Loop device (bruges til at montere image-filer som diske)
`/dev/usb/`	USB-enheder (organiseret under andre mapper også)
`/dev/input/`	Tastaturer, mus osv.
`/dev/video0`	Kamera-enhed (webcam)

🔠 Typer af device-filer

- **Block devices** – adgang i blokke (f.eks. diske): `/dev/sda` , `/dev/mmcblk0`
- **Character devices** – bytevis adgang (f.eks. tastatur, mus): `/dev/tty` , `/dev/input/mice`
- **Pseudo-devices** – specialfunktioner: `/dev/null` , `/dev/random` , `/dev/zero`

DEVICE TREE

✅ Kort opsummering:

Punkt	Forklaring
Beskriver	Hardware til Linux-kernen
Bruges på	ARM/embedded platforme
Fordel	OS og drivere kan være generiske og genbrugelige
Filtyper	`.dts`, `.dtb`, `.dtbo`

Vil du se hvordan man laver sit eget `.dts` og kompilere det med `dtc`, eller hvordan Raspberry Pi bruger overlays?

🌳 Hvad er **Device Tree**?

Device Tree (DT) er en **datafil**, der beskriver **hardwareenheder** i et system – især de enheder, som **operativsystemet (typisk Linux)** ikke selv kan finde ud af automatisk.

🧠 Hvorfor findes Device Tree?

På x86/PC'er bruges BIOS/UEFI til at fortælle styresystemet om hardware.
Men på mange **embedded platforme** (ARM-baserede) er der **ingen BIOS**.

👉 Derfor skal der bruges en **Device Tree** til at beskrive:

- Hvilke enheder findes?
- Hvor sidder de?
- Hvilken driver skal bruges?
- Hvilke GPIO'er, interrupts, clocks m.m. er forbundet?

✅ Hvorfor det er korrekt:

Felt	Betydning
`fragment@3`	Navn på fragment – nummeret er vilkårligt
`target = <&spi2>`	Viser, at vi tilføjer til SPI-bus 2
`myspi@1`	Enheden placeres på chip select 1
`reg = <1>;`	Matcher slave select 1 korrekt
`compatible = "myspi"`	Bruges til driver binding

KODE FOR ATTRIBUTTER

Question 13 (5 points) ✓ *Saved*

Hvilket kodeeksempel initialiserer attribute groups korrekt for følgende attributes?

```c
static ssize_t ui_status_store(struct device *dev, struct device_attribute *attr, sta

static ssize_t ui_status_show(struct device *dev, struct device_attribute *attr, char
```

```c
DEVICE_ATTR_RW(ui_status);
static struct attribute *ui_attrs[] = {
  &dev_attr_ui_status.attr, NULL, }
ATTRIBUTE_GROUPS(ui);
```

```c
DEVICE_ATTR_RW(ui_status);
static struct attribute *ui_attrs[] = {
  &dev_attr_ui_status_show.attr, &dev_attr_ui_status_store.attr, NULL, }
ATTRIBUTE_GROUPS(ui);
```

```c
DEVICE_ATTR_WO(ui_status);
static struct attribute *ui_attrs[] = {
  &dev_attr_ui_status_show.attr, &dev_attr_ui_status_store.attr, NULL, }
ATTRIBUTE_GROUPS(ui_status);
```

Her følges følgende opskrift:

Her skal man skrive det op på følgende måde:

…USING MACROS

```
/* Assuming Names:              */
/* group name = led             */
/* attribute name = led_state   */
/* RW/RO/WO = RW (led_state_store() + led_state_show()) */

DEVICE_ATTR_RW(led_state); // Creates dev_attr_led_state

static struct attribute *led_attrs[] = {
    &dev_attr_led_state.attr,
    // More attributes could go in here,
    NULL,
    };

TTRIBUTE_GROUPS(led); // Creates led_groups, expects led_attrs
```

lacros creates variables and inits structures

🧠 Hvorfor det er korrekt:

- `DEVICE_ATTR_WO(name)` er en **makro** til at definere en attribut med **kun en** `store()` **-funktion** (altså write-only).
- `&dev_attr_mcp3202_mode.attr` refererer til den attribut, makroen genererer.
- `ATTRIBUTE_GROUPS(mcp3202);` linker attributten korrekt til sysfs via attributgrupper.

❌ De forkerte valgmuligheder:

1. `DEVICE_ATTR_RO()` – giver kun read-adgang (og ingen `store()` -funktion).
2. **Bruger** `mcp3202_mode.attr` **direkte** – det kræver manuel oprettelse, ikke korrekt her uden ekstra kontekst.
3. **Forkert makroargument i** `DEVICE_ATTR_WO(dev_attr_mcp3202_mode)` – den forventer kun navnet (`mcp3202_mode`), ikke hele objektet.

SOURCE TREE

!!!HEADERS

🌳 Hvad er en **source tree**?

En **source tree** (kildekode-træ) er den **organiserede mappestruktur**, som indeholder **hele kildekoden** til et projekt – typisk et stort system som:

- Linux-kernen
- En embedded platform (f.eks. Buildroot)
- Et C/C++-program
- Et firmwareprojekt

📦 Struktur – hvad består en source tree af?

Her er et eksempel på, hvordan en **Linux kernel source tree** kan se ud:

bash Kopiér Rediger

```
linux/
├── arch/            # Arkitektur-specifik kode (f.eks. ARM, x86)
├── drivers/         # Alle hardwaredrivere
├── include/         # Headerfiler (.h), kernel-API'er
├── kernel/          # Kernelens core-funktioner (scheduler, mm, osv.)
├── fs/              # Filesystemer (ext4, tmpfs, vfat, ...)
├── init/            # Boot/init-funktioner
├── mm/              # Memory management
├── scripts/         # Værktøjer til at bygge
├── tools/           # Brugerværktøjer
├── Documentation/   # Info om subsystems, interfaces
└── Makefile         # Topniveau build-script
```

Spørgsmålet er:

"Hvad er din primære kilde til at finde information om hvad flagene til `request_irq` hedder?"

Det **korrekte svar** er:

✅ `<source tree>/include/linux/interrupt.h`

LIB/MODULES

TL;DR:
`/lib/modules` indeholder **drivere og kerne-moduler** for den aktuelle Linux-kerne. Her henter systemet moduler (f.eks. netværk, USB) som kan indlæses dynamisk med fx `modprobe`.

Du har valgt `/lib/modules`, og det er **helt korrekt** ✅

Spørgsmålet:

> *"...hvor skal man kopiere et kernemodul hen, hvis det skal kunne indlæses under boot?"*

Korrekt svar:

> `/lib/modules`

Hvorfor?

- Det er **standardplaceringen** for kerne-moduler (.ko-filer).
- Ved boot læser systemet herfra, især via `modprobe`, `initramfs`, og `depmod`.
- Hvis du lægger et modul her og opdaterer med `depmod`, kan det indlæses automatisk under boot.

Godt klaret!

KERNEMODULER OG .KO

KERNEL MODULES

MAKING KERNEL MODULES

It's plain-C, but with limitations!

Can only use the functionality present in the kernel:

- No access to the standard C / C++ libraries
- Header files is found in /rpi-5.xx.yy/include/ (or elixir.bootlin.com)
- No floating point operations

rrors in kernel modules can be fatal!

- There is no protection inside the kernel

imited Memory

- The kernel has only a small stack
- Do not allocate a lot of memory
- Use dynamic allocation for larger structures

TOP HALF OG BOTTOM HALF

When a hardware device (like a keyboard, network card, or disk) needs the CPU's attention, it triggers an **interrupt**. The Linux kernel responds in two stages:

🔷 Top Half

- This is the **immediate** response to an interrupt.
- It runs in **interrupt context**.
- Its job is to **quickly acknowledge** the interrupt and do **minimal, time-critical work** (e.g., copying data from a device buffer).
- Must be **fast and non-blocking**, because it disables other interrupts while running.

🔷 Bottom Half

- This part handles the **less time-sensitive** tasks.
- It is **deferred work**, scheduled to run **later**, outside the interrupt context.
- Can sleep or block, and does more extensive processing (e.g., updating system buffers, notifying user-space programs).

Question 12 (5 points) ✓ *Saved*

I forbindelse med interrupts, hvad må udføres i top-half? (Vælg 2)

Select 2 correct answer(s)

- [x] Opgaver som er tidskritiske
- [ ] Opgaver som kræver meget processering
- [ ] Opgaver som bruger floating-point operationer
- [x] Opgaver som ikke kan blokere
- [ ] Opgaver som afventer input fra andre enheder

VMALLOC OG KMALLOC

🧠 Forklaring:

Funktion	Beskrivelse
`vmalloc()`	Allokerer **virtuel hukommelse**, som **ikke behøver at være fysisk sammenhængende.** Velegnet til større allokeringer.
`kmalloc()`	Allokerer **fysisk sammenhængende** hukommelse. Hurtigere, men ikke egnet til store blokke.
`void *my_mem;`	Bare en pointer – ikke en allokeringsfunktion.
`new()`	En **C++-konstruktion**, ikke gyldig i kernel C-programmering.

TIMER FUNKTION

Spørgsmålet lyder:

> **Hvad må du ikke gøre i en timer funktion?**

🧠 Forstå konteksten

En **timer-funktion** i Linux kernel (typisk registreret med `timer_setup()` og kaldt af kernel efter en given tid) **kører i softirq-kontekst** (altså som en bottom half). Det betyder:

- Den må **ikke sove eller vente** (ingen blocking).
- Den må gerne tilgå simple data.
- Den må ikke kalde funktioner, der kan blokere (f.eks. vente på hardware eller mutex).

Gennemgang af svarmulighederne:

1. ✅ `gpio_get_value()`
   → OK i en timer. Det er en **non-blocking** funktion til at læse GPIO-status.
   ✔ **Må gerne bruges.**
2. ❌ `spi_sync()`
   → Denne funktion **kan sove**, fordi den sender og venter på svar over SPI.
   🚫 **Må ikke bruges i en timer!**
3. ✅ `int a = 0;`
   → Lokal variabel? Helt fint.
   ✔ **Må gerne bruges.**
4. ✅ `wake_up_interruptible()`
   → OK i en timerfunktion, bruges til at vække ventende tråde.
   ✔ **Må gerne bruges.**

✅ Du har valgt det **rigtige svar:**

> **Det er OK at kalde `add_timer()` i en timer-funktion.**

🧠 Hvorfor?

Timer-funktioner kører i **interrupt context (softirq)**, og derfor må du:

- ✅ **Udføre kort, ikke-blokerende kode**
- ❌ **Ikke kalde funktioner der kan sove/blokere**

🔍 Vurdering af mulighederne:

Funktion	Tilladt i timer-funktion?	Forklaring
`spi_sync()`	❌ Nej	Kan **blokere** – bruges i process context
`copy_to_user()`	❌ Nej	Kræver adgang til user-space memory → ikke tilladt i interrupt
`wait_event_interruptible()`	❌ Nej	**Blokerer** → ikke tilladt i interrupt eller timer
`add_timer()`	✅ Ja	Bruges til at **gensætte** en timer → non-blocking og tilladt

---

✅ TL;DR:

> I en timer-funktion må du **kun bruge ikke-blokerende kernel-funktioner**.
> `add_timer()` er safe – resten er **no-go** i interrupt context.

Vil du have et kodeeksempel på en periodisk timer med `add_timer()`?

Question 10 (Mandatory) (5 points) ✓ *Saved*

Hvad er ok at gøre i en timer funktion?

- ◯ Kalde spi_sync()
- ◯ Kalde copy_to_user()
- ◯ Kalde wait_event_interruptible()
- ◉ Kalde add_timer()

FOPS

› In kernel space, the fops operations (read/write), can put the calling application to sleep (and wake it up) using special functions.

DEV_ATTR

Rigtig god skabelon, følg denne uanset hvilken opgave du får:

…USING MACROS

```
/* Assuming Names:              */
/* group name = led            */
/* attribute name = led_state  */
/* RW/RO/WO = RW (led_state_store() + led_state_show()) */

DEVICE_ATTR_RW(led_state); // Creates dev_attr_led_state

static struct attribute *led_attrs[] = {
    &dev_attr_led_state.attr,
    // More attributes could go in here,
    NULL,
    };

ATTRIBUTE_GROUPS(led); // Creates led_groups, expects led_attrs
```

› Macros creates variables and inits structures
› Be careful about naming!!!

What Macro?!? See https://en.wikipedia.org/wiki/C_preprocessor

UDEV

> › Dynamic node creation from user space
> › Udev, Sysfs a.o.

✅ TL;DR: **udev**

- **udev** = Device manager for Linux kernel
- Kører i **brugerrum** (userspace)
- Reagerer på events fra kernel via **udevd**
- Skaber device nodes i `/dev`
- Håndterer **regler** (udev rules) i `/etc/udev/rules.d/`
- Typisk brugt til:
  - Navngivning af enheder
  - Automatisk opsætning
  - Tilknytning af attributter (fra `/sys`)

🧠 Hovedidé:

Kernel opretter enhed → `udev` reagerer → opretter device node i `/dev` + anvender regler

Vil du have et konkret udev rule-eksempel?

INODE

✅ TL;DR: **inode** (index node)

- En **inode** er en datastruktur i filsystemet, som indeholder metadata om en fil.
- **Indeholder ikke** filnavn eller indhold, men:
  - Filens **størrelse**
  - **Rettigheder** (permissions)
  - **Ejer** (user/group ID)
  - **Tidsstempler** (access, modify, change)
  - **Antal hard links**
  - **Pointere til data-blokke**

🧠 Nøgleidé:

> **Inode = alt om filen, undtagen navnet**

Navnet er gemt i mappestrukturen, som **peger på inode**.

SYSTEMCTL

✅ TL;DR: `systemctl`

- Kommando til at **styre systemd-tjenester** på Linux
- Bruges til at starte, stoppe, aktivere osv. system-services

🧠 Nøgleidé:

> `systemctl` = Din fjernbetjening til services i systemd-baserede systemer

SYSTEM V

✅ TL;DR: **System V init (SysVinit)**

- Ældre Linux **init-system** (før `systemd`)
- Starter tjenester i **sekventiel rækkefølge** via **shell-scripts**
- Scripts findes i:
  `/etc/init.d/` og `/etc/rc*.d/`

🧠 Nøgleidé:

> **System V init = rækkefølge-baseret script-styring af systemstart**
> Bruges stadig i ældre systemer – men er i praksis erstattet af `systemd`.

FUNKTION POINTER

*remove er en funktion pointer

```
int (*remove)(struct platform device *);
```

PAGE TABLES

PAGE TABLES

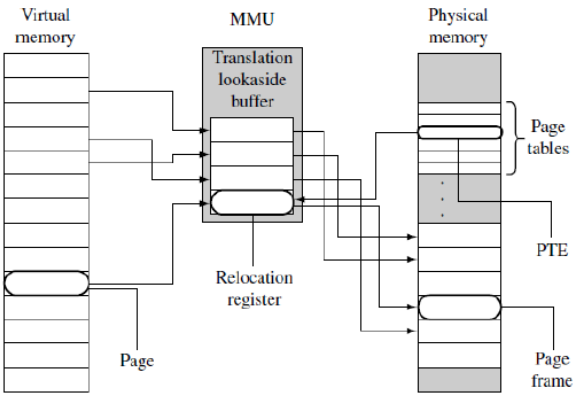
Page tables store information about:

- Physical address + size
- Access permissions
- Cache + Write buffer info

The Translation Lookaside buffer (TLB) is a cached version of the most recently used page tables

Page tables are managed by Linux. Each PTE is related to one proces

Multiple PTEs can point to the same physical address. Why?



🧠 Hvad er page tables?

Page tables bruges af operativsystemet og CPU'en til at **oversætte virtuelle adresser til fysiske adresser** i hukommelsen.

📦 Virtuel hukommelse vs fysisk hukommelse

- **Virtuel adresse**: Den adresse, som et program bruger.
- **Fysisk adresse**: Den faktiske placering i RAM.

Page tables laver mappet mellem disse to.

🧱 Grundbegreber

- **Page**: En fast størrelse blok af virtuel hukommelse (typisk 4 KB).
- **Frame**: En tilsvarende blok i fysisk hukommelse.

📘 Hvordan virker det?

1. Virtuel adresse opdeles i to dele:
    - **Page number** (bruges som indeks i page table)
    - **Offset** (forskydning inde i siden)
2. Page table slår page number op → giver frame number.
3. Fysisk adresse = frame number + offset.

🛡️ Beskyttelse og isolation

- Page tables gør det muligt at:
  - Beskytte hukommelse (f.eks. skrivebeskyttet)
  - Give hver proces sit eget virtuelle adresselayout

🏗️ Typer af page tables

- **Single-level:** Simpelt, men spilder plads.
- **Multi-level:** Spar plads, mere fleksibel (f.eks. 2- eller 4-niveau i x86-64).
- **Inverted page table:** Indexeret efter fysisk adresse (bruges i nogle systemer).

⚡ Hastighedsoptimering

- **TLB (Translation Lookaside Buffer):** Cache for page table opslag → meget hurtigere.

📜 Eksempel (for x86-64)

- 64-bit adresse → opdeles i:
  - 9 bits for hvert niveau (PGD, PUD, PMD, PTE)
  - 12-bit offset (giver 4KB pages)

🧠 TL;DR i én sætning:

Page tables oversætter virtuelle adresser til fysiske adresser, så programmer kan bruge deres egen hukommelse uden at bekymre sig om den fysiske RAM.

TLB

✅ TL;DR: **TLB (Translation Lookaside Buffer)**

- **TLB** = Cache for virtuelle til fysiske adresser
- Bruges af **MMU** (Memory Management Unit)
- Gemmer nyligt brugte **side-tabeller** (page table entries)
- ➕ **Hurtigere memory access**
- ➖ Ved ændringer i page tables skal TLB **invalideres** (TLB flush)

🧠 Nøgleidé:

> **TLB** = **hurtig oversættelse fra virtuel til fysisk adresse** – som en adresse-huskeliste for CPU'en

The Translation Lookaside buffer (TLB) is a cached version of the most recently used page tables

HEAP

✅ TL;DR: **Heap**

- **Heap** = Dynamisk hukommelsesområde i RAM
- Bruges til **runtime allokering** (f.eks. `malloc()` i C)
- Hukommelsen skal **frivilligt frigives** (f.eks. `free()`)
- Vokser **opad** i adresse-rummet (modsat stack)

🧠 Nøgleidé:

> **Heap** er der, hvor programmer gemmer **data med ukendt levetid**, f.eks. arrays, buffers, structs.

SLAB ALLOCATOR

✅ TL;DR: **Slab Allocator (Linux kernel)**

- **Slab allocator** = effektiv hukommelsesmanager i kernel space
- Bruges til at **allokere objekter af fast størrelse**
- Reducerer fragmentering og forbedrer performance
- Objekter genbruges → hurtig allokering og frigivelse

🧠 Nøgleidé:

> Kernel bruger slab allocator til at håndtere strukturer som `task_struct`, `inode`, osv.

VMA

✅ TL;DR: **VMA (Virtual Memory Area)**

- **VMA** = Et område i en proces' virtuelle adresse-rum
- Repræsenteres i kernel som en `struct vm_area_struct`
- Hver VMA har start/slut-adresse + tilladelser (r/w/x) og formål (fx stack, heap, mmap osv.)
- Bruges til at styre hukommelseskortlægning pr. proces

🧠 Nøgleidé:

> **En VMA er en sammenhængende blok af virtuel hukommelse med ens egenskaber**

MAKROER

Numlock led'en har en tilhørende attribute, max_brightness, som kan findes her:

```
stud@stud-virtual-machine:~$ ls -l
/sys/class/leds/input1\:\:numlock/max_brightness
-r--r--r-- 1 root root 4096 Dec 10 16:25
/sys/class/leds/input1::numlock/max_brightness
```

-r--r--r-- 1 root root betyder read only
Der er ingen skriveadgang (w mangle

Ud fra ovenstående, med hvilken makro må attributen være defineret?

- ✔ ◉ DEVICE_ATTR_RO Opretter en read-only attribut
- ○ DEVICE_ATTR_WO Opretter en write-only attribut
- ○ DEVICE_ATTR_WR Læse/skrive (write+read) adgang
- ○ DEVICE_ATTR_RE Findes ikke – ugyldig mulighed

📂 Sysfs tilladelser vs. makro

Makro	Funktioner i driver	Sysfs-permission (`ls -l`)	Forklaring
`DEVICE_ATTR_RO`	Kun `show()`	`-r--r--r--`	**Alle kan læse**, ingen kan skrive
`DEVICE_ATTR_WO`	Kun `store()`	`--w--w--w-`	**Alle kan skrive**, ingen kan læse
`DEVICE_ATTR_RW`	Både `show()` og `store()`	`-rw-r--r--`	**Ejer kan læse/skrive**, andre kan læse
`DEVICE_ATTR(name, 0600, ...)`	Custom	`-rw-------`	Kun root/ejeren kan læse/skrive

🔢 Numeriske tilladelser (`mode`) i Linux

Tilladelser skrives som tre cifre (fx `0644`), hvor hvert ciffer repræsenterer rettigheder for:

1. **Ejer**
2. **Gruppe**
3. **Andre**

Hvert ciffer er summen af:

Tilladelse	Værdi	Symbol
Læs	4	`r`
Skriv	2	`w`
Udfør	1	`x`

✅ Typiske kombinationer

Mode	Symbolsk	Forklaring
`0777`	`rwxrwxrwx`	Alle har fulde rettigheder
`0755`	`rwxr-xr-x`	Ejer: alt; gruppe/andre: læse+udfør
`0744`	`rwxr--r--`	Ejer: alt; andre: kun læse
`0666`	`rw-rw-rw-`	Alle har læse- og skriveadgang
`0644`	`rw-r--r--`	Ejer: læse+skrive; andre: læse
`0600`	`rw-------`	Kun ejer har adgang
`0400`	`r--------`	Kun ejer må læse

🛠️ Brug i `device attribute`

DOCUMENTATION

Question 15 (Mandatory) (5 points)

Linux source tree indeholder ud over kildekoden til Linux også dokumentation.

I en device tree fil kan jeg angive en gpio port's retning mm. ved at sætte værdien af "flag". Værdien og den tilhørende betydning kan være forskellig fra fabrikat til fabrikat.

I hvilken folder, under Linux source tree, finder jeg information om hvad et flags værdi betyder (eks. '1' = output), for en given platform? (source tree på Golden Image: ~/sources/rpi-4.14/)

- ○ ./Documentation/devicetree/readme Generel info om Device Tree, ikke flags
- ○ ./Documentation/devicetree/gpio Handler om GPIO API og subsystem – ikke platform-DT flags
- ✔ ◉ ./Documentation/devicetree/bindings Det er mappen har detaljer om flags, egenskaber, datatyper og betydningen af værdier i Device Tree
- ○ ./Documentation/devicetree/flags Findes typisk ikke – trick-svar

DEVICE DRIVERS AND APPLICATION

Det korrekte svar er:

✅ **"En device driver er en udvidelse af funktionaliteten i operativ systemet, mens en applikation er et brugerprogram som kan anvende funktionaliteten til f.eks. at overføre data mellem bruger og eksterne enheder"**

🧠 Forklaring:

- **Device drivers**:
  - Kører i **kernel space**.
  - Har **fuld adgang til hardware og systemressourcer**.
  - Er en **del af operativsystemets funktionalitet** – de udvider det med evnen til at snakke med specifik hardware.
- **Applikationer**:
  - Kører i **user space**.
  - Har **begrænset adgang** til systemressourcer.
  - Bruger systemkald til at kommunikere med OS (og dermed indirekte med hardware via drivere).

❌ Hvorfor de andre svar er forkerte:

- **"En applikation og en device driver kode har de samme rettigheder..."**
  → Forkert: Device drivers har **kernel-adgang**, applikationer har **brugeradgang**.
- **"Applikationskode afvikles ikke direkte på CPU'en..."**
  → Forkert: **Alt kode**, inkl. applikationer, køres af CPU'en – forskellen er privilegieniveauet (user vs. kernel).
- **"En device driver kører ikke i virtuel hukommelse..."**
  → Forkert: Både drivers og applikationer bruger **virtuel hukommelse** – men på forskelligt privilegieniveau.

🧠 User Space (brugerrum)

- Hvor **applikationer** kører (f.eks. webbrowser, teksteditor).
- Begrænset adgang til systemressourcer (ingen direkte adgang til hardware).
- Skal bruge **systemkald** for at tale med kernel/hardware.
- Garanterer stabilitet: hvis noget går galt, crasher kun applikationen.

🛠️ Kernel Space (kernerum)

- Hvor **operativsystemets kerne og device drivers** kører.
- Har **fuld adgang til hardware og systemhukommelse**.
- Koden kører med højere privilegier og kan påvirke hele systemet.
- Fejl her kan medføre **systemnedbrud** (kernel panic).

🔁 Kommunikation:

Applikationer i user space bruger **systemkald** (fx `read()`, `write()`, `ioctl()`) til at interagere med kernel space.

Her får du en enkel **oversigtstabel** over forskellen mellem **User Space** og **Kernel Space**:

Egenskab	User Space	Kernel Space
Hvem kører her?	Applikationer (f.eks. Firefox, Python)	Kernel, device drivers, systemkald
Adgangsniveau	Begrænset adgang	Fuld adgang til hardware og systemressourcer
Stabilitet	Fejl påvirker kun én applikation	Fejl kan crashe hele systemet (kernel panic)
Hukommelse	Virtuel hukommelse, isoleret fra kernel	Deler fysisk hukommelse og styrer alt hukommelse
Eksempler på kode	C/C++ programmer, shell scripts, GUI apps	Linux kernel, USB-driver, filsystemdrivere
Kald til funktionalitet	Via **systemkald** (fx `read()`, `write()`)	Kalder direkte funktioner og styrer hardware
Eksempel	`ls /dev` viser enhedsmapper	Driveren håndterer `/dev/ttyUSB0` bag kulissen

🧠 Metafor:

Tænk på **User Space** som passagerer i et tog, og **Kernel Space** som lokomotivføreren. Passagererne kan bede om at stoppe (systemkald), men det er føreren (kernel) der styrer toget (hardware).

Vil du have den som grafisk figur eller slide?

KERNEL SPACE

🧠 TL;DR: **Kernel Space**

- **Kører operativsystemets kerne og drivere**
- Har **fuld adgang** til hardware og systemressourcer
- **Kritisk område** – fejl kan crashe hele systemet
- Håndterer **systemkald** fra user space
- Taler direkte med CPU, hukommelse og enheder

➡️ Bruges kun af **OS og drivere**, **ikke almindelige apps**

USER SPACE

🧠 TL;DR: **User Space**

- Hvor **alle brugerprogrammer** kører (f.eks. browser, terminal, apps)
- Har **begrænset adgang** til systemet og hardware
- Kommunikerer med kernel via **systemkald**
- **Sikker** – fejl påvirker kun den enkelte app, ikke hele systemet
- Kan **ikke** direkte tale med hardware (skal gå gennem kernel)

➡️ Bruges af **brugere og apps**, ikke af operativsystemets kerne.

> Jeg vil foretage bestemmelsen af korrektionsfaktoren i **user-space applikationen**, fordi det blot er en beregning baseret på data fra sensoren. Man skal kun bruge kernel-space til hardwareadgang, mens databehandling er mere fleksibelt og sikkert i user-space.

✅ Bedste svar: User-space applikation

> **Hvor?**
> Kalibrering og beregning af korrektionsfaktoren bør foretages i **user-space applikationen**.

> **Hvorfor?**

- Det er en **beregning**, ikke direkte hardware-adgang.
- Det kræver ikke realtid.
- Brugere og udviklere kan lettere justere og teste korrektioner.
- Det holder **driveren simpel og generel** (driveren bør kun hente rå data).

> ✨ Fordele:

- Lettere at debugge
- Ingen behov for at genkompilere kernel-driver ved ændringer
- Mindre risiko for at lave fejl i kritisk kernel-kode

❌ Kernel-space: Når er det relevant?

> Du bør kun lægge det i kernel-space **hvis**:

- Det er **tidskritisk**
- Det kræver **sikker adgang til hardware**
- Eller det er del af en lukket pipeline, f.eks. i ISR eller DMA bufferhåndtering

Det er ikke tilfældet her – du læser 4 værdier via I2C og beregner en faktor → ingen tidspres = **brug user-space**.

GPIO

```
fragment@0 {
        target-path = "/";
          __overlay__ {
            mygpio: mygpio@0 {
              compatible = "mygpio";
              gpios = <&gpio 57 0>;
              };
          };
      };
```

Denne her henviser til gpio 57, hvor den sætter den som indgang, idet den er 0.

1 for udgang, og 0 for indgang.

Der skal være & ellers virker det ikke

Herudover så kan gpio'ers funktioner vælges ved de her forskellige. 4 tager fra 40 til 49. 1 tager fra1-19 osv.

Question 10 (Mandatory) (5 points) ✓ *Saved*

Med hvilket register i BCM2835 kan man vælge funktionen af GPIO 49?

- (●) GPFSEL4
- ( ) GPFSEL1
- ( ) GPFSEL2
- ( ) GPFSEL3

SPI_READ_BYTE

Selvfølgelig! Her er din oprindelige funktion – **opdateret til at sende kanalnummeret først**, som opgaven kræver. Jeg har tilføjet `//` kommentarer på **alle linjer**, så du nemt kan følge med:

c

Kopiér Rediger

```c
int my_spi_read(struct spi_device *spi, u8 channel, u8 *data) {
    struct spi_transfer t[2];         // To SPI-overførsler: én for at sende, én for at modtage
    struct spi_message m;             // SPI-beskedstruktur

    u8 tx_buf[1];                     // Buffer til at sende kanalnummer
    u8 rx_buf[1];                     // Buffer til at modtage svaret

    tx_buf[0] = channel;              // Læg kanalnummer i sendebufferen

    memset(t, 0, sizeof(t));          // Nulstil overførselsstrukturen
    spi_message_init(&m);             // Initialisér SPI-beskeden
    m.spi = spi;                      // Brug den SPI-enhed, vi fik som parameter

    t[0].tx_buf = tx_buf;             // Første overførsel: send kanalnummer
    t[0].len = 1;                     // Send 1 byte
    spi_message_add_tail(&t[0], &m); // Tilføj første overførsel til beskeden

    t[1].rx_buf = rx_buf;             // Anden overførsel: modtag ét byte svar
    t[1].len = 1;                     // Modtag 1 byte
    spi_message_add_tail(&t[1], &m); // Tilføj anden overførsel til beskeden

    int err = spi_sync(m.spi, &m);    // Udfør SPI-transaktionen (blokkerende kald)
    *data = rx_buf[0];                // Gem modtaget værdi i *data (via pointer)

    return err;                       // Returnér evt. fejlstatus (0 hvis OK)
}
```


```
int my_spi_read(struct spi_device *spi, u8 channel, u8 *data) {
    struct spi_transfer t[2];       // To SPI-overførsler: én for at sende, én for at modtage
    struct spi_message m;           // SPI-beskedstruktur

    u8 tx_buf[1];                   // Buffer til at sende kanalnummer
    u8 rx_buf[1];                   // Buffer til at modtage svaret

    tx_buf[0] = channel;            // Læg kanalnummer i sendebufferen

    memset(t, 0, sizeof(t));        // Nulstil overførselsstrukturen
    spi_message_init(&m);           // Initialisér SPI-beskeden
    m.spi = spi;                    // Brug den SPI-enhed, vi fik som parameter
```

```
    t[0].tx_buf = tx_buf;          // Første overførsel: send kanalnummer
    t[0].len = 1;                  // Send 1 byte
    spi_message_add_tail(&t[0], &m); // Tilføj første overførsel til beskeden

    t[1].rx_buf = rx_buf;          // Anden overførsel: modtag ét byte svar
    t[1].len = 1;                  // Modtag 1 byte
    spi_message_add_tail(&t[1], &m); // Tilføj anden overførsel til beskeden

    int err = spi_sync(m.spi, &m);   // Udfør SPI-transaktionen (blokkerende kald)
    *data = rx_buf[0];             // Gem modtaget værdi i *data (via pointer)

    return err;                    // Returnér evt. fejlstatus (0 hvis OK)
}
```

SPI_WRITE

```
int my_spi_write(struct spi_device *spi, u8 value) {
    struct spi_transfer t[1];       // Kun én SPI-overførsel er nødvendig
    struct spi_message m;           // SPI-beskedstruktur

    u8 tx_buf[1];                   // Buffer til at holde værdien der skal sendes
    tx_buf[0] = value;              // Læg data i buffer – det der skal sendes

    memset(t, 0, sizeof(t));        // Nulstil overførselsstrukturen
    spi_message_init(&m);           // Initialisér SPI-besked
    m.spi = spi;                    // Brug den SPI-enhed, vi har fået som parameter

    t[0].tx_buf = tx_buf;           // Angiv at vi vil sende tx_buf
    t[0].rx_buf = NULL;             // Vi læser ikke noget tilbage, kun write
    t[0].len = 1;                   // Send præcis 1 byte
    spi_message_add_tail(&t[0], &m); // Tilføj overførslen til SPI-beskeden

    int err = spi_sync(m.spi, &m);   // Udfør SPI-transaktionen (blokkerende kald)

    return err;                     // Returnér status (0 = OK, <0 = fejl)
}
```

DRIVERS

En **driver** (eller **enhedsdriver**) er et program, der fungerer som et **bindeled mellem operativsystemet og hardwareenheder**. Den sørger for, at operativsystemet og applikationer kan kommunikere med hardware – uden at de behøver at kende detaljerne om, hvordan den fungerer.

🔧 Hvad gør en driver?

Når du f.eks. tilslutter en mus, et tastatur, et netværkskort eller en printer:

- Operativsystemet bruger **driveren** til at sende kommandoer til enheden.
- Driveren oversætter disse kommandoer til noget, som den specifikke hardware forstår.
- Hvis enheden sender data tilbage, oversætter driveren det til noget, som operativsystemet kan bruge.

🧠 Eksempel:

Uden driver:

Program → "Gem fil på disk" → ❌ Forstår ikke, hvordan disken virker

Med driver:

Program → "Gem fil på disk" → **Driver** → sender korrekt besked til harddisken → ✅

🧩 Opsummering:

> En **driver** er et softwarelag, der gør det muligt for operativsystemet at kommunikere med hardware, uden at programmet selv behøver at kende hardwareens detaljer.

EKSEMPLER

Typer af drivere:

Type	Eksempel
Block driver	Harddiske, SSD'er, USB-drev
Character driver	Serielport, tastatur, mus
Network driver	Ethernet- og Wi-Fi-adaptere
Platform driver	Enheder indbygget på specifik hardware
SPI/I2C driver	Sensorer, ADC'er, små perifere enheder

Opsummering:

> En **platform driver** bruges til at håndtere hardware, der ikke kan detekteres automatisk, men som er **beskrevet manuelt** i systemet. Det er almindeligt i embedded Linux-systemer.

🧠 Eksempel:

Tænk på en **I/O-chip**, en **GPIO-controller**, eller en **sensor** der sidder direkte forbu til processoren – Linux ved ikke, at den findes, før vi fortæller det.

🔧 Komponenter i en platform driver:

1. **Platform device**
    - Repræsenterer hardware. Beskrives i Device Tree (`.dts` filer) eller i boarc filer.
2. **Platform driver**
    - Kode i kerne-modulet, der håndterer enheden: init, read, write, etc.

🧩 Korrekt match:

Device Type	Valgt	Enhed	Hvorfor?
Not really a device	4	**CPU**	CPU er ikke en enhed med enhedsnoder (fx i `/dev`)
Block Device	1	**Floppy Disk**	Lagerenhed der læser/skriver i blokke
Character Device	3	**Serial Port**	Leverer data byte for byte
Network Interface	2	**Wimax**	Trådløs netværksteknologi

🧠 TL;DR:

- **CPU** → kernel håndterer den direkte – ikke en "device"
- **Block** → typisk diske, USB'er, SSD'er
- **Character** → tastatur, seriel port, sensorer
- **Network** → interface som eth0, wlan0, wimax

Vil du have et kort skema over alle device-typer og eksempler til print?

DEVICES

1. Character Device

🕹️ Læser/skriver data **én byte ad gangen**

🧵 Eksempler: tastatur, seriel port, sensor

---

2. Block Device

📦 Læser/skriver data i **blokke (typisk 512B eller mere)**

💽 Eksempler: harddisk, SD-kort, SSD

---

3. Network Interface

🌐 Bruges til **dataoverførsel via netværk**

📡 Eksempler: Ethernet, Wi-Fi-adaptere

---

4. Composite Device

🔀 **Kombinerer flere device-typer** i én enhed

📱 Eksempel: USB-enhed med både mus, tastatur og l[illegible]er

🗂️ Oversigt: Enhedstyper og device-klasser

Enhed	Device Type	Beskrivelse
Accelerometer	🧵 *Character Device*	Læser små værdier direkte, én ad gangen
Memory Stick	💽 *Block Device*	Lager, læser/skriver i større blokke
WLAN Adapter	🌐 *Network Interface*	Netværksoverførsel, bruger sockets
USB Device	🔀 *Composite Device*	Kan have flere funktioner (lager, input osv.)

🧠 Hurtig huskeregel:

- **Character = små strømme af data**
- **Block = lagermedier**
- **Network = kommunikation**
- **Composite = alt-i-én-enhed**

FLOATING POINT OG IKKE TILLADT I LINUX:

✅ **Korrekt valgt:** `temp = get_temp() + 0.2;` **er ikke tilladt i en Linux device driver.**

🧠 Forklaring:

Linux kernel (og dermed device drivers) **må ikke bruge floating point (kommatal)** – som f.eks. `0.2`.

Hvorfor ikke?

- Kernelens **floating point-enhed (FPU)** er **deaktiveret**, for at beskytte systemets stabilitet og ydeevne.
- Brug af float kræver **gemme/genskabe FPU-status**, hvilket er **dyrt og farligt** i kernel-mode.
- Kerneldrivere **skal bruge heltal** (fx `int`, `fixed point`, `u32`, `s64`) for præcise og sikre beregninger.

🔍 Hvad med de andre muligheder?

Kode	Tilladt?	Forklaring
`kmalloc(sizeof(var));`	✅ Ja	Kernel-standard til hukommelse
`udelay(10);`	✅ Ja	Mikrosekunders aktiv ventetid
`printk(KERN_ERR "...")`	✅ Ja	Bruges til fejllogning i kernel

✅ TL;DR:

> ❌ **Floating point (`+ 0.2`) er ikke tilladt i kernel** – brug faste heltalsformater i stedet.

Vil du se et eksempel på fixed-point beregning i stedet for float?

HVOR FORSKELLIGE TING LIGGER

Emneområde	Valgt	Korrekt
rootfs	3	✔ *home folder*
Device tree	1	✔ *overlays*
sysfs	4	✔ *classes and devices*
Source tree	2	✔ *Linux include files*

🧠 **Tip til huskning:**

- `sysfs` = systemstruktur i `/sys` → enheder & klasser
- `device tree` = hardwarebeskrivelse → bruges med overlays
- `source tree` = kildekode inkl. headers
- `rootfs` = basefilstruktur → inkluderer bl.a. `/home`

Du nailed den. Skal jeg sende dig et samlet cheat sheet over disse begreber?

tldr også

🧠 **TL;DR – hurtig huskeregel**

- **rootfs → home folder**
  > Din normale rod-filstruktur med /home, /bin, /etc osv.
- **Device tree → overlays**
  > Hardwarebeskrivelse + små overlay-filer, der kan lægges ovenpå.
- **sysfs → classes & devices**
  > Kernel viser enheds-/klasse-info i /sys.
- **Source tree → Linux include files**
  > Selve kernelkilde + headers i `include/linux/…`.

CHECKSUM

En checksum, også kaldet en kontrolsum eller hash-sum, er en værdi beregnet fra en datablok, der bruges til at kontrollere data for ændringer eller fejl. Den virker som en digital fingeraftryk og kan bruges til at verificere, om dataene er blevet ændret under transmission eller lagring.

MEMORY MANAGEMENT UNIT

Question 14 (Mandatory) (5 points) ✓ *Saved*

Hvad er primære opgave for processorens Memory Management Unit?

- ◯ At styre Cache coherency
- ◉ At omsætte virtuelle adresser til fysiske
- ◯ At styre ekstene memory devices
- ◯ At styre memory endian

🧠 Hvad laver MMU (Memory Management Unit)?

> Den **vigtigste opgave** for MMU er at:
>
> **Omsætte virtuelle adresser til fysiske adresser**

📌 Hvorfor er det vigtigt?

- Gør det muligt for **hver proces at tro**, at den har sit eget, isolerede memory.
- Giver **beskyttelse** mellem processer og kernel.
- Understøtter **paging og virtual memory**.

❌ De forkerte svar – kort forklaret:

Valgmulighed	Hvorfor forkert?
Cache coherency	Styres af CPU og cache controller, ikke MMU
Eksterne memory devices	Styres via bus-controllers (I2C, SPI, mm.)
Memory endian	Bestemmes af CPU-arkitektur, ikke MMU

🔁 TL;DR:

> MMU = Virtuel → fysisk adresseoversættelse + hukommelsesbeskyttelse.

Copy_to_user og copy_from_user

✅ **Fælles for både `copy_from_user()` og `copy_to_user()`:**

- Returnerer **antal bytes, der IKKE blev kopieret**.
- Hvis returneringsværdien er **0**, betyder det at **hele kopieringen lykkedes**.
- Hvis returneringsværdien er **≠ 0**, skete der en **fejl** under kopieringen.