

METODER OG FUNKTIONER

Funktioner:

Mdelay()

`mdelay()` — TL;DR (på dansk)

- 🕒 Forsinker udførelsen i millisekunder
- 🔄 Busy-wait (CPU'en er aktiv hele tiden)
- ✅ Mere præcis end `msleep()`
- ❌ Bruger flere CPU-ressourcer
- ❌ Ikke strømbesparende
- ⚠️ Brug kun ved behov for høj præcision, fx tidlig boot eller hardware-nær kode

vs Brug `msleep()` når:

- Du kan leve med lidt unøjagtighed
- Du vil lade CPU'en arbejde videre med andre opgaver

MSleep()

`msleep()` — TL;DR (på dansk)

- 🛌 Sover i millisekunder
- 🧠 Frigiver CPU'en → andre processer kan køre imens
- ❌ Ikke super præcis (kan variere lidt afhængigt af systemets "jiffy"-tid)
- ✅ God til energibesparelse og multitasking
- 🔄 Afhænger af scheduler


vs Brug `mdelay()` kun hvis du har brug for høj præcision og kan acceptere høj CPU-brug

Alloc_chrdev_region()


Forklaring:

- `alloc_chrdev_region()`
 - Denne funktion **allokerer et major- og minor-nummer** i kernelens device-tabeller.
 - Det er **første skridt** for at kunne knytte en character device til kernelens device-node system.
 - Den returnerer et `dev_t`, som bruges videre i `cdev_add()` og `device_create()`.

Device_create()

-  `device_create()`
 - Opretter en device-fil under `/dev`, men kræver at du allerede har major/minor via `alloc_chrdev_region()`.

Cdev_add()

-  `cdev_add()`
 - Registrerer en character device i kernelens cdev-system, men **bruger dev_t** som allerede er allokeret.

Class_create()

-  `class_create()`
 - Bruger man til at oprette en klasse i `/sys/class`, typisk før `device_create()`.

TOP

`top` er en kommandolinje-baseret overvågningskommando i Linux, der viser realtidsinformation om systemets ressourcer og processer.

📌 Grundlæggende om `top` -kommandoen:

```
bash
```

[Kopier](#) [Rediger](#)

```
top
```

Dette viser en dynamisk og løbende opdateret oversigt over:

- CPU-brug
- Hukommelsesforbrug
- Kørende processer
- Systemets belastning (load average)
- Brugere og systemtid

DMESG

Kommandoen `dmesg` i Linux viser **kernelmeddelelser** – altså beskeder fra Linux-kernen om systemets hardware og drivere.

📌 Hvad gør `dmesg` ?

```
bash
```

[Kopier](#) [Rediger](#)

```
dmesg
```

Den viser output fra **ring buffer** (en midlertidig log), som kernen skriver til under systemets opstart og ved hardwarebegivenheder.

VMALLOC OG KMALLOC'

✅ TL;DR: `kmalloc()` vs `vmalloc()` (Linux kernel)

| Egenskab | <code>kmalloc()</code> | <code>vmalloc()</code> |
|------------------|------------------------------|---|
| Hukommelsestype | Fysisk sammenhængende RAM | Virtuelt sammenhængende, ikke fysisk |
| Hastighed | ✅ Hurtig | 🐢 Langsommere |
| Cache | ✅ Kan caches direkte | ❌ Ikke cachevenlig |
| Brug i hardware? | ✅ Ja (DMA, mmio, buffer) | ❌ Nej – ikke egnet til DMA |
| Typisk brug | Små/mellemstore allokeringer | Store allokeringer (f.eks. flere sider) |

UDELAY()

✅ TL;DR: `udelay()`

- `udelay(n)` laver en aktiv forsinkelse i n mikrosekunder (μ s)
- Bruges i Linux kernel code (ikke i user-space!)
- Eksempel:

```
c
udelay(100); // vent 100  $\mu$ s
```

Kopier Rediger

Bruges typisk til:

- Tidskritisk hardware-initialisering
 - Vent på kort hardware-respons (f.eks. GPIO, SPI)
-

Bemærk:

- Blokerer CPU'en aktivt → ineffektiv ved længere ventetid
 - Til længere delays: brug `mdelay()` (millisekunder) eller **sleep-funktioner**
-

Nøgleidé:

`udelay(n)` = vent præcist $n \mu\text{s}$, men spild CPU-tid mens du gør det.

Konklusion:

- `udelay()` blokerer CPU aktivt, ja
 - Men den blokerer ikke i kernel-scheduling-forstand, og kan derfor bruges i interrupts
-

Question 2 (5 points) ✓ *Saved*

I hvilke situationer vil man vælge at bruge udelay frem for usleep? (Vælg 2)

Select 2 correct answer(s)

- ☐ Når der er behov for at frigive CPU ressourcer i systemet
- ☒ Når præcision er vigtigt
- ☐ Når præcision ikke er afgørende
- ☐ Når forsinkelsen er større end 5 ms
- ☒ Når man ikke må blokere, f.eks. i et interrupt

Her er der flere lignende:

Herunder:

`Udelay()`

`Mdelay()`

`Usleep_range()`

`Msleep()`

`Schedule_timeout()`

`Msleep_interruptible()`

Delay- og sleep-funktioner i kernel space

| Funktion | Enhed | Type | Må bruges i interrupt? | Præcision | Kommentar |
|-------------------------------------|---------------|-----------------------|------------------------|-----------------------|--|
| <code>udelay(n)</code> | Mikrosekunder | Busy-wait | ✅ Ja | ✅ Høj | Aktiv ventetid (CPU spildes) |
| <code>mdelay(n)</code> | Millisekunder | Busy-wait | ✅ Ja | ⚠️ OK, men ineffektiv | Bruges sjældent – loopet <code>udelay</code> |
| <code>usleep_range(min, max)</code> | Mikrosekunder | Sleep (kontekstskift) | ❌ Nej | ✅ Nogenlunde | Bedste præcise sleep i ikke-interrupt |
| <code>msleep(n)</code> | Millisekunder | Sleep | ❌ Nej | ❌ Lav | Simpel sleep, lav præcision |
| <code>schedule_timeout()</code> | Jiffies | Sleep + Timeout | ❌ Nej | ⚠️ Afhænger af HZ | Bruges til vent med timeout i tasks |
| <code>msleep_interruptible()</code> | Millisekunder | Sleep (kan afbrydes) | ❌ Nej | ❌ Lav | Kan afbrydes via signaler |

Vigtige tommelfingerregler:

- Interrupt context = `udelay()`
- <10ms delay og ikke i interrupt = `usleep_range()`
- >10ms delay = `msleep()` eller `schedule_timeout()`

OPEN()

`open()` – TL;DR

- Kaldes, når en bruger **åbner enheden**, f.eks. via `open("/dev/mit_device", ...)`.
- Er en del af `file_operations` structen:

c

 Kopiér  Rediger

```
struct file_operations fops = {  
    .open = my_driver_open,  
    ...  
};
```

Formål

- Bruges til at **initialisere enhedstilstand**, f.eks.:
 - Allokere ressourcer (f.eks. buffer eller IRQ)
 - Tjekke hardwarestatus
 - Forberede enheden til brug

Bemærk

- Den **må ikke sove** hvis kaldt fra kernelkontekst (sjældent tilfældet i `open`).
- Hvis du **requester en interrupt her**, skal du huske at **frigive den i** `release()`.

Brug `open()` til:

- Starte hardware
- Låse ressourcer
- Tilknytte `private_data` til `filp->private_data`

RELEASE()



`release()` – TL;DR

- Kaldes, når en bruger **lukker enheden**, f.eks. via `close(fd);`
- Modsvarer `open()` – det er her, du **rydder op** og frigiver ressourcer.



Typisk brug

- Frigive:
 - IRQ'er (hvis du brugte `request_irq()` i `open()`)
 - Dynamisk allokeret hukommelse
 - Hardware-ressourcer
- Lukke kommunikation med hardware, hvis nødvendigt



Brug `release()` til:

| Formål | Eksempel |
|-------------------------|---|
| Frigive IRQ | <code>free_irq()</code> |
| Rydde buffer/hukommelse | <code>kfree(filp->private_data)</code> |
| Lukke forbindelse | Afslutte samtale med hardware |

PROBE()

`probe()` – TL;DR

- `probe()` kaldes automatisk, når en **driver matches med en hardware-enhed** (typisk via device tree eller platform bus).
- Det er stedet, hvor du **initialiserer enheden**, f.eks.:
 - Allokere ressourcer (I/O memory, IRQs, buffers)
 - Opretter device filer (via `device_create()`)
 - Registrerer enhedens tilstand

Hvorfor bruge `probe()` ?

Fordele

IRQ request

Enhedsinitialisering

Mapping af I/O ressourcer

Du bør undgå



`open()` i platform-drivers

Forsinket init

Brug af magic numbers

INIT()

`init()` – TL;DR

- `init()` er ikke en del af `file_operations`, men er kernel-modulets **initialiseringsfunktion**.
- Det er den funktion, som kaldes **én gang**, når modulet loades med `insmod`.

Typisk brug:

- Registrere driveren i systemet
- Oprette device class
- Allocere major/minor nummer
- Registrere `file_operations` (f.eks. via `cdev_add`)

TL;DR konklusion:

`init()` gør dette:

Registrerer driver (cdev, class, device)

Allokerer systemressourcer (major/minor)

Kaldes én gang ved modulindlæsning

EXIT()



`exit()` – TL;DR

- `exit()` er **afslutningsfunktionen** for et kernel modul.
- Kaldes **automatisk**, når du fjerner modulet med `rmmod`.
- Bruges til at **rydde op efter** `init()`, så der ikke lækker ressourcer



Typisk oprydning i `exit()`:

- Fjern `/dev/`-filen: `device_destroy(...)`
- Slet klassen: `class_destroy(...)`
- Fjern device fra kernel: `cdev_del(...)`
- Frigiv device number: `unregister_chrdev_region(...)`

⚠️ TILBÆGGE:

Alt du allokerer i `init()`, skal frigives i `exit()` – ellers lækker du ressourcer (filer, klasser, IRQs osv.).



TL;DR Konklusion

`exit()` gør dette:

Kaldes ved `rmmod`

Rydder op efter `init()`

Frigiver enhedsnumre, sletter device/class/IRQ

READ()

`read()` – TL;DR

- `read()` kaldes, når brugeren forsøger at **læse fra en device**, f.eks.:

TL;DR Konklusion

`read()` bruges til...

Returnere data fra driver til bruger

Kontrollere hvor meget data der gives

Sikre korrekt kopiering via `copy_to_user`

WRITE()



`write()` – TL;DR

- `write()` kaldes, når en bruger skriver til din device, f.eks.:



TL;DR Konklusion

`write()` bruges til...

Modtage data fra bruger til kernel-driveren

Gemme, tolke eller videresende det til hardware

Logge eller validere brugerinput i kernel space

ENHEDSREGISTRERING

Alloc_chrdev_region

Register_chrdev

Cdev_init

Cdev_add

Cdev_del

Unregister_chrdev_region



Enhedsregistrering

| Funktion | Beskrivelse |
|---|--|
| <code>alloc_chrdev_region()</code> | Allokerer major/minor nummer |
| <code>register_chrdev()</code> | Simpel device-registrering (ældres metode) |
| <code>cdev_init()</code> | Initialiserer <code>cdev</code> struct |
| <code>cdev_add()</code> | Registrerer character device |
| <code>cdev_del()</code> | Afregistrerer device igen |
| <code>unregister_chrdev_region()</code> | Frigiver major/minor nummer |

PLATFORM/DEVICE-DRIVER INTEGRATION

Of_match_table

Platform_get_resource

Devm_*



Platform/device-driver integration

| Funktion | Beskrivelse |
|--------------------------------------|----------------------------------|
| <code>probe()</code> | Kaldes når enheden matches |
| <code>remove()</code> | Kaldes når enheden fjernes |
| <code>of_match_table</code> | Match af device tree noder |
| <code>platform_get_resource()</code> | Få adgang til I/O eller IRQ |
| <code>devm_*</code> -funktioner | Automatisk hukommelseshåndtering |

DIVERSE NYTTIGE FUNKTIONER:

Copy_to_user

Copy_from_user

Kzalloc og kcalloc

Kfree

Printk

Wake_interruptible

Wait_event_interruptible

Diverse nyttige funktioner

| Funktion | Beskrivelse |
|---|--|
| <code>copy_to_user()</code> | Kopier data til user-space |
| <code>copy_from_user()</code> | Kopier data fra user-space |
| <code>kzalloc()</code> / <code>kcalloc()</code> | Allokerer kernel-hukommelse |
| <code>kfree()</code> | Frigør hukommelse |
| <code>printk()</code> | Logger til kernel log (<code>dmesg</code>) |
| <code>wake_up_interruptible()</code> | Vækker tråd i ventetilstand |
| <code>wait_event_interruptible()</code> | Blokering med wakeup |

MAJOR OG MINER FUNKTIONER

| Funktion | Formål |
|----------------------|---|
| <code>MKDEV()</code> | Kombinér major + minor til <code>dev_t</code> |
| <code>MAJOR()</code> | Udpak major fra <code>dev_t</code> |
| <code>MINOR()</code> | Udpak minor fra <code>dev_t</code> |

F_pos

F_inode

F_path

F__owner

◆ f_pos

→ Filens læse/skriveposition (offset).

Bruges i f.eks. `read()` og `write()` til at holde styr på hvor i filen man er.

◆ f_inode

→ Pointer til inode-strukturen, som beskriver selve filen på disk.

Bruges til at få adgang til metadata som filtype, størrelse, device m.m.

◆ f_path

→ Indeholder sti og mount-info for filen.

Giver adgang til filens placering i filsystemet (`dentry` , `mnt`).

◆ f_owner

→ Bruges til asynkron I/O og signalering (f.eks. `SIGIO`).

Indeholder info om hvilken proces der ejer filen ift. signalhåndtering.



REGISTER_CHRDEV

TL;DR:

`register_chrdev()` is a Linux kernel function used to register a simple character device driver.

- **Inputs:**
 - `major` : major number (use `0` to auto-assign)
 - `name` : device name
 - `fops` : file operations (e.g., `read`, `write`)
- **Returns:** major number or error code
- **Used for:** creating basic character devices
- **Note:** It's outdated — prefer `alloc_chrdev_region()`, `cdev_init()`, and `cdev_add()` for modern drivers.