

## Contents

Kapitel 1: Introduktion til Linux og Hardware Abstraktion .....	4
1.1 Generelle koncepter .....	4
1.2 Device Nodes og Udev .....	4
1.3 Grundlæggende Linux kommandoer .....	4
Kapitel 2: Kernel Interface og Kernel Modules.....	5
2.1 Kernel Modules – Grundlæggende .....	5
2.2 Registrering af Character Devices.....	5
2.3 Ansvarsområder for vigtige filoperationer.....	5
2.4 Eksempel på simpel driver med kommentarer .....	6
Kapitel 3: Interrupts og Deferred Work (Bottom Half) .....	9
3.1 Grundlæggende om Interrupts .....	9
3.2 Samspil mellem User Space og Interrupts.....	9
3.3 Kodeeksempel med kommentarer (open, read, ISR, bottom half) .....	9
Kapitel 4: Device Tree og Platform Drivers .....	12
4.1 Introduktion til Device Tree.....	12
4.2 Device Tree Bindinger.....	12
4.3 Eksempel Device Tree fragment .....	12
4.4 Platform Driver probe og release .....	12
4.5 Registrering af platform driver .....	13
4.6 Platform driver vs. Character driver .....	14
Kapitel 5: SPI, I2C og Andre Bussesystemer .....	15
5.1 Introduktion til Busser i Linux.....	15
5.2 SPI – Grundlæggende .....	15
5.3 SPI CPHA (Clock Phase) kort forklaret.....	15
5.4 Hvordan aflæses CPHA på timingdiagram? .....	15
5.5 SPI i Linux Driver .....	17
5.6 Eksempel: SPI læsning af 16-bit data med 8-bit transfers .....	17
5.7 I2C – Grundlæggende.....	18
5.8 I2C i Linux Driver .....	18
5.9 Eksempel: Skrive 16-bit værdi til I2C register.....	18
5.10 Device Tree eksempler til SPI og I2C .....	19
5.11 Eksamensspørgsmål (eksempler) .....	19
Kapitel 6: Timers og Workqueues .....	20
6.1 Timers i Linux Kernel .....	20

6.2 Oprettelse og brug af en timer .....	20
6.3 Vigtige pointer om timers.....	21
6.4 Workqueues – Deferred Work i process-kontekst.....	21
6.5 Oprettelse og brug af workqueue .....	21
6.6 Sammenfatning .....	21
6.7 Eksamensspørgsmål (eksempler).....	22
Kapitel 7: Memory Management og MMU .....	23
7.1 Introduktion til Memory Management .....	23
7.2 MMU (Memory Management Unit).....	23
7.3 Page Tables .....	23
7.4 Translation Lookaside Buffer (TLB).....	23
7.5 Address Translation Flow .....	23
7.6 Kernel Memory Allocation .....	24
7.7 Eksempel: Allokering og frigivelse af kernel hukommelse .....	24
7.8 Eksamensspørgsmål (eksempler).....	25
Kapitel 8: Kommunikation mellem User Space og Kernel .....	26
8.1 Introduktion.....	26
8.2 Systemkald og Device Filer.....	26
8.3 Kernel-User Data Transfer .....	26
8.4 Eksempel: Read og Write funktioner med kommentarer .....	26
8.5 Brug af ioctl() for kontrol.....	27
8.6 Eksamensspørgsmål (eksempler).....	27
Kapitel 9: Debugging og Logning .....	28
9.1 printk og kernel logs .....	28
9.2 Tips til effektiv debugging i kernel.....	28
9.3 Eksamensrelevante spørgsmål.....	28
Kapitel 10: Prioritering og Rækkefølge af Metodekald i Linux Device Drivers.....	29
10.1 Initialisering (probe / init).....	29
10.2 Ressourceanmodning .....	29
10.3 Device oprettelse.....	30
10.4 IRQ registrering .....	30
10.5 Afslutning (remove / exit) .....	32
10.6 Filoperationer .....	32
10.7 Andre typiske rækkefølge-spørgsmål .....	33
10.8 Typiske fejl ved forkert rækkefølge .....	33
Kapitel 11: Generel Fejlhåndtering i Linux Device Drivers .....	34

11.1 Hvorfor fejlhåndtering er vigtig .....	34
11.2 Fejlhåndtering i probe() og init() .....	34
11.3 Eksempel på probe() med fejlhåndtering og rollback .....	34
11.4 Fejlhåndtering i remove() og exit() .....	35
11.5 Fejlhåndtering ved resource requests .....	35
11.6 Generelle gode råd.....	35
11.7 Fejlhåndterings-template til probe(), remove(), init() og exit() .....	35

# Kapitel 1: Introduktion til Linux og Hardware Abstraktion

## 1.1 Generelle koncepter

- Linux er en monolitisk kernel med dynamisk load af moduler (drivers osv.).
- Alt i Linux opfattes som filer, inkl. hardware via device nodes i /dev.
- HAL (Hardware Abstraction Layer) sikrer, at user-space applikationer ikke skal kende hardwaredetaljer.
- Kommunikation mellem user-space og kernel sker via systemkald og device filer.

## 1.2 Device Nodes og Udev

- Device nodes repræsenterer hardware i /dev via major/minor numre.
- udev kører i user space og opretter/fjerner automatisk device nodes.
- Manual oprettelse med mknod er sjældent nødvendig i moderne systemer.

### Dynamisk oprettelse og fjernelse af device node i kernel driver:

```
static dev_t devno;
static struct class *my_class;

int init_module(void) {
    alloc_chrdev_region(&devno, 0, 1, "mydevice"); // Alloker major/minor
    my_class = class_create(TTHIS_MODULE, "my_class"); // Opret class
    device_create(my_class, NULL, devno, NULL, "mydev0"); // Opret device node
    /dev/mydev0
    return 0;
}

void cleanup_module(void) {
    device_destroy(my_class, devno); // Fjern device node
    class_destroy(my_class); // Fjern class
    unregister_chrdev_region(devno, 1); // Frigiv major/minor
}
```

## 1.3 Grundlæggende Linux kommandoer

- pwd – viser nuværende mappe.
- ps -A – viser alle kørende processer.
- scp user@ip:file . – kopierer filer fra remote maskine til lokal.
- dmesg – viser kernel log.

# Kapitel 2: Kernel Interface og Kernel Modules

## 2.1 Kernel Modules – Grundlæggende

- Kernel moduler kan indlæses og fjernes dynamisk uden reboot.
- Init- og exit-funktioner angives med module\_init() og module\_exit().

```
static int __init my_module_init(void) {
    printk(KERN_INFO "My module loaded\n");
    return 0;
}

static void __exit my_module_exit(void) {
    printk(KERN_INFO "My module unloaded\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

## 2.2 Registrering af Character Devices

- Dynamisk allokering af major/minor numre via alloc\_chrdev\_region().
- Initialisering af cdev med cdev\_init().
- Tilføjelse af cdev til kernel med cdev\_add().
- Oprettelse af device class og device node med class\_create() og device\_create().

## 2.3 Ansvarsområder for vigtige filoperationer

Funktion	Ansvarsområde
init()	Initialiserer driver, allokerer ressourcer, registrerer devices
exit()	Frigiver ressourcer, sletter devices, rydder op
open()	Åbner device, initialiserer state, allokerer eventuelle brugerressourcer
read()	Læser data, håndterer synkronisering, kopierer til user space
write()	Modtager data, opdaterer device state
release()	Lukker device, frigiver eventuelle låste ressourcer

## 2.4 Eksempel på simpel driver med kommentarer

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>           // Fil operationer
#include <linux/cdev.h>          // Char device struktur og funktioner
#include <linux/uaccess.h>        // copy_to_user, copy_from_user

static dev_t devno;             // Holder major og minor nummer for device
static struct cdev my_cdev;     // Char device struktur
static struct class *my_class;   // Sysfs klasse pointer

// open() kaldes når /dev/mydevice åbnes
// Her kan initialisering eller allokering pr. filhåndtag ske
// Skal være hurtig og må ikke sove
static int my_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device opened\n");
    return 0; // Returner 0 for succes
}

// read() kaldes når data skal læses fra device til user space
// Håndterer offset for korrekt læsning og EOF
// copy_to_user() kopierer sikkert data til user space buffer
static ssize_t my_read(struct file *file, char __user *buf, size_t count, loff_t
*offset) {
    char msg[] = "Hello from kernel\n";
    size_t len = sizeof(msg);

    // Returner 0 (EOF), hvis al data er læst
    if (*offset >= len)
        return 0;

    // Kopier data til user space, returner -EFAULT hvis fejler
    if (copy_to_user(buf, msg + *offset, len - *offset))
        return -EFAULT;

    *offset = len; // Opdater offset så næste read returnerer EOF
    return len; // Returner antal bytes læst
}

// write() kaldes når user space skriver data til device
// Begrænser kopieret data til kernel buffer størrelse for at undgå overflow
// copy_from_user() kopierer sikkert data fra user space
static ssize_t my_write(struct file *file, const char __user *buf, size_t count,
loff_t *offset) {
    char kbuf[100];

    if (count > sizeof(kbuf))
```

```

    count = sizeof(kbuf); // Begræns count for sikkerhed

    if (copy_from_user(kbuf, buf, count))
        return -EFAULT; // Returner fejl hvis kopiering mislykkes

    printk(KERN_INFO "Received from user: %.*s\n", (int)count, kbuf);
    return count; // Returner antal bytes modtaget
}

// release() kaldes når filen lukkes
// Her kan man frigive ressourcer allokeret i open()
static int my_release(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device closed\n");
    return 0;
}

// Struct der indeholder pointers til driverens fil operationer
static struct file_operations my_fops = {
    .owner = THIS_MODULE, // Angiver at modulen ejer disse operationer
    .open = my_open,      // open() funktion
    .read = my_read,      // read() funktion
    .write = my_write,    // write() funktion
    .release = my_release, // release() funktion
};

// Modul init funktion, registrerer device i kernel og opretter device node
static int __init my_driver_init(void) {
    int ret;

    // Dynamisk allokering af major og minor nummer til device
    ret = alloc_chrdev_region(&devno, 0, 1, "mydev");
    if (ret) {
        printk(KERN_ERR "Failed to allocate chrdev region\n");
        return ret; // Returner fejl hvis det mislykkes
    }

    // Initialiserer char device strukturen med vores file_operations
    cdev_init(&my_cdev, &my_fops);

    // Registrer char device med kernel
    ret = cdev_add(&my_cdev, devno, 1);
    if (ret) {
        printk(KERN_ERR "Failed to add cdev\n");
        unregister_chrdev_region(devno, 1); // Ryd op hvis fejl
        return ret;
    }

    // Opretter en device class, som gør at device vises i sysfs og /dev
}

```

```

my_class = class_create(THIS_MODULE, "myclass");
if (IS_ERR(my_class)) {
    printk(KERN_ERR "Failed to create class\n");
    cdev_del(&my_cdev); // Ryd op ved fejl
    unregister_chrdev_region(devno, 1);
    return PTR_ERR(my_class);
}

// Opretter device node i /dev/mydevice
device_create(my_class, NULL, devno, NULL, "mydevice");

printk(KERN_INFO "Driver loaded (major=%d, minor=%d)\n", MAJOR(devno),
MINOR(devno));
return 0;
}

// Modul exit funktion, rydder op ved module unload
static void __exit my_driver_exit(void) {
    device_destroy(my_class, devno); // Slet device node
    class_destroy(my_class); // Slet class
    cdev_del(&my_cdev); // Slet cdev struktur
    unregister_chrdev_region(devno, 1); // Frigiv major/minor numre

    printk(KERN_INFO "Driver unloaded\n");
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dig selv");
MODULE_DESCRIPTION("Eksempel på driver med open, read, write og release med
kommentarer");

```

# Kapitel 3: Interrupts og Deferred Work (Bottom Half)

## 3.1 Grundlæggende om Interrupts

- Interrupts afbryder CPU'en ved hardwareevents og kræver hurtig behandling i Interrupt Service Routine (ISR).
  - ISR kaldes **top half**, skal være kort og hurtig, må ikke sove eller blokere.
  - Tungere behandling udskydes til **bottom half**, som kører i process-kontekst via workqueues eller tasklets.
  - Bottom half kan sove og udføre længerevarende arbejde.
- 

## 3.2 Samspil mellem User Space og Interrupts

- open(), read(), write() og release() giver user space interface til device.
  - ISR læser hardwarestatus og planlægger deferred work.
  - Bottom half bearbejder data og vækker eventuelt ventende read() med wake\_up\_interruptible().
  - read() kan vente på data via wait\_event\_interruptible().
- 

## 3.3 Kodeeksempel med kommentarer (open, read, ISR, bottom half)

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/interrupt.h>
#include <linux/workqueue.h>
#include <linux/wait.h>
#include <linux/sched.h>

static dev_t devno;
static struct cdev my_cdev;
static struct class *my_class;

static DECLARE_WAIT_QUEUE_HEAD(read_queue);      // Ventekø for read()
static int data_available = 0;                    // Flag om data er klar
static char kernel_buffer[100];                  // Buffer til data

static struct work_struct my_work;               // Work struct til deferred
                                                // behandling

// Bottom half: kører i process-kontekst
static void my_work_handler(struct work_struct *work) {
    printk(KERN_INFO "Bottom half: processing data\n");

    // Simulerer data behandling og gemmer resultatet i kernel_buffer
```

```

        snprintf(kernel_buffer, sizeof(kernel_buffer), "Data processed at
jiffies=%lu\n", jiffies);

        // Marker data som klar og væk op læsere i ventekøen
        data_available = 1;
        wake_up_interruptible(&read_queue);
    }

// Top half: ISR, hurtig og kort
static irqreturn_t my_isr(int irq, void *dev_id) {
    printk(KERN_INFO "Top half: Interrupt received\n");

    // Planlæg bottom half arbejde
    schedule_work(&my_work);

    return IRQ_HANDLED;
}

// open() initialiserer driver state ved åbn af device
static int my_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device opened\n");
    data_available = 0; // Ingen data klar ved åbn
    return 0;
}

// read() venter på at data er klar og kopierer til user space
static ssize_t my_read(struct file *file, char __user *buf, size_t count, loff_t
*offset) {
    int ret;

    // Vent (bloker) på at data_available bliver sat af bottom half
    if (wait_event_interruptible(read_queue, data_available))
        return -ERESTARTSYS;

    if (*offset >= sizeof(kernel_buffer))
        return 0; // EOF

    if (count > sizeof(kernel_buffer) - *offset)
        count = sizeof(kernel_buffer) - *offset;

    ret = copy_to_user(buf, kernel_buffer + *offset, count);
    if (ret != 0)
        return -EFAULT;

    *offset += count;
    data_available = 0; // Reset flag for næste data

    return count;
}

// release() kaldes ved luk af device
static int my_release(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device closed\n");
    return 0;
}

static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
}

```

```

    .read = my_read,
    .release = my_release,
};

static int __init my_driver_init(void) {
    int ret;

    INIT_WORK(&my_work, my_work_handler); // Initier work struct

    ret = alloc_chrdev_region(&devno, 0, 1, "mydev");
    if (ret) return ret;

    cdev_init(&my_cdev, &my_fops);
    ret = cdev_add(&my_cdev, devno, 1);
    if (ret) return ret;

    my_class = class_create(THIS_MODULE, "myclass");
    if (IS_ERR(my_class)) return PTR_ERR(my_class);

    device_create(my_class, NULL, devno, NULL, "mydevice");

    ret = request_irq(IRQ_NUMBER, my_isr, IRQF_SHARED, "my_irq", NULL);
    if (ret) return ret;

    printk(KERN_INFO "Driver loaded and IRQ registered\n");
    return 0;
}

static void __exit my_driver_exit(void) {
    free_irq(IRQ_NUMBER, NULL);
    flush_work(&my_work);

    device_destroy(my_class, devno);
    class_destroy(my_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(devno, 1);

    printk(KERN_INFO "Driver unloaded\n");
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver with top and bottom half handling");

```

# Kapitel 4: Device Tree og Platform Drivers

## 4.1 Introduktion til Device Tree

- Device Tree beskriver hardware og forbindelser til kernel uafhængigt af platform.
- Device træet kompileres til binært overlay, som kernel loader ved boot.

## 4.2 Device Tree Bindinger

- Enheder defineres med compatible, reg, interrupts osv.
- Kernel bruger disse til at matche driver og device.

## 4.3 Eksempel Device Tree fragment

```
&gpio {  
    mygpio: mygpio@0 {  
        compatible = "myvendor,mygpio";  
        gpios = <57 0>; // GPIO 57 som input  
    };  
};
```

## 4.4 Platform Driver probe og release

- Platform drivers bruger device tree til at matche devices med drivere baseret på compatible string.
- Driverens probe() kaldes når device matches.
- of\_gpio\_count(), gpio\_request(), gpio\_direction\_input() er typiske kald i probe().

```
static int my_platform_probe(struct platform_device *pdev) {  
    int gpio_count, i, ret;  
  
    gpio_count = of_gpio_count(pdev->dev.of_node);  
    if (gpio_count < 0) return gpio_count;  
  
    for (i = 0; i < gpio_count; i++) {  
        int gpio = of_get_gpio(pdev->dev.of_node, i);  
        if (gpio < 0) return gpio;  
  
        ret = gpio_request(gpio, "mygpio");  
        if (ret) return ret;  
  
        ret = gpio_direction_input(gpio);  
        if (ret) {  
            gpio_free(gpio);  
            return ret;  
        }  
    }  
}
```

```

    device_create(my_class, NULL, devno, NULL, "mydevice");
    return 0;
}

static int my_platform_remove(struct platform_device *pdev) {
    int gpio_count, i;

    gpio_count = of_gpio_count(pdev->dev.of_node);
    if (gpio_count < 0) return gpio_count;

    for (i = 0; i < gpio_count; i++) {
        int gpio = of_get_gpio(pdev->dev.of_node, i);
        if (gpio >= 0) gpio_free(gpio);
    }

    device_destroy(my_class, devno);
    return 0;
}

```

## 4.5 Registrering af platform driver

```

// Device Tree match tabel: definerer hvilke device tree "compatible"
strings driveren understøtter
static const struct of_device_id my_of_match[] = {
    { .compatible = "myvendor,mydevice", },
    { /* sentinel */ } // Marker slutningen af tabellen
};
MODULE_DEVICE_TABLE(of, my_of_match);

// Definerer platform driver strukturen
static struct platform_driver my_platform_driver = {
    .probe = my_platform_probe,          // Kaldt når device matches og skal
initialiseres
    .remove = my_platform_remove,       // Kaldt når device fjernes
    .driver = {
        .name = "my_platform_device", // Navn på driver
        .of_match_table = my_of_match, // Pointer til device tree match
table
    },
};

// Makro til at registrere platform driver ved module load og afregistrere
ved unload
module_platform_driver(my_platform_driver);

```

## 4.6 Platform driver vs. Character driver

- **Platform driver:**
  - Hardware-orienteret driver, som styrer platformspecifikke enheder baseret på device tree.
  - Har probe() og remove() til at alloker og frigive hardware-ressourcer.
  - Kan implementere forskellige device-typer (char, block, netværk).
- **Character driver:**
  - Interface-lag til brugerprogrammer via device filer i /dev.
  - Definerer open(), read(), write(), release() for dataoverførsel.
  - Kan være implementeret som en platform driver hvis den styrer hardware.

# Kapitel 5: SPI, I2C og Andre Bussesystemer

## 5.1 Introduktion til Busser i Linux

- **SPI (Serial Peripheral Interface)** og **I2C (Inter-Integrated Circuit)** er almindeligt brugte serielle busser til kommunikation med perifere enheder.
  - Linux understøtter begge via dedikerede subsystems med standardiserede API'er og driver-modeller.
  - Enheder beskrives typisk i device tree med relevante properties for bus-adressering, clock, GPIOs osv.
- 

## 5.2 SPI – Grundlæggende

- SPI er et full-duplex serielt kommunikationsprotokol med 4 ledninger: MOSI, MISO, SCLK, og CS (chip select).
  - Data sendes og modtages synkront med clock.
  - Konfiguration inkluderer clock polarity (CPOL), phase (CPHA), og bits per word.
- 

## 5.3 SPI CPHA (Clock Phase) kort forklaret

- **CPHA (Clock Phase)** bestemmer, på hvilken clock edge data samples (læses) og skiftes (skubbes) ud på SPI-bussen.
  - Der findes to CPHA-tilstande:
    - **CPHA = 0:** Data samples på **første (leading) clock edge** efter CS går aktiv (typisk stigende edge, hvis CPOL=0). Data ændres (skiftes) på den **anden (trailing) edge**.
    - **CPHA = 1:** Data samples på **anden (trailing) clock edge** og skiftes på den **første (leading) edge**.
  - Med andre ord: CPHA bestemmer, om data samples sker på den første eller anden clock edge efter chip select aktiveres.
- 

## 5.4 Hvordan aflæses CPHA på timingdiagram?

1. Find tidspunktet hvor **CS (chip select)** går aktivt (ofte lavt).
2. Se på de første to clock edges (stigende eller faldende, afhængigt af CPOL).
3. Hvis data (MISO/MOSI) bliver **læst/samples** på den **første clock edge** efter CS aktiveres, er **CPHA=0**.
4. Hvis data derimod først samples på den **anden clock edge**, er **CPHA=1**.

## Huskeregel med én sætning pr. mode

### 1. Mode 0 (CPOL=0, CPHA=0)

**Clock starter lav.** Data samples på *første stigning (rising edge)* efter CS# går lav.  
(Data skal være klar allerede før første clock edge.)

### 2. Mode 1 (CPOL=0, CPHA=1)

**Clock starter lav.** Data samples på *første fald (falling edge)* efter CS# går lav.  
(Data kan opdateres ved rising edge – samples først ved falling.)

### 3. Mode 2 (CPOL=1, CPHA=0)

**Clock starter høj.** Data samples på *første fald (falling edge)* efter CS# går lav.  
(Data skal være klar før første clock edge.)

### 4. Mode 3 (CPOL=1, CPHA=1)

**Clock starter høj.** Data samples på *første stigning (rising edge)* efter CS# går lav.  
(Data kan opdateres ved falling edge – samples først ved rising.)

---

## Visuelt lynskema (“hvornår er clock high/low” og “hvornår samples”)

- **CPOL = 0:** Clock **idle = lav** (står i 0, når ikke aktiv)
  - **CPOL = 1:** Clock **idle = høj** (står i 1, når ikke aktiv)
  - **CPHA = 0:** Data samples på *første edge* efter CS# går lav  
(Data skal være klar straks, clock aktiveres)
  - **CPHA = 1:** Data samples på *andet edge* efter CS# går lav  
(Data må ændres efter første edge, samples først næste gang)
- 

## Nem praktisk metode (trin-for-trin)

### 1. Kig på clock, når CS# går lav

- Hvis clock starter *lavt* → CPOL = 0
- Hvis clock starter *højt* → CPOL = 1

### 2. Se hvornår data bliver samlet

- Hvis **på første edge** (straks): CPHA = 0
- Hvis **på andet edge** (altså et dummy edge først): CPHA = 1

### 3. Identificer edge-type

- Ved CPOL=0: Første edge er altid *rising*
- Ved CPOL=1: Første edge er altid *falling*

---

## 5.5 SPI i Linux Driver

- struct spi\_device repræsenterer en SPI slave enhed.
  - Kommunikation foregår med spi\_sync(), spi\_async(), eller spi\_write()/spi\_read().
  - Data overføres via spi\_message og spi\_transfer structs.
- 

## 5.6 Eksempel: SPI læsning af 16-bit data med 8-bit transfers

```
int spi_read_16bit(struct spi_device *spi, u8 cmd, u16 *data) {  
    struct spi_transfer t[3];      // Array af SPI transfers  
    struct spi_message m;         // SPI besked, samler transfers  
    u8 data_msb, data_lsb;        // Modtagne bytes  
    int ret;  
  
    memset(t, 0, sizeof(t));       // Nulstil transfer array  
    spi_message_init(&m);         // Initialiser besked  
  
    // Send kommando byte  
    t[0].tx_buf = &cmd;  
    t[0].len = 1;  
    spi_message_add_tail(&t[0], &m);  
  
    // Modtag MSB  
    t[1].rx_buf = &data_msb;  
    t[1].len = 1;  
    spi_message_add_tail(&t[1], &m);  
  
    // Modtag LSB  
    t[2].rx_buf = &data_lsb;  
    t[2].len = 1;  
    spi_message_add_tail(&t[2], &m);  
  
    ret = spi_sync(spi, &m);       // Send besked og vent på svar  
    if (ret)  
        return ret;  
  
    *data = (data_msb << 8) | data_lsb; // Sammensæt data  
    return 0;  
}
```

## 5.7 I2C – Grundlæggende

- I2C bruger to ledninger: SDA (data) og SCL (clock).
  - En master styrer clock og adresserer slaves via 7- eller 10-bit adresser.
  - Data sendes sekventielt i bytes med ACK/NACK signalering.
- 

## 5.8 I2C i Linux Driver

- struct i2c\_client repræsenterer en slave enhed.
  - Standard funktioner til at læse/skrive via SMBus:
    - i2c\_smbus\_read\_byte(), i2c\_smbus\_write\_byte()
    - i2c\_smbus\_read\_word\_data(), i2c\_smbus\_write\_word\_data()
  - Driveren bruger ofte disse funktioner i probe(), read(), eller i attribute callbacks.
- 

## 5.9 Eksempel: Skrive 16-bit værdi til I2C register

```
static int set_threshold(struct i2c_client *client, u16 threshold) {  
    int ret;  
  
    // Skriv 16-bit word til register 0x86 (eksempel)  
    ret = i2c_smbus_write_word_data(client, 0x86, threshold);  
    if (ret < 0) {  
        dev_err(&client->dev, "Failed to write threshold\n");  
        return ret;  
    }  
  
    return 0;  
}
```

## 5.10 Device Tree eksempler til SPI og I2C

```
&spi0 {                                // SPI-controller 0 på boardet
    status = "okay";                      // Aktiverer SPI-controlleren

    my_spi_device: mydevice@0 {  // SPI slave device med chip select 0
        compatible = "myvendor,myspi"; // Driverens match-string
        reg = <0>;                  // Chip Select (CS0)
        spi-max-frequency = <1000000>; // Maks frekvens 1 MHz
        spi-cpol;                   // Clock polarity = 1 (idle høj)
        spi-cpha;                   // Clock phase = 1 (data sampled på anden
clock edge)
    };
};

&i2c1 {                                // I2C-controller 1 på boardet
    status = "okay";                      // Aktiverer I2C-controlleren

    my_i2c_device: myi2c@50 {   // I2C slave device med adresse 0x50
        compatible = "myvendor,myi2c"; // Driverens match-string
        reg = <0x50>;                // Slave adresse
    };
};
```

## 5.11 Eksamensspørgsmål (eksempler)

Spørgsmål	Svar / nøgleord
Hvordan læses 16-bit data over SPI med 8-bit transfers?	Flere spi_transfer med 1 byte pr transfer
Hvilken funktion i Linux driver anvendes til at sende en spi_message?	spi_sync()
Hvordan skrives 16-bit word til et i2c register?	i2c_smbus_write_word_data()
Hvilke Device Tree properties bruges til SPI?	compatible, reg, spi-max-frequency, spi-cpol, spi-cpha
Hvilke Device Tree properties bruges til I2C?	compatible, reg

# Kapitel 6: Timers og Workqueues

## 6.1 Timers i Linux Kernel

- Timers bruges til at udføre opgaver på et senere tidspunkt, fx polling eller periodiske opgaver.
  - En timer består af en struct timer\_list, som man initialiserer og konfigurerer med callback-funktion og timeout.
  - Timer callback kører i interrupt-kontekst, derfor må den **ikke** blokere eller sove.
- 

## 6.2 Oprettelse og brug af en timer

```
#include <linux/timer.h>

static struct timer_list my_timer;

// Timer callback funktion, der kaldes når timeren udløber
static void my_timer_callback(struct timer_list *timer) {
    printk(KERN_INFO "Timer callback triggered\n");

    // Genstart timer hvis man ønsker periodisk timer
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(1000)); // 1000 ms = 1
sek
}

static int __init my_init(void) {
    printk(KERN_INFO "Initializing timer\n");

    // Initialiser timer med callback funktion
    timer_setup(&my_timer, my_timer_callback, 0);

    // Start timer, sæt timeout til nu + 1 sekund
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(1000));
    return 0;
}

static void __exit my_exit(void) {
    printk(KERN_INFO "Deleting timer\n");

    // Slet timer ved module exit for at undgå at callback kører efter
unload
    del_timer(&my_timer);
}
```

## 6.3 Vigtige pointer om timers

- Timer callback kører i interrupt kontekst og må derfor ikke kalde funktioner, der kan blokere eller sove, fx spi\_sync() eller wait\_event\_interruptible().
- Brug mod\_timer() til at genstarte eller ændre timeout for en timer.
- Brug del\_timer() eller del\_timer\_sync() for at slette en timer sikkert.

---

## 6.4 Workqueues – Deferred Work i process-kontekst

- Workqueues tillader at køre tunge opgaver i process-kontekst, hvor søvn og blokering er tilladt.
- Typisk bruges workqueues i interrupt-bundne drivere til at udsætte langvarige opgaver til senere, efter top half har kørt.

---

## 6.5 Oprettelse og brug af workqueue

```
#include <linux/workqueue.h>

static struct work_struct my_work;

// Deferred work handler
static void my_work_handler(struct work_struct *work) {
    printk(KERN_INFO "Workqueue handler running\n");
    // Langvarige eller blokerende opgaver kan udføres her
}

static int __init my_init(void) {
    printk(KERN_INFO "Initializing workqueue\n");
    INIT_WORK(&my_work, my_work_handler);
    schedule_work(&my_work); // Planlæg arbejde med det samme
    return 0;
}

static void __exit my_exit(void) {
    printk(KERN_INFO "Flushing workqueue\n");
    flush_work(&my_work); // Vent på at arbejdet er færdigt før unload
}
```

## 6.6 Sammenfatning

Koncept	Kørselstilstand	Blokering tilladt?	Typisk brug	🔗
Timer callback	Interrupt kontekst	Nej	Hurtige, korte tidsstyrede opgaver	
Workqueue handler	Process kontekst	Ja	Tunge, blokerende opgaver deferred fra ISR	

## 6.7 Eksamensspørgsmål (eksempler)

Spørgsmål	Svar / nøgleord
Hvordan initialiseres en timer?	<code>timer_setup()</code>
Hvordan starter man en timer?	<code>mod_timer()</code>
Må timer callback sove?	Nej
Hvilken funktion bruges til deferred work?	<code>INIT_WORK()</code> og <code>schedule_work()</code>
Hvordan venter man på at workqueue arbejde er færdigt?	<code>flush_work()</code>

# Kapitel 7: Memory Management og MMU

## 7.1 Introduktion til Memory Management

- Linux bruger en virtuel hukommelsesmodel, hvor hver proces får sit eget isolerede virtuelle adresseområde.
  - Hukommelsesstyring oversætter virtuelle adresser til fysiske adresser vha. MMU (Memory Management Unit).
  - Dette sikrer isolation, sikkerhed og effektiv hukommelsesudnyttelse.
- 

## 7.2 MMU (Memory Management Unit)

- MMU er hardware, som oversætter virtuelle adresser til fysiske adresser.
  - MMU bruger datastrukturen kaldet **page tables** til at slå op i mappings.
  - Den kontrollerer også adgangstilladelser som læse, skrive og udføre.
- 

## 7.3 Page Tables

- Virtuelle adresser opdeles i sideindeks og offset.
  - Hierarkiske page tables (2-4 niveauer) bruges til at finde fysisk side for en virtuel adresse.
  - Hver page table entry indeholder:
    - Fysisk adresse på siden.
    - Rettighedsflag (read/write/execute).
    - Statusflag (valid, dirty, accessed osv.).
- 

## 7.4 Translation Lookaside Buffer (TLB)

- TLB er en cache i MMU, som lagrer de mest brugte sideoversættelser for hurtigere adgang.
  - Ved TLB hit returneres fysisk adresse hurtigt.
  - Ved TLB miss går MMU til page tables for opslag, og opdaterer TLB.
  - TLB forbedrer performance betydeligt.
- 

## 7.5 Address Translation Flow

1. Processen benytter en virtuel adresse.

2. MMU søger først i TLB for oversættelse.
  3. TLB hit → hurtig adgang til fysisk adresse.
  4. TLB miss → opslag i page tables.
  5. Page fault → kernel håndterer ved at hente eller initialisere siden.
  6. TLB opdateres med den nye mapping.
- 

## 7.6 Kernel Memory Allocation

- Kernel allokérer hukommelse med:
    - kmalloc() til sammenhængende fysisk hukommelse (små blokke).
    - vmalloc() til sammenhængende virtuel hukommelse (kan være ikke-fysisk sammenhængende).
  - Hukommelse skal frigives med kfree().
- 

## 7.7 Eksempel: Allokering og frigivelse af kernel hukommelse

```
#include <linux/slab.h>

void example_alloc(void) {
    char *buffer;

    // Alloker 256 bytes sammenhængende fysisk hukommelse
    buffer = kmalloc(256, GFP_KERNEL);
    if (!buffer) {
        printk(KERN_ERR "kmalloc failed\n");
        return;
    }

    // Brug buffer til data...

    kfree(buffer); // Frigiv hukommelsen efter brug
}
```

## 7.8 Eksamensspørgsmål (eksempler)

Spørgsmål	Svar / nøgleord
Hvad er MMU's primære funktion?	Oversætte virtuelle adresser til fysiske adresser
Hvad er en page table?	Datastruktur til hukommelses-mapping og rettigheder
Hvad er TLB?	Hurtig cache for sideoversættelser i MMU
Hvordan fungerer TLB?	Cacher sideoversættelser for hurtigere oversættelse
Hvad sker ved en page fault?	Kernel henter siden fra swap eller initialiserer den
Hvilke funktioner bruges i kernel til hukommelsesallokering?	kmalloc(), vmalloc()

# Kapitel 8: Kommunikation mellem User Space og Kernel

## 8.1 Introduktion

- Kommunikation mellem user space applikationer og kernel drivere foregår typisk via device filer i /dev.
  - Interface sker via filoperationerne: open(), read(), write(), ioctl(), mmap() osv.
  - Data overføres via sikre kopier mellem user space og kernel space.
- 

## 8.2 Systemkald og Device Filer

- Når user space kalder open("/dev/mydevice"), kaldes driverens open() funktion.
  - read() og write() bruges til dataoverførsel.
  - ioctl() bruges til kontrol og konfiguration.
  - mmap() kan mappe kernel hukommelse direkte til user space for høj performance.
- 

## 8.3 Kernel-User Data Transfer

- Brug **copy\_to\_user()** til at sende data fra kernel til user space.
  - Brug **copy\_from\_user()** til at modtage data fra user space til kernel.
  - Disse funktioner sikrer sikkerhed, da kernel ikke må tilgå user space direkte.
- 

## 8.4 Eksempel: Read og Write funktioner med kommentarer

```
// read: Kopierer data fra kernel buffer til user space
static ssize_t my_read(struct file *file, char __user *buf, size_t count,
loff_t *offset) {
    char data[] = "Data from kernel\n";
    size_t datalen = sizeof(data);

    if (*offset >= datalen)
        return 0; // EOF

    if (count > datalen - *offset)
        count = datalen - *offset;

    // Kopier til user space, returner -EFAULT ved fejl
    if (copy_to_user(buf, data + *offset, count))
        return -EFAULT;

    *offset += count;
```

```

    return count;
}

// write: Kopierer data fra user space til kernel buffer
static ssize_t my_write(struct file *file, const char __user *buf, size_t
count, loff_t *offset) {
    char kbuf[100];

    if (count > sizeof(kbuf))
        count = sizeof(kbuf);

    // Kopier fra user space, returner -EFAULT ved fejl
    if (copy_from_user(kbuf, buf, count))
        return -EFAULT;

    printk(KERN_INFO "Received from user: %.s\n", (int)count, kbuf);

    return count;
}

```

## 8.5 Brug af ioctl() for kontrol

- ioctl() bruges til at sende kontrolkommandoer og konfigurationer til driveren.
- Kommandoerne defineres med makroer i en headerfil.
- Driverens unlocked\_ioctl() fortolker og udfører kommandoerne.

## 8.6 Eksamensspørgsmål (eksempler)

Spørgsmål	Svar / nøgleord
Hvordan kopierer man data fra kernel til user space?	copy_to_user()
Hvordan kopierer man data fra user til kernel?	copy_from_user()
Hvilke filoperationer bruges til user-kernel kommunikation?	open(), read(), write(), ioctl()
Hvad bruges mmap() til i driver-kontekst?	Mappe kernel hukommelse til user space

# Kapitel 9: Debugging og Logning

## 9.1 printk og kernel logs

- **printk()** er Linux kernens måde at skrive logbeskeder på — svarer til printf i user space.
- Logniveauer som KERN\_INFO, KERN\_ERR bruges til at kategorisere vigtighed.
- Output kan læses med kommandoen dmesg eller i filen /var/log/kern.log.

**Eksempel:**

```
printk(KERN_INFO "Driver loaded successfully\n");
printk(KERN_ERR "Error: device not responding\n");
```

## 9.2 Tips til effektiv debugging i kernel

- Brug printk() til at spore flow og fejl i driveren.
- Undgå at fyldе kernel loggen unødig — brug passende logniveauer.
- Brug dmesg til at inspicere kernel logs efter test/kørsel.
- Kommenter og strukturér printk-beskeder for nemmere fejlfinding.

---

## 9.3 Eksamensrelevante spørgsmål

Spørgsmål	Svar
Hvilken funktion bruges til kernel log?	printk()
Hvordan ser man kernel log beskeder?	dmesg
Hvilke printk logniveauer kendes?	KERN_INFO, KERN_ERR osv.

# Kapitel 10: Prioritering og Rækkefølge af Metodekald i Linux Device Drivers

## 10.1 Initialisering (probe / init)

- probe() kaldes ved device-driver match og indeholder typisk:
    1. Læs hardware-ressourcer (of\_gpio\_count(), platform\_get\_resource())
    2. Alloker ressourcer (gpio\_request(), clk\_get())
    3. Konfigurer hardware (gpio\_direction\_input(), register opsætning)
    4. Registrer interrupts (request\_irq())
    5. Initialiser software-strukturer (timers, workqueues)
    6. Opret device node (device\_create())
  - init() kalder ofte platform\_driver\_register() og global init.
- 

## 10.2 Ressourceanmodning

- Typisk rækkefølge:
  1. alloc\_chrdev\_region()
  2. class\_create()
  3. gpio\_request()
  4. request\_irq()
- Ved fejl frigøres ressourcer i omvendt rækkefølge.

### Rækkefølge af store API-funktionskald under driver-load

1. **alloc\_chrdev\_region()**
  - Allokerer major/minor nummer til din karakter-enhed.
  - KALDES typisk i module\_init/init-funktion.
2. **class\_create()**
  - Opretter sysfs class, der bruges til device nodes (valgfri, men næsten altid nødvendig for moderne drivere).
  - KALDES i module\_init/init-funktion (ofte lige efter alloc\_chrdev\_region).
3. **platform\_driver\_register()**
  - Registrerer din platform driver hos kernel.
  - KALDES i module\_init/init-funktion (efter ovenstående).

4. (*Kernel matcher nu din driver med en device fra device tree, og kalder din probe()!*)
5. **of\_gpio\_count()**
  - Kaldes i din probe() for at læse antallet af GPIO'er i device tree.
6. **gpio\_request()** (*og evt. andre resource-requests*)
  - Reservér hardware ressourcer (fx GPIO, IRQ, mm.).
  - Kaldes i probe() lige efter du har læst konfiguration.
7. **cdev\_init()** og **cdev\_add()**
  - Initialiserer og tilføjer din char device til kernel.
  - Kaldes i probe() (eller module\_init for simple drivere).
8. **device\_create()**
  - Opretter device node i /dev.
  - Kaldes i slutningen af din probe() efter alle ressourcer er klar.

---

## 10.3 Device oprettelse

- class\_create() → device\_create() → opsæt sysfs attributter.
- 

## 10.4 IRQ registrering

- request\_irq() kaldes efter hardware-konfiguration og resource-allokering.
- free\_irq() kaldes **før** frigivelse af hardware-ressourcer ved remove().

**Top half (ISR):**

**Formål:**

Kør så lidt som muligt, og ALTID så hurtigt som muligt! Må IKKE sove eller lave langvarigt arbejde.

**Her skal du udføre:**

1. **Check om gpio pin reelt er aktiveret**
  - Læs den relevante status direkte i ISR'en for at afvise spurious interrupts.
  - Dette er typisk et enkelt register-read (hurtigt).
2. **Lav tidsstempel**
  - Gem f.eks. ktime\_get() eller jiffies i en buffer/variabel.
  - Tidsstempel er hurtigt at tage, så det er ok i top half.

### 3. Planlæg bottom half

- Schedule et workqueue-job/tasklet til resten (se nedenfor).
- 

#### Bottom half (workqueue/tasklet):

##### Formål:

Alt der tager tid, kan sove eller involverer user space/datahåndtering.

##### Her skal du udføre:

#### 4. Læs data via SPI

- SPI kan være langsomt og involverer ofte kernel-funktioner der må sove (fx spi\_sync()).
- Dette MÅ IKKE ligge i top half – gør det i bottom half!

#### 5. Lav checksum

- Når data er læst i ro og mag, kan du beregne checksum.
- Dette kan tage tid (afhænger af datamængde), så det skal ligge i bottom half.

#### 6. Overfør til user-space

- Værdier og resultater skal gemmes i kernel-buffere, og user space vækkes med fx wake\_up\_interruptible().
- Selve copy\_to\_user() kaldes typisk i din read()-funktion, som user space selv kalder, men vækning skal ske her.

Opgave	Top half (ISR)	Bottom half (workqueue/tasklet)
Tjek GPIO-pin	✓	
Tag tidsstempel	✓	
Læs data via SPI		✓
Beregn checksum		✓
Overfør til user space / væk læser		✓ (fx. med wake_up_interruptible())

#### Hvorfor denne opdeling?

- **Top half:** Kun ultrakorte, deterministiske opgaver! Ingen sove/ventetid.
  - **Bottom half:** Alt, der kan tage tid eller kræver kernel-funktioner som kan sove/blokere (SPI, checksum, synk til user space).
- 

#### Husk til eksamen:

- Skriv altid: "Tidskritiske, korte og ikke-sovende opgaver i top half, alt andet i bottom half."
  - SPI-adgang og databehandling = bottom half!
  - User space kommunikation = bottom half (wake\_up\_interruptible()/buffer opdatering).
- 

## 10.5 Afslutning (remove / exit)

- Ved remove():
  1. free\_irq()
  2. Frigør hardware-ressourcer (gpio\_free(), clk\_put())
  3. device\_destroy() og class\_destroy()
  4. unregister\_chrdev\_region()

Forklaring på rækkefølge ved rmmod af platform driver:

### 1. my\_driver\_exit()

Din modul-exit-funktion er *det første* kernel kalder, når du laver rmmod.  
Herinde laver du typisk **platform\_driver\_unregister()**.

### 2. platform\_driver\_unregister()

Når du afregistrerer din platform driver, vil alle matchende devices blive fjernet.  
Platform driveren vil for *hver enhed* kalde din **my\_driver\_remove()** (remove-callback).

### 3. my\_driver\_remove()

Her rydder du typisk op for hver device, **fx kalder device\_destroy()** for hver oprettet device.

### 4. device\_destroy()

Fjerner selve device-filen i /dev.

---

## 10.6 Filoperationer

Funktion	Ansvar
open()	Initialiserer per-file ressourcer
read()	Læser data, håndterer synkronisering
write()	Modtager data, opdaterer device state

release()	Frigiver per-file ressourcer
-----------	------------------------------

## 10.7 Andre typiske rækkefølge-spørgsmål

- Timer: timer\_setup(), mod\_timer(), del\_timer\_sync()
- Workqueue: INIT\_WORK(), schedule\_work(), flush\_work()
- DMA: Allocate → Configure → Start → Free
- Bus driver: spi\_register\_driver(), i2c\_add\_driver() i init; probe() og remove() til enhedshåndtering.

## 10.8 Typiske fejl ved forkert rækkefølge

Fejl	Forklaring
device_create() før class_create()	Klasse skal eksistere før device node oprettes
IRQ registreres før ressourcer allokeres	Kan føre til ustabilitet eller fejl
Ressourcer frigives i forkert rækkefølge	Kan forårsage leaks eller kernel panics
Brug af ressourcer før allokering	Kan skabe konflikter eller crashes
Manglende fejlhåndtering og oprydning	Ressourcer kan blive låst eller utilgængelige

# Kapitel 11: Generel Fejlhåndtering i Linux Device Drivers

## 11.1 Hvorfor fejlhåndtering er vigtig

- Fejl kan opstå ved hardware-ressourceanmodninger, allokeringer, IRQ-registrering, device creation osv.
  - Uden korrekt håndtering kan det føre til resource leaks, ustabil kernel eller systemcrash.
  - Derfor er robust fejlhåndtering nødvendig i alle faser af driverens livscyklus.
- 

## 11.2 Fejlhåndtering i probe() og init()

- Tjek altid returværdi for funktioner der kan fejle.
  - Hvis en fejl opstår, skal du rydde op i alle tidligere tildelte ressourcer i omvendt rækkefølge (rollback).
  - Brug goto til at organisere oprydningskode på en overskuelig måde.
- 

## 11.3 Eksempel på probe() med fejlhåndtering og rollback

```
static int my_platform_probe(struct platform_device *pdev) {
    int ret;
    int gpio_num;

    ret = gpio_request(gpio_num, "mygpio");
    if (ret) {
        dev_err(&pdev->dev, "Failed to request GPIO\n");
        return ret;
    }

    ret = request_irq(pdev->irq, my_isr, 0, "myirq", pdev);
    if (ret) {
        dev_err(&pdev->dev, "Failed to request IRQ\n");
        goto err_free_gpio;
    }

    ret = device_create(...);
    if (IS_ERR(ret)) {
        dev_err(&pdev->dev, "Failed to create device\n");
        goto err_free_irq;
    }
    return 0;

err_free_irq:
    free_irq(pdev->irq, pdev);
err_free_gpio:
    gpio_free(gpio_num);
    return ret;
}
```

## 11.4 Fejlhåndtering i remove() og exit()

- Ved fjernelse skal ressourcer frigives i omvendt rækkefølge af deres tildeling.
  - Sørg for, at alle resourcesluger bliver frigivet for at undgå leaks.
- 

## 11.5 Fejlhåndtering ved resource requests

Funktion	Fejlhåndtering
alloc_chrdev_region()	Tjek return værdi, ryd op ved fejl
class_create()	Tjek for IS_ERR(), frigør ved fejl
device_create()	Tjek for IS_ERR(), frigør ved fejl
gpio_request()	Returner fejl direkte, ryd op ved fejl
request_irq()	Returner fejl, ryd op tidligere ressourcer
kmalloc()	Returner NULL ved fejl, ryd op

## 11.6 Generelle gode råd

- Organiser oprydningskode i probe() med goto og etiketter for klarhed og vedligeholdelse.
  - Brug dev\_err() og printk() til at logge fejl og lette debugging.
  - Overvej at bruge devm\_-API'er (device managed), som automatisk rydder op ved device-fjernelse.
  - Test fejlsituationer ved at simulere fejl i anmodninger.
- 

## 11.7 Fejlhåndterings-template til probe(), remove(), init() og exit()

### Probe:

```
static int my_driver_probe(struct platform_device *pdev) {
    int ret;

    // Anmod om ressourcer 1 (fx GPIO)
    ret = gpio_request(...);
    if (ret) {
        dev_err(&pdev->dev, "Failed to request GPIO\n");
        return ret;
    }

    // Anmod om ressourcer 2 (fx IRQ)
    ret = request_irq(...);
    if (ret) {
```

```

    dev_err(&pdev->dev, "Failed to request IRQ\n");
    goto err_free_gpio; // Ryd op GPIO ved fejl
}

// Anmod om ressourcer 3 (fx device node)
ret = device_create(...);
if (IS_ERR(ret)) {
    dev_err(&pdev->dev, "Failed to create device\n");
    goto err_free_irq; // Ryd op IRQ og GPIO ved fejl
}

// Succes
return 0;

err_free_irq:
    free_irq(...);
err_free_gpio:
    gpio_free(...);
    return ret;
}

```

### Forklaring:

- Efter hver kritisk anmodning tjekkes return-værdien.
- Ved fejl springes til en oprydningsetiket, der frigør allerede allokerede ressourcer i omvendt rækkefølge.
- Oprydningskoden samles for at undgå duplikat kode og for overskuelighed.

### Remove:

```

static int my_driver_remove(struct platform_device *pdev) {
    // Frigiv device node
    device_destroy(my_class, devno);

    // Frigiv IRQ-linje
    free_irq(pdev->irq, pdev);

    // Frigiv GPIO
    gpio_free(gpio_num);

    // Frigiv clock, hvis brugt
    clk_put(my_clk);

    // Frigiv andre ressourcer efter behov

    pr_info("Driver removed successfully\n");
}

```

```
    return 0;  
}
```

#### Kommentarer:

- Frigør ressourcer i omvendt rækkefølge af tildeling.
- Sørg for at alle allokerede ressourcer i probe() friges.
- Brug printk/pr\_info/dev\_info til at logge fjernelsesstatus.

#### Init og exit:

```
static int __init my_module_init(void) {  
    int ret;  
  
    // Registrer platform driver  
    ret = platform_driver_register(&my_platform_driver);  
    if (ret) {  
        pr_err("Failed to register platform driver\n");  
        return ret;  
    }  
  
    pr_info("Module loaded\n");  
    return 0;  
}  
  
static void __exit my_module_exit(void) {  
    // Afregistrer platform driver  
    platform_driver_unregister(&my_platform_driver);  
  
    pr_info("Module unloaded\n");  
}  
  
module_init(my_module_init);  
module_exit(my_module_exit);
```

#### Kommentarer:

- Ved init skal du tjekke returværdi for registrering af driver og håndtere fejl.
- Ved exit skal du sørge for at afregistrere driver korrekt.
- Brug printk/pr\_info/pr\_err til status og fejlbeskeder.