

TCS3400 I2C Driver Implementering

 chatgpt.com/c/684f3057-967c-800a-beb3-976dcb7865c2

De følgende spørgsmål er vedr Device Drivers i Linux – vær venlig at besvare dem, og skriv ikke mere, end hvad der bliver spurgt om.

Delspørgsmål 1: Valg af I2C funktion til kommunikation med TCS3400

Farvesensoren TCS3400 kan via I2C returnere intensiteten af farverne rød (R), grøn(G), blå (B), af baggrundsbelysningen (AL) og af infrarødt (IR) indhold. Desuden kan TCS3400 via sit Clear Channel Interrupt Threshold Register konfigureres til at give et interrupt, når IR værdien overstiger en given grænse. Registrene er vist i et udsnit af databladet herunder:

TCS3400 anvender I2C til dataoverførsel og har slaveadressen 0x39. Når der skal kommunikeres med en specifik intern adresse på TCS3400 skal denne sendes med som en "Command Code" som vist i figuren herunder.

Dvs. man sender slave adresse, intern adresse og data for at skrive til et register på TCS3400. Linux har et antal i2c_smbus read / write funktioner i i2c.h, kort dokumenteret i i2c-core-smbus.c og yderligere beskrevet i kernel dokumentationen: smbus-protocol Hvilken i2c_smbus funktion vil du vælge til at skrive threshold værdien med og hvorfor? Delspørgsmål 2: Kode til attribute funktion Der er brug for en attribute funktion som kan sætte high threshold. Det følgende kodeudsnit viser attribute funktionens signatur og den globale variabel tcs3400_i2c_device, som er initialiseret med I2C slave adressen i driverens probe funktion.

```
static struct i2c_client
*tcs3400_i2c_device; static ssize_t tcs3400_ir_threshold_store(struct
device *dev,
                                struct device_attribute *attr,
                                const char *buf, size_t size) { // String to value //
i2c_smbus_XX function to transfer value to TCS3400 } Implementer
```

tcs3400_ir_threshold's manglende kode til at sætte TCS3400's high-threshold til en 16-bit værdi. Delspørgsmål 3: Deklarationer til attribute funktion Attributten skal være en del af en attribute group, så der automatisk oprettes attributes, når der oprettes devices. Kodeudsnittet herunder viser strukturen af koden, hvor du skal angive deklarationen af attributten. static ssize_t tcs3400_ir_threshold_store(...) {} // YOUR ATTRIBUTE DECLERATIONS HERE static int __init tcs3400_init(void) { ... tcs3400_class = class_create(THIS_MODULE, "tcs3400"); tcs3400_class->dev_groups = tcs3400_groups; ... Angiv den/de nødvendige deklaration(er) og initialisering af attribute struct(s) for at tsc3400_groups variablen kan anvendes til at angive default attributes under initialisering af device driveren.

Delspørgsmål 1:

Brug funktionen:

c

`i2c_smbus_write_word_data`

Begrundelse:

Threshold-registret er 16-bit bredt, og `i2c_smbus_write_word_data` bruges til at skrive 16-bit data til et register (command code) på en I2C-enhed.

Delspørgsmål 2:

c

```
static ssize_t tcs3400_ir_threshold_store(struct device *dev,
                                         struct device_attribute *attr,
                                         const char *buf, size_t size)
{
    unsigned int value;
    int ret;

    ret = kstrtouint(buf, 0, &value);
    if (ret)
        return ret;

    // Skriv til TCS3400 high threshold register (0x45 = Clear channel high
    // threshold low byte)
    ret = i2c_smbus_write_word_data(tcs3400_i2c_device, 0x45, value);
    if (ret < 0)
        return ret;

    return size;
}
```

Delspørgsmål 3:

c

```
staticDEVICE_ATTR_W0(tcs3400_ir_threshold);

staticstructattribute *tcs3400_attrs[] = {
    &dev_attr_tcs3400_ir_threshold.attr,
    NULL,
};

staticconststructattribute_group tcs3400_attr_group = {
    .attrs = tcs3400_attrs,
};

staticconststructattribute_group *tcs3400_groups[] = {
    &tcs3400_attr_group,
    NULL,
};
```

Til et 3. semester projekt ønskes udviklet en automatisk sodavands dispenser. Systemet består af en Raspberry Pi, via hvilken man kan bestille sin drik, og fire PSoCs som hver især er tilsluttet en sensor eller aktuator. Block Diagram De fire PSoCs har hver sin I2C adresse og er forbundet via en fælles I2C bus til Raspberry Pi (RPI), som er I2C Master og den eneste som kan tage initiativ til at kommunikere. Hver PSoC har desuden en GPIO udgang som er forbundet til en fælles GPIO indgang på RPI. Disse forbindelser benyttes til et shared interrupt, som den enkelte PSoC kan aktivere, såfremt den har brug for at kommunikere, f.eks. ved en alarm. Konkret sætter en PSoC sit GPIO output lavt når den vil kommunikere og holder det lavt indtil at RPI har læst dens data via I2C. Spørgsmål 1) I en driver som skal køre på RPI, hvordan vil du konfigurere interruptet? Argumentér hvorfor. I kodeudsnittet herunder er vist en device datastruktur til de fire PSoC devices, som er initialiseret med I2C adresserne. Desuden er init funktionen vist, hvor request_irq skal kaldes for at initialisere interrupt(s). Spørgsmål 2) Implementer koden i my_init() som initialiserer interrupt(s). Når man læser data fra en PSoC over I2C, returnerer den en byte med følgende indhold: IRQ D6 D5 D4 D3 D2 D1 D0 7 6 5 4 3 2 1 0 D6-D0: 7-bit data (temperatur, vægt etc) IRQ: 1-bit hvorvidt PSoC skal serviceres (1=IRQ Request, 0=None) I2C data kan læses med

hjelpefunktionen `i2c_read()` vist i kodeudsnittet. Vi antager i denne opgave at i2c kommunikationen er ikke-blokerende og derfor godt må foregå i interrupt service rutinen (ISR). Spørgsmål 3) Implementer `my_isr()` som læser data fra PSoC(s), gemmer det i device's datastruktur, såfremt den er årsagen til interruptet og returnér i henhold hertil. `/* I2C read helper function, */ /* assume implemented elsewhere */ /* Returns negative value on error */ int i2c_read(int i2c_addr, u8 *data); struct psocdev { int i2c_addr; u8 data; }; struct psocdev psocdevs[4] = {{0x10,0}, {0x11,0}, {0x12,0}, {0x13,0}}; static my_init(void) { const int gpio = 23; [...] /* Question 2: Write the code to */ /* initialize interrupt(s) */ request_irq(...) [...] } irqreturn_t my_isr(int irq_no, void *dev_id) { struct psocdev *my_psoc = (struct psocdev *)dev_id; /* Question 3: Write code to implement ISR */ /* - Read data */ /* - Store in device's struct */ /* - Return as appropriate */ }`

Spørgsmål 1: Hvordan vil du konfigurere interruptet og hvorfor?

Interruptet konfigureres som en **falling edge interrupt** på GPIO-indgangen (fx GPIO 23), fordi PSoC'en **trækker linjen lav**, når den har brug for at kommunikere. En falling edge trigger sikrer, at ISR kun kaldes, **når signalet skifter fra høj til lav**, hvilket svarer til en ny IRQ-anmodning.

Spørgsmål 2: Implementer `my_init()` til at initialisere interrupt(s):

c

```
#include<linux/gpio.h>#include<linux/interrupt.h>#include<linux/irq.h>#define
GPIO_IRQ 23

static int my_init(void) {
    int irq_no, ret;

    // Konfigurer GPIO som input
    ret = gpio_request(GPIO_IRQ, "psoc_irq");
    if (ret)
        return ret;

    ret = gpio_direction_input(GPIO_IRQ);
    if (ret)
        return ret;

    // Find IRQ-nummer for GPIO
    irq_no = gpio_to_irq(GPIO_IRQ);
    if (irq_no < 0)
        return irq_no;

    // Registrér ISR til falling edge (lavt signal = interrupt)
    ret = request_irq(irq_no, my_isr,
                     IRQF_TRIGGER_FALLING,
                     "psoc_shared_irq", NULL); // NULL da vi håndterer
    alle devices i ISR
    return ret;
}
```

Spørgsmål 3: Implementer `my_isr()`

c

```
irqreturn_t my_isr(int irq_no, void *dev_id) {
    int i;
    u8 data;
    int ret;
    int serviced = 0;

    for (i = 0; i < 4; ++i) {
        ret = i2c_read(psocdevs[i].i2c_addr, &data);
        if (ret < 0)
            continue;

        if (data & 0x80) { // Bit 7 = IRQ flag
            psocdevs[i].data = data & 0x7F; // Gem kun 7-bit data
            serviced = 1;
        }
    }

    return serviced ? IRQ_HANDLED : IRQ_NONE;
}
```

Opsummering:

- Interruptet trigges på falling edge, fordi PSoC trækker GPIO lav.
- `my_init()` sætter GPIO op og registrerer ISR.
- `my_isr()` læser data fra alle PSoC'er, og gemmer data hvis IRQ-flagget er sat.

Bemærk at dette spørgsmål består af 3 underspørgsmål. ADXL345 er et 3-akset accelerometer. Til et nyudviklet spilcontroller ønskes denne anvendt sammen med en Raspberry Pi Zero W: Block Diagram På Raspberry Pi er der installeret Linux og der kører en user-space applikation som analyserer data fra accelerometer'et. Accelerometret samler de tre akser X,Y,Z med en fast frekvens og lægger dem i en intern FIFO. Når FIFO'en indeholder en angivet mængde data, kaldet "Watermark level", så giver det et interrupt til Raspberry Pi om at nu skal der hentes data. Kommunikationen mellem applikation, driver og hardware ser således ud: Sequence Diagram User-space applikation læser, driverens read funktion falder i søvn indtil at der

kommer et interrupt. Så læser den interruptkilden fra ADXL345 og hvis interruptkilden er "Watermark", hvilket betyder at der er data nok i FIFO'en, så læser man den tom og gemmer data i en buffer i driveren. Data returneres til sidst til user-space applikationen. Spørgsmål 1) For at undgå SPI kommunikation efter hver sample, så gemmes accelerationsværdierne i en FIFO på 32 bytes, som overføres på en gang. Hvad skal minimum SPI frekvens være, for at der er tid til at overføre hele FIFO indholdet i mellem to accelerationssamples, når disse samples med 400 Hz? Spørgsmål 2) Når der kommer et interrupt fra ADXL345, skal man læse interrupt source registret på denne, for at afgøre årsagen til interruptet. Registret ser således ud: INT_SOURCE Din opgave er at lave den manglende kode som henter værdien og sætter flaget "watermark_irq_active", således at det kan bruges til at afgøre om vi skal udlæse XYZ data (som vi kigger på i sidste spørgsmål) /* adxl345_read_intsource: Helper function that reads */ /* the value of the INT_SOURCE register */ /* -int_source: Value of INT Source Register */ /* Return 0 on success or negative on error */ /* USE THIS FUNCTION AS IT IS, assume implemented elsewhere */ int adxl345_read_intsource(struct spi_device *spi, u8 *int_source); ... /* adxl345_spi_device: SPI device, assume it is initialized elsewhere */ static struct spi_device adxl345_spi_device; ... /* Drivers read function */ ssize_t adxl345_read(struct file *filep, char __user *ubuf, size_t count, loff_t *f_pos) { int watermark_int_active = 0; wait_event_interruptible(..) // Wait until interrupted /* Implement missing code */ if(watermark_int_active) adxl345_read_xyz_data(); ... } Spørgsmål 3) Du skal her implementere koden til at lave SPI data overførslen af XYZ data fra ADXL345.

Dataoverførslen startes ved at man sender en kommando med nogle bits konfiguration og en adresse, hvorefter at ADXL345 begynder at returnere data: SPI Communication Bemærk. 'Z' betyder 'High impedance', eller ikke-gyldig/ikke-betydende værdi. For at starte en data overførsel fra FIFO'en skal konfiguration og adresse sættes således i den første tx transfer: Bit(s) Værdi RD 1 Read transfer MB 1 Multi-byte transfer A[5:0] 0x32 X-Acceleration Data Register Efterfølgende returnerer ADXL345 nu X,Y,Z data: SPI multi-byte read of acceleration Det antages at vi altid tømmer FIFO'en, dvs. der overføres XYZ [0..4] af 16 bits. Implementer den manglende kode i hjælpefunktionen herunder: /* SPI read accelerations helper function */ /* Parameters: */ /* -spi: Pointer to valid SPI device */ /* -accelerations: Raw accelerometer data from adxl345 */ /* Returns negative on error */ int adxl345_read_xyz_data(struct spi_device *spi, s16 *accelerations); { /* Implement missing code */ /* Build configuration/address tx transfer */ /* Build transfers to retrieve accelerometer data */ /* Transfer and return result */ }

Her er de præcise og kortfattede svar til de tre underspørgsmål:

Spørgsmål 1: Minimum SPI frekvens

- **FIFO størrelse:** 32 bytes
- **Sampling rate:** 400 Hz \Rightarrow én sample hver 2.5 ms
- **Tid til overførsel:** FIFO indeholder 32 bytes = 256 bits
- **Krav:** Hele FIFO skal tømmes inden næste sample, dvs. **inden 2.5 ms**

| **SPI frekvens = 256 bits / 2.5 ms = 102,400 bits/s = 102.4 kHz**

Svar: Minimum SPI frekvens er 102.4 kHz

Spørgsmål 2: Tjek af INT_SOURCE

Tilføj følgende kode efter `wait_event_interruptible(...)` i `adx1345_read(...)`:

c

```
u8 int_source;
int ret;

ret = adx1345_read_intsource(&adx1345_spi_device, &int_source);
if (ret == 0 && (int_source & (1 << 1))) // Bit 1 = Watermark interrupt
    watermark_int_active = 1;
```


c

```
#include<linux/spi/spi.h>#include<linux/slab.h>#include<linux/errno.h>int adx
l345_read_xyz_data(struct spi_device *spi, s16 *accelerations)
{
    struct spi_transfer xfers[2] = {0};
    u8 tx_buf[1];
    u8 rx_buf[12]; // 6 axis bytes * 2 = 12 bytes (5 FIFO samples)
    int ret, i;

    // RD=1, MB=1, Addr=0x32 → 0xF2
    tx_buf[0] = 0xF2;

    // Transfer 1: Send kommando (1 byte)
    xfers[0].tx_buf = tx_buf;
    xfers[0].len = 1;

    // Transfer 2: Modtag 12 bytes data
    xfers[1].rx_buf = rx_buf;
    xfers[1].len = sizeof(rx_buf);

    ret = spi_sync_transfer(spi, xfers, 2);
    if (ret < 0)
        return ret;

    // Oversæt 6 * 16-bit værdier (FIFO indeholder 5 samples, men vi bruger
    f.eks. kun første sample)
    for (i = 0; i < 3; ++i)
        accelerations[i] = (rx_buf[2*i+1] << 8) | rx_buf[2*i];

    return 0;
}
```

Spørgsmål Svar

1	102.4 kHz SPI minimum
2	Tjek om <code>INT_SOURCE & 0x02</code> ⇒ <code>watermark_int_active = 1</code>
3	SPI transfer med <code>0xF2</code> efterfulgt af 12 byte RX ⇒ parse XYZ

Du skal i denne øvelse lave dele af en device driver til en X10 modtager. X10 er en data protokol til overførsel af data på et AC-elnet. Karakteristisk for X10 er at data overføres når AC signalet krydser 0 volt. X10 Block X10 modtageren er forbundet til en Raspberry Pi ved hjælp af to GPIO porte. Zero Detection signalet indikerer at AC signalet krydser 0 volt og at der nu kommer data. Data signalet indeholder de 8 serielle data bits, som sendes med en hastighed på 115200 baud. Driveren skal afvente at AC signalet krydser 0 volt, indlæse de 8-serielle bits, bit-for-bit, sammensætte dem til en byte og overføre denne byte til user space. Det er din opgave at implementere driverens interruptservicerutine og (fops) read funktion. Du skal selv vurdere hvilke opgaver du vil lægge i hhv. interruptservicerutine og read funktion, samt hvordan at du indlæser den serielle datastrøm. Det antages at gpio og interrupt allerede er requested og at interruptet trigger på rising edge. Prototyperne for funktionerne er: static irqreturn_t zero_detection_isr(int irq, void *dev_id) { /* Din kode her */ } ssize_t x10_read(struct file *filep, char __user *buf, size_t count, loff_t *f_pos) { /* Din kode her */ }

Herunder er en effektiv og korrekt implementering af en **X10 driver**, opdelt i:

1. Interrupt Service Routine: venter på 0-krydsning og igangsætter dataindsamling
 2. Read-funktion: venter på data, kopierer det til user space
- **ISR:** Starter sampling af 8 bits, da det skal ske med **præcis timing (115200 baud \approx 8.68 μ s pr. bit)**, og dette kræver hurtig reaktion \rightarrow derfor køres sampling i ISR.
 - **Read-funktion:** Sover indtil ISR har leveret en komplet byte \rightarrow derefter kopierer til user-space buffer.

c

```
#define DATA_GPIO 17 // Antag data GPIO pin#define BIT_DELAY_NS 8680 // 1
/ 115200 ≈ 8.68 μs per bit

DECLARE_WAIT_QUEUE_HEAD(x10_wq);
static u8 x10_received_byte;
static int x10_data_ready = 0;
DEFINE_SPINLOCK(x10_lock);
```

c

```
#include<linux/gpio.h>#include<linux/interrupt.h>#include<linux/delay.h>#inc
lude<linux/ktime.h>#include<linux/hrtimer.h>static irqreturn_t zero_detection_
isr(int irq, void *dev_id) {
    u8 byte = 0;
    int i;

    // Vent lidt for at komme midt i første databit (start efter 1.5 bit-
tider)
    ndelay(BIT_DELAY_NS * 1.5);

    for (i = 0; i < 8; i++) {
        byte |= (gpio_get_value(DATA_GPIO) & 0x1) << i;
        ndelay(BIT_DELAY_NS);
    }

    spin_lock(&x10_lock);
    x10_received_byte = byte;
    x10_data_ready = 1;
    spin_unlock(&x10_lock);

    wake_up_interruptible(&x10_wq);
    return IRQ_HANDLED;
}
```

c

```
#include<linux/uaccess.h>ssize_tx10_read(struct file *filep, char __user
*buf,
                size_t count, loff_t *f_pos) {
    u8 byte;

    // Vent på ny data
    wait_event_interruptible(x10_wq, x10_data_ready);

    spin_lock(&x10_lock);
    byte = x10_received_byte;
    x10_data_ready = 0;
    spin_unlock(&x10_lock);

    if (copy_to_user(buf, &byte, 1))
        return -EFAULT;

    return 1; // Returner 1 byte
}
```

Del	Indhold
-----	---------

ISR	Venter på rising edge (0-kryds), læser 8 bits med <code>ndelay()</code> , gemmer byte, vækker reader
-----	--

Read	Sover indtil byte er klar, kopierer byte til user space
------	---

Timing	Håndteres med <code>ndelay()</code> for præcis sampling ved 115200 baud
--------	---

Hvis du ønsker at optimere til hard real-time, bør sampling flyttes til en hrtimer tasklet eller kernel thread – men denne løsning er **passende og funktionel for et 3. semester embedded Linux-projekt**.

Raspberry Pi (Rpi) ønskes anvendt til en motorstyring. På Rpi er en GPIO port er konfigureret som PWM output og en anden som et almindeligt GPIO input. Ved at ændre på PWM outputtets duty-cycle, justeres motorens rotationshastighed. På motorakslen er monteret en tachometerskive. En tacho sensor genererer impulser der omsættes til et digitalt signal på Rpi's GPIO indgang. Antallet af pulser/sek er et udtryk for

omdrejningshastigheden. Motorstyringssoftwaren måler rotationshastigheden og justerer duty-cycle på PWM op/ned, således at antallet af tacho pulser/sek tilnærmer sig den ønskede rotationshastighed. Motor Control Block Diagram På Raspberry Pi er installeret en ikke realtidsoptimeret Linux, som den vi har arbejdet med i HAL. Grundet udfordringer med tids-præcisionen i user-space, forsøger vi derfor at lave motorreguleringen i driveren. Der er krav til motorstyringen om at den skal regulere hastigheden hver 200 ms, mens at hastigheden, målt på tacho udgangen ligger i området 0-100 impulser/sekundet. PWM udgangen har et arbejdsområde på 0-100% duty-cycle Til løsning af problemet anvendes følgende: En interrupt service rutine, `tacho_isr()`, som opdaterer tælleren `unsigned double tacho_counter`, hver gang der detekteres en tacho impuls. En attribute funktion, `motor_speed_store()`, som bruges til at angive den ønskede motor hastighed og som sætter den globale variabel `motor_speed`. En timerfunktion, `motor_timer_function()`, som regulerer hastigheden, på basis af `tacho_counter` og `motor_speed`, ved at justere motorens pwm 1 duty-cycle procent op- eller ned (Således at motoren over tid tilnærmer sig den ønskede hastighed) Det er din opgave at implementere koden til `motor_speed_store()` og `motor_timer_function()`. Herunder ses funktionerne som skal kodes, samt hjælpefunktioner og globale variable: //File: `motorcontrol_driver.c`

```
unsigned double tacho_counter; // Tacho impulse counter value
unsigned int motor_speed; // Desired motor speed 0-100
void set_motor_pwm(unsigned int duty_cycle); // Helper function to set PWM 0-100
static irqreturn_t tacho_isr(int irq, void *dev_id) { ++tacho_counter; return IRQ_HANDLED; }
static void motor_timer_function (unsigned long arg) { /* Your Code Here */ }
static ssize_t motor_speed_store(struct device *dev, struct device_attribute *attr, const char *buf, size_t size) { /* Your Code Here */ }
```

Her er en simpel og funktionel implementering til **`motor_speed_store`** og **`motor_timer_function`**, der opfylder kravene:

1. **`motor_speed_store`**

- Læser ønsket hastighed (0-100) fra `buf`
- Opdaterer global `motor_speed`
- Returnerer antal læste bytes eller fejl

c

```
#include<linux/kernel.h>#include<linux/errno.h>static ssize_t motor_speed_store(struct device *dev, struct device_attribute *attr, const char *buf, size_t size)
{
    unsigned long val;
    int ret;

    ret = kstrtoul(buf, 10, &val);
    if (ret)
        return ret;
    if (val > 100)
        return -EINVAL;

    motor_speed = (unsigned int)val;
    return size;
}
```

2. motor_timer_function

- Kaldes hver 200 ms (fx vha. kernel timer eller hrtimer)
- Beregner tacho impulser pr. sekund: $\text{freq} = \frac{\text{tacho_counter}}{0.2} = \text{tacho_counter} \times 5$
- Sammenligner med `motor_speed`
- Justerer PWM duty-cycle op/ned med 1% for gradvist at nærme sig ønsket hastighed
- Nulstiller `tacho_counter` til 0 for næste interval
- Kalder `set_motor_pwm()` med ny duty-cycle

c

```
static unsigned int current_duty_cycle = 0;

static void motor_timer_function(unsigned long arg)
{
    unsigned int measured_speed;

    // Beregn impulser pr. sekund (200 ms * 5 = 1 s)
    measured_speed = (unsigned int)(tacho_counter * 5);

    // Nulstil tæller for næste periode
    tacho_counter = 0;

    if (measured_speed < motor_speed && current_duty_cycle < 100) {
        current_duty_cycle++;
        set_motor_pwm(current_duty_cycle);
    } elseif (measured_speed > motor_speed && current_duty_cycle > 0) {
        current_duty_cycle--;
        set_motor_pwm(current_duty_cycle);
    }

    // Timer skal genstartes udenfor her (fx via mod_timer i init)
}
```

Bemærkninger

- `motor_timer_function` forudsætter at den kaldes præcist hver 200 ms via kernel timer (fx `mod_timer`).
- `current_duty_cycle` holder den aktuelle PWM-værdi, start fx med 0 ved init.
- `tacho_counter` inkrementeres i ISR ved hver puls (atomisk opsamling anbefales ved multithread, men er uden for scope her).
- Justeringen ændrer PWM gradvist med 1% per kald.

Sig til hvis du ønsker forslag til opsætning af timer eller mere præcis atomic håndtering!

Du skal I denne opgave skrive read funktionen til en character driver som anvender et SPI interface. Systemets Linux device (Raspberry Pi) er forbundet til en 8-kanals A/D konverter ved hjælp af en SPI forbindelse. Interface between Linux device and A/C converter Protokollen imellem de to enheder er vist i figuren Diagram of SPI communication between Raspberry and external device En dataoverførsel starter ved, at Raspberry Pi sender en 8-bit værdi med kanal nummeret, herefter returnerer A/D konverteren den målte 8-bit værdi for den pågældende kanal. De enkelte kanaler skal kunne tilgås vha. 8 noder i /dev mappen med fælles majornummer og individuelle minor numre. Det er din opgave at implementere driverens read funktion. Opgaven opdeles i to dele: a) Lav en hjælpe funktion til læsning fra et SPI interface, `spi_read_byte`, som står for opbygning- og udveksling af SPI beskeder b) Lav en character driver read funktion, `my_spi_cdrv_read`, som læser fra SPI interfacet og overfører data user-space. Du skal tage udgangspunkt i kodeudsnittet nedenfor. Skriv din besvarelse ind i tekstfeltet.

```
/****** Device Driver Code *****/ /* Global variables */ struct
spi_device myspi_dev; /* Already initialized with a pointer to our spi device
*/ /* SPI Read byte helper function */ int spi_read_byte(struct spi_device*
sdev, u8 channel, u8 *data){ /* Implement the code needed here */ /* Error
handling not required */ } /* Character driver read function */ ssize_t
myspi_read(struct file *filep, char __user *ubuf, size_t count, loff_t *f_pos) {
/* Implement the code needed here */ /* Error handling not required */ }
/****** Code End *****/
```

Her er en løsning til begge dele, baseret på opgaveteksten:

a) `spi_read_byte` — Hjælpefunktion til SPI læsning

- Sender kanalnummeret som én byte TX
- Modtager data i RX-buffet (8 bit)
- Bruger `spi_sync_transfer()` med `struct spi_transfer` og `struct spi_message`

c

```
#include<linux/spi/spi.h>int spi_read_byte(struct spi_device* sdev, u8
channel, u8 *data)
{
    int ret;
    u8 tx_buf[1] = { channel };
    u8 rx_buf[1] = { 0 };
    struct spi_transfer transfer = {
        .tx_buf = tx_buf,
        .rx_buf = rx_buf,
        .len = 1,
    };
    struct spi_message msg;

    spi_message_init(&msg);
    spi_message_add_tail(&transfer, &msg);

    ret = spi_sync(sdev, &msg);
    if (ret == 0)
        *data = rx_buf[0];

    return ret;
}
```

b) **myspi_read** — Character driver read funktion

- Får **minor** nummer via **iminor(filep->f_inode)** (minor = kanal)
- Kalder **spi_read_byte()** med kanalnummer
- Overfører 1 byte data til user space via **copy_to_user**
- Returnerer antal bytes læst (her 1)

c

```
#include<linux/uaccess.h>#include<linux/fs.h>ssize_tmyspi_read(struct file
*filep, char __user *ubuf,
                        size_t count, loff_t *f_pos)
{
    u8 data;
    int minor;
    int ret;

    /* Begræns læsning til 1 byte */if (count < 1)
        return0;

    minor = iminor(filep->f_inode);

    /* Læs data fra SPI for denne kanal */
    ret = spi_read_byte(&myspi_dev, (u8)minor, &data);
    if (ret)
        return ret;

    /* Overfør til user space */if (copy_to_user(ubuf, &data, 1))
        return -EFAULT;

    return1;
}
```

Kommentar:

- `spi_read_byte()` håndterer SPI-kommunikation
- `myspi_read()` bruger filens minor nummer til at vælge kanal
- Læser altid 1 byte pr. read-kald
- Forenklet uden fejlhåndtering udover brugerplads-kopiering

Sig til, hvis du vil have det udvidet med f.eks. bufferhåndtering eller fejlkontrol!