

# Linux Treiber Workshop

## Eine Einführung in die Linux Treiber Programmierung

Johannes Roith

22.03.2025

- Embedded Software Entwickler
- Embedded Linux YouTube Channel
- [Meine Webseite](#) mit Links zu GitHub, Mastadon, LinkedIn, ...
- Ein Treiber von mir hat es in den Linux Kernel geschafft

- <https://www.buymeacoffee.com/johannes4linux>
- <https://paypal.me/johannes4linux>
- Super Thanks bei meinen YouTube Videos

# Agenda

- 1 Der Linux Kernel
- 2 Linux Kernel Programmierung auf dem Raspberry Pi
- 3 Der I2C Bus
- 4 Ein Linux I2C Treiber
- 5 Makefile zum Bauen des I2C Treibers
- 6 Module verwalten in einer Shell
- 7 I2C Geräte über das sysfs hinzufügen
- 8 PCF8574 IO Expander
- 9 Auf den I2C Bus Zugreifen
- 10 Erstellen von sysfs Einträgen

11 Ein richtiges Hello World Kernel Module

12 Das Makro `module_i2c_driver`

- Der Kernel eines Betriebssystems: hardwareabstrahierende Schicht
- Einheitliche Schnittstelle (API, Systemcalls) unabhängig von Rechnerarchitektur
- Aufgaben des Linux-Kernels:
  - Speicherverwaltung
  - Prozessverwaltung
  - Multitasking
  - Lastverteilung
  - Zugriff auf Hardware über Treiber
- Applikationen nutzen Systemcalls (open, close, read, write, ioctl, ...): benötigt keine genaue Kenntnis der Hardware
- Linux: modularer monolithischer Kernel mit nachladbaren Modulen

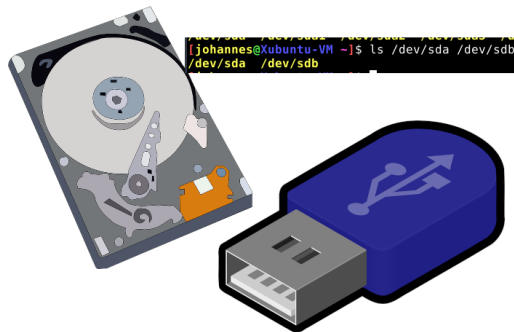
# Der Linux Kernel

```
Terminal -
File Edit View Terminal Tabs Help

CPU[ 3.4%] Tasks: 91, 161 thr: 1 running
Mem[ 786M/7.72G] Load average: 0.71 0.48 0.18
Swp[ 0K/3.81G] Uptime: 00:01:47

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ vCommand
1 root 20 0 98M 11592 8196 S 0.0 0.1 0:03.02 /sbin/init auto
837 root 20 0 373M 114M 54512 S 4.1 1.4 0:03.00 /usr/lib/xorg/X
674 root 20 0 1216M 30288 19148 S 0.0 0.4 0:01.19 /usr/lib/snapd/
1653 johannes 20 0 608M 57824 42600 S 0.7 0.7 0:00.96 xfce4-terminal
1524 johannes 39 19 704M 31820 20196 S 0.0 0.4 0:00.84 /usr/libexec/tr
1538 johannes 20 0 568M 180M 81096 S 0.0 1.3 0:00.81 xfwm4 --display
1822 johannes 20 0 11404 4688 3600 R 0.7 0.1 0:00.66 htop
1591 johannes 20 0 543M 60056 39808 S 0.0 0.7 0:00.64 xfdesktop --dis
1178 root 20 0 1216M 30288 19148 S 0.0 0.4 0:00.50 /usr/lib/snapd/
1590 johannes 20 0 598M 53572 40232 S 0.0 0.7 0:00.50 /usr/lib/x86_64
761 root 20 0 1108M 41936 30172 S 0.0 0.5 0:00.48 /usr/bin/contai
911 root 20 0 1212M 74464 51064 S 0.0 0.9 0:00.43 /usr/bin/docker
348 root 19 -1 79364 46492 45248 S 0.0 0.6 0:00.40 /lib/systemd/sy
1220 johannes 9 -11 612M 24476 17908 S 0.0 0.3 0:00.39 /usr/bin/pulsea
1216 johannes 20 0 460M 79224 60272 S 0.0 1.0 0:00.34 xfce4-session
1581 johannes 20 0 487M 37608 27388 S 0.0 0.5 0:00.33 xfce4-panel --d
1110 root 20 0 373M 114M 54512 S 1.4 1.4 0:00.26 /usr/lib/xorg/X

F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit
```



```
/dev/sda /dev/sda1 /dev/sda2 /dev/sda3 /dev/sda4 /dev/sda5 /dev/sda6 /dev/sda7 /dev/sda8 /dev/sda9 /dev/sda10 /dev/sda11 /dev/sda12 /dev/sdb /dev/sdb1 /dev/sdb2 /dev/sdb3 /dev/sdb4 /dev/sdb5 /dev/sdb6 /dev/sdb7 /dev/sdb8 /dev/sdb9 /dev/sdb10 /dev/sdb11 /dev/sdb12
```

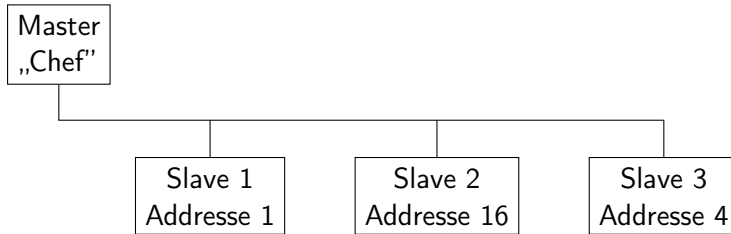
# Linux Kernel Programmierung auf dem Raspberry Pi

- Pakete aktualisieren mit: `sudo apt update && sudo apt upgrade -y`
- Kernel Headers installieren: `sudo apt install -y raspberrypi-kernel-headers`
- Build Werkzeuge, wie gcc, make, ... installieren: `sudo apt install -y build-essential`
- Reboot, um ggf. neuen Kernel zu laden: `sudo reboot`



- Einfacher Zweidrahtbus
- Datenleitung: *SDA*
- Taktleitung: *SCK*
- Frequenzen: 100kbit/s, 400kbit/s, 1Mbit/s
- Pull-Up Widerstand bei Leitungen notwendig

# Der I2C Bus



# Ein Linux I2C Treiber

## Header und kompatible Geräte

```
/* Benötigter Header */  
#include <linux/i2c.h>  
  
/* Benenne alle kompatiblen Geräte */  
static struct i2c_device_id my_ids[] = {  
    {"my_dev"},  
    {} /* leeres Element signalisiert das Ende der Liste */  
};  
MODULE_DEVICE_TABLE(i2c, my_ids);
```

# Ein Linux I2C Treiber

## Probe- und Remove Funktionen

```
/* Funktion wird aufgerufen, wenn ein kompatibles I2C Gerät hinzugefügt wird
   */
static int my_probe(struct i2c_client *client, const struct i2c_device_id *
    id)
{
    printk("Hallo vom I2C Slave mit der Adresse: 0x%x\n", client->addr);
    return 0;
}

/* Funktion wird aufgerufen, wenn ein kompatibles I2C Gerät entfernt wird */
static void my_remove(struct i2c_client *client)
{
    printk("Bye, bye, I2C\n");
}
```

# Ein Linux I2C Treiber

## Treiber anlegen

```
/* Fasse kompatible Geräte, Probe- und Remove-Funktionen in Treiber zusammen
 */
static struct i2c_driver my_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .id_table = my_ids,
    .driver = {
        .name = "my-i2c-driver",
    }
};

/* Registriere Treiber */
module_i2c_driver(my_driver);
/* Informationen über Treiber */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes Roith");
MODULE_DESCRIPTION("Ein Hello World I2C Teiber");
```

# Makefile zum Bauen des I2C Treibers

```
# Kernel Header Makefile kompiliert i2c_hello.c automatisch zu i2c_hello.o
file
obj-m += i2c_hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

# Module verwalten in einer Shell

- Befehle als root ausführen, z.B. `sudo insmod dmesg`
- `lsmod` zeigt die geladenen Module an
- `dmesg` zeigt die Kernel Logs an
- `insmod <modulname>` lädt das Modul `<modulname>` in den Kernel
- `rmmod <modulname>` entfernt das Modul `<modulname>` aus den Kernel
- `modprobe <modulname>` lädt das Modul `<modulname>` inklusive seiner Abhängigkeiten in den Kernel
- `modinfo <modulname>` zeigt die Meta-Daten (Autor, Lizenz, Beschreibung, ...) des Modul `<modulname>` an

# I2C Geräte über das sysfs hinzufügen

```
# In I2C-2 Geräte Ordner wechseln
```

```
cd /sys/bus/i2c/devices/i2c-1
```

```
# Geräte my_dev mit I2C Adresse 0x12 hinzufügen
```

```
echo "my_dev 0x12" | sudo tee new_device
```

```
# Geräte mit I2C Adresse 0x12 entfernen
```

```
echo "0x12" | sudo tee delete_device
```



- Implementiere das Kernelmodul auf dem Raspberry Pi. Das kompatible Gerät für den Treiber soll *rgb\_brd* heißen.
- Baue das Modul über ein Makefile
- Lade das Kernelmodul
- Prüfe ob das Modul geladen ist
- Füge ein I2C Gerät hinzu
- Prüfe das Kernellog
- Entlade das Modul

- Schreibzugriff setzt Ausgänge P0 - P7
- Lesezugriff liest Wert P0 - P7
- Taster angeschlossen an P0
- Bei Eingang: Setzte Port auf 1, Taster zieht Eingang auf GND, d.h. wird eine 1 gelesen ist der Taster nicht gedrückt, wird eine 0 gelesen ist er gedrückt.
- LED Grün angeschlossen an P1, LED Rot an P2, LED Blau an P3
- Ausgang auf 0 gesetzt: LED ist an
- Ausgang auf 1 gesetzt: LED ist aus

Bit:	0	1	2	3	4	5	6	7
Wert für:	P0	P1	P2	P3	P4	P5	P6	P7

# Auf den I2C Bus Zugreifen

```
struct i2c_client *my_client;
```

Die Struktur vom Typ `struct i2c_client` wird verwendet, um ein I2C Gerät im Kernel zu verwalten. Über den Zeiger `my_client` können wir anschließend auf das Gerät zugreifen, z.B. um Daten zu lesen oder zu schreiben.

```
s32 i2c_smbus_read_byte(struct i2c_client *my_client);
```

Liest ein Byte vom I2C Gerät `my_client`. Im Fehlerfall wird ein negativer Fehlercode zurückgegeben, ansonsten der gelesene Wert.

```
s32 i2c_smbus_write_byte(struct i2c_client *my_client, u8 value);
```

Schreibt das Byte `value` zum I2C Gerät `my_client`. Im Fehlerfall wird ein negativer Fehlercode zurückgegeben, ansonsten eine 0.

- Lasse die RGB LED eine Farbe deiner Wahl anzeigen, indem du in der Probe Funktion P1-P3 des PCF8574 beschreibst
- Schalte die RGB LED in der Remove Funktion wieder ab
- Kompiliere und teste das Kernel Modul
- Zusatzaufgabe: Lies in der Probe Funktion den Wert des Tasters an P0 ein und gib ihn im Kernel Log aus.

- *sysfs*: Virtuelles Dateisystem
- Darstellung und Verwaltung von *Kernel Objekten* (`kobject`)
- Ermöglicht Interaktion mit Treibern
- *Kernel Objekt*: Ordner in *sysfs*
- *Kernel Objekt* bietet Möglichkeit Attribute (dargestellt als Dateien) anzulegen über die wir mit den Treiber vom Userspace aus kommunizieren können
- Vorgehen: Show und Store Funktionen Implementieren, Attribute erstellen, Kernel Objekt erstellen, *sysfs* Datei mit Kernel Objekt verknüpfen

# Show und Store Funktionen und Attribute

```
/* Benötigter Header */
#include <linux/kobject.h>

static ssize_t mydev_show(struct kobject *kobj, struct kobj_attribute *attr,
                          char *buffer)
{
    return sprintf(buffer, "Hello world!\n");
}

static ssize_t mydev_store(struct kobject *kobj, struct kobj_attribute *attr
                          , const char *buffer, size_t count)
{
    printk("I got %s\n", buffer);
    return count;
}

static struct kobj_attribute mydev_attr = __ATTR(my_attr, 0660, mydev_show,
          mydev_store);
```

# kobject erstellen und mit Attribute verknüpfen

```
struct kobject * my_kobj */
/* in init oder probe Funktion */
int status;

my_kobj = kobject_create_and_add("my_kobj", my_kobj);
if (!my_kobj) {
    printk("Error creating kernel object\n");
    return -ENOMEM;
}

status = sysfs_create_file(my_kobj, &mydev_attr.attr);
if (status) {
    printk("Error creating /sys/my_kobj/my_attr\n");
    return status;
}
```

# kobject und Attribute löschen

```
/* in exit oder remove Funktion */  
sysfs_remove_file(my_kobj, &mydev_attr.attr);  
kobject_put(my_kobj);
```



- Erstelle im sysfs einen Ordner *rgb\_led*
- Erstelle im Ordner *rgb\_led* eine Datei *led*
- Überlade die store Funktion, sodass man die drei LEDs ansteuern kann. Wird der String 011 in die Datei geschrieben, wird die rote LED auf 0, die grüne auf 1 und die blaue auf 1 gesetzt
- Zusatzaufgabe: Erstelle im Ordner *rgb\_led* eine Datei *taster* über dessen show Funktion der Wert des Tasters ausgelesen werden kann.

# Ein richtiges Hello World Kernel Module

```
#include <linux/module.h>
#include <linux/init.h>
int __init my_init(void)
{
    printk("hello_kernel - Das Unheil nimmt seinen Lauf...\n");
    return 0;
}
void __exit my_exit(void)
{
    printk("hello_kernel - Da ist der Kernel aber nochmal glimpflich
        davongekommen!\n");
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes Roith");
MODULE_DESCRIPTION("A simple hello world LKM");
module_init(my_init);
module_exit(my_exit);
```

# Das Makro module\_i2c\_driver

Das Makro `module_i2c_driver(my_driver)` erzeugt in etwa folgenden Code:

```
static int i2c_driver_init(void)
{
    return i2c_add_driver(&my_driver);
}
module_init(i2c_driver_init);

static void i2c_driver_exit(void)
{
    i2c_del_driver(&my_driver);
}
module_exit(i2c_driver_exit);
```