

# Linux Treiber Workshop

## Eine Einführung in die Linux Treiber Programmierung

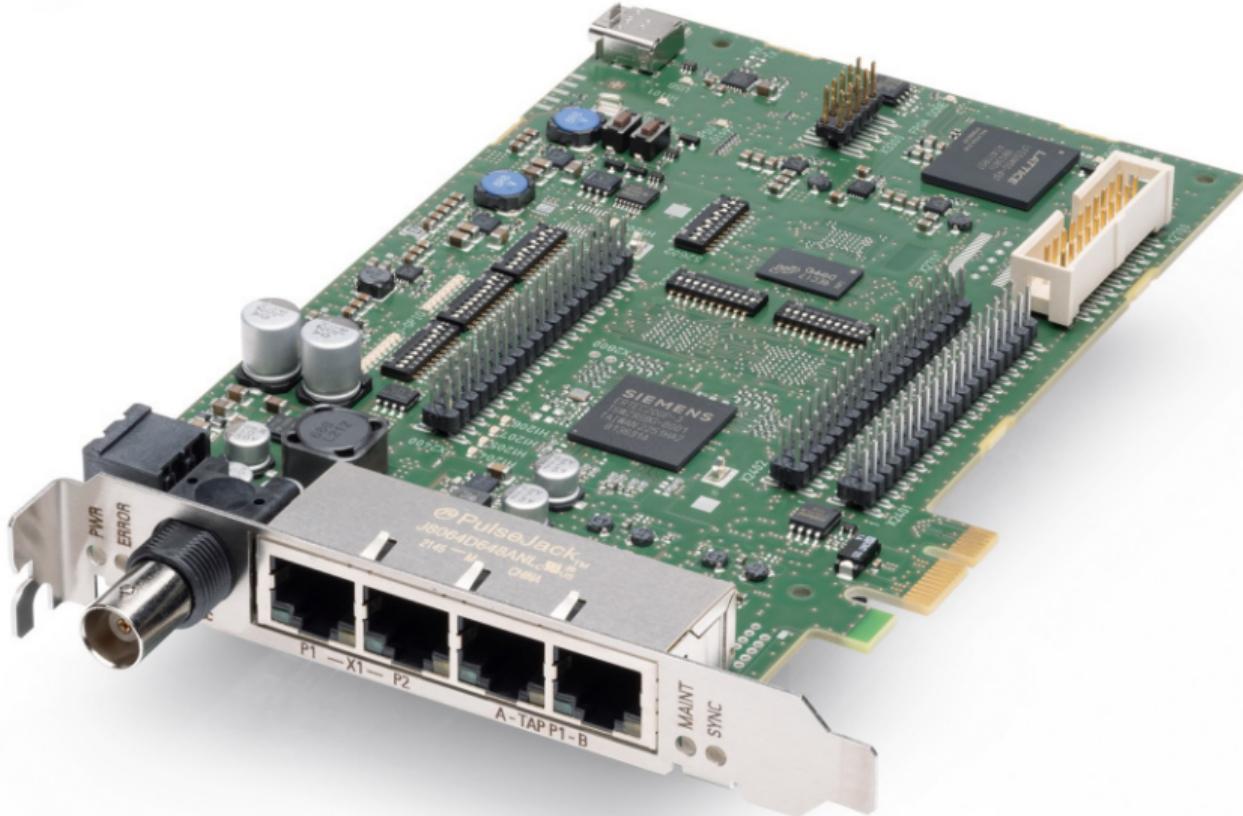
Johannes Roith

05.12.2025



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

# Über meine Arbeit



# Agenda

- 1 Der Linux Kernel
- 2 Entwicklungsumgebung für Linux Treiber Programmierung
- 3 Ein Hello World Kernel Module
- 4 Makefile zum Bauen des Kernel Moduls
- 5 Module verwalten in einer Shell

# Agenda

- 6 Der I2C Bus
- 7 Ein Linux I2C Treiber
- 8 I2C Geräte über das sysfs hinzufügen
- 9 PCF8574 IO Expander
- 10 Auf den I2C Bus Zugreifen
- 11 Erstellen von sysfs Einträgen
- 12 Geräte Hinzufügen über den Device Tree

# Material für den Workshop



<https://github.com/Johannes4Linux/ese25>

- 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
- Ziel des Workshops: Implementierung eines einfachen nachladbaren Kernel Moduls als Treiber für die Aufsteckplatine

# 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[OK/3.81G] Uptime: 00:01:47

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
 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 100M 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 60856 39888 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 17988 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
```

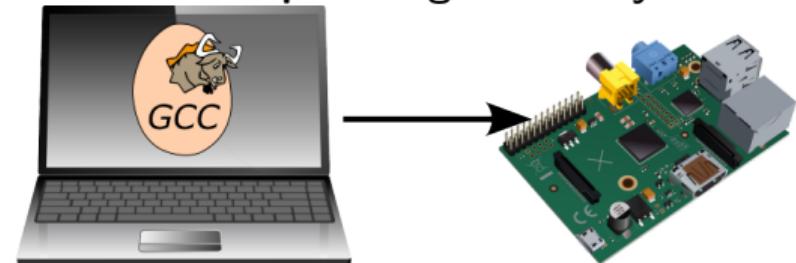


# Entwicklungsumgebung für Linux Treiber Programmierung

Entwicklung auf Zielsystem



Cross-Kompilierung für Zielsystem



# Linux Treiber 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`

# Ein 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("Ein einfaches Linux Kernel Modul");
module_init(my_init);
module_exit(my_exit);
```

# Makefile zum Bauen des Kernel Moduls

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

# Ergebnis des Builds: Ladbares Kernel Object hello.ko
```

# Module verwalten in einer Shell

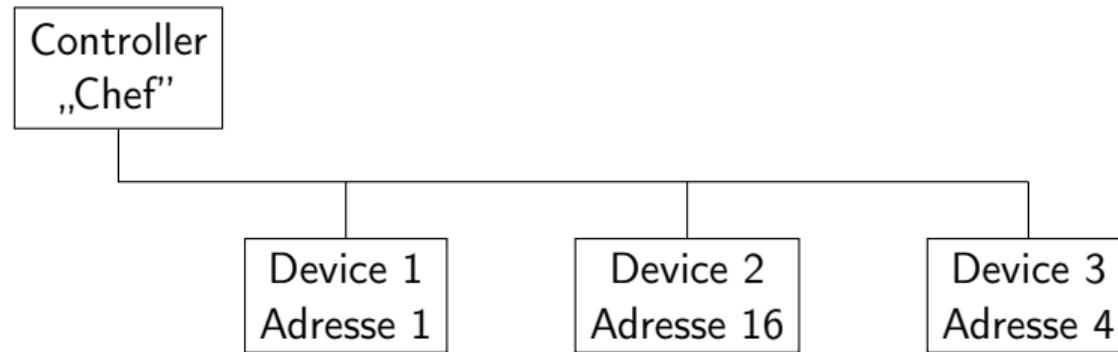
- `lsmod` zeigt die geladenen Module an
- `dmesg` zeigt die Kernel Logs an
- `insmod hello.ko` lädt das Modul `hello.ko` in den Kernel
- `rmmmod hello.ko` entfernt das Modul `hello.ko` aus den Kernel
- `modinfo ./hello.ko` zeigt die Meta-Daten (Autor, Lizenz, Beschreibung, ...) des Modul `hello.ko` an
- `modprobe industrialio` lädt das Modul `industrialio` inklusive seiner Abhängigkeiten in den Kernel

# Aufgabe

- Implementieren Sie das Kernelmodul `hello` auf dem Raspberry Pi im Ordner `aufgabe_1`.
- Bauen Sie das Modul über ein Makefile.
- Laden Sie das Kernelmodul.
- Prüfen Sie das Kernellog und ob das Modul geladen ist.
- Entladen Sie das Modul.

- 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)
{
    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

Bündeln der Treiber Informationen

```
/* 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",
    }
};
```

# Ein Linux I2C Treiber

## Registrieren des Treibers

```
static int __init my_driver_init(void)
{
    return i2c_add_driver(&my_driver);
}
static void my_driver_exit(void)
{
    i2c_del_driver(&my_driver);
}
module_exit(my_driver_exit);
module_init(my_driver_init);
```

oder alternativ:

```
module_i2c_driver(my_driver);
```

# I2C Geräte über das sysfs hinzufügen

```
# In I2C-1 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_devices

# Geräte mit I2C Adresse 0x12 entfernen
echo "0x12" | sudo tee delete_device
```

# Aufgabe

- Erweitern Sie das Kernelmodul `rgb_brd.c` auf dem Raspberry Pi im Ordner `aufgabe_2` wie folgt:
  - Das kompatible Gerät soll `rgb_brd` heißen
  - In der Probe Funktion soll die I2C Adresse ins Kernel Log geschrieben werden
  - Beim Entladen soll eine Nachricht ins Kernel Log geschrieben werden
  - Fügen Sie die fehlenden Meta-Daten zum Modul ein
- Bauen Sie das Modul über ein Makefile
- Laden Sie das Kernelmodul
- Prüfen Sie, ob das Modul geladen ist
- Fügen Sie ein I2C Gerät hinzu
- Prüfen Sie das Kernellog
- Entladen Sie das Modul

# PCF8574 IO Expander

- Schreibzugriff setzt Ausgänge P0 - P7
- Lesezugriff liest Wert P0 - P7
- Taster angeschlossen an P0
- Bei Eingang: Setze 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 Rot angeschlossen an P1, LED Grün 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.

# Aufgabe

- Kopieren Sie die Datei `aufgabe_2/rgb_brd.c` nach `aufgabe_3` und erweitern Sie den Treiber.
- Lassen Sie die RGB LED eine Farbe Ihrer Wahl anzeigen, indem Sie in der Probe Funktion P1-P3 des PCF8574 beschreiben.
- Schalten Sie die RGB LED in der Remove Funktion wieder ab.
- Kompilieren und testen Sie das Kernel Modul.
- Zusatzaufgabe: Lesen Sie in der Probe Funktion den Wert des Tasters an P0 ein und geben Sie ihn im Kernel Log aus.

# Erstellen von sysfs Einträgen

- *sysfs*: Virtuelles Dateisystem dient als Schnittstelle zwischen Kernel und Userspace
- 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);
```

# Aufgabe

Im Ordner `aufgabe_4` befindet sich ein Beispieltreiber, der ein Kernel Objekt und ein Attribut anlegt. Erweitern Sie den Treiber wie folgt:

- Das Kernel Attribut soll den Namen `rgb_brd` erhalten.
- Verknüpfen Sie das Objekt `rgb_brd` mit dem Attribut `led`.
- Überladen Sie die `store` Funktion für das Attribut `led`, sodass man die drei LEDs ansteuern kann. Wird der String `011` in die Datei geschrieben, wird die rote LED ausgeschaltet, die grüne und die blaue eingeschaltet.
- Zusatzaufgabe: Erstellen Sie ein weiteres Attribut `taster`, über dessen `show` Funktion der Wert des Tasters ausgelesen werden kann.

# Geräte Hinzufügen über den Device Tree

## Der Device Tree

- ARM/Open RISC V Systeme haben keine automatische Hardwareerkennung wie z.B. das BIOS bei x86 Systemen
- Der Linux Kernel benötigt Informationen, welche Geräte verfügbar sind
- Device Tree liefert diese Informationen
- Device Tree fasst die verfügbaren Geräte in einer Baumstruktur zusammen
- Die Device Tree Sourcen (dts) und Device Tree Source Includes (dtsi) muss kompiliert werden (dtb: Device Tree Binary)
- Device Tree verfügbar unter /sys/firmware/devicetree/base
- Umwandeln in lesbare Form: `dtc -I fs -O dts -s /sys/firmware/devicetree/base > dt.dts`
- Device Tree kann auch über Overlays erweitert werden. Vorteil: nicht der ganze Device Tree muss neu kompiliert werden, sollte ein Gerät hinzugefügt werden

# Geräte Hinzufügen über den Device Tree

## Device Tree Overlay für I2C Gerät

```
/dts-v1;
/plugin/;
/ {
    fragment@0 {
        target = <&i2c1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <0>;
            my_dev: my_dev@12 {
                compatible = "brightlight,mydev";
                status = "okay";
                reg = <0x12>;
            };
        };
    };
};
```

Kompilieren des Overlays mit `dtc -@ -I dts -O dtb -o testoverlay.dtbo testoverlay.dts`



# Geräte Hinzufügen über den Device Tree

## Erweiterung des I2C Treibers

```
/* Device Tree kompatible Geräte benennen */
static struct of_device_id my_driver_of_ids[] = {
    { .compatible = "brightlight,my_dev", },
    { /* sentinel */ }
};

MODULE_DEVICE_TABLE(of, my_driver_ids);

/* OF IDs zum Treiber Struct hinzufügen */
static struct i2c_driver my_driver = {
    ...
    .driver = {
        .name = "my-i2c-driver",
        .of_match_table = my_driver_of_ids,
    }
};
```

# Aufgabe

- Kopieren Sie die Datei `aufgabe_4/rgb_brd.c` nach `aufgabe_5` und erweitern Sie den Treiber.
- Erstellen Sie einen Device Tree Overlay für das `rgb_brd`.
- Kompilieren und laden Sie den Device Tree Overlay.
- Erweitern Sie den I2C Treiber, damit Geräte auch über den Device Tree hinzugefügt werden können.
- Bauen und testen Sie den Treiber mit Device Tree Unterstützung.

# (Literatur) Empfehlungen

- Madieu: Linux Device Driver Development (ISBN: 1803240067)
- [Linux Device Driver 3rd Edition](#)
- Quade: Linux Treiber entwickeln (ISBN-10: 3988890383)
- [Microconsult Kernel-Treiberentwicklung](#)
- [Microconsult Embedded-Echtzeit-Linux](#)