Linux Treiber Workshop

Eine Einführung in die Linux Treiber Programmierung

Johannes Roith

16.03.2024

Agenda

1. Der Linux Kernel
2. Linux Kernel Programmierung auf dem Raspberry Pi
3. Ein erstes Hello-World Kernelmodul
4. Makefile zum Kompilieren des Moduls
5. Module verwalten mit der Bash
6. GPIOs ansteuern
7. Gerätenummern und zeichenorientierte Geräte

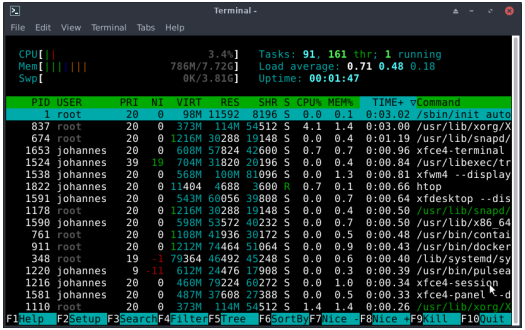
Backup

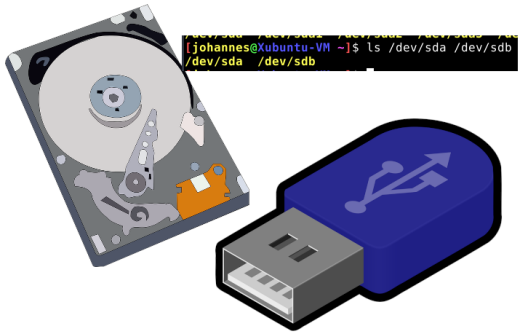
8. GPIO Interrupts
9. ioctl
10. Gerätedatei mit udev erstellen
11. Der Device Tree

Der Linux Kernel

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

Der Linux Kernel





- 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 erstes Hello-World Kernelmodul

```
#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);
```

Makefile zum Kompilieren des Moduls

```
obj-m += hello_kernel.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
```

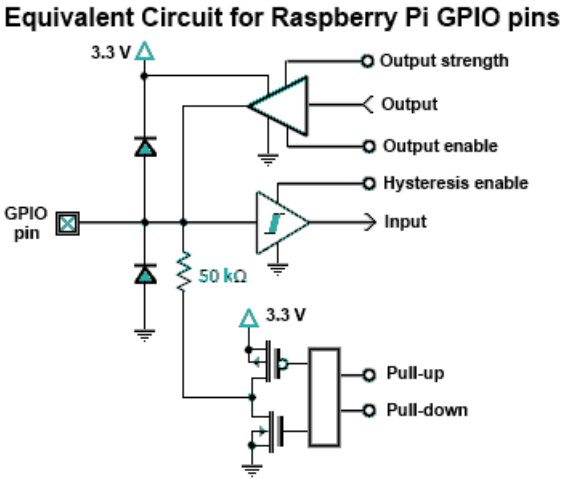
Module verwalten mit der Bash

- `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

Aufgabe

- Implementiere das Kernelmodul auf dem Raspberry Pi
- Baue das Modul über ein Makefile
- Lade das Kernelmodul
- Prüfe das Kernellog und ob das Modul geladen ist
- Entlade das Modul

GPIOs ansteuern



GPIOs ansteuern

```
int gpio_request(gpio_nr, "gpio-label")
```

Fordert den GPIO Pin `gpio_nr` an. Das Label erscheint in `gpioinfo`. Bei Erfolg gibt die Funktion eine 0 zurück, ansonsten einen Fehlercode. Benötigter Header: `<linux/gpio.h>`

```
int gpio_direction_input(gpio_nr)
int gpio_direction_output(gpio_nr, value)
```

Konfiguriert den GPIO Pin `gpio_nr` als Ein- oder Ausgang. Bei der Konfiguration als Ausgang muss der Initialzustand des Pins übergeben werden. Bei Erfolg gibt die Funktion eine 0 zurück, ansonsten einen Fehlercode.

GPIOs ansteuern

`void gpio_set_value(gpio_nr, value)`

Setzt den GPIO Pin `gpio_nr` auf den Ausgabewert `value`. Der Pin muss zuvor als Ausgang konfiguriert worden sein.

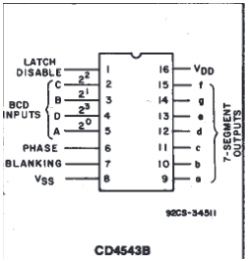
`int gpio_get_value(gpio_nr)`

Liest den aktuellen Eingabewert von GPIO Pin `gpio_nr` ein. Der Pin muss zuvor als Eingang konfiguriert worden sein. Rückgabewert ist der Zustand des IO Pins.

`void gpio_free(gpio_nr)`

Gibt den GPIO Pin `gpio_nr` frei. Anschließend kann er von anderen Kernel-Modulen oder aus dem Userspace benutzt werden.

7 Segmentanzeigen Dekoder



A	B	C	D	Segmentanzeige
0	0	0	0	0
1	0	0	0	1
0	1	0	0	2
1	1	0	0	3
0	0	1	0	4
1	0	1	0	5
0	1	1	0	6
1	1	1	0	7
0	0	0	1	8
1	0	0	1	9
alle anderen Kombinationen				leer

Aufgabe

- Anschlüsse: A: 11, B: 9, C: 25, D: 8
- In der Init Funktion sollen nun die vier GPIOs für den 7 Segmentanzeigen-Dekoder initialisiert werden.
- Anschließend soll eine Zahl von 0 bis 9 angzeigt werden.
- In der Exit Funktion werden die GPIOs wieder freigegeben
- Baue und Teste das Kernelmodul
- Zusatzaufgabe: Initialisiere Pin 24 als Eingang

Gerätenummern und zeichenorientierte Geräte

```
user@workshop:~ $ ls -l /dev/gpiochip0 /dev/mmcblk0p1
crw-rw---- 1 root gpio 254, 0 Nov 26 10:55 /dev/gpiochip0
brw-rw---- 1 root disk 179, 1 Nov 26 10:55 /dev/mmcblk0p1
user@workshop:~ $ cat /proc/devices
Character devices:
  ...
  254 gpiochip
  ...

Block devices:
  ...
  179 mmc
  ...
user@workshop:~ $ cat mydriver.c
...
static int __init ModuleInit(void) {
        retval = register_chrdev(64, "label", &fops);
...
```

Benötigter Header: `<linux/fs.h>`

```
struct file_operations {
        struct module *owner;
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
             *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *); /* close */
        ...
};
```

Struct enthält Zeiger zu Callback Funktionen. Es müssen nicht alle Funktionen implementiert werden.

```
ssize_t seg_write(struct file *file, const char __user *buf, size_t cnt, loff_t * off)
```

- `file`: Informationen zur geöffneten Datei, z.B. Gerätenummer (Major und Minor)
- `buf`: Textbuffer mit zu schreibenden Daten (`__user` gibt an, dass es ein Pointer aus dem Userspace ist, der in den Kernelspace übertragen werden sollte)
- `cnt`: Größe des Textbuffers in Byte
- `off`: Pointer mit Offset in Datei, kann nach Schreiben um die Anzahl der geschriebenen Bytes erhöht werden

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)
```

Belegt die Major-Gerätenummer `major`. Unter `/proc/devices` wird ein Eintrag mit der Major-Gerätenummer und dem Namen `name` erstellt. `fops` gibt die verfügbaren Dateioperationen an. Bei Erfolg gibt die Funktion eine 0 zurück, ansonsten einen Fehlercode.

```
int unregister_chrdev(unsigned int major, const char *name)
```

Gibt die Major-Gerätenummer `major` mit den Namen `name` frei. Bei Erfolg gibt die Funktion eine 0 zurück, ansonsten einen Fehlercode.

`copy_from_user(void *dst, const void __user *src, unsigned long len)`

Kopiert `len` Bytes aus den Userspace Buffer `*src` in den Kernelspace Buffer `*dst`.

`copy_to_user(void __user *dst, const void *src, unsigned long len)`

Kopiert `len` Bytes aus den Kernelspace Buffer `*src` in den Userspace Buffer `*dst`.

`sudo mknod /dev/seg c 64 0`

Erstellt die zeichenorientierte Gerätedatei `/dev/seg`. Der Gerätedatei wird die Major-Gerätenummer 64 und die Minor-Gerätenummer 0 zugewiesen.

`echo 7 > /dev/seg`

Schreibt den String `"7\n\0"` in die Gerätedatei `/dev/seg`. Je nach Dateiberechtigung muss der Befehl als Root ausgeführt werden.

`cat /dev/seg`

Liest die Gerätedatei so lange, bis das Ende der Datei (read gibt 0 zurück) erreicht wird. Je nach Dateiberechtigung muss der Befehl als Root ausgeführt werden.

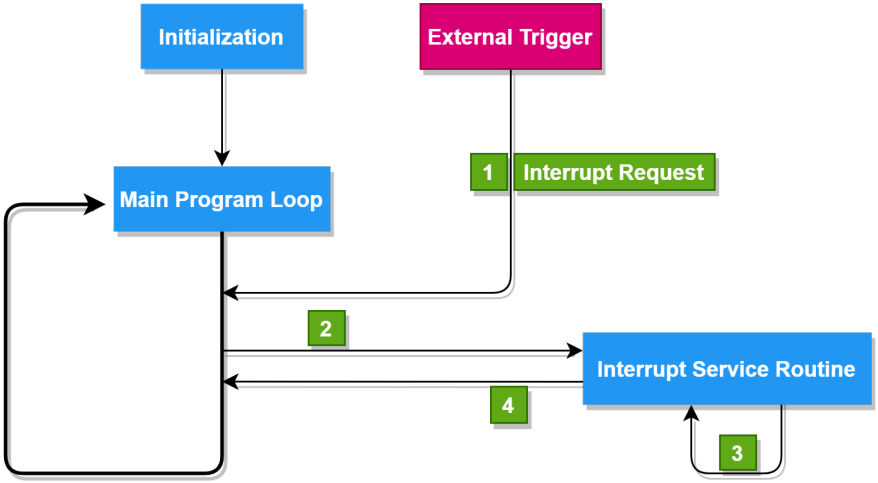
`head -n 1 /dev/seg`

Liest die Gerätedatei so lange, bis ein `'\n'` Newline Character erreicht wird. Je nach Dateiberechtigung muss der Befehl als Root ausgeführt werden.

Aufgabe

- Erstelle in der Init-Funktion des Treibers ein Character-Device und verknüpfe es mit der Major Geräte Nummer 64
- Gib die Resourcen für das Character-Device sowie die Gerätenummer in der Exit-Funktion wieder frei.
- Implementiere den write-Callback, sodass eine Zahl von 0-9 an der 7 Segmentanzeige angezeigt wird.
- Teste den Treiber
- Zusatzaufgabe: Implementiere den read-Callback, sodass der aktuelle Zustand des Tasters eingelesen werden kann.

GPIO Interrupt



GPIO Interrupt

```
int gpio_to_irq(int gpio_nr)
```

Gibt die Interrupt Nummer für den GPIO `gpio_nr` zurück.

```
int request_irq(int irq_nr, irq_handler_t handler, unsigned long flags, const char *name,
void *dev_data)
```

Aktiviert den Interrupt `irq_nr`. Über die `flags` kann angegeben werden, wann der Interrupt ausgelöst wird (z.B. `IRQF_TRIGGER_RISING` für steigende Flanke). Bei aktivem IRQ wird die `handler` Funktion aufgerufen. Als Parameter wird die IRQ Nummer und falls gesetzt, die `dev_data` übergeben. Der `name` taucht in `/proc/interrupts` auf.

GPIO Interrupt

Beispiel für ISR

```
irqreturn_t my_irq_handler(int irq_nr, void *data)
{
        printk("IRQ %d aktiv\n", irq_nr);
        /* Tue irgendwas */
        if(/* IRQ nicht behandelt */)
                return IRQ_NONE;
        else /* IRQ Behandelt */
                return IRQ_HANDLED;
}
```

ioctl

- input/output control
- Systemaufruf in Unix für spezifische Steuerbefehle, die nicht über read/write gesetzt werden können
- Befehle und Parameter können eigenständig festgelegt werden

```c
/* Datei cmd.h */

#define SET_ANSWER _IOW('a', 'a', int32_t *)
#define GET_ANSWER _IOR('a', 'b', int32_t *)

/* Alternativ ginge auch
 * #define SET_ANSWER 0x1
 * #define GET_ANSWER 0x2
 */
```

ioctl

ioctl im Treiber als Callback

```
#include "cmd.h"
int32_t answer;

static long int my_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
        switch(cmd) {
                case SET_ANSWER:
                        return copy_from_user(&answer, (int32_t *) arg, 4);
                case GET_ANSWER:
                        return copy_to_user((int32_t *) arg, &answer, 4);
                default:
                        return -EINVAL;
        }
}
struct file_operations fops = {
        .unlocked_ioctl = my_ioctl
};
```

```c
#include "cmd.h"
int32_t answer;

int fd = open(DEVFILE, O_RDWR);

ioctl(fd, SET_ANSWER, &answer);
ioctl(fd, GET_ANSWER, &answer);

printf("Die Antwort auf alles ist %d\n", answer);
```

Gerätedatei mit udev erstellen

Gerätenummer reservieren und Characterdevice erstellen

```
int register_chrdev_region(dev_t devnr, unsigned cnt, const char *name)
```

Registriert `cnt` Gerätenummern ab der Gerätenummer `devnr`. In `/proc/devices` wird ein Eintrag mit `MAJOR(devnr)` und den Namen `name` erstellt. Benötigter Header: `linux/fs.h`

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

Initialisiert das Characterdevice `cdev` mit den Dateioperationen in `fops`. Benötigter Header: `linux/cdev.h`

```
int cdev_add(struct cdev *cdev, dev_t devnr, unsigned cnt)
```

Fügt das Characterdevice `cdev` zum System hinzu und verknüpft es mit `cnt` Gertäenummern ab `devnr`.

Gerätedatei mit udev erstellen

Klasse und Gerät erstellen

```
struct class * class_create(struct module *owner, const char *name)
```

Legt eine neue Klasse unter `/sys/class/` mit den Namen `name` an. Rückgabezeiger mit `IS_ERR` auf Gültigkeit prüfen!

```
struct device *device_create(struct class *class, struct device *parent, dev_t devnr, void *drvdata, const char *name, ...)
```

Legt ein neues Gerät in der Klasse `class` an und verknüpft es mit der Gerätenummer `devnr`. Der Name `name` taucht dann in `/dev/` als Gerätedatei auf. Rückgabezeiger mit `IS_ERR` auf Gültigkeit prüfen!

Gerätedatei mit udev erstellen

Aufräumen der Gerätenummer, der Klasse und des Geräts

`void void device_destroy(struct class *class, dev_t devnr)`

Zerstört das Gerät in der Klasse `class`, das mit der Gerätenummer `devnr` verknüpft ist.

`void class_destroy(struct class *class)`

Zerstört die Klasse `class`.

`void unregister_chrdev_region(dev_t devnr, unsigned cnt)`

Gibt `cnt` Gerätenummern ab `devnr` frei.

- 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

```
/dts-v1/;
/plugin/;
/ {
        fragment@0 {
                target-path = "/";
                __overlay__ {
                        my_device {
                                compatible = "brightlight,mydev",
                                a-gpio = <&gpio 11 0>;
                                status = "okay";
                        };
                };
        };
};
```

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

Probe- und Remove-Funktion

```
#include <linux/property.h>
#include <linux/of_device.h>

static int foo_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        ...
        return 0;
}

static int foo_remove(struct platform_device *pdev)
{
        ...
}
```

Treiber für Device Tree Gerät

Kompatible Geräte und Treiber-struct

```
static struct of_device_id foo_ids[] = {
        { .compatible = "brightlight,mydev" },
        {},
};
MODULE_DEVICE_TABLE(of, foo_ids);

static struct platform_driver foo_driver = {
        .probe = foo_probe,
        .remove = foo_remove,
        .driver = {
                .name = "foo",
                .of_match_table = foo_ids,
        }
};
```

Treiber für Device Tree Gerät

Treiber registrieren

```
int __init foo_init(void)
{
        return platform_driver_register(&foo_driver);
}
void __exit foo_exit(void)
{
        platform_driver_unregister(&foo_driver);
}
module_init(foo_init);
module_exit(foo_exit);
```

oder:

```
module_platform_driver(foo_driver);
```

```
bool device_property_present(const struct device *dev, const char *propname)
```

Prüft, ob in `pdev->dev` die Property `propname` existiert. Falls es die Property gibt, wird eine `1`, ansonsten `0` zurückgegeben.

```
struct gpio_desc *gpiod_get(struct device *dev, const char *label, enum gpiod_flags flags
```

Importiert und initialisiert den GPIO mit den Label `<label>-gpio` aus dem Device Tree Eintrag für das Gerät `dev = pdev->dev`. Die `flags` geben die Richtung an. `GPIO_IN` für Konfiguration als Eingang, `GPIO_OUT_LOW` für Ausgang. Rückgabezeiger mit `IS_ERR` auf Gültigkeit prüfen!
Benötigter Header: `<linux/gpio/consumer.h>`.

```
void gpiod_set_value(struct gpio_desc *gpio, int value)
```

Setzt Ausgangswert des GPIOs `gpio` auf den Wert `value`.

```
int gpiod_get_value(struct gpio_desc *gpio)
```

Gibt den Eingangswert des GPIOs `gpio` zurück.

```
void gpiod_put(struct gpio_desc *gpio)
```

Gibt den GPIO `gpio` wieder frei.