Linux Treiber Workshop

Eine Einführung in die Linux Treiber Programmierung

Johannes Roith

15.09.2023

Agenda

1. Der Linux Kernel
2. Unterschiede zwischen Userspace und Kernel Programmierung
3. Linux Kernel Programmierung auf dem Raspberry Pi
4. Ein erstes Hello-World Kernelmodul
5. Makefile zum Kompilieren des Moduls
6. Module verwalten mit der Bash
7. Der Device Tree
8. Device Tree Overlays
9. Das I2C Subsystem
10. Das IIO Subsystem

Der Linux Kernel

- Abstrahierung der Hardware
- Zuständig für Speicherverwaltung, Prozessverwaltung, Multitasking, Lastverteilung, Sicherheitserzwingung und Eingabe/Ausgabe-Operationen
- Monolithischer Kernel mit nachladbaren Modulen
- Zugriff auf Kernelfunktionen aus Userspace über Syscalls (open, close, read, write, ...)

Unterschiede zwischen Userspace und Kernel Programmierung

	Userspace Programm	**Kernel Modul**
Resourcenzugriff:	Eingeschränkter Zugriff, benötigt Berechtigungen	Uneingeschränkter Zugriff
Hardwarezugriff:	Nur über Treiber möglich	Direkter Zugriff auf Hardware möglich
Programm starten:	Kann einfach über User gestartet werden	Muss in den Kernel geladen werden
Latenz:	Höhere Latenz (Syscalls)	Geringe Latenzen

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`

Ein erstes Hello-World Kernelmodul

```c
#include <linux/module.h>
#include <linux/init.h>
int __init my_init(void)
{
        printk("hello_kernel - Hello Kernel!\n");
        return 0;
}
void __exit my_exit(void)
{
        printk("hello_kernel - Goodbye Kernel!\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
- `modinfo <modulname>` gibt Informationen über das Modul <modulname> aus

Aufgabe

Teste das einfache Kernelmodul auf Deinem Raspberry Pi.

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 → nicht der ganze Device Tree muss neu kompiliert werden, sollte ein Gerät hinzugefügt werden

Device Tree Overlays

```
/dts-v1/;
/plugin/;
/ {
        fragment@0 {
                target = <&i2c1>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        my_adc: my_adc@10 {
                                compatible = "brightlight,myadc";
                                reg = <0x10>;
                                status = "okay";
                        };
                };
        };
};
```

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

- Kompatibles Gerät benennen
- Probe und Remove Funktionen implementieren
- Treiberstruktur erstellen
- Treiber zum Kernel hinzufügen

```
#include <linux/i2c.h>

static struct of_device_id foo_of_ids[] = {
        {
                .compatible = "brightlight,myadc",
        }, {},
};
MODULE_DEVICE_TABLE(of, foo_of_ids);

static struct i2c_device_id foo_ids[] = {
        {"my_adc", &foo_drv_data}, {},
};
MODULE_DEVICE_TABLE(i2c, foo_ids);
```

Das I2C Subsystem

Probe und Remove Funktionen implementieren

```
static int foo_probe(struct i2c_client *client, const struct i2c_device_id *
    id)
{
        ...
}

static void foo_remove(struct i2c_client *client)
{
        ...
}
```

```
static struct i2c_driver foo_driver = {
        .probe = foo_probe,
        .remove = foo_remove,
        .id_table = foo_ids,
        .driver = {
                .name = "foo",
                .of_match_table = foo_of_ids,
        },
};
```

Das I2C Subsystem

Treiberstruktur zum Kernel hinzufügen

```
int __init my_init(void)
{
        return i2c_add_driver(&foo_driver);
}

void __exit my_exit(void)
{
        i2c_del_driver(&foo_driver);
}
```

oder

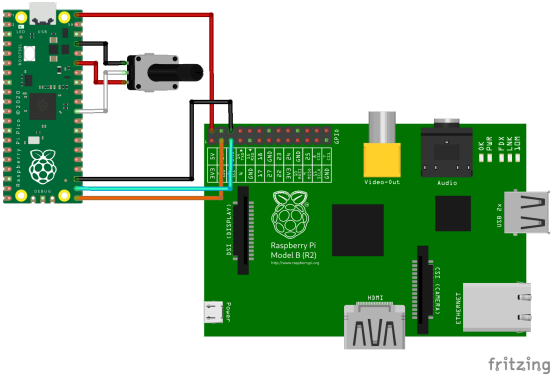
```
module_i2c_driver(foo_driver);
```

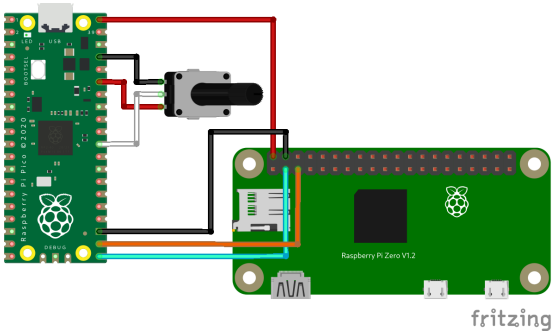
- `u8 i2c_smbus_read_byte(struct i2c_client *client)` liest ein Byte vom I2C Bus
- `s32 i2c_smbus_write_byte(struct i2c_client *client, u8 data)` schreibt den Wert der Variable data auf den I2C Bus
- `void mdelay(int delay_ms)` Funktion wartet delay_ms Millisekunden. Benötigter header: `linux/delay.h`
- `void i2c_set_clientdata(struct i2c_client * client, void *data)` Setzt Daten für den I2C Client. Diese können später wieder abgerufen werden
- `void *i2c_get_clientdata(struct i2c_client * client)` Ruft die gespeicherten Daten des I2C Clients ab
- `void *devm_kalloc(struct device *dev, size_t size, gfp_t gfp)` Allokiert size Byte an Speicher. Der Speicher ist an die Lebensdauer des Gerätes dev gebunden und wird automatisch freigegeben, sobald das Gerät nicht mehr benötigt wird. Über gfp können Allocation flags übergeben werden. Üblich ist `GFP_KERNEL`.

- Raspberry Pi Pico als Testgerät für Treiberentwicklung
- Poti ist an ADC0 (Analog Digital Converter) angeschlossen
- Wert des ADCs kann über I2C ausgelesen werden
- Ein Lesezugriff gibt den letzen ausgelesenen ADC Wert zurück
- Schreibzugriffe konfigurieren das Gerät. Wird der Wert 1 geschrieben, wird der ADC alle 500ms ausgelesen, wird eine 2 geschrieben, wird der ADC einmal ausgelesen. Bei allen anderen Werten wird der ADC nicht ausgelesen

Das I2C Subsystem

Der Raspberry Pi Pico I2C ADC





- `i2cdetect -y 1` Scant den Bus /dev/i2c-1 nach verfügbaren Geräten ab
- `i2cget -y 1 0x10` Liest ein Byte von dem I2C-Gerät mit der Adresse 0x10
- `i2cset -y 1 0x10 0x12` Schreibt den Wert 0x12 zu dem I2C-Gerät mit der Adresse 0x10

Aufgaben

- Erstelle einen Treiber für den Raspberry Pi Pico I2C ADC. in der Probe Function soll eine ADC Wandlung angestoßen, ausgelesen und das Ergebnis im Kernelog angezeigt werden
- Erstelle ein Device Tree Overlay für den Raspberry Pi Pico I2C ADC
- Compiliere den Device Tree Overlay und lade ihn mit `dtoverlay <overlay>`
- Teste Deinen Treiber

Das IIO Subsystem

- Industrial I/O
- Ermöglicht Zugriff auf verschiedene Sensorarten (ADCs, Temperatur-, Druck, Beschleunigungssensoren, ...)
- Standartisiert Schnittstelle zu verschiedenen Sensoren
- Zugriff über das IIO-Charaktergerät oder das IIO-Abstraktionsgerät
- Bietet verschiedene Funktionen zum Auslesen der Sensoren (Lesen von Rohdaten, Kalibirierung, Skalierung, ...)

- Struct für IIO Gerät anlegen
- Read Funktion implementieren
- Channels und IIO Info anlegen
- Gerät in Probe Funktion anlegen

```
#include <linux/iio/iio.h>
#include <linux/iio/sysfs.h>

struct foo {
        ...
};

static int foo_read_raw(struct iio_dev * indio_dev, struct iio_chan_spec
    const * chan, int *val, int *val2, long mask) {
        if(mask == IIO_CHAN_INFO_RAW) {
                /* Lese Sensor aus */
                ...
        } else
                return -EINVAL;
        return IIO_VAL_INT;
}
```

Das IIO Subsystem

Channels und IIO Info anlegen

```
static const struct iio_chan_spec foo_channels[] = {
        {
                .type = IIO_VOLTAGE,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        }
};

static const struct iio_info foo_info = {
        .read_raw = foo_read_raw,
};
```

```
int my_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
        struct iio_dev *indio_dev;
        struct foo *adc;

        indio_dev = devm_iio_device_alloc(&client->dev, sizeof(struct
            iio_dev));
        if(!indio_dev)
                return -ENOMEM;

        adc = iio_priv(indio_dev);
        adc->client = client;
```

```
        indio_dev->name = id->name;
        indio_dev->info = &foo_info;
        indio_dev->modes = INDIO_DIRECT_MODE;
        indio_dev->channels = foo_channels;
        indio_dev->num_channels = ARRAY_SIZE(foo_channels);

        return devm_iio_device_register(&client->dev, indio_dev);
}
```

Aufgaben

- Erweitere den Treiber, sodass der ADC über IIO ausgelesen werden kann. Achte darauf, in Probe das Dauerhafte Auslesen des ADCs zu aktivieren (Zuerst eine 1 schreiben).
- Für den Treiber wird das IIO Kernelmodul benötigt. Du kannst es über `modprobe industrialio` laden.
- Lese den ADC Wert über `/sys/bus/iio/devices/iio:device0/in_voltage_raw` aus
- Erweitere den Treiber, sodass in der Remove Funktion der ADC ausgeschalten wird (Schreibe eine 0 über I2C).