Linux Driver Workshop

An introduction to Linux Driver Programming

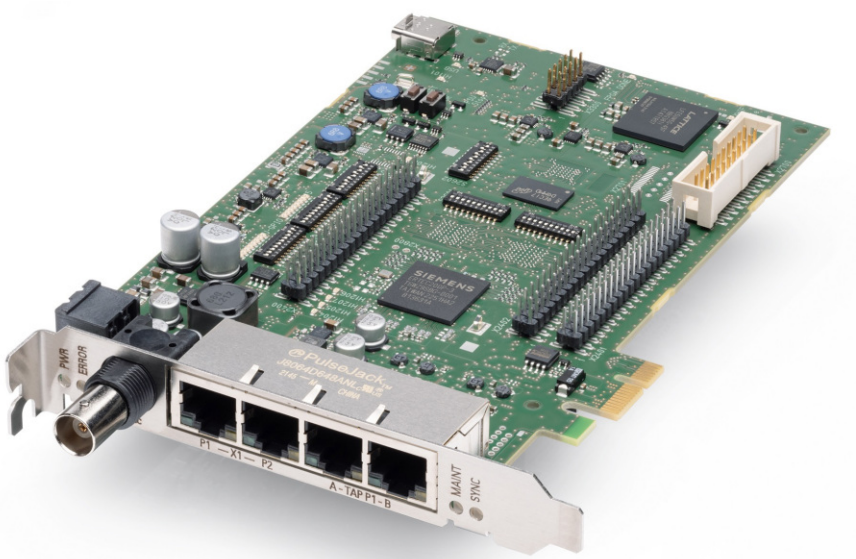
Johannes Roith

05.12.2025

About me

- Embedded Software Developer at Siemens AG
- Embedded Linux YouTube Channel
- My website with links to GitHub, Mastadon, LinkedIn, ...
- A driver of mine made it into the Linux Kernel

About my work



Agenda

1. The Linux Kernel
2. Development Enviroment for Linux Driver Programming
3. A Hello World Linux Kernel Module
4. Makefile for building the Kernel Module
5. Managing Modules in a shell

Agenda

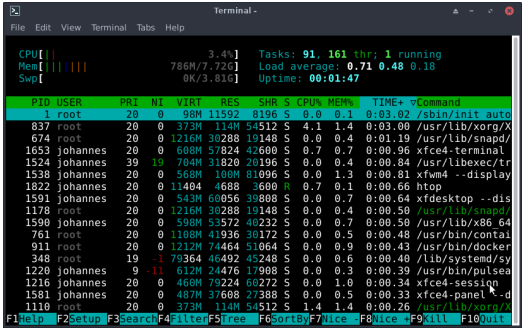
Material for the workshop

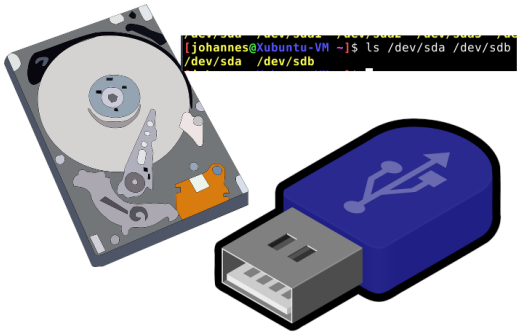
https://github.com/Johannes4Linux/ese25

The Linux Kernel

- Kernel of an operating system: hardware abstraction layer
- Uniform interface (API Systemcalls) independent from PC architecture
- Tasks of the Linux-Kernels:
  - Memory management
  - Process management
  - Multitasking
  - Load balancing
  - Access to hardware over drivers
- Applications are using systemcalls (open, close, read, write, ioctl, ...): they don't need knowledge about the underlying hardware
- Linux: modular monolithic Kernel with loadable modules

The Linux Kernel

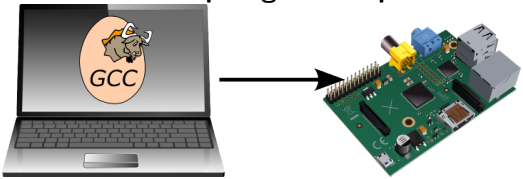




Development Enviroment for Linux Driver Programming

On Target Development

Cross-Compiling Development



- Update packages with: `sudo apt update && sudo apt upgrade -y`
- Install Kernel Headers: `sudo apt install -y raspberrypi-kernel-headers`
- Install build tools like gcc, make, ...: `sudo apt install -y build-essential`
- Reboot, to start updated kernel: `sudo reboot`

A Hello World Linux Kernel Module

```c
#include <linux/module.h>
#include <linux/init.h>
int __init my_init(void)
{
        printk("hello_kernel - The disaster takes its course...\n");
        return 0;
}
void __exit my_exit(void)
{
        printk("hello_kernel - The kernel got off lightly there.\n");
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes Roith");
MODULE_DESCRIPTION("A simple Kernel Module");
module_init(my_init);
module_exit(my_exit);
```

Makefile for building the Kernel Module

```
# Kernel Header Makefile compiles hello.c to hello.o file automatically
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
# Result of the Builds: Loadable Kernel Object hello.ko
```

Managing Modules in a shell

- `lsmod` lists all loaded modules
- `dmesg` shows the kernel's log
- `insmod hello.ko` load the module `hello.ko` into the kernel
- `rmmod hello` removes the module <`hello` from the kernel
- `modinfo ./hello.ko` shows the meta-data (author, licence, description, ...) of the module `hello.ko`
- `modprobe industrialio` loads the module `industrialio` together with all its dependencies

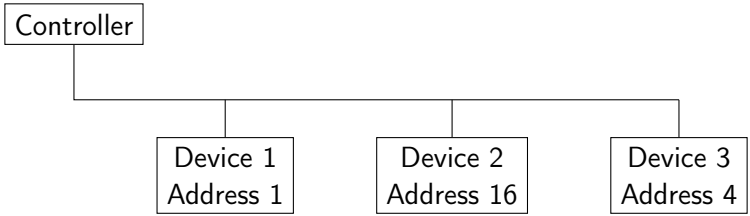
Exercise

- Implement the Kernel Module `hello` on the Raspberry Pi in the folder `aufgabe_1`.
- Build the Module with the Makefile.
- Load the Module.
- Check the Kernel's log and if the Module is loaded.
- Remove the Module

The I2C bus

- Simple two wire bus
- Data line: *SDA*
- Clock line: *SCK*
- Supported frequencies: 100kbit/s, 400kbit/s, 1Mbit/s
- Pull-Up resistor on both signals necessary

The I2C bus


Controller
Device 1
Address 1
Device 2
Address 16
Device 3
Address 4

A Linux I2C Driver

Header and compatible devices

```c
/* Required Header */
#include <linux/i2c.h>

/* Name all compatible devices */
static struct i2c_device_id my_ids[] = {
        {"my_dev"},
        {} /* empty element signalizes end of list*/
};
MODULE_DEVICE_TABLE(i2c, my_ids);
```

A Linux I2C Driver

Probe- and Remove functions

```
/* Function gets called when a compatible device is added */
static int my_probe(struct i2c_client *client)
{
        printk("Hello from I2C device with address: 0x%x\n", client->addr);
        return 0;
}

/* Function gets called when a compatible device is removed */
static void my_remove(struct i2c_client *client)
{
        printk("Bye, bye, I2C\n");
}
```

A Linux I2C Driver

Bundle driver struct

```
/* Bundle compatbile devices, probe and remove functions and driver info
    into driver struct */
static struct i2c_driver my_driver = {
        .probe = my_probe,
        .remove = my_remove,
        .id_table = my_ids,
        .driver = {
                .name = "my-i2c-driver",
        }
};
```

A Linux I2C Driver

Register the driver

```
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);
```

or:

```
module_i2c_driver(my_driver);
```

Adding I2C devices over sysfs

```
# Change to I2C device folder
cd /sys/bus/i2c/devices/i2c-1

# Add I2C device mydev with address 0x12
echo "my_dev 0x12" | sudo tee new_devices

# Removes I2C device with address 0x12
echo "0x12" | sudo tee delete_device
```

Exercise

- Implement the Kernel Module `rgb_brd.c` on the Raspberry Pi in the folder `aufgabe_2` as follows:
  - The name of the compatible device should be *rgb_ brd*
  - The probe function should print out the I2C address of the device into the kernel's log
  - Another kernel's log message should be written to the log when removing the device
- Compile the kernel with the Makefile
- Load the module
- Check that the module is loaded
- Add a compatible I2C device
- Check the Kernel's log
- Remove the device and the module

PCF8574 IO Expander

- Write access writes output values P0 - P7
- Read access reads current values P0 - P7
- Button connected to P0
- For input operation: Set Output to 1, Button pulls pin to GND. When reading a 1, the button is not pushed, when reading a 0 it is pushed
- Red LED connected to P1, green to P2, blue to P3
- Set output to 0: LED is On
- Set output to 1: LED is Off

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

Accessing the I2C bus

```
struct i2c_client *my_client;
```

The `struct i2c_client` is used, to manage an I2C device in the kernel. With the pointer `my_client` we can access the device, e.g. for reading or writing data.

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

Reads a byte from the I2C device `my_client`. If an error occurs, the function returns a negative error code, else the read byte.

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

Writes the byte `value` to the I2C device `my_client`. If an error occurs, the function returns a negative error code, else 0.

Exercise

- Copy the file `aufgabe_2/rgb_brd.c` to `aufgabe_3` and edit the copy in this folder.
- Light up the RGB LED in a color of your choice. You can do so, by writing to P1-P3 ofthe PCF8574 in the probe function of the driver.
- Turn of the RGB LED in the remove function.
- Compile and test the module.
- Additional task: Read the state of the button on P0 in the probe function and write it tothe Kernel's log.

Creation of sysfs entries

- *sysfs*: Virtual filesystem
- Display and management of *Kernel Objects* (`kobject`)
- Allows interaction with the driver
- *Kernel Object*: Folder in *sysfs*
- *Kernel Object* can have attributes (represented as files) over which the driver can exchange data with user space.
- Procedure: Implement show and store functions, create attribute, create Kernel Object, link sysfs files with Kernel Object

Show and store functions and attribute

```c
/* Required 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);
```

Create kobject and link it with the attribute

```
struct kobject * my_kobj */
/* in init or probe function */
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;
}
```

Delete kobject and attribute

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

Exercise

In `aufgabe_4` you can find a demo driver for creating a Kernel Object and an Attribute. Modify the driver as follows:

- Rename the Kernel object to *rgb_brd*.
- Create the attribute *led* of the object *rgb_brd*.
- Implement the store function for this attribute, so you can control the RGB LED over it. By writing the string `011` to the attribute, the red LED should be set t0 0, the green LED to 1 and the blue LED to 1.
- Additional Task: Create a second attribute `button` of the Kernel Object `rgb_brd`. Implement a show function to read out the current state of the button.

- ARM/Open RISC V systems do not have automatic hardware detection like e.g. the BIOS on x86 systems
- The Linux kernel requires information on which devices are available → Device Tree provides this information
- Device Tree summarizes the available devices in a tree structure
- The Device Tree Sources (dts) and Device Tree Source Includes (dtsi) must be compiled (dtb: Device Tree Binary)
- Device Tree available under `/sys/firmware/devicetree/base`
- Convert to readable form: `dtc -I fs -O dts -s /sys/firmware/devicetree/base > dt.dts`
- Device tree can also be extended via overlays → not the whole device tree has to be recompiled if a device is added

Adding Devices over the Device Tree

Device Tree Overlay for an I2C device

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

Compile the overlay with `dtc -@ -I dts -O dtb -o testoverlay.dtbo testoverlay.dts`

Adding Devices over the Device Tree

Expand the I2C Device Driver

```
/* Name compatible Device Tree Devices */
static struct of_device_id my_driver_of_ids[] = {
        { .compatible = "brightlight,my_dev", },
        { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, my_driver_ids);

/* Add OF IDs to driver struct */
static struct i2c_driver my_driver = {
        ...
        .driver = {
                .name = "my-i2c-driver",
                .of_match_table = my_driver_of_ids,
        }
};
```

Exercise

- Copy `aufgabe_4/rgb_brd.c` to `aufgabe_5` and edit the copy in this folder.
- Create a device tree overlay for the *rgb_brd* device.
- Compile the device tree overlay
- Load the device tree overlay with `sudo dtoverlay rgb_brd.dtbo`
- Add support for device tree devices to the I2C device driver.
- Build and test the driver with device tree support.

(Literature) Recommendations

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