Linux Driver Workshop

An introduction to Linux Driver Development

Johannes Roith

13.10.2024

About me

- Embedded Software Developer
- Embedded Linux YouTube Channel
- my website with links to GitHub, Mastadon, LinkedIn, ...
- A driver of mine has made it into the Linux kernel

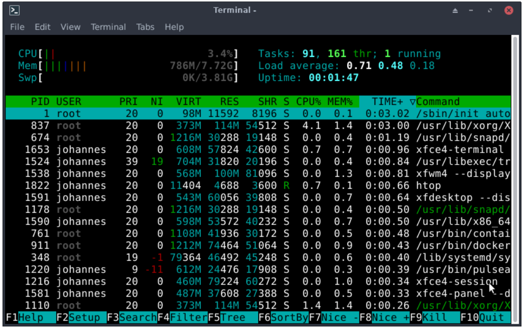
Agenda

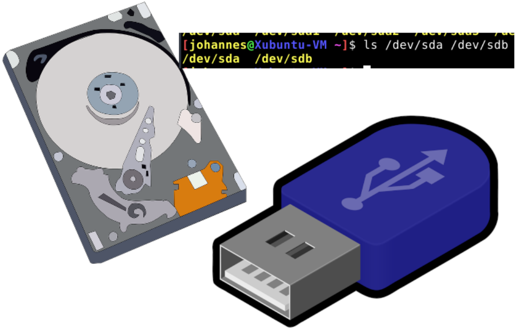
Backup

The Linux Kernel

- The kernel of an operating system: hardware-abstracting layer
- Uniform interface (API, system calls) independent of computer architecture
- Tasks of the Linux kernel:
  - Memory management
  - Process management
  - Multitasking
  - Load balancing
  - Access to hardware via drivers
- Applications use system calls (open, close, read, write, ioctl, . . .): does not require precise knowledge of the hardware
- Linux: modular monolithic kernel with loadable modules

The Linux Kernel





Linux Kernel Programming on a Raspberry Pi

- 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 load new kernel (if installed during update): `sudo reboot`

A first hello world 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 - But the kernel got off lightly again!\n");
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes Roith");
MODULE_DESCRIPTION("A simple hello world LKM");
module_init(my_init);
module_exit(my_exit);
```

Listing 1: hello_kernel.c

Makefile to compile the Module

```
# Kernel header makefile compiles hello_kernel.c automatically to
    hello_kernel.o
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
```

Managing modules in a shell

- `lsmod` lists the loaded modules
- `dmesg` shows the kernel's log
- `insmod <modulname>` loads the module `<modulname>` into the Kernel
- `rmmod <modulname>` removes the module `<modulname>` from the Kernel
- `modprobe <modulname>` loads the module `<modulname>` together with all dependencies
- `modinfo <modulname>` shows the Meta data (author, license, description, ...) of the module `<modulname>`

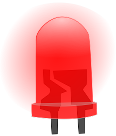
Exercise

- Implement the Hello World Kernel Module on your Raspberry Pi
- Compile the module with the Makefile
- Load the module
- Check the kernel's log and if the module is loaded
- Remove the module from the kernel

GPIO: General Purpose Input/Output

Digital Output

- drive output high (3.3V on RPi, digital 1)
- drive output low (GND, 0V on RPi, digital 0)

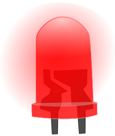


Controlling GPIOs

GPIO: General Purpose Input/Output

Digital Output

- drive output high (3.3V on RPi, digital 1)
- drive output low (GND, 0V on RPi, digital 0)



Digital Input

- 0 was read: Pin connected to GND (0V)
- 1 was read: Pin connected to 3.3V

Controlling GPIOs

```
struct gpio_desc *my_gpio;
```

The structure of type `struct gpio_desc` is used to manage a GPIO in the kernel. We can then access the GPIO via the pointer `my_gpio`. The structure and the following functions are contained in the header `linux/gpio/consumer.h`.

```
struct gpio_desc *gpio_to_desc(unsigned int gpio);
```

Converts a GPIO number `gpio` into a GPIO descriptor. If a valid number was passed, a pointer to the GPIO is returned, otherwise `NULL`.

```
int gpiod_direction_input(struct gpio_desc *my_gpio);
int gpiod_direction_output(struct gpio_desc *my_gpio, int value);
```

Configures the GPIO pin `my_gpio` as an input or output. When configuring as an output, the initial state of the pin must be transferred. If successful, the function returns 0, otherwise an error code.

Controlling GPIOs

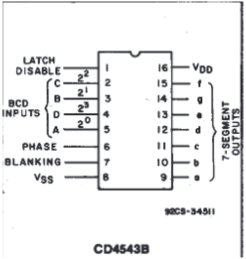
`void gpiod_set_value(my_gpio, value)`

Sets the GPIO pin `my_gpio` to the output value `value`. The pin must have been previously configured as an output.

`int gpiod_get_value(my_gpio)`

Reads the current input value of GPIO pin `my_gpio`. The pin must have been previously configured as an input. Return value is the state of the IO pin.

7 Segment Display Decoder



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
all other compinations				empty

Exercise

- connections: A: 11, B: 9, C: 25, D: 8
- Initialize the four GPIOs for the 7-segment display as outputs in the Init function of the driver
- A number from 0 to 9 should then be displayed.
- Set all output GPIOs to 0 in the Exit function
- Build and test the kernel module
- Additional task: Initialize pin 24 as an input and read its state in the Init function

Device numbers and character-oriented devices

```
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);
...
```

Required 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 contains pointers to callback functions. Not all functions need to be implemented.

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

- `file`: Information about the open file, e.g. device number (major and minor)
- `buf`: Text buffer with data to be written (`__user` indicates that it is a pointer from the user space that should be transferred to the kernel space)
- `cnt`: Size of the text buffer in bytes
- `off`: Pointer with offset in file, can be increased by the number of bytes written after writing

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

Assigns the major device number `major`. An entry with the major device number and the name `name` is created under `/proc/devices`. `fops` specifies the available file operations. If successful, the function returns a 0, otherwise an error code.

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

Releases the major device number `major` with the name `name`. If successful, the function returns a 0, otherwise an error code.

Device numbers and character-oriented devices

Copy data

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

Copies `len` bytes from the userspace buffer `*src` to the kernelspace buffer `*dst`.

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

Copies `len` bytes from the kernel space buffer `*src` to the user space buffer `*dst`.

Device numbers and character-oriented devices

Create a device file and use it in Bash

`sudo mknod /dev/seg c 64 0`

Creates the character-oriented device file `/dev/seg`. The device file is assigned the major device number 64 and the minor device number 0.

`echo 7 > /dev/seg`

Writes the string `"7\n\0"` to the device file `/dev/seg`. Depending on the file authorization, the command must be executed as root.

Device numbers and character-oriented devices

Create a device file and use it in Bash

`cat /dev/seg`

Reads the device file until the end of the file (read returns 0) is reached. Depending on the file authorization, the command must be executed as root.

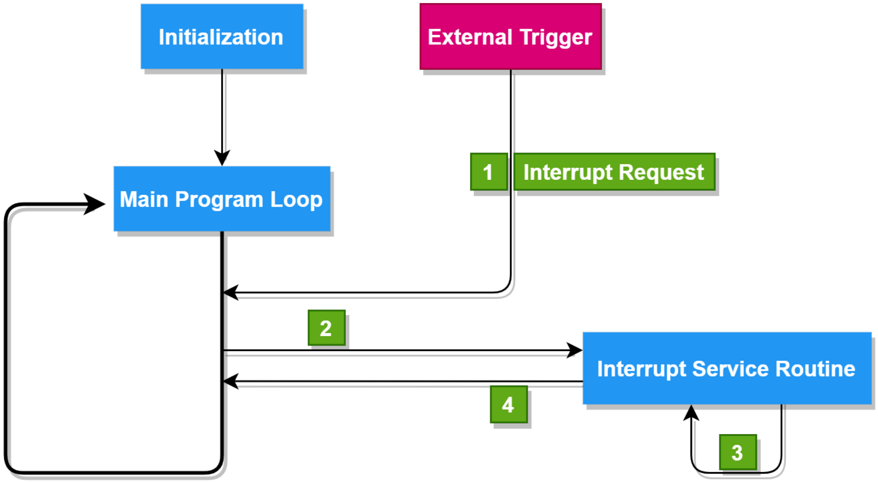
`head -n 1 /dev/seg`

Reads the device file until a `'\n'` newline character is reached. Depending on the file authorization, the command must be executed as root.

Exercise

- Assign the major device number 64 in the Init function of the driver and release it again in the Exit function.
- Implement the write callback so that a number from 0-9 is displayed on the 7-segment display.
- Test the driver
- Additional task: Implement the read callback so that the current status of the button can be read.

GPIO Interrupt



GPIO Interrupt

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

Returns the interrupt number for the GPIO `gpio_nr`.

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

Activates the interrupt `irq_nr`. The `flags` can be used to specify when the interrupt is triggered (e.g. `IRQF_TRIGGER_RISING` for rising edge). If the IRQ is active, the `handler` function is called. The IRQ number and, if set, the `dev_data` are passed as parameters. The `name` appears in `/proc/interrupts`.

GPIO Interrupt

Example for ISR

```
irqreturn_t my_irq_handler(int irq_nr, void *data)
{
        printk("IRQ %d activ\n", irq_nr);
        /* Tue irgendwas */
        if(/* IRQ not handled */)
                return IRQ_NONE;
        else /* IRQ handled */
                return IRQ_HANDLED;
}
```

ioctl

- input/output control
- System call in Unix for specific control commands that cannot be set via read/write
- Commands and parameters can be defined independently

ioctl

Define ioctl commands and parameters

```c
/* File cmd.h */

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

/* Alternativley you can use
 * #define SET_ANSWER 0x1
 * #define GET_ANSWER 0x2
 */
```

ioctl callback in driver

```c
#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("The answer to everything is %d\n", answer);
```

Create Device Files with udev

Request device number and create character device

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

Registers `cnt` device numbers from the device number `devnr`. An entry with `MAJOR(devnr)` and the name `name` is created in `/proc/devices`. Required header: `linux/fs.h`

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

Initializes the character device `cdev` with the file operations in `fops`. Required header: `linux/cdev.h`

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

Adds the character device `cdev` to the system and links it to `cnt` device numbers from `devnr`.

Create Device Files with udev

Create class and device

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

Creates a new class under `/sys/class/` with the name `name`. Check return pointer with `IS_ERR` for validity!

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

Creates a new device in the class `class` and links it to the device number `devnr`. The name `name` then appears in `/dev/` as a device file. Check return pointer with `IS_ERR` for validity!

Create Device Files with udev

Cleanup device number, device and class

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

Destroys the device in the class `class` that is linked to the device number `devnr`.

`void class_destroy(struct class *class)`

Destroys the class `class`.

`void unregister_chrdev_region(dev_t devnr, unsigned cnt)`

Free `cnt` device numbers from `devnr`.

- 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

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

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

Driver for Device Tree Devices

Probe- und Remove-Function

```
#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)
{
        ...
}
```

Driver for Device Tree Devices

Compatible Devices and driver-struct

```c
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,
        }
};
```

Driver for Device Tree Devices

Register Driver

```c
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);
```

or:

```c
module_platform_driver(foo_driver);
```

Check Device Tree Properties

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

Checks whether the property `propname` exists in `pdev->dev`. If the property exists, a `1` is returned, otherwise `0`.

Driver for Device Tree Devices

Use GPIOs

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

Imports and initializes the GPIO with the label `<label>-gpio` from the device tree entry for the device `dev = pdev->dev`. The `flags` indicate the direction. `GPIO_IN` for configuration as input, `GPIO_OUT_LOW` for output. Check return pointer with `IS_ERR` for validity! Required header: `<linux/gpio/consumer.h>`.

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

Sets the output value of the GPIO `gpio` to the value `value`.

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

Returns the input value of the GPIO `gpio`.

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

Releases the GPIO `gpio` again.