

1.

a)

Character Devices werden ohne Buffer direkt gelesen und geschrieben. Block Devices werden über einen Cache gelesen und geschrieben. Block Devices müssen Random Access sein, Character Devices nicht zwingend. Character Devices haben 'c' als erste Spalte im Output von `ls -l`. Block Devices haben 'b'.

b)

Die Major Number stellt die Beziehung zwischen Device und Treiber dar. Meistens besitzt ein Treiber eine Major Number. Bei neueren Linux Kernels können sich mehrere Treiber eine Major Number teilen. Die Minor Number wird zur Unterscheidung mehrerer Devices verwendet. Auf das spezielle Device kann entweder mit einem direkten Pointer oder über ein lokales Array, von Devices, mit der Minor Number als Index, zugegriffen werden. Die zwei Numbers stehen vor dem Datum der letzten Modifikation. Als erstes wird die Major Number angezeigt und als zweites die Minor, getrennt durch ein Komma.

```
crw-rw-rw- 1 root root 1, 3 Apr 11 2002 null
```

c)

Die beiden Methoden werden benötigt um Daten vom User in den Kernel Mode und umgekehrt zu kopieren. Vom Aufbau her sind sie ähnlich zu `memcpy`.

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
```

### 2.1.1

In der `file_operations` Struktur wird eine `read` zum Auslesen des Ledstatus und `write` Funktion zum Setzen des Ledstatus benötigt. Die Major Number wird dynamisch auf eine freie Nummer festgelegt. Die Minor Number wird auf 0 gesetzt da immer nur ein Device verwendet wird.

### 2.1.2

*request\_mem\_region*

Reserviert den physikalischen Speicherbereich.

*ioremap\_nocache*

Mappt den physikalischen Speicherbereich auf einen virtuellen Speicher

*alloc\_chrdev\_region*

Registriert eine gewisse Anzahl an Device Nummern.

*cdev\_init*

Initialisiert eine `cdev` Struktur.

*cdev\_add*

Fügt ein Character Device zum System hinzu.

**Code:**

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include "gpio_defs.h"
#include "hwaccess.h"

#define DEVICE_NAME "leds"
#define MINOR_NUMBER 0
#define NUMBER_OF_DEVICES 1

MODULE_LICENSE("Dual BSD/GPL");

struct gpio_dev_t gpio_dev;

static ssize_t ledDriver_read(struct file *filp, /* see include/linux/fs.h */
                             char *buffer, /* buffer to fill with data */
                             size_t length, /* length of the buffer */
                             loff_t * offset)
{
    buffer[0] = readSwitches(gpio_dev.vaddr);
    return 1;
}

static ssize_t ledDriver_write(struct file *filp, const char *buffer, size_t length, loff_t
* offset)
{
    setLeds(gpio_dev.vaddr, buff[0]);
    return 0;
}

struct file_operations ledDriver_fops = {
    read:    ledDriver_read,
    write:   ledDriver_write
};

static int ledDriver_init(void)
{
    if (request_mem_region(GPIO_BASE, sizeof(GpioRegs), DEVICE_NAME) == NULL)
    {
        printk(KERN_ALERT "Error requesting memory\n");
        return -1;
    }
    gpio_dev.vaddr = (GpioRegs*)ioremap_nocache(GPIO_BASE, sizeof(GpioRegs));

    if (alloc_chrdev_region(&gpio_dev.first, MINOR_NUMBER, NUMBER_OF_DEVICES, DE-
VICE_NAME))
    {
        goto fail_alloc_chrdev;
    }

    cdev_init(&gpio_dev.cdev, &ledDriver_fops);

    if (cdev_add(&gpio_dev.cdev, gpio_dev.first, NUMBER_OF_DEVICES) < 0)
    {
        goto fail_cdev_add;
    }
}
```

```

        initLeds(gpio_dev.vaddr);
        initSwitches(gpio_dev.vaddr);

        return 0;

fail_cdev_add:
    unregister_chrdev_region(gpio_dev.first, NUMBER_OF_DEVICES);

fail_alloc_chrdev:
    iounmap((void*)gpio_dev.vaddr);
    release_mem_region(GPIO_BASE, sizeof(GpioRegs));

    return -1;
}

static void ledDriver_exit(void)
{
    cdev_del(&gpio_dev.cdev);

    unregister_chrdev_region(gpio_dev.first, NUMBER_OF_DEVICES);

    iounmap((void*)gpio_dev.vaddr);
    release_mem_region(GPIO_BASE, sizeof(GpioRegs));
}

module_init(ledDriver_init);
module_exit(ledDriver_exit);

```