Praktikum Treiber

Frühlingssemester 2020

M. Thaler, A. Schmid

Inhaltsverzeichnis

1 Einleitung

1.1 Ziele

In diesem Praktikum werden sie sich mit Modulen und Treibern beschäftigen. sie erhalten dabei einen Einblick in den Aufbau des Linux Kernels, speziell mit Sicht auf die Ansteuerung von Hardware. Gleichzeitig lernen sie Problemstellungen im Zusammenhang mit der Funktionalität von Betriebssystemkernen kennen.

1.2 Aufbau des Praktikums

Das Praktikum besteht aus zwei Teilen: einem Theorieteil und den eigentlichen Aufgabenstellungen. Da sie schon im Theorieteil mit Linux Modulen arbeiten, ist es notwendig als erstes die `kbuild`-Infrastruktur und die Linux Kernel Headers zu installieren, falls sie dies nicht schon gemacht haben ist. Eine Anleitung finden sie in Abschnitt 1.5.

Die vorbereiteten Files zum Praktikum finden sie im Archiv: **`Treiber.tgz`**.

In Abschnitt 2 geben wir Ihnen eine kurze Einführung zu Linux Modulen, die die Basis für die Implementierung von Treibern bilden. Dabei werden sie einige einfache Übungen mit Modulen durchlaufen.

In Abschnitt 3 folgen die Aufgabenstellungen. sie werden zuerst einen einfachen Treiber implementieren, mit dem sie Daten (Strings) im Kernel abspeichern und wieder auslesen können. In einem zweiten Teil werden sie eine Ampelsteuerung realisieren, die auf einen Treiber für das Parallel-Port zugreift. Für Rechner ohne Parallel-Port steht ein Kernelmodul zur Verfügung, das das Port emuliert, sowie für die Visualisierung der Ampel ein Java-Programm oder eine einfache C-Implementation mit Pseudografik.

Am Schluss (Abschnitt 4 und 5) finden sie die zusätzliche theoretische Grundlagen sowie die benötigten Kernel-Befehle.

1.3 Durchführung und Leistungsnachweis

Es gelten grundsätzlich die Vorgaben ihres Dozenten zur Durchführung der Praktika und zu den Leistungsnachweisen im Kurs BSy. Die Inhalte des Praktikums gehören zum Prüfungsstoff.

1.4 Zu beachten

Arbeiten sie soweit wie möglich als Anwender. Einige Befehle wie das Laden und Entladen von Modulen und Treibern (`insmod` bzw. `rmmod`) erfordern jedoch root-Rechte: verwenden sie **`sudo -i`** bzw. **`sudo su`** oder loggen sie sich jeweils im Arbeitsverzeichnis mit **`su root`** ein, falls ihre Linux-Distribution das zulässt.

1.5 Installation der Kernel Headers und der `kbuild`-Infrastruktur

Informieren Sie sich zuerst bei Ihrer Distribution, welche Packete für die Übsetzung von Modulen benötigt werden. Für das Übersetzen von Kernel-Modulen und Treibern, benötigen sie in der Regel die zu ihrem Kernel passenden Kernel Headers und eventuell die entsprechende `kbuild`-Infrastruktur (Debian). Ob die Linux Kernel Headers installiert und `kbuild` sind, können sie wie folgt überprüfen:

```
ls /usr/src/linux-kbuild* (nur Debian)
ls /lib/modules/`uname -r`  bzw.  dpkg --list linux-headers*
```

Überprüfen sie auch, ob die Versionen mit der Ausgabe von **`uname -r`** übereinstimmen.
Installieren sie, falls notwendig die `linux-headers` und bei Debian `linux-kbuild`.

1.6 Literatur

[1] Alessandro Rubini, Jonathan Corbet, Greg Kroha-Hartmann, *LINUX Device Drivers*, 3rd edition, O'Reilly, June 2005.

[2] Robert Love, *Linux Kernel Development*, 3rd ed., Pearson, 2010.

[3] Jürgen Quade, Eva-Katharina Kunst, *Linux-Treiber entwickeln*, 2. Auflage, dpunkt.verlag, 2006.

[4] http://tldp.org, *Linux Kernel Module Programming Guide*

[5] http://lwn.net/Kernel/LDD3, *Linux Device Drivers*, On-Line Book (A. Rubini).

2 Einführung Linux-Kernel und Module

2.1 Aufbau und Funktionalität des Linux-Kernels

Der Linux-Kernel ist grundsätzlich monolithisch aufgebaut, d.h. dass Hauptfunktionen fest in den Kernel kompiliert sind. Zur Laufzeit können jedoch Module dynamisch von Hand oder automatisch[1] nachgeladen bzw. wieder entfernt werden. Der Kernel wird dabei entsprechend vergrössert oder verkleinert. Module werden beim Laden Teil des Kernels und haben Zugriff auf alle seine Symbole, Funktionen, etc. Gerätetreiber werden während der Entwicklungsphase typischerweise als Module realisiert und von Hand geladen.

2.2 Gerätetreiber

Aufgabe der Gerätetreiber (Device Driver oder Treiber) ist es, I/O-Geräte zu verwalten und dem Benutzer definierte Schnittstellen für den Zugriff auf diese Geräte zur Verfügung zu stellen.

Linux unterscheidet folgende vier Gerätetypen:

Character Devices Character Devices werden wie Files (Streams) behandelt, sie können einzelne Zeichen verarbeiten. Einige Character Devices können nur sequentiell gelesen werden, z.B. die Parallelschnittstelle oder die Tastatur, d.h. es gibt keine Möglichkeit, den File-Zeiger zu positionieren.

Block Devices Block Devices realisieren meist Filesysteme, da Daten hier nur blockweise gelesen oder geschrieben werden.

SCSI-Devices SCSI-Devices werden treiberintern speziell behandelt, Benutzer können sie als Character oder Block Devices ansprechen.

Network Interfaces Netzwerk Interfaces verwalten die Kommunikation mit anderen Rechnern und sind nicht im Device File System (in /dev) eingetragen (siehe [1] für Details).

2.3 Module

Werden Treiber fest in den Kernel kompiliert, muss bei jeder Änderung am Treiber der Kernel neu übersetzt und der Computer neu gestartet werden. Daher werden wir die Treiber als Module kompilieren und zur Laufzeit manuell laden und entfernen.

2.3.1 Header-Files

Die Header-Files für die Kompilation von Kernel-Modulen werden nicht in `/usr/include` gesucht, sondern in `/lib/modules/`uname -r`/build/include`.

[1] Für das automatische Laden und Entfernen von Modulen möchten wir auf die entsprechende Literatur verweisen.

2.3.2 Modulaufbau

Module besitzen einen definierten Einsprungpunkt und ein definiertes Ende. Das einfachste Modul muss dazu die zwei Prozeduren **init_module()** und **cleanup_module()** implementieren. Als Beispiel das "Hello World" Modul[2] im Verzeichnis `./hello`:

```
#define MODULE
#include <linux/module.h>
. . .
int init_module(void) {
   printk(KERN_ALERT "*** Hello World from Module ***\n");
   return(0);
}
void cleanup_module(void) {
   printk(KERN_ALERT "*** Good Bye from Module ***\n");
}
```

Beim Laden eines Moduls, wird die Prozedur **init_module()** aufgerufen, beim Entfernen die Prozedur **cleanup_module()** ausgeführt. Die beiden Prozeduren werden für Initialisierungen bzw. Freigabe von Ressourcen verwendet. Die Preprozessor-Anweisung `#define MODULE` signalisiert den Bibliotheken, dass es sich um ein Modul handelt, das dynamisch geladen und entfernt werden soll.

2.3.3 Kompilation von Modulen

Für die Kompilation der Module verwenden wir grundsätzlich Makefiles, möchten hier aber aus verschiedenen Gründen nicht auf die Details eingehen. Hier das Beispiel für unser Hello-World Beispiel (Kernelmodule werden in 2 Durchgängen übersetzt, deshalb das "ifneq"):

```
ifneq ($(KERNELRELEASE),)
    obj-m  := HelloWorld.o
else
    KDIR   := /lib/modules/$(shell uname -r)/build
    PWD    := $(shell pwd)
    all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
        @rm -f *.mod.o *.mod.c *.o > /dev/null
    clean:
        @echo cleaning $(PWD)..
        @rm -f *.o core* *.ko *.cmd *. .*.ko .*.cmd *.mod.c > /dev/null
        @rm -fr .tmp_versions > /dev/null
        @rm -f *.symvers *.order
endif
```

2.4 Laden und Entfernen von Modulen

Mit den beiden Shell-Befehlen **insmod** und **rmmod** können sie Module in den Kernel laden bzw. wieder entfernen. Mit **lsmod** können sie alle geladenen Module auflisten.
Damit sie Kernel-Meldungen beim Laden und Entladen der Module sehen, öffnen sie am besten ein zusätzliches Fenster und geben folgenden Befehl ein (ev. müssen sie `sudo` verwenden oder sich als `root` anmelden):

```
tail -f /var/log/syslog
```

[2] `KERN_ALERT` in `printk()` bestimmt die Priorität der Meldung. Es gibt insgesamt 8 Priortätslevel von 0 bis 7 (`KERN_ALERT` entspricht Level 1). Für Prioritäten >0 werden die Kernelmeldungen ev. nicht angezeigt, (siehe [3])

Wechseln sie ins Verzeichnis `./hello`, wo sich "Hello World" befindet und übersetzen sie das Module mit `make`. Beachten sie, dass kompilierte Module die Extension `ko` (Kernel Object) haben.

Module "Hello World" kompilieren und laden

```
root# make
root# insmod HelloWorld.ko
Message from syslogd@diva at Feb 13 16:07:40 ...
   kernel:[ 1970.696170] *** Hello World from Module ***
```

Module entladen

```
root# rmmod HelloWorld
Message from syslogd@diva at Feb 13 16:08:07 ...
   kernel:[ 1997.989149] *** Good Bye from Module ***
```

2.4.1 Setzen von Parametern beim Laden von Modulen

Beim Laden eines Moduls können beliebig viele Parameter übergeben werden:

```
insmod <modulname> <Paramter1=Wert1> <Paramter2=Wert2> ...
```

Folgende Typen für Parameter stehen zur Verfügung: **byte**, **short**, **int**, **long**, **string**.

Auch globale Variablen eines Moduls können als Parameter verwendet werden, sie werden beim Laden des Moduls auf den entsprechenden Wert gesetzt. Im Modul initialisierte Variablen werden mit dem angegebenen Parameterwert überschrieben (siehe Beispiel). Alle Variablen, die zur Initialisierung vorgesehen sind, müssen mit

```
module_parm(Variable, type-description, perms);
```

bekannt gemacht werden. Das Makro **MODULE_PARM** und die unten stehenden Paramtertypen sind in `<linux/module.h>` definiert. Die Parameter Typen müssen mit folgenden Kürzeln definiert werden (nur Auswahl):

```
int      für integer
charp    für string (char *)
perms    S_IRUGO (alle dürfen lesen)
```

Beispiel für ein Modul mit Übergabe von Parametern (siehe Verzeichnis `./paramod`)

```
#define MODULE
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/stat.h>

int   globalInt  = 10;
char* globalChar = "default text";
module_param(globalInt,  int,   S_IRUGO);
module_param(globalChar, charp, S_IRUGO);

int init_module(void) {
    printk(KERN_ALERT "*** ParaModule: %d - %s ***\n", globalInt, globalChar);
    return(0);
}
void cleanup_module(void) {
    printk(KERN_ALERT "*** Good Bye from ParaModule ***\n");
}
```

Laden sie das Modul mit `insmod` und unten stehenden Parametern. Beachten sie, dass der String keine Leerzeichen (white spaces) enthalten darf:

```
#root: insmod ParaModule.ko globalInt=55 globalChar="DasIstMeinText"
Message from syslogd@diva at Feb 13 16:15:05 ...
  kernel:[ 2415.633499] *** ParaModule: 5555 - DasIstMeinText ***
```

Vergessen sie nicht das Modul wieder zu entfernen.

2.4.2 Freigabe von Modulen

Grundsätzlich sorgt der Kernel dafür, dass ein noch referenziertes Modul nicht entladen werden kann. Dazu wird pro Modul ein Referenzzähler geführt, der vom Kernel selbst verwaltet wird. Für weitere Hinweise verweisen wir sie auf die Literatur.

2.4.3 Versionskontrolle (optional)

Module sollten die Kernelversion definieren, sie wird beim Laden des Moduls von **`insmod`** mit der aktuellen Kernelversion verglichen. Kernel ab Version 2.0 definieren das Symbol in <`linux/version.h`> (wird automatisch von <`linux/module.h`> eingebunden), daher muss das Symbol nicht explizit definiert werden. Wenn keine Version definiert werden soll, muss das Symbol `__NO_VERSION__` vor dem Einbinden von <`linux/module.h`> definiert werden.

2.4.4 Kernel Symbol-Tabellen (optional)

Module können dem Kernel Symbole zur Verfügung stellen (Adressen von Prozeduren und Variablen). Diese müssen beim Laden des Modul mit Hilfe des Macros `EXPORT_SYMBOL` an den Kernel exportiert werden (bitte Vorsicht, damit keine Namenskonflikte entstehen, d.h. die Namen dürfen im Kernel nur einmal vorhanden sein):

```
EXPORT_SYMBOL(name);
```

Das Makro ist in <`linux/module.h`> definiert, muss jedoch *aktiviert* werden: vor dem Einbinden von <`linux/module.h`> müssen sie das Symbol `EXPORT_SYMTAB` definieren:

```
#include <linux/module.h>
/* Definitionen von Prozeduren und Variablen, z.B: */
int demo_variable;
EXPORT_SYMBOL_GPL(demo_variable);
```

Beim Kompilieren eines Moduls wird das File `Module.symvers` erzeugt, das eine Liste mit den exportierten Symbolen enthält. Will ein anderes Modul auf diese Symbole zugreifen, müssen erstens diese Symbole im C-Programmcode als extern deklariert werden und zweitens muss das File `Module.symvers` mit den exportieren Symbolen ins Verzeichnis des C-Codes kopiert werden (es gibt andere Möglichkeiten die Symbole bekannt zu machen, das Kopieren des Files ist jedoch am einfachsten).

Die aktuellen Einträge der Kernel Symbol-Tabelle können sie mit folgendem Befehl anzeigen:

```
'cat /proc/kallsyms'
```

Anmerkung: das Emulationsmodul für die Parallel-Schnittstelle ist mit Kernel-Symbolen realisiert.

3 Aufgaben

3.1 Einführung

Das gesamte Praktikum basiert auf einem einfachen Memory-Treiber. Den Treiber finden sie bei den zur Verfügung gestellten Files (siehe Abschnitt 1.2). Das Praktikum besteht aus drei Teilaufgaben:

Aufgabe 1

sie installieren den Treiber `MemDriver` für das Gerät **`MemDev`**, das Daten (Strings) speichern kann. Dabei lernen sie die wichtigsten Grundfunktionen von Treibern anhand verschiedener Beispiele kennen.

Aufgabe 2

sie erweitern den Treiber für **`MemDev`**, verbessern die Verwaltung der Daten und erweitern die Funktionalität.

Aufgabe 3

sie schreiben einen eigenen Treiber für den Parallel-Port und implementieren eine einfache Ampelsteuerung (Hardware für die Ampeln steht zur Verfügung). Für Rechner ohne Parallel-Port stellen wir Ihnen ein Kernel-Modul zur Verfügung, das die Parallel-Schnittstelle emuliert, sowie eine *Software-Ampel*, die mit der emulierten Schnittstelle kommuniziert.

3.2 Aufgabe 1

Der Treiber für **`MemDev`** schreibt Daten (char Buffer) in seinen privaten Kernel-RAM-Bereich. Der bereits vorhandene *Inhalt* des Speichers wird gelöscht und neu geschrieben. Beim Lesen des Gerätes wird der Inhalt ausgegeben, aber nicht gelöscht, d.h. er kann beliebig oft ausgelesen werden.

3.2.1 Vorgehen

a) Legen sie zwei Device Files (char device) im Verzeichnis `/dev` mit `mknod` an (siehe auch Abschnitt 4.1.3 und man-Pages).

- Namen: MemDev0 und MemDev1
- Major-Nummer: 120
- Minor-Nummer: 0 bzw. 1

Setzen sie die Dateizugriffsrechte wie folgt: `chmod 766 /dev/MemDev0` (bzw. `/dev/MemDev1`).

Hinweis: nach einem Neustart des Systems müssen die Device Files erneut angelegt werden.

b) Übersetzen sie den Treiber im Verzeichnis `aufgabe1` mit `make` und ebenfalls das Testprogramm `DriverTest.c` in Verzeichnis `./tests/DriverTest`.

c) Laden sie mit `insmod` das Treibermodul **`MemDriver.ko`** im Verzeichnis `aufgabe1`.

d) Kontrollieren sie mit den nachfolgenden Befehlen, ob und wie das Modul installiert ist:

```
more /proc/devices
lsmod
```

Welche Informationen können sie aus den Befehlen bzw. Files lesen?

e) Testen sie den Treiber mit DriverTest. Das Programm und der Treiber geben einfache Statusmeldungen aus. Sie verstehen die Meldungen, wenn sie den Programmcode des Treibers und des Testprogramms analysieren. Was stellen sie bezüglich zeitlicher Reihenfolge der Meldungen fest? Werden alle Meldungen dargestellt (vor allem die des Treiber)?

Hinweis: **DriverTest** akzeptiert als Parameter den Namen eines Devices, z.B. **/dev/MemDev1** und bzw. oder einen String zwischen " ". Werden keine Parameter übergeben, wird der Default-Device **/dev/MemDev0** verwendet, ebenso ein vordefinierter Text.

f) Testen sie den Treiber zusätzlich mit folgenden Befehlen:

```
ls -l > /dev/MemDev0
cat /dev/MemDev0
ls -l > /dev/MemDev1
cat /dev/MemDev1
```

Was stellen sie fest ? Was sind mögliche Ursachen für diesen Effekt (Hinweis: welche Information liefert der Treiber sehr wahrscheinlich nicht) ?

3.3 Aufgabe 2

Wie sie in Aufgabe 1 festgestellt haben, hat der Treiber einen Schönheitsfehler. In dieser Aufgabe werden wir den Treiber ausbauen und verbessern.

3.3.1 Vorgehen

a) Ändern sie den Treiber so, dass die Daten pro Zugriff (nach dem Öffnen des Devices) nur noch einmal ausgelesen werden, aber trotzdem nicht verloren gehen. Der Treiber muss mit **DriverTest** und den Befehlen **ls** und **more** funktionieren.

Hinweis:
Verwenden sie für die Implementierung den Parameter **loff_t *offset**, der bei jedem Aufruf von `MemRead()` (bzw. dem Filesytem) die entsprechende Offsetposition enthält. Beachten sie, dass sie die neue Offsetposition selbst berechnen und in `offset` zurückgeben müssen.

b) Erweitern sie den Treiber so, dass beim Schreiben auf **/dev/MemDev1** alle Kleinbuchstaben in Grossbuchstaben umgewandelt werden (und auch so gespeichert werden). Denken sie daran, dass sich `/dev/MemDev0` und `/dev/MemDev1` nur in der Minor-Nummer unterscheiden. Implementieren sie dazu eine entsprechende `write()`-Funktionen (z.B. `MemWrite0()` und `MemWrite1()`) und speichern sie beim Öffnen des Devices diese Funktionen in der File Operations Table. Weitere Informationen finden sie in Abschnitt 4.4.2 und in [1].

c) (optional) Erweitern sie den Treiber um die Positionierungs-Funktion `MemSeek()`. Mit dieser Funktion kann die Schreib-/Leseposition beliebig gesetzt werden.

Hinweis: Die neue Position muss einerseits in `filp->f_pos` abgelegt werden und andererseits mit `return` zurückgegeben werden. Wie der Parameter `whence` interpretiert werden muss, finden sie in Abschnitt 4.3.4 und mit **man lseek**.

d) (optional) Erweitern sie den Treiber zusätzlich so, dass auch die Schreibposition frei gewählt werden kann. Zuerst muss dazu der Text ab der aktuellen Position gelöscht und anschliessend durch den neuen Text ersetzt werden.

3.4 Aufgabe 3

In dieser Aufgabe implementieren sie eine einfache Ampel-Steuerung. Dazu verwenden sie eine Ampel, die am Parallel-Port angeschlossen werden kann: die rote, gelbe und grüne LED lassen sich einzeln ansteuern und der Zustand eines Tasters kann abgefragt werden.

Für die Ansteuerung der Ampel-Hardware müssen sie einen einfachen Treiber (`ParDriver.c`) implementieren, der ein Byte auf den Parallel-Port schreiben und ein Byte vom Port lesen kann.

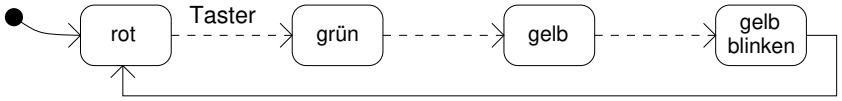
Vorgehen

a) Wir haben im Verzeichnis `aufgabe3/ParPortDriver` ein `Makefile`, das Rahmenprogramm für den Treiber `ParDriver.c` und `io.h` vorbereitet.

Verwenden sie den **`MemDevice`**-Treiber als Vorlage. Schreiben sie die Funktionen `MemWrite()` und `MemRead()` so um, dass ein Zeichen zum Parallel-Port geschrieben (`outb()`), bzw. davon gelesen wird (`inb()`) (siehe Abschnitt 3.4.2). **Hinweis**: da nur ein Byte gelesen bzw. geschrieben wird, ist es nicht notwendig den dynamisch allozierten Buffer zu verwenden.

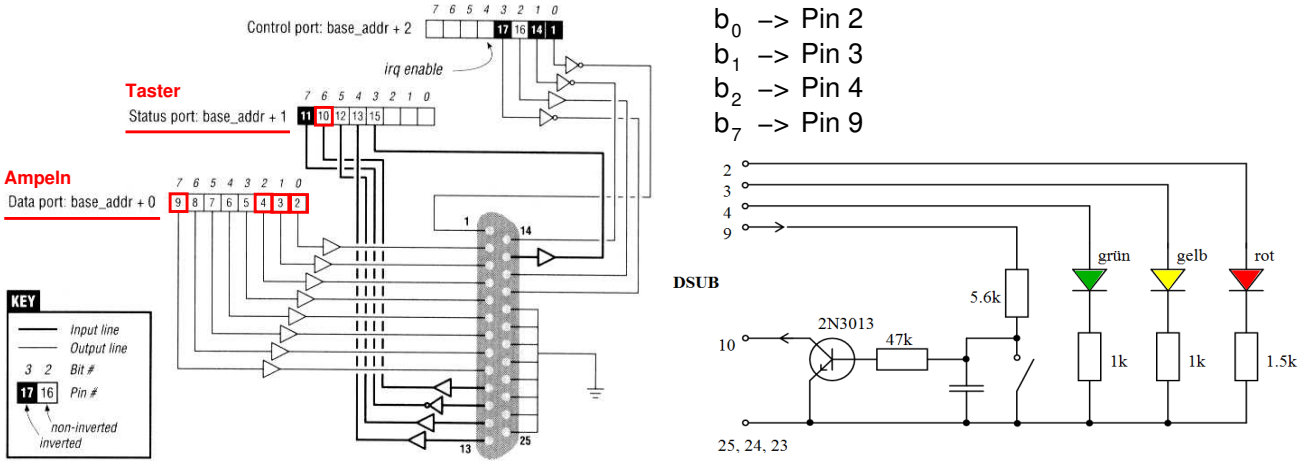
Die Ampel wird am Port **`LP1`** angeschlossen (**`base_addr 0x0378`**): die LEDs am Data Port, der Taster am Status Port (Pin 10: Bit 6 des Status Ports) (siehe Abschnitt 3.4.1) Der Taster kann nur eingelesen werden, wenn Pin 9 (Data Port Bit 7) auf 1 gesetzt wird, das gilt auch für die Port-Emulation.

b) Im Verzeichnis `aufgabe3/ampelController` haben wir einen Programmrahmen für die Ampel-Steuerung vorbereitet. Implementieren sie folgenden Ablauf (verwenden sie als Vorlage das Testprogramm `driverTest.c` im Verzeichnis `tests`.



Zustandsübergänge mit gestrichelten Pfeilen enthalten eine Zeitverzögerung, der Zustand rot wird aufgrund eines gedrückten Tasters verlassen. Das Einlesen des Tasters müssen sie mit Polling realisieren.

3.4.1 Ampelschaltung und das Parallel-Port (Konfiguration: normal)



3.4.2 Parallel-Port Emulation

Die aktuellen Laptops besitzen keine Parallel-Schnittstelle mehr. Wir stellen Ihnen deshalb ein Kernel-Modul zur Verfügung, das diesen Port emuliert, sowie eine Ampel in Java oder C, die mit dem mit dem emulierten Port kommuniziert. Der Treibers kann entwickelt werden wie wenn ein echtes Port verwendet wird, ausser dass für das emulierte Port zusätzlich das mitgelieferte File `io.h` als letztes Header-File eingebunden werden muss. Wie das emulierte Port übersetzt, geladen, getestet und genutzt werden kann finden sie im File **`INSTALLATION`** und **`README`**.

Beachten sie folgendes:

- die Parallel-Port Emulation funktioniert nur mit den 8-Bit IO-Befehlen `inb()` und `outb()`:

```
unsigned char inb(unsigned short portaddr);
void          outb(char data, unsigned short portaddr);
```

- das Modul `ParPortEmul` muss vor dem Modul `ParPortDriver` übersetzt werden, weil das Makefile zum `ParPortDriver` das File `Module.symvers` von `ParPortEmul` benötigt.
- Die Java Ampel wird als jar-Archiv mitgeliefert (`java -jar ampel.jar`), die C Ampel steht als Source-Code mit einem `makefile` zur Verfügung.

4 Linux Treiber für Character Devices

4.1 Device Files

Aus Anwendersicht besteht in Unix-Systemen kein Unterschied zwischen Geräten und Files, d.h. sie können u.a. geschrieben, gelesen, geöffnet und geschlossen werden. Geräte (Devices) sind im Filesystem im Verzeichnis `/dev` eingetragen, z.B. erscheint ein EIDE-Disk als `/dev/sda`.

Zuerst betrachten wird die Einträge im Filesystem, diese Einträge werden Device Files genannt. Mit **`ls -l /dev`** erhalten wir z.B. folgende Ausgabe (nur Auszug):

```
Zugriffs-  Inodes  User Gruppe    Major Minor Erstellungs-    Filename des
rechte                                       Datum          Devices
brw-rw----   1     root  disk       3,   0    Dec 11  2012    sda
brw-rw----   1     root  disk       3,   1    Dec 11  2012    sda1
brw-rw----   1     root  disk       3,   2    Dec 11  2012    sda2
...
crw-rw-rw-   1     root  tty        3, 176    Dec 11  2012    tty
crw-rw-rw-   1     root  tty        3, 177    Dec 11  2012    tty0
```

Die Ausgabe enthält die Zugriffsrechte mit dem Typ für Block- oder Character-Device (b oder c), zwei Identifikationsnummern (Major- und Minor-Nummer) und einen Namen.

4.1.1 Verbindung: Anwender - Treiber - Kernel

Das Betriebssystem stellt die Verbindung zwischen Treiber und Device-File über die Mayor-Nummer her. Der Anwender greift auf den Treiber über den File Namen, z.B. `/dev/tty0`, mit den System Calls `open()`, `read()`, `write()`, `close`, etc. zu.

4.1.2 Major- und Minor-Nummern

Die Major-Nummer [3] ist eine grundsätzlich frei wählbare Nummer im Bereich {0..255}. Bedingung ist, dass noch kein Treiber mit der gleichen Nummer installiert ist. Für Testzwecke stehen drei Bereiche mit Major-Nummern zur Verfügung: {60 .. 63}, {120 .. 127} und {240 .. 254}, sie werden nie für *echte* Treiber verwendet.

Mit den Minor-Nummern (Bereich {0 .. 255}) wird der Zugriff innerhalb eines Treibers geregelt und ist deshalb für das Betriebssystem nicht interessant. Z.B. ist es möglich über die Minor-Nummer 0 auf Parallel-Port 0, über Nummer 1 auf Parallel-Port 1 zuzugreifen, etc. Ähnlich werden mehrere Disks mit verschiedenen Adressen über den gleichen Treiber angesprochen.

4.1.3 Device Files erstellen

Das Device File kann ohne installierten Treiber existieren, das Öffnen mit `open()` des Device Files führt dann aber zu einem Fehler. Die Device Files werden mit dem Befehl **`mknod`** erstellt. Dabei muss der Filetyp (c=char, b=block), die Major- und Minor-Nummer angegeben werden:

```
root# mknod pfad/filename [c|b] majornumber minornumber
```

[3] In den aktuellen Kerneln werden neu aus Gründen der Flexibilität Device Nummern eingeführt, die Mayor- und Minor-Nummern werden jedoch weiterhin unterstützt (siehe z.B. [2]).

4.2 Treiber beim Betriebssystem anmelden und entfernen

4.2.1 Treiber anmelden

Der Treiber bzw. das Modul meldet sich in **init_module()** (siehe auch Abschnitt 2.3.2) mit der Prozedur **register_chrdev()** beim Betriebssystem an:

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

Rückgabewert der Prozedur `register_chrdev()` ist im fehlerfreien Fall eine eine Null für o.k., eine Zahl grösser als Null für eine dynamisch zugeteilte Major Nummer. Bei einem Fehler wird eine Zahl kleiner als Null zurückgegeben. Die Argumente von `register_chrdev()` sind:

major Die gewünschte Major-Number (muss mit Major-Nummer des Device Files übereinstimmen).

***name** Ein frei wählbarer String, der den Namen des Treibers des Devices enthält.

***fops** Ein Zeiger auf eine Struktur, in der die Treiberfunktionen definiert sind.

Soll die Major-Nummer dynamisch zugewiesen werden, muss als Major Nummer die Zahl 0 übergeben werden. Die Prozedur gibt in diesem Fall die zugewiesene (positive) Major-Nummer zurück oder eine negative Fehlernummer.
Erfolgreich registrierte Treiber können mit dem Befehl `more /proc/devices` angezeigt werden.

4.2.2 Treiber entfernen

In der Prozedur **cleanup_module()** (siehe auch Abschnitt 2.3.2) wird der Eintrag in der File Operations Table mit Hilfe der Prozedur **unregister_chrdev()** entfernt:

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

Ein falsch angegebener Name hat unangenehme Folgen: der Treibereintrag wird nicht entfernt, das Modul ist jedoch nicht mehr in den Kernel eingebunden! Der Fehler kann entweder mit einem *Reparaturmodul* behoben werden, das als einzige Aktion in der Prozedur `cleanup_module()` den entsprechenden Eintrag entfernt, ... oder wenn sie etwas Zeit haben: booten sie ihren Rechner neu (wohl die sicherste Methode).

4.3 Treiberfunktionen `open()`, `release()`, `read()`, `write()` und `llseek()`

Wie wir schon gesehen haben, interagiert der Anwender über verschiedenste System Calls mit dem Treiber. Der Treiber muss also Funktioen bzw. Prozeduren für diese Systemaufrufe bereitstellen, die *File Operations* genannt werden (... alle Geräte werden ja wie Files behandelt).

4.3.1 File Operationen

Als Programmierer müssen sie nur die notwendigen Treiberfunktionen implementieren. Die Treiberfunktionen bzw. File Operationen werden in der Struktur **file_operations** eingetragen, (definiert in <linux/fs.h>). Ein Liste mit den wichtigsten Fileoperationen und Parametern finden sie in Abschnitt 4.3.5. Aus Gründen der Einfachheit und um den Treiber portabel zu halten, sollte die in C99 (auch GNU-C) definierte struct-Initialisierung verwendet werden. Angenommen sie haben die folgenden vier Treiber-Prozeduren implementiert:

```
int MyOpen (struct inode* inode, struct file* filp)       {...}
int MyRelease (struct inode* inode, struct file* filp)    {...}
ssize_t MyRead (struct file* filp, char *buf, ...)        {...}
ssize_t MyWrite (struct file* filp, const char *buf, ...) {...}
```

dann können sie die Prozeduren wie folgt in die Struktur file_operations eintragen:

```
struct file_operations MyFileOps = {
        .owner   = THIS_MODULE,
        .open    = MyOpen,
        .release = MyRelease,
        .read    = MyRead,
        .write   = MyWrite,
};
```

4.3.2 Die Funktionen `open()`

Die Funktion **`open()`** wird für die Initialisierung des Treibers beim Öffnen des Gerätes benötigt.
Die wichtigsten Aufgaben der Funktion `open()` sind bzw. können sein:

- überprüfen des Devices auf Fehler (wie `device_not_ready`, Hardwareprobleme, etc.)
- initialisieren des privaten Zeigers `private_data` in der Filestruktur
- bestimmen der Minor-Nummer und ev. neue Treiberfunktions-Tabelle (`f_op`) setzen
- erhöhen und ev. auch abfragen eines *Usage Count* um zu verhindern, dass ein Device mehr als einmal geöffnet werden kann (falls dies nicht möglich ist).

Mit Hilfe verschiedener Funktionen, Macros und den entsprechenden Datenfeldern können Informationen aus dem Kernel gelesen werden.

Die Minor-Nummer kann beim Öffnen dazu verwendet werden, den Treiber unterschiedlich zu initialisieren. So kann z.B. beim seriellen Port anhand der Minor-Nummer die Portadresse gesetzt werden. Die Device Files `/dev/zero` und `/dev/null` zeigen auf den gleichen Treiber: aufgrund der Minor Nummer werden die entsprechenden Funktionen für `read()` und `write()` in die Treiberfunktions-Tabelle eingefügt.

4.3.3 Die Funktionen `release()`

Die Funktion **`release()`** wird beim Schliessen des Treibers durch `close()` aufgerufen und ist dafür verantwortlich, allozierte Ressourcen freizugeben und *aufzuräumen*, z.B. Kernelspeicher freigeben, etc.

Folgendes können Aufgaben der Release-Funktion sein:

- dekrementieren des Usage Counts
- freigeben von Ressourcen, die während der Laufzeit des Devices alloziert wurden. Werden die Ressourcen nicht freigegeben, bleibt z.B. der entsprechende Speicherbereich im Kernel reserviert, bis das Betriebssystem neu gestartet wird!
- falls notwendig das Device bzw. die Hardware ausschalten

4.3.4 Die Funktionen `read()`, `write()` und `llseek()`

Lesen und Schreiben eines Devices beinhaltet auch einen Datentransfer zwischen Treiber (Kernel Space) und Anwendung (User Space). Da die Anwendung ausschliesslich im Virtual Memeory *lebt*, ist es notwendig die Daten zwischen User- und Kernel Space mit speziellen Funktionen zu kopieren:

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

Die Funktion `read()`

Mit der Funktion `read()` werden Daten vom Device gelesen:

```
ssize_t (*read) (struct file* filp, char* buf,
                                    size_t count, loff_t *off);
```

filp Zeiger auf die File Datenstruktur

buf Zeiger auf Buffer im User Space, entspricht dem Pointer **void *to** in `copy_to_user()`

count Anzahl Zeichen, die gelesen werden sollen

Der Rückgabewert gibt an, wie viele Zeichen effektiv gelesen wurden, d.h. er kann auch kleiner als `count` oder sogar Null sein.

Die Funktion `write()`

Mit der Funktion `write()` werden Daten auf das Device geschrieben:

```
ssize_t (*write) (struct file* filp, const char* buf,
                                     size_t count, loff_t *off);
```

filp Zeiger auf die File Datenstruktur

buf Zeiger auf Buffer im User Space, entspricht dem Pointer **void *from** in `copy_to_user()`

count Anzahl Zeichen, die übergeben werden sollen

Der Rückgabewert gibt an, wie viele Zeichen effektiv in den Kernel kopiert wurden, d.h. er kann auch kleiner als `count` oder sogar Null sein.

Anmerkung

wenn die Funktionen `copy_to_user()` oder `copy_from_user()` erfolgreich sind, wird 0 zurückgegeben (also nicht die Anzahl transferiertet Bytes), sonst eine Zahl > 0, nämlich die Anzahl **nicht** transferierter Bytes.

Die Funktion `llseek()`

Mit der Funktion `llseek()` Funktion kann die Lese- bzw. Schreibposition gesetzt werden.

```
loff_t (*llseek) (struct file * filp, loff_t off, int whence);
```

filp Zeiger auf die File Datenstruktur

off Offset zur Berechnung der neuen Position

whence wie der neue Offset berechnet werden soll

- **0 (SEEK_SET)** **off** ist gleich der neuen Read/Write-Position
- **1 (SEEK_CUR)** **off** wird zur aktuellen Position addiert
- **2 (SEEK_END)** **off** wird zur Grösse des Files (End-of-File) addiert

In Klammern sind die Konstanten angegeben, die vom Anwenderprogramm verwendet werden können, sie sind in <unistd.h> definiert. Diese Konstanten können in Treiberprogrammen jedoch nicht verwendet werden!

Die neue Fileposition wird im Treiber in der Variable **filp->f_pos**, zudem gibt die Funktion `llseek()` bei korrektem Aufruf den aktuellen bzw. neuen Offset relativ zum Beginn des Files zurück oder einen entsprechenden Fehlercode (negativer Wert).

Weitere Informationen zu `lseek` finden sie unter `man lseek` (System Call).

4.3.5 Die wichtigsten Fileoperationen und Basis-Typen der Parameter

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file*, const char*, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                       unsigned long, unsigned long);
};
```

struct file *

Pointer auf geöffnetes File → Informationen aus Filestruktur, siehe <linux/fs.h>

loff_t

Typ **long** definiert in <linux/types.h> und <asm/posix_32.h> bzw. <asm/posix_64.h>

ssize_t

Typ **int** definiert in <linux/types.h> und <asm/posix_32.h> bzw. <asm/posix_64.h>

struct inode *

Zeiger auf eine Inode-Struktur, die Inode-Struktur stellt ein File auf dem Disk dar und ist in <linux/fs.h> definiert. Aus den Informationen in der Inode-Struktur kann z.B. die Minor- und Major-Nummer des Device-Files ausgelesen werden.

4.4 Fehler-Codes, Funktionen, Macros und Datenfelder

4.4.1 Fehler-Codes

Die Fehler Codes unter Linux sind immer negativ. Die entsprechenden Konstanten im C-Code sind jedoch positiv definiert, d.h ein Error Code muss wie folgt verwendet werden: -ENODEV, -EBUSY, etc.

4.4.2 Major- und Minor-Nummern

Bei jedem Aufruf einer Treiberfunktion kann aus der Inode-Struktur die Major- und Minor-Nummer bestimmt werden. Zugriff über die Inode-Struktur:

```
struct file * filp;
struct inode * inode;
inode = filp->f_path.dentry->d_inode;
```

Die Major- und Minor-Nummer ist im Feld `i_rdev` der Inode-Struktur abgelegt und darf nur über folgende Macros gelesen werden:

```
MAJOR(inode->i_rdev)    /* gibt die Major-Nummer */
MINOR(inode->i_rdev)    /* gibt die Minor-Nummer */
```

oder alternativ:

```
imajor(inode)           /* gibt die Major-Nummer */
iminor(inode)           /* gibt die Minor-Nummer */
```

4.4.3 Informationen zum aktuellen Prozess

Der Pointer **current** zeigt auf eine Struktur, die Informationen zum aktuellen Prozess enthält (z.B. die Prozess-ID, den Befehlsnamen, etc.).

```
current->pid             /* aktuelle Prozess - ID */
current->comm            /* Kommandoname des aktuellen Prozess */
```

Die Variable **current** ist vom Typ `task_struct` und ist in <linux/sched.h> definiert. Anhand der Definition können alle verfügbaren Informationen abgerufen werden.

5 Kernel-Memory

Im Kernel kann Speicher mit `kmalloc()` reserviert und mit `kfree()` freigegeben werden, die beiden Funktionen sind in <linux/slab.h> definiert:

```
void* kmalloc (size_t size, int priority);
void  kfree (void* ptr);
```

`kmalloc()` alloziert die Anzahl **size** Bytes und gibt einen Zeiger auf den Speicherbereich zurück, bei Fehler wird der NULL-Pointer zurückgegeben. Als Priorität für Speicherallozierung im Kernel wird `GFP_KERNEL` verwendet, weitere Prioritäten sind in <linux/mm.h> definiert (siehe auch [2]). `kfree()` gibt den Speicherbereich auf den **ptr** zeigt wieder frei.

Achtung Beim Entfernen eines Modules ist es sehr wichtig, dass allozierter Speicher freigegeben wird, weil dieser Speicher sonst bis zum Neustart des Betriebssystems nicht mehr verwendet werden kann.