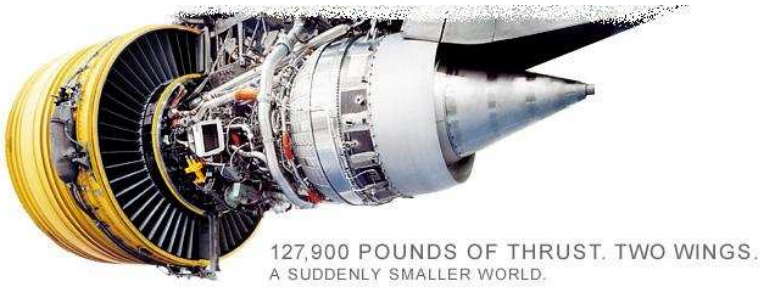
Praktikum TreiberSI

Module, Treiber und Ampeln

Linux Kernel 2.6

Frühlingssemester 2012

M. Thaler, A. Schmid
K. Rege, J. Zeman



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 Organisatorisches

Das Praktikum besteht aus zwei Teilen: einem Theorieteil und einem Praxisteil. Da Sie schon im Theorieteil mit Linux Modulen arbeiten werden, ist es notwendig als erstes **kbuild** und die Linux Kernel Headers zu installieren, falls dies nicht schon geschehen ist. Eine Anleitung dazu finden Sie in **Abschnitt 1.6.**

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

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 Aufbau des Praktikums

Zuerst geben wir Ihnen einen kurzen Überblick zu Linux Modulen (speziell Abschnitt 2.2), die die Basis für die Implementierung von Treibern bilden. Dabei werden Sie einige einfache Übungen mit Modulen machen. In Abschnitt 3 folgen die Aufgabenstellungen, in den Abschnitten 4 und 5 finden Sie die dazu notwendigen theoretischen Grundlagen sowie die grundlegenden I/O- und Kernel-Befehle.

Im ersten Teil entwerfen Sie einen einfachen Treiber, mit dem Sie Daten (Strings) im Kernel abspeichern und wieder auselesen können. Im zweiten Teil werden Sie eine Ampelsteuerung realisieren, die auf einen Treiber für das Parallel-Port zugreift. Für Rechner ohne Parallel-Port steht ein Kernelmoduld zur Verfügung, das ein Parallel-Port emuliert und ein Java-Programm mit dem die Ampel visualisert wird.

1.5 Zu Beachten

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

1.6 Installation der Kernel Header und von kbuild

Für das Übersetzen von Kernelmodulen und Treibern, benötigen Sie einerseits die zu Ihrem Kernel passenden Kernel Headers sowie das entsprechende kbuild (ein Makefile Framework für die Kompilation des Kernels). Ob **kbuild** und die Linux Kernel Headers vorhanden sind, können Sie wie folgt überprüfen:

```
ls /usr/src/linux-kbuild*
lib/modules/`uname -r`   oder   ls /dpkg --list linux-headers*
uname -r
```

Überprüfen Sie auch, ob die Versionen mit der Ausgabe von `uname` übereinstimmen.

1.7 Literatur

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

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

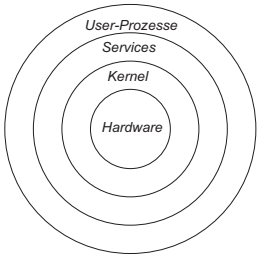
2 Einführung zum Linux-Kernel und zu Modulen

2.1 Allgemeine Übersicht

2.1.1 Betriebsysteme und Schichtenmodell

Das Betriebsystem stellt grundsätzlich die Schnittstelle zwischen Computer Hardware und Benutzern resp. Benutzerprogrammen her. Es verwaltet, steuert und koordiniert die verschiedenen Hardwarekomponenten.

Schichtenmodell unter Linux:



User-Prozesse:	Prozesse, die vom Anwender gestartet werden
Services:	Dienste, die typischerweise zum Betriebsystem gehören: die Shell, die graphische Benutzeroberfläche, die System Call Schnittstelle, etc.
Kernel:	Kern des Betriebsystem
Hardware:	Prozessor, Speicher, I/O-Geräte, etc.

Hauptaufgaben resp. Komponenten des Linux-Kernels

Prozess Scheduler:	Der Scheduler steuert die Zuteilung von Rechnerzeit an die Prozesse.
Interprozess-Kommunikation:	Stellt die Verfahren für die Interprozess-Kommunikation (Signale, Pipes, etc.) zur Verfügung und verwaltet die benötigten Ressourcen.
Memory Management:	Verwaltet die virtuellen Adressräume der einzelnen Prozesse, ist auch verantwortlich für Paging und Swapping.
Virtual File System:	Abstrahiert unterschiedliche Hardwarekomponenten und stellt eine einheitliche Schnittstelle zu den verschiedenen Filesystemen zur Verfügung.
Gerätetreiber:	Stellt eine einheitliche Schnittstelle zur Hardware zur Verfügung.
Netzwerk:	Verwaltet die Netzwerkschnittstellen und -verbindungen.

Aufbau und Funktionalität des Linux-Kernels

Der Linux-Kernel ist grundsätzlich monolithtisch aufgebaut, d.h. dass Hauptfunktionen fest in den Kernel kompiliert sind. Zur Laufzeit können jedoch Module dynamisch von Hand oder automatisch[1] nachgeladen resp. 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.

[1] Für das automatische Laden und Entfernen von Modulen ist der Kernel Deamon **`kerneld`** verantwortlich. Wir werden im folgenden jedoch nicht näher auf diese Möglichkeit eingehen (siehe dazu [1] und [2]).

2.1.2 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).

2.2 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.

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

2.2.1 Modulaufbau

Module besitzen einen definierten Einsprungpunkt und ein definiertes Ende. Das einfachste Modul implementiert dazu zwei entsprechende Prozeduren, hier als Beispiel das "Hello World" Modul[2] im Verzeichnis `hello`:

```
/* HelloWorld.c */

#define MODULE
#include <linux/module.h>
. . .

int init_module(void) {
   printk("<0>*** Hello World from Module ***\n");
    return(0);
}
void cleanup_module(void) {
        printk("<0>*** Good Bye from Module ***\n");
}
```

Beim Laden eines Moduls, wird die Prozedur `init_module` aufgerufen, beim Entfernen die Prozedur `cleanup_module` ausgeführt. Die beide Prozeduren werden für Initialisierungen resp. 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.

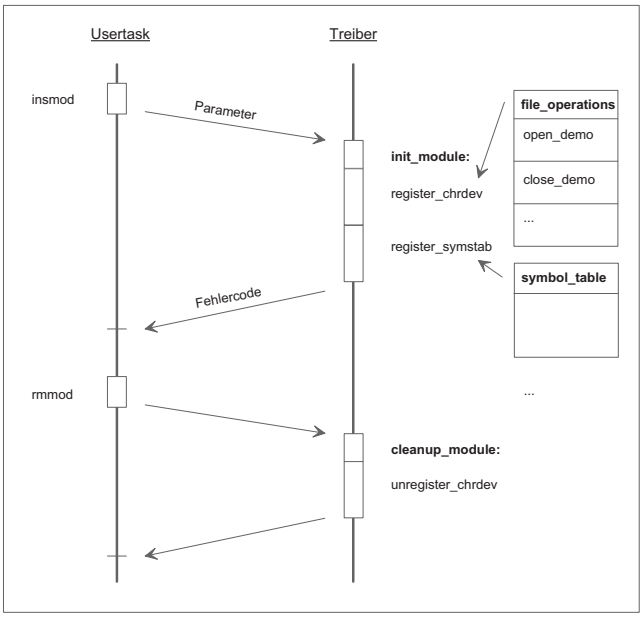
[2] <0> in printk bestimmt die Priorität der Meldung: 0 höchste Priorität, Priorität > 0: ev. wird die Kernelmeldung nicht angezeigt. printk wird sonst wie printf verwendet.

2.2.2 Kompilation von Modulen

Für die Kompilation der Module verwenden wir aus Komplexitätsgründen grundsätzlich Makefiles, wir möchten hier aber aus verschiedenen Gründen nicht auf die Details eingehen. Hier das Beispiel für unser Hello-World Beispiel (Kernelmodule werde in 2 Durchgängen übersetzt, deshlab 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
```

2.2.3 Laden und Entfernen von Modulen



Die beiden Shell-Befehle `insmod` und `rmmod` laden Module in den Kernel resp. entfernen sie wieder. Das einfache "HelloWorld"-Modul finden Sie im Verzeichnis `hello`. Die Module haben als Extension ko (Kernel Object).

Beispiel:

```
root@inpc25# make HelloWorld.c
root@inpc26# insmod HelloWorld.ko
Message from syslogd@diva at Feb 13
16:07:40 ...
 kernel:[ 1970.696170] *** Hello World
from Module ***


root@inpc25# rmmod HelloWorld.ko
Message from syslogd@diva at Feb 13 16:08:07 ...
 kernel:[ 1997.989149] *** Good Bye from Module ***
```

Damit Sie die Kernel-Meldungen beim Laden und Entladen der Module sehen, öffnen Sie am besten ein neues Fenster und geben folgenden Befehl ein (ev. müssen Sie sich als root anmelden):

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

Mit **`lsmod`** können Sie die geladenen Module auflisten.

2.2.4 Setzen von Parametern beim Laden von Modulen

Beim Laden eines Modules können Parameter übergeben werden:

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

Folgende Typen für Parameter sind definiert: `byte`, `short`, `int`, `long`, `string` die Anzahl der Parameter ist dabei beliebig.

Als Parameter können die Namen von globalen Variablen des Modul verwendet werden. Die globalen Variablen werden beim Laden des Moduls auf den entsprechenden Parameterwert gesetzt. Wird die Variable im Modul initialisiert, wird sie 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 Paramtertypen sind im Headerfile `<linux/module.h>` definiert. Die Paremeter Typen müssen mit folgenden Kürzeln definiert werden (Auswahl):

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

Beispiel für ein Modul mit Übergabe von Parametern (im Verzeichnis "paramod")

```
/* ParaModule.c */

#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("<0>*** ParaModule: %d - %s ***\n", globalInt, globalChar);
  return(0);
}

void cleanup_module(void) {
  printk("<0>*** Good Bye from ParaModule ***\n");
}
```

Laden Sie das Modul mit **`insmod`**, wobei der String **keine Leerzeichen** (white spaces) enthalten darf:

```
insmod ParaModul.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 entladen.

2.2.5 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 automatisch erhöht wird. Der Anwender kann dies jedoch auch selbst tun, indem er den Befehl `try_module_get(THIS_MODULE)` beim Eintritt ins Modul ausführt (falls der Rückgabewert = 0 ist, war des Modul nicht mehr geladen). Der Referenzzähler kann mit dem Befehl `module_put(THIS_MODULE)` dekrementiert werden. In den Dateien zum Praktikum finden Sie entsprechende Anwendungen. Weitere Hinweise finden Sie auch in der Literatur.

Achtung: wenn die Referenzzählerverwaltung nicht stimmt, kann das Modul nicht mehr entladen werden.

2.2.6 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 vor dem Einbinden von <linux/module.h> das Symbol __NO_VERSION__ definiert werden.

2.2.7 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 der Headerdatei `<linux/module.h>` definiert, muss zuerst 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 weitere Möglichkeiten, das Kopieren des Files ist jedoch am einfachsten).

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

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

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

3 Aufgaben

3.1.1 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 **`MemDevice`**, das Daten (Strings) speichern kann. Dabei lernen Sie die wichtigsten Grundfunktionen von Treibern anhand verschiedener Beispiele kennen.

Aufgabe 2

Sie erweitern den Treiber für **`MemDevice`**, verbessern die Verwaltung der Daten und bauen zusätzliche Funktionalität ein.

Aufgabe 3

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

3.2 Aufgabe 1

Der Treiber für **`MemDevice`** 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) Installieren Sie die Dateien zum Praktikum und übersetzen Sie den Treiber im Verzeichnis `aufgabe1` mit make:
   - `make`
   - übersetzen Sie auch das Testprogramm `DriverTest.cc` im Unterverzeichnis `test`.

b) Legen Sie zwei Device Files (char device) im Verzeichnis `/dev` mit `mknod` an (siehe auch Abschnitt 4.1.2 und man-Pages)
   - Namen: `MemDev0` und `MemDev1`
   - Major-Nummer: 120
   - Minor-Nummer: 0 resp. 1
   - setzen Sie die Dateizugriffsrechte wie folgt: `chmod 766 /dev/MemDev0"` (und `MemDev1`)

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

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 resp. 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 Testprogrammes 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 resp. oder einen String zwischen " ", wenn keine Angaben gemacht werden, 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 wahrschinlich nicht) ?

3.3 Aufgabe 2

Wie Sie bei Aufgabe 1 gesehen haben, hat der Treiber noch Schönheitsfehler. In dieser Aufgabe werden wir ihn ausbauen und verbessern.

3.3.1 Vorgehen

a) Ändern Sie den Treiber so, dass er nur noch einmal ausgelesen werden kann, sein Inhalt jedoch nicht verloren geht. Der Treiber muss mit `DriverTest` und den Befehlen `ls` und `more` funktionieren.

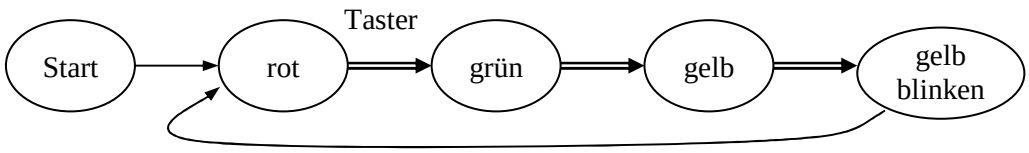
b) Erweitern Sie den Treiber so, dass beim Schreiben auf `/dev/MemDev1` alle Kleinbuchstaben in Grossbuchstaben umgewandelt werden (und auch so gespeichert werden). Denken Sie daran, dass sich `/dev/MemDev0` und `/dev/MemDev1` nur in der Minor-Nummer unterscheiden. Implementieren Sie dazu eine enteprechende `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 auch in [1].

c) Erweitern Sie den Treiber um die Funktion `MemLLseek`. Mit dieser Funktion soll die Leseposition beliebig gesetzt werden können. Beim Auslesen des Strings soll der Positionszeiger um die Anzahl Bytes erhöht werden, die soeben gelesen wurde. Ändern Sie das Programm `DriverTest` so, dass 10 Bytes von **`MemDevice`** gelesen werden und das Gerät bei jedem Lesen zuerst geöffnet und anschliessend geschlossen wird. Dieser Vorgang soll solange wiederholt werden, bis der gesamte Inhalt des Strings gelesen wurde.

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 einen einfachen Treiber (**`ParDriver.c`**), der auf den Parallel-Port schreibt und vom Parallel-Port liest. Als Hilfsmittel verwenden Sie eine einfache Ampel, die am Port angeschlossen wird und bei der Sie die rote, gelbe und grüne LED einzeln ansteuern können. Weiter kann der Zustand eines Tasters eingelesen werden. Ihre Ampelsteuerung soll folgenden Ablauf realisieren:

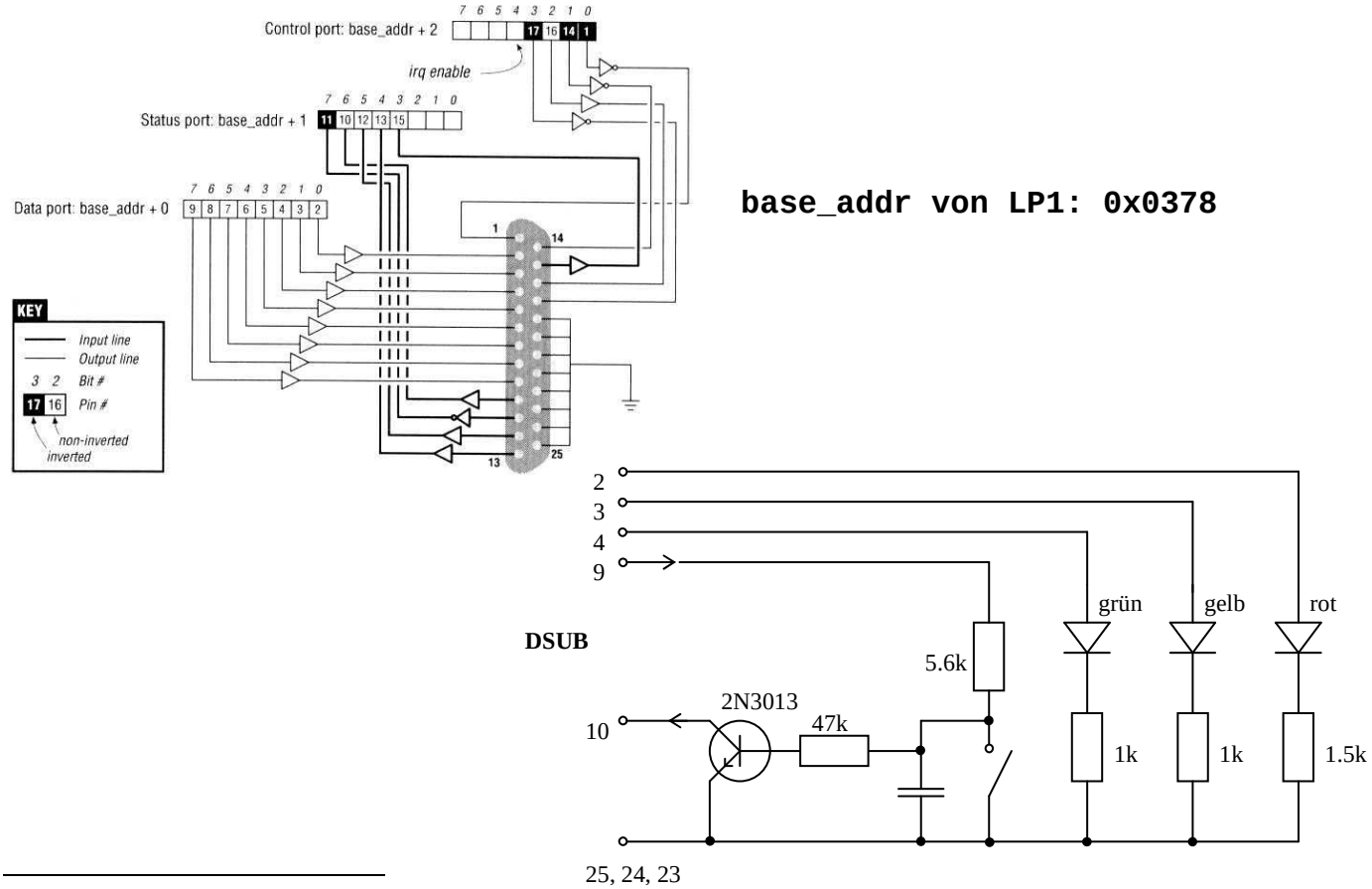


Zustandsübergänge mit doppelten Pfeilen enthalten eine Zeitverzögerung[3], der Zustand rot wird aufgrund eines gedrückten Tasters verlassen.

Vorgehen

Wir haben für Sie das Verzeichnistruktur `aufgabe3/ParPortDriver` mit einem `Makefile`, einem Rahmenprogramm `ParDriver.c` und `io.h` vorbereitet. Verwenden Sie den Treiber für **`MemDevice`** als Vorlage. Schreiben Sie die Funktion `MemWrite` und `MemRead` so um, dass ein Zeichen zum Parallel-Port geschrieben, resp. davon gelesen wird. Das Einlesen des Tasters können Sie mit Polling realisieren. Gehen Sie davon aus, dass die Ampel am Port LP1 angeschlossen ist (`base_addr 0x0378`). Der Taster kann nur eingelesen werden, wenn Pin 9 auf 1 gesetzt wird (Ansteuerung Transistor), gilt auch für Port-Emulation.

3.4.1 Ampels**chaltung und Parallel-Port (Konfiguration: normal)**



[3] Wählen Sie selbst eine geeignete Zeitverzögerung

3.4.2 Parallel-Port Emulation

Da heute die meisten Laptops keine Parallel-Schnittstelle mehr besitzen, stellen wir Ihnen ein Kernel-Modul zur Verfügung, das dieses Port emuliert. Zudem haben wir eine Ampel in Java implementiert, die über das LP1 Port angesteuert werden kann.

Bei der Treiberentwicklung müssen Sie sich nicht darum kümmern, ob das Parallel-Port oder das emulierte Port verwendet wird, ausser dass zum Nutzen des emulierten Ports zusätzlich das mitgelieferte Header-File `io.h` eingebunden werden muss (letztes inlucde-File).

Wie das emulierte Port übersetzt, geladen und genutzt werden kann finden Sie im File INSTALLATION und README. Die Ampel wird als jar-Archiv mitgeliefert (`java -jar ampel.jar`).

Wichtige Hinweise:

- **die Parallel-Port Emulation funktioniert nur mit den 8-Bit IO-Befehlen `inb()` und `outb()`**
- **das Modul `ParPortEmul` muss vor dem Modul `ParPortDriver` übersetzt werden, weil das Makefile zum `ParPortDriver` das File `Module.symvers` von `ParPortEmul` benötigt**

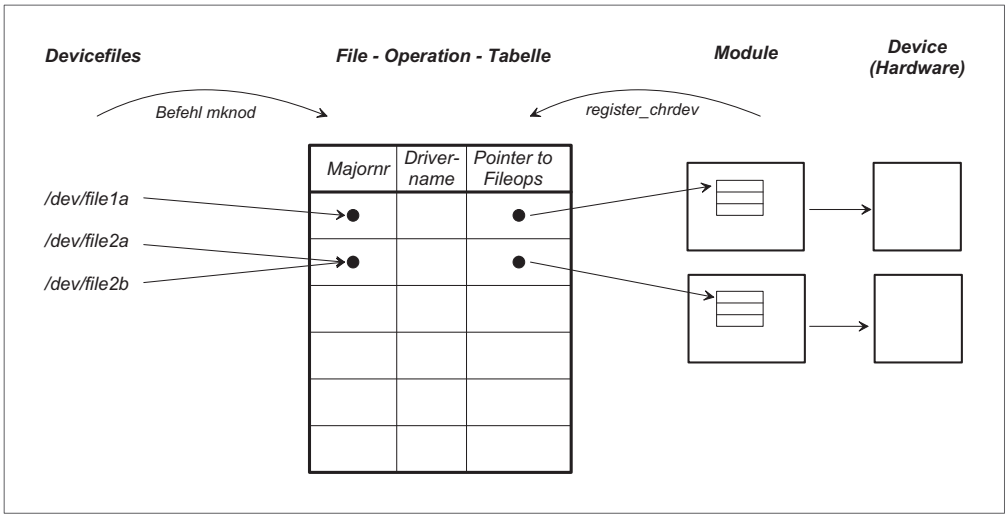
4 Treiber unter Linux

4.1 Treiber für Character Devices

4.1.1 Übersicht

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 daher im Filesystem im Verzeichnis **`/dev`** eingetragen, z.B. erscheint ein EIDE-Disk als **`/dev/hda`**.

Wie kann man nun auf diese Geräte zugreifen und wie wird eine Verbindung zwischen dem Device File, d.h. dem Eintrag im Verzeichnis `/dev` und dem Treiber hergestellt ?



4.1.2 Device Files

Zuerst betrachten wird die Einträge im Filesystem (diese Einträge werden Device Files genannt). Sie enthalten einen Namen, zwei Identifikationsnummern (Major- und Minor-Nummer) und einen Typ (b oder c). Der **Typ** gibt an, ob das entsprechende Gerät ein Block oder Character Device ist.

Mit der **Major-Nummer** stellt das Betriebsystem die Verbindung zum Treiber her. Device Files mit gleichen Major-Nummern zeigen auf den gleichen Treiber!

Die Major-Nummer kann grundsätzlich frei gewählt werden (Bereich 0..255). Bedingung ist jedoch, 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 der **Minor-Nummer** wird innerhalb des Treibers der Zugriff geregelt. So ist es z.B. möglich, über die Minor-Nummer 0 auf Parallel-Port 0, über Nummer 1 auf Parallel-Port 1 zuzugreifen, etc. Verwendet wird diese zusätzliche Aufteilung z.B. bei Disks, die alle mit dem gleichen Treiber, jedoch mit verschiedenen Adressen angesprochen werden.

Die Minor-Nummer interessiert das Betriebsystem nicht und kann vom Treiberentwickler frei gewählt werden; der mögliche Bereich ist 0 .. 255.

Mit dem Befehl `ls -l` im Device-Verzeichnis `/dev` können die Major- und Minor-Nummer der einzelnen Devices angezeigt werden (Spalte 5 = major number, Spalte 6 = minor number).

Auszug aus `/dev`

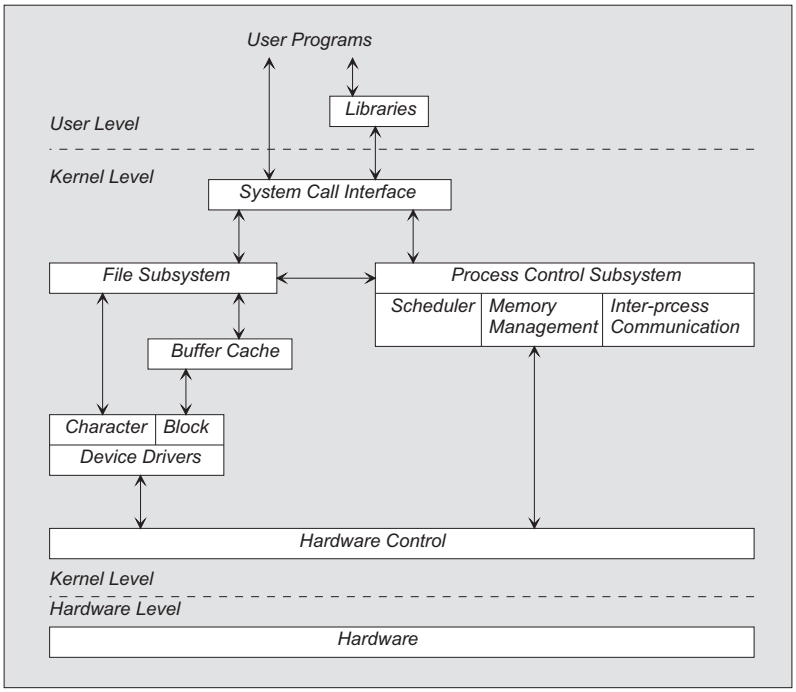
```
Zugriffs-  Inodes  User Gruppe    Major Minor Erstellungs-     Filename des
rechte                                       Datum            devices

brw-rw----   1     root  disk       3,   0    Dec 11  1998      hda
brw-rw----   1     root  disk       3,   1    Dec 11  1998      hda1
brw-rw----   1     root  disk       3,  10    Dec 11  1998      hda10
brw-rw----   1     root  disk       3,  11    Dec 11  1998      hda11
...
crw-rw-rw-   1     root  tty        3, 176    Dec 11  1998      ttya0
crw-rw-rw-   1     root  tty        3, 177    Dec 11  1998      ttya1
crw-rw-rw-   1     root  tty        3, 178    Dec 11  1998      ttya2
```

Das Device File kann alleine existieren (es muss kein entsprechender Treiber installiert sein). Zugriffe auf Geräte, deren Treiber nicht installiert sind, führen jedoch zu einem Fehler, wenn sie mit `open()` geöffnet werden.
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@inpc25% mknod pfad/filename [c|b] majornumber minornumber
```

4.1.3 Treiberaufrufe

Ist der Teiber installiert, können auf dem entsprechenden Device Fileoperationen ausgeführt werden. Diese Operationen sind System-Calls und greifen auf Funktionen zu, die vom Treiber zur Verfügung gestellt werden (müssen).



Was sind System-Calls?

Normalerweise läuft der Prozessor im Benutzermodus. In diesem Modus hat er wenig Privilegien, er kann z.B. nicht auf den Speicher anderer Prozesse zugreifen, einige Prozessorbefehle sind gesperrt und er kann er nicht direkt auf Kernelfunktionen zugreifen.

Der Kernel stellt jedoch eine definierte Anzahl Dienstleistungs-Prozeduren (System-Calls) zur Verfügung. Wird ein System-Call aufgerufen, wechselt der Prozess in den Kernelmodus.

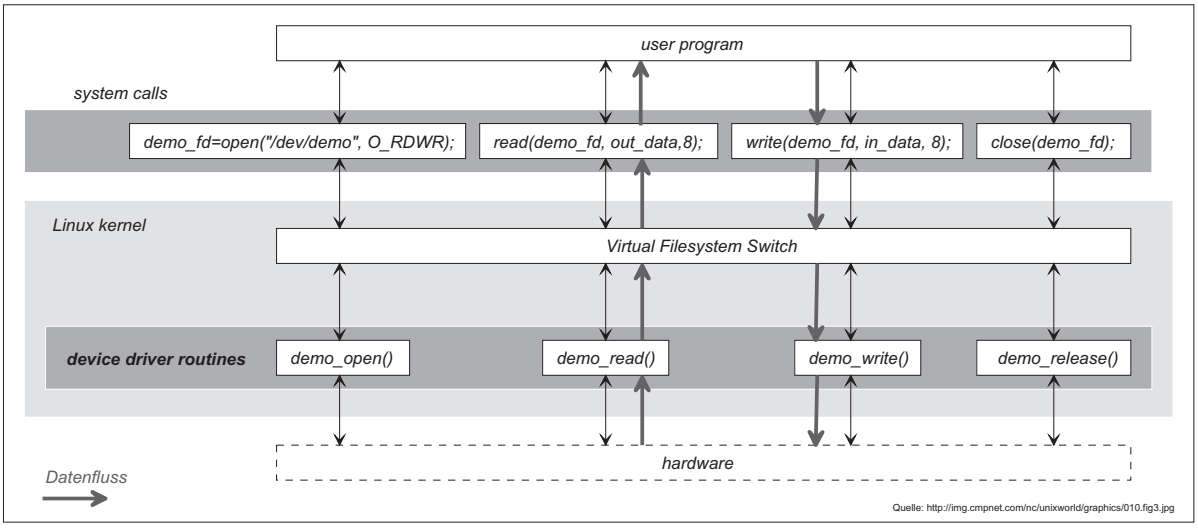
System-Calls haben gesicherte Einsprungstellen in den Kernel, der Zugriff geschieht über Bibliotheken.

Die am meisten verwendeten System-Calls für den Aufruf der Treiberoperationen sind:

```
open (), close(), read (), write ()
```

4.1.4 Treiber ins Betriebsystem einbinden und entfernen (optional)

Was jetzt noch fehlt, ist die Verbindung des Treibers zum Device File. Die Verbindung zwischen dem Device File und den Treiberfunktionen wird vom Virtual File-System mit der sogenannten File Operations Table realisiert.



Der Treiber resp. das Modul wird in der Prozedur `init_module` mit dem Befehl `register_chrdev` beim System angemeldet:

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

Rückgabewert der Prozedur `register_chrdev` ist im fehlerfreien Fall eine Zahl grösser gleich Null , bei einem Fehler eine Zahl kleiner als Null.

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 Nummer die Zahl "0" übergeben werden. Die Prozedur gibt in diesem Fall die (positive) Major-Nummer zurück oder eine negative Fehlernummer.

Erfolgreich registrierte Treiber können mit dem Befehl `more /proc/devices` angezeigt werden.

In der Prozedur `cleanup_module` wird der Eintrag in der File Operations Table mit dem Befehl `unregister_chrdev` entfernt:

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

Wird ein falscher Name angegeben, hat dies unangenehme Folgen: der Treibereintrag wird nicht aus der File Operations Table entfernt, das Modul ist jedoch nicht mehr in den Kernel eingebunden! Das Lesen des Files `/proc/devices` erzeugt eine Fehlermeldung, da der Zeiger auf das Modul ins Leere zeigt! Der Fehler kann nur 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).

4.1.5 Grundgerüst der Treiberfunktionen (optional)

Fileoperationstabelle

Die Fileoperationen resp. Treiberfunktionen für ein Device müssen in eine Struktur `file_operations` eingetragen werden (definiert in `<linux/fs.h>`). Hier die wichtigsten Einträge:

```
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 (*ioctl) (struct inode *, 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);
};
```

Als Programmierer müssen Sie entsprechende Prozeduren implementieren, allerdings nur diejenigen, die für die Ansteuerung des Gerätes wirklich benötigt werden und das sind meist nur `open()`, `release()`, `read()` und `write()`, die restlichen Einträge *müssen* auf NULL gesetzt werden.

Um dies zu vereinfachen und um den Treiber auch portabler zu halten, sollte die in C99 (auch unterstützt von 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, size_t count, loff_t *unnown) {...}
ssize_t MyWrite (struct file* filp, const char *buf, size_t count, loff_t *unnown){...}
```

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,
};
```

Es müssen also nur diejenigen Prozeduren eingetragen werden, die auch wirklich benötigt werden, Sie müssen also weder die Reihenfolge noch alle Einträge in `file_operations` kennen.

Wie Sie gesehen haben, stehen etwa 18 Einträge für die Treiberprogrammierung zur Verfügung, einige werden nie gebraucht, andere sind ein Muss.

Jeder Fileoperation werden Parameter mit unterschiedlichen Typen übergeben, die wichtigsten sind:

`struct file *`	Pointer auf das geöffnete File, die Struktur ist in `<linux/fs.h>` definiert, Informationen aus der Filestruktur werden vom Treiber benötigt
`loff_t`	Typ long (definiert in `<linux/types.h>` und `<asm-i386/posix.h>`)
`ssize_t`	Typ `int` (`<linux/types.h>` und `<asm-i386/posix.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.

Zugriff auf die Inode:
```
struct file * filp; struct inode * inode;
inode = filp->f_dentry->d_inode;
```

Achtung: Die Filestruktur entspricht einem "offenen File" (oder eben auch einem Device). Die Inode-Struktur hingegen stellt ein File auf dem Disk dar (Datei im Filesystem) mit File-Rechten, Inode-Informatioenen, etc.

4.1.6 Die Funktionen `open()` und `release()`

Die wichtigsten Funktionen eines Treiber sind `open`, `release(close)`, `read` und `write`. Wir werden deshalb im folgenden näher auf diese Funktionen eingehen.

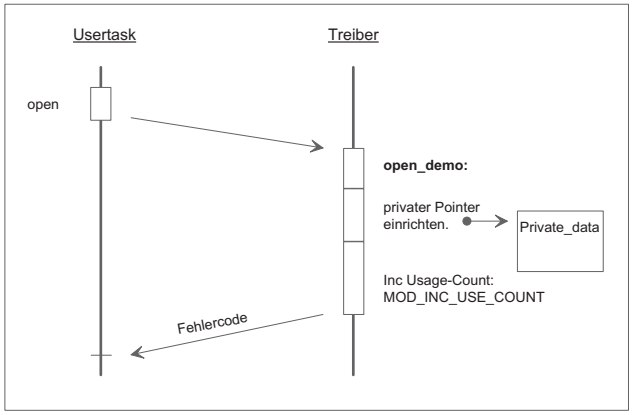
Die Funktion `open()`

Die Funktion **`open()`** wird für die Initialisierung des Treibers beim Öffnen des Gerätes benötigt.

Die wichtigsten Aufgaben der Funktion `open()` sind resp. 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 Treiberoperations-Tabelle (`f_op`) setzen.
- erhöhen und ev. auch testen des Usage Count mit `try_module_get(THIS_MODULE)` (eigentlich geschieht dies automatisch durch den Kernel, kann aber auch manuell gemacht werden)

Mit Hilfe verschiedener Funktionen, Macros und den entsprechenden Datenfeldern können Informationen aus dem Kernel gelesen werden. Eine Auswahl dazu finden Sie weiter unten im Abschnitt "Funktionen, Macros und Datenfelder".



Die Minor-Nummer kann 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 Min

Nummer werden die entsprechenden Funktionen für `read()` und `write()` in der Treiberoperations-Tabelle eingefügt.

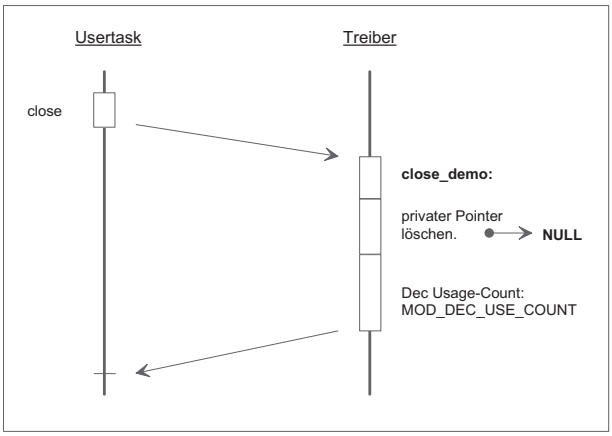
Der Treiber selbst weiss nicht, über welchen Device File Namen er angesprochen wird. Er kennt nur die Major resp. Minor-Nummer des Device Files. D.h. der Anwender kann den Device File Namen beliebig ändern. Tatsächlich sind auch im `/dev` - Verzeichnis für die gleiche Kombination von Minor- / Major-Nummern verschiedene Filenamen vorhanden.

Die Funktion `release()`

Die Funktion `release()` wird beim Schliessen des Treibers von `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 mit der Funktion `module_put(THIS_MODULE)`, wird die Vergessen, kann das Modul auch nicht mehr deinstalliert werden (module in use)!
- Freigeben von Resourcen, die während der Laufzeit des Devices alloziert wurden. Werden die Ressourcen nicht freigegeben, bleibt der entsprechende Speicherbereich im Kernel reserviert, bis das Betriebsystem neu gestartet wird!
- Falls notwendig das Device resp. die Hardware ausschalten.



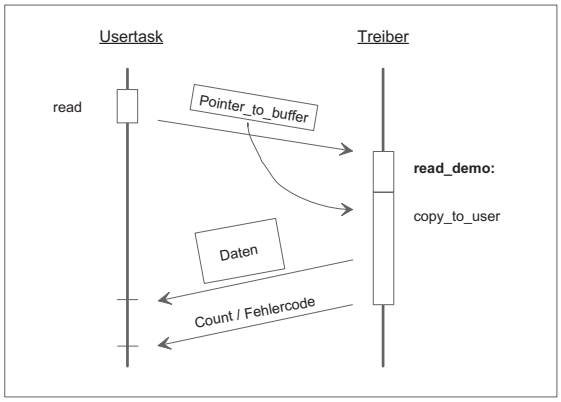
4.1.7 Die Funktionen `read()` und `write()`

Lesen und Schreiben eines Devices ist gleichbedeutend mit einem Datentransfer zwischen Treiber (im Kernel Space) und Applikation (im User Space). Dies kann nicht über "normalen" Zeiger geschehen, da der Prozess zur Zeit dieses Datentransfers im Kernel Space ausgeführt wird, die Daten aber auch im User Space abgelegt verfügbar sein müssen (Zeiger-Operationen werden normalerweise nur im aktuellen "space" ausgeführt). Zudem ist auch nicht sichergestellt, ob der Anwenderprozess überhaupt im Hauptspeicher steht: er könnte z.B. "geswapped" sein.

Eine Lösung bieten spezielle "cross copy" Befehle, die in `<asm/uaccess.h>` definiert sind. Diese Befehle sind für die verschieden Datentypen optimiert (char, short, int, long), für den Transfer ganzer Speicherbereiche stehen die Funktionen `copy_to_user()` und `copy_from_user()` zur Verfügung.

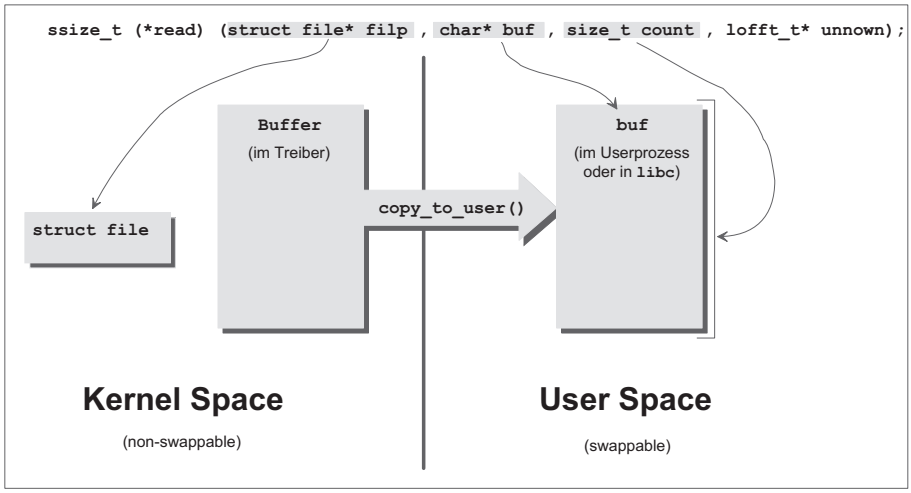
Die Funktion `read()`

An die Funktion `read()` des Treibers übergibt der Kernel den Inode- und File-Pointer, einen Zeiger auf den Datenbereich im User Space und die Anzahl der zu kopierenden Bytes (die zwei letzten Werte werden der `read()` Funktion von der Anwendung übergeben).



Definition der Funktion `read()`:

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



Mit dem Befehl `copy_to_user()` werden die Daten vom Kernel in den User Space kopiert:

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

Der Zeiger `to` wird direkt vom Funktionsaufruf übernommen (der Zeiger `buf`), `from` zeigt auf den treiberinternen Speicherbereich, `len` ist die Anzahl zu kopierender Bytes. Zwischen `count` beim Funktionsaufruf `read` und `len` der Funktion `copy_to_user` besteht folgender Unterschied:

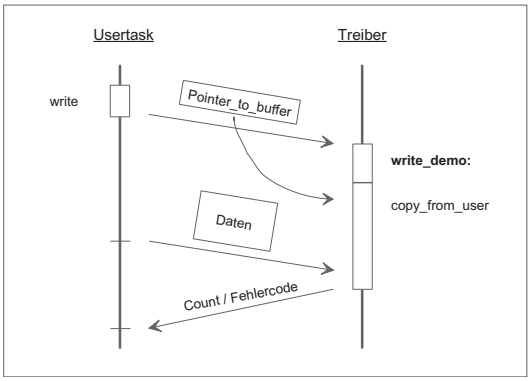
`count` gibt die gewünschte Anzahl zu lesender Bytes an, der Treiber muss aber überprüfen, ob die Anzahl Bytes auch wirklich gelesen werden kann resp. überhaupt vorhanden ist. Die effektive Anzahl Bytes, die gelesen werden kann, muss in `len` der Funktion `copy_to_user` übergeben werden und als Rückgabewert der Funktion `read()` angegeben werden! Der Anwenderprozess wiederum muss überprüfen, ob die angeforderte Anzahl Bytes mit der gelieferten Anzahl Bytes übereinstimmt.

Write-Funktion

Mit der Funktion `write()` werden Daten vom User in den Kernel Space kopiert :

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

Der Zeiger `to` zeigt in diesem Fall in den Kernel Space, `from` wird vom Funktionsaufruf übernommen (entspricht dem Zeiger `buf`).



Auch bei der Funktion `write()` besteht ein Unterschied zwischen `count` des Funktionsaufrufs `write` und `len` der Funktion `copy_from_user`:

`count` (in der Funktion `write`) gibt die gewünschte Anzahl zu schreibender Bytes an, der Treiber muss intern überprüfen, ob diese Anzahl auch wirklich geschrieben werden kann. Die Anzahl Bytes, die effektiv kopiert werden

kann, muss in `len` (in der Funktion `copy_from_user`) sowie als Rückgabewert der Funktion `write` angegeben werden! Der Anwenderprozess wiederum muss selbst überprüfen, ob die gewünschte Anzahl Bytes mit der effektiv geschriebenen Anzahl Bytes übereinstimmt.

Der Befehl `copy_from_user()` transferiert die Daten vom User- in den Kernel Space :

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

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.*

4.1.8 llseek - Funktion (optional)

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

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

Der Funktion `llseek` werden neben `filp` die zwei folgenden Parameter übergeben:

- **`loff_t off`**: Offset zur Berechnung der neuen Position. Der Typ `loff_t` entspricht dem Typ `long` und ist im File `<asm-i386/posix.h>` definiert
- **`int whence`**: Startwert für die Offsetberechnung, `whence` kann drei verschiedene Werte annehmen

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 im File `/usr/include/unistd.h` definiert. Diese Konstanten können in Treiberprogrammen nicht verwendet werden!

Speichern der Fileposition

Die Fileposition wird im Treiber in der Variable `filp->f_pos` (Typ `loff_t`) gespeichert und wird bei Schreib- und Leszugriffen auch von den Funktionen `read()` und `write()` verändert.

Rückgabewert

Die Funktion `llseek()` gibt bei korrektem Aufruf den aktuellen resp. neuen Offset relativ zum Beginn des Files an oder einen entsprechenden Fehlercode (negativer Wert).

Weitere Informationen zu llseek und lseek finden Sie unter:

`man llseek`	System-Call - Aufruf
`man lseek`	Funktionsaufruf lseek

4.1.9 Funktionen, Macros und Datenfelder (optional)

Fehlercodes (Error Codes)

Die Error 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.

Bestimmen der Major- und Minor-Nummer

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

Zugriff auf Inode-Struktur:

```
struct file * filp;
struct inode * inode;
inode = filp->f_dentry->d_inode;
```

Die Major- und Minor-Nummer ist im Feld `i_rdev` der Inode-Struktur gespeichert und ist traditionell vom Typ `dev_t`. Aktuelle Kernel definieren das Feld mit Datentyp `kdev_t`: `kdev_t` ist eine Blackbox, die mit den einzelnen Kernelversionen wechseln kann und in `<linux/kdev_t>` definiert ist, gleichzetig muss auch `<linux/fs.h>` eingebunden sein.

Aus diesem Grund dürfen die Major- und Minor-Nummer nur über die Makros `MAJOR` und `MINOR` gelesen werden:

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

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 */
```

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

5 I/O- und Kernel-Memory-Befehle

5.1 Zugriffe auf I/O-Ports

5.1.1 Byte-Operationen

Mit den folgenden I/O-Befehlen können einzelne Datenwerte zu den Ports geschrieben oder gelesen werden:

- **8 Bit Ports**

```
unsigned char inb (unsigned short Port);
void outb (char Wert, unsigned short Port);
```

- **16 Bit Port Zugriffe**

```
unsigned short inw (unsigned short Port);
void outw (int Wert, unsigned short Port);
```

- **32 Bit Port Zugriffe**

```
unsigned int inl (unsigned short Port);
void outl (long int Wert, unsigned short Port);
```

Diese I/O-Befehle sind im Headerfile `<asm/io.h>` definiert.

5.1.2 Stringoperationen

Mit den folgenden I/O-Befehlen können ganze Speicherblöcke zwischen einem I/O-Port und dem Speicher transferiert werden:

- **8 - Port - Zugriffe**

```
insb(unsigned short Port, void* bufferaddresse, unsigned long count);
outsb(unsigned short Port,const void* bufferaddresse, unsigned long count);
```

- **16 - Port - Zugriffe**

```
insw unsigned short Port, void* bufferaddresse, unsigned long count);
outsw(unsigned short Port, const void* bufferaddresse, unsigned long count);
```

- **32 - Port - Zugriffe**

```
insl(unsigned short Port, void* bufferaddresse, unsigned long count);
outsl(unsigned short Port, const void* bufferaddresse, unsigned long count);
```

Die Befehle kopieren vom oder zum Port die Anzahl Zeichen, die in `count` übergebenen werden. Typ `char`, `short int` oder `int`. Als Quelle resp. Ziel dient der Buffer, auf den der Pointer "`bufferadresse`" zeigt.

5.1.3 Pausing I/O

Wenn der Prozessor zu schnell auf die I/O-Ports zugreift, können Daten verloren gehen. Als Abhilfe müssen beim Zugriff auf die I/O-Ports kurze Pausen eingefügt werden. Dies kann z.B. mit Hilfe von Befehlen, die kleine Pausen einfügen, geschehen:

I/O - Befehle mit Pausen

- `inb_p` (entspricht `inb` mit Pause)
- `outb_p` (entspricht `outb` mit Pause)
- `inw_p` (entspricht inw mit Pause)
- etc.

Die Befehle benötigen die gleichen Parameter wie die I/O-Befehle ohne Pause.

Weiter Infos zu Delays finden Sie im Mini-HOWTO zu IO-Ports:

`http://www.faqs.org/docs/Linux-mini/IO-Port-Programming.html`

5.2 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 Bytes in `size` 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. `kfree` gibt den Speicherbereich auf den `ptr` zeigt wieder frei.

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