NTB

Interstaatliche Hochschule
für Technik Buchs

FHO Fachhochschule Ostschweiz

Einführung in die Linux Treiberentwicklung

Modul:	Ingenieurinformatik III
Kurs:	Betriebssysteme
Autoren:	Urs Graf, Andreas Kalberer, Martin Züger
Version	1.3
Datum	6.11.2015

Inhaltsverzeichnis

1. Einführung **3**

1.1. Einleitung 3
1.2. Kursinhalt 3
1.3. Die Entwicklungsumgebung 4

2. Grundlagen **5**

2.1. Aufbau des Linux Kernels 5
2.2. Kernelspace und Userspace 6
2.3. Module und Gerätetypen/Klassen 7
2.4. Gerätedateien 8
2.5. Virtuelle Dateisysteme und Kernelschnittstellen 10
2.6. Treiber aus Sicht einer Applikation 11
2.7. Make zum Erstellen von Kernelmodulen verwenden 12
2.8. Ein erstes Kernelmodul 14
2.9. Kernelmodule im Vergleich zu herkömmlichen Applikationen 15

3. Treiberentwicklung **17**

3.1. Initialisierung 17
3.2. Fehlerbehandlung während der Initialisierungsphase 18
3.3. Aufräumen 18
3.4. Allozieren und Freigeben von Gerätenummern 19
3.5. Datei-Operationen 20
3.6. Die file Struktur 23
3.7. Die inode Struktur 24
3.8. Zeichengeräte erzeugen und registrieren 24
3.9. Automatisches Erstellen von Gerätedateien 25
3.10. Open / Release 27
3.11. Read 28
3.12. Write 28
3.13. I/O Control 29

4. Cross Development **31**

4.1. Die Zielplattform 31
4.2. Cross Toolchain 33

5. Zugriff auf Hardware **35**
5.1. Hardware-Ressourcen reservieren 35
5.2. Zugriff auf Hardware Ressourcen 36
5.3. Treiber für GPIO, Version 1 37
5.4. Treiber für GPIO, Version 2 39

6. Literaturverzeichnis **40**

A. Anhang **41**
A. Zusätzliche Informationen zum OMAP Board 41
B. Operationen der Struktur file_operations 43

1. Einführung

1.1. Einleitung

Wenn man heute von Linux spricht, denken die meisten an ein Desktopbetriebssystem wie z.B. Ubuntu Linux. Das obwohl weltweit nur auf ca. 2% aller Desktop-Computer eine Linux-Distribution als Betriebssystem eingesetzt wird. Dabei ist Linux in anderen Bereichen viel präsenter. So laufen z.B. 60% aller Webserver im Internet unter GNU/Linux oder die meisten der schnellsten Supercomputer der Welt.
Auch im Endverbraucher-Segment ist Linux sehr verbreitet und zwar auf Smartphones. Android von Google, welches einen Marktanteil von über 80%[1] hat, basiert auf dem Linux Kernel. Aber auch weniger bekannte Smartphone-Betriebssysteme wie Firefox OS (Mozilla), Bada (Samsung) oder Sailfish OS (Jolla) verwenden den Linux-Kernel. Auch auf vielen anderen Geräten wie z.B. Router, Firewalls, NAS, Smart-TVs, etc. wird häufig ein auf Linux basierendes Betriebssystem eingesetzt.

Durch diesen sehr weiten Einsatzbereich wurden für den Linux-Kernel in den letzten Jahren tausende Treiber für die unterschiedlichste Hardware entwickelt. Und es werden mit jeder Version mehr. In diesem Kurs werden wir uns die Entwicklung eines solchen Treibers ansehen.

1.2. Kursinhalt

In diesem Kursteil erhalten Sie einen Einstieg in die Treiberentwicklung für Linux. Dazu werden wir als erstes die Grundlagen besprechen: Aufbau des Kernels, Schnittstelle zwischen Userspace und Kernelspace und der Übersetzungsprozess. Anschliessend werden wir in die Treiberentwicklung einsteigen und einige Pseudotreiber erstellen. Zum Schluss werden wir einen Gerätetreiber für eine echte Hardware schreiben. Dafür verwenden wir ein Embedded System mit einem ARM-Prozessor wodurch wir uns noch mit dem Thema *Cross Development* befassen werden.

[1] 84.6% gemäss Strategy Analytics im 2. Quartal 2014

1.3. Die Entwicklungsumgebung

Für diesen Kursteil benötigen Sie eine Linux-Installation, die verwendete Distribution spielt keine Rolle. Die Installation kann in einer virtuellen Maschine oder auch nativ erfolgen. Sie benötigen folgende Entwicklungswerkzeuge:

- GNU C Compiler (gcc)
- GNU Make
- Source Code oder Headerdateien des verwendeten Kernels
- Ncurses Bibliothek inkl. Header Dateien (libncurses5-dev)

Später, wenn wir Treiber für ein ARM-Board entwickeln, brauchen wir noch einen passenden Crosscompiler und einen für diese Hardware angepassten Linux-Kernel, dazu jedoch mehr in Kapitel 4.

2. Grundlagen

2.1. Aufbau des Linux Kernels

Auf einem UNIX-System erledigen mehrere gleichzeitig laufende Prozesse unterschiedliche Aufgaben. Die meisten dieser Prozesse fordern Systemressourcen an. Dies kann einfach nur Rechenzeit oder Speicher sein, es können aber auch Netzwerkverbindungen oder ganz andere Ressourcen sein. All dies wird vom Kernel bearbeitet und zur Verfügung gestellt. Dieser ist ein relativ grosser und komplexer "Haufen Code", der sich nicht ganz einfach in einzelne Aufgaben aufteilen lässt, denn vieles kann nicht klar voneinander getrennt werden. Der Linux-Kernel übernimmt die folgenden Rollen:

Prozessverwaltung (Process Management) Der offensichtlichste Teil der Prozessverwaltung ist wohl das Scheduling sowie das Erstellen und wieder Zerstören von Prozessen. Dazu gehört aber auch die Kommunikation zwischen einzelnen Prozessen (IPC, Inter Process Communication).

Speicherverwaltung (Memory Management) Der Linux-Kernel erstellt für jeden einzelnen Prozess, der gestartet wird, einen virtuellen Adressraum und limitiert somit die Ressourcen für einen Prozess.

Dateisysteme (Filesystems) Dateisysteme nehmen in einem UNIX eine zentrale Rolle ein. Denn unter UNIX kann so ziemlich alles als Datei angesehen werden. Aber dazu später noch mehr. Der Linux-Kernel unterscheidet sich im Bereich Dateisysteme von anderen UNIX-Kernel insbesondere durch die überaus grosse Anzahl unterstützter Dateisysteme.

Gerätesteuerung (Device Control) Viele Systemoperationen werden am Ende auf ein physikalisches Gerät abgebildet. Kernel-Code der spezifisch für ein bestimmtes Gerät entwickelt wurde, heisst Gerätetreiber. Nicht dazu zählen Prozessor und Speicher, da spricht man von architekturspezifischem Code. Dieser Leitfaden soll eine Einführung in die Entwicklung solcher Gerätetreiber geben.

Netzwerkbetrieb (Networking) Ein weiterer wichtiger Bereich ist der Netzwerkbetrieb. Da die meisten Netzwerkoperationen nicht spezifisch für einen Prozess sind, muss auch diese Aufgabe vom Kernel übernommen werden. Dazu gehören sowohl das Sammeln, Identifizieren und Weiterreichen von Paketen an den richtigen Prozess

Aufgabe 1: Entwicklungsumgebung vorbereiten

a) Installieren Sie die Headerdateien zum aktuell verwendeten Kernel. Sie können sich die genaue Version des laufenden Kernels mit folgendem Befehl anzeigen lassen: `uname -r`. Wenn Sie Debian oder Ubuntu verwenden, können Sie das passende Paket folgendermassen installieren: `$ sudo apt-get install linux-headers-$(uname -r)`

b) Für alle Programme und Makefiles in diesem Skript können Sie einen beliebigen Texteditor benützen und auf der Kommandozeile arbeiten. Ich empfehle Ihnen eine Entwicklungsumgebung zu benutzen, z.B. *KDevelop*. Starten Sie *KDevelop* und dort drin eine neue *Session*. Für die folgenden Aufgaben erzeuge ich pro Aufgabe ein eigenes *CMake*-Projekt. Diese Projekte lege ich in einem separaten Verzeichnis ab, z.B. *∼/DriverDev*. *CMake* erzeugt die notwendigen Makefiles selber.

c) Überprüfen Sie die Installation von `gcc` und `make` indem Sie das folgende Programm übersetzen und ausführen. In *eclipse* benutzen Sie dazu ein *Managed Project*. Das Makefile wird dabei automatisch erzeugt.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int dev = open("/dev/stdout",O_WRONLY);
    if(dev != -1) {
        write(dev,"Hello World!\n",13);
        close(dev);
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Studieren Sie den Programmcode, was passiert hier?

2.2. Kernelspace und Userspace

Eine Aufgabe des Betriebssystems ist es, einen konsistenten Blick auf die Hardware des Computers zu gewährleisten. Es muss ausserdem dafür sorgen, dass Programme unabhängig voneinander

ander ablaufen und dass ein unerlaubter Zugriff auf Ressourcen verhindert wird. Dies ist jedoch nicht ganz einfach und erfordert, dass die CPU eine Trennung von Systemsoftware und Applikationen erzwingt. Jeder moderne Prozessor ist dazu in der Lage. Dazu werden verschiedene Betriebsebenen in der CPU selbst implementiert. Diese Ebenen erlauben nur ein bestimmtes Set von Operationen. Ausgeführter Code kann nur durch eine begrenzte Anzahl "Tore"von einer dieser Ebenen auf eine andere gelangen. Der Linux-Kernel verwendet dieses Hardwarefeature, benötigt jedoch nur zwei solcher Ebenen. Der Kernel selbst läuft im sogenannten Supervisor-Mode, in welchem alles erlaubt ist. Eine normale Anwendung hingegen läuft auf der niedrigsten Ebene, im User-Mode. In diesem werden unerlaubte Zugriffe auf die Hardware oder in den Speicher verhindert.

Im Zusammenhang mit Software wird hingegen von Userspace und Kernelspace gesprochen. Dies bezieht sich auf die verschiedenen Speicherabbildungen und die damit verbundenen unterschiedlichen Adressräume. Unter UNIX und somit auch unter Linux findet der Wechsel zwischen Userspace und Kernelspace über Systemaufrufe und Hardwareinterrupts statt.

Der Kernel, und damit auch alle Module, laufen (wie man am Namen leicht erraten kann) im Kernelspace, während normale Anwendungen im sogenannten Userspace ablaufen.

2.3. Module und Gerätetypen/Klassen

2.3.1. Module

Der Linux-Kernel ist sehr umfangreich, so besteht beispielsweise die Version 3.6 aus rund 15.9 Millionen Zeilen Quellcode. Er bringt Treiber für fast jede nur erdenkliche Hardware mit und es werden Dutzende von Dateisystemen, sämtliche relevanten Netzwerkprotokolle und viele weitere Funktionen mitgeliefert. Von all dieser Funktionalität wird normalerweise nur ein Bruchteil verwendet. So benötigt man z.B. als Endanwender lediglich einen Treiber für die eigene Netzwerkkarte und nicht auch Treiber für alle anderen. Damit nun nicht für jeden Computer ein spezifischer, nur die notwendigen Treiber enthaltender Kernel kompiliert werden muss, werden sogenannte Kernelmodule eingesetzt. Ein solches Modul erweitert den Kernel um eine bestimmte Fähigkeit. So kann der Kernel beispielsweise um die Fähigkeit erweitert werden eine bestimmte Hardware zu nutzen. Man spricht in diesem Fall von einem Treibermodul. Ein Modul kann aber auch ein neues Dateisystem implementieren oder dem Kernel ein neues Netzwerkprotokoll beibringen. Module können während des Betriebs geladen und wieder entladen werden. Die Handhabung von Kernelmodulen wird durch verschiedene Kommandozeilentools erleichtert.

2.3.2. Geräteklassen

Bei Linux wird zwischen drei Geräteklassen unterschieden. Es gibt Zeichengeräte (char devices), wie zum Beispiel eine serielle Schnittstelle, Blockgeräte (block devices), die über ein Dateisystem verfügen und Netzwerk Schnittstellen (networking devices) für den Austausch von

Daten zwischen Systemen. Zeichengeräte und Blockgeräte werden unter Linux als Bytesströme angesprochen. Im System werden sie wie herkömmliche Dateien angezeigt und auch gleich verwendet, dazu aber später noch mehr. Der einzige Unterschied zwischen Zeichen- und Blockgeräten liegt in der Verwaltung der Daten im Kernel.

Netzwerk Schnittstellen repräsentieren normalerweise an das System angeschlossene Hardware, können aber auch ein reines Software-Gerät sein. Netzwerkschnittstellen unterscheiden sich komplett zu Zeichen- und Blockgeräten und können nicht über eine Datei angesprochen werden. Aus zeitlichen Gründen werden wir uns in diesem Kurs nur mit Zeichengeräten beschäftigen.

2.4. Gerätedateien

2.4.1. Unix: Alles ist eine Datei

Wie bereits erwähnt, setzen UNIX-Systeme sehr stark auf das Konzept der Dateisysteme. Unter UNIX (und somit auch unter Linux) gibt es sogenannte Gerätedateien. Das sind spezielle Dateien, die eine einfache Kommunikation zwischen dem Userspace und dem Kernel - und damit letztendlich mit der Hardware eines Computers - ermöglichen. Unixoide Systeme unterscheiden die folgenden drei Typen von Gerätedateien:

- character devices: zeichenorientierte Geräte (c)
- block devices: blockorientierte Geräte (b)
- socket devices: socketorientierte Geräte (s)

Um was für einen Typ es sich bei einer Gerätedatei handelt, kann mit dem Commando `file` herausgefunden werden:

```
$ file /dev/log
/dev/log: socket

$ file /dev/ttyS0
/dev/ttyS0: character special

$ file /dev/sda
/dev/sda: block special
```

Wird ein Verzeichnisinhalt mit `ls -l` ausgegeben, können Gerätedateien ebenfalls erkannt werden:

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 2010-12-08 11:37 /dev/sda
brw-rw---- 1 root disk 8, 1 2010-12-08 11:37 /dev/sda1
brw-rw---- 1 root disk 8, 2 2010-12-08 11:37 /dev/sda2
brw-rw---- 1 root disk 8, 5 2010-12-08 11:37 /dev/sda5
```

Die gesuchte Information steckt im ersten Zeichen einer Zeile: `b` steht für Blockgeräte, `c` für Zeichengeräte und `s` für Socketdateien. Normale Dateien werden mit einem Bindestrich (-) markiert, während Verzeichnisse mit `d` und symbolische Links mit `l` gekennzeichnet werden.

Der *Filesystem Hierarchy Standard* schreibt vor, dass sich die Gerätedateien im Verzeichnis `/dev` befinden müssen. Dies ist bei den üblichen GNU/Linux-Distributionen auch der Fall.

Die benötigten Gerätedateien werden auf modernen GNU/Linux-Systemen beim Booten automatisch durch ein Programm namens *udev* erstellt. Natürlich können solche Dateien aber auch von Hand angelegt werden. Dazu wird das Kommandozeilentool `mknod` verwendet. Die Syntax dieses Kommandos ist:

```
mknod [OPTION]... NAME TYPE [MAJOR MINOR]
```

Als Typ kann `c` für ein Zeichengerät oder `b` für ein Blockgerät eingesetzt werden. Mit den Major- bzw. Minornummern befassen wir uns später noch im Detail. Die möglichen Optionen und eine detaillierte Beschreibung dieses Tools finden Sie wie gewohnt in den Manpages.

2.4.2. Major- und Minornummer

Wir wissen, dass über eine Gerätedatei mit dem Kernel und somit auch mit der darunterliegenden Hardware kommuniziert werden kann. Nur woher weiss der Kernel, was er machen soll, wenn wir eine bestimmte Gerätedatei ansprechen? Wenn wir z.B. die Datei */dev/ttyS0* öffnen und etwas hinein schreiben, erwarten wir, dass dieser Text über die erste serielle Schnittstelle ausgegeben wird. Dabei muss der Kernel aber einerseits wissen, welcher Treiber dafür verantwortlich ist, und auch welche der seriellen Schnittstellen gemeint ist.

Genau für diese Zuordnung sind die Major- und Minornummern zuständig. Die Majornummer gibt an, in welchem Treiber die nötigen Funktionen für die Ansteuerung eines Geräts implementiert worden sind. Über die Minornummer hingegen wird definiert, welche Funktionen für das entsprechende Gerät zuständig sind. Somit ist es möglich in einem Treiber die Funktionen für mehrere Geräte zu implementieren. Dies ist z.B. bei den seriellen Schnittstellen der Fall:

```
$ ls -al /dev/ttyS*
crw-rw---- 1 root dialout 4, 64 2010-12-08 11:37 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 2010-12-08 11:37 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 2010-12-08 11:37 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 2010-12-08 11:37 /dev/ttyS3
```

In diesem Beispiel ist zu erkennen, dass die Gerätedateien für alle vier Schnittstellen die Majornummer 4 tragen. Die Minornummer hingegen unterscheidet sich (64 bis 67). Hier wird also die Minornummer verwendet, um unterscheiden zu können, welche der vier Schnittstellen nun gemeint ist.

Wenden wir uns nun der Kernelseite zu. Die Major- und Minornummer werden vom Kernel in der Struktur vom Typ *dev_t* (welche in *linux/types.h* definiert ist) abgelegt. Seit der Kernelversion 2.6.0 ist dieser Datentyp 32 Bit breit, wobei für die Majornummer 12 Bit und für die Minornummer 20 Bit reserviert sind. Bei der Programmierung eines Treiber sollten jedoch nie Annahmen über die interne Organisation der Major und Minornummer gemacht werden. Besser ist es mit Hilfe von Makros den Typ *dev_t* in Major- und Minornummer umzuwandeln und umgekehrt. Falls dem Programmierer nur der *dev_t* Typ bekannt ist, kann er über das Makro *MAJOR(dev_t dev)* die Majornummer und über *MINOR(dev_t dev)* die Minornummer herausfinden. Sind ihm jedoch die Major- und Minornummern bekannt, kann er diese über *MKDEV(int major, int minor)* in den Typ *dev_t* umwandeln.

Aufgabe 2: Gerätedateien
Erstellen Sie mit Hilfe von *mknod* unter */dev* eine neue Zeichengerätedatei (Typ c) mit dem Namen *myTestDev*. Verwenden Sie als Majornummer 240 und als Minornummer 0. Überprüfen Sie anschliessend, ob die Gerätedatei mit den gewünschten Eigenschaften erstellt worden ist.

Hinweis: Was bewirkt die Gerätedatei?
Mit *mknod* wird nur eine Gerätedatei mit wählbaren Major / Minornummer angelegt. Dies passiert aus dem Userspace (hier aus einer Shell). Es ist ist so ohne weiteres möglich, die gleichen Nummern mehrfach zu verwenden. Der Kernel kann mit diesen Dateien vorerst noch gar nichts anfangen. Wir werden später in unserem eigenen Treiber diese Gerätedateien aus dem Kernelspace erzeugen.

2.5. Virtuelle Dateisysteme und Kernelschnittstellen

Ein virtuelles Dateisystem ist eine Abstraktionsschicht oberhalb der eigentlichen Dateisysteme. Es bietet eine einheitliche API für unterschiedliche Dateisysteme. Bei diesen kann es sich um klassische Dateisystem für einen Datenträger handeln wie beispielsweise ext4 oder NTFS aber auch um solche die keine eigentlichen Dateien auf einem Datenträger repräsentieren wie z.B. *procfs* oder *sysfs*. Solche Dateisysteme dienen als Schnittstelle zum Kernel und können für den Datenaustausch mit diesem verwendet werden.

2.5.1. procfs

Das *proc* Filesystem ermöglicht es, Informationen über Kernelsubsysteme, wie zum Beispiel Speichernutzung, angeschlossene Peripherie usw. zu erhalten. Ebenfalls kann das Verhalten des Kernels teilweise gesteuert werden, ohne dass die Quellen neu kompiliert werden müssen. Weiter ist es darüber möglich, Kernelmodule zu laden oder einen Neustart des Systems auszulösen.

2.5.2. sysfs

Das *sysfs* ist ein virtuelles Filesystem, mit dem Informationen über Kernelobjekte in den Userspace übergeben werden können. Dabei können nicht nur Informationen über Geräte und Treiber abgerufen werden, sie können darüber auch aus dem Userspace konfiguriert werden. *sysfs* ist im Gegensatz zu *procfs* stark hierarchisch geschachtelt und nicht dazu ausgelegt von Personen direkt gelesen zu werden. Im *sysfs* gibt es auch rein binäre Schnittstellen, das heisst, sie liegen nicht mehr in ASCII Textform vor.

Eine gute Übersicht bezüglich Kernelschnittstellen wie*procfs* und *sysfs* bietet das Kapitel 2 in "Kernel Space - User Space Interfaces" von Ariane Keller[2].

2.5.3. udev

Mit *udev* werden im Linux-Kernel die Device Nodes dynamisch verwaltet. Seit der Kernel Version 2.6 ersetzt *udev* die frühere Implementation *devfs* und *hotplug*. *udev* ist für die Verwaltung des */dev* Verzeichnisses verantwortlich und ist im Userspace angesiedelt. Ebenfalls werden alle Aktionen, die aus dem Userspace ausgelöst werden, damit realisiert. Solche Aktionen sind zum Beispiel das Hinzufügen und Entfernen von Geräten und Treibern. *udev* baut auf dem oben erwähnten *sysfs* auf, welches die Geräte aus dem Kernel im Userspace sichtbar macht. Wenn zum Beispiel ein Gerät hinzugefügt wird, werden Kernel-Events ausgelöst, die dann *udev* im Userspace darüber informieren.

2.6. Treiber aus Sicht einer Applikation

Aus der Sicht einer Applikation bestehen zwischen einer normalen Datei und einer Gerätedatei kaum Unterschiede. Ein Zeichengerät muss wie eine Datei vor dem eigentlichen Zugriff geöffnet und nach erledigter Arbeit wieder geschlossen werden. Ein Treiber kann jedoch im Gegensatz zu herkömmlichen Dateien noch weitere Funktionen, wie zum Beispiel einen I/O-Control Aufruf anbieten. Das Programmbeispiel (Listing 1) soll die Handhabung von Gerätedateien verdeutlichen.

Listing 1: Beispiel Verwendung einer Gerätedatei

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5
6  int main(void) {
7      int dev=open("/dev/stdout",O_WRONLY);    /* 1) */
8      if(dev != -1){                           /* 2) */
9          write(dev,"Hello World!\n",13);      /* 3) */
10         close(dev);                          /* 4) */
```

[2]Online zu finden unter http://people.ee.ethz.ch/ arkeller/linux/kernel_user_space_howto.html

```
11          return EXIT_SUCCESS;
12      }
13      return EXIT_FAILURE;
14  }
```

1) Gerätedatei öffnen (nur schreibbar). *open* liefert einen Dateidescriptor (einfach eine Ganzzahl) zurück. Ist dieser positiv, war das Öffnen erfolgreich.

2) Prüfen ob Datei erfolgreich geöffnet wurde.

3) Die Zeichenfolge "Hello World!" in die Datei schreiben.

4) Datei wieder schliessen.

Aufgabe 3: Zugriff auf Files

Lesen Sie in den Manpages zu *open*, *write*, *read* und *close*. Achten Sie darauf, dass Sie sich die Funktionen in den Manpages zu den Systemaufrufen (*man 2 open*) anschauen und nicht in den Shell-Befehlen (*man open* oder *man 1 open*). Welche Headerdateien müssen Sie einbinden? Wie können Sie überprüfen, ob eine Datei korrekt geöffnet worden ist?

Aufgabe 4: Treiber aus Sicht einer Anwendung

/dev/random ist ein Gerät, das Zufallszahlen generiert. Schreiben sie ein Programm, das in einer Schleife nacheinander 1000 Byte-Zufallszahlen aus der Gerätedatei */dev/random* ausliest und auf die Konsole ausgibt. Beobachten Sie was passiert. Wie können sie die Generierung von neuen Zufallszahlen anregen?

2.7. Make zum Erstellen von Kernelmodulen verwenden

Ein Kernelmodul kann relativ leicht mit Hilfe von *make* übersetzt werden. Mit `make` können die Arbeitsschritte Compilierung, Linken usw. automatisiert werden. Die dazu notwendigen Anweisungen werden in einem Makefile festgehalten. Um ein Kernelmodul mit einem Makefile übersetzen zu können, ist es jedoch notwendig, dass die Kernelheader auf dem System installiert sind. Make wird angewiesen, das toplevel-Makefile des Kernels auszuführen. Dieses ruft anschliessend das selbst erstellte Makefile auf. Dazu werden die Option C und eine Variable M verwendet:

```
$ make -C <Kernel-Dir> M=<Working-Dir> modules
```

Die Option -C weist `make` an, in einem ersten Schritt ins Kernelverzeichnis zu wechseln. Somit wird das Target *modules* im Makefile dieses Verzeichnisses ausgeführt. Mit der Variable M kann dem Makefile im Kernelverzeichnis mitgeteilt werden, dass es auch das Makefile im übergebenen Verzeichnis aufruft. Im selbsterstellten Makefile muss somit im einfachsten Fall nur eine Zeile stehen:

```
obj-m := example.o
```

Diese Zuweisung bewirkt, dass ein Kernelmodul aus dem File *example.o* erzeugt werden soll (mit der Endung .ko). Das Objekt-File wird aus dem dazugehörigen .c File erstellt und muss nicht explizit angegeben werden. Der Aufruf von `make` mit allen Optionen und Pfadangaben ist jedoch relativ mühsam. Viel komfortabler wäre es doch, wenn das Makefile ein Target mit z.B. dem Namen *modules* hätte, dann genügt ein `make modules`. Genau dies kann mit dem folgenden Makefile erreicht werden:

Listing 2: Beispiel Makefile

```
ifeq ($(KERNELRELEASE),)
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)

modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions

.PHONY: modules clean

else
    obj-m := example.o
endif
```

Dieses Makefile arbeitet in zwei Schritten. Im ersten Schritt überprüft es woher der Aufruf kommt. Wird es durch den Benutzter aufgerufen, ist die Variable *KERNELRELEASE* noch nicht definiert. Daher wird der obere Teil des Makefiles ausgeführt.

Als erstes wird der Variablen *KERNELDIR* zugewiesen, wo sich die aktuellen Kernelsourcen bzw. Header befinden. Anschliessend muss festgestellt werden, woher der Aufruf kommt, damit das toplevel Makefile im Kerneltree dieses Makefile ein zweites Mal aufrufen kann. Dies geschieht mit der Zeile *PWD := $(shell pwd)*. Als nächstes wird nun das Toplevel Makefile im Kernelverzeichnis aufgerufen.

Dieses ruft, nachdem die Build-Konfiguration erstellt wurde, dann wiederum das oben stehende Makefile auf. Da nun die Variable *KERNELRELEASE* definiert ist, wird jetzt der untere Teil *obj-m := hello.o* ausgeführt.

Der ganze Buildprozess kann nun mit dem einfachen Kommando `make modules` über die Kommandozeile gestartet werden. Mit dem Kommando `make clean` werden alle durch make erstellten Files wieder gelöscht.

Hinweis: Zuweisungen in Makefile

Es gibt drei verschiedene Zuweisungsarten:

:=	speichert Wert von rechter Seite der Zuweisung in Variable auf der linken Seite (Standard).
=	wirkt wie eine Formel: die Definition auf der rechten Seite wird abgespeichert und der Wert wird jeweils evaluiert, wenn die Variable auf der linken Seite der Zuweisung verwendet wird.
?=	bewirkt, dass der voranstehenden Variable nur ein Wert zugewiesen wird, wenn sie zuvor noch nicht definiert wurde.

2.8. Ein erstes Kernelmodul

Wenden wir uns nun endlich einem ersten Beispiel, dem fast obligaten „Hello World“ zu. Obwohl nur die nötigsten Funktionen implementiert sind, dürfen wir mit gutem Gewissen von einem Kernelmodul sprechen, jedoch noch nicht von einem Treiber.

Listing 3: Hello-World-Modul

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3
4  MODULE_AUTHOR("urs.graf@ntb.ch");           /* 1) */
5  MODULE_DESCRIPTION("Hello world module");
6  MODULE_LICENSE("GPL");
7
8  static int hello_init(void) {                    /* 2) */
9      printk(KERN_ALERT "Hello, world\n");
10     return 0;
11 }
12
13 static void hello_exit(void) {
14     printk(KERN_ALERT "Goodbye, cruel world\n"); /* 3) */
15 }
16
17 module_init(hello_init);                         /* 4) */
18 module_exit(hello_exit);
```

Erklärungen zum Hello-World-Modul:

1) Über diese Makros werden einige Metainformationen für das Modul festgelegt. Zwingend notwendig ist die Lizenz. Für ein Kernelmodul sollte die BSD oder GPL Lizenz gewählt werden.

2) Die Funktion *hello_init* wird unmittelbar nach dem Laden des Moduls in den Kernel durch diesen aufgerufen. In dieser Funktion werden in der Regel alle Ressourcen alloziert die für den Betrieb des Moduls gebraucht werden. Im Kernelspace haben wir keinen Zugriff auf die normale C-Bibliothek. Es gibt also kein *printf*. Eine abgespeckte Variante davon gibt es mit *printk* im Kernel selber. Allerdings kann *printk* keine Ausgabe auf eine Konsole machen, sondern schreibt in einen Log-Buffer, das Kernel-Log.

3) Die Funktion *hello_exit* wird während des Entladens eines Moduls aufgerufen. In dieser Funktion müssen alle reservierten Ressourcen freigegeben werden. Wird dies nicht korrekt ausgeführt, werden die Ressourcen erst bei einem Neustart des Systems freigeben.

4) Über das Makro *module_init* wird dem Kernel die Initialisierungsfunktion mitgeteilt. Die Exitfunktion wird im Kernel über das Makro *module_exit* registriert.

Aufgabe 5: Hello World Modul

Erstellen Sie eine neue c-Datei (*hello_mod.c*) und implementieren Sie darin das Hello World Beispiel. Das benötigte Makefile können Sie dem Beispiel aus dem Abschnitt 2.7 entnehmen. Übersetzen Sie das Modul mit `make modules`. Danach laden Sie dieses neue Modul mit `insmod` in den Kernel. Um nun die Ausgabe sehen zu können, benötigen Sie das Werkzeug `dmesg`. Informieren Sie sich was dieses Tool macht und setzen Sie es ein, um die Hello-World Ausgabe angezeigt zu bekommen. Ganz praktisch im Zusammenhang mit `dmesg` ist auch das Standardwerzeug `tail`. Mit Hilfe von `lsmod` können Sie eine Liste aller geladenen Kernelmodule ausgeben. In dieser sollte auch Ihr Modul zu finden sein. Entfernen Sie zum Schluss Ihr Modul mit `rmmod` wieder.

Hinweis: Entwicklungsumgebung für Kernelmodule

Sie können das notwendige C-File und das Makefile mit einem beliebigen Texteditor erstellen und `make` dann auf der Kommandozeile aufrufen. *CMake* lässt sich für das Erstellen des Makefile nicht gewinnbringend einsetzen, weil der Kernel selber bereits interne Makefile aufweist, die ja auch aufgerufen werden. Sie können auch für Kernelmodule *KDevelop* verwenden. Sie benutzen dazu den *Custom Makefile Project Manager*. Lassen Sie sich im Unterricht zeigen, wie Sie dazu vorgehen müssen.

2.9. Kernelmodule im Vergleich zu herkömmlichen Applikationen

Bevor wir uns der eigentlichen Treiberentwicklung zuwenden können, müssen wir noch ein paar Aspekte behandeln, die sich von der herkömmlichen Applikationsentwicklung unterscheiden.

- Bei der Programmierung eines Treibers muss bedacht werden, dass dieser jederzeit unterbrochen und Rechenzeit einer anderen Anwendung zugeteilt werden kann. Aus diesem Grund muss sichergestellt werden, dass das gleichzeitige Benutzen von Ressourcen zu keinen Problemen führen kann. Linux stellt hierfür eine Reihe von Hilfsmittel zur Verfügung, die wir später noch kennen lernen.
- Da der Kernel selbst einen sehr kleinen Stack besitzt, muss darauf geachtet werden, dass dessen Gebrauch möglichst gering gehalten wird (also beispielsweise keine Rekursion verwenden).
- Gleitkoma-Operationen sind im Kernel nicht möglich, da bei einem Kontextwechsel keine Sicherung der entsprechenden Register stattfindet.

- Aufrufe von Funktionen denen ein Doppelstrich "__" vorangestellt ist müssen mit Bedacht eingesetzt werden, da es sich hier um um "low level" Aufrufe im Kernel handelt. Ein falscher Gebrauch dieser Funktionen kann zu einem instabilen System führen.
- Beim Programmieren von Treibern unter Linux muss sich der Programmierer bewusst sein, dass er keinen Zugriff auf die herkömmliche C-Bibliothek hat. Der Kernel stellt jedoch selbst eine Bibliothek mit den wichtigsten Funktionen zur Verfügung.

3. Treiberentwicklung

In diesem Kapitel werden wir nun in die eigentliche Treiberprogrammierung für Linux einsteigen und die wichtigsten Treiberfunktionen kennen lernen.

3.1. Initialisierung

Während der Initialisierung eines Moduls werden alle Funktionen, die es gegen aussen zur Verfügung stellt, im Kernel registriert und benötigte Ressourcen reserviert. Die Initialisierung ist im Listing 4 zu sehen.

Listing 4: Initialisierung

```
1 static int __init initialization_function(void) {
2     // Initalization code here
3 }
4 module_init(initialization_function);
```

Der Name der Initialisierungsfunktion kann frei gewählt werden, es ist jedoch zwingend, dass dem Kernel über das Makro *module_init* deren Funktionszeiger übergeben wird. Da die Funktion nicht von aussen aufgerufen wird, sollte sie immer statisch deklariert werden.

Das Schlüsselwort *__init* in der Deklaration der Funktion gibt dem Kernel den Hinweis, dass diese Funktion nur während der Initialisierungsphase gebraucht wird und deren belegter Speicher nach erfolgter Initialisierung freigegeben werden kann. Globale Daten, die wie die Initialisierungsfunktion nur während der Initialisierungsphase gebraucht werden, können mit dem Kürzel *__initdata* deklariert werden. Falls dies verwendet wird, muss jedoch unbedingt sichergestellt werden, dass nach der Initialisierung keine Funktion mehr auf diese Variable zugreift.

Falls die Initialisierung eines Moduls erfolgreich ist, wird der Wert 0 zurückgeben. Ansonsten ein Fehlercode.

> **Hinweis: Achtung**
> Sobald Funktionen im Kernel registriert worden sind, können diese auch von aussen aufgerufen werden. Um einen fehlerfreien Betrieb sicherzustellen, müssen somit alle verwendeten Ressourcen vorgängig reserviert werden.

3.2. Fehlerbehandlung während der Initialisierungsphase

Während der Initialisierung muss immer daran gedacht werden, dass die Registrierung von Funktionen und das Reservieren von Ressourcen fehlschlagen kann. Ein einfaches Beispiel hierfür ist das Reservieren von Speicher der entweder nicht vorhanden ist, oder durch ein anderes Modul schon benutzt wird. Somit muss immer überprüft werden, ob eine Reservierung beziehungsweise Registrierung erfolgreich war. Ist dies nicht der Fall, muss entschieden werden, ob das Modul dennoch mit eingeschränkter Funktionalität weiterbetrieben werden kann. Ansonsten müssen alle Registrierungen beziehungsweise Reservierungen rückgängig gemacht werden. Obwohl goto Statements in der C Programmierung verpönt sind, werden diese normalerweise für die Fehlerbehandlung in einem Modul eingesetzt, da es diese vereinfacht und der Code übersichtlicher wird. Eine Fehlerbehandlung während der Initialisierungsphase kann folgendermassen aussehen:

Listing 5: Typische Fehlerbehandlung

```
1  static int __init my_init_function(void) {
2      int err;
3
4      //registration takes a pointer and a name
5      err = register_this(ptr1,"myDriver");
6      if(err) goto fail_this;
7      err = register_that(ptr2, "myDriver");
8      if(err) goto fail_that;
9      err = register_those(ptr3,"myDriver");
10     if(err) goto fail_those;
11
12     return 0; //success
13
14     fail_those: unregister_that(ptr2,"myDriver");
15     fail_that:  unregister_this(ptr1,"myDriver");
16     fail_this:  return err; //propagate the error
17 }
```

In diesem Beispiel werden beim Kernel drei verschiedene fiktive Funktionen registriert. Tritt während der Initialisierung ein Fehler auf, springen die goto Anweisungen zu den Fehlerbehandlungen, welche die erfolgreich registrierten Funktionen beim Kernel abmelden und anschliessend dem Benutzer einen Fehlercode zurückgeben.

3.3. Aufräumen

Um einen fehlerfreien Betrieb des Kernels sicherzustellen, müssen beim Entfernen eines Moduls alle reservierten Ressourcen wieder freigegeben werden. Dies geschieht über eine Aufräumfunktion (engl. *exit function* oder *cleanup function*). Listing 6 zeigt wie eine solche aussehen kann.

Wie bei der Initialisierungsfunktion kann auch für die Aufräumfunktion der Name frei gewählt werden und muss dem Kernel über das Makro *module_exit* mitgeteilt werden. Das Kürzel *__exit* gibt an, dass der Code nur beim Entfernen des Moduls ausgeführt werden darf.

Das Freigeben von Funktionen und Ressourcen sollte gewöhnlich in umgekehrter Reihenfolge zur Initialisierung geschehen.

Listing 6: Aufräumen

```
1 static void __exit cleanup_function(void){
2     //cleanup code here
3 }
4 module_exit(cleanup_function);
```

3.4. Allozieren und Freigeben von Gerätenummern

Damit Treiber von aussen über die Major- und Minornummern angesprochen werden können, müssen diese von einem Modul reserviert werden. Hierfür gibt es zwei Vorgehensweisen. Beide unten vorgestellten Allozierungsfunktionen sind in *linux/fs.h* deklariert. Bei der statischen Allozierung muss der Programmierer vorgängig wissen, welche Major- und Minornummern auf einem System nicht belegt sind. Wird ein Treiber nur für das eigene System programmiert, stellt dies kein Problem dar. Sobald jedoch ein Treiber für andere Benutzer freigegeben wird, kann nicht mehr mit Sicherheit gesagt werden, dass die benötigten Nummern auch wirklich frei sind. Die statische Allozierung sieht folgendermassen aus:

```
int register_chrdev_region(dev_t first, unsigned int count, char* name)
```

Wobei *dev_t first* die erste Gerätenummer angibt, die das Modul registrieren möchte. Der Minoranteil von *first* ist normalerweise 0, kann jedoch auch anders gewählt werden. *first* kann mit Hilfe des Makros *MKDEV*[3] erzeugt werden. Der Integer *count* gibt an, wie viele aufeinander folgende Gerätenummer man registrieren möchte. Ist *count* gross, muss beachtet werden, dass der Bereich mehrere Majornummern umfassen kann. Der letzte Parameter *name* gibt an, mit welchem Namen die reservierten Gerätenummern in Verbindung gebracht werden sollen. Über den Rückgabewert kann bestimmt werden ob die Registrierung erfolgreich war oder nicht.

Bei der dynamischen Allozierung wird dem Modul durch den Kernel eine freie Majornummer zugeteilt. Somit kann der Treiber auf allen kompatiblen Systemen benutzt werden. Der Nachteil dieser Variante liegt jedoch darin, dass eine Gerätedatei nicht mehr mit einer fixen Majornummer in das Devicefilesystem eingehängt werden kann. Dies hat zur Folge, dass nach jedem Laden des Moduls in den Kernel auch die Gerätedatei neu erstellt werden muss. Möchte man Gerätenummern dynamisch registrieren sieht der Aufruf folgendermassen aus:

```
int alloc_chrdev_region(dev_t* dev, unsigned int firstminor,
                        unsigned int count, char* name)
```

Hier entsprechen die Parameter *name* und *count* sowie der Rückgabewert der statischen Allozierung. Über *firstminor* wird festgelegt, welches die kleinste Minornummer sein soll und über *dev*

[3] siehe Abschnitt 2.4.2 auf Seite 9

wird ein Zeiger auf das registrierte Gerät zurückgegeben. Werden registrierte Gerätenummern nicht mehr gebraucht, sollten diese wieder freigegeben werden. Dies erfolgt über die Funktion *unregister_chrdev_region*. Dabei muss als erster Parameter die erste Major/Minornummer übergeben werden und als zweiter Parameter die Anzahl Gerätenummern, die freigegeben werden sollen.

```
void unregister_chrdev_region(dev_t first, unsigned int count)
```

Hinweis: Registrierte Majornummern anzeigen
Welche Majornummern im Kernel momentan registriert sind, kann der Datei /proc/devices entnommen werden.

Aufgabe 6: Gerätenummern allozieren
Schreiben Sie ein Modul, das beim Laden zwei aufeinanderfolgende Gerätenummern alloziert. Über eine Konstante soll festgelegt werden können, ob die Allozierung statisch oder dynamisch erfolgen soll. Geben Sie nach erfolgreicher Allozierung die Majornummer aus. Vergessen Sie nicht, beim Entladen des Moduls die allozierten Gerätenummern wieder freizugeben.

Laden und entfernen Sie Ihr Modul testweise. Machen Sie das je einmal mit statischer und dynamischer Allokation und studieren Sie die Log-Ausgaben.

3.5. Datei-Operationen

Nachdem wir nun Gerätenummern registriert haben, müssen wir noch festlegen, welche Funktionen der Treiber einer Benutzerin zur Verfügung stellt. Diese Funktionen werden Dateioperationen genannt und sind über die Struktur *file_operations* (aus *linux/fs.h*) definiert. Jedes Feld in dieser Struktur muss entweder auf eine Funktion im Treiber, oder NULL zeigen. Alle Dateien die in Linux geöffnet sind (intern repräsentiert durch eine Struktur *file*, die wir uns im nächsten Kapitel ansehen werden), besitzen eine eigene Beschreibung der Funktionen, die dem Benutzer zur Verfügung stehen. Wie zum Beispiel *open* und *read*. Nachfolgend werden die wichtigsten Operationen aufgeführt. Eine komplette Liste kann im Anhang B ab Seite 43 gefunden werden.

Die meisten Treiber implementieren längst nicht alle Operationen, sondern nur die für sie relevanten. Die Initialisierung der Operationen kann dabei, wie in Listing 7 gezeigt, nur für die benötigten Felder erfolgen, ohne dass dazu die Reihenfolge der einzelnen Felder berücksichtigt werden muss.

Listing 7: Beispiel für die Initialisierung der Dateioperationen

```
1 struct file_operations my_fops = {
2     .owner           = THIS_MODULE,
3     .read            = my_read,
4     .write           = my_write,
5     .unlocked_ioctl  = my_ioctl,
```

```
6       .open           = my_open,
7       .release        = my_close
8   };
```

3.5.1. Die häufigsten Operationen der Struktur file_operations

owner

```
struct module *owner
```

Dieses Feld bezieht sich nicht auf eine eigentliche Dateioperation. Es ist ein Zeiger auf den Besitzer dieses Moduls und stellt sicher, dass ein Modul nicht entladen wird, solange dessen Operationen benutzt werden. Normalerweise wird es über das Makro *THIS_MODULE* initialisiert, welches in *linux/module.h* definiert ist.

open

```
int (*open) (struct inode *, struct file *);
```

Parameter:

`struct inode *`	inode pointer
`struct file *`	access mode (O_RDONLY, O_WRONLY, O_RDWR)

open ist jeweils die erste Operation die beim Gebrauch einer Gerätedatei aufgerufen wird. Wird diese Funktion durch den Treiber nicht angeboten, ist das Öffnen des Geräts immer erfolgreich. Der Treiber wird jedoch nicht darüber informiert.

release

```
int (*release) (struct inode *, struct file *);
```

Parameter:

`struct inode *`	inode pointer
`struct file *`	file descriptor

Die Operation *release* wird aufgerufen wenn die file Struktur freigegeben wird. Teilen sich mehrere Prozesse ein file Struktur (zum Beispiel nach einem *fork* oder *dump*) wird die *release* Operation erst aufgerufen, wenn alle Kopien geschlossen worden sind. Wie *open* kann *release* auch NULL sein.

read

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Parameter:

`struct file *`	file descriptor
`char __user *`	read data is written into this buffer to user
`size_t`	count, number of bytes to read
`loff_t *`	offset

Liest Daten von einem Gerät. Zeigt diese Operation auf *NULL*, wird *-EINVAL* ("fehlerhaftes Argument") zurückgegeben. Ein nichtnegativer Rückgabewert gibt an, wie viele Bytes erfolgreich gelesen werden konnten.

write

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

Parameter:

`struct file *`	file descriptor
`const char __user *`	write data from user
`size_t`	count, number of bytes to write
`loff_t *`	offset

Schreibt Daten in das Gerät. Falls diese Operation auf *NULL* zeigt, wird dem Aufrufenden *–EINVAL* zurückgeben. War der Schreibvorgang erfolgreich, wird die Anzahl geschriebener Bytes zurückgegeben.

ioctl

```
int (*unlocked\_ioctl) (struct inode *, struct file *, unsigned int,
              unsigned long);
```

Parameter:

`struct inode *`	inode pointer
`struct file *`	file descriptor
`unsigned int`	ioctl number
`unsigned long`	parameter

Der *unlocked_ioctl* Systemaufruf ermöglicht das Ausführen von gerätespezifischen Kommandos. Wird die Operation nicht unterstützt, wird *–ENOTT* („No such ioctl for device“) zurückgeben.

3.6. Die file Struktur

Die Struktur *file* (definiert in *linux/fs.h*) ist die zweitwichtigste Datenstruktur die in Gerätetreibern gebraucht wird. Die Struktur *file* darf nicht mit den FILE Zeigern herkömmlicher C-Programme verwechselt werden. FILE ist in der C Bibliothek definiert und tritt nie im Kernel Code auf! Die Struktur *file* hingegen ist eine Kernelstruktur und kann nicht in Benutzerprogrammen gebraucht werden. Die Struktur *file* repräsentiert eine geöffnete Datei (Dies betrifft nicht nur Gerätetreiber sondern alle geöffneten Dateien). Sie wird vom Kernel bei einem *open* Aufruf erstellt und jeder Funktion übergeben, die mit der geöffneten Datei agiert. Sobald alle Instanzen der Datei geschlossen worden sind, gibt der Kernel die Struktur wieder frei. Im Kernelcode wird ein Zeiger auf die Struktur *file* normalerweise *file* oder *filp* genannt. Um Verwechslungen mit der Struktur selbst zu vermeiden, empfehlen wir, ihn *filp* zu nennen.

3.6.1. Die wichtigsten Felder der Struktur file

f_mode

```
mode_t f_mode;
```

Der Dateimodus identifiziert die Datei durch die Bits *FMODE_READ* und *FMODE_WRITE* als entweder lesbar oder schreibbar (oder beides). Bei einem Aufruf überprüft der Kernel selbst ob der Benutzer die nötigen Rechte für die entsprechende Funktion besitzt. Ist dies nicht der Fall, wird der Aufruf zurückgewiesen, ohne dass der Treiber darüber informiert wird.

f_pos

```
loff_t f_pos;
```

Hier wird die aktuell Schreib- und Leseposition abgelegt. Der Treiber kann diesen Wert auslesen um die aktuelle Position festzustellen, sollte diesen Wert jedoch nicht verändern. Falls bei einem *read* oder *write* Aufruf die Position geändert werden muss, wird empfohlen dies über das letzte an die Funktion übergebene Argument zu tun. Die einzige Ausnahme ist der Aufruf *llseek*, welcher die Position in einer Datei ändert.

file_operations

```
struct file_operations *f_op;
```

Zeiger auf die Operationen, welche mit der Datei assoziiert sind. Diese Operationen wurden in Kapitel 3.5 vorgestellt. Der Kernel weist die Zeiger bei der Ausführung der Funktion *open* zu und liest sie immer, wenn er eine Operation weiterleiten muss. Der Wert in *filp->f_op* wird nie für später abgespeichert; das bedeutet, dass Sie die Datei-Operationen ihrer Datei ändern können, wann immer Sie wollen. Die neuen Methoden gelten dann unmittelbar nach dem Rücksprung zum Aufrufer. Dieses Verfahren erlaubt die Implementation unterschiedlichen Verhaltens für die gleiche Major-Nummer, ohne dass ein zusätzlicher Systemaufruf eingeführt werden muss. Die Möglichkeit, die Datei-Operationen zu ersetzen, ist das Kernel-Äquivalent zum Überschreiben von Methoden in der objektorientierten Programmierung.

```
void *private_data;
```

Bei einem *open* Systemaufruf wird dieser Zeiger auf *NULL* gesetzt bevor die *open* Funktion aufgerufen wird. Allozierter Speicher kann über diesen Zeiger verwaltet werden. Dieser muss jedoch in der *release* Operation wieder freigegeben werden. *private_data* ist eine gute Möglichkeit um Zustandsinforationen zwischen Systemaufrufen abzuspeichern.

3.7. Die inode Struktur

Wird eine Datei von mehreren Prozessen gleichzeitig geöffnet, besitzt sie mehrere der im vorgängigen Kapitel beschriebenen *file* Strukturen. Diese werden in der Struktur *inode* zusammengefasst, welche eine Menge Informationen über eine Datei beinhaltet. Für die Treiberprogrammierung sind jedoch nur zwei Felder relevant.

```
dev_t i_rdev;
```

Falls *inode* eine Gerätedatei repräsentiert, beinhaltet dieses Feld die aktuelle Gerätenummer.

```
struct cdev *i_cdev
```

Das Feld *i_cdev* beinhaltet einen Zeiger auf die Struktur *cdev*, falls es sich bei der Datei um eine Zeichengerätedatei handelt.

3.8. Zeichengeräte erzeugen und registrieren

Ein Zeichengerät wird im Kernel über die Struktur *cdev* repräsentiert. Zuerst muss also eine solche Struktur *cdev* erzeugt und anschliessend im Kernel registriert werden. Die Definition von *struct cdev* und deren Hilfsfunktionen befinden sich in *linux/cdev.h*. Diese muss daher in jedem Modul für Zeichengeräte eingebunden werden. Die Allozierung und Registrierung von *struct cdev* sieht folgendermassen aus.

Listing 8: Allozierung und Registrierung von struct cdev

```
1  struct cdev *my_cdev;
2
3  static int __init my_init_function {
4      .
5      .
6      my_cdev = cdev_alloc();                  /* 1) */
7      my_cdev->owner = THIS_MODULE;            /* 2) */
8      my_cdev->ops = &my_fops;                 /* 3) */
9      err = cdev_add(my_cdev, my_dev_t,1);     /* 4) */
10     if(err) goto cdev_add_failed;
11     .
12     .
13     cdev_add_failed:                         /* cleanup */
14 }
```

1) Allozieren des Speicherbedarfs für die Struktur *cdev* mittels der Funktion *cdev_alloc()*;

2) Zuweisung des Besitzermoduls

3) Zuweisen der Dateioperationen. Diese müssen wie in Abschnitt 3.5 gezeigt, definiert worden sein.

4) Registrieren des Zeichengeräts im Kernel mittels *int cdev_add(struct cdev *dev, dev_t num, unsigned int count)*. Wobei *dev_t num* der Gerätenummer entspricht und *count* der Anzahl Geräte die aufeinander folgend registriert werden sollen.

Bei der Registrierung eines Geräts im Kernel, müssen folgende Punkte beachtet werden.

a) Falls beim Aufruf von *cdev_add* ein negativer Wert zurückgeben wird, ist die Registrierung fehlgeschlagen und das Gerät kann nicht benutzt werden.

b) War die Registrierung erfolgreich, wird das Gerät vom Kernel als verfügbar betrachtet. Daher muss sichergestellt werden, dass zu diesem Zeitpunkt alle benötigten Ressourcen reserviert und bereit sind.

Wird ein Gerät nicht länger benötigt, muss es über *void cdev_del(struct cdev *dev)* aus dem System entfernt werden. Dies gilt auch beim Entfernen eines Moduls aus dem Kernel.

3.9. Automatisches Erstellen von Gerätedateien

Nachdem das Zeichengerät im Kernel registriert ist, müssen die Gerätedateien (Device Nodes) erstellt werden. Dies hatten wir zuerst von Hand mit dem Befehl *mknod*[4] gemacht. Eleganter ist die dynamische Erzeugung mittels *device_create* (deklariert in *linux/device.h*). Diese Funktion sendet einen *uevent* an *udev*[5], um in */dev* die Device Nodes zu erstellen.

[4] siehe Abschnitt 2.4.2 auf Seite 9
[5] siehe Abschnitt 2.5.3 auf Seite 11

```
struct device * device_create(struct class *, struct device *, dev_t,
                              void *, const char *, ...);
```

Parameter:

`struct class *`	pointer to class
`struct device *`	pointer to parent struct device, if any
`dev_t`	dev_t of the char device
`void *`	data to be added for callbacks
`const char *`	string, device name
`...`	variable arguments

Die Struktur *class*, welche als erster Parameter übergeben wird, muss vor dem Ausführen der Funktion über *class_create* erstellt werden. Diese Funktion erstellt im *sysfs* einen neuen Eintrag. Listing 9 zeigt die Verwendung. Wenn mehrere Geräte in einer for-Schleife registriert werden sollen, kann dies, wie in Listing 10 gezeigt, passieren.

Listing 9: Beispiel Device Node erstellen

```
1 struct class *my_class = class_create(THIS_MODULE, "MyClass");
2 device_create(my_class, NULL, dev, NULL, "MyNameOfTheDevive");
```

Listing 10: Beispiel mehrere Device Nodes erstellen

```
1 for(i = 0; i < NOF_DEVS; i++) {
2     device_create(my_class, NULL, MKDEV(MAJOR(dev), i), NULL, "my%d", i);
3 }
```

> **Hinweis: Userspace - Kernelspace**
> Wir hatten aus dem Userspace mit *mknod* auch eine Gerätedatei erstellt (allerdings nur diese Datei). Mit *device_create* wird auch eine solche, aber aus dem Kernelspace, erstellt.

Jetzt wird *udev* eine neue Gerätedatei erstellen. Diese Gerätedatei "gehört" dem Benutzer root. Damit auch alle anderen Benutzer darauf zugreifen können, müssen die Rechte richtig gesetzt werden. Dies kann von Hand mit dem Programm *chmod* passieren. Viel eleganter ist es aber, wenn *udev* die Datei direkt mit den richtigen Rechten erstellt. Dazu kann eine udev-Regel erstellt werden. Diese werden in Textdateien im Verzeichnis */etc/udev/rules.d/* definiert. Der Dateiname jeder Regeldatei besteht aus zwei Teilen, einer Ordnungsnummer und einem beschreibenden Text getrennt durch einen Bindestrich, also z.B. *70-foobar.rules*. Die Regel selber sieht dann wie folgt aus:

```
KERNEL=="testDevice", OWNER="ntb", GROUP="root", MODE="0666"
```

Wobei *testDevice* der Name der Gerätedatei ist.

Beim Entfernen des Moduls müssen auch die erstellten Device-Nodes wieder entfernt werden.

Listing 11: Beispiel Device Node entfernen

```
1 device_destroy(my_class, dev);  // delete sysfs entries
2 class_destroy(my_class);
3 cdev_del(...);
```

Aufgabe 7: Deviceregistrierung

Schreiben Sie nun einen kompletten Treiber, der sich zuerst eine freie Gerätenummer alloziert, dann das Device registriert und den Device Node erzeugt. Öffnen Sie dann eine Root-Shell und laden Sie das Modul. Prüfen Sie im Kernel-Log, ob das Laden erfolgreich war. Listen Sie die Gerätedateien auf (unter */dev*). Ist eine neue Datei für Ihr Modul vorhanden und hat diese die korrekten Rechte?

3.10. Open / Release

Die Funktion *open* (siehe Abschnitt 3.5) wird durch den Kernel beim Öffnen der dazugehörigen Gerätedatei aufgerufen. In dieser Funktion werden die für den späteren Gebrauch notwendigen Initialisierungen und Allozierungen durchgeführt. In den meisten Treibern werden in der *open* Funktion folgende Tätigkeiten durchgeführt.

- Funktionscheck der Hardware
- Initialisierung des Gerätes, falls es zum ersten Mal geöffnet wird
- Anpassung der Dateioperationen, falls dies notwendig ist

Ist das Durchführen dieser Tätigkeiten für den Betrieb des Treibers nicht notwendig, kann auf die Implementierung der *open* Funktion verzichtet werden. In diesem Fall meldet der Kernel jedes Öffnen der Gerätedatei als erfolgreich.

Die Funktion *release* wird aufgerufen, sobald die letzte Instanz einer geöffneten Datei geschlossen wird. In dieser Funktion werden alle in *open* allozierten Ressourcen wieder freigegeben und das Gerät beim letzten *release* Aufruf wieder heruntergefahren.

Aufgabe 8: Open/Release

Erweitern Sie Ihr letztes Modul durch eine *open* und eine *release* Funktion. Diese sollen beim Aufruf jeweils eine Nachricht ausgeben. Vergessen Sie nicht die Registrierung dieser Dateioperationen in der Struktur *file_operations*.
Die zwei Funktionen bewirken vorerst noch gar nichts. Implementieren Sie eine kleine Testanwendung, mit der Sie das eben erstellte Modul testen können. Rufen Sie in dieser Testanwendung die Funktionen *open* und *release* auf.
Achtung: Sie müssen hier zwei Projekte erzeugen. Das erste Projekt (z.B. mit Namen *OpenReleaseModule*) hat ein spezielles Kernelmodul-Makefile und übersetzt ein Kernelmodul. Dieses laden Sie dann mit `insmod`. Anschliessend testen Sie Ihr neues Modul aus. Dazu erstellen Sie ein zweites Projekt (z.B. mit Namen *OpenReleaseTest*). Das kann ein Standardprojekt mit CMake sein.

Was würde passieren, wenn Ihre vom Treiber erstellte Gerätedatei nur Root-Rechte hat und Sie die Funktion *open* darauf aufrufen?

3.11. Read

Über die Funktion *read* können Daten vom Treiber gelesen werden. Sie wird über den Systemaufruf *ssize_t read(int fd, void *buf, size_t count)* (siehe Manpage) aufgerufen. Beim Aufruf der Funktion *read* wird dem Modul als Parameter ein Zeiger auf die zu kopierenden Daten sowie deren Anzahl übergeben. Der Rückgabewert entspricht der Anzahl kopierter Zeichen. Da ein Kernelmodul nicht direkt in den User Memory Bereich zugreifen darf, muss für das Kopieren der Daten ein Makro verwendet werden (definiert in *linux/uaccess.h*):

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

Bei einem Aufruf dieser Funktion wird die Anzahl der nicht kopierten Zeichen zurückgeben, im fehlerfreien Fall 0. Um einzelne Zeichen vom Kernel-Space in den User-Space zu kopieren reicht das folgende Makro:

```
int put_user(val, dest);
```

Dabei wird das Zeichen an die Adresse des Pointers *dest* kopiert. Der Wert des Zeichens kann 1, 2, 4 oder 8 Bytes haben, abhängig vom Datentyp des Parameters *val*. Im fehlerfreien Fall gibt die Funktion 0 zurück, ansonsten einen negativen Fehlercode. Listing 12 zeigt ein Beispiel für die Verwendung von *put_user*.

Listing 12: Beispiel Read

```
1 ssize_t my_read_func(struct file *from, char __user *data, size_t size, loff_t *offs){
2     int val = 12;
3     if(put_user(val,data)) return 0;
4     return sizeof(val);
5 }
```

3.12. Write

Mittels der Funktion *write* (siehe Manpage) können Daten auf ein Gerät geschrieben werden. Da der direkte Zugriff auf Daten im User-Space nicht erlaubt ist, müssen diese wie bei *read* über Makros kopiert werden:

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

Dieses kopiert einen Speicherbereich vom User-Space in den Kernel-Space wobei die Parameter denen von *read* entsprechen. Einzelne Zeichen können wie folgt kopiert werden:

```
int get_user(var, src);
```

Ein einfaches Beispiel für die korrekte Verwendung von *get_user* ist in Listing 13 zu sehen.

Listing 13: Beispiel Write

```
1 ssize_t priv_buf_write(struct file *f, const char __user *data, size_t size, loff_t *
      offs ){
2     int i = 0;
3     u8 val;
4     if(!get_user(val,data)){ //1 Byte in den Kernelspace kopieren
5         i = 1;
6         printk(KERN_ALERT "Wert gelesen --> %d\n",val);
7     }
8     return i;
9 }
```

Aufgabe 9: Read/Write

Implementieren Sie in Ihrem Modul die Funktion *read*. Kopieren Sie bei einem Aufruf dieser Funktion eine fixe Zeichenkette in den User-Space. Implementieren Sie weiter die Funktion *write*. Kopieren Sie bei einem Aufruf dieser Funktion die übergebene Anzahl Bytes in den Kernelspace und schreiben Sie diese mit *printk* in den Kernellog. Testen Sie das Programm mit einer Testanwendung oder direkt von der Konsole aus dem Befehl `cat` oder `less`.

Hinweis: Testen mit Linux-Systemprogrammen

Statt eine Testapplikation im Userspace zu schreiben, kann häufig mit Systemprogrammen getestet werden. `cat /dev/testDevice` ruft nacheinander die Funktionen *open*, *read* und *close* auf. Probieren Sie es aus.

3.13. I/O Control

Über die Funktion *unlocked_ioctl* ist es möglich, gerätespezifische Operationen durchzuführen. Sie wird durch den Anwender über den folgenden Systemaufruf aufgerufen (siehe Manpage):

```
int ioctl(int d, int request, ...);
```

Nebst dem Kommando *request* werden dem Modul auch zusätzliche Parameter übergeben, welche die Steuerung des Geräts ermöglichen. Obwohl beim Systemaufruf mehrere Parameter übergeben werden können, ist dies in der Praxis eher unüblich. Vielmehr wird eine zum Kommando gehörige Datenstruktur definiert und diese mittels eines Zeigers dem Modul übergeben. Im Modul ist der zum Kommando gehörige Parameter per Definition vom Typ *unsigned long*, so dass in den meisten Fällen eine Typenwandlung unumgänglich ist. Das Beispiel in Listing 14 zeigt ein typisches Beispiel für eine Implementierung von *unlocked_ioctl*.

Listing 14: Beispiel für die Implementierung von ioctl

```
long my_ioctl(struct file *f, unsigned int cmd, unsigned long val){
    switch(cmd){
        case ONE:
            // .
            // .
            break;
        case TWO:
            // .
            // .
            break;
        default:
            // .
            // .
            break;
    }
    return 0;
}
```

4. Cross Development

Die meisten eingebetteten Systeme weisen eine andere Architektur auf als normale PCs. Stark verbreitet sind auf solchen Systemen ARM, MIPS und PowerPC Prozessoren. Dadurch ergibt sich das Problem, dass sich die Plattform, auf der entwickelt wird, von der Zielplattform, auf der die Anwendung später laufen wird, unterscheidet. Übersetzt der Entwickler sein Projekt auf seinem Rechner mit dem normalen C-Compiler, so wird die Anwendung auf der Zielhardware nicht lauffähig sein. Es ist also notwendig, die Anwendung für eine Fremdarchitektur zu übersetzen. Man spricht in diesem Fall von *crosscompiling*.
Dieses Kapitel geht auf die Entwicklung von Kernelmodulen für solche Fremdarchitekturen ein.

4.1. Die Zielplattform

4.1.1. Die Hardware

Für die Linux-Treiberentwicklung auf einem eingebetteten System verwenden wir das *Zoom OMAP-L138 eXperimenter* von Logic PD (siehe Abbildung 4.1). Der Prozessor des Boards ist ein OMAP-L138 von Texas Instruments. Dieser basiert auf einem DSP-Kern (TMS320C6748) und einem ARM926.

Das Betriebssystem wird über eine SD-Karte geladen. Diese besitzt zwei Partitionen: Auf der einen ist das Root-Filesystem abgelegt und auf der anderen die Boot-Dateien mit dem kompilierten Linux-Kernel als *uImage*. Auf der seriellen Schnittstelle des Systems steht eine Konsole bereit.

Aufgabe 10: Board Inbetriebnahme

a) Schliessen Sie das Board an (Netzwerkkabel und Versorgung).

b) Stellen Sie alle Dip-Schalter auf *off* damit der Prozessor von der SD-Karte bootet.

c) Schalten Sie nun das Board ein. Der Bootvorgang dauert etwa 10 Sekunden.

d) Starten Sie in der virtuellen Maschine ein Terminal und führen Sie anschliessend den Befehl *ssh -l root <board-name>* aus. Damit öffnen Sie eine Shell über das SSH[6]-Protokoll. Damit das funktioniert, muss auf dem Target ein SSH-Server laufen. Dieser wird in der aktuellen Konfiguration beim Aufstarten des Systems bereits gestartet.

e) Melden Sie sich mit dem Passwort *toor* an.

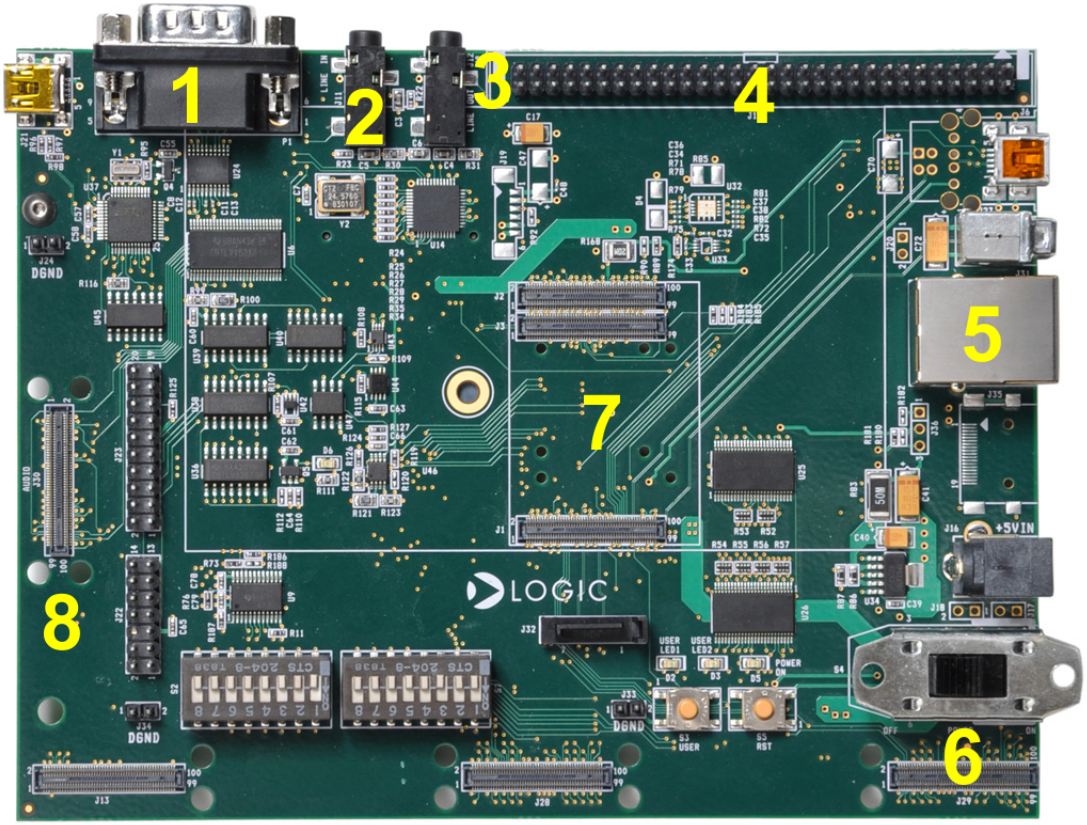

Abbildung 4.1.: Zoom OMAP L138 Experimenter Kit ohne Display

Abbildung 4.1 zeigt das verwendete Entwicklungsboard. Die wichtisten Komponenten sind gekennzeichnet[7]:

1 RS-232 Schnittstelle
2 Line-in stereo (links), Line-out stereo (rechts)
3 SD-Card Slot (auf Rückseite)
4 LCD Anschluss
5 Ethernet Schnittstelle
6 Ein- / Ausschalter
7 OMAP L-138 Steckplatz
8 Anschluss GPIO Zusatzboard

Hinweis: Board ausschalten
Um inkonsistente Daten zu vermeiden, muss vor jedem Ausschalten des Targets das Kommando `sync` oder `halt` ausgeführt werden!

[6] siehe http://en.wikipedia.org/wiki/Secure_Shell
[7] Der komplette User Guide und Quick Start Guide zum Zoom OMAP Board ist auf dem Infoportal zu finden: http://wiki.ntb.ch/infoportal/embedded_systems/omapl138_tms320c6748/start

4.1.2. Datenaustausch

Für den Datenaustausch zwischen unserer virtuellen Maschine und dem Board verwenden wir `scp`[8]. Damit können Dateien ganz ähnlich wie mit `cp` kopiert werden, nur eben über Rechnergrenzen hinweg. `scp` verwendet für den Kopiervorgang SSH (Secure Shell). Um das Ganze etwas komfortabler zu gestalten, werden wir den Kopiervorgang in unser Makefile integrieren. Die Syntax für einen Kopiervorgang sieht folgendermassen aus:

```
scp sourcefile username@host:destination
```

Dabei werden wir in diesem Kurs alle Dateien im Verzeichnis */media/ram* ablegen. Das nachfolgende Beispiel zeigt den Kopiervorgang des Kernelmoduls *hello.ko* auf das Target mit dem Hostnamen *es092*:

```
scp hello.ko root@es092.ntb.ch:/media/ram
```

Hinweis: Netzwerkprobleme

Falls die Netzwerkverbindung oder das Kopieren mittels *scp* nicht richtig funktioniert, prüfen Sie auf dem Board die Netzwerkparameter und ob der SSH-Daemon gestartet ist. Gehen Sie dazu wie folgt vor:

1. Verbinden Sie dazu das Board mit einem seriellen Kabel mit dem PC.
2. Starten Sie anschliessend *Putty* und konfigurieren Sie die serielle Verbindung wie folgt: 115200 Bps, 8 Data Bits, No Parity, 1 Stop Bit.
3. Starten Sie nun das Board. Falls es bereits läuft, drücken Sie einfach die Enter-Taste und melden Sie sich als *root* an (Passwort: *toor*).
4. Überprüfen Sie die Netzwerkeinstellungen mittels *ifconfig* und *route*.
5. Starten Sie den SSH-Daemon neu.

4.2. Cross Toolchain

Unsere Toolchain besteht aus verschiedenen Komponenten. Neben einem Crosscompiler, einem Assembler und einem Linker gehören auch noch diverse Hilfsprogramme und Bibliotheken zur Toolchain. Um die Anwendungen von denen für das Host-System unterscheiden zu können, beginnen alle mit einem plattformspezifischen Präfix: *arm-buildroot-linux-gnueabi-*. Das heisst, mit dem Kommando `gcc hello.c` übersetzen Sie die Datei *hello.c* für den Host, mit `arm-buildroot-linux-gnueabi-gcc hello.c` hingegen für das OMAP-Board.
Im Image der virtuellen Maschine ist bereits die Crosstoolchain für das OMAP-Board installiert. Diese Toolchain wurde gemäss Anleitung auf dem Infoportal zusammengestellt:
http://wiki.ntb.ch/infoportal/software/linux/buildroot/zoom/start.

[8] siehe http://en.wikipedia.org/wiki/Secure_copy

Aufgabe 11: Hello-World Module auf Zoom-Board

Nehmen Sie das Hello-World Module aus dem Kapitel 2.8 und erstellen Sie ein neues Makefile-Projekt. Passen Sie im Makefile das Kernelverzeichnis für den OMAP-Prozessor an. Suchen Sie das passende Verzeichnis in Ihrem Homeverzeichnis.

Ausserdem müssen Sie noch angeben, dass für eine Fremdarchitektur compiliert wird und welcher Compiler verwendet werden soll. Dazu muss das Target *modules* neu so aussehen:

```
$(MAKE) ARCH=arm CROSS_COMPILE=\
~/zoom/buildroot-toolchain/bin/arm-buildroot-linux-gnueabi- -C \
$(KERNELDIR) M=$(PWD) modules
```

Übersetzen Sie nun das Hello-World Modul für die ARM-Achitektur und kopieren Sie das Compilat auf das Target. Ergänzen Sie Ihr *Makefile* um ein Target *copy2board*, welches das erstellte Kernelmodul automatisch auf Ihr Board hinunter lädt. Das Kernelmodul soll auf dem Target im Verzeichnis */media/ram* abgelegt werden.
In einer Shell laden und entladen Sie dort dieses Modul (genau gleich wie vorher auf dem Host). Lesen Sie den Kernel-Log auf dem Target. Funktioniert es?

5. Zugriff auf Hardware

5.1. Hardware-Ressourcen reservieren

Unter Linux gibt es mehrere Möglichkeiten, wie Hardware angesteuert werden kann. Aus zeitlichen Gründen werden wir uns jedoch nur mit der Ansteuerung über Speicherbereiche beschäftigen. Bei dieser Methode wird die Hardware direkt über eine Adresse im Speicherbereich angesprochen. Der Kernel muss benötigte Bereiche vorgängig reservieren. Damit wird verhindert, dass ein anderer Treiber auf den gleichen Speicherbereich zugreifen kann. Dies erfolgt über die folgende Funktion (definiert in *linux/ioport.h*):

```
struct resource *request_mem_region(unsigned long from,
                                    unsigned long length, const char *name);
```

Wobei der erste Parameter die Startadresse des gewünschten Bereichs und der zweite Parameter die gewünschte Grösse ist. Zudem kann der Speicherbereich mit einem Namen versehen werden. War die Reservierung erfolgreich, wird die Adresse der neuen Ressource zurückgeben. Im Fehlerfall ein Zeiger auf NULL. Sie können die aktuell reservierten Bereiche in einer Konsole anzeigen mit `cat /proc/iomem`. Testen Sie das einmal auf dem Host und dem Target aus.

Wird eine Ressource nicht mehr gebraucht, muss sie mit *release_mem_region* freigegeben werden:

```
release_mem_region(unsigned long from, unsigned long length)
```

Als nächstes muss für eine bestimmte physische Adresse die entsprechende virtuelle Adresse (in unserem Fall für den Kernelspace) bestimmt werden. Dies entspricht einem Mapping in den aktuellen Speicherbereich und kann mit folgender Funktion erledigt werden (definiert in <*asm/io.h*>). Als Rückgabewert erhält man einen Zeiger auf den Speicherbereich.

```
void __iomem  *ioremap(unsigned long address, unsigned long size)
```

Wird die Ressource nicht länger gebraucht, ist es auch hier notwendig, diese über *iounmap* freizugeben.

```
iounmap(volatiole void __iomem *addr)
```

Das Beispiel in Listing 15 soll die Reservierung von Hardware Ressourcen verdeutlichen.

Listing 15: Beispiel für die Reservierung von Hardware Ressourcen

```
1 //register memory region and remap it
2 if(request_mem_region(MEM_ADDR,MEM_SIZE,NAME) == NULL)
3     goto mem_reg_fail;
4 basePtr = ioremap(MEM_ADDR,MEM_SIZE);
5 if(basePtr == NULL)
6     goto mem_remap_fail;
```

5.2. Zugriff auf Hardware Ressourcen

Nachdem Hardware Ressourcen reserviert worden sind und ein Remapping durchführt wurde, ist es theoretisch möglich, direkt mittels des über die Funktion *ioremap* zurückgegebenen Zeigers auf die Hardware zuzugreifen. Je nach Hardware wird dieser direkte Zugriff jedoch nicht unterstützt. Aus diesem Grund bietet der Kernel eine Reihe von Funktionen an, über die der Zugriff ermöglicht wird (in *<linux/io.h>*.

- *__u8 ioread8(void __iomem *addr)* liest ein Byte von der Adresse *addr* und gibt dieses dem Aufrufer zurück
- *void iowrite8(__u8 value, void __iomem *addr)* schreibt ein Byte auf die Adresse *addr*
- *__u16 ioread16(void __iomem *addr)* liest ein 16 bit Wort von der Adresse *addr* und gibt dieses dem Aufrufer zurück.
- *void iowrite16(__u16 value, void __iomem *addr)* schreibt ein 16 bit Wort auf die Adresse *addr*
- *__u32 ioread32(void __iomem *addr)* liest ein 32 bit Wort von der Adresse *addr* und gibt dieses dem Aufrufer zurück
- *void iowrite32(__u32 value, volatile void __iomem *addr)* schreibt ein 32 bit Wort auf die Adresse *addr*.

Nachfolgendes Beispiel zeigt den Zugriff auf das Data-Register eines I/O-Bereichs.

Listing 16: Beispiel für den Zugriff auf Hardware Ressourcen

```
1 //read value from 8 bit register
2 myVal = ioread8(basePtr + offset);
3 myVal |= SOME_BITS;
4 //write 8 bit to hardware
5 iowrite8(myVal, (basePtr + offset));
```

Den Zugriff auf reale Hardware können wir natürlich nicht auf dem Host in der VirtualBox durchführen, ausser wir würden reale Hardware des Hosts nehmen.

5.3. Treiber für GPIO, Version 1

Nun ist es Zeit, einen "richtigen" Treiber zu schreiben. Wir möchten damit die 4 LEDs und die 2 Taster auf einem kleinen Erweiterungsboard ansteuern. Die LEDs sind auf den Pins GPIO[2], GPIO[6], GPIO[13] und GPIO[15], die Taster auf GPIO[0] und GPIO[1]. Der OMAPL138 bietet einen riesigen Funktionsumfang. Trotzdem ist die Anzahl der Pins relativ stark beschränkt. Aus diesem Grund dienen alle Pins mehreren Einheiten und werden darum multiplexed. Es gibt zwei Registerbereiche, die die Steuerregister für diese GPIO enthalten. Beide Bereiche müssen Sie je separat reservieren und remappen. Die notwendigen Definitionen für die Bereiche und auch die jeweiligen Grössen der Bereiche finden Sie im Headerfile *omapL138Exp_io.h* im Anhang.

1. Pin-Multiplexregister: *PINMUX_REG_BASE* und *PINMUX_REG_SIZE*
2. Pin-Register: *GPIO_REG_BASE* und *GPIO_REG_SIZE*

Sie müssen die Funktionsweise dieser Pins in den entsprechenden Pin-Multiplexregister wie folgt konfigurieren:

```
temp = ioread32(pinmuxBasePtr + PINMUX1_OFFSET);
temp |= 0x88800080; // set GP0[0], GP0[1], GP0[2], GP0[6] as GPIO
temp &= 0x888FFF8F;
iowrite32(temp,pinmuxBasePtr + PINMUX1_OFFSET);

temp = ioread32(pinmuxBasePtr + PINMUX0_OFFSET);
temp |= 0x00000808; // set GP0[13], GP0[15] as GPIO
temp &= 0xFFFFF8F8;
iowrite32(temp,pinmuxBasePtr + PINMUX0_OFFSET);
```

wobei *temp* vom Typ *uint32_t* und *pinmux_baseptr* der Rückgabewert der Funktion *ioremap* ist. Mit Vorteil fügen Sie diesen Code in der Funktion *open* ein.
Noch ein Wort zu Pointerarithmetik. Die Konfigurationsregister sind allesamt 4-Bytes grosse Register. Also definiert man die Variable *pinmuxBasePtr* wie folgt:

```
static unsigned int *pinmuxBasePtr;
```

Wenn Sie nun auf das Register *PINMUX1* zugreifen möchten, dessen Adresse um um 4 höher ist als die Basisadresse, so darf wie im vorher gezeigten Beispiel einfach geschrieben werden:

```
temp = ioread32(pinmuxBasePtr + PINMUX1_OFFSET);
```

und *PINMUX1_OFFSET* ist richtigerweise im Headerfile definiert als 1!

Aufgabe 12: Treiber Version 1 für das OMAP-L138 Board
Erstellen Sie ein Modul *zoomOmap_io.c*, in welchem Sie einen Treiber für die Leuchtdioden des OMAP Erweiterungsboard implementieren. Der Zugriff auf die Hardware soll vorerst nur über die Funktionen *open* und *release* möglich sein. Die Adressen der Register können Sie dem Headerfile *omapL138Exp_io.h* im Anhang entnehmen.

a) Verschaffen Sie sich einen Überblick über die Funktionsweise der I/O-Ports auf dem OMAP-L138 Board. Details finden Sie dazu im Manual “OMAP-L138 DSP+ARM Processor: Technical Reference Manual” im Kapitel 21 ab Seite 897.

b) Als erstes implementieren Sie die Funktionen für das Laden und Entladen des Moduls. Beim Laden gehen Sie gleich vor wie auf dem Host. Erzeugen Sie ein Device mit einem passenden Namen. Anders als auf dem Host muss nun auch noch der Speicherbereich für die Pin-Multiplexing-Register und die GPIO-Register reserviert werden und das Remapping gemacht werden.

c) Implementieren Sie auch die Funktionen *open* und *close*. In *open* konfigurieren Sie die Pin-Multiplexing-Register wie oben gezeigt, dann setzen Sie das Data Direction Register für die LEDs so, dass diese 4 Pins Ausgänge sind. Achtung: Damit ein bestimmter Pin ein Ausgang ist, muss eine ’0’ im Data Direction Register an der entsprechenden Bitposition stehen. Zum Schluss schreiben Sie ein beliebiges Muster auf diese 4 Pins.

d) Erweitern Sie Ihr Makefile um ein Target, das den Treiber auf das Zoom-Board kopiert. Laden Sie dort anschliessend das Modul.

Hinweis: Achtung Makefile

Wir müssen beim Erstellen eines neuen Projektes und des dazugehörigen Makefiles uns stets überlegen, welches Ziel wir verfolgen. Je nachdem ob wir ein Kernelmodul oder ein Programm im Userspace als Ziel haben, müssen wir das passende Makefile mit den dazugehörigen Compilerdirektiven erstellen. Und zudem müssen wir nun aufpassen, für welche Plattform wir Code erzeugen wollen (Host (gcc), Target (arm-linux-gnueabi-gcc)).

Aufgabe 13: Testprogramm für Treiber Version 1

Jetzt schreiben Sie ein Testprogramm für diesen ersten Treiber. Darin müssen Sie das Device nur öffnen und wieder schliessen. Jetzt sollten die LEDs mit dem definierten Muster leuchten. **Achtung:** Für das Testprogramm benutzen Sie ein normales CMake-Projekt. CMake wird das notwendige Makefile erstellen. Anders aber als das Makefile für ein Applikationsprogramm auf dem Host muss der Compiler Code für einen ARM-Prozessor erzeugen. Also muss die Definition für den Compiler wie folgt lauten:

```
CC=arm-linux-gnueabi-gcc
```

Wir erreichen das indem wir beim Erzeugen des CMake-Projektes unter *Extra Arguments* das notwendige Toolchainfile angeben mit `-DCMAKE_TOOLCHAIN_FILE=~/zoom/buildroot-toolchain/share/buildroot/toolchainfile.cmake` Dann lohnt es sich auch hier, ein weiteres Target zu erstellen, um dieses Programm auf die Zielplattform zu kopieren. Führen Sie dieses Testprogramm aus. Was passiert?

5.4. Treiber für GPIO, Version 2

Nachdem die erste Version des Treibers läuft, wollen wir nun die endgültige Version erstellen und testen.

Aufgabe 14: Treiber für das OMAP-L138 Board

a) Erweitern Sie Ihren Treiber um die Funktionen *read* und *write*. Damit sollen die LEDs angesteuert und die Zustände der Taster ausgelesen werden können. Wählen Sie eine sinnvolle Kodierung zwischen übertragenem Datum und Led- resp. Tasternummer. Testen Sie das gleich aus (siehe Aufgabe 15).

b) Nun implementieren Sie auch noch die Funktion *unlocked_ioctl*. An sich haben wir gar keine sinnvolle Funktion, die mit *unlocked_ioctl* zu lösen wäre und die mit *read* oder *write* nicht bewerkstelligt werden kann. Schliesslich ist unser Device ja auch nur sehr rudimentär. Implementieren Sie *unlocked_ioctl* so, dass über unterschiedliche Kommandos die Zustände der Led's oder die Zustände der Taster in den Kernellog geschrieben werden. Benutzen Sie dazu die zwei vordefinierten Commands *READ_LEDS* und *READ_BUTTONS* aus *omapL138Exp_io.h*.

Aufgabe 15: Testprogramm für Treiber Version 2

a) Schreiben Sie ein Testprogramm für die Funktionen *read* und *write*. Lassen Sie z.B. ein Lauflicht laufen und gleichzeitig geben Sie die Zustände der zwei Tastschalter über die Konsole aus. Nach 30 Sekunden soll das Programm terminieren.

b) Ändern Sie das Testprogramm so ab, dass das Programm auch terminiert, wenn beide Taster gedrückt werden.

c) Benutzen Sie in Ihrem Testprogramm auch die Funktion *unlocked_ioctl* und lesen Sie anschliessend den Kernellog, um das Resultat zu überprüfen.

6. Literaturverzeichnis

[1] DANIEL P. BOVET; MARCO CESATI; Understanding the Linux Kernel; 2002 O'Reilly Media, ISBN 0-596-00213-0

[2] JONATHAN CORBET; ALESSANDRO RUBINI; GREG KROA-HARTMANN; Linux Device Drivers; 2005 O'Reilly Media, ISBN 0-596-00590-3

[3] JÜRGEN QUADE; EVA-KATHARINA KUNST; Linux-Treiber entwickeln; 2006 dpunkt Verlag, ISBN 3-89864-392-1

[4] SREEKISHNAN VENKATESWARAN; Essential Linux Device Drivers; 2008 Prentice Hall, ISBN 0-13-239655-6

[5] WOLFGANG MAUERER; Linux Kernel Architecture; 2008 Wiley Publishing Inc, ISBN 978-0-470-34343-2

[6] Diverse Autoren; The Linux Kernel API; URL: http://www.kernel.org/doc/htmldocs/kernel-api/ (Stand 4.12.2013)

A. Anhang

A. Zusätzliche Informationen zum OMAP Board

Elektrische Beschaltung der relevanten Komponenten

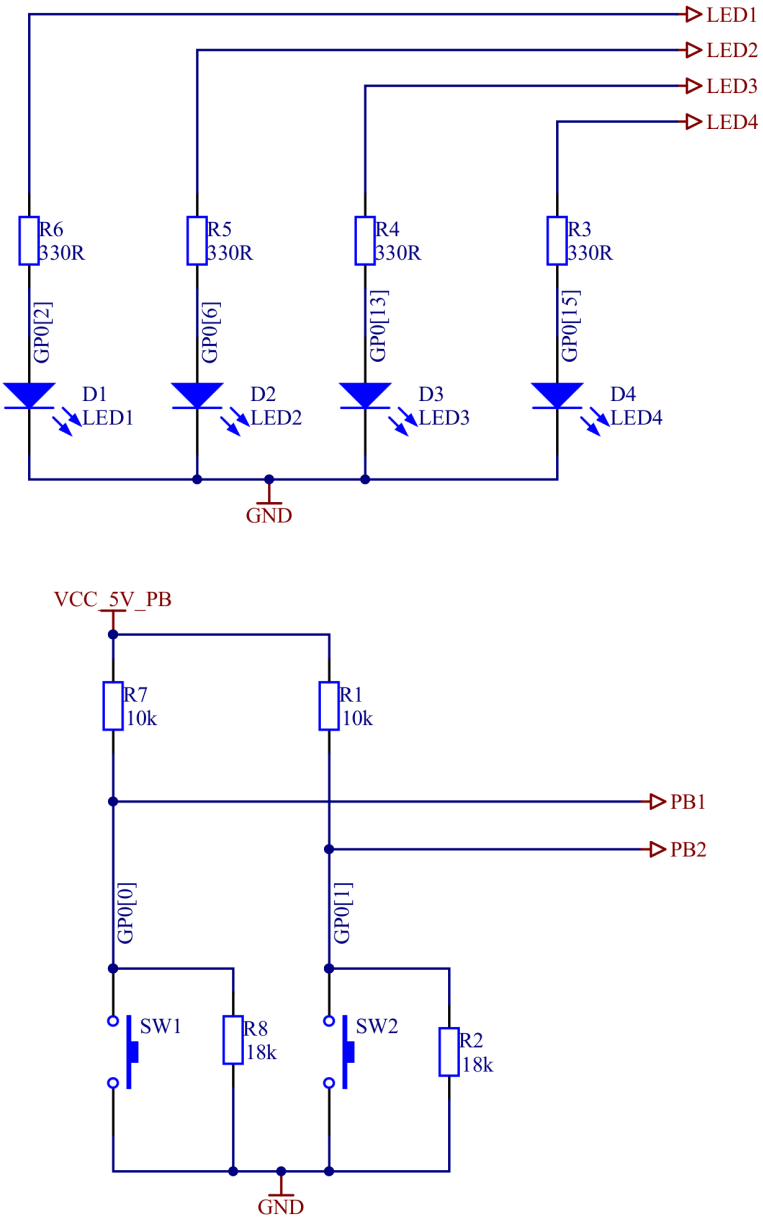

Abbildung A.1.: Beschaltung der LEDs und der Taster

Headerfile für den GPIO-Treiber

Listing 17: Headerfile omapL138Exp_io.h

```
#ifndef OMAPL138Exp_IO_H_
#define OMAPL138Exp_IO_H_

// Pinmux Register Definitions
#define PINMUX_REG_BASE      0x01C14120
#define PINMUX_REG_SIZE      0x0C
#define PINMUX0_OFFSET       0
#define PINMUX1_OFFSET       1
#define PINMUX2_OFFSET       2

// GPIO Register Definitions
#define GPIO_MEM_BASE        0x01E26000
#define GPIO_BANK01_BASE     (GPIO_MEM_BASE + 0x10)

#define GPIO_REG_BASE        GPIO_BANK01_BASE   // GPIO on J30
#define GPIO_REG_SIZE        0x28               // Memory Region GPIO Bank 0 and 1
#define GPIO_NAME            "GPIO"
#define GPIO_DDR_OFFSET      0
#define GPIO_DATA_OUT_OFFSET 1
#define GPIO_DATA_IN_OFFSET  4

// GPIO definitions
#define GPIO0_0              0
#define GPIO0_1              1
#define GPIO0_2              2
#define GPIO0_6              6
#define GPIO0_13             13
#define GPIO0_15             15

// Commands for ioctl
#define IOCTL_TYPE           'g'
#define READ_LEDS            _IOR(IOCTL_TYPE, 0, int)
#define READ_BUTTONS         _IOR(IOCTL_TYPE, 1, int)

#endif /*OMAPL138Exp_IO_H_*/
```

B. Operationen der Struktur file_operations

B.1. aio_fsync

```
int (*aio_fsync)(struct kiocb *, int);
```

Die asynchrone Variante von *fsync*.

B.2. aio_read

```
ssize_t (*aio_read)(struct kiocb *, char __user *,  size_t, loff_t);
```

Parameter:

`struct kiocb *`	file descriptor
`char __user *`	read data is written into this buffer
`size_t`	count, number of bytes to read
`loff_t *`	offset

Initiiert eine asynchrone Leseoperation. Zeigt diese Operation auf *NULL*, wird die (synchrone) *read* Operation aufgerufen.

B.3. aio_write

```
ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t,
         loff_t *);
```

Parameter:

`struct kiocb *`	file descriptor
`const char __user *`	write data
`size_t`	count, number of bytes to write
`loff_t *`	offset

Initiiert einen asynchronen Schreibervorgang.

B.4. check_flags

```
int (*check_flags)(int)
```

Diese Operation erlaubt die Überprüfung der Flags welche bei einem *fcntl* Aufruf übergeben werden.

B.5. dir_notify

```
int (*dir_notify)(struct file *, unsigned long)
```

Benutzen Applikationen *fcntl* um Veränderungen im Dateisystem festzustellen, wird diese Operation aufgerufen. Diese Operation ist nur für Dateisysteme von Interesse.

B.6. fasync

```
int (*fasync) (int, struct file *, int);
```

Diese Operation wird dazu gebraucht, um dem Gerät mitzuteilen, dass sich das *FASYNC* Flag geändert hat.

B.7. flush

```
int (*flush) (struct file *);
```

Die flush Funktion wird aufgerufen wenn ein Prozess seine Kopie eines Dateideskriptors schliesst und soll jede noch ausstehende Operation ausführen. Der *flush* Aufruf darf nicht mit dem *fsync* Aufruf verwechselt werden, welcher durch ein Benutzerprogramm aufgerufen wird. Gegenwärtig wird die *flush* Funktion nur von wenigen Treibern implementiert. Zeigt *flush* auf *NULL* wird der Aufruf vom Kernel ignoriert.

B.8. fsync

```
int (*fsync) (struct file *, struct dentry *, int);
```

Diese Operation ist das Backend des *fsync* Systemaufrufs. Ist dieser Zeiger *NULL*, wird *-EINVAL* zurückgegeben.

B.9. get_unmapped_area

```
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                   unsigned long, unsigned long,
                                   unsigned long);
```

Diese Operation wird gebraucht um einen passenden Speicherbereich zu finden, welcher in ein Speichersegment des entsprechenden Geräts abgebildet werden kann. Die meisten Treiber müssen diese Funktion nicht implementieren.

B.10. unlocked_ioctl

```
long (*unlocked_ioctl) (struct file *, unsigned int,
              unsigned long);
```

Parameter:

`struct file *`	file descriptor
`unsigned int`	ioctl number
`unsigned long`	parameter

Der *ioctl* Systemaufruf ermöglicht das Ausführen von gerätespezifischen Kommandos. Wird die Operation nicht unterstützt, wird *–ENOTT („No such ioctl for device“)* zurückgeben.

B.11. llseek

```
loff_t (*llseek) (struct file *, loff_t, int);
```

Parameter:

`struct file *`	file descriptor
`loff_t`	offset
`int`	whence (SEEK_SET, SEEK_CUR, SEEK_END)

Über die *llseek* Operation kann die aktuelle Lese- und Schreibeposition geändert werden. Die neue Position wird als positiver Wert zurückgeben. Der *lofft_t* Parameter ist ein „long Offset“ und mindestens 64 Bit breit. Tritt bei der Positionierung ein Fehler auf wird eine Fehlermeldung, repräsentiert durch einen negativen Wert, zurückgegeben. Ist der *loff_t* Funktionszeiger *NULL*, verändern Positionierungsaufrufe (eventuell auf unvorhersagbare Weise) den Positionszähler in der Struktur *file*.

B.12. lock

```
int (*lock) (struct file *, int, struct file_lock *);
```

Die *lock* Operation wird für die Implementierung von Dateisperren gebraucht. Locking ist für reguläre Dateien unverzichtbar, wird jedoch praktisch nie für Gerätetreiber gebraucht.

B.13. mmap

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

mmap fordert eine Abbildung von Gerätespeicher auf den Speicher des aufrufenden Prozesses an. Wird dieser Aufruf nicht unterstützt, wird *–ENODEV* zurückgeben.

B.14. open

```
int (*open) (struct inode *, struct file *);
```

Parameter:

`struct inode *`	inode pointer
`struct file *`	file descriptor

open ist jeweils die erste Operation die beim Gebrauch einer Gerätedatei aufgerufen wird. Wird diese Funktion durch den Treiber nicht angeboten, ist das Öffnen des Geräts immer erfolgreich. Der Treiber wird jedoch nicht darüber informiert.

B.15. owner

```
struct module *owner
```

Dieses Feld bezieht sich nicht auf eine eigentliche Dateioperation. Es ist ein Zeiger auf den Besitzer dieses Moduls und stellt sicher, dass ein Modul nicht entladen wird solange dessen Operationen benutzt werden. Normalerweise wird es über das Makro *THIS_MODULE* initialisiert, welches in <*linux/module.h*> definiert ist.

B.16. poll

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

Die *poll* Funktion ist das Backend für die drei Systemaufrufe *poll*, *epoll* und *select*. Sie werden für die Anfrage gebraucht, ob ein Lese- oder Schreibvorgang blockierend sein kann oder nicht. Der Aufruf von *poll* gibt eine Bit Maske zurück, die angibt ob ein nicht blockierender Aufruf möglich ist. Weiter wird der Kernel dadurch eventuell mit Informationen beliefert, die dazu gebraucht werden können, den Aufrufenden Prozess solange schlafen zu legen bis ein Gerät les- oder beschreibbar wird. Ist der Operationszeiger null wird angenommen, dass das Gerät nicht blockierend lesbar und beschreibbar ist.

B.17. read

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Parameter:

`struct file *`	file descriptor
`char __user *`	read data is written into this buffer
`size_t`	count, number of bytes to read
`loff_t *`	offset

Ermöglicht das Lesen von Daten aus dem Gerät. Zeigt diese Operation auf *NULL*, wird *–EINVAL* („Fehlerhaftes Argument“) zurückgeben. Ein nichtnegativer Rückgabewert gibt an, wie viele Bytes erfolgreich gelesen werden konnten.

B.18. readdir

```
int (*readdir) (struct file *, void *, filldir_t);
```

Parameter:

`struct file *`	file descriptor
`void *`	
`filldir_t`	

Dieses Feld wir nur für Dateisysteme gebraucht.

B.19. readv, writev

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
         loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
         loff_t *);
```

Diese beiden Methoden implementieren so genannte scatter/gather-Lese- und Schreiboperationen. Applikationen müssen von Zeit zu Zeit einzelne Lese- oder Schreib-Operationen durchführen, bei denen mehrere Speicherbereiche betroffen sind. Diese Systemaufrufe erlauben dies, ohne die Daten zusätzlich kopieren zu müssen.

B.20. release

```
int (*release) (struct inode *, struct file *);
```

Parameter:

`struct inode *`	inode pointer
`struct file *`	file descriptor

Die Operation *release* wird aufgerufen wenn die *file* Struktur freigegeben wird. Teilen sich mehrere Prozesse ein *file* Struktur (zum Beispiel nach einem *fork* oder *dump*) wird die *release* Operation erst aufgerufen, wenn alle Kopien geschlossen worden sind. Wie *open* kann *release* auch *NULL* sein.

B.21. sendfile

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t,
         void *);
```

Implementiert die Lesehälfte des *sendfile* Systemaufrufs, welcher Daten mit minimalem Aufwand von einem Dateideskriptor zu einem andern kopiert. Bei Gerätetreiber wird diese Operation normalerweise nicht implementiert.

B.22. sendpage

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                     loff_t *,int);
```

sendpage ist die andere Hälfte von *sendfile*, und ermöglicht das Senden von Daten zu einer korrespondierenden Datei. Wie *sendfile* wird diese Operation von Gerätetreibern normalerweise nicht implementiert.

B.23. write

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

Parameter:

`struct file *`	file descriptor
`const char __user *`	write data
`size_t`	count, number of bytes to write
`loff_t *`	offset

Schreibt Daten in das Gerät. Falls diese Operation auf *NULL* zeigt, wird dem Aufrufenden *–EINVAL* zurückgeben. War der Schreibvorgang erfolgreich, wird die Anzahl geschriebener Bytes zurückgegeben.