Linux Kernel-Module

Boguslaw Sylla und Patrick Schorn

Systemprogrammierung WS2008/09

Inhaltsverzeichnis

1 Was ist ein Kernel-Modul?

Kernel-Module erweitern einen Kernel um weitere Funktionalität und können beim Kernel je nach Bedarf ein- und ausgehängt werden, und das sogar ohne das laufende System zu unterbrechen. Ein Kernel-Modul wird zu Objekt-Code kompiliert und beim Kernel eingebunden, es ist aber keine selbständig laufende Datei, die beispielsweise unter unix-artigen Betriebssystemen im ELF-Format gespeichert werden.

In den meisten Fällen werden Kernel-Module als Gerätetreiber (Device Driver) eingesetzt, d.h. um Hardwarefunktionalität durch das Betriebssystem, und das heißt dessen Kernel, jeder möglichen Software auf einem System zur Verfügung zu stellen.

Dabei gibt es in gewachsenen und etablierten Kerneln wie dem von Linux, je nach Einsatzzweck, eine Vielzahl von APIs (mit eigenen C-Headern). Es ist beispielsweise etwas anderes, die Funktionalität zum automatischen Ein- und Aushängen von Geräten wie USB-Sticks zu programmieren, als z. B. ein neues Dateisystem vergleichbar zu ext3.

An diesen zwei Beispielen sehen wir auch, dass die Programmierung von Kernelmodulen keineswegs immer etwas mit Hardwareprogrammierung zu tun haben muss. ext3 ist beispielsweise unabhängig vom Festplattentyp, solange die Festplattenhersteller bestimmte Spezifikationen bei der Hardwareherstellung beachten.

Da Kernel-Module, selbst wenn sie nicht für ein bestimmtes Gerät geschrieben wurden, zwischen Hard- und Software liegen, kann es sein, dass es mehrere ähnliche Kernel-Module mit unterschiedlich großem Funktionalitätsumfang geben kann. Beispielsweise kann das bei Closed Source Grafiktreibern sein oder bei großen Archivierungssystemen. Bei der Kernelmodul-Programmierung für eine Serie ähnlicher Geräte ist also eine weitere Modularisierung, d. h. Aufteilung in mehrere Module denkbar und in Erwägung zu ziehen.

In diesem Projekt geht es nicht um Hardwareherstellung und deren zu beachtende Spezifikationen, sondern um einen Überblick über die APIs von Linux 2.6 Kerneln. Zu erwähnen wäre hier, dass andere unix-artige Kernel wie BSD andere APIs haben und Kernel-Module nicht unter allen Unix-Systemen und Linux portabel sind, so dass man sie für jedes Betriebssystem ganz neu programmieren muss. Man sollte das bei der Konzeption eines Kernel-Moduls beachten, um sich da nicht zu viel Arbeit zu machen.

Aber selbst wenn man Kernel-Module für ein bestimmtes Betriebssystem wie Linux schreibt, kann es zu einem API-Wechseln kommen, so dass ein Kernel-Modul mit einer neueren Kernel-Version nicht mehr zusammenarbeitet und umgeschrieben werden muss. Das bedeutet aber nicht, dass Diskettenlaufwerke irgendwann nicht mehr funktionieren, denn die API-Schreiber sorgen bei einer Umstellung dafür, dass eine neue API mit alten Geräten funktioniert. Nur die Software-Schreiber, d. h. Kernel-Modul- oder Tool-Schreiber, müssen ihre Programme an neue APIs anpassen. Bei älteren APIs hat man da weniger zu befürchten als bei ganz neuen.

Bei der Programmierung von Kernel-Modulen, insbesondere für Geräte, sollte man sich nur auf die bloße Bereitstellung von Funktionalitäten konzentrieren und nicht darauf, wer später wie und warum zugreifen darf oder nicht. Diese Art von Politik ist bei jedem Betriebssystem über der Kernel-Modul-Programmierung angesiedelt und hat mit ihr nichts zu tun. Hier ist natürlich nicht das gegenseitige Ausschließen von Funktionen/Threads auf Programmebene gemeint, sondern die Zugriffsrechteverwaltung auf höherer Betriebssystemebene.

Oftmals werden neben der Programmierung eines Kernel-Moduls auch Tools zu dessen Konfiguration programmiert, ohne dass diese gleich zu Kernel-Modulen erklärt werden. Das ist unter Linux auch vermutlich besser so, weil hier Kernel-Module nur bestimmte Funktionen der APIs nutzen dürfen und keine unsicheren Funktionen beispielsweise der Standard-Library. Es steht somit ein ähnlicher Befehl wie printf zur Verfügung, aber das echte printf selbst kann man in einem Kernel-Modul nicht nutzen. Bei Konfigurationstools für Module hat man dagegen volle Freiheit.

1.1 Unterschiede zwischen Kernel-Modulen und normalen Programmen

Zu allererst muss man bei Programmen zwischen einer Applikation und einem Tool unterscheiden.

Während ein Tool ein einfaches Programm mit wenig Funktionalität in der Shell darstellt und nicht nur von Menschen, sondern auch oft von anderen Programmen verwendet wird, ist eine Applikation eine Sammlung von Tool-Funktionalitäten, die nur von Menschen verwendet wird.

Ein Kernel-Modul dagegen ist eine Art wartender Service im System, der eine textuelle Bedienung in der Shell ermöglichen kann aber nicht muss und meist nur von Tools oder Applikationen verwendet wird.

Ein Tool arbeitet meist vom Start des Prozesses bis zu seinem Beenden. Der Prozess eines Kernel-Moduls wird mit dem Einhängen des Kernel-Moduls gestartet und wartet dann bis irgendwelche Anfragen kommen um diese dann abzuarbeiten.

Der Anfang und das Ende des Prozesses eines Kernel-Moduls wird meist mit dem Starten und dem Runterfahren des Betriebssystems initialisiert. Während der Prozess eines Tools nur dann existiert, wenn er gebraucht wird.

Wenn ein Programm einen *segmentation fault* hat, wird es einfach beendet und es passiert nichts. Bei einem *kernel fault* eines Kernel-Moduls funktioniert das betreffende Modul nicht oder das ganze System bricht sogar irreparabel zusammen. Das Debuggen solcher Fehler ist anders als bei normalen Programmen.

Ein Kernel-Modul läuft nicht wie ein Tool im *User Mode* (*user space*) sondern im *Kerlen Mode* (*kernel space*, *supervisor mode*). Dazu muss man wissen, dass unix-artige Betriebssysteme wie Linux, mindestens diese zwei Modi kennen, die sich darin unterscheiden, dass mit jedem Modus, der nähe am eigentlichen Kernel ist, auch umso mehr systemkritische Funktionalitäten zur Verfügung stehen. Im Grunde ist das ein Zwiebel-Schalen-System mit immer engeren Kreisen des Vertrauens, um das Betriebssystem vor gewollten oder ungewollten Manipulationen zu schützen.

Es gibt nur wenige Tore zwischen *User Mode* und *Kernel Mode*. Und die Aufgabe eines Kernel-Moduls kann darin bestehen solche Tore in Form von *System Calls* und *Interrupts* zu realisieren.

Bei der Entwicklung von Kernel-Modulen, muss man auch immer beachten, dass ein Kernel-Modul mehrfach zur gleichen Zeit benutzt werden könnte und das zu unerwartetem Verhalten führen kann. Deswegen muss man Kernel-Module immer unter dem Aspekt von *Konkurrierenden Anfragen* entwickeln und dafür geeignete System-Funktionen und Algorithmen einsetzen.

Der Kernel nutzt evtl. einen sehr kleinen Stack und kann nur wenige oder nur kleine automatische Variablen von Funktionen speichern. Deswegen sollte man keine großen Variablen anlegen und falls

man das muss, dann sollten sie dynamisch während einer laufenden Funktion angelegt und wieder freigegeben werden.

In den zahlreichen APIs verbergen sich auch Funktionen mit zwei Unterstrichen ("__"), die signalisieren, dass sie Hilfsfunktionen für die offiziellen Funktionen der APIs sind und nur mit größter Vorsicht benutzt werden sollten.

Kernel-Module nutzen keine Floating-Point-Arithmetik, weil das zu lange dauern würde und es dafür normalerweise keinen Bedarf gibt.

1.2 Grobe Einteilung eines Kernels

- *Prozess Management*

Der Scheduler eines Kernels managt die Verteilung von CPU-Ressourcen für Prozesse und kümmert sich um die Kommunikation zwischen den Prozessen über Signale, Pipes, Interprozesskommunikations-Primitiven etc.

- *Memory Management*

Der Kernel weist jedem Prozess virtuellen Speicher zu, um die Limitierung durch den realen Speicher zu umgehen und kümmert sich um jede damit verbundene Funktionalität die von Memory-Schedulern genutzt wird. Ein Memory-Scheduler ist üblicherweise eine bei der Kompilierung eines Programms eingebundene Bibliothek die auch eine andere als die bei der Verwendung von GCC sein kann (siehe Hoare). So eine Memory-Scheduler-Bibliothek stellt Funktionen wie malloc, calloc, realloc und free bereit. Die Benutzung eines anderen Memory-Schedulers kann darin begründet sein, dass der von GCC zwar schneller sein kann, aber dafür auch bis zu 50 mit free freigegebene MB weiterhin für den Prozess reserviert hält, um weitere Allozierungen zu beschleunigen, und nicht an das Betriebssystem für andere Prozesse frei gibt. So etwas passiert unter Linux garantiert ab einer bestimmten Größe von insgesamt alloziertem Speicher bei realloc, anscheinend aber nicht bei malloc und calloc.

- *Filesystems*

Speziell Linux ist stark dateisystem-orientiert und benötigt dazu eine bestimmte Verzeichnis-Hierarchie auf dem Medium auf dem es läuft. Insbesondere Verzeichnisse wie /proc und meist /medium zeigen das. In /proc beispielsweise kann jeder Prozess oder auch der User selbst über jeden gestarteten Prozess Laufzeitinformationen einsehen, als ob es sich um einfache Textdateien handeln würde. Zudem kann man darin auch Kernel-Module einbinden um weitere Informationen oder Funktionalitäten zur Verfügung zu stellen und dabei mit cat oder Pipes operieren.

- *Device Control*

Mit Ausnahme der CPUs, der Speicherriegel und weniger anderer Hardware-Einheiten, braucht jedes Gerät einen Device Driver der als Kernel-Modul für den Kernel erreichbar ist. Hier knöpft dieses

Projekt an.

- *Network*

Jede Netzwerkaktion muss durch den Kernel gemanagt werden, da das Netzwerk allen interessierten Prozessen zur Verfügung stehen muss.

1.3 Grobe Einteilung von Kernel-Modulen

Hier folgt nur eine sehr grobe Einteilung von Kernel-Modulen.

- *Character Devices*

So ein Kernel-Modul ist ein Stream von Bytes der normalerweise mindestens *open*, *close*, *read*, und *write* implementiert und sich deswegen fast wie eine einfache Datei verhält. Allerdings kann man bei einer gewöhnlichen Datei auf jedes Byte direkt zugreifen (Random Access), während man bei einem solchen Character Device nur hin- und zurück bewegen kann ähnlich *putc* und *getc*. Beispiel für solche Kernel-Module sind */dev/console* und */dev/tty*. Alles was sich im Verzeichnis */dev* befindet wird als *Filesystem Node* bezeichnet.

- *Block Devices*

Anders als Character Devices geht diese Kernel-Modul-Art, oft eine Festplatte, blockweise beim Lesen und Schreiben vor. Üblicherweise sind das 512 Byte oder etwas Zweier-Exponentielles. Dennoch verhalten sich beide -Kernel-Modul-Arten beim Zugriff auf so einen Kernel-Modul selbst genauso, d. h. man kann auch hier außerhalb des Kernel-Moduls Zeichenweise zugreifen, obwohl das Kernel-Modul bzw. der Kernel Blockweise zugreift. Auch ein Block Device wird als Filesystem Node in */dev* repräsentiert.

- *Network Interfaces*

Bei dieser Art von Kernel-Modul wird über Pakete kommuniziert. Auch hier wird zwar ein eindeutiger Name wie *eth0* zur Verfügung gestellt allerdings nicht als Filesystem Node in */dev*.

- *File Systems*

Ein Dateisystem wir ext3 arbeitet hardwareunabhängig auf Block Devices und regelt die Kommunikation die nötig ist damit ein Block Device mit Tools wie *cp* und *ls* umgehen kann. Es geht insbesondere um das Management von Inodes, Verzeichnisse, Dateien und Superblocks. Damit ein Dateisystem ordentlich funktioniert, muss eine Vielzahl an Funktionen bereitgestellt werden, die der Kernel von so einem Kernel-Modul erwartet.

Das folgende Bild zeigt einige Kernel-Modul-Arten, die jeweils einen unterschiedlichen Schwerpunkt haben und daher auch unterschiedliche APIs unter Linux nutzen müssen.

Eine Unterteilung z. B. in USB-Module, Serialport-Module, SCSI-Module ist wenig sinnvoll. Ein USB-Modul kann nämlich alles drei sein: Character Device (USB Serial Port), ein Block Device (USB Memory Card Reader) oder auch ein Network Interface (USB Ethernet Interface).

1.4 Sicherheit bei Kernel-Modulen

Wenn ein offizieller Linux Kernel auf einem System läuft, dann können Kernel-Module nur im Superuser-Modus ein- und ausgehängt werden. Das kann man zwar ändern, auch durch ein Tool oder ein Kernel-Modul das dafür programmiert wurde, man sollte es aber nicht. Denn hat ein Kernel-Modul ein Sicherheitsloch, hat der Kernel ein Sicherheitsloch und damit ein kompletter Rechner oder sogar ein ganzes Rechner-Netzwerk. Und das nur durch ein winziges Kernel-Modul, das sich mit allen anderen Kernel-Modulen und dem Kernel selbst den kompletten Speicherraum teilt und damit jedes erdenkliche Recht hat.

Ab dem einhängen eines Kernel-Moduls, besteht die Gefahr ein Sicherheitsloch verursacht zu haben und über andere Kernel-Module ein Linux-System oder auch reale Hardware dauerhaft zu beschädigen.

Möchte man wirklich für die Öffentlichkeit Kernel-Module zur Verfügung stellen, dann muss man sich auch mit den speziell in C vorkommenden sicherheitsanfälligen Programmiermethoden und Gefahren wie *Buffer Overrun* auseinandersetzen. Hier verweisen wir auf weitere Literatur zu diesem für Kernel-Module sehr wichtigen Thema.

1.5 Lizenzierung

Es ist umstritten ob Kernel-Module für den Linux-Kernel nicht eigentlich genau wie der Kernel selbst unter der GPL mit frei zugänglichem Quellcode veröffentlicht werden müssten. Bislang werden binäre Kernel-Module aber geduldet. Wobei dabei immer die Gefahr besteht, dass ein Kernel-Modul-Herstellen für eine neuere Linux-Version keine neuen Kernel-Module mehr liefert. Eine Weiterpflege ohne freien Quellcode ist dann ohne sehr aufwändige Disassemblierung unmöglich.

Der einzige rationale Grund für ausschließlich binär verbreitete Kernel-Module, ist nicht die spionierende Konkurrenz, sondern mögliche Geschäftsschädigung durch korrumpierte Kernel-Modul-Versionen aus dem Internet.

Entwickelt man Kernel-Module nur zum Eigengebrauch ohne diese zu verbreiten, so ist die Lizenzierung ohnehin völlig irrelevant, da der Urheber eines Kernel-Moduls selbst nach GPL jedes Recht hat, inklusive dem Gebrauchsrecht, da die GPL erst ab einer Veröffentlichung gilt und nicht schon ab der ersten Kompilierung oder dem ersten Byte in der Quellcode-Datei.

Mit anderen Worten darf beispielsweise ein Forma Kernel-Module programmieren und für sich selbst nutzen ohne den Quellcode veröffentlichen zu müssen, sogar wenn irgendwann die GPL definitiv auch

für Kernel-Module gelten sollte.

Dies gilt aber nicht für mit GCC, oder unter Windows mit MinGW, kompilierte Programme. Nutzt man die darin enthaltene Standard-Library beim Kompilieren, ist die ganze ausführbare Datei, wie die Bibliothek selbst, GPL und man muss den Quellcode beim Veröffentlichen auch freigeben.

Mit anderen Worten wird es zwar geduldet Kernel-Module nur binär zu verbreiten. Bei den Konfigurationstools, die man gegen die Standard-Library von GCC kompiliert hat, muss man dagegen die Quellcodes freigeben. Die ausführbare Datei enthält dann nämlich teilweise die Standard-Library. Kernel-Module werden hingegen gar nicht mit der Standard-Library kompiliert.

Viele Programmierer meinen aber, dass selbst wenn man einen Nicht-GPL Compiler statt GCC nimmt um ein Kernel-Modul zu kompilieren, müssen Teile der unter GPL lizenzierten Linux-Kernel-APIs hinein kompiliert werden, womit dann jedes Kernel-Modul unter die GPL fällt sobald es veröffentlicht wird. Und somit sind ausschließlich binär verbreitete Kernel-Module wie Grafiktreiber tatsächlich nur geduldet.

2 Welche Voraussetzungen zum Entwickeln von Kernel-Modulen gibt es?

Es ist möglich für eine Kerner-Version Kernel-Module zu entwickeln, ohne den betreffenden Kernel auf dem System laufen zu lassen. Dennoch ist es besser ihn zu installieren, weil man damit eine komfortable Testumgebung hat.

Zusätzlich muss man aber auf jeden Fall zur Ziel-Kerner-Version passende Kernel-Quell-Codes installieren. Entweder von www.kernel.org oder durch einen Packet-Manager wie Synaptic unter Debian oder Ubuntu.

Desweiteren werden auch die passenden Header benötigt, was also bei einem Packet-Manager mindestens zwei zu installierende Pakete extra macht, falls der zusätzliche Ziel-Kernel schon installiert ist, wovon wir im Nachfolgenden immer ausgehen.

Um die passenden Quellen und Header zu installieren, sollte man zuerst seine laufende Kernel-Version herausfinden.

```
user@linux:~$ uname -r

2.6.24-22-generic
```

Ein komplettes Entwicklungs-Linux-System sollte man vielleicht zusätzlich besser ohne sensible Daten in einer Virtuellen Maschine laufen lassen. Es gibt beispielsweise eine kostenlose von Sun namens VirtualBox.

In *Documentation/Changes* der Kernel-Quellen kann man nachlesen, welche Versionen von Programmen

oder Bibliotheken für die jeweilige Kernel-Version nötig sind. Zu alt oder auch zu neu kann da manchmal ein Problem sein.

Auf jeden Fall sollten GCC und Make installiert sein.

2.1 Beispiel: Ein Hello-World-Modul

Nachfolgend wird zur Einführung ein kurzes Beispiel für ein Kernel-Modul unter Linux vorgestellt.

```
#include <linux/init.h>
#include <linux/module.h>

static int hello_init(void)
{
     /* TODO: make preliminaries here */
     printk(KERN_ALERT "Hello, world\n");
     return 0;
}

static void hello_exit(void)
{
     /* TODO: make cleanup here */
     printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_DESCRIPTION("A simple hello world module hwm1.");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Milhouse van Houten");
MODULE_ALIAS("hwm1");
```

Die Zwei Header <*linux/init.h*> und <*linux/module.h*> sind für jedes Kernel-Modul Pflicht. Je nach API-Gewichtung folgen aber normallerweise weitere Header.

In diesem Beispiel werden zwei Funktionen definiert, die jeweils eine Zeichenkette beim Ein- bzw. Aushängen des Kernel-Moduls ausgeben. Dazu müssen sie durch die zwei Makros *module_init* und *module_exit* für diesen Zweck bestimmt werden. Von den beiden ist nur *module_init* Pflicht, da es keine Main-Funktionen in Kernel-Modulen gibt und der Kernel wissen muss, wo er beginnen soll. Gibt man mit *module_exit* keine Cleanup-Funktion an, dann erlaubt der Kernel das Aushängen eines Moduls nicht.

Hier wird noch ein weiteres Pflicht-Makro namens *MODULE_LICENSE* verwendet, um die Lizenz für dieses Kernel-Modul abfragbar zu machen. Hängt man ein Kernel-Modul ohne dieses Makro ein, beschwert sich der Kernel mit einer Warnung-Meldung darüber, weil das Kernel-Modul als *Proprietary* gilt. Die vom Kernel als Open Source erkennbaren Zeichenketten für dieses Makro sind *GPL* (für jede GPL-Version), *GPL v2* (nur für GPL-Version 2), *GPL and additional rights*, *Dual BDS/GPL* und *Dual MPL/GPL*.

Die anderen drei hier verwendeten Makros die mit *MODULE_* beginnen und von denen es noch mehr gibt, sind selbsterklärend. Sie sind nicht Pflicht aber sicher wünschenswert. Alle diese Module-Makros werden neuerdings an das Ende von Quell-Dateien geschrieben.

Wundern könnte man sich bislang nur, über die Verwendung der Funktion *printk* statt *printf*. Bei der Kernel-Modul-Entwicklung nutzt man nur speziell dafür vorgesehene Header und die Standard-Library gehört nicht dazu. Wenn aber ein Kernel-Modul beim Kernel eingehängt wird, dann stehen ihm alle auch dem Kernel selbst bekannten Symbole zur Verfügung, wozu dann auch *printk* gehört.

Die Zeichenkette die sich hinter *KERN_ALERT* verbirgt, signalisiert dem Kernel eine hohe Priorität für die beiden Verwendungen von *printk*, da die Ausgabe der Nachrichten sonst ungewiss ist, je nach Kernel-Einstellungen.

Dieses einfache Kernel-Modul kann mit einem einzeiligen Makefile (namens Makefile) kompiliert werden. Sofern die Quell-Datei des Beispiel-Moduls *hello.c* lautet, sieht der Inhalt des Makefiles wie folgt aus.

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

Nachdem eine Kernel-Modul kompiliert wurde, kann es mit Super-User-Rechten eingehängt und ausprobiert werden. Dazu benutzt man das Kommando *su* und gibt sein Root-Passwort ein oder man gib nur *sudo su* ein, worauf hin man nur sein User-Passwort eingeben braucht mit dem man sich beispielsweise bei Ubuntu angemeldet hat.

Nachfolgen kompilieren wir das Kernel-Modul, melden uns als Super-User mit *sudo su* an, hängen das Kernel-Modul mit insmod ein und hängen es gleich wieder mit *rmmod* aus.

```
user@linux:~$ make
user@linux:~$ sudo su
user@linux:~# insmod ./hello.ko

Hello, world

user@linux:~# rmmod hello

Goodbye cruel world

user@linux:~# exit
user@linux:~$
```

Bei der Benutzung eines Kernel-Moduls muss man natürlich nicht die ganze Zeit als Super-User angemeldet zu sein. Das gilt nur fürs Ein- und Aushängen von Kernel-Modulen.

3 Wie werden Kernel-Module kompiliert?

Der Build-Prozess eines Kernel-Moduls ist mittlerweile einfacher geworden als bei älteren Linux-Versionen, aber dennoch kompliziert genug, dass man ihn bei Weitem nicht vollständig vorstellen

kann. Wer wirklich mehr darüber erfahren möchte, sollte in *Documentation/kbuild* der Kernel-Quellen nachlesen.

- Angenommen, ein Kernel-Modul besteht nur aus einer Quell-Datei namens *modul.c* und soll auch *modul.ko* heißen, dann lautet der Inhalt des Makefiles wie folgt.

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

- Wenn ein Kernel-Modul anders heißen soll, als die Quell-Datei oder aus mehreren Quell-Dateien besteht, z.B. aus *file1.c* und *file2.c*, dann wird es komplizierter.

```
obj-m := modul.o
module-objs := file1.o file2.o
```

- Befinden sich die Quell-Codes des Ziel-Kernels in *~/kernel-2.6* ohne dass er auf dem System läuft, kann man Make angeben welche Quell-Codes verwendet werden sollen.

```
user@linux:~$ make -C ~/kernel-2.6 M=`pwd` modules
```

Dabei zeigt das Flag *-C* in welches Verzeichnis Make wechseln soll und *M=* in welches zurückgekehrt werden soll. Das Wort *modules* bezieht sich auf die Zeile mit *obj-m* in dem Makefile und signalisiert Make, was es bauen soll.

- Die Kernel-Entwickler haben ein einfaches Idom für Makefiles entwickelt.

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Dieses Makefile ruft man entweder ohne Argumente, wobei dann gegen die Quell-Codes des laufenden Kernels kompiliert wird, oder man ruft Make mit dem Argument *KERNELDIR=* auf und gibt das Verzeichnis der Quell-Codes eines Ziel-Kernels an. Welchen Aufruf man auch wählt, am Ende wird das Default-Target aufgerufen, das wie man sieht Make nochmal wie im Beispiel zuvor aufruft. Dieses Makefile kann leicht an eigene Kernel-Module angepasst werden, wobei dann hauptsächlich die Zeile mit *obj-m* abzuändern ist.

- Normalerweise sind Makefiles deutlich grösser und haben meist ein Clean-Target und ein Helpscreen-Target.

```
obj-m    += hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD  := $(shell pwd)

help:
    @echo "Usage:"
    @echo " make all        # Compile and generate all"
    @echo " make clean      # Remove all unnessesary files"
    @echo " make help       # This help screen"
all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

4 Wie werden Kernel-Module im Linux-Kernel ein- und ausgehängt?

- Geladen werden Kernel-Module mit *insmod*, das auch Kommandozeilen-Argumente annehmen kann und wie *ld* ein Kernel-Modul beim Kernel einhängt und dessen Symbole verlinkt.

- Noch etwas komfortabler ist *modprobe* beim Einhängen. Wenn ein Kernel-Modul im Kernel nichtaufgelöste Symbole referenziert, dann bricht *insmod* ab. Wogegen *modprobe* nach Modulen mit den fehlenden Symbolen im Standard-Verzeichnis für installierte Kernel-Module sucht und diese ggf. auch einhängt und bei Erfolg nicht abbricht einzuhängen. Um Abhängigkeiten muss sich ein User also nicht kümmern, wenn alle nötigen Kernel-Module installiert sind und er *modprobe* nutzt.

- Mit *rmmod* kann ein Kernel-Modul ausgehängt werden, aber nur wenn der Kernel der Meinung ist, dass das Kernel-Modul gerade nicht gebraucht wird und wenn das Kernel-Modul nicht so geschrieben wurde, das Aushängen zu verhindern. Das Aushängen kann auch erzwungen werden, was natürlich nicht empfohlen wird.

- Möchte man nachsehen welche Kernel-Module gerade eingehängt sind, kann man mit *cat* in der virtuellen Datei */proc/modules* des *Proc File System* nachgucken oder dies mit *lsmod* erledigen. Die Ausgaben unterscheiden sich aber geringfügig. Im Verzeichnis */sys/module* des *Sysfs Virtual Filesystem* kann man weitere Hinweise zu eingehängten Kernel-Modulen finden.

5 Wie entwickelt man Kernel-Module für mehrere Kernel-Versionen?

Man muss beachten, dass ein Kernel-Modul für jede neue Kernel-Version neu kompiliert werden muss. Was bedeutet dass die beiden zueinander passen müssen, sonst sieht man eine Fehlermeldung wie die folgende.

```
user@linux:~$ sudo su
user@linux:~# insmod hello.ko

Error inserting './hello.ko': -1 Invalid module format
```

Weitere Informationen erhält man bei Fehlermeldungen immer aus der Log-Datei des Systems, normalerweise ist das */var/log/messages*.

Möchte man nur ein Kernel-Modul für mehrere Kernel-Versionen schreiben, dann sollte man die Teile seines Quell-Codes die unterschiedlich für mehrere Kernel-Versionen sind, jeweils in eigene Header und damit C-Dateien auslagern.

Um nun auf unterschiedlichen Kernel-Versionen zu reagieren, die das Kernel-Modul einbinden sollen, muss man sich der Präprozessor-Anweisungen *#ifdef...#elif...#else...#endif* bedienen und der Versions-Makros des Headers *linux/version.h*, der aber schon mit dem üblichen Header *linux/module.h* eingefügt wird.

Am besten ist es, wenn man versionsabhängigen Code hinter Low-Level Makros oder Funktionen verbirgt, die High-Level-Funktionen dann nutzen können. Somit steigert man die Lesbarkeit des Codes.

Die drei wichtigsten Makros sind die folgenden.

- UTS_RELEASE

Hinter diesem Makro verbirgt sich die Zeichenkette der Kernel-Quellen mit denen das Kernel-Modul kompiliert wurde, z. B. "2.6.10".

- LINUX_VERSION_CODE

Dieses Makro gibt eine binäre Repräsentation der Version des Kernels, der versucht das Kernel-Modul einzuhängen. Beispielsweise ist der Code der Kernel-Version "3.2.10" in drei Bytes 0x03020A.

- KERNEL_VERSION(major,minor,release)

Man kann auch leicht eine beliebige Versionsnummer in ihre binäre Repräsentation umwandeln, die man dann mit LINUX_VERSION_CODE vergleichen kann. Aus KERNEL_VERSION(2,6,10) wird dann 0x02060A.

Man schnürt so ein Quellcode-Paket für mehrere Kernel-Versionen, z. B. für 2.6.11 und 2.6.10. Wenn dann jemand versucht mit Kernel-Quellen der Version 2.6.10 zu kompilieren, wird sein Compiler mit den Präprozessor-Anweisungen und den richtigen Versions-Makros zum richtigen Quell-Code-Strang navigiert. Dabei macht es nichts, dass die anderen Quell-Code-Strängen für andere Kernel-Quell-Versionen nicht kompilierbar sind, weil sie vom Compiler einfach überlesen werden.

6 Wie entwickelt man Kernel-Module für mehrere Plattformen?

Wenn man Kernel-Module für unterschiedliche Prozessoren entwickelt, kann man seinen Quellcode auf gleiche Weise wie für unterschiedliche Kernel-Versionen aufteilen.

Je nach Plattform ist dann auch evtl. eine Unterteilung in weitere Targets beim Makefile sinnvoll, die einen Compiler wie GCC zusätzlich mit plattform-optimierenden Attributen aufrufen.

7 Wie kann man Kernel-Module auf einander aufbauend entwickeln?

Es ist möglich Kernel-Module beispielsweise in High-Level-Module und Low-Level zu unterteilen. Ein High-Level-Modul kann nur allgemeine Funktionalität zur Verfügung stellen und dabei evtl. nach CPU, PCI-Karte, Modem etc. auf weitere spezielle Kernel-Module zurückgreifen.

Dazu müssen den High-Level-Modul aber die nötigen Symbole (Funktionsnamen, Variablen, Makros) bekannt sein. Die müssen in Low-Level-Modulen dem ganzen Kernel bekannt gemacht werden, sonst würde der Kernel mit Symbolen die sich womöglich auch noch gleichen, zugemüllt werden. Dazu gibt es zwei Makros die außerhalb von Funktionen, also im globalen Raum eines Kernel-Moduls platziert werden müssen.

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

Die Version mit *_GPL* macht ein Symbol nur Kernel-Modulen mit GPL-Lizenz verfügbar.

8 Was ist zu tun, wenn Fehler auftreten?

Wenn Fehler auftreten, z. B. der Allozierung von Speicher, dann muss bei der Kernel-Modul-Entwicklung immer auf Fehler reagiert werden um beispielsweise nichtnutzbar belegten Speicher wieder frei zu geben. Ein Kernel-Modul muss nicht unbedingt nach einem Fehler ausgehängt werden, sondern kann durchaus oftmals mit weniger Funktionalität weiter laufen. Darum muss sich aber immer der Programmierer kümmern und bei Fehlern immer aufräumen.

Bei der Kernel- und der Kernel-Modul-Entwicklung wird oftmals zum Aufräumen der goto-Befehl benutzt, weil das den Quellcode deutlich einfacher macht.

```
<linux/errno.h>

int my_init_function(void)
{
```

```
    int err;

    /* registration takes a pointer and a name */
    err = register_1(ptr1, "skull");
    if (err) goto fail_1;

    err = register_2(ptr2, "skull");
    if (err) goto fail_2;

    err = register_3(ptr3, "skull");
    if (err) goto fail_3;

    /* success */
    return 0;

    /* cleanup */
    fail_3: unregister_2(ptr2, "skull");
    fail_2: unregister_1(ptr1, "skull");
    fail_1: return err;
}

void my_cleanup_function(void)
{
    unregister_3(ptr3, "skull");
    unregister_2(ptr2, "skull");
    unregister_1(ptr1, "skull");
    return;
}
```

Im Allgemeinen bedeuten negative Rückgabewerte einen Fehler. Dazu hält der Header <*linux/errno.h*> Makros wie ENODEV und ENOMEM bereit.

Interessant, weil kurz, ist der Cleanup-Bereich in der Init-Funktion mit den Sprungmarken für die Gotos der If-Anweisungen. Hätte man hier auf den Gebrauch der Gotos verzichtet, dann hätte man die Cleanup-Funktionen in den If-Anweisungen doppelt und dreifach schreiben müssen.

9 Wie kann man Parameter von Kernel-Modulen nutzen?

Beim Einhängen eines Kernel-Moduls mit *insmod* und *modprobe* können Parameter gesetzt werden, die das Kernel-Modul vorher definiert hat. Je nach festgelegten Zugriffsrechten für die Parameter sind diese dann auch später änderbar oder nur lesbar.

Zusätzlich kann *modprobe* aber auch Parameter aus seiner Konfigurationsdatei */etc/modprobe.conf* lesen.

Angenommen es gäbe in einem Hello-World-Modul zwei Parameter *howmany* als Angabe wie oft ein Begrüßungssatz ausgegeben werden soll und *whom* als Angabe wer begrüßt werden soll. Dann kann man die beiden Parameter z. B. mit *insmod* wie folgt setzen.

```
user@linux:~$ sudo su
user@linux:~# insmod hello howmany=3 whom="Master"
```

Im Hello-World-Modul können die Parameter mit Default-Werten definiert werden.

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Wie man erkennen kann, werden Parameter dem Kernel mit *module_param* bekannt gemacht. Wobei das erste Argument das Symbol der Variable ist, das zweite ist der Datentyp und das dritte sind die Zugriffsrechte für den Parameter.

Als Datentypen stehen die folgenden zur Verfügung.

- *bool, invbool*

Ein boolescher Parameter für TRUE und FALSE bzw. invertierter boolescher Wert mit umgekehrter Bedeutung. Die zugehörige C-Variable ist *int*.

- *charp*

Ein Zeiger auf ein Char-Array für Zeichenketten.

- *int, long, short, uint, ulong, ushort*

Ganzzahlige Datentypen variabler Länge mit und ohne Vorzeichen.

Möchte man wirklich andere als die unterstützten Datentypen für Parameter, dann sollte man sich in <*linux/moduleparam.h*> umschauen.

Für die Zugriffsrechte sollte man die Makros des Headers <*linux/stat.h*> nutzen. Meist sind das *S_IRUGO* Parameter die gelesen, aber nicht geändert werden können und zum anderen *S_IRUGO|S_IWUSR* damit zumindest ein Rooter Veränderungen vornehmen kann.

Es ist auch möglich kommaseparierte Array-Parameter mitzugeben.

```
module_param_array(name,type,num,perm);
```

Hier gibt *num* zusätzlich die Anzahl der anzugebenden Werte an.

10 Was sind Device Numbers?

Von Linux erkannte und mountbare Geräte sind im Verzeichnis */dev/* zu finden. Bei der Ausgabe der Dateien in diesem Verzeichnis mit *ls -l*, sieht man bei jedem Device ganz links beispielsweise ein *c* für Character Driver oder ein *b* für Block Driver.

```
brw-rw---- 1 root disk 8,  0 2008-12-01 17:00 sda
```

```
brw-rw---- 1 root disk 8,  1 2008-12-01 17:00 sda1
brw-rw---- 1 root disk 8,  2 2008-12-01 17:00 sda2
brw-rw---- 1 root disk 8, 16 2008-12-01 16:05 sdb
brw-rw---- 1 root disk 8, 17 2008-12-01 16:05 sdb1
crw-rw---- 1 root root 4,  0 2008-12-01 17:00 tty0
crw------- 1 root root 4,  1 2008-12-01 16:01 tty1
```

Desweiteren sieht man zwei Nummern, da wo sonst die Größe einer Datei stehen müsste. Die erste ist die *Major Number*, die normalerweise nur einmal pro bestimmten Treiber vergeben wird (one-major-one-driver). Und die Zweite ist die *Minor Number*, die nur einmal pro Device vergeben wird.

Um einem Kernel-Modul eine Device-Number zu spendieren, muss man sich zweier Makros aus dem Header<*linux/types.h*> bedienen.

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Und um eine Device Number des Typs *dev_t* zu erhalten, bedient man sich eines Makros aus dem Header <*linux/kdev_t.h*>.

```
MKDEV(int major, int minor);
```

11 Welche Verzeichnisse sind für die Kernel-Modul-Entwicklung interessant?

- Kernel Header Files - Viele der zur Kernel-Modul-Entwicklung nötigen Header befinden sich je nach Kernel-Version in einem eigenen Verzeichnis. Daher das Shell-Kommando 'uname -r' zur Ermittlung der laufenden Kernel-Version.
  - [/lib/modules/`uname -r`/build/include/]

- Drivers (Kernel Modules) - Kernel-Module müssen binärkompatibel zu einer bestimmten Kernel-Version sein. Alle mit dem Shell-Kommando 'install' installierbaren Kernel-Module landen in solchen Verzeichnissen.
  - [/lib/modules/`uname -r`/kernel/drivers/*module-name*.ko] - das systemweit installierte Kernel-Modul

- Binary Files - Oft werden Tools zur Konfiguration von Kernel-Modulen programmiert. Man programmiert für sie in Kernel-Modulen Funktionen, die diese Tools dann im User Space dann nutzen.
  - [/usr/bin/]

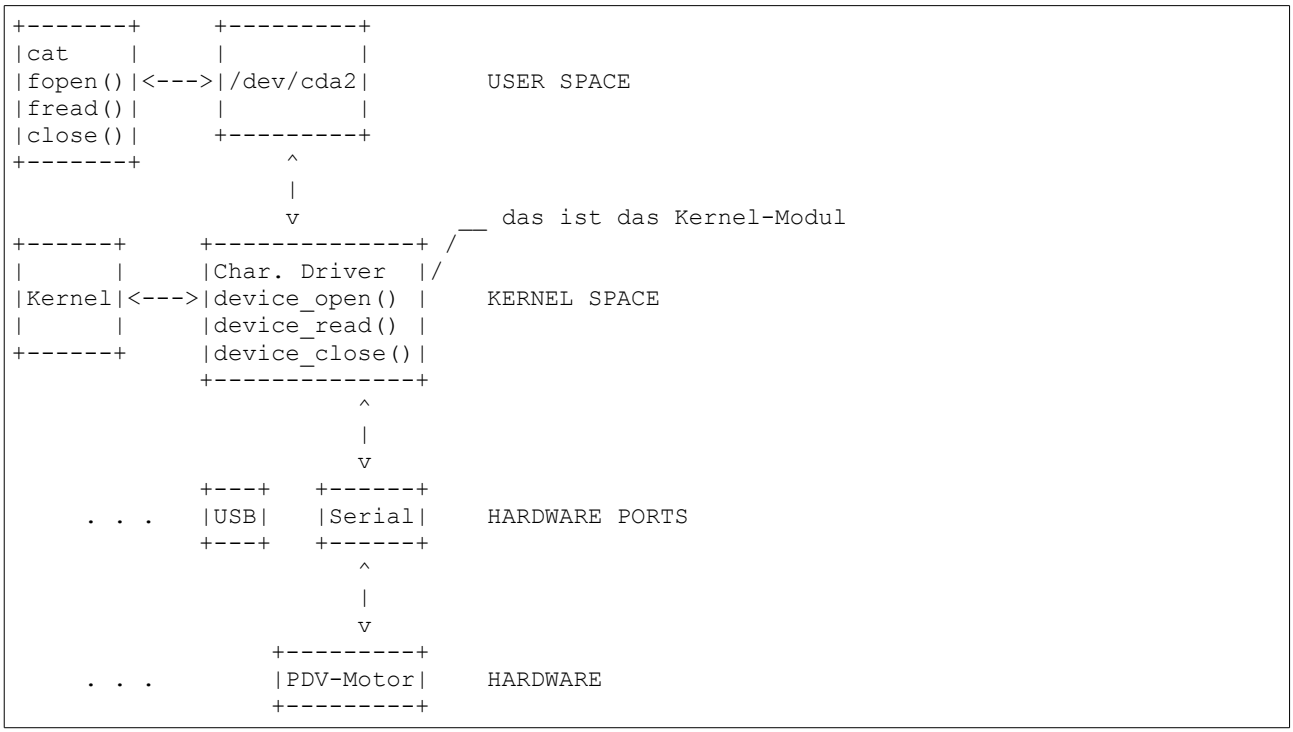
- Init Scripts – Nach einem Neustart des Systems können viele Einstellungen, auch bezüglich Kernel-Modulen, automatisch durchgeführt werden.
  - [/etc/init.d/]

- Processes - Listen laufender Prozesse von Tools und Kernel-Modulen und eingehängte Kernel-Module selbst
  - [/proc/devices] - Liste aller eingehängten Character und Block Device Driver mit Device Numbers
  - [/proc/modules] - Liste aller eingehängten Kernel-Module
  - [/proc/*module-name*] - das eigene Kernel-Modul

- Devices - Gemountete Device Files die immer zu irgend einem eingehängten Device Driver (Kernel-Modul) gehören
  - [/dev/*device-name*] - das Device File eines Character oder Block Devices

12 Was sind Character Device Driver?

In diesem Abschnitt wollen wir einen einfachen Read-Only Character Driver vorstellen für das wir so genannte Character Devices im Devices-Verzeichnis (/dev/) erzeugen wollen, um durch diese mit Programmen wie cat und echo aus dem User Space heraus in den Kernel Space greifen zu können, d.h. dort vom Character Driver nur bereitgestellte Funktionen zu nutzen.



Hinter dem Character Driver bzw. Kernel-Modul könnte sich reale Hardware befinden die wir auf diese Weise sicher nutzen könnten. Darauf wollen wir aus Gründen der Einfachheit verzichten und das Kernel-Modul einfach Nachrichten ausgeben lassen.

Schauen wir uns zu erst den einfachen Code eines selbstgeschriebenen cat-Programms an.

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

// usage: cat filename+
int main(int argc, char** argv)
{
     // wenn mehr als ein Argument übergeben
     if ( argc > 1 ) {
          int i;
          // für jedes Argument außer dem ersten
          for ( i = 1; i < argc; i++ ) {
               #define READSIZE 1024
               unsigned char buffer[READSIZE];// Lesepuffer
               int buffread = 0;      // Anzahl momentan gelesener Bytes im Puffer
               char *filename = argv[i];
               // Datei öffnen
               FILE *fp;
               if ( (fp = fopen(filename, "r")) == NULL ) {
                    fprintf(stderr,   "error: can't open %s\n", filename);
                    exit(EXIT_FAILURE);
               }
               // für jedes Zeichen der Datei filename
               while( (buffread = fread(buffer,sizeof(char),READSIZE-1,fp)) > 0 ) {
                    int k;
                    for ( k = 0; k < buffread; k++ ) {
                         printf("%c",buffer[k]);
                    }
               }
               // Datei schliessen
               close(fp);
          }
          // mit OK beenden
          return 0;
     } // sonst
     else {
          // Fehlerausgeben
          fprintf(stderr, "error: wrong arguments\n");
          // mit Fehler beenden
          return 1;
     }
}
```

Darin sehen wir die Funktionen fopen() und fread(), die beide das Öffnen und Lesen einer Datei ermöglichen.

Da unser Character Driver einfach als Datei bzw. Device File (z.B. /dev/cda2) unter Linux ansprechbar sein soll, muss es das Öffnen und das Lesen dieser Datei jeweils mit einer Funktion (z.B. device_open() und device_read()) bereitstellen.

Dazu geben wir jetzt einfach den kompletten Character Driver stückweise nacheinander an und zeigen hinterher, wie man ihn in der Linux-Shell nutzt.

12.1 Der Character Driver Quellcode

- Als erstes importiert unser Character Driver (chardev.c) einige Linux-Header und vereinbart

simple Präprozessor-Definitionen.

```
/************************************************************************
 *  chardev.c - Device Driver zum Einhängen (insmod ./chardev.ko) und Erstellen
 *  eines Read-Only Character Devices (cat /proc/devices), dass sagt wie oft
 *  ein erzeugtes (mknod /dev/DEVICE_NAME c Major Minor) Device File gelesen
 *  wurde (cat /dev/DEVICE_NAME).
 ************************************************************************/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>   /* put_user() */

#define SUCCESS 0
#define DEVICE_NAME "chardev"  /* Device-Name in /proc/devices */
#define BUF_LEN 80         /* Max. Länge der Nachricht des Divices */
```

- Wir schicken dem C-Compiler einige Prototypen für die im Kernel-Modul benutzten Funktionen voraus, die eigentlich in eine separate Header-Datei gehören. (Dieser kleiner Character Driver wäre auch ohne Header und ohne Prototypen möglich, wir wollen aber möglichst realistisch programmieren.)

```
/************************************************************************
 *  PROTOTYPEN - normalerweise in Header-Datei (chardev.h) die eingebunden wird
 ************************************************************************/
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
```

- Es werden einige globale Variablen vereinbart und vor Allem die für Character Driver wichtige Struktur file_operations. Darin sind Funktionspointer anzugeben zu den Funktionen, die wir noch schreiben werden. In diesem Kernel-Modul reichen uns vier Funktionen.

```
/************************************************************************
 *  GLOBALE VARIABLEN - normalerweise alle static damit Werte zwischen Aufrufen
 *  des Device Files (cat /dev/DEVICE_NAME) erhalten bleiben
 ************************************************************************/
static int Major;     /* Major-Nummer für den Device-Driver */
static int Device_Open = 0;    /* Ist Device schon offen? mult. Zugriff verhindern */
static char msg[BUF_LEN];  /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = { /* Ein Struct von mehreren aus linux/fs.h */
    .read = device_read,       /* Device File lesen */
    .write = device_write,     /* Device File schreiben */
    .open = device_open,       /* Device File öffnen */
    .release = device_release  /* Device File schliessen (eigentlich close) */
    /* ... weitere Funktions-Pointer */
};
```

- Wenn ein Kernel-Modul beim Kernel ein- bzw. ausgehängt wird, dann wird jeweils eine Init- bzw. eine Cleanup-Prozedur aufgerufen.

```
/************************************************************************
 *  PROZEDUREN - meist einmaliger Aufruf und ohne Parameter
 ************************************************************************/
```

- Die Init-Prozedur soll versuchen das Kernel-Modul richtig einzuhängen und im Erfolgsfall eine Nachricht ausgeben unter Welcher dynamisch vergebenen Major-Nummer der Character Driver im Linux-System erreichbar ist.

```
/* INIT - wird aufgerufen, wenn Modul eingehängt wird (insmod ./chardev.ko oder modprobe
./chardev.ko) */
int init_module(void)
{
    /* einhängen */
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if ( Major < 0 ) {
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        return Major;
    }

    printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
    printk(KERN_INFO "the driver, create a dev file with\n");
    printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
    printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
    printk(KERN_INFO "the device file.\n");
    printk(KERN_INFO "Remove the device file and module when done.\n");

    return SUCCESS;
}
```

- Die Cleanup-Prozedur soll das Kernel-Modul ohne irgendwelche Prüfungen im Linux-System einfach aushängen und vorher eine Nachricht ausgeben.

```
/*  CLEANUP - wird aufgerufen, wenn Modul ausgehängt wird (rmmod chardev) */
void cleanup_module(void)
{
    printk(KERN_INFO "My good friend major number %d is gone!\n", Major);

    /* aushängen */
    unregister_chrdev(Major, DEVICE_NAME);
    /* VERALTET:
    int ret = unregister_chrdev(Major, DEVICE_NAME);
    if (ret < 0) {
        printk(KERN_ALERT "Error in unregister_chrdev: %d\n", ret);
    } */
}
```

- Nun benötigen wir noch die vier Methoden zum Öffnen, Lesen, Schreiben und Schliessen von Device Files (/dev/cda etc.) des Device Drivers.

```
/************************************************************************
 *  METHODEN - für vorgegebene Structs aus Linux-Headern
 ************************************************************************/
```

- Wir fangen mit der Methode zum Öffnen an.

```
/* OPEN - wird aufgerufen, wenn Device File geöffnet wird (cat /dev/DEVICE_NAME) */
static int device_open(struct inode *inode, struct file *file)
{
    /* Minor Number herausfinden z.B. 0 für ganzes Gerät und 1,2... für Partitionen */
    int Minor = inode->i_rdev & 0xFF;

    /* static Counter mit erhaltendem Wert zwischen allen OPEN-Events */
    static int counter = 0;

    /* wenn Device File schon offen, dann abbrechen */
    if ( Device_Open ) return -EBUSY;
    Device_Open++;        /* Flag setzen und weiteres OPEN verbieten */

    /* Nachricht für READ vorbereiten */
    sprintf(msg, "%s%d: I already told you %d times Hello world!\n", DEVICE_NAME,
Minor, counter++);
    msg_Ptr = msg;

    try_module_get(THIS_MODULE);

    return SUCCESS;
}
```

- Wenn ein Programm wie cat ein Device File geöffnet hat, dann will es die Datei evtl. Lesen.

```
/* READ - wird aufgerufen, wenn Device File (nach Öffnen) gelesen wird */
static ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *
offset)
{
    /* Anzahl der in den Buffer geschriebenen Bytes */
    int bytes_read = 0;

    /* wenn am Ende der Nachricht, dann 0 zurück geben als Signal für EOF */
    if ( *msg_Ptr == 0 ) return 0;

    /* Nachricht in den Buffer übernehmen */
    while (length && *msg_Ptr) {
        /* Der Buffer ist im User Data Segment (Buffer des User-Programms) und
         * nicht im Kernel Data Segment, daher funktioniert die Zuweisung von
         * unserem Nachricht-Buffer zum User-Buffer nicht einfach mit "*". */
        put_user(*(msg_Ptr++), buffer++);

        length--;
        bytes_read++;
    }

    /* Anzahl gelesener Bytes zurück geben */
    return bytes_read;
}
```

- Evtl. möchte ein Programm wie echo in ein Device File schreiben. Wir legen darauf aber keinen Wert und geben eine Fehlermeldung aus, das unser Kernel-Modul das nicht unterstützt. Die Methode hätten wir deshalb auch gar nicht schreiben brauchen und sie eigentlich auch nicht der Struktur file_operations als Funktionspointer angeben brauchen.

```
/* WRITE - wird aufgerufen, wenn Device File (nach Öffnen) geschrieben wird (echo
"F.U.B.A.R." > /dev/DEVICE_NAME) */
```

```
static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t *
off)
{
    printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
    return -EINVAL;
}
```

- Die letzte der vier Methoden ist zum Schliessen eines Device Files.

```
/* RELEASE - wird aufgerufen, wenn Device File geschlossen wird (eigentlich CLOSE) */
static int device_release(struct inode *inode, struct file *file)
{
    Device_Open--;        /* Flag zurück setzen und weiteres OPEN erlauben */

    module_put(THIS_MODULE);

    return 0;
}
```

- Kernel-Module können, müssen aber nicht, auch mit Informationen versehen werden. Mehrfachnennungen sind möglich. Weitere Makros finden sich in *linux/module.h*.

```
/*****************************************************************************
 *  INFORMATIONS - in umgekehrter Reihenfolge für modinfo chardev.ko
 *****************************************************************************/

MODULE_DESCRIPTION("A simple Character Device Driver Kernel Module.");
MODULE_LICENSE("Friesenschwur v1.1 (mit Geh-Wohn-Heiz-Recht)");
MODULE_AUTHOR("Milhouse van Houten");
MODULE_AUTHOR("Krusty Krustofski");
```

12.2 Das Makefile

```
obj-m    += chardev.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD  := $(shell pwd)

help:
    @echo ""
    @echo "Makefile for a read-only Character Device Driver."
    @echo "Root privileges are not required for compilation."
    @echo ""
    @echo "Usage:"
    @echo " make all        # Compile and generate all"
#   @echo " make dbg        # Compile with -g flag"
#   @echo " make install    # Compile and install"
    @echo " make clean      # Remove all unnessesary files"
    @echo " make help       # This help screen"
#   @echo " make version    # Print version and build numbers"
    @echo ""

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
```

```
    # standard clean of Make doesn't delete everything
    # $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
    # there might be other files like *~ from editors or core dumps like core
    # write your own clean commands carefully
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions *.markers *.order
*.symvers
```

In der Shell müssen wir nur *make all* eingeben damit eine Datei namens chardev.ko erzeugt wird. Das ist das einzuhängende Kernel-Modul bzw. der Character Driver.

12.3 Das Installations-Script

Mit einem Installations-Scrip lassen sich Kernel-Module in systemweit verfügbaren Ordnern installieren. Hierzu einfach *sh ./install.sh* mit dem nachfolgenden Inhalt ausführen.

```
#!/bin/sh
install -m 644 chardev.ko /lib/modules/`uname -r`/kernel/drivers/chardev.ko
/sbin/depmod -a
```

Das Shell-Kommando *install* bewirkt das gleiche wie *mv* und *chmod*. Das heißt, es verschiebt das Kernel-Modul in ein dafür vorgesehenes Verzeichnis und setzt die Rechte. Dabei bedeutet 644 so viel wie *-rw-r--r--*.

Man sieht hier auch ein eingebettetes Shell-Kommando *uname -r*, das die Versions-Nummer des aktuell laufenden Kernels einfügt. Für einen anderen installierten Kernel einfach dessen Nummer rein schreiben. Aber dafür sollte das Kernel-Modul auch kompiliert worden sein.

Schließlich bewirkt *depmod -a*, dass das installierte Kernel-Modul und dessen Abhängigkeiten in Konfigurationsdateien geschrieben werden.

12.4 Das Deinstallations-Script

Das nachfolgende Script deinstalliert das Kernel-Modul wieder, hängt es allerdings nicht aus.

```
#!/bin/sh
rm -f /lib/modules/`uname -r`/kernel/drivers/chardev.ko
/sbin/depmod -a
```

13 Was ist bei Shell-Sessions für Kernel-Module zu tun?

13.1 Erstellen von Kernel-Modulen (mit Makefile)

Ein hier als Beispiel verwendete Makefile (§12.2) ist in einem anderen Abschnitt zu finden.

```
user@linux:~$ make

make -C /lib/modules/2.6.18-6-686/build SUBDIRS=/home/bsyll001 modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.18-6-686'
  CC [M]  /home/bsyll001/chardev.o
  Building modules, stage 2.
  MODPOST
  CC      /home/bsyll001/chardev.mod.o
  LD [M]  /home/bsyll001/chardev.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.18-6-686'

user@linux:~$
```

13.2 Anzeigen der im Kernel-Modul enthaltene Informationen

```
user@linux:~$ modinfo ./chardev.ko

filename:       ./chardev.ko
author:         Milhouse van Houten
license:        Friesenschwur v1.1 (mit Geh-Wohn-Heiz-Recht)
description:    A simple Character Device Driver Kernel Module.
srcversion:     D173B21E8FFBF8C12A40332
depends:
vermagic:       2.6.27-8-eeepc SMP mod_unload modversions PENTIUMM

user@linux:~$
```

13.3 Installieren systemweit erreichbarer Kernel-Module (mit Installations-Script und ohne einzuhängen)

Ein hier als Beispiel verwendete Installations-Script (§12.3) ist in einem anderen Abschnitt zu finden.

```
user@linux:~$ sudo su
user@linux:~# sh ./install.sh
user@linux:~# exit
user@linux:~$
```

13.4 Deinstallieren systemweit erreichbarer Kernel-Module (mit Deinstallations-Script und ohne auszuhängen)

Ein hier als Beispiel verwendete Deinstallations-Script (§12.4) ist in einem anderen Abschnitt zu finden.

```
user@linux:~$ sudo su
user@linux:~# sh ./deinstall.sh
user@linux:~# exit
user@linux:~$
```

13.5 Einhängen von systemweit erreichbar installierten Kernel-Modulen

```
user@linux:~$ sudo su
user@linux:~# modprobe chardev
user@linux:~# exit
user@linux:~$
```

13.6 Einhängen von nicht systemweit erreichbar installierten Kernel-Modulen

```
user@linux:~$ sudo su
user@linux:~# insmod ./chardev.ko
user@linux:~# exit
user@linux:~$
```

13.7 Aushängen von eingehängten Kernel-Modulen

```
user@linux:~$ sudo su
user@linux:~# rmmod chardev
user@linux:~# exit
user@linux:~$
```

13.8 Anzeigen von Log Files

13.8.1 Anzeigen der Konfigurations-Datei für Log Files

```
user@linux:~$ cat /etc/syslog.conf

#  /etc/syslog.conf     Configuration file for syslogd.
#
#                       For more information see syslog.conf(5)
#                       manpage.

#
# First some standard logfiles.  Log by facility.
#

auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslog
#cron.*                         /var/log/cron.log
daemon.*                        -/var/log/daemon.log
kern.*                          -/var/log/kern.log
lpr.*                           -/var/log/lpr.log
mail.*                          -/var/log/mail.log
```

```
user.*                          -/var/log/user.log
uucp.*                          /var/log/uucp.log

#
# Logging for the mail system.  Split it up so that
# it is easy to write scripts to parse these files.
#
mail.info                       -/var/log/mail.info
mail.warn                       -/var/log/mail.warn
mail.err                        /var/log/mail.err

# Logging for INN news system
#
news.crit                       /var/log/news/news.crit
news.err                        /var/log/news/news.err
news.notice                     -/var/log/news/news.notice

#
# Some `catch-all' logfiles.
#
*.=debug;\
        auth,authpriv.none;\
        news.none;mail.none     -/var/log/debug
*.=info;*.=notice;*.=warn;\
        auth,authpriv.none;\
        cron,daemon.none;\
        mail,news.none          -/var/log/messages

#
# Emergencies are sent to everybody logged in.
#
*.emerg                         *

#
# I like to have messages displayed on the console, but only on a virtual
# console I usually leave idle.
#
#daemon,mail.*;\
#       news.=crit;news.=err;news.=notice;\
#       *.=debug;*.=info;\
#       *.=notice;*.=warn       /dev/tty8

# The named pipe /dev/xconsole is for the `xconsole' utility.  To use it,
# you must invoke `xconsole' with the `-file' option:
#
#    $ xconsole -file /dev/xconsole [...]
#
# NOTE: adjust the list below, or you'll go crazy if you have a reasonably
#      busy site..
#
daemon.*;mail.*;\
        news.crit;news.err;news.notice;\
        *.=debug;*.=info;\
        *.=notice;*.=warn       |/dev/xconsole

user@linux:~$
```

13.8.2 Anzeigen des Log Files Verzeichnisses

```
user@linux:~$ ls /var/log/

acpid            daemon.log.3.gz  kern.log.0        scrollkeeper.log.1
acpid.1.gz       debug            kern.log.1.gz     scrollkeeper.log.2
acpid.2.gz       debug.0          kern.log.2.gz     syslog
acpid.3.gz       debug.1.gz       kern.log.3.gz     syslog.0
acpid.4.gz       debug.2.gz       lastlog           syslog.1.gz
apache2          debug.3.gz       lpr.log           syslog.2.gz
aptitude         dmesg            mail.err          syslog.3.gz
aptitude.1.gz    dmesg.0          mail.info         syslog.4.gz
auth.log         dmesg.1.gz       mail.log          syslog.5.gz
auth.log.0       dmesg.2.gz       mail.warn         syslog.6.gz
auth.log.1.gz    dmesg.3.gz       messages          user.log
auth.log.2.gz    dmesg.4.gz       messages.0        user.log.0
auth.log.3.gz    dpkg.log         messages.1.gz     user.log.1.gz
bittorrent       dpkg.log.1       messages.2.gz     user.log.2.gz
boot             dpkg.log.2.gz    messages.3.gz     user.log.3.gz
btmp             exim4            mysql             uucp.log
btmp.1           faillog          mysql.err         wtmp
cups             fontconfig.log   mysql.log         wtmp.1
daemon.log       fsck             mysql.log.1.gz    Xorg.0.log
daemon.log.0     gdm              news              Xorg.0.log.old
daemon.log.1.gz  installer        pycentral.log
daemon.log.2.gz  kern.log         scrollkeeper.log

user@linux:~$
```

13.8.3 Anzeigen des Kernel Messages Log Files (letzte zehn Nachrichten)

```
user@linux:~$ sudo su
user@linux:~# cat /var/log/kern.log | tail -n 10

Jan 12 13:39:03 debian kernel: lo: Disabled Privacy Extensions
Jan 12 13:39:03 debian kernel: IPv6 over IPv4 tunneling driver
Jan 12 13:39:13 debian kernel: eth1: no IPv6 routers present
Jan 12 14:55:29 debian kernel: chardev: module license 'unspecified' taints kernel.
Jan 12 14:55:29 debian kernel: I was assigned major number 253. To talk to
Jan 12 14:55:29 debian kernel: the driver, create a dev file with
Jan 12 14:55:29 debian kernel: 'mknod /dev/chardev c 253 0'.
Jan 12 14:55:29 debian kernel: Try various minor numbers. Try to cat and echo to
Jan 12 14:55:29 debian kernel: the device file.
Jan 12 14:55:29 debian kernel: Remove the device file and module when done.

user@linux:~# exit
user@linux:~$
```

13.8.4 Anzeigen des Debug Messages Log Files (ohne spezielles Kommando, letzte zehn Nachrichten)

```
user@linux:~$ sudo su
user@linux:~# cat /var/log/debug | tail -n 10

Jan 12 13:38:05 debian kernel: CPU: After all inits, caps: 0febfbff 00000000 00000000
```

```
00000080 00000000 00000000 00000000
Jan 12 13:38:05 debian kernel: PCI: Probing PCI hardware (bus 00)
Jan 12 13:38:05 debian kernel: Boot video device is 0000:00:0f.0
Jan 12 13:38:05 debian kernel: ACPI: PCI Interrupt Routing Table [\_SB_.PCI0._PRT]
Jan 12 13:38:05 debian kernel: PCI: Setting latency timer of device 0000:00:01.0 to 64
Jan 12 13:38:05 debian kernel: Probing IDE interface ide1...
Jan 12 13:38:05 debian kernel: sda: Mode Sense: 5d 00 00 00
Jan 12 13:38:05 debian kernel: sda: Mode Sense: 5d 00 00 00
Jan 12 13:38:17 debian kernel: vmmemctl: started kernel thread pid=2561
Jan 12 13:39:13 debian kernel: eth1: no IPv6 routers present

user@linux:~# exit
user@linux:~$
```

13.8.5 Anzeigen der Debug Messages (mit speziellem Kommando, letzte zehn Nachrichten)

```
user@linux:~$ dmesg | tail -n 10

lo: Disabled Privacy Extensions
IPv6 over IPv4 tunneling driver
eth1: no IPv6 routers present
chardev: module license 'unspecified' taints kernel.
I was assigned major number 253. To talk to
the driver, create a dev file with
'mknod /dev/chardev c 253 0'.
Try various minor numbers. Try to cat and echo to
the device file.
Remove the device file and module when done.

user@linux:~$
```

13.9 Anzeigen eingehängter Kernel-Module

13.9.1 Anzeigen eingehängter Kernel-Module (mit speziellem Kommando, erste zehn Module)

```
user@linux:~$ lsmod | head -n10

Module                  Size  Used by
chardev                 3092  0
ipv6                  226272  15
gameport               14632  1 snd_ens1371
parport                33256  3 ppdev,lp,parport_pc
mbcache                 8356  1 ext3

user@linux:~$
```

13.9.2 Anzeigen eingehängter Kernel-Module (ohne spezielles Kommando)

```
user@linux:~$ cat /proc/modules

chardev 3092 0 - Live 0xd087d000
ipv6 226272 15 - Live 0xd0b37000
gameport 14632 1 snd_ens1371, Live 0xd0a19000
parport 33256 3 ppdev,lp,parport_pc, Live 0xd09f9000
mbcache 8356 1 ext3, Live 0xd0898000

user@linux:~$
```

13.9.3 Anzeigen eingehängter von Character und Block Device Driver Kernel-Modulen (mit Device Numbers)

```
user@linux:~$ cat /proc/devices

Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
 13 input
 14 sound
 29 fb
 99 ppdev
116 alsa
128 ptm
180 usb
253 chardev

Block devices:
  1 ramdisk
  2 fd
  7 loop
  8 sd
 22 ide1
 65 sd
254 device-mapper

user@linux:~$
```

13.9.4 Anzeigen von erzeugten Device Files für Device Driver Kernel-Module

Ein *c* forne links heißt Character Device File und ein *b* heißt Block Device File.

```
user@linux:~$ ls -l /dev/

crw-rw---- 1 root video    10, 175 2009-01-12 13:37 agpgart
crw-rw---- 1 root audio    14,   4 2009-01-12 13:37 audio
```

```
crw-r--r-- 1 root root    253,   0 2009-01-12 17:02 cda
crw-r--r-- 1 root root    253,   1 2009-01-12 17:02 cda1
crw-r--r-- 1 root root    253,   2 2009-01-12 17:02 cda2
brw-r--r-- 1 root root    253,   3 2009-01-12 17:11 cda3
brw-rw---- 1 root disk      8,   0 2009-01-12 13:37 sda
brw-rw---- 1 root disk      8,   1 2009-01-12 13:37 sda1
brw-rw---- 1 root disk      8,   2 2009-01-12 13:37 sda2
brw-rw---- 1 root disk      8,   5 2009-01-12 13:37 sda5
crw-rw-rw- 1 root root      5,   0 2009-01-12 14:28 tty
crw-rw---- 1 root root      4,   0 2009-01-12 13:37 tty0
crw------- 1 root root      4,   1 2009-01-12 13:39 tty1

user@linux:~$
```

13.10 Erzeugen von Device Files für Character und Block Device Driver Kernel-Module (Kernel-Module können auch für mehrere Rollen programmiert werden)

Unterschied beim Erstellen eines Character Device Files bzw. eine Block Device Files ist nur der Buchstabe c bzw. b.

```
user@linux:~$ su
user@linux:~# mknod /dev/cda c 253 0
user@linux:~# mknod /dev/cda1 c 253 1
user@linux:~# mknod /dev/cda2 c 253 2
user@linux:~# mknod /dev/cda3 b 253 3
user@linux:~# exit
user@linux:~$
```

13.11 Entfernen von Device Files für Character und Block Device Driver Kernel-Module

```
user@linux:~$ su
user@linux:~# rm /dev/cda
user@linux:~# rm /dev/cda1
user@linux:~# rm /dev/cda2
user@linux:~# rm /dev/cda3
user@linux:~# exit
user@linux:~$
```

13.12 Testen von Device Files und deren Character und Block Device Driver Kernel-Modulen

```
user@linux:~$ cat /dev/cda

chardev0: I already told you 0 times Hello world!
```

```
user@linux:~$ cat /dev/cda1

chardev1: I already told you 1 times Hello world!

user@linux:~$ cat /dev/cda2

chardev2: I already told you 2 times Hello world!

user@linux:~$ cat /dev/cda3

cat: /dev/cda3: Kein passendes Gerät bzw. keine passende Adresse gefunden

user@linux:~$
```

13.13 Umleiten von STDOUT-Nachrichten ins Nichts

```
user@linux:~$ echo "test"                # schreibt Nachricht auf STDOUT

test

user@linux:~$ echo "test" > /dev/null    # unterdrückt STDOUT-Nachricht
user@linux:~$
```

13.14 Umleiten von STDERR-Nachrichten ins Nichts

```
user@linux:~$ cat /                              # schreibt Nachricht auf STDERR

cat: /: Ist ein Verzeichnis

user@linux:~$ cat / > /dev/null                  # unterdrückt STDERR-Nachricht nicht
(sondern nur STDOUT)

cat: /: Ist ein Verzeichnis

user@linux:~$ cat / 2> /dev/null                 # unterdrückt STDERR-Nachricht (aber
nicht STDOUT)
user@linux:~$ cat / > /dev/null 2> /dev/null     # unterdrückt STDOUT- und STDERR-
Nachricht
user@linux:~$
```

14 Referenzen

- http://oreilly.com/catalog/linuxdrive3/book/index.csp (kostenlose PDFs zu Linux Device Drivers - Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, 3. Edition)
- http://tldp.org/LDP/lkmpg/2.6/html (The Linux Kernel Module Programming Guide zu Linux Kernel v2.6.4)
- http://www.linux.it/~rubini/docs (ältere Artikel von Alessandro Rubini)