

Systemsoftware

Linux Gerätetreiber I

Prof. Dr. Michael Mächtel

Informatik, HTWG Konstanz

Version vom 07.05.17

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

Applikationsschnittstelle für den Gerätezugriff

- Interfaces für Gerät- und Dateizugriff sind identisch.
 - cat /etc/passwd
 - cat /proc/cpuinfo
 - cat /dev/console
- Pro Gerät mindestens eine Datei im Dateisystem:
 - **Gerätedatei**
- Systemfunktionen:
 - open(), close(), read(), write()
 - ioctl(), select()/poll(), fcntl(), seek()

open()/close()

- „Öffnen“ des Gerätes:
 - Ist die Datei vorhanden/ist der Treiber überhaupt geladen?
 - Stimmen die Zugriffsrechte?
 - Welcher Zugriff auf das Gerät wird verlangt?
 - Rückgabewert: Filedeskriptor
- „Aufräumen“:
 - Freigeben der bei *open* allozierten Ressourcen.
 - Nach dem *close*-Aufruf befindet sich das Gerät wieder in einem definierten Zustand.

read()

```
ssize_t read(int fd, char *buffer,  
            size_t max_bytes_to_read);
```

- Lesen von Daten (aus einer Datei oder aus der Hardware in den Speicherbereich der Applikation).
- Returnwert:
 - Anzahl der gelesenen Bytes (Minimum 1 Byte).
 - Fehlercode (-EINTR, -EAGAIN,...).

write()

```
ssize_t write(int fd,  
             char *buffer, size_t MaxBytesToWrite);
```

- Schreiben von Daten (aus der Applikation in eine Datei oder auf die Hardware).
- Write: returniert die Anzahl der geschriebenen Bytes bzw. Fehlercode:
 - Soll eine definierte Anzahl Bytes geschrieben werden, muss in einer Schleife geschrieben werden.
 - Die Fehlerbedingung „Unterbrechung durch ein Signal“ (-EINTR) ist abzufangen (Aufruf wiederholen).

ioctl()

```
int ioctl( int fd, int command, ...);
```

- Universal-Funktion zur Kommunikation mit dem Treiber.
- Unterstützt frei definierbare Kommandos mit frei definierbaren Parametern
 - Beispiel: ioctl zum „atomic“ write-read
- Kommandos und Parameter werden durch den Treiber festgelegt.

mmap()

```
void * mmap(void *start, size_t length,  
           int prot, int flags,  
           int fd, off_t offset);  
int munmap(void *start, size_t length);
```

- Funktion, um Speicherbereiche eines Geräts (Hardware) in den Adressraum einer Applikation einzublenden.
- Damit kann die Applikation performant (ohne Übergang in den Kernel-Mode) auf Hardware (-Register) zugreifen.
 - Beispiel: xorg

select()/poll()

```
int select( int n, fd_set *read_fds,  
           fd_set *write_fds, fd_set *exception_fds,  
           struct timeval *timeout );
```

- Funktion um festzustellen, ob an einem oder an mehreren Geräten (Dateien oder Netzverbindungen) Daten zum Lesen bereit liegen bzw. Daten geschrieben werden können.
 - Der Aufruf erfolgt zeitüberwacht.
 - Mit dieser Systemfunktion kann eine Applikation ohne zu pollen mehrere Kanäle (Dateien) auf Ein- oder Ausgaben überwachen.
 - Über die Makros FD_SET, FD_ZERO und FD_ISSET können die Datenstrukturen „read_fds“, „write_fds“ und „exception_fds“ initialisiert und später überprüft werden.

Beispiel select()

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set rfd;
    struct timeval tv;
    int retval;

    FD_ZERO(&rfd);
    FD_SET(0, &rfd);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfd, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval)
        printf("Data is available now.\n");
    /* FD_ISSET(0, &rfd) will be true. */
    else
        printf("No data within five seconds.\n");

    exit(0);
}
```

lseek()

- Mit dieser Funktion kann in einer Datei (bzw. bei einem Blockdevice) gezielt ein bestimmtes Byte gelesen bzw. geschrieben werden.
- Die Positionierung kann relativ
 - zum Dateianfang
 - zum Dateiende
 - zur aktuellen Position erfolgen.

Beispiel: Lesen des Bootsektors

```
int main( int argc, char **argv, char **envp )
{
    int i, j, fd;
    unsigned char buffer[512];

    if ((fd=open("/dev/hda", O_RDONLY))<0) {
        perror( "/dev/hda" );
        exit( -1 );
    }
    if (read( fd, buffer, sizeof(buffer)) != sizeof(buffer)) {
        perror( "read" );
        exit( -2 );
    }
    for (i=0; i<sizeof(buffer)/16; i++) {
        for( j=0; j<16; j++ ) {
            printf( "%2.2x ",buffer[i*16+j] );
        }
        printf("\n");
    }
    close( fd );
    return 0;
}
```

Beispiel: Ausgabe auf Drucker

```
int main( int argc, char **argv, char **envp )
{
    int fd;
    char *ptr="aaaaaa";

    if ((fd=open(„/dev/lp0“,O_WRONLY))<0) {
        perror(„/dev/lp0“);
        exit(-1);
    }
    if (write(fd,ptr,strlen(ptr))<strlen(ptr)) {
        perror(„write“);
    }
    close( fd );
    return 0;
}
```

Kontrollfluss

- Für den Zugriff auf ein Gerät gibt es zwei Zugriffsarten:
 - blocking mode
 - non blocking mode
- Beim „blocking mode“ wird eine Applikation in den Zustand „schlafend“ versetzt, falls beim read-Aufruf angeforderte Daten noch nicht zur Verfügung stehen.
- Beim „blocking mode“ wird eine Applikation in den Zustand „schlafend“ versetzt, falls bei einem write-Aufruf auszugebende Daten noch nicht geschrieben werden können.

Beispiel: Blocking read()

```
#include <stdio.h>

int main( int argc, char **argv, char **envp )
{
    int fd=0; // stdin, standardmaessig im blocking mode
    char buffer[512];

    printf("Blocking-Mode: WARTEN AUF EINGABE ...\\n");

    // blockierendes Warten
    read( fd, buffer, sizeof(buffer) );

    printf("Eingabe getaetigt...\\n");
}
```

Return beim NON-Blocking Mode

- Beim „non-blocking mode“ kommt der Aufruf (Systemcall) direkt zurück, auch wenn keine Daten gelesen bzw. geschrieben werden konnten.
- Dieser Fall ist am Fehlercode -EAGAIN erkenntlich.
- Der Zugriffsmode wird beim Zugriff auf Dateien entweder
 - beim Öffnen angegeben oder
 - nach dem Öffnen mit der Systemfunktion „fcntl“.

Beispiel: Kein Blocking beim read()

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

int main( int argc, char **argv, char **envp )
{
    int fd=0; // stdin, standardmaessig im blocking mode
    int ret;
    char buffer[512];

    fcntl( fd, F_SETFL, O_NONBLOCK);
    printf("Nonblocking-Mode: KEIN WARTEN\n");

    ret=read( fd, buffer, sizeof(buffer) );

    printf("Returnwert: %d\n", ret);
    return 0;
}
```

Zusammenfassung

- Das einheitliche Interface für den Zugriff auf Dateien oder auf Geräte entlastet den Programmierer.
- Der Datenfluss ist relativ simpel programmiert.
- Der Kontrollfluss (blocking/non blocking) ist wesentlich komplexer.
- ‘iocontrols’ bieten dem Treiberentwickler sehr viel Freiheit, sind aber fehlerträchtig und sollten vermieden werden.

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

Kernel Module

- Treiber sind (meist) Module
- Module werden dynamisch geladen:
 - Geringerer Ressourcen-Verbrauch im Kernel.
 - Einfache Treiberentwicklung (es ist kein ständiges Rebooten notwendig).
- **insmod** lädt das Modul
- **rmmod** entlädt das Modul
- **lsmod** listet die geladenen Module

Modul-Code

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

int init_module(void)
{
    printk("init_module called\n");
    return 0;
}

void cleanup_module(void)
{
    printk("cleanup_module called\n");
}

MODULE_LICENSE ("GPL");
```

Aufgaben der Funktion *init_module()*

- Modul bei Kernel-Subsysteme anmelden
 - IO-Subsystem (Treiber)
 - PCI
 - Sys-Filesystem
 - proc-Filesystem
- Initialisierung von Objekten (z.B. Waitqueues)
- Reservieren von Ressourcen

Aufgaben der Funktion *cleanup_module()*

- Freigeben der allozierten Systemressourcen
- Abmelden des Moduls beim System

Lizenzzangaben

- Module sollen angeben, welcher Lizenz sie unterliegen.
- Dazu dient das Makro „MODULE_LICENSE()“.
- Folgende Lizenzen stehen zur Auswahl:
 - „GPL“
 - „GPL and additional rights“
 - „Dual BSD/GPL“
 - „Dual MPL/GPL“
 - „Proprietary“

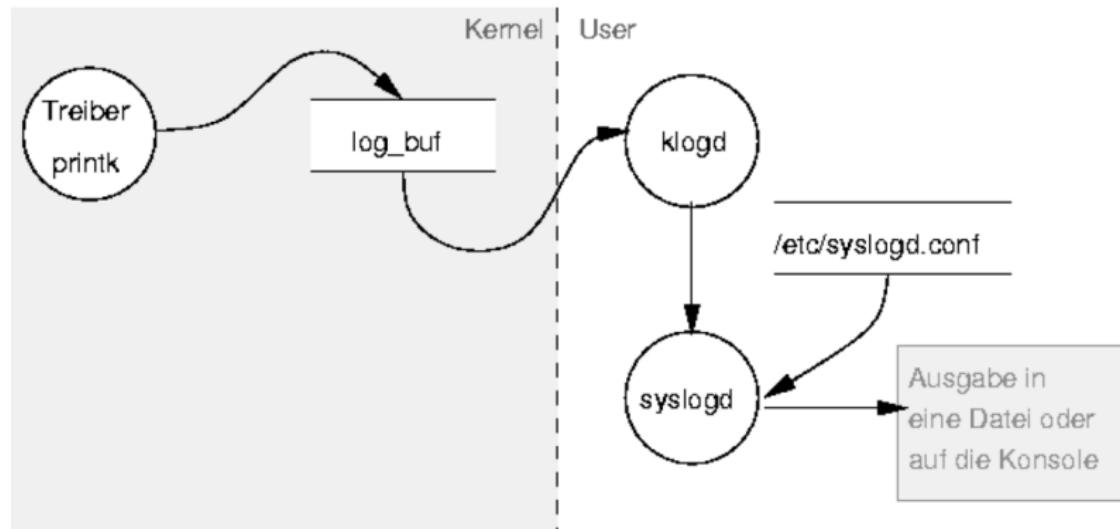
Interpretation der Lizenzangaben

- Besitzt ein Modul zwei Lizenzen, zählt die GPL.
- Das Kommando „modinfo“ zeigt die Lizenz an.
 - **modinfo -l modulename.ko**
- Fehler, die aus einem proprietären Modul herrühren, werden nicht verfolgt.
- Hersteller können sich auf Fehler ihrer Module beschränken.

Debugausgaben

- Im Kernel gibt es die zu printf() korrespondierende Funktion printk().
- Ausgaben von printk() werden im Syslog protokolliert.
- Die verschiedenen Syslog-Level (LOG_ALERT, LOG_INFO, LOG_NOTICE, LOG_DEBUG,...) werden über die drei ersten Zeichen des Formatstrings gesteuert.

Log Buffer



Log Level

Tabelle Syslog Level

Symbol	Wert	Bedeutung
KERN_EMERG	0	Das System ist nicht mehr zu gebrauchen.
KERN_ALERT	1	Sofortige Maßnahmen sind erforderlich.
KERN_CRIT	2	Der Systemzustand ist kritisch.
KERN_ERR	3	Fehlerzustände sind aufgetreten.
KERN_WARNING	4	Warnung.
KERN_NOTICE	5	Wichtige Nachricht, kein Fehler.
KERN_INFO	6	Zur Information.
KERN_DEBUG	7	Debug-Information.

Beispiel Log Ausgabe

```
printf("<7>user will %d bytes  
kopieren.\n", count );  
  
printf(KERN_DEBUG "user will %d bytes  
kopieren.\n", count );
```

pr_debug() und *pr_info()*

- Über *pr_debug()* und *pr_info()* sind übersichtliche Ausgaben (bei bedingter Compilierung) möglich:

```
pr_debug("Lesefifo=%d\n", fifoindex );
//entspricht:
#ifdef DEBUG
printf( KERN_DEBUG "Lesefifo=%d\n",fifoindex );
#endif
pr_info("Treiber erfolgreich geladen.\n");
//entspricht:
printf( KERN_INFO "Treiber erfolgreich geladen.\n");
```

Zusätzliche Debugausgaben

- Zusätzlich gibt es noch Funktionen, die einen Stacktrace ausgeben:
 - `WARN()`; -> Code wird weiterbearbeitet.
 - `WARN_on(bedingung)`; -> Code wird weiterbearbeitet.
 - `WARN_on_once(bedingung)`; -> Code wird weiterbearbeitet.
 - `BUG()`; -> Code wird abgebrochen.
 - `BUG_ON(bedingung)`; -> Code wird abgebrochen.

Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := module.o

else
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
| $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Modul Test

The screenshot shows a terminal window titled "root@ezs-mobil:/tmp/treiber - Befehlsfenster - Konsole". The terminal content is as follows:

```
root@ezs-mobil:/tmp/treiber # make ← Modul generieren
make -C /lib/modules/2.6.13-PM/build M=/tmp/treiber modules
make[1]: Entering directory '/usr/src/linux-2.6.13'
CC [M] /tmp/treiber/mod1.o
Building modules, stage 2.
MODPOST
LD [M] /tmp/treiber/mod1.ko
make[1]: Leaving directory '/usr/src/linux-2.6.13'
root@ezs-mobil:/tmp/treiber # insmod mod1.ko ← Modul laden (Extension „.ko“)
root@ezs-mobil:/tmp/treiber # lsmod | head -n5
Module           Size  Used by
mod1            1216   0
af_packet       23176   0
i915            20288   2
drm             68692   3 i915
root@ezs-mobil:/tmp/treiber # tail -f -n 5 /var/log/messages
Sep 14 20:26:16 localhost kernel: init_module called
Sep 14 20:27:31 localhost kernel: cleanup_module called
Sep 14 20:27:52 localhost kernel: init_module called ← Aktivität im Syslog verifizieren
Sep 14 20:28:10 localhost kernel: cleanup_module called
Sep 14 20:28:26 localhost kernel: init_module called

root@ezs-mobil:/tmp/treiber # rmmod mod1 ← Modul entladen
root@ezs-mobil:/tmp/treiber # █
```

Annotations in blue text with arrows point to specific parts of the terminal output:

- "Modul generieren" points to the "make" command.
- "Modul laden (Extension „.ko“)" points to the "insmod mod1.ko" command.
- ".ko=kernel object" points to the ".ko" extension in the lsmod output.
- "Aktivität im Syslog verifizieren" points to the kernel activity entries in the /var/log/messages log.
- "Modul entladen" points to the "rmmod mod1" command.

Modul Template

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

static int __init mod_init(void)
{
    printk("mod_init called\n");
    return 0;
}

static void __exit mod_exit(void)
{
    printk("mod_exit called\n");
}

module_init( mod_init );
module_exit( mod_exit );

MODULE_LICENSE("GPL");

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Michael Mächtel");
MODULE_DESCRIPTION("Modul Template");
MODULE_SUPPORTED_DEVICE("none");
```

Diese Makros abstrahieren
den Unterschied zwischen
Modul- und Built-In-Code.

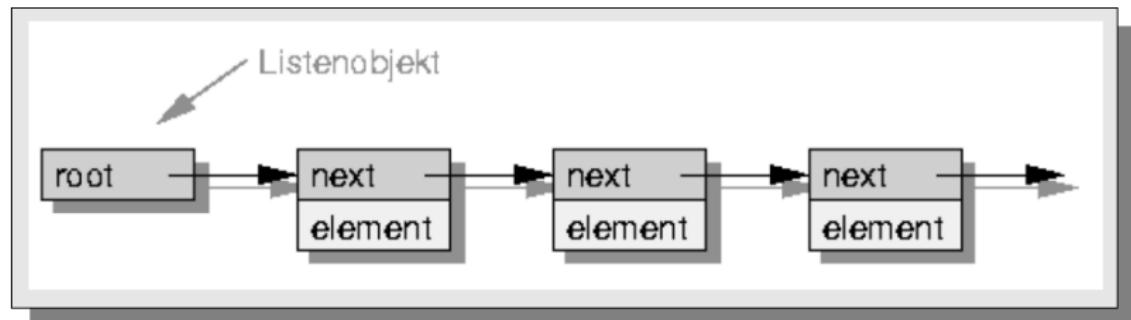


Kernel-Code Tips

- Allgemeines
 - Nutzung der Bibliotheksfunktionen?
 - Fließkomma-Operationen ?
 - Speicherschutz ?
 - Eingeschränkter Stackbereich !
- Code:
 - GoTo im Kernel
 - gcc Extensions:

```
static struct file_operations fops = {
    .open = driver_open,
    .release = driver_close,
    .read = driver_read,
    .write = driver_write,
    .poll = driver_poll,
};
```

Kernel ‘Objektorientierung’ Verkettete Liste



```
typedef struct _liste {  
    struct _liste *next;  
    void *element;  
} liste;
```

Lösung 1

```
typedef struct _liste {
    struct _liste *next;
    void *element;
} liste;

static liste *root = NULL;
// Liste mit globalem Root-Zeiger.
// Nicht objektorientiert – nicht
// mehrfach verwendbar.
static liste *liste_add( liste *new_element )
{
    liste *lptr;

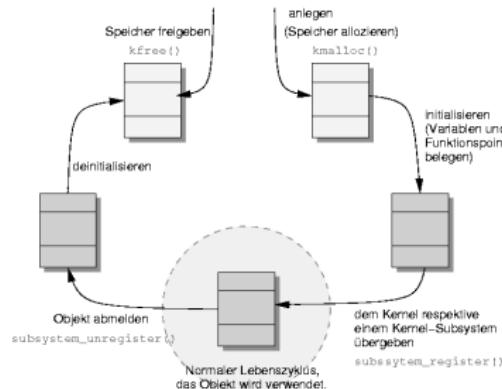
    if( root == NULL )
        return( NULL );
    for( lptr=root; lptr->next; lptr=lptr->next )
        ;
    lptr->next = calloc( 1, sizeof(liste) );
    lptr->next->element = new_element;
    return lptr->next;
}
```

Lösung 2

```
// Mehrfach verwendbare Liste
// (mit objektorientierter Datenhaltung)
// Der aufrufspezifische Parameter ist rootptr.
// Dieser speichert Informationen über die
// Aufrufe hinweg, für jede Instantiierung
// (der Liste) jedoch getrennt.
static liste *liste_add( list *rootptr,
    |list *new_element )
{
    liste *lptr;

    if( rootptr == NULL )
        return NULL;
    for( lptr=rootptr; lptr->next; lptr=lptr->next )
        ;
    lptr->next = calloc( 1, sizeof(liste) );
    lptr->next->element = new_element;
    return lptr->next;
}
```

Lifetime Kernel-Objekt



- `my_timer.data = zaehler_obj;` //Objekt (Zustands-Daten)
- `my_timer.function = zaehlen;` //Adresse der Funktion (Code)
- `add_timer(&my_timer);` //Übergabe von Code und Daten

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

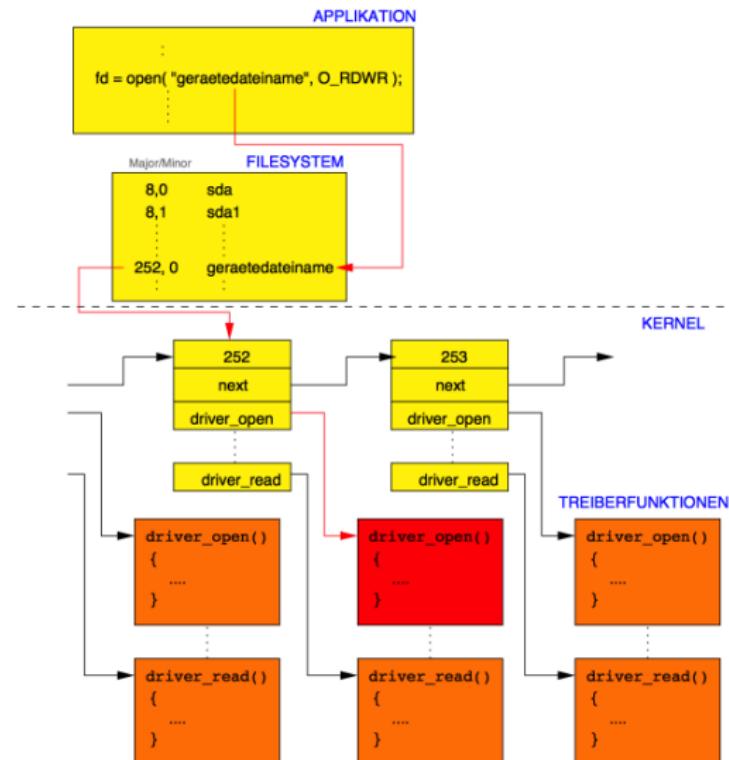
Der Treiber ...

- ist im Kernel als ein „Objekt“ repräsentiert:
 - Instantiierung der struct *file_operations*
 - Registrieren des Objekts beim Chardev-Subsystem.
 - Grundprinzip: Für jede Applikationsfunktion gibt es eine Funktion (Methode) im Kernel-Treiber-Objekt:
 - open - driver_open
 - close – driver_close
 - read – driver_read
 - write – driver_write
 - ...

Zugriff auf den Treiber durch die Applikation

- Ruft die Applikation „read()“ auf, wird im Kernel die zugehörige „driver_read()“-Funktion aufgerufen.
- driver_read() führt den Datentransfer durch:
 - Lesen der Daten von der Hardware.
 - Kopieren der gelesenen Daten in den Speicherbereich der Applikation
- PROBLEMATIK: Viele Treiber – viele Funktionen „driver_read()“
 - Die „richtige“ Funktion muss ausgewählt werden...

Übersicht: Viele driver_reads()



Treiberidentifikation

- Der Treiber meldet sich beim IO-Subsystem unter einer eindeutigen Kennung (Major-Nummer) an.
- Auf der User-Ebene gibt es eine Zuordnung zwischen dieser Major-Nummer und einem (Geräte-) Dateinamen (Attribut der Datei).
- Im Kernel sind Major- und Minornummern durch eine 32-Bit breite Gerätenummer ersetzt worden.

Majornummer

- Gerätedateien werden ‘attribuiert’ mit
 - Art der Gerätedatei (Character- oder Blockdevice)
 - Majornumber (zur Identifikation des Treibers)
 - Minornumber (zur Unterscheidung in einzelne logische Geräte)
- Kommando:
mknod MyDeviceName c Major Minor
- Für die Major-Nummer stehen nur 8 Bit zur Verfügung -> also gibt es maximal 256 unterschiedliche Geräte (zu wenig).
- Abhilfe:
 - Gerätenummern (Kategorien)
 - dynamische Vergabe der Major-Nummer und dynamische Generierung eines Gerätedateieintrags.

Minornummer

- Kodierung zusätzlicher Informationen:
 - Funktionalitäten (z.B. die Art des Handshaking bei der seriellen Schnittstelle)
 - Zugriffsbereiche (z.B. die einzelnen Partitionen einer Festplatte)
 - Eine spezifische Hardware (z.B. ob die 1. oder die 2. serielle Schnittstelle verwendet werden soll) – Ein Treiber für mehrere (gleiche respektive ähnliche) Geräte.
- Logische Sicht auf reale oder virtuelle Geräte.
- Belegung/Bedeutung bei Character-Devices frei.
- Belegung/Bedeutung bei Block-Devices vorgegeben.

Vom Modul zum Treiber ...

```
...
static int major;
static struct file_operations fops;
static int __init mod_init(void)
{
    if((major=register_chrdev(0,"TestDriver",&fops))==0) {
        return -EIO;
    }
    return 0;
}
...
static void __exit mod_exit(void)
{
    unregister_chrdev( major,"TestDriver" );
}

module_init( mod_init );
module_exit( mod_exit );
```

Treiber Objekt

Majornumber

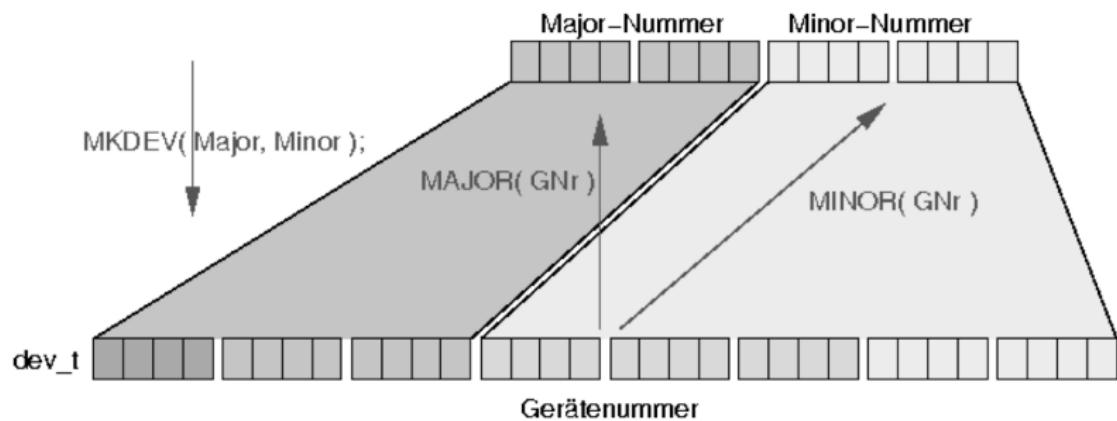
Anmelden des Treibers beim Chardev-Subsystem (VFS) des Kernels.

Abmelden des Treibers beim Chardev-Subsystem (VFS) des Kernels.

Gerätenummern

- Major- und Minor-Nummern sind 8 Bit breit
 - damit sind theoretisch 2^{16} (65536) Geräte möglich
- Gerätenummern sind 32 Bit breit
 - Über 4 Milliarden Geräte ansprechbar
- Bei Gerätenummern gibt es keine Unterscheidung zwischen Major und Minor Nummer
- Ein Treiber kann selbst festlegen, wie viele Geräte er maximal bedienen möchte.
- Damit beide Systeme nebeneinander existieren können wurde ein Mapping eingeführt

Mapping auf Gerätenummern

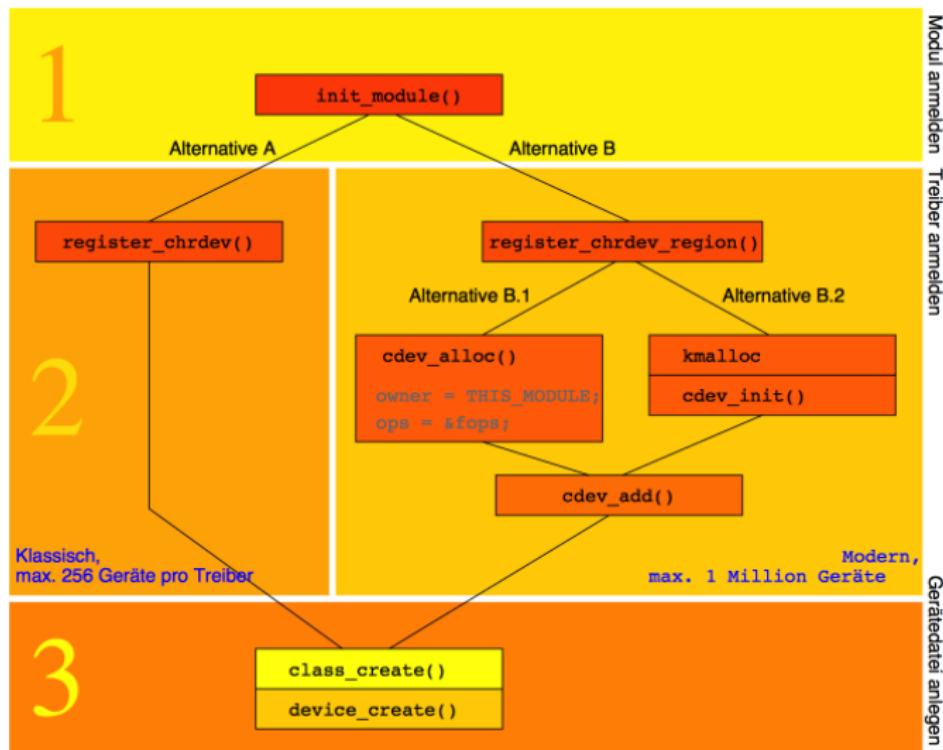


Zugriff auf die Minor-Nummern per Makro:

- per Makro:
 - *MINOR()*: über struct device
 - *iminor()*: über „struct inode“
- „struct inode“ wird *driver_open()* und *driver_close()* als Parameter übergeben.
 - *minor=iminor(geraetedatei);*
- *driver_read()*, *driver_write()* bekommen nur „struct file“ übergeben:
 - *struct file* enthält einen Verweis auf *struct inode*
 - *minor=iminor(instance->f_dentry->d_inode);*

- Gerätenummern-Bereich reservieren.
 - `alloc_chrdev_region()`
 - `register_chrdev_region()`
- Treiber beim Kernel anmelden.
 - `cdev_alloc()` – Objekt reservieren
 - `cdev_add()` – Anmeldung (Objekt instanzieren)
- sysfs-Verzeichnis erstellen, damit automatisiert durch **udev** Gerätedateien angelegt werden.

Übersicht: The Modern Art



Modern: mod_init()

```
...  
if( alloc_chrdev_region(&template_dev_number, 0, 1, TEMPLATE) < 0 )  
    return -EIO;  
driver_object = [cdev_alloc(); /* Anmeldeobjekt reservieren */  
if( driver_object==NULL )  
    goto free_device_number;  
driver_object->owner = THIS_MODULE;  
driver_object->ops = &fops; Treiber anmelden  
if( cdev_add(driver_object, template_dev_number, 1) )  
    goto free_cdev;  
/* Eintrag im Sysfs, damit Udev den Geraetedateieintrag erzeugt. */  
template_class = class_create( THIS_MODULE, MY_CLASS_NAME );  
device_create( template_class, NULL, template_dev_number,  
    NULL, "%s", MY_DEVICE_FILE_NAME );  
return 0;  
free_cdev:  
    kobject_put( &driver_object->kobj ); Aufräumen  
free_device_number:  
    unregister_chrdev_region( template_dev_number, 1 );  
    return -EIO;  
}
```

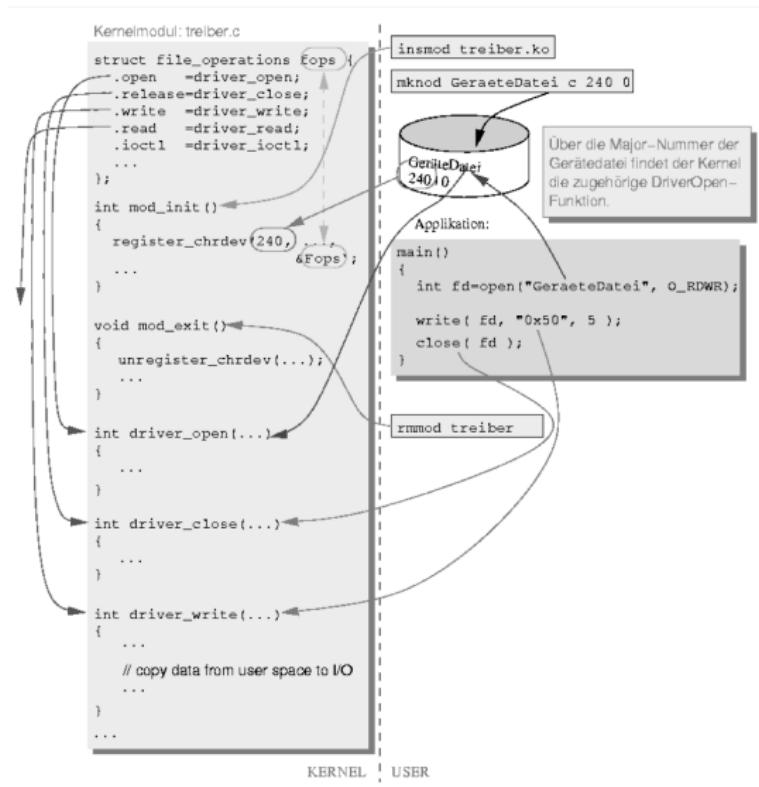
Modern: mod_exit()

```
static void __exit template_exit( void )
{
    device_destroy( template_class, template_dev_number );
    class_destroy( template_class );
    /* Abmelden des Treibers */
    cdev_del( driver_object );                                Treiber abmelden
    unregister_chrdev_region( template_dev_number, 1 );       Gerätenummer freigeben
    return;
}
```

Gerätedateien automatisch anlegen

- Die Gerätedatei kann in einem Standard Linux System über **udev** automatisiert angelegt werden.
 - Diese Technik erleichtert den Umgang mit vom Kernel vergebenen Majornummern.
- Dazu muss sich der Treiber bei einer Gerätekasse über das Gerätemodell (SYS-Filesystem) anmelden.
 - Das Gerätemodell erwartet eine Gerätenummer.
 - Beim Abmelden vom Gerätemodell wird die Gerätedatei wieder entfernt.

Übersicht der BASIS Funktionen



Treiber open() Funktion

- Zugriffsüberwachung
- Initialisierung
- Rückgabewert:
 - 0: Zugriff gestattet
 - EIO, -EBUSY, ...: Zugriff verweigert

Weitere Aufgaben der Treiber open() Funktion

- Überprüfen von dedizierten Zugriffsrechten (z.B. Sicherstellen, dass zu einem Zeitpunkt nicht mehr als ein Prozess bzw. eine Treiberinstanz auf den Treiber zugreifen).
 - Parameter:
 - struct inode und
 - struct file
- Allokation und Initialisierung von (Hardware-) Ressourcen.
- Funktion gibt 0 zurück, wenn das Gerät geöffnet werden darf, ansonsten einen Fehlercode (<asm/errno.h>).

Beispiele für Treiber open() Funktionen

```
static int driver_open( struct inode *geraedetei,
    struct file *instanz )
{
    return 0;
}

static int write_count=0;
...
static int driver_open( struct inode *geraedetei, struct file *instanz )
{
    if( instanz->f_flags&0_RDWR || instanz->f_flags&0_WRONLY ) {
        if( write_count > 0 ) {
            return -EBUSY;
        }
        write_count++;
    }
    return 0;
}
```

Treiber close() Funktion

- Aufgabe: Ressourcenfreigabe
- Überführen der Hardware in den sicheren Zustand.
- Rückgabewert: 0
- Beispiel:

```
static int driver_close( struct inode *geraetedatei,
    struct file *instanz )
{
    if( instanz->f_flags&O_RDWR
        || instanz->f_flags&O_WRONLY ) {
        write_count--;
    }
    return 0;
}
```

Treiber read() Funktion

- Datentransfer zwischen Kernel- und User-Space.
- Zugriff auf (eventuell vorhandene) Hardware.
- Rückgabewert:
 - Anzahl der transferierten Bytes.
 - Alternativ: Fehlercode.
- Funktion zum Kopieren von Daten:
 - `copy_to_user(char *to, char *from, int count)`

Code Beispiel Treiber read() Funktion

```
static ssize_t driver_read( struct file *instanz,
    char *user, size_t count, loff_t *offset )
{
    int not_copied, to_copy;

    to_copy = strlen(hello_world)+1;
    to_copy = min( to_copy, count );
    not_copied=copy_to_user(user,hello_world,to_copy);
    return to_copy-not_copied;
}
```

Datentransfer vom
Kernel zum User

Datentransfer zwischen User- und Kernel-Space

- Zugriff auf den User-Space ist vom Kernel aus nicht direkt möglich.
- Funktionen zum Datentransfer:
 - `copy_to_user(to, from, len)`
 - `copy_from_user(to, from, len)`
 - `get_user(variable, source);`
 - `put_user(variable, source);`
- Zusätzlich gibt es Funktionen für den Zugriff ohne Zugriffsschutz... (`__copy_to_user()`...)

Parameter der Treiber read() und write() Funktion

- Beschreibung der Parameter:
 - struct file *instanz: Info über die Treiberinstanz
 - char *user: Speicherbereich im User-Space
 - size_t count: Größe des Buffers im User-Space
 - loff_t offs: Offset
- Returnwert:
 - Anzahl der kopierten Bytes oder
 - Fehlercode (negativer Wert)

Treiber write() Funktion

- Datentransfer zwischen User- und Kernel-Space.
- Zugriff auf (eventuell vorhandene) Hardware.
- Rückgabewert:
 - Anzahl der transferierten Bytes.
 - Alternativ: Fehlercode.
- Funktion zum Kopieren von Daten:
 - `copy_from_user(char *to, char *from, int count)`

Beispiele für Treiber write() Funktionen

```
static ssize_t driver_write( struct file *instanz,
    char __user *buffer, size_t count, loff_t *offset )
{
    return count;
}

static ssize_t driver_write( struct file *instanz, const char *user,
    size count, loff_t *offs )
{
    int to_copy;
    int not_copied;

    printk("DriverWrite called\n");
    to_copy = min( count, sizeof(fifo_buf) );
    not_copied = copy_from_user( fifo_buf, user, ToCopy );
    printk("%s", fifo_buf );
    return to_copy-not_copied;
}

static struct file_operations Fops = {
    .owner = THIS_MODULE,
    .write = driver_write,
};
```

Zusammenfassung: Treiber read() /write() Funktion

- Die Systemcalls read() und write() triggern im Treiber die Funktionen driver_read() und driver_write().
- Die Funktionen geben die Anzahl der transferierten Bytes zurück.
- Zum Datentransfer werden die Funktionen copy_to_user() und copy_from_user() eingesetzt.
- Logische Geräte werden innerhalb der Lese- oder Schreibfunktion über die Minornummer identifiziert.

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

Zugriffsmodi im Treiber

- Blockierend:
 - Wenn angeforderte Daten nicht vorhanden sind, muss die Applikation warten, bis Daten vorliegen.
- Nicht-blockierend:
 - Wenn angeforderte Daten nicht vorhanden sind, wird die Anwendung darüber informiert (Fehlercode -EAGAIN)

Zugriffsmodi werden im Treiber (read/write) realisiert

- Non-blocking:

```
//no data available  
if( TreiberInstanz->f_flags & O_NONBLOCK ) //non  
blocking mode  
    return -EAGAIN; //return
```

- Blocking:

```
//no data available  
if( !(TreiberInstanz->f_flags & O_NONBLOCK) )  
//blocking mode  
    wait_event_interruptible( &wq_read,  
    condition ); //sleep  
//data available  
...
```

Taskzustand beeinflussen

- Funktionen:
 - `wait_event()`
 - `wait_event_interruptible()`
 - `wait_event_killable()`
 - `wait_event_interruptible_timeout()`
 - `wait_event_killable_timeout()`
 - `wait_event_for_completion()`
- Interruptible-Varianten sind durch Signals unterbrechbar.
- Killable-Varianten sind nur durch Signals unterbrechbar, die zum Ende des zugehörigen Jobs führen.
 - `SIGKILL (9)`
 - Alle Signals, die nicht explizit abgefangen werden.
- Weitere Funktionen um den Taskzustand zu beeinflussen:
 - `wake_up()`
 - `wake_up_interruptible()`

Übersicht Modi bei *driver_read()*

```
#define READ_POSSIBLE (atomic_read(&bytes_available)!=0)
...
ssize_t driver_read( struct file *instance, char __user *buffer,
                     size_t max_bytes_to_read, loff_t *offset)
{
    size_t to_copy, not_copied;
    char kernelmem[128]; /* anpassen */

    ...
    if( !READ_POSSIBLE && (instance->f_flags&O_NONBLOCK) ) {
        return -EAGAIN;
    }
    if( wait_event_interruptible( wq_read, READ_POSSIBLE ) ) {
        return -ERESTARTSYS;
    }
    to_copy = min((size_t)atomic_read(&bytes_available),
                  max_bytes_to_read);
    not_copied = copy_to_user( buffer, kernelmem, to_copy );
    atomic_sub( to_copy-not_copied, &bytes_available );
    return to_copy-not_copied;
}
```

NON_Blocking Mode

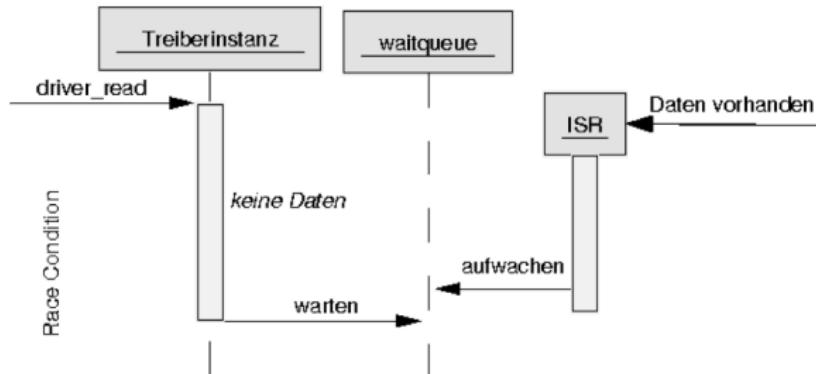
Blocking Mode

Zugriffsmodi bei `wait_event_interruptible()`

- Rechenprozess in den Zustand „schlafen“ (**TASK_INTERRUPTIBLE**) überführen:
 - Objekt vom Typ „`wait_queue_head_t`“ instanzieren.
 - Auf einer `wait_queue` können beliebig viele Rechenprozesse schlafen.
 - `wait_event_interruptible(my_wait_queue, Wartebedingung);`
- Aufwecken
 - `wake_up_interruptible(&my_wait_queue);`

wait_event_interruptible() ist ein Makro

- *wait_event_interruptible()* ist ein Makro:
 - Es wird direkt mit der *wait_queue* parametriert, nicht mit dessen Adresse.
 - Das Makro ist erforderlich, um eine Race-Condition zu verhindern.
 - Race-Condition: Das *wake_up()* (durch die ISR) kommt vor dem Schlafen-legen.



Einfaches Schlafen

- Wer einen Prozess wirklich nur schlafen legen will, kann sich der folgenden Befehlssequenz bedienen:

```
set_current_state(TASK_INTERRUPTIBLE);  
schedule();
```

- Auf diese Art wird der Prozess in den unterbrechbaren Wartezustand versetzt.
- Für ein nicht unterbrechbares Warten verfährt man analog:

*set_current_state(TASK_NONINTERRUPTIBLE);
schedule();*

Codefragment Zugriffsmodus

```
static wait_queue_head_t wait_queue_for_read, wait_queue_for_write;
...
    init_waitqueue_head( &wait_queue_for_read );
    init_waitqueue_head( &wait_queue_for_write );
...
// keine Daten vorhanden
if( instanz->f_flags & O_BLOCKING ) { // blocking mode
    retval = wait_event_interruptible( wait_queue_for_read,
        data_available );
    // nach dem Aufwecken arbeitet der Prozess hier weiter
    ...
int InterruptServiceRoutine( ... )
{
...
// jetzt sind Daten vorhanden
data_available = 1;
wake_up_interruptible( &wait_queue_for_read );
...
}
```

Objekt instanzieren

Objekt initialisieren

Warten, solange die Wartebedingung nicht erfüllt ist.

Wartebedingung ist erfüllt

Aufwecken nach Schlafen

- `wait_event()` schläft genau so lange, bis die Wartebedingung erfüllt ist.
- `wait_event_interruptible()` schläft genau so lange, bis entweder die Wartebedingung erfüllt ist oder aber ein Signal den Prozess aufweckt.
 - Die Variante `wait_event_interruptible_timeout()` berücksichtigt zusätzlich noch ein Timeout.
 - Das Auftreten eines Signals während des Schlafens muss im Treiber berücksichtigt werden!

Signale

- Jedem Rechenprozess sind 64 Signals zugeordnet.
 - 32 Signals werden jedoch nur genutzt.
- Jedes Signal ist durch ein Bit repräsentiert.
 - Die Zuordnung von Bit im Bitfeld ist in `<asm/signal.h>` definiert.
- Der Returnwert der Funktion `wait_event_interruptible()` gibt an, ob das Schlafen aufgrund eines Signals beendet wurde:
 - **ERESTARTSYS**

Schlafen in read()/write() durch Signal beendet

- In den Funktionen `driver_read()` und `driver_write()` wird das Schlafen durch ein Signal beendet:
 - Es werden typischerweise keine Daten in den User-Space kopiert.
 - Die Funktionen geben ihrerseits den Wert `ERESTARTSYS` zurück:

```
...
    /* keine Daten vorhanden */
    if( instanz->f_flags & O_BLOCKING ) { /* blocking mode */
        retval = wait_event_interruptible( wait_queue_for_read,
            data_available );
        if( retval == -ERESTARTSYS )
            return -ERESTARTSYS;
    }
    /* Daten kopieren */
```

Codefragment *driver_read()*

```
#define READ_POSSIBLE (atomic_read(&bytes_available)!=0)
...
ssize_t driver_read( struct file *instance, char __user *buffer,
                     size_t max_bytes_to_read, loff_t *offset)
{
    size_t to_copy, not_copied;
    char kernelmem[128]; /* anpassen */

    ...
    if( !READ_POSSIBLE && (instance->f_flags&O_NONBLOCK) ) {
        return -EAGAIN;
    }
    if( wait_event_interruptible( wq_read, READ_POSSIBLE ) ) {
        return -ERESTARTSYS;
    }
    to_copy = min((size_t)atomic_read(&bytes_available),
                  max_bytes_to_read);
    not_copied = copy_to_user( buffer, kernelmem, to_copy );
    atomic_sub( to_copy-not_copied, &bytes_available );
    return to_copy-not_copied;
}
```

NON_Blocking Mode

Blocking Mode

Codefragment *driver_write()*

```
#define WRITE_POSSIBLE (atomic_read(&bytes_that_can_be_written)!=0)
...
ssize_t driver_write( struct file *instance, const char __user *buffer,
|           size_t max_bytes_to_write, loff_t *offset)
{
    size_t to_copy, not_copied;
    char kernelmem[128]; /* Groesse anpassen */

    if( !WRITE_POSSIBLE && (instance->f_flags&O_NONBLOCK) ) {
        |   return -EAGAIN;
    }
    if( wait_event_interruptible(wq_write,WRITE_POSSIBLE) ) {
        |   return -ERESTARTSYS;
    }
    to_copy = min((size_t) atomic_read(&bytes_that_can_be_written),
        |           max_bytes_to_write );
    not_copied = copy_from_user( kernelmem, buffer, to_copy );
    write_data_to_hardware();
    atomic_sub( to_copy-not_copied, &bytes_that_can_be_written );
    return to_copy-not_copied;
}
```

The diagram consists of two blue arrows. One arrow points from the text "NON_Blocking Mode" to the line "if(!WRITE_POSSIBLE && (instance->f_flags&O_NONBLOCK)) {". The other arrow points from the text "Blocking Mode" to the line "if(wait_event_interruptible(wq_write,WRITE_POSSIBLE)) {".

Zusammenfassung Zugriffsmodi

- Für die Realisierung eines Treibers wird benötigt:
 - struct file_operations
 - Methoden (driver_open, driver_close, driver_read, driver_write)
- Zugriffsmodi werden im Treiber realisiert.
- Dazu kann der Treiber die Applikation schlafen legen, falls Daten nicht vorhanden sind.

Übersicht

1 Geräte API

2 Basis Modul

3 Datentransfer

4 Zugriffsmodi

5 Treiberinstanzen

Treiberinstanzen

- Mit jedem Aufruf von `open()` legt der Kernel eine Struktur vom Typ `struct file` an. Diese `struct file` repräsentiert eine Treiberinstanz.
 - Treiber müssen sehr häufig instanzenspezifische Daten abspeichern.
 - Deklarieren Sie für instanzenspezifische Parameter eine Datenstruktur.
- Reservieren Sie beim `driver_open()` Speicher für ein solches Objekt.
 - Speichern Sie die Adresse des Objektes in dem übergebenen Objekt vom Typ `struct file` ab.
 - Bei jeder weiteren Treiberfunktion kann auf dieses Objekt zugegriffen werden.
- Beim `driver_close()` muss der Speicher wieder freigegeben werden.

Beispiel für instanzspezifische Daten

```
... int driver_open( struct inode devfile, struct file instance ) {  
    instance->private_data =  
        kmalloc( sizeof(struct inst_data) ); ...
```

Zusammenfassung Treiberinstanz

- Viele Datenstrukturen des Kernels bieten Hooks an, um dort eigene (modul- oder treiberspezifische) Datenstrukturen anzukoppeln.
- Eine Treiberinstanz ist im Kernel durch die Struktur *struct file* repräsentiert.
- Diese Struktur hält mit dem Element „private_data“ ein Element bereit, um dort instanzenspezifische Parameter anzukoppeln.