Ablauf

Inhalt

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
    - ...

Treiber-Identifikation

- 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.

Majornumber

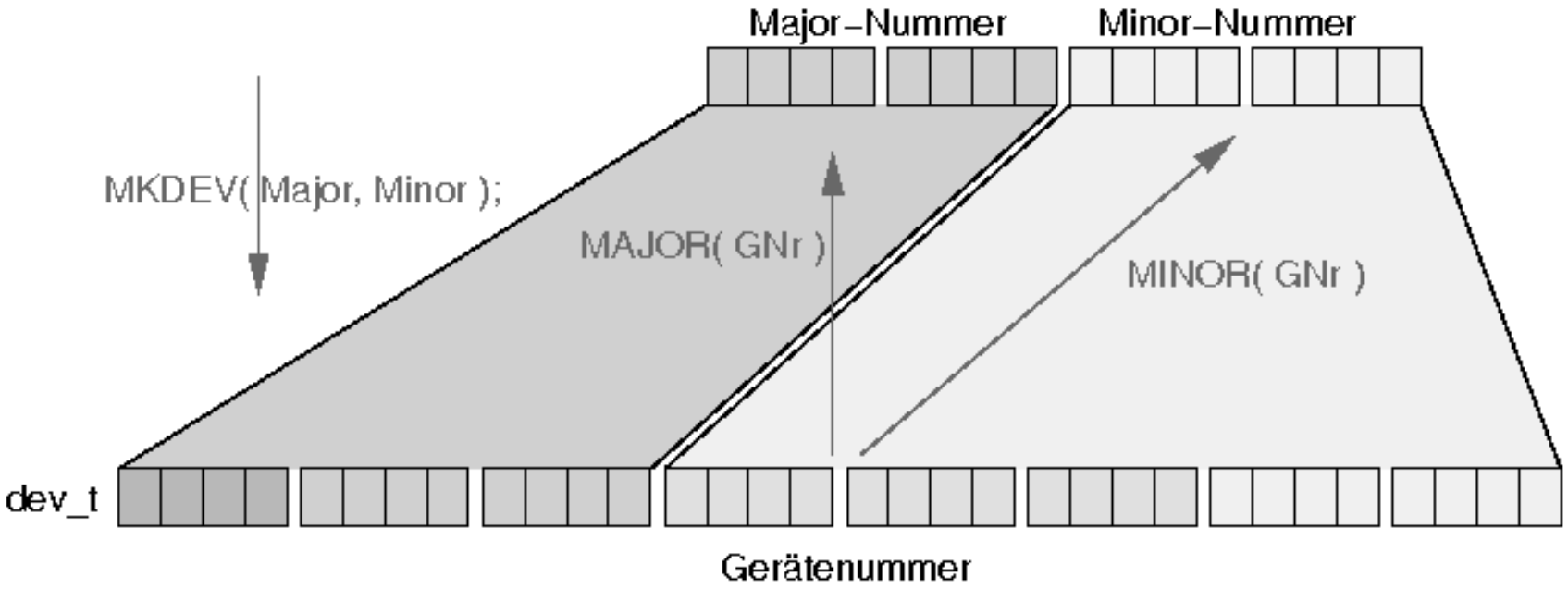
- Gerätedateien werden „attributiert“ mit
  - Art der Gerätedatei (Character- oder Blockdevice)
  - Majornumber (zur Identifikation des Treibers)
  - Minornumber (zur Unterscheidung in einzelne logische Geräte)
- Kommando:

```
mknod MyDeviceName c Major Minor
```

Majornumber

- Für die Major-Nummer stehen nur 8 Bit zur Verfügung --> also gibt es maximal 256 unterschiedliche Geräte (zu wenig).
- Abhilfe:
  - Gerätenummern (bis dato nicht wirklich im Einsatz).
  - dynamische Vergabe der Major-Nummer und dynamische Generierung eines Gerätedateieintrags.

Abbildung auf Gerätenummern



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.

Minornummern

- Zugriff auf die Minor-Nummern 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 );`

Vom Modul zum Treiber ...

```
...

static struct file_operations fops;
static int __init mod_init(void)
{
    if( register_chrdev(240,"TestDriver",&fops)==0 ) {
        return 0;
    }
    return -EIO;
}
...
static void __exit mod_exit(void)
{
    unregister_chrdev( 240,"TestDriver" );
}

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

Treiber-Objekt

Majornumber

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

Abmelden des Treibers

Gerätedatei automatisiert anlegen

- Die Gerätedatei kann über Udev automatisiert angelegt werden.
  - Diese Technik erleichtert den Umgang mit vom Kernel vergebenen Majornummern.
- Dazu muss sich der Treiber bei einer Geräteklasse über das Gerätemodell (Sys-Filesystem) anmelden.
  - Das Gerätemodell erwartet eine Gerätenummer.
  - `MKDEV(major,minor)` erzeugt die Gerätenummer.
- Beim Abmelden vom Gerätemodell wird die Gerätedatei wieder entfernt.

Codebeispiel – Gerätedatei anlegen

```
...
static int __init driver_init(void)
{
    if( (major=register_chrdev(0,"book",&fops)) ) {
        device_create(&input_class,NULL,
            MKDEV(major,minor),
            NULL,"book%d",minor);
        return 0;
    }
    return -EIO;
}

static void __exit driver_exit(void)
{
    ...
    device_destroy(&input_class,
            MKDEV(major,minor));
    ...
}
```

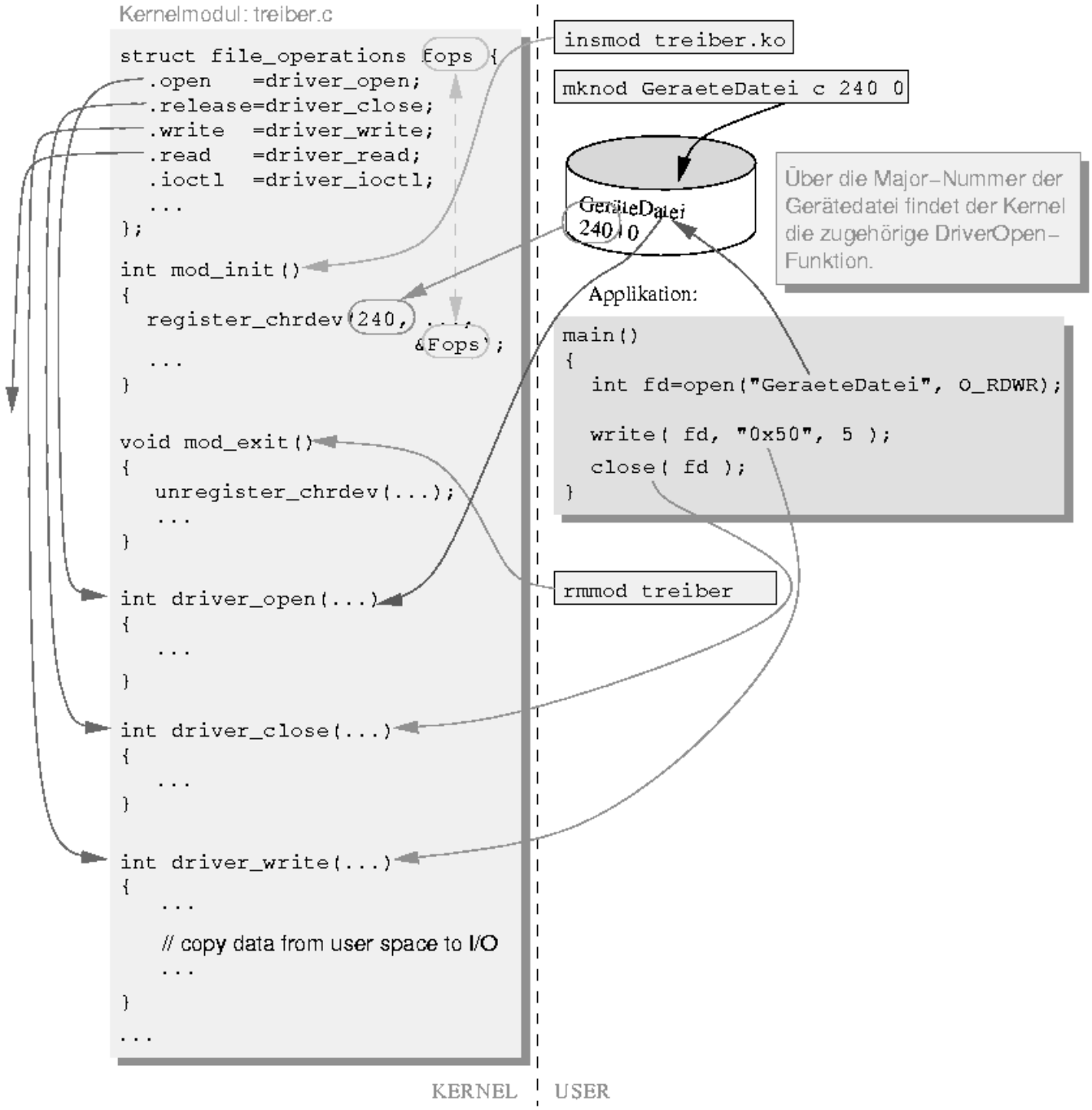
Gerät (struct device *)

Parentdirectory

Übung 1.4: Anmelden

- Kodieren Sie das Anmelden des Moduls beim Chardev-Subsystem (`register_chrdev()`). Bauen Sie geeignete „`printk()`“-Statements ein, die darüber Aufschluss geben, ob das Anmelden erfolgreich abgelaufen ist oder nicht. Wählen Sie als Major-Nummer die 240 aus.
- Vergessen Sie nicht, auch das Abmelden zu implementieren!
- Generieren und testen Sie das Modul.
- Führen Sie dazu auch das Kommando
  - `cat /proc/devices` aus.

Basis Funktionen



Open-Funktion im Treiber

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

Open-Funktion im Treiber

- 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>`).

driver_open

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

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

driver_close

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

```
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;
}
```

Übung 1.5: driver_open/driver_close

- Erstellen Sie auf Basis des Templates ein Modul namens openclose.c und passen Sie das Makefile entsprechend an.
- Kodieren Sie die Funktionen `driver_open()` und `driver_close()`, mit jeweils einem `printk()`.
- Generieren Sie den Treiber und legen Sie eine zugehörige Gerätedatei (`mknod`) mit dem Namen `geraetedatei` an.
- Überwachen Sie die Ausgaben des Treibers mit Hilfe von `tail -f /var/log/messages`
- Testen Sie den Treiber durch Aufruf des Kommandos:
  - `cat geraetedatei`.

Übung 1.5: driver_open/driver_close

- Modifizieren Sie die Funktion `driver_open()`, so dass immer nur ein Rechenprozess auf den Treiber zugreifen kann. Verwenden Sie dazu eine globale Variable, die die Anzahl der zur Zeit zugreifenden Instanzen mitzählt.
- Generieren Sie Ihren Treiber und testen Sie ihn mit dem vorgestellten `cat`-Befehl.

HINWEIS: Der Zugriff auf die globale Variable stellt einen kritischen Abschnitt dar! Mehrere Rechenprozesse können gleichzeitig die Variable überprüfen und/oder verändern. Linux bietet zum Schutz der Variable den Datentyp „atomic_t“ an.
Die hier notwendigen Funktionen lauten:

- `int atomic_inc_and_test( atomic_t *v );`
- `void atomic_dec( atomic_t*v);`

Übung 1.5: driver_open/driver_close

- Schreiben Sie eine eigene Applikation, mit der Sie auf den Treiber zugreifen. Dazu müssen Sie eine `open()`- und eine `close()`-Funktion implementieren. Nennen Sie die Quellcode-Datei `access.c`.
- Generieren Sie die Applikation (`make access`) und testen Sie erneut den Treiber.

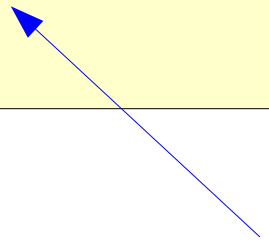
driver_read

- 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 )`

driver_read

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

    to_copy = min( strlen(hello_world)+1, 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 );`

Treiber-Write / Treiber-Read

- 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)

Übung 1.7: driver_read

- Ergänzen Sie Ihren Treiber um eine `driver_read()`-Funktion, die das virtuelle Gerät „`/dev/zero`“ implementiert. Dieses Gerät soll bei Leseanfragen jeweils eine 0 zurückgeben.
- Testen Sie den Treiber mit Hilfe des `cat`- und des `hexdump`-Kommandos:
  - `cat geraetedatei | hexdump`

Übung 1.7: driver_read

- Ergänzen Sie die Funktion `driver_read()` derart, dass sie nur dann eine '0' zurückgibt, wenn über die Minornummer 1 zugegriffen wird.
- Ergänzen Sie die Funktion `driver_read()` derart, dass beim Zugriff über die Minornummer 0 der String “hello world“ zurückgegeben wird.
- Testen Sie den Treiber mit Hilfe des `cat`-Kommandos.
- Begründen Sie, warum `cat` den String wiederholt ausgibt.
- Erweitern Sie Ihre Applikation, um den String „`hello world`“ vom Treiber zu lesen.

driver_write

- 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 )`

Übung 1.8: driver_write

- Ergänzen Sie Ihren Treiber um eine `driver_write()`-Funktion, die das virtuelle Gerät „`/dev/null`“ realisiert. Alle Daten, die auf das Gerät geschrieben werden, werden „gefressen“.
- Testen Sie den Treiber mit Hilfe des `cat`-Kommandos.

Codebeispiel: driver_write

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

Codebeispiel: driver_write

```
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,
};
```

Austesten des Treibers

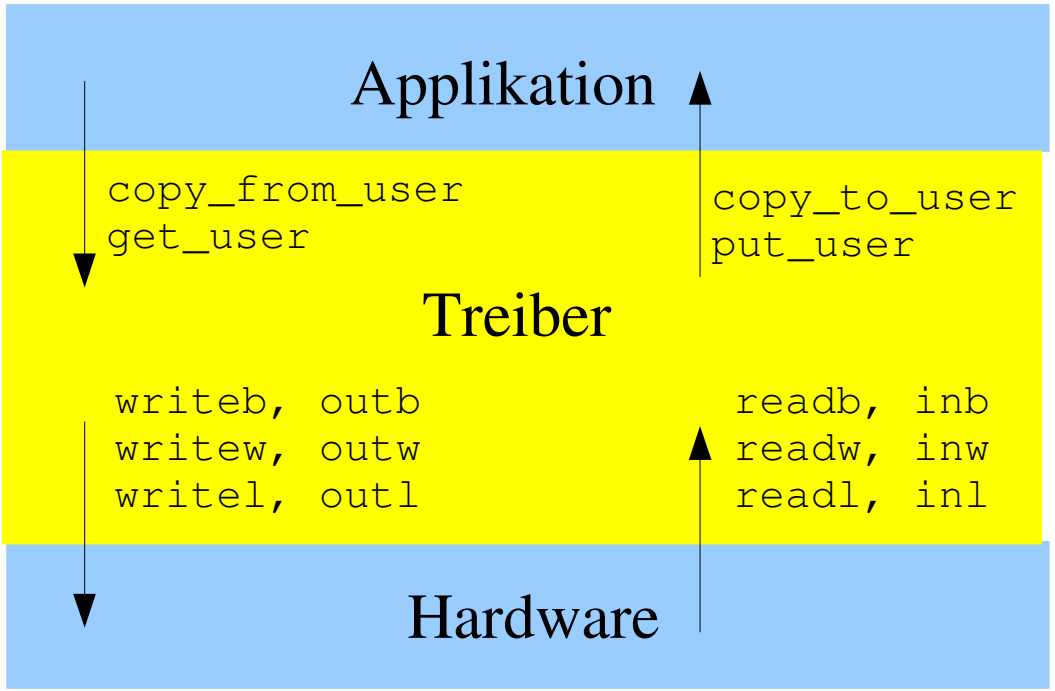
```
$ make
$ mknod MyDevice c 240 0
$ cat /etc/motd > MyDevice
$ tail -f /var/log/messages
```

Zusammenfassung

- 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.

Hardware-Anbindung

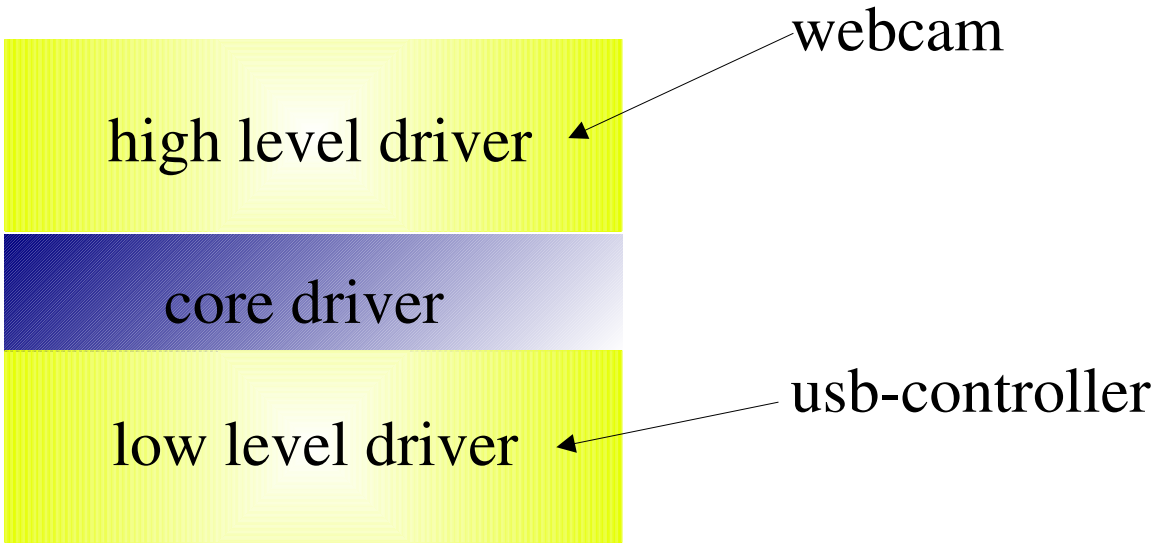
Datentransfer User/Kernel/Hardware



Datentransfer Kernel-/User-Space

- `copy_to_user`: Kopieren eines Speicherbereichs (wie memcpy)
- `copy_from_user`: Kopieren eines Speicherbereichs (wie memcpy)
- `put_user`: Kopieren eines „orginären“ Datentyps (char, short, int, long)
- `get_user`: Kopieren eines „orginären“ Datentyps (char, short, int, long)

Hardwarezugriff



Hardware-Anbindung

- „low level“-Treiber greifen direkt auf die Hardware zu.
- „high level“-Treiber nutzen Funktionen, die ihnen die „core driver“ zur Verfügung stellen. USB-Geräte greifen beispielsweise nie selbst direkt auf Hardware zu.

Datentransfer Kernel/Hardware (low level)

- Memory Mapped:
  - write[bwl]
  - read[bwl]
- IO-Ports:
  - out[bwl], out[bwl]_p, outs[bwl]
  - in[bwl], in[bwl]_p, ins[bwl]

Prof. Dr.-Ing. J. Quade

Datentypen

u8, u16, u32, u64	vorzeichenlose Datentypen
s8, s16, s32, s64	vorzeichenbehaftete
__le16, __le32, __le64	little endian
__be16, __be32, __be64	big endian

Datenstrukturen

Normale Ablage

Adresse	Variable	
0xbffffa8c	u8	a
0xbffffa8d	s8	b
0xbffffa8e	frei	
0xbffffa8f		
0xbffffa90	s32	c
0xbffffa91		
0xbffffa92		
0xbffffa93		
0xbffffa94	s8	d
0xbffffa95	frei	
0xbffffa96	u16	e
0xbffffa97		

Gepackte Ablage

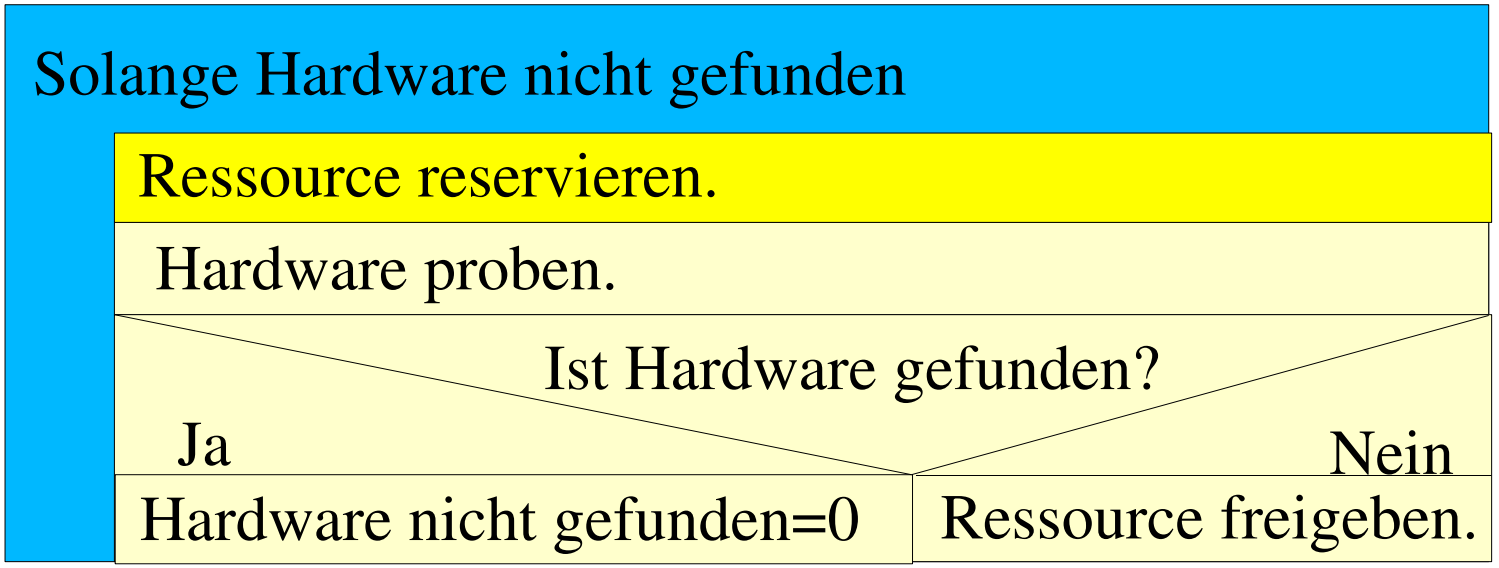
Adresse	Variable	
0xbffffa8c	u8	a
0xbffffa8d	s8	b
0xbffffa8e	s32	c
0xbffffa8f		
0xbffffa90		
0xbffffa91		
0xbffffa92	s8	d
0xbffffa93	u16	e
0xbffffa94		
0xbffffa95		
0xbffffa96		
0xbffffa97		

```
struct _LetterVar {
    __u8 a;
    __s8 b;
    __s32 c;
    __s8 d;
    __u16 e;
} __attribute__ ((packed));
```

Hardware-Erkennung

- Vor dem Zugriff müssen die Hardware-Ressourcen reserviert werden.
- Ressourcen:
  - IO-Ports: `request_region(), release_region()`
  - Interrupts: `request_irq(), free_irq()`
  - Memory: `request_mem_region(), release_mem_region()`
  - DMA-Kanäle: `request_dma(), release_dma()`

Manuelle Ressourcen-Detektion



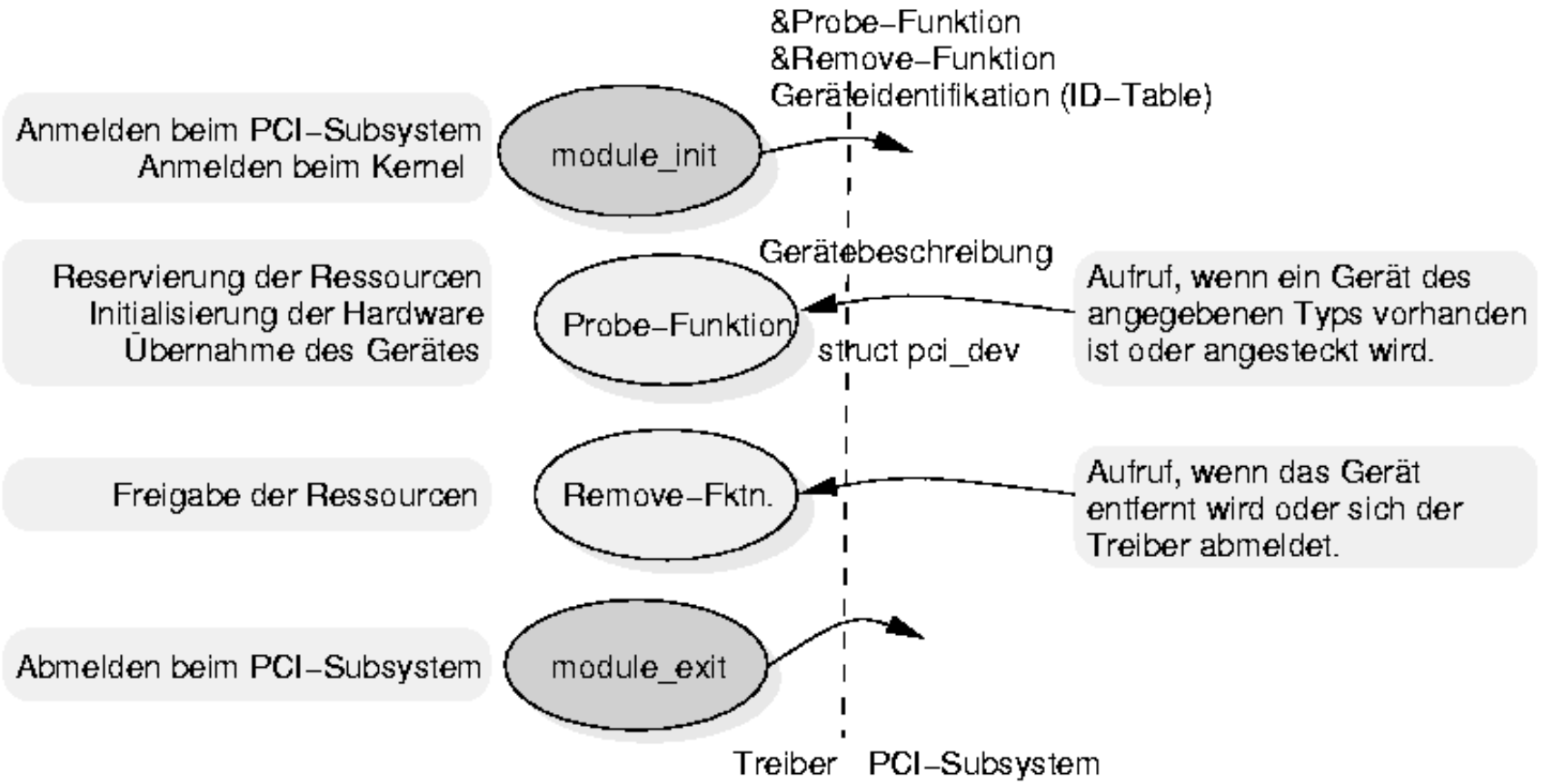
Hardware-Anbindung

- Anbindung einer PCI-Hardware:
  - Der Treiber spezifiziert die Geräte (Vendor- und Device-ID), für die er verantwortlich ist, sowie eine „Probe“ und eine „Remove“ Funktion.
  - Der Treiber meldet sich beim PCI-Subsystem an.
  - Ist eine entsprechende Hardware vorhanden, wird die Probefunktion aufgerufen. Der Treiber bekommt die zu allozierenden Ressourcen mitgeteilt.
  - Nachdem die Ressourcen alloziert wurden, ist der „low level“ Zugriff wie beschrieben möglich.

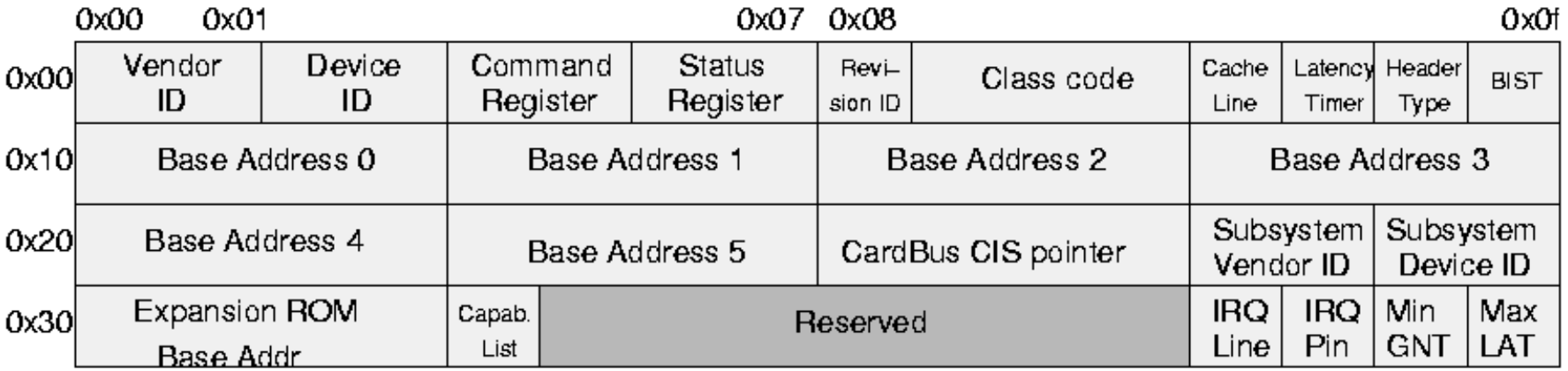
Hardware-Anbindung: PCI

- Treiber- und Geräteinitialisierung sind voneinander getrennt.
- Standardnamen für Hardware-Erkennung und Geräteinitialisierung:
  - probe
  - init_one

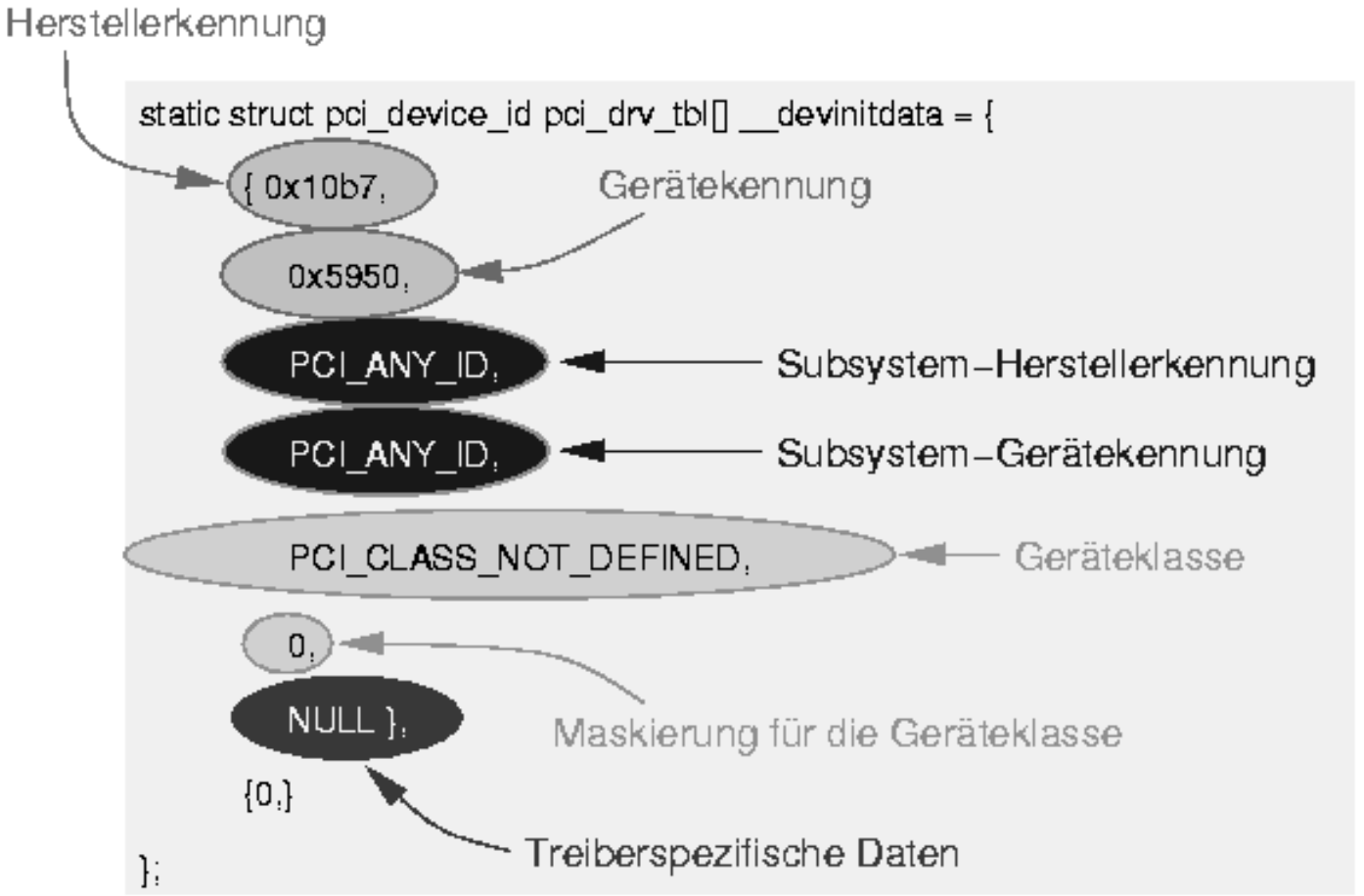
PCI


&Probe–Funktion
&Remove–Funktion
Geräteidentifikation (ID–Table)
Anmelden beim PCI–Subsystem
Anmelden beim Kernel
module_init
Gerätebeschreibung
Reservierung der Ressourcen
Initialisierung der Hardware
Übernahme des Gerätes
Probe–Funktion
struct pci_dev
Aufruf, wenn ein Gerät des angegebenen Typs vorhanden ist oder angesteckt wird.
Freigabe der Ressourcen
Remove–Fktn.
Aufruf, wenn das Gerät entfernt wird oder sich der Treiber abmeldet.
Abmelden beim PCI–Subsystem
module_exit
Treiber
PCI–Subsystem

PCI Geräte-Identifikation



PCI: Device Identifikation



Code-Beispiel

```
static struct pci_device_id pci_drv_tbl[] __devinitdata = {
    { MY_VENDOR_ID, MY_DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
    { 0, }
};

static struct pci_driver pci_drv = {
    .name=     "pci_drv",
    .id_table= pci_drv_tbl,
    .probe=    device_init,
    .remove=   device_deinit,
};

static int __init pci_drv_init(void)
{
    if((my_major_nr=register_chrdev(0,"PCI-Driver",&pci_fops))) {
        if( pci_module_init(&pci_drv) < 0 ) {
            unregister_chrdev(my_major_nr,"PCI-Driver");
            return -EIO;
        }
        return 0;
    }
    return -EIO;
}
```

Geräte-Identifikation

PCI-Treiber-Objekt

Funktion zur Geräteinitialisierung

Treiberinitialisierung

Anmelden beim PCI-Subsystem

Code-Beispiel - Fortsetzung

Treiber-Deinitialisierung

Abmelden beim PCI-Subsystem.

```
...
static void __exit pci_drv_exit(void)
{
    pci_unregister_driver( &pci_drv );
    unregister_chrdev(my_major_nr,"PCI-Driver");
}

module_init(pci_drv_init);
module_exit(pci_drv_exit);
```

Geräte-Initialisierung

```
static unsigned long ioport=0L, iolen=0L, memstart=0L, memlen=0L;
...
static int device_init(struct pci_dev *device, const struct pci_device_id *id)
{
    pci_enable_device( device ); // Geraet aktivieren
    if( request_irq(device->irq,pci_isr,SA_INTERRUPT|SA_SHIRQ,"pci_drv",device) ) {
        return -EIO;
    }
    ioport = pci_resource_start( device, 0 );
    iolen = pci_resource_len( device, 0 );
    if( request_region( ioport, iolen, device->dev.kobj.name )==NULL ) {
        free_irq( Device->irq, device );
        return -EIO;
    }
    memstart = pci_resource_start( device, 1 );
    memlen = pci_resource_len( device, 1 );
    if( request_mem_region( memstart, memlen, device->dev.kobj.name )==NULL ) {
        release_region( ioport, iolen );
        free_irq( Device->irq, device );
        return -EIO;
    }
    return 0;
}
```

Adresslagen abfragen.

Ressourcen reservieren.

Zusammenfassung PCI

- Treiberinitialisierung:
  - Anmelden beim PCI-Subsystem
  - Hierzu Übergabe der Funktionsadressen zur Geräte-initialisierung und -deinitialisierung
  - Übergabe der PCI-Hardware-Identifikation
- Geräteinitialisierung
  - Adresslagen der Ressourcen abfragen
  - Ressourcen reservieren

Zugriffsmodi

Infos:

- Quade,Kunst: Linux-Treiber entwickeln. 2. Auflage, dpunkt-Verlag, 2006, S.136-140.

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
```

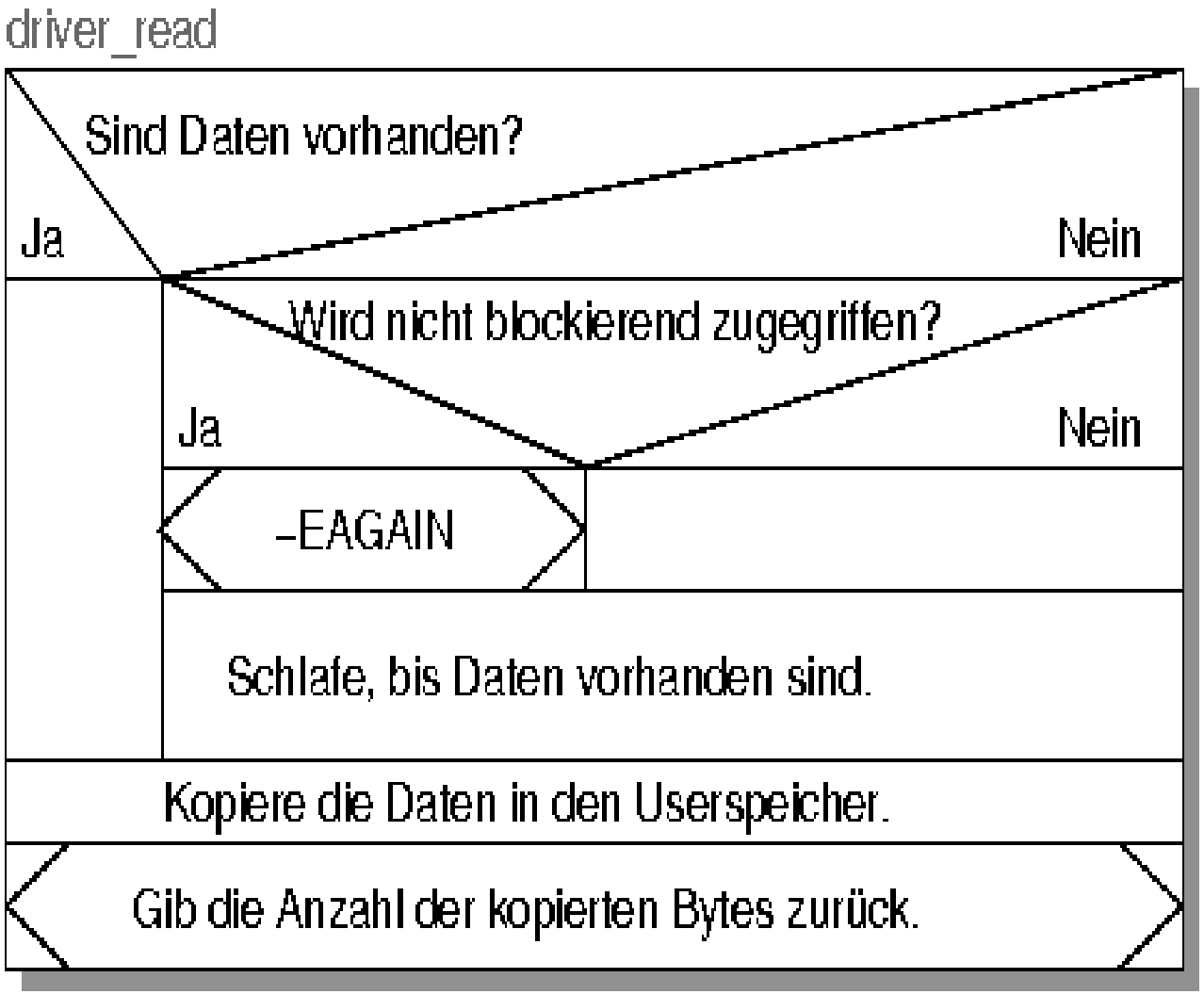
Blocking:

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

Zugriffsmodi

- Der Taskzustand kann über die folgenden Befehle beeinflusst werden:
  - `wait_event()`
  - `wait_event_interruptible()`
  - `wait_event_interruptible_timeout()`
  - `wait_event_for_completion()`
  - `wake_up()`
  - `wake_up_interruptible()`
- Interruptible-Funktionen sind durch Signals unterbrechbar.

Zugriffsmodi

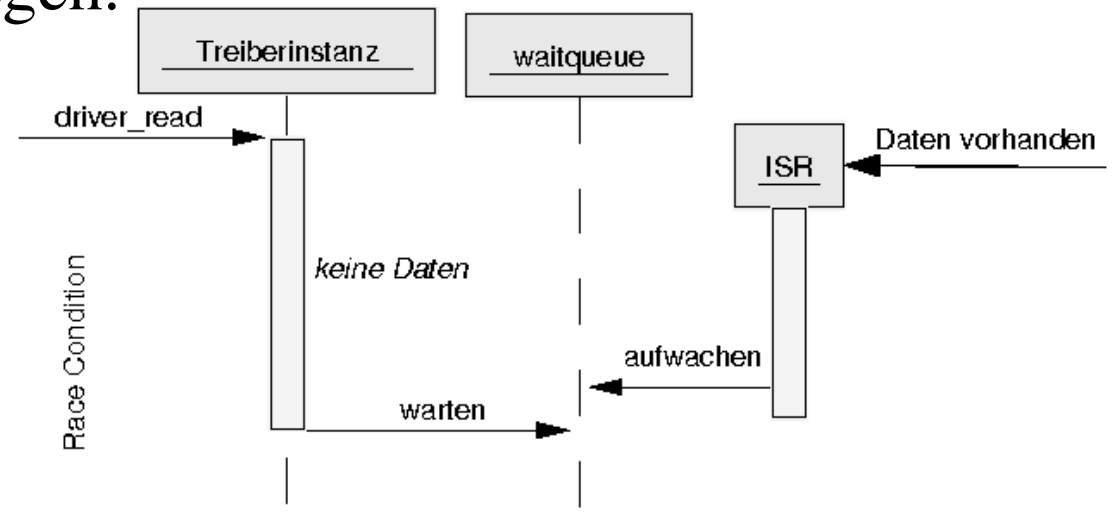


Zugriffsmodi: wait_event_interruptible

- 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);`

Zugriffsmodi: wait_event_interruptible

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



Codefragment „Zugriffsmodus“

Objekt instanzieren.

```
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 initialisieren.

Warten, solange die Wartebedingung nicht erfüllt ist.

Wartebedingung ist erfüllt

Signals

- `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!

Signals

- 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`

Signals

- 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
```

Übung 1.9: Zugriffsmodus

1. Schreiben Sie eine driver_write Funktion, die die übergebenen Daten in einen internen Buffer ablegt.
2. Schreiben Sie eine driver_read Funktion, die die im Buffer befindlichen Daten der aufrufenden Applikation übergibt. Übergebene Daten sollen aus dem Buffer gelöscht werden.
   Sind im Buffer keine Daten vorhanden, soll die Applikation, die driver_read aufgerufen hat, schlafen gelegt werden.

Übung 1.9: Zugriffsmodus

3. Compilieren und laden Sie den Treiber.
4. Testen Sie ihren Treiber, in dem Sie folgendermaßen vorgehen:
   a) Lesen Sie von dem durch das Treiber bediente Gerät. Da der Buffer leer ist, muss die lesende Applikation (z.B. cat geraetedatei) schlafen gelegt werden.
   b) Schreiben Sie Daten auf das durch den Treiber bediente Gerät (z.B. durch >>echo "hallo" >geraetedatei<<). Jetzt muss im ersten Fenster die Ausgabe erscheinen, die zugehörige Applikation sich danach wieder schlafen legen.

Zusammenfassung

- 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.

Treiberinstanzen

Infos:

- Quade,Kunst: Linux-Treiber entwickeln. 2. Auflage, dpunkt-Verlag, 2006, S.140-142.

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.

Treiberinstanzen

- 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.

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

Übung 1.10: Treiberinstanz

- Kopieren Sie das Modul „hello.c“ in die Datei „hello2.c“.
- Erweitern Sie „hello2.c“ deart, dass zugreifende Instanzen genau einmal den String „Hello World“ zurückbekommen.
- Deklarieren Sie für den Zähler, der die Anzahl der bereits gelesenen Bytes zählt, eine eigene Datenstruktur.
- Reservieren Sie den Speicher für diesen Zähler (Datenstruktur) dynamisch unter Zuhilfenahme der Funktion kmalloc().
- Vergessen Sie nicht den Speicher in der Funktion driver_close() wieder freizugeben.

Zusammenfassung

- 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.