

# **Treiberentwicklung, Echtzeit- und Betriebssysteme**

Sommersemester 2014  
Prof. Dr. Peter Gerwinski

Stand: 25. April 2014

Soweit nicht anders angegeben:

Copyright © 2014 Peter Gerwinski

Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)

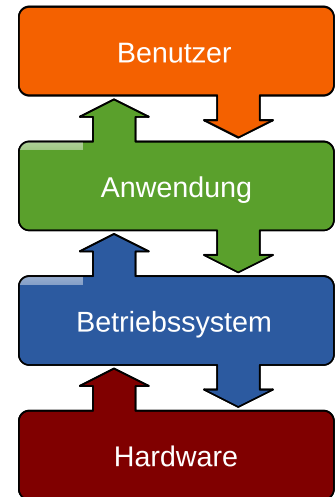
# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>4</b>
<b>2 Treiber</b>	<b>5</b>
2.1 Einführung in die Treiber-Programmierung . . . . .	5
2.1.1 Beispiel: Speicher direkt ansprechen . . . . .	5
2.1.2 Beispiel: Grundstruktur eines Kernel-Moduls . . . . .	6
2.2 Kommunikation zwischen Treiber und Anwenderprogramm: Gerätedatei . . . . .	7
2.2.1 Datei-Operationen aus Sicht des Treibers . . . . .	7
2.2.2 Datei-Operationen aus Sicht des Anwenderprogramms . . . . .	7
2.2.3 Datei-Operationen aus Sicht des Betriebssystemkerns . . . . .	8
<b>3 Echtzeit</b>	<b>8</b>
3.1 Was ist Echtzeit? . . . . .	8
3.2 Multitasking . . . . .	8
3.3 Ressourcen . . . . .	8
<b>4 Speicherverwaltung</b>	<b>8</b>
4.1 Bank Switching . . . . .	8
4.2 Speichersegmentierung . . . . .	8
4.3 Virtuelle Speicherverwaltung . . . . .	8
<b>5 Grafik</b>	<b>8</b>
5.1 Hardwarenahe Aspekte . . . . .	8
5.2 Low Level vs. High Level . . . . .	8
5.3 Hardwarebeschleunigung . . . . .	8
5.4 2d-Grafikbibliotheken . . . . .	8
5.5 3d-Grafikbibliotheken . . . . .	9
<b>6 Massenspeicher</b>	<b>9</b>
6.1 Block-Gerätedateien . . . . .	9
6.2 Dateisysteme . . . . .	9
<b>7 Netzwerk</b>	<b>9</b>
<b>8 Sicherheit</b>	<b>9</b>

# 1 Einführung

Was ist ein Betriebssystem?

- Software, die zwischen Hardware und Anwendung vermittelt
- Mikro-Controller:  
Anwendung greift *direkt* auf Hardware zu
- Eingebettetes System:  
Anwendung startet automatisch
- Arbeitsplatz-Computer: *Oberfläche (Shell)*
- Ressourcen-Verwaltung



Was gehört zum Betriebssystem?

- Betriebssystemkern: *Kernel*
- Benutzeroberfläche: *Shell*  
text- oder grafikorientiert  
(im engeren Sinne: Kommandozeile)
- Werkzeuge zur Verwaltung von Ressourcen  
(z. B. Festplatten formatieren)
- Graphische Benutzeroberfläche: *GUI*
- Texteditor
- Entwicklungswerkzeuge (Compiler usw.),  
Skriptsprachen
- Internet-Software: Web-Browser, E-Mail-Client usw.
- Multimedia-Software
- Büro-Anwendungssoftware

Ja, klar!

Hmm ... vielleicht.

In dieser Lehrveranstaltung:

- Treiberentwicklung  
wie in *Angewandte Informatik* (5. Sem.), „größer“
- Echtzeitsysteme  
wie in *Vertiefung Systemtechnik* (7. Sem.), „größer“
- neu: Betriebssysteme

Statt Klausur: Projektaufgabe, z. B.:

- neuartiger Treiber (z. B. für neuartige Hardware)
- neuartige Echtzeit-Funktionalität
- Sonstiges

Wiederholung:

- ? Hardwarenahe Programmierung
- Theorie der Echtzeit-Systeme
- ? Sonstiges

## 2 Treiber

### 2.1 Einführung in die Treiber-Programmierung

Als erstes schauen wir uns einen existierenden Treiber an, das wir für einen konkreten praktischen Zweck benötigen.

#### 2.1.1 Beispiel: Speicher direkt ansprechen

Wir betrachten die folgende Situation:

- An die *General Purpose Input/Output (GPIO)*-Anschlüsse eines ARM-Cortex-A7-Prozessors sind schließbare Kontakte („Taster“) angeschlossen.
- Eine Software ([tastatur.c](#)) fragt die Anschlüsse ab und setzt Eingaben in Buchstaben um („Tastatur“).
- In einem späteren Beispiel wird ein Treiber entwickelt, der die Kontakte als „echte“ Tastatur anderen Programmen zur Verfügung stellt.

Auf Mikro-Controllern ist es üblich, Speicheradressen „einfach so“ direkt anzusprechen. Auf diese Weise erfolgt z. B. der Zugriff auf Input- und Output-Ports.

Beispiel: LED an einem AVR ATmega blinken lassen

```
#include <avr/io.h>

#define F_CPU 8000000
#include <util/delay.h>

int main (void)
{
    DDRB = 0x83; /* Bit 7, 1 und 0 auf Port B für Output konfigurieren */
    PORTB = 0; /* Bits an Port B auf 0 setzen */
    DDRC = 0x70; /* Bit 4, 5 und 6 an Port C für Output konfigurieren */
    PORTC = 0; /* Bits an Port C auf 0 setzen */
    while (1)
    {
        _delay_ms (500);
        PORTC = 0x40; /* Bit 6 an Port C auf 1 setzen */
        _delay_ms (500);
        PORTC = 0x00;
    }
    return 0;
}
```

Bei `DDRB`, `PORTC` usw. handelt es sich um Präprozessor-Macros, die über einen **volatile**-Pointer direkt eine Speicherzelle (mit bekannter numerischer Adresse) als Variable zur Verfügung stellen. (Details wurden in der Lehrveranstaltung *Angewandte Informatik* behandelt.)

Genau dieselbe Vorgehensweise ist auf einem „richtigen“ Computer *mit Betriebssystem* nicht möglich. (Unter manchen Betriebssystemen, darunter MS-DOS und ältere Versionen von MS-Windows, geht es trotzdem.) Stattdessen spricht man den Speicher über einen *Treiber* an.

Unter Linux stellt der Treiber eine Pseudo-Datei `/dev/mem` zur Verfügung, eine sogenannte *Character Device*. Diese verhält sich in vielerlei Hinsicht wie eine „normale“ Datei; insbesondere liegt sie in einem „normalen“ Verzeichnis, kann „normal“ umbenannt und verschoben werden, und man kann mit „normalen“ Lese- und Schreibbefehlen darauf zugreifen.

Für unsere konkrete Anwendung „Zugriff auf Input- und Output-Ports“ verwenden wir eine Bibliothek `gpio_lib`. Diese spricht die Datei anstatt über „normale“ Lese- und Schreibbefehle über *Memory Mapping* an, also als ein Array im Speicher. Auf dieses Detail wird an dieser Stelle nicht weiter eingegangen; stattdessen betrachten wir, wie der Kernel die Pseudo-Datei `/dev/mem` realisiert.

`/dev/mem` ist eine Character Device, daher findet man ihren Treiber im Unterverzeichnis `drivers/char` des Linux-Quelltextes in einer Datei `mem.c`.

Für das Lesen von Speicher ist die Funktion

```
static ssize_t read_mem(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos)
```

zuständig.

Die Funktion prüft mehrmals, ob der Lesezugriff zulässig ist und bricht andernfalls mit einer Fehlermeldung ab.

Der eigentliche Lesezugriff erfolgt über einen Aufruf der Funktion `copy_to_user()`.

Hierbei handelt es sich tatsächlich um einen Präprozessor-Macro, der stark von der verwendeten Hardware abhängig und daher für jede Architektur unterschiedlich implementiert ist.

Eine Volltextsuche im Quelltextbaum, z. B. mit `grep "copy_to_user" $(find . -name "*.h")`, fördert einige Implementationen zutage, darunter z. B. die Datei `arch/arm/include/asm/uaccess.h`, die für den von uns verwendeten ARM-Prozessor zuständig ist.

## 2.1.2 Beispiel: Grundstruktur eines Kernel-Moduls

Um einen neuen Treiber für den Linux-Kernel zu schreiben, gibt es zahlreiche Anleitungen im Netz, z. B.: <http://www.cyberciti.biz/tips/compiling-linux-kernel-module.html>

Falls noch nicht vorhanden, installiert man zunächst die Entwicklungswerkzeuge (Compiler in der passenden Version usw.) sowie die Header-Dateien des Kernels:

```
apt-get install linux-headers-$(uname -r)
```

(Auch wenn hier nur die Header-Dateien angegeben sind, werden die Entwicklungswerkzeuge automatisch mitinstalliert.)

Die nachfolgenden Beispiele sind dem Linux Documentation Project (LDP) entnommen:

<http://www.tldp.org/LDP/lkmpg/2.6/html/x121.html>

So wie ein „normales“ Programm eine `main()`-Funktion enthält, enthält ein Treiber – ein *Kernel-Modul* – eine Funktion `init_module()`, die beim Laden des Moduls ausgeführt wird, und eine Funktion `cleanup_module()`, die beim Entfernen des Moduls ausgeführt wird.

Das „Hello World!“ unter den Treibern ist ein Treiber, bei dem diese Funktionen lediglich einen Text ausgeben: `hellomod-1.c`

```
/*
 * hello-1.c – The simplest kernel module.
 */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello_world_1.\n");

    /*
     * A non 0 return means init_module failed; module can't be loaded.
     */
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye_world_1.\n");
}
```

Da ein Treiber keinen Zugriff auf einen „normalen“ Bildschirm hat, wird der Text mit `printk()` anstelle von `printf()` ausgegeben. Anstatt auf einem Bildschirm landet er in einem Speicherbereich, der üblicherweise von einem im Hintergrund laufenden Programm (*daemon*) in eine Log-Datei geschrieben wird.

Das Compilieren des Moduls ist eng mit dem Compilieren des Kernels verknüpft. Um den Aufwand überschaubar zu halten, ist das meiste automatisiert, so daß es genügt, ein [Makefile](#) zu schreiben, das das neue Modul beim [Makefile](#) des Kernels „anmeldet“.

Tatsächlich müssen die Funktionen nicht (mehr) `init_module()` und `cleanup_module()` heißen – siehe [hellomod-2.c](#).

## 2.2 Kommunikation zwischen Treiber und Anwenderprogramm: Gerätedatei

Ein einfaches Beispiel für einen Treiber, der tatsächlich etwas macht, ist einer, der einem Anwenderprogramm eine Information bereitstellt. Diese könnte z. B. von einem Sensor stammen; in unserem Fall handelt es sich lediglich um den Text `"Hello,_world!\n"`.

Siehe: [chardev-1.c](#)

Im Dateisystem legt man eine Gerätedatei an (hier: `/dev/chardev`).

Diese ist aus Sicht des Anwenderprogramms über ihren Dateinamen ansprechbar. Aus Sicht des Moduls ist sie über zwei Zahlen charakterisiert, die *Major* und *Minor Device Number*.

### 2.2.1 Datei-Operationen aus Sicht des Treibers

Ähnlich einer Klasse (objektorientierte Programmierung) stellt das Kernel-Modul mehrere Funktionen („Methoden“) zur Verfügung, um auf die Datei lesend und schreibend zugreifen zu können.

Unter Linux geschieht dies über eine Struktur `struct file_operations`. Diese enthält u. a. die Felder `.read`, `.write`, `.open` und `.release`, bei denen es sich um Zeiger auf Funktionen handelt, die die entsprechenden Datei-Operationen implementieren.

Unter Microsoft Windows geschieht i. w. dasselbe über ein Array `pDriverObject->MajorFunction` mit den Indices `IRP_MJ_READ`, `IRP_MJ_WRITE`, `IRP_MJ_CREATE` und `IRP_MJ_CLOSE`.

(Quelle: <http://www.codeproject.com/Articles/9504/Driver-Development-Part-Introduction-to-Drivers>; siehe auch: <http://myworks2012.wordpress.com/2012/10/07/how-to-compile-windows-driver-using-mingw-gcc/> sowie [http://www.fccps.cz/download/adv/frr/win32\\_ddk\\_mingw/win32\\_ddk\\_mingw.html](http://www.fccps.cz/download/adv/frr/win32_ddk_mingw/win32_ddk_mingw.html))

### 2.2.2 Datei-Operationen aus Sicht des Anwenderprogramms

Das klassische „Hello, world!“-Programm nutzt die Bibliotheksfunktion `printf()`, um den String `"Hello,_world!\n"` zur Standardausgabe, eine Datei `stdout`, zu schreiben.

Bei der Optimierung ersetzt der Compiler den Aufruf von `printf()` durch einen Aufruf der „leichtgewichtigeren“ Funktion `puts()`. Beide Funktionen werden in der Systembibliothek `glibc` implementiert. Unter Debian GNU/Linux 7.1 „wheezy“ handelt es sich dabei um die Version 2.13 für eingebettete Systeme, `eglibc-2.13`.

Die Funktion `puts()` puffert die Eingabe und prüft auf konkurrierenden Zugriff mehrerer Anwenderprogramme auf dieselbe Ressource. (Dies wird später separat ausführlich behandelt. Siehe auch die Lehrveranstaltung: *Vertiefung Systemtechnik*)

Um letztlich die Daten in die Datei zu schreiben, ruft `puts()` die Bibliotheksfunktion `write()` auf.

Bei dieser schließlich handelt es sich um den Aufruf einer Systemfunktion, die hardwaremäßig anders implementiert sein kann als eine „normale“ Funktion. Der Grund dafür ist der folgende:

Prozessoren für PCs (Intel: ab 80286; andere: bereits früher) unterscheiden *Privilegierungsstufen (Ringe, Domains)* für verschiedene Programme. Üblich sind mindestens 4 Stufen; Betriebssysteme nutzen üblicherweise nur 2.

Der *Betriebssystemkern (Kernel)* läuft im *Kernel Space* in der höchsten Privilegierungsstufe, *Ring 0*. Anwenderprogramme laufen im *User Space* in der niedrigsten Privilegierungsstufe, *Ring 3*. In Ring 0 stehen zusätzliche Befehle zur Verfügung, u. a. zur Regelung des Zugriffes auf Speicher, I/O-Ports und Interrupts. (Zusätzliche, dazwischenliegende Ringe finden z. B. bei *Virtualisierung* Verwendung.

Der Aufruf einer Funktion im Kernel Space aus dem User Space heraus muß über eine Schnittstelle erfolgen, die den Erwerb zusätzlicher Rechte kontrolliert.

Unter 32-Bit-Intel-Linux sowie DOS-basierten Versionen von Microsoft Windows werden hierfür Software-Interrupts aufgerufen; unter 64-Bit-Intel-Linux sowie neueren Versionen von Microsoft Windows geschieht dies über einen speziellen Assembler-Befehl `syscall`.

Treiber können im Kernel Space oder im User Space laufen.  
Die bisher betrachteten Treiber laufen im Kernel Space.

### 2.2.3 Datei-Operationen aus Sicht des Betriebssystemkerns

Der Kernel identifiziert die Datei über ihren *Index Node (Inode)*.

Diesem entnimmt er den Satz von Funktionen, die für den Zugriff auf diese spezielle Datei zuständig sind, und reicht letztlich den Systemaufruf der Funktion `write()` durch an die entsprechende Methode `.write` innerhalb der `struct file_operations`.

## 3 Echtzeit

### 3.1 Was ist Echtzeit?

### 3.2 Multitasking

### 3.3 Ressourcen

## 4 Speicherverwaltung

### 4.1 Bank Switching

### 4.2 Speichersegmentierung

### 4.3 Virtuelle Speicherverwaltung

## 5 Grafik

### 5.1 Hardwarenahe Aspekte

### 5.2 Low Level vs. High Level

### 5.3 Hardwarebeschleunigung

### 5.4 2d-Grafikbibliotheken

Systembibliothek (so „low-level“ wie möglich):

- X11 (hauptsächlich Unix): Xlib
- MS-Windows: GDI

GUI-Bibliotheken:

- plattformunabhängig: Gtk+, Qt, wxWidgets, JFC, ...
- nur MS-Windows: MFC



## **5.5 3d-Grafikbibliotheken**

- plattformunabhängig: OpenGL
- nur MS-Windows: Direct3d

## **6 Massenspeicher**

### **6.1 Block-Gerätedateien**

### **6.2 Dateisysteme**

## **7 Netzwerk**

## **8 Sicherheit**