

Lernzettel

Geräteunabhängige Ein-/Ausgabe und
Treiberarchitektur: Abstraktionen,
Kernel-Interfaces, Treiberdesign in C

Universität: Technische Universität Berlin
Kurs/Modul: Systemprogrammierung
Erstellungsdatum: September 6, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Systemprogrammierung

Lernzettel: Geräteunabhängige Ein-/Ausgabe und Treiberarchitektur: Abstraktionen, Kernel-Interfaces, Treiberdesign in C

(1) Zielsetzung und Kontext. Die geräteunabhängige E/A trennt Anwendungslogik von konkreter Hardware. Der Kernel kapselt Treiberlogik von Benutzerspace-Anwendungen und bietet über Abstraktionen eine einheitliche Schnittstelle an. Das Lernziel hier ist ein klares Verständnis, wie Abstraktionen, Kernel-Interfaces und Treiberdesign in C zusammenwirken, um Geräte unabhängig von der jeweiligen Hardware nutzbar zu machen.

(2) Abstraktionen der E/A. - Virtuelle Geräte und das Virtual File System (VFS) bilden eine konsistente Schicht zwischen Anwendungen und Treibern. - Major/Minor-Nummern kodieren Geräteidentität; /dev-Verzeichnis dient als zentrale Benutzerschnittstelle. - Block- vs. Character-Devices: Blockgeräte (Zugriff in Blöcken) vs. Character-Geräte (Zugriff in Streams). - Standardisierte Dateischnittstellen (open, read, write, close, ioctl) ermöglichen Hardware-Abstraktion.

(3) Kernel-Interfaces. - Kernel-Seiten liefern eine logische Schnittstelle für Treiber: `struct file_operations` definiert die Aufrufe, die der Treiber bereitstellt (z. B. `open`, `read`, `write`, `release`, `ioctl`). - Treiber registrieren sich als Kernel-Module oder Plattformtreiber; Probing- und Remove-Logik verwalten Lebenszyklus und Ressourcen. - Benutzerspace-Interaktion erfolgt über Kopieren von Daten mittels `copy_to_user` / `copy_from_user`.

(4) Treiberdesign in C – Grundaufbau. - Modul-Initialisierung und -Aufräumen: `module_init` / `module_exit`. - Kernel-Schnittstelle: `struct file_operations fops` mit Implementierungen für `open`, `read`, `write`, `ioctl`, ggf. `llseek`. - Ressourcenmanagement: Speicherebenen (I/O-Register, DMA-Puffer), IRQ-Handling (Anfordern von Interrupts), Synchronisation (Spinlocks, Mutex). - Sicherheit und Robustheit: Fehlerbehandlung, saubere Freigabe von Ressourcen, Prüfen von Nutzerdatenzugriffen.

(5) Skeleton eines Linux-Gerätetreibers (Character Device) in C.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>    // copy_to_user / copy_from_user
#include <linux/cdev.h>

#define DEV_NAME "mydrv"
#define DEV_MAJOR 240

static int my_open(struct inode *inode, struct file *filp) { return 0; }
static int my_release(struct inode *inode, struct file *filp) { return 0; }

static ssize_t my_read(struct file *filp, char __user *buf, size_t count, loff_t *offset)
{
    const char *msg = "Hallo aus dem Treiber!\n";
    size_t len = strlen(msg);
    if (*offset >= len) return 0;
    if (count > len - *offset) count = len - *offset;
    if (copy_to_user(buf, msg + *offset, count)) return -EFAULT;
```

```
    *offset += count;
    return count;
}

static ssize_t my_write(struct file *filp, const char __user *buf,
                        size_t count, loff_t *offset)
{
    char kbuf[128];
    if (count > sizeof(kbuf) - 1) count = sizeof(kbuf) - 1;
    if (copy_from_user(kbuf, buf, count)) return -EFAULT;
    kbuf[count] = '\0';
    printk(KERN_INFO "mydrv: wrote: %s\n", kbuf);
    return count;
}

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .write = my_write,
    .release = my_release,
};

static int __init my_init(void)
{
    int ret;
    ret = register_chrdev(DEV_MAJOR, DEV_NAME, &fops);
    if (ret < 0) {
        printk(KERN_ERR "mydrv: cannot register device\n");
        return ret;
    }
    printk(KERN_INFO "mydrv: registered with major %d\n", DEV_MAJOR);
    return 0;
}

static void __exit my_exit(void)
{
    unregister_chrdev(DEV_MAJOR, DEV_NAME);
    printk(KERN_INFO "mydrv: unregistered\n");
}

module_init(my_init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kursleiter");
MODULE_DESCRIPTION("Beispiel-Geräte-Treiber-Skelett");
```

(6) Erweiterte Treiber-Funktionen (Beispiele). - Speicher-Mapped IO (MMIO): Zugriff auf Geräte-Register via `ioremap` / `iounmap`. - Interrupts (IRQ): `request_irq` / `free_irq` mit ISR-Handler. - DMA-Unterstützung: Zuweisung von Pufferbereichen, Abgleich von Adressen, Synchronisation mit dem Device. - Zugriff auf Benutzerdaten: Verwendung von `copy_to_user` / `copy_from_user`.

(7) Testing, Debugging und Profiling. - Kernel-Logs: `printk` mit Level (`KERN_INFO`, `KERN_DEBUG`, `KERN_ERR`). - Benutzerseitige Tests über `cat` / `echo` auf `/dev/mydrv`. - Tools: `dmesg`, `perf`, `ftrace` für Performance und Trace.

(8) Sicherheitsrelevante Aspekte und Nachhaltigkeit. - Strenge Validierung von Nutzerdaten; Copy-Operationen prüfen. - Zugriffskontrollen: Berechtigungen des Dateisystems, Capability-Checks. - Ressourcenhygiene: saubere Freigabe von Speicher, IRQs und DMA-Ressourcen. - Energie- und Ressourcenbewusstsein: effiziente Sperrmechanismen, minimale Kernel-Workloads.

(9) Kurzzusammenfassung. - Geräteunabhängige E/A wird durch Abstraktionen wie VFS, Major/Minor und verschiedene Device-Typen realisiert. - Kernel-Interfaces definieren, wie Treiber im Kernel reisen und mit Benutzernspace interagieren. - Treiberdesign in C folgt klaren Mustern: Modulein-/auschluss, `file_operations`, Ressourcen- und Interrupt-Management, sichere Kommunikation mit Benutzernspace. - Sicherheit, Testing und Nachhaltigkeit sind integrale Bestandteile des Treiberentwicklungsprozesses.