

# Treiberentwicklung, Echtzeit- und Betriebssysteme

Prof. Dr. rer. nat. Peter Gerwinski

28. April 2025

# Treiberentwicklung, Echtzeit- und Betriebssysteme

## 1 Einführung

1.1 Was ist ein Betriebssystem?

1.2 Zu dieser Lehrveranstaltung

## 2 Unix

2.1 Grundkonzepte

2.2 Die Kommandozeile: Grundlagen

2.3 Dateisysteme

2.4 Ein- und Ausgabeströme

2.5 Pipes

2.6 Verzweigungen und Schleifen

## 3 Treiberentwicklung

3.0 Massenspeicher und Gerätedateien

3.1 Mikrocontroller

3.2 Betriebssysteme ohne Speicherschutz

3.3 Linux-Kernel-Module

...

## 3 Treiberentwicklung

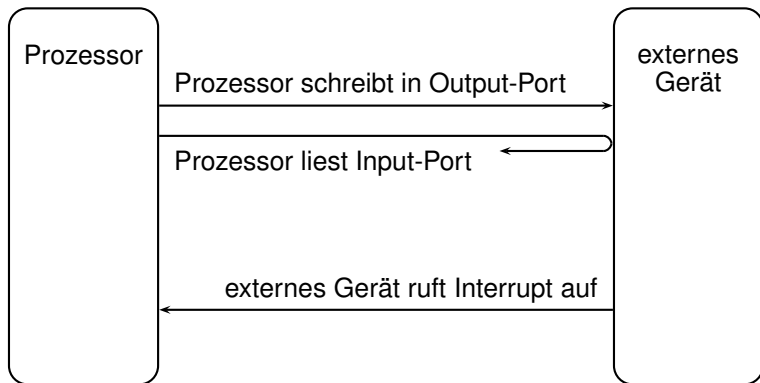
### 3.0 Massenspeicher und Gerätedateien

- Linux: Ein angeschlossener USB-Stick erscheint als Gerätedatei: `/dev/sdb`
- „Normale“ Benutzung: `mount`  
(Siehe: Unix, Datenträger *einhängen*)
- Gerätedatei: Direktzugriff auf die Bytes des Datenträgers  
Anwendungen: Datensicherung, Datenrettung

## 3 Treiberentwicklung

### 3.1 Mikrocontroller

Kommunikation mit externen Geräten



## 3.1 Mikrocontroller

### 3.1.1 I/O-Ports

In Output-Port schreiben = Aktoren ansteuern

Beispiel: LED

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

```
...
```

```
DDRC = 0x70;    binär: 0111 0000
```

```
PORTC = 0x40;    binär: 0100 0000
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

*Details: siehe Datenblatt und Schaltplan*

## 3.1 Mikrocontroller

### 3.1.1 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

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

```
...
```

```
DDRC = 0xfd;          binär: 1111 1101
```

```
while ((PINC & 0x02) == 0) binär: 0000 0010
```

```
; /* just wait */
```

Herstellerspezifisch!

DDR = Data Direction Register

Bit = 1 für Output-Port

Bit = 0 für Input-Port

*Details: siehe Datenblatt und Schaltplan*

Praktikumsaufgabe in *Hardwarenahe Programmierung*: Druckknopfampel

## 3.1 Mikrocontroller

### 3.1.2 Interrupts

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“

Beispiel: eingebaute Uhr

statt Zählschleife (`_delay_ms`):  
Hauptprogramm kann  
andere Dinge tun

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

... „Dies ist ein Interrupt-Handler.“  
Interrupt-Vektor darauf zeigen lassen

```
ISR (TIMER0B_COMP_vect)
{
    PORTD ^= 0x40;
}
```

Herstellerspezifisch!

Initialisierung über spezielle Ports: `TCCR0B`, `TIMSK0`

*Details: siehe Datenblatt und Schaltplan*

## 3.1 Mikrocontroller

### 3.1.2 Interrupts

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“

Beispiel: Taster

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

```
...
```

```
ISR (INT0_vect)
```

```
{  
    PORTD ^= 0x40;  
}
```

statt *Busy Waiting*:  
Hauptprogramm kann  
andere Dinge tun

Herstellerspezifisch!

Initialisierung über spezielle Ports: `EICRA`, `EIMSK`

*Details: siehe Datenblatt und Schaltplan*



## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“

Beispiel: Taster

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

```
...
```

```
uint8_t key_pressed = 0;
```

```
ISR (INT0_vect)
```

```
{  
    key_pressed = 1;  
}
```

```
int main (void)
```

```
{
```

```
...
```

```
while (1)
```

```
{
```

```
    while (!key_pressed)
```

```
        ; /* just wait */
```

```
    PORTD ^= 0x40;
```

```
    key_pressed = 0;
```

```
}
```

```
return 0;
```

```
}
```

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Externes Gerät ruft (per Stromsignal) Unterprogramm auf  
Zeiger hinterlegen: „Interrupt-Vektor“  
Beispiel: Taster

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

```
...
```

```
volatile uint8_t key_pressed = 0;
```

```
ISR (INT0_vect)
```

```
{  
    key_pressed = 1;  
}
```

```
int main (void)
```

```
{
```

```
...
```

```
while (1)
```

```
{
```

```
    while (!key_pressed)
```

```
        ; /* just wait */
```

```
    PORTD ^= 0x40;
```


```
    key_pressed = 0;
```

```
}
```

```
return 0;
```

```
}
```

**volatile:**  
Speicherzugriff  
nicht wegoptimieren



## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (* (volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (* (volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

Zahl: 0x2B

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

→  $(*(\text{volatile uint8\_t } *) ((0x0B) + 0x20)) = 0x01;$

Umwandlung in Zeiger  
auf **volatile** uint8\_t

Zahl: 0x2B

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (* (volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

Umwandlung in Zeiger  
auf **volatile** uint8\_t

Zahl: 0x2B

Dereferenzierung des Zeigers



## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (* (volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

Umwandlung in Zeiger  
auf **volatile** uint8\_t

Zahl: 0x2B

Dereferenzierung des Zeigers

→ **volatile** uint8\_t-Variable an Speicheradresse 0x2B

## 3.1 Mikrocontroller

### 3.1.3 volatile-Variable

Was ist eigentlich PORTD?

```
avr-gcc -Wall -Os -mmcu=atmega328p blink-3.c -E
```

```
PORTD = 0x01;
```

```
→ (* (volatile uint8_t *) ((0x0B) + 0x20)) = 0x01;
```

Umwandlung in Zeiger  
auf **volatile** uint8\_t

Zahl: 0x2B

Dereferenzierung des Zeigers

→ **volatile** uint8\_t-Variable an Speicheradresse 0x2B

→ PORTA = PORTB = PORTC = PORTD = 0 ist eine schlechte Idee.

## 3.2 Betriebssysteme ohne Speicherschutz

- FreeDOS (früher: PC-DOS, MS-DOS, DR-DOS, Novell-DOS, aber auch Apple-DOS, Commodore-Kernal, ...):  
Direkter Zugriff auf Speicher, I/O-Ports und Interrupts

## 3.2 Betriebssysteme ohne Speicherschutz

- FreeDOS (früher: PC-DOS, MS-DOS, DR-DOS, Novell-DOS, aber auch Apple-DOS, Commodore-Kernal, ...):  
Direkter Zugriff auf Speicher, I/O-Ports und Interrupts
- Bildschirm (Textmodus): ab Speicherzelle **B800:0000**  
abwechselnd Zeichen (CP 437) und Attribut (Farbe, Hintergrund)

## 3.2 Betriebssysteme ohne Speicherschutz

- FreeDOS (früher: PC-DOS, MS-DOS, DR-DOS, Novell-DOS, aber auch Apple-DOS, Commodore-Kernal, ...):  
Direkter Zugriff auf Speicher, I/O-Ports und Interrupts
- Bildschirm (Textmodus): ab Speicherzelle **B800:0000**  
abwechselnd Zeichen (CP 437) und Attribut (Farbe, Hintergrund)
- Bildschirm (Grafikmodus): später

## 3.3 Linux-Kernel-Module

- **#include** <linux/module.h>  
**#include** <linux/kernel.h>
- Zwei „Hauptprogramme“: `init_module()`, `cleanup_module()`
- `printk()` statt `printf()`
- Compilieren mit speziellem **Makefile**

## 3.3 Linux-Kernel-Module

- **#include** <linux/module.h>  
**#include** <linux/kernel.h>
- Zwei „Hauptprogramme“: `init_module()`, `cleanup_module()`
- `printk()` statt `printf()`
- Compilieren mit speziellem **Makefile**
- Angabe der Lizenz ist wichtig: **MODULE\_LICENSE()**