

Angewandte Informatik

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

4. Dezember 2017

Angewandte Informatik

Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung
- 2 Einführung in C
- 3 Bibliotheken
- 4 Hardwarenahe Programmierung
 - 4.1 Bit-Operationen
 - 4.2 I/O-Ports
 - 4.3 Interrupts
 - 4.4 volatile-Variable
 - 4.5 Byte-Reihenfolge – Endianness
 - 4.6 Speicherausrichtung – Alignment
- 5 Algorithmen
 - 5.1 Differentialgleichungen
 - ...

...

4 Hardwarenahe Programmierung

4.1 Bit-Operationen

4.1.1 Zahlensysteme

Basis	Name	Beispiel	Anwendung
2	Binärsystem	1 0000 0011	Bit-Operationen
8	Oktalsystem	0403	Dateizugriffsrechte (Unix)
10	Dezimalsystem	259	Alltag
16	Hexadezimalsystem	0x103	Bit-Operationen
256	(keiner gebräuchlich)	0.0.1.3	IP-Adressen (IPv4)

4.1.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

4.1.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
<code>&</code>	Und	Bits gezielt löschen
<code> </code>	Oder	Bits gezielt setzen
<code>^</code>	Exklusiv-Oder	Bits gezielt invertieren
<code>~</code>	Nicht	Alle Bits invertieren
<code><<</code>	Verschiebung nach links	Maske generieren
<code>>></code>	Verschiebung nach rechts	Bits isolieren

Numerierung der Bits: von rechts ab 0

Bit Nr. 3 auf 1 setzen: `a |= 1 << 3;`

Bit Nr. 4 auf 0 setzen: `a &= ~(1 << 4);`

Bit Nr. 0 invertieren: `a ^= 1 << 0;`

Abfrage, ob Bit Nr. 1 gesetzt ist: `if (a & (1 << 1))`

4.1.2 Bit-Operationen in C

3 & 6

„bitweises
Und“

$$\begin{array}{r} 0011 \\ \& 0110 \\ \hline 0010 \end{array}$$

= 2

3 && 6 = 1 „logisches Und“

3 & 4

↑
„true“

$$\begin{array}{r} 0011 \\ \& 0100 \\ \hline 0000 \end{array}$$

= 0

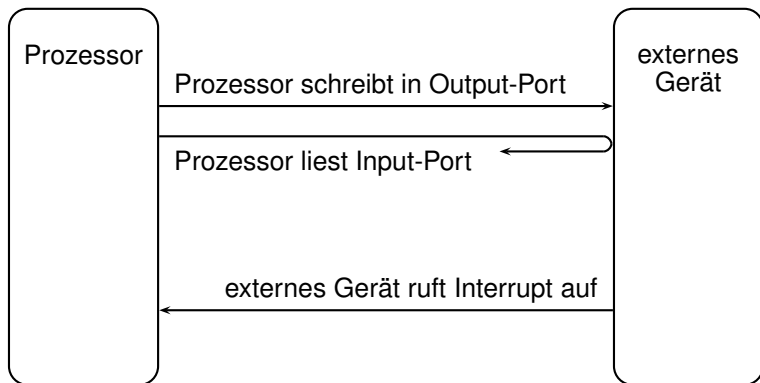
↑
„false“

3 && 4 = 1

4.2 I/O-Ports

4.3 Interrupts

Kommunikation mit externen Geräten



4.2 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

4.2 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: Druckknopfampel

4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

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

```
...
```

```
DDRC = 0xfd;
```

```
while (PINC & 0x02 == 0)
```

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

Eingang verbunden mit 5 V \longrightarrow Bit ist 1.

Eingang verbunden mit 0 V \longrightarrow Bit ist 0.

Eingang verbunden mit gar nichts \longrightarrow ???

4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

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

```
...
```

```
DDRC = 0xfd;
```

```
while (PINC & 0x02 == 0)
```

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

Eingang verbunden mit 5 V \longrightarrow Bit ist 1.

Eingang verbunden mit 0 V \longrightarrow Bit ist 0.

Eingang verbunden mit gar nichts \longrightarrow undefiniert!

4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

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

```
...
```

```
DDRC = 0xfd;
```

```
while (PINC & 0x02 == 0)
```

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

Eingang verbunden mit 5 V → Bit ist 1.

Eingang verbunden mit 0 V → Bit ist 0.

Eingang verbunden mit gar nichts → undefiniert!

→ Pull-Up- und Pull-Down-Widerstände

4.2 I/O-Ports

Aus Input-Port lesen = Sensoren abfragen

Beispiel: Taster

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

```
...
```

```
DDRC = 0xfd;
```

```
while (PINC & 0x02 == 0)
```

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

Eingang verbunden mit 5 V → Bit ist 1.

Eingang verbunden mit 0 V → Bit ist 0.

Eingang verbunden mit gar nichts → undefiniert!

→ Pull-Up- und Pull-Down-Widerstände

Internen Pull-Up-Widerstand einschalten: `PORTC |= 0x02`

4.3 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

4.3 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

4.4 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;
```

```
}
```


4.4 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



Angewandte Informatik

Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
 - 4.1 Bit-Operationen
 - 4.2 I/O-Ports
 - 4.3 Interrupts
 - 4.4 volatile-Variable
 - 4.5 Byte-Reihenfolge – Endianness
 - 4.6 Speicherausrichtung – Alignment
- 5 Algorithmen**
 - 5.1 Differentialgleichungen
 - ...

...

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Speicherzellen:

04	03
----	----

 Big-Endian „großes Ende zuerst“
für Menschen leichter lesbar

03	04
----	----

 Little-Endian „kleines Ende zuerst“
bei Additionen effizienter

→ Geschmackssache

... **außer bei Datenaustausch!**

4.5 Byte-Reihenfolge – Endianness

4.5.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

Welche Bits liegen wo?

—→ Geschmackssache

... **außer bei Datenaustausch!**

- Dateiformate
- Datenübertragung

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian
- XBM-Dateien: Little-Endian

4.5 Byte-Reihenfolge – Endianness

4.5.2 Dateiformate

Audio-Formate: Reihenfolge der Bytes in 16- und 32-Bit-Zahlen

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Grafik-Formate: Reihenfolge der Bits in den Bytes

- PBM-Dateien: Big-Endian, MSB first
- XBM-Dateien: Little-Endian, LSB first

MSB/LSB = most/least significant bit

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides

4.5 Byte-Reihenfolge – Endianness

4.5.3 Datenübertragung

- RS-232 (serielle Schnittstelle): LSB first
- I²C: MSB first
- USB: beides
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

Angewandte Informatik

Hardwarenahe Programmierung

<https://gitlab.cvh-server.de/pgerwinski/hp.git>

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
- 4 Hardwarenahe Programmierung**
 - 4.1 Bit-Operationen
 - 4.2 I/O-Ports
 - 4.3 Interrupts
 - 4.4 volatile-Variable
 - 4.5 Byte-Reihenfolge – Endianness
 - 4.6 Speicherausrichtung – Alignment
- 5 Algorithmen**
 - 5.1 Differentialgleichungen
 - ...

...