

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

10. Dezember 2018

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 Programmierung von Mikrocontrollern

4.3 I/O-Ports

4.4 Interrupts

4.5 volatile-Variable

4.6 Byte-Reihenfolge – Endianness

4.7 Binärdarstellung negativer Zahlen

4.8 Speicherausrichtung – Alignment

5 Algorithmen

5.1 Differentialgleichungen

...

...

4.1.1 Zahlensysteme

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

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

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

C-Datentypen für Bit-Operationen:

#include <stdint.h>

	8 Bit	16 Bit	32 Bit	64 Bit
mit Vorzeichen	int8_t	int16_t	int32_t	int64_t
ohne Vorzeichen	uint8_t	uint16_t	uint32_t	uint64_t

Ausgabe:

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

...

uint64_t x = 42;

printf ("Die_Antwort_lautet:_% " PRIu64 "\n", x);

4.2 Programmierung von Mikrocontrollern

Cross-Entwicklungswerkzeuge


- laufen auf „normalem“ Computer, erzeugen Software für anderen Computer oder Mikrocontroller

- Cross-Compiler, -Assembler und -Linker

```
avr-gcc -Wall -Os -mmcu=atmega328p blink.c -o blink.elf
```

- Werkzeuge zum Herunterladen¹ auf den Mikrocontroller

```
avr-objcopy -O ihex blink.elf blink.hex  
avrdude -P /dev/ttyACM0 -c arduino -p m328p \  
-U flash:w:blink.hex
```



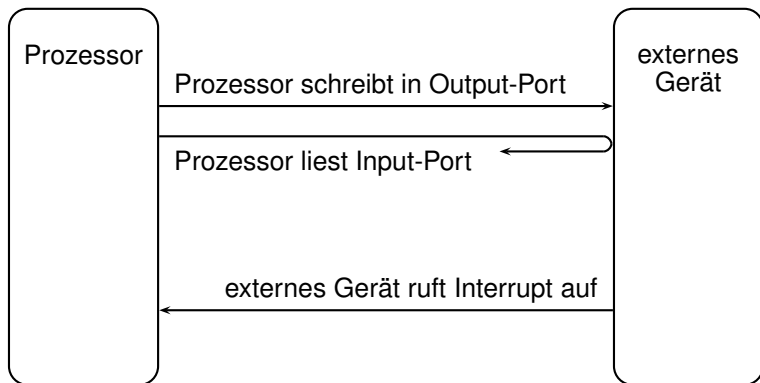
„Es geht in der nächsten Zeile weiter.“

¹So ist die Sprechweise.

4.3 I/O-Ports

4.4 Interrupts

Kommunikation mit externen Geräten



4.3 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

4.3 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

4.4 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>
```

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

„Dies ist ein Interrupt-Handler.“

Interrupt-Vektor darauf zeigen lassen

Herstellerspezifisch!

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

Details: siehe Datenblatt und Schaltplan

4.4 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

4.5 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;
```

```
}
```

4.5 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



4.5 volatile-Variable

Was ist eigentlich PORTD?

4.5 volatile-Variable

Was ist eigentlich PORTD?

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

4.5 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

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


4.5 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

4.5 volatile-Variable

Was ist eigentlich PORTD?

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

```
PORTD = 0x01;
```

→
$$\underbrace{*(\text{volatile uint8_t } *)}_{\text{Umwandlung in Zeiger auf volatile uint8_t}} \underbrace{((0x0B) + 0x20)}_{\text{Zahl: 0x2B}} = 0x01;$$

4.5 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

4.5 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

4.5 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.

4.6 Byte-Reihenfolge – Endianness

4.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.
Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen
Welche Bits liegen wo?

4.6 Byte-Reihenfolge – Endianness

4.6.1 Konzept

Eine Zahl geht über mehrere Speicherzellen.

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

Welche Bits liegen wo?

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

4.6 Byte-Reihenfolge – Endianness

4.6.1 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“

4.6 Byte-Reihenfolge – Endianness

4.6.1 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

4.6 Byte-Reihenfolge – Endianness

4.6.1 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“

4.6 Byte-Reihenfolge – Endianness

4.6.1 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

4.6 Byte-Reihenfolge – Endianness

4.6.1 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

4.6 Byte-Reihenfolge – Endianness

4.6.1 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!**

4.6 Byte-Reihenfolge – Endianness

4.6.1 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

4.6 Byte-Reihenfolge – Endianness

4.6.2 Dateiformate

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

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

4.6 Byte-Reihenfolge – Endianness

4.6.2 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

4.6 Byte-Reihenfolge – Endianness

4.6.2 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

4.6 Byte-Reihenfolge – Endianness

4.6.2 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

4.6 Byte-Reihenfolge – Endianness

4.6.3 Datenübertragung

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

4.6 Byte-Reihenfolge – Endianness

4.6.3 Datenübertragung

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

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

8-Bit-Zahlen ohne Vorzeichen: `uint8_t`

→ Zahlenwerte von `0x00` bis `0xff` = 0 bis 255

→ $255 + 1 = 0$

8-Bit-Zahlen mit Vorzeichen: `int8_t`

`0xff` = 255 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $127 + 1 = -128$

4.7 Binärdarstellung negativer Zahlen

Speicher ist begrenzt!

→ feste Anzahl von Bits

16-Bit-Zahlen ohne Vorzeichen: `uint16_t`

→ Zahlenwerte von `0x0000` bis `0xffff` = 0 bis 65535

→ $65535 + 1 = 0$

`uint8_t`

0 bis 255

$255 + 1 = 0$

16-Bit-Zahlen mit Vorzeichen: `int16_t`

`0xffff` = 65535 ist die „natürliche“ Schreibweise für -1 .

→ Zweierkomplement

`int8_t`

`0xff` = 255 = -1

Oberstes Bit = 1: negativ

Oberstes Bit = 0: positiv

→ $32767 + 1 = -32768$

Literatur: <http://xkcd.com/571/>

4.7 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

4.7 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

4.7 Binärdarstellung negativer Zahlen

Frage: Für welche Zahl steht der Speicherinhalt

a3	90
----	----

 (hexadezimal)?

Antwort: Das kommt darauf an. ;—)

als <code>int8_t</code> :	−93	(nur unteres Byte, Little-Endian)
als <code>uint8_t</code> :	163	(nur unteres Byte, Little-Endian)
als <code>int16_t</code> :	−28509	
als <code>uint16_t</code> :	37027	
<code>int32_t</code> oder größer:	37027	(zusätzliche Bytes mit Nullen aufgefüllt)

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;
```

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;    uint8_t a;  
uint16_t b;        uint8_t c;  
uint8_t c;         uint16_t b;
```

4.8 Speicherausrichtung – Alignment

```
#include <stdint.h>
```

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Speicheradresse durch 2 teilbar – „16-Bit-Alignment“

- 2-Byte-Operation: effizienter
- ... oder sogar nur dann erlaubt

→ Compiler optimiert Speicherausrichtung

```
uint8_t a;  
uint8_t dummy;  
uint16_t b;  
uint8_t c;  
  
uint8_t a;  
uint8_t c;  
uint16_t b;
```

Fazit:

- **Adressen von Variablen sind systemabhängig**
- Bei Definition von Datenformaten Alignment beachten → effizienter

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 Programmierung von Mikrocontrollern

4.3 I/O-Ports

4.4 Interrupts

4.5 volatile-Variable

4.6 Byte-Reihenfolge – Endianness

4.7 Binärdarstellung negativer Zahlen

4.8 Speicherausrichtung – Alignment

5 Algorithmen

5.1 Differentialgleichungen

...

...