

# Angewandte Informatik Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

20. November 2017

# Angewandte Informatik

## Hardwarenahe Programmierung

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

### 1 Einführung

### 2 Einführung in C

...

#### 2.14 Parameter des Hauptprogramms

#### 2.15 String-Operationen

### 3 Bibliotheken

#### 3.1 Der Präprozessor

#### 3.2 Bibliotheken einbinden

#### 3.3 Bibliotheken verwenden

#### 3.4 Projekt organisieren: make

### 4 Hardwarenahe Programmierung

### 5 Algorithmen

#### 5.1 Differentialgleichungen

...

...

## 2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; i < argc; i++)
        printf ("argv[%d]=\n", i, argv[i]);
    return 0;
}
```

## 2.14 Parameter des Hauptprogramms

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; *argv; i++, argv++)
        printf ("argv[%d]=\n%s\n", i, *argv);
    return 0;
}
```

## 2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello,_world!\n";
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    printf ("%s\n", hello + 7);
```

```
    printf ("%zd\n", strlen (hello + 7));
```

```
    hello[5] = 0;
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    return 0;
```

```
}
```

## 2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char *anton = "Anton";
```

```
    char *zacharias = "Zacharias";
```

```
    printf ("%d\n", strcmp (anton, zacharias));
```

```
    printf ("%d\n", strcmp (zacharias, anton));
```

```
    printf ("%d\n", strcmp (anton, anton));
```

```
    char buffer[100] = "Huber_";
```

```
    strcat (buffer, anton);
```

```
    printf ("%s\n", buffer);
```

```
    return 0;
```

```
}
```

## 2.15 String-Operationen

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    sprintf (buffer, "Die_Antwort_lautet:%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

## 2 Einführung in C

Sprachelemente weitgehend komplett

Es fehlen:

- Ergänzungen (z. B. ternärer Operator, **union**, **unsigned**, **volatile**)
- Bibliotheksfunktionen (z. B. **malloc()**)

—> werden eingeführt, wenn wir sie brauchen

- Konzepte (z. B. rekursive Datenstrukturen, Klassen selbst bauen)

—> werden eingeführt, wenn wir sie brauchen, oder:

—> Literatur

(z. B. Wikibooks: C-Programmierung,  
Dokumentation zu Compiler und Bibliotheken)

- Praxiserfahrung

—> Übung und Praktikum: nur Einstieg

—> selbständig arbeiten



# 3 Bibliotheken

## 3.1 Der Präprozessor

**#include**: Text einbinden

- **#include <stdio.h>**: Standard-Verzeichnisse – Standard-Header
- **#include "answer.h"**: auch aktuelles Verzeichnis – eigene Header

**#define VIER 4**: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:  
**#define VIER (2 + 2)**
- Konvention: Großbuchstaben

## 3 Bibliotheken

### 3.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

```
extern int answer (void);
```

```
extern int printf (__const char *__restrict __format, ...);
```

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- Zusammenfügen zu ausführbarem Programm durch den *Linker*
- vorcompilierte Bibliothek: `-lfoo`  
= Datei `libfoo.a` in Standard-Verzeichnis

### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

- Include-Dateien:

```
#include <GL/gl.h>
```

```
#include <GL/glu.h>
```

```
#include <GL/glut.h>
```

- Compiler-Aufruf:

```
gcc -Wall -O cube-1.c opengl-magic.c -lGL -lGLU -lglut \  
-o cube-1
```

### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void draw (void)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

```
glutDisplayFunc (draw);
```

→ OpenGL ruft immer dann, wenn es etwas zu zeichnen gibt, die Funktion `draw` auf.

### 3.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void key_handler (unsigned char key, int x, int y)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

gedrückte Taste

Mausposition

```
glutKeyboardFunc (key_handler);
```

→ OpenGL ruft immer dann, wenn eine Taste gedrückt wurde, die Funktion `key_handler` auf.

## 3.4 Projekt organisieren: make

- Regeln
- Makros

## 3.4 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c answer.h  
gcc -Wall -O answer.c -c
```

```
philosophy.o: philosophy.c answer.h  
gcc -Wall -O philosophy.c -c
```

- Makros

## 3.4 Projekt organisieren: make

- Regeln
- Makros

```
TARGET = philosophy  
OBJECTS = philosophy.o answer.o  
HEADERS = answer.h  
CFLAGS = -Wall -O
```

```
$(TARGET): $(OBJECTS)  
    gcc $(OBJECTS) -o $(TARGET)
```

```
answer.o: answer.c $(HEADERS)  
    gcc $(CFLAGS) answer.c -c
```

```
philosophy.o: philosophy.c $(HEADERS)  
    gcc $(CFLAGS) philosophy.c -c
```

```
clean:  
    rm -f $(OBJECTS) $(TARGET)
```



## 3.4 Projekt organisieren: make

- explizite und implizite Regeln

TARGET = philosophy

OBJECTS = philosophy.o answer.o

HEADERS = answer.h

CFLAGS = -Wall -O

\$(TARGET): \$(OBJECTS)

gcc \$(OBJECTS) -o \$(TARGET)

%.o: %.c \$(HEADERS)

gcc \$(CFLAGS) \$< -c

clean:

rm -f \$(OBJECTS) \$(TARGET)

- Makros

## 3.4 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

# Angewandte Informatik

## Hardwarenahe Programmierung

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

### 1 Einführung

### 2 Einführung in C

### 3 Bibliotheken

#### 3.1 Der Präprozessor

#### 3.2 Bibliotheken einbinden

#### 3.3 Bibliotheken verwenden

#### 3.4 Projekt organisieren: make

### 4 Hardwarenahe Programmierung

#### 4.1 Bit-Operationen

#### 4.2 I/O-Ports

#### 4.3 Interrupts

...

### 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>&amp;</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>&lt;&lt;</code>	Verschiebung nach links	Maske generieren
<code>&gt;&gt;</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>

...

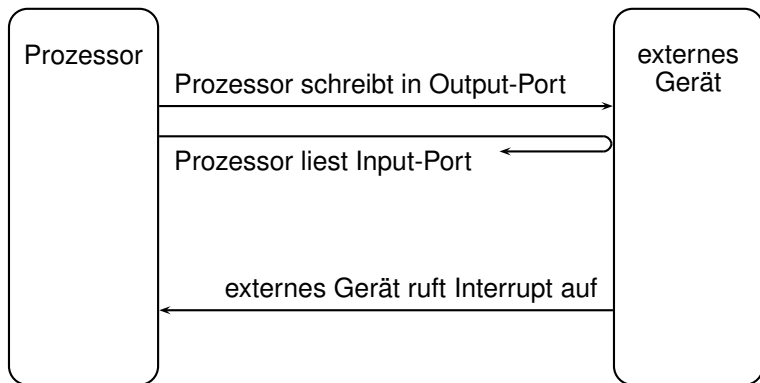
uint64\_t x = 42;

printf ("Die\_Antwort\_lautet:\_%" PRIu64 "\n", x);

## 4.2 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!

*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!

*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!

*Details: siehe Datenblatt und Schaltplan*

Praktikumsaufgabe: Druckknopfampel

# Angewandte Informatik

## Hardwarenahe Programmierung

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

### 1 Einführung

### 2 Einführung in C

### 3 Bibliotheken

#### 3.1 Der Präprozessor

#### 3.2 Bibliotheken einbinden

#### 3.3 Bibliotheken verwenden

#### 3.4 Projekt organisieren: make

### 4 Hardwarenahe Programmierung

#### 4.1 Bit-Operationen

#### 4.2 I/O-Ports

#### 4.3 Interrupts

...

### 5 Algorithmen

#### 5.1 Differentialgleichungen

...

...