

# Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

26. November 2018

# 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

...

...

## 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: GTK+)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk+-3.0
-pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfBuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -pthread -I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -I/usr/include/gtk-3.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz -I/usr/include/pango-1.0 -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lgtk-3 -lgdk-3 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0 -o hello-gtk
```

## 3.3 Bibliothek verwenden (Beispiel: GTK+)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk+-3.0` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk+-3.0) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-3.0`

Bibliotheken:

u. a. `-lgtk-3 -lcairo`

### 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean draw (GtkWidget *widget, cairo_t *c, gpointer data)
{
    /* Zeichenbefehle */
    ...
    return FALSE;
}
```

repräsentiert den  
Bildschirm, auf den  
gezeichnet werden soll

optionale Zusatzinformationen  
für draw(), typischerweise  
ein Zeiger auf ein struct

```
g_signal_connect (drawing_area, "draw", G_CALLBACK (draw), NULL);
```

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




## 3.3 Bibliothek verwenden (Beispiel: GTK+)

Selbst geschriebene Funktion übergeben: *Callback*

```
gboolean timer (GtkWidget *widget)
{
    /* Rechenbefehle */
    ...
```

Dieser Bereich soll  
neu gezeichnet werden.



```
    gtk_widget_queue_draw_area (widget, 0, 0, WIDTH, HEIGHT);
    g_timeout_add (50, (GSourceFunc) timer, widget);
    return FALSE;
}
```

In weiteren 50 Millisekunden soll  
die Funktion erneut aufgerufen werden.

Explizite Typumwandlung  
eines Zeigers (später)



```
g_timeout_add (50, (GSourceFunc) timer, drawing_area);
```

→ GTK+ ruft nach 50 Millisekunden die Funktion `timer` 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

# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x(t) = \int v_x(t) dt = \int v_{0x} dt = x_0 + v_{0x} \cdot t$$

$$y'(t) = v_y(t) \quad \Rightarrow \quad y(t) = \int v_y(t) dt = \int v_{0y} - g \cdot t dt = y_0 + v_{0y} \cdot t - \frac{1}{2}gt^2$$

$$v'_x(t) = 0 \quad v_x(t) = \int 0 dt = v_{0x}$$

$$v'_y(t) = -g \quad v_y(t) = \int -g dt = v_{0y} - g \cdot t$$

# 5 Algorithmen

## 5.1 Differentialgleichungen

### Beispiel 1: Gleichmäßig beschleunigte Bewegung

$$x'(t) = v_x(t) \quad x += v_x * dt;$$

$$y'(t) = v_y(t) \quad y += v_y * dt;$$

$\Rightarrow$

Siehe: [gtk-13.c](#)

$$v'_x(t) = 0 \quad v_x += 0 * dt;$$

$$v'_y(t) = -g \quad v_y += -g * dt;$$



# 5 Algorithmen

## 5.1 Differentialgleichungen

**Beispiel 1: Gleichmäßig beschleunigte Bewegung**

**Beispiel 2: Mathematisches Pendel**

$$\varphi'(t) = \omega(t)$$

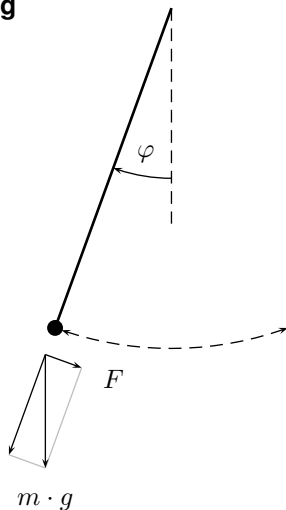
$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch):  
Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch):  
Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

**Beispiel 3: Weltraum-Simulation**

Praktikumsaufgabe



# 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)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

### 4.1.1 Zahlensysteme

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

- 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>&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))`

# 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

...

...