

Hardwarenahe Programmierung

Prof. Dr. rer. nat. Peter Gerwinski

28. November 2024

Hardwarenahe Programmierung

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

- 1 Einführung**
- 2 Einführung in C**
- 3 Bibliotheken**
 - 3.1 Der Präprozessor
 - 3.2 Bibliotheken einbinden
 - 3.3 Bibliotheken verwenden
 - 3.4 Callbacks
 - 3.5 Projekt organisieren: make
- 4 Hardwarenahe Programmierung**
 - 4.1 Bit-Operationen
 - 4.2 I/O-Ports
 - 4.3 Interrupts
 - 4.4 volatile-Variable
 - ...
- 5 Algorithmen**
- 6 Objektorientierte Programmierung**
- 7 Datenstrukturen**

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 SIX 6: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define SIX (1 + 5)
- 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>

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`

- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß:

```
$ pkg-config --cflags --libs gtk4
-I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/
include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/i
nclude -I/usr/include/harfbuzz -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/libmount -I/usr/
include/blkid -I/usr/include/fribidi -I/usr/include/cai
ro -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.
0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo
-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-go
bject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l
glib-2.0
```

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c -I/usr/include/gtk-4.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/graphene-1.0 -I/usr/lib/x86_64-linux-gnu/graphene-1.0/include -mfpmath=sse -msse -msse2 -pthread -lgtk-4 -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo -lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -l glib-2.0 -o hello-gtk
```


3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

→ Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

Optionen:

u. a. viele Include-Verzeichnisse:

`-I/usr/include/gtk-4.0`

Bibliotheken:

u. a. `-lgtk-4`

3.3 Bibliothek verwenden (Beispiel: GTK)

- `#include <gtk/gtk.h>`
- Mit `pkg-config --cflags --libs gtk4` erfährt man, welche Optionen und Bibliotheken man an `gcc` übergeben muß.

—> Compiler-Aufruf:

```
$ gcc -Wall -O hello-gtk.c $(pkg-config --cflags --libs  
    gtk4) -o hello-gtk
```

- Auf manchen Plattformen kommt es auf die Reihenfolge an:

```
$ gcc -Wall -O $(pkg-config --cflags gtk4) \  
    hello-gtk.c $(pkg-config --libs gtk4) \  
    -o hello-gtk
```

(Backslash = „Es geht in der nächsten Zeile weiter.“)

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

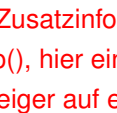
Selbst geschriebene Funktion übergeben: *Callback*

```
static void hello (GtkWidget *this, gpointer user_data)
{
    char *world = user_data;
    printf ("Hello, %s!\n", world);
}
```

...

```
g_signal_connect (button, "clicked", G_CALLBACK (hello), "world");
```

optionale Zusatzinformationen
für hello(), hier ein String
oft ein Zeiger auf ein struct



→ GTK ruft immer dann, wenn der Button betätigt wurde,
die Funktion `hello` auf.

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */
```

```
    ...
```

```
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

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

3.4 Callbacks

Selbst geschriebene Funktion übergeben: *Callback*

```
static void draw (GtkDrawingArea *drawing_area, cairo_t *c,  
                  int width, int height, gpointer user_data)
```

```
{  
    /* Zeichenbefehle */  
    ...  
}
```

```
...
```

```
gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (drawing_area),  
                                draw, NULL, NULL);
```

repräsentiert den
Bildschirm, auf den
gezeichnet werden soll

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

3.5 Projekt organisieren: make

- Regeln
- Makros

3.5 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

3.5 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)

3.5 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

3.5 Projekt organisieren: make

- explizite und implizite Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

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	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>

...

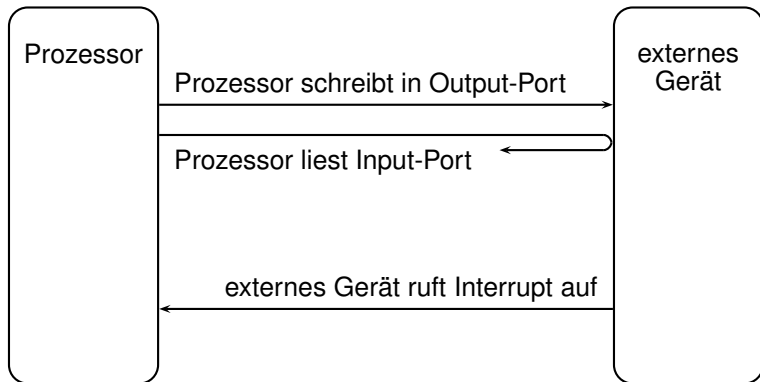
uint64_t x = 42;

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

4.2 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