Übungen zu Grundlagen der systemnahen Programmierung in C (GSPiC) im Sommersemester 2018

2018-05-11

Bernhard Heinloth

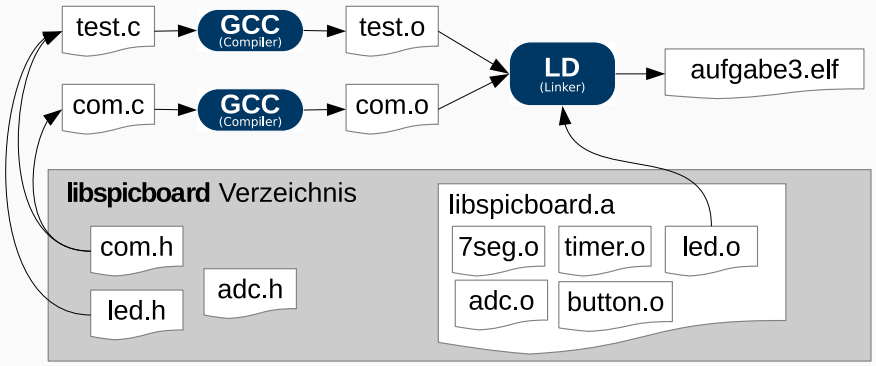
Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme und Betriebssysteme

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Module

Ablauf vom Quellcode zum laufenden Programm



1. Präprozessor
2. Compiler
3. Linker
4. Programmer/Flasher

Schnittstellenbeschreibung

- Header Dateien enthalten die Schnittstelle eines Moduls
  - Funktionsdeklarationen
  - Präprozessormakros
  - Typdefinitionen
- Header Dateien können u.U. mehrmals eingebunden werden
  - `led.h` bindet `avr/io.h` ein
  - `button.h` bindet `avr/io.h` ein
  - ⇝ Funktionen aus `avr/io.h` mehrmals deklariert
- Mehrfachinkludierung/Zyklen vermeiden ⇝ **Include-Guards**
  - Definition und Abfrage eines Präprozessormakros
  - Konvention: Makro hat den Namen der .h-Datei, '.' ersetzt durch '_'
  - z.B. für `button.h` ⇝ `BUTTON_H`
  - Inhalt nur einbinden, wenn das Makro noch nicht definiert ist
- Vorsicht: flacher Namensraum ⇝ möglichst eindeutige Namen

Schnittstellenbeschreibung

- Erstellen einer .h-Datei (Konvention: gleicher Name wie .c-Datei)

```c
#ifndef COM_H
#define COM_H
/* fixed-width Datentypen einbinden (im Header verwendet) */
#include <stdint.h>

/* Datentypen */
typedef enum {
    ERROR_NO_STOP_BIT, ERROR_PARITY,
    ERROR_BUFFER_FULL, ERROR_INVALID_POINTER
} COM_ERROR_STATUS;

/* Funktionen */
void sb_com_sendByte(uint8_t data);
...
#endif //COM_H
```


Initialisierung eines Moduls

- Module müssen Initialisierung durchführen
  - zum Beispiel Portkonfiguration
  - **Java:** mit Klassenkonstruktoren möglich
  - **C:** kennt kein solches Konzept
- *Workaround:* Modul muss bei erstem Aufruf einer seiner Funktionen ggf. die Initialisierung durchführen
  - muss sich merken, ob die Initialisierung schon erfolgt ist
  - Mehrfachinitialisierung vermeiden
- Anlegen einer Init-Variable
  - Aufruf der Init-Funktion bei jedem Funktionsaufruf
  - Init-Variable anfangs 0
  - Nach der Initialisierung auf 1 setzen

Initialisierung eines Moduls

- `initDone` ist initial `0`
- wird nach der Initialisierung auf 1 gesetzt

⇝ Initialisierung wird nur ein mal durchgeführt

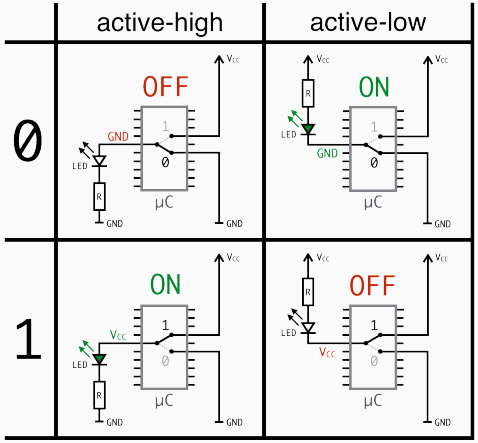
```c
static void init(void){
    static uint8_t initDone = 0;
    if (initDone == 0) {
      initDone = 1;
      ...
    }
}

void mod_func(void) {
    init();
    ...
```


Ein- & Ausgabe über Pins

Ausgang: active-high & active-low

Ausgang je nach Beschaltung:

active-high high-Pegel (logisch 1; V_{cc} am Pin) → LED leuchtet
active-low low-Pegel (logisch 0; *GND* am Pin) → LED leuchtet

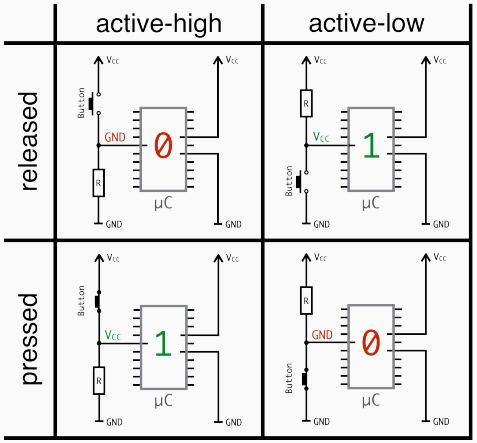


Eingang: active-high & active-low

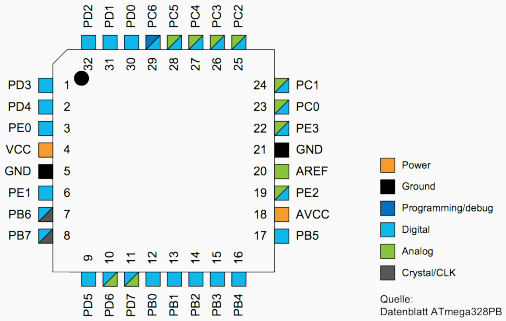
Eingang je nach Beschaltung:

active-high Button gedrückt → high-Pegel (logisch 1; V_{cc} am Pin)
active-low Button gedrückt → low-Pegel (logisch 0; *GND* am Pin)

interner pull-up-Widerstand (im ATmega328PB) konfigurierbar



Konfiguration der Pins



- jeder I/O-Port des AVR-µC wird durch drei 8-bit Register gesteuert:
  - Datenrichtungsregister (DDRx = data direction register)
  - Datenregister (PORTx = port output register)
  - Port Eingabe Register (PINx = port input register, nur-lesbar)
- jedem Anschluss-Pin ist ein Bit in jedem der 3 Register zugeordnet

I/O-Port-Register

DDRx hier konfiguriert man Pin i von Port x als Ein- oder Ausgang
- `Bit i = 1` → Pin i als Ausgang verwenden
- `Bit i = 0` → Pin i als Eingang verwenden

PORTx Auswirkung **abhängig von DDRx**:
- ist Pin i **als Ausgang konfiguriert**, so steuert Bit i im PORTx Register ob am Pin i ein high- oder ein low-Pegel erzeugt werden soll
  - `Bit i = 1` → high-Pegel an Pin i
  - `Bit i = 0` → low-Pegel an Pin i
- ist Pin i **als Eingang konfiguriert**, so kann man einen internen pull-up-Widerstand aktivieren
  - `Bit i = 1` → pull-up-Widerstand an Pin i (Pegel wird auf high gezogen)
  - `Bit i = 0` → Pin i als tri-state konfiguriert

PINx Bit i gibt aktuellen Wert des Pin i von Port x an (nur lesbar)

Beispiel: Initialisierung eines Ports

- Pin 3 von Port C (PC3) als Ausgang konfigurieren und PC3 auf Vcc schalten:

```
01 DDRC |= (1 << PC3); /* =0x08; PC3 als Ausgang nutzen... */
02 PORTC |= (1 << PC3); /* ...und auf 1 (=high) setzen */
```

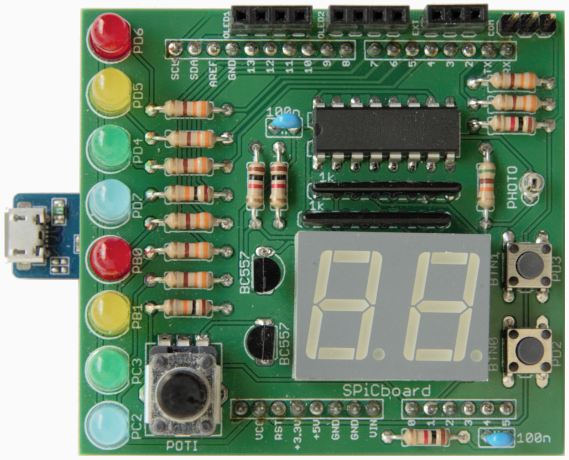
- Pin 2 von Port D (PD2) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
01 DDRD &= ~(1 << PD2); /* PD2 als Eingang nutzen... */
02 PORTD |= (1 << PD2); /* pull-up-Widerstand aktivieren */
03 if((PIND & (1 << PD2)) == 0){ /* den Zustand auslesen */
04   /* ein low Pegel liegt an, der Taster ist gedrückt */
05 }
```

- Die Initialisierung der Hardware wird in der Regel einmalig zum Programmstart durchgeführt

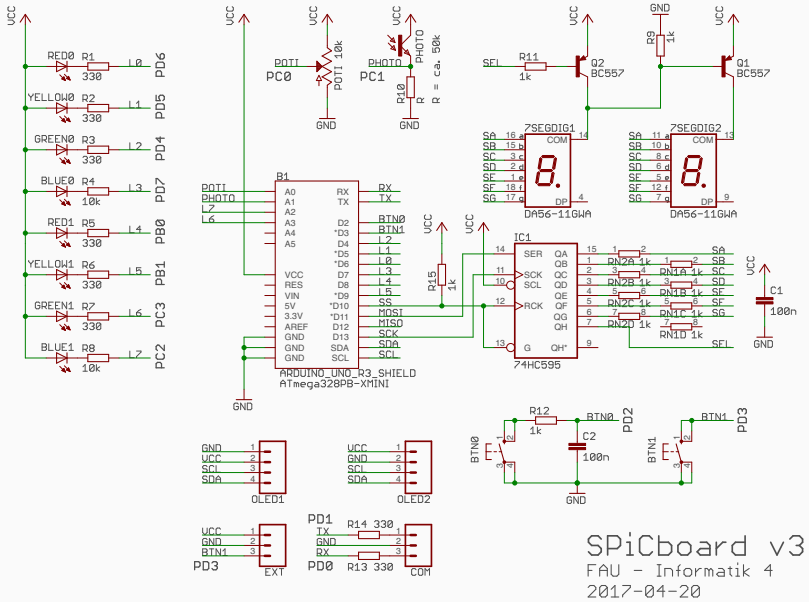
Aufgabe: LED-Modul

LED-Modul – Übersicht



- LED 0 (RED0) ⇒ PD6 ⇒ Port D, Pin 6 ⇒ Bit 6 in PORTD und DDRD
- ...
- LED 7 (BLUE1) ⇒ PC2 ⇒ Port C, Pin 2 ⇒ Bit 2 in PORTC und DDRC

SPiCboard Schaltplan

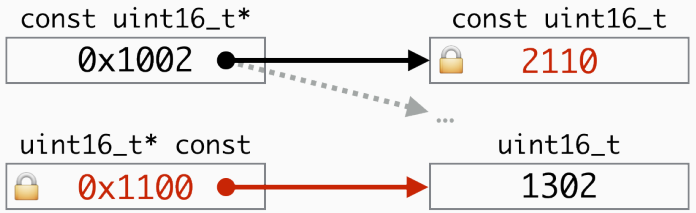


LED-Modul – Aufgabe

- LED-Modul der SPiCboard-Bibliothek selbst implementieren
  - Gleiches Verhalten wie das Original
  - Beschreibung: `http://www4.cs.fau.de/Lehre/SS18/V_SPIC/Uebung/doc`
- Testen des Moduls
  - Eigenes Modul mit einem Testprogramm (`test.c`) linken
  - Andere Teile der Bibliothek können für den Test benutzt werden
- LEDs des SPiCboards
  - Anschlüsse und Namen der einzelnen LEDs können dem Übersichtsbildchen entnommen werden
  - Alle LEDs sind **active-low**, d.h. leuchten wenn ein low-Pegel auf dem Pin angelegt wird
  - PD6 = Port D, Pin 6

Exkurs: const uint8_t* vs. uint8_t* const

- `const uint8_t*`
  - ein Pointer auf einen `uint8_t`-**Wert**, der konstant ist
  - Wert nicht über den Pointer veränderbar
- `uint8_t* const`
  - ein **konstanter Pointer** auf einen (beliebigen) `uint8_t`-Wert
  - Pointer darf nicht mehr auf eine andere Speicheradresse zeigen



Port- und Pin-Array

- Port und Pin Definitionen (in `avr/io.h`)

```
01 #define PORTD (* (volatile uint8_t*)0x2B)
02 ...
03 #define PD0     0
04 ...
```

- Adressoperator: &
- Dereferenzierungsoperator: *
- Port Array:

```
01 static volatile uint8_t * const ports[] = { &PORTD,
02                                             ...,
03                                             &PORTC };
```

- Pin Array:

```
01 static uint8_t const pins[] = { PD6, ..., PC2 };
```


Elegante Variante

- Port und Pin in einer Struktur zusammenfassen:

```
01 struct led {
02     volatile uint8_t * const ddr;
03     volatile uint8_t * const port;
04     const uint8_t pin;
05 };
```

- Zuweisung:

```
01 static const struct led led_red = { &DDRD,
02                                     &PORTD,
03                                     PD6
04 };
```

- Array ebenfalls möglich:

```
01 static const struct led led_array[] = {
02     {&DDRD, &PORTD, PD6},
03     ...,
04     {&DDRC, &PORTC, PC2}
05 };
```

AVR-Studio Projekteinstellungen

- Projekt wie gehabt anlegen
  - Initiale Quelldatei: `test.c`
  - Dann weitere Quelldatei `led.c` hinzufügen
- Wenn nun übersetzt wird, werden die Funktionen aus dem eigenen LED-Modul verwendet
- Andere Teile der Bibliothek werden nach Bedarf hinzugebunden
- Temporäres Deaktivieren zum Test der Originalfunktiononen:

```
01 #if 0
02   ....
03 #endif
```

- Sieht der Compiler diese “Kommentare”?
- Wie kann der Code wieder einkommentiert werden?

Testen des Moduls

```
01 void main(void){
02   ...
03   // 1.) Testen bei korrekter LED-ID
04   int8_t result = sb_led_on(RED0);
05   if(result != 0){
06     // Test fehlgeschlagen
07     // Ausgabe z.B. auf 7-Segment-Anzeige
08   }
09   // Einige Sekunden warten
10
11   // 2.) Testen bei ungueltiger LED-ID
12   ...
13 }
```

- Schnittstellenbeschreibung genau beachten (inkl. Rückgabewerte)
- Testen **aller möglichen Rückgabewerte**
- Fehler wenn Rückgabewert nicht der Spezifikation entspricht