

Systemnahe Programmierung in C (SPiC)

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2011

http://ww4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC



- [1] *ATmega32 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. 8155-AVR-07/09. Atmel Corporation. July 2009. URL: http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_GSPIC/Uebungen/doc/mega32.pdf.
 - [2] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop, et al. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter /proj/i4gspic/pub). Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: <http://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1>.
 - [3] Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1978.
 - [4] Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960.
- [GDI] Elmar Nöth, Peter Wilke, and Stefan Steidl. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2010 (jährlich). URL: <http://www5.informatik.uni-erlangen.de/lectures/ws-1011/grundlagen-der-informatik-gdi/folien/>.



- [5] Dennis MacAlistair Ritchie and Ken Thompson. "The Unix Time-Sharing System". In: *Communications of the ACM* 17.7 (July 1974), pp. 365–370. DOI: 10.1145/361011.361061.
- [GDI-Ü] Stefan Steidl, Marcus Prümmer, and Markus Mayer. *Grundlagen der Informatik*. Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2010 (jährlich). URL: <http://www5.informatik.uni-erlangen.de/lectures/ws-1011/grundlagen-der-informatik-gdi/uebung/>.
- [6] David Tennenhouse. "Proactive Computing". In: *Communications of the ACM* (May 2000), pp. 43–45.
- [7] Jim Turley. "The Two Percent Solution". In: *embedded.com* (Dec. 2002). <http://www.embedded.com/story/0EG20021217S0039>, visited 2011-04-08.



Veranstaltungsüberblick

Teil A: Konzept und Organisation

1 Einführung

2 Organisation

Teil B: Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Teil C: Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

Teil D: Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

21 Prozesse II



Systemnahe Programmierung in C (SPiC)

Teil A Konzept und Organisation

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2011

http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC



Überblick: Teil A Konzept und Organisation

1 Einführung

- 1.1 Ziele der Lehrveranstaltung
- 1.2 Warum μ -Controller?
- 1.3 Warum C?
- 1.4 Literatur

2 Organisation

- 2.1 Vorlesung
- 2.2 Übung
- 2.3 Lötabend
- 2.4 Prüfung
- 2.5 Semesterüberblick

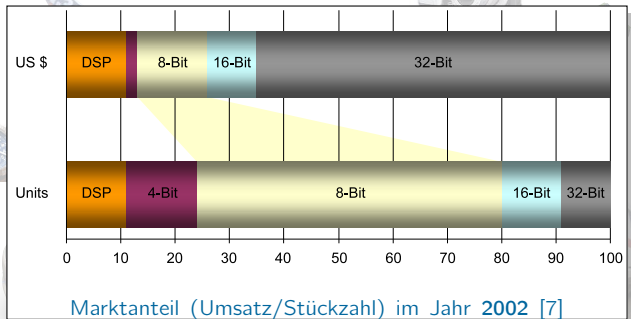


- **Vertiefen** des Wissens über Konzepte und Techniken der Informatik für die Softwareentwicklung
 - Ausgangspunkt: Grundlagen der Informatik (GdI)
 - Schwerpunkt: Systemnahe Softwareentwicklung in C
- **Entwickeln** von Software in C für einen μ -Controller (μ C) und eine Betriebssystem-Plattform (Linux)
 - SPiCboard-Lehrentwicklungsplattform mit ATmega- μ C
 - **Praktische Erfahrungen** in hardware- und systemnaher Softwareentwicklung machen
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen für die Entwicklung systemnaher Software
 - Die Sprache C verstehen und einschätzen können
 - Umgang mit Nebenläufigkeit und Hardwarenähe
 - Umgang mit den Abstraktionen eines Betriebssystems (Dateien, Prozesse, ...)



Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren wurden im Jahr 2000 in einem **eingebetteten System** verbaut [6]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und μ -Controller, **8-Bit oder kleiner** [6, 7]



Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren wurden im Jahr 2000 in einem **eingebetteten System** verbaut [6]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und μ -Controller, **8-Bit oder kleiner** [6, 7]
- **Relevant:** **25 Prozent** der Stellenanzeigen für EE-Ingenieure enthalten die Stichworte *embedded* oder *automotive* (<http://stepstone.com>, 4. April 2011)

Bei den oberen Zahlen ist gesunde Skepsis geboten

- Die Veröffentlichungen [6, 7] sind **um die 10 Jahre** alt!
- Man kann dennoch davon ausgehen, dass die **relativen Größenordnungen** nach wie vor stimmen
 - 2011 liegt der Anteil an 8-Bitern (vermutlich) noch weit über 50 Prozent
 - 4-Bitter dürften inzwischen jedoch weitgehend ausgestorben sein



Motivation: Die ATmega- μ C-Familie (8-Bit)

Type	Flash	SRAM	IO	Timer 8/16	UART	I ² C	AD	Price (€)
ATTINY11	1 KiB		6	1/-	-	-	-	0.31
ATTINY13	1 KiB	64 B	6	1/-	-	-	4*10	0.66
ATTINY2313	2 KiB	128 B	18	1/1	1	1	-	1.06
ATMEGA4820	4 KiB	512 B	23	2/1	2	1	6*10	1.26
ATMEGA8515	8 KiB	512 B	35	1/1	1	-	-	2.04
ATMEGA8535	8 KiB	512 B	32	2/1	1	1	-	2.67
ATMEGA169	16 KiB	1024 B	54	2/1	1	1	8*10	4.03
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	5.60
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	7.91

ATmega-Varianten (Auswahl) und Großhandelspreise (DigiKey 2006)

- Sichtbar wird: **Ressourcenknappheit**
 - **Flash** (Speicher für Programmcode und konstante Daten) ist **knapp**
 - **RAM** (Speicher für Laufzeit-Variablen) ist **extrem knapp**
 - Wenige Bytes „Verschwendung“ \rightsquigarrow signifikant höhere Stückzahlkosten



Motivation: Die Sprache C

- Systemnahe Softwareentwicklung erfolgt überwiegend in **C**
 - **Warum C?** (und nicht Java/Cobol/Scala/<Lieblingssprache>)
- C steht für eine Reihe hier wichtiger Eigenschaften
 - Laufzeiteffizienz (CPU)
 - Übersetzter C-Code läuft direkt auf dem Prozessor
 - Keine Prüfungen auf Programmierfehler zur Laufzeit
 - Platzeffizienz (Speicher)
 - Code und Daten lassen sich sehr kompakt ablegen
 - Keine Prüfung der Datenzugriffe zur Laufzeit
 - Direktheit (Maschinennähe)
 - C erlaubt den direkten Zugriff auf Speicher und Register
 - Portabilität
 - Es gibt für **jede** Plattform einen C-Compiler
 - C wurde „erfunden“ (1973), um das Betriebssystem UNIX portabel zu implementieren [3, 5]



~> **C** ist die **lingua franca** der systemnahen Softwareentwicklung!



- **Lehrziel:** Systemnahe Softwareentwicklung in C
 - Das ist ein sehr umfangreiches Feld: **Hardware-Programmierung**, **Betriebssysteme**, Middleware, Datenbanken, Verteilte Systeme, Übersetzerbau, ...
 - Dazu kommt dann noch das Erlernen der Sprache C selber
- **Ansatz**
 - Konzentration auf zwei Domänen
 - μ -Controller-Programmierung
 - Softwareentwicklung für die Linux-Systemschnittstelle
 - Gegensatz μ C-Umgebung \leftrightarrow Betriebssystemplattform erfahren
 - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
 - **Hohe Relevanz** für die Zielgruppe (ME)



- Das Handout der Vorlesungsfolien wird online und als 4 × 1-Ausdruck auf Papier zur Verfügung gestellt
 - Ausdrücke werden vor der Vorlesung verteilt
 - Online-Version wird vor der Vorlesung aktualisiert
 - Handout enthält (in geringem Umfang) zusätzliche Informationen

- **Das Handout kann eine eigene Mitschrift nicht ersetzen!**



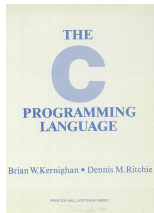
[2] Für den Einstieg empfohlen:

Manfred Dausmann, Ulrich Bröckl, Dominic Schoop, et al. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter `/proj/i4gspic/pub`). Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: <http://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1>



[4] Der „Klassiker“ (eher als Referenz geeignet):

Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960



- Inhalt und Themen
 - Grundlegende Konzepte der systemnahen Programmierung
 - Einführung in die Programmiersprache C
 - Unterschiede zu Java
 - Modulkonzept
 - Zeiger und Zeigerarithmetik
 - Softwareentwicklung auf „der nackten Hardware“ (ATmega- μ C)
 - Abbildung Speicher \leftrightarrow Sprachkonstrukte
 - Unterbrechungen (*interrupts*) und Nebenläufigkeit
 - Softwareentwicklung auf „einem Betriebssystem“ (Linux)
 - Betriebssystem als Ausführungsumgebung für Programme
 - Abstraktionen und Dienste eines Betriebssystems
- Termin: Mi 08:15–09:45, KS II
 - Einzeltermin am 5. Mai (Do), 10:15–11:45, H9
 - insgesamt 14–15 Vorlesungstermine



- Kombinierte Tafel- und Rechnerübung (jeweils im Wechsel)
 - Tafelübungen
 - Ausgabe und Erläuterung der Programmieraufgaben
 - Gemeinsame Entwicklung einer Lösungsskizze
 - Besprechung der Lösungen
 - Rechnerübungen
 - selbstständige Programmierung
 - Umgang mit Entwicklungswerkzeug (AVR Studio)
 - Betreuung durch Übungsbetreuer
- Termin: Initial 6 Gruppen zur Auswahl
 - Anmeldung über Waffel (siehe Webseite): Heute, 10:00 – Do, 08:00
 - Bei nur 2–3 Teilnehmern behalten wir uns eine Verteilung auf andere Gruppen vor. Ihr werdet in diesem Fall per E-Mail angeschrieben.



Programmieraufgaben

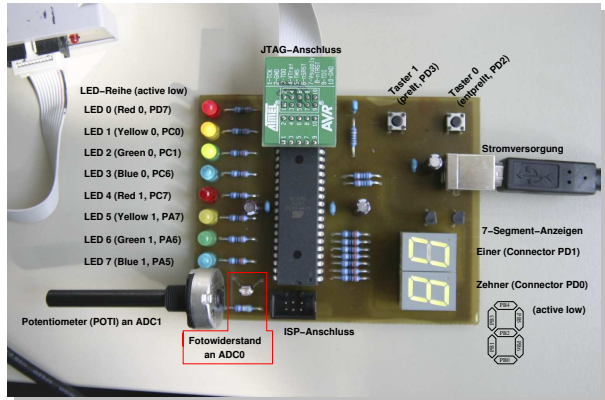
- Praktische Umsetzung des Vorlesungsstoffs
 - Fünf Programmieraufgaben (Abgabe ca. alle 14 Tage) → 2-7
 - Bearbeitung wechselseitig alleine / mit Übungspartner
- Lösungen mit Abgabeskript am Rechner abgeben
 - Lösung wird durch Skripte überprüft
 - Wir korrigieren und bepunktet die Abgaben und geben sie zurück
 - Eine Lösung wird vom Teilnehmer an der Tafel erläutert (impliziert Anwesenheit!)
- ★ Abgabe der Übungsaufgaben ist **freiwillig**; → 2-6
es können jedoch bis zu **10% Bonuspunkte**
für die Prüfungsklausur erarbeitet werden!

Unabhängig davon ist die Teilnahme an den Übungen **dringend empfohlen!**



Übungsplattform: Das SPiCboard

- ATmega32- μ C
- JTAG-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor



- Ausleihe zur Übungsbearbeitung möglich
- Oder noch besser \leftrightarrow selber Löten



- Die Fachschaften (EEI / ME) bieten drei „Lötabende“ an
 - Teilnahme ist freiwillig
 - (Erste) Lötterfahrung sammeln beim Löten eines eigenen SPiCboards
- **Termine:** 17./18./19. Mai, jeweils 18:00–21:00
- **Anmeldung:** über Waffel (siehe Webseite)
- **Kostenbeitrag:** 12–13 EUR (SPiCBoard)
22 EUR (ISP, falls gewünscht)



- Prüfung (Klausur)
 - Termin: voraussichtlich Ende Juli / Anfang August
 - Dauer: 90 min
 - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe

- Klausurnote \mapsto Modulnote
 - Bestehensgrenze (in der Regel): 50% der möglichen Klausurpunkte (KP)
 - Falls **bestanden** ist eine Notenverbesserung möglich durch Bonuspunkte aus den Programmieraufgaben
 - Basis (Minimum): 50% der möglichen Übungspunkte (ÜP)
 - Jede weiteren 5% der möglichen ÜP \mapsto +1% der möglichen KP
 - \rightsquigarrow 100% der möglichen ÜP \mapsto +10% der möglichen KP



Semesterüberblick

Siehe http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC

KW	Mo	Di	Mi	Do	Fr	Themen
18	02.05.	03.05.	04.05.	05.05.	06.05.	<i>Einführung, Organisation, Java nach C, Abstraktion, Sprachüberblick, Datentypen</i>
			VL1	VL2		
19	09.05.	10.05.	11.05.	12.05.	13.05.	<i>Variablen, Ausdrücke, Kontrollstrukturen, Funktionen, Makros</i>
			VL3	A1 (Blink)		
20	16.05.	17.05.	18.05.	19.05.	20.05.	<i>Mikrocontroller-Systemarchitektur, -Softwareentwicklung</i>
			VL4	A2 (Snake)		
21	23.05.	24.05.	25.05.	26.05.	27.05.	<i>Programmstruktur, Module, komplexe Datentypen</i>
			VL5	A3 (Spiel)		
22	30.05.	31.05.	01.06.	02.06.	03.06.	<i>Zeiger und Felder</i>
			VL6	Himmelfahrt		
23	06.06.	07.06.	08.06.	09.06.	10.06.	<i>Nebenläufigkeit</i>
	A4(LED)		VL 7			
24	13.06.	14.06.	15.06.	16.06.	17.06.	<i>Speicherorganisation</i>
	Pfingsten/Berg		VL8	A5 (Ampel)		



Semesterüberblick

Siehe http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC

KW	Mo	Di	Mi	Do	Fr	Themen
25	20.06.	21.06.	22.06.	23.06.	24.06.	Dateisysteme
			VL9	Fronleichnam		
26	27.06.	28.06.	29.06.	30.06.	01.07.	Prozesse
	A6 (PrintDir)		VL10			
27	04.07.	05.07.	06.07.	07.07.	08.07.	Signale
	A7 (fish)		VL11			
28	11.07.	12.07.	13.07.	14.07.	15.07.	Threads, Koordinierung
	A7 (tqsh)		VL12			
29	18.07.	19.07.	20.07.	21.07.	22.07.	pthreads, Threads in Java
			VL13			
30	25.07.	26.07.	27.07.	28.07.	29.07.	Wiederholung
	Wdh.		VL14			
29	19.07.	20.07.	21.07.	22.07.	23.07.	Fragestunde
			VL15			



Dozenten Vorlesung



Daniel Lohmann



Jürgen Kleinöder

Organisatoren des Übungsbetriebs



Wanja Hofer



Moritz Strübe



Christoph Erhardt



Dirk Wischermann



Techniker (Ausleihe SPiCboard und Debugger)



Harald Jungunst



Matthias Schäfer



Daniel Christiani

Übungsleiter



Daniel Back



Fabian Fersterra



Markus Müller



Sebastian
Schinabeck



Matthias Völkel



Systemnahe Programmierung in C (SPiC)

Teil B Einführung in C

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2011

http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC



Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Schichtarchitekturen und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor



Das erste C-Programm

- Das berühmteste Programm der Welt in **C**

```
#include <stdio.h>

int main(int argc, char** argv) {
    // greet user
    printf("Hello World!\n");
    return 0;
}
```

- Übersetzen und Ausführen (auf einem UNIX-System)

```
lohmann@latte:~/src$ gcc -o hello hello-linux.c
lohmann@latte:~/src$ ./hello
Hello World!
lohmann@latte:~/src$
```

Gar nicht so
schwer :-)



Das erste C-Programm – Vergleich mit Java

■ Das berühmteste Programm der Welt in **C**

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     // greet user
5     printf("Hello World!\n");
6     return 0;
7 }
```

■ Das berühmteste Programm der Welt in **Java**

```
1 import java.lang.System;
2 class Hello {
3     public static void main(String[] args) {
4         /* greet user */
5         System.out.println("Hello World!");
6         return;
7     }
8 }
```



- **C-Version** zeilenweise erläutert
 - 1 Für die Benutzung von `printf()` wird die **Funktionsbibliothek** `stdio.h` mit der **Präprozessor-Anweisung** `#include` eingebunden.
 - 3 Ein C-Programm startet in `main()`, einer **globalen Funktion** vom Typ `int`, die in genau einer **Datei** definiert ist.
 - 5 Die Ausgabe einer Zeichenkette erfolgt mit der **Funktion** `printf()`. (`\n` ~ Zeilenumbruch)
 - 6 Rückkehr zum Betriebssystem mit **Rückgabewert**. 0 bedeutet hier, dass kein Fehler aufgetreten ist.

- **Java-Version** zeilenweise erläutert
 - 1 Für die Benutzung der **Klasse** `out` wird das **Paket** `System` mit der `import`-Anweisung eingebunden.
 - 2 Jedes Java-Programm besteht aus mindestens einer **Klasse**.
 - 3 Jedes Java-Programm startet in `main()`, einer **statischen Methode** vom Typ `void`, die in genau einer **Klasse** definiert ist.
 - 5 Die Ausgabe einer Zeichenkette erfolgt mit der **Methode** `println()` aus der Klasse `out` aus dem Paket `System`. [`↔` GDI, IV-191]
 - 6 Rückkehr zum Betriebssystem.



Das erste C-Programm für einen μ -Controller

- „Hello World“ für AVR-ATmega (SPiCboard)

```
#include <avr/io.h>

void main() {
    // initialize hardware: LED on port D pin 7, active low
    DDRD  |= (1<<7); // PD7 is used as output
    PORTD |= (1<<7); // PD7: high --> LED is off

    // greet user
    PORTD &= ~(1<<7); // PD7: low --> LED is on

    // wait forever
    while(1){
    }
}
```

μ -Controller-Programmierung
ist „irgendwie anders“.

- Übersetzen und **Flashen** (mit AVR Studio) ↪ Übung
- Ausführen (SPiCboard):  (rote LED leuchtet)



Das erste C-Programm für einen μ -Controller

- „Hello World“ für AVR ATmega (vgl. [↔ 3-1](#))

```
1 #include <avr/io.h>
2
3 void main() {
4     // initialize hardware: LED on port D pin 7, active low
5     DDRD  |= (1<<7); // PD7 is used as output
6     PORTD |= (1<<7); // PD7: high --> LED is off
7
8     // greet user
9     PORTD &= ~(1<<7); // PD7: low --> LED is on
10
11    // wait forever
12    while(1){
13    }
14 }
```



- μ -Controller-Programm zeilenweise erläutert
(Beachte Unterschiede zur Linux-Version \leftrightarrow 3-3)
 - 1 Für den Zugriff auf Hardware-Register (DDRD, PORTD, bereitgestellt als **globale Variablen**) wird die **Funktionsbibliothek** `avr/io.h` mit `#include` eingebunden.
 - 3 Die `main()`-Funktion hat **keinen Rückgabewert** (Typ `void`). Ein μ -Controller-Programm läuft **endlos** \rightsquigarrow `main()` terminiert nie.
 - 5 Zunächst wird die **Hardware** initialisiert (in einen definierten Zustand gebracht). Dazu müssen **einzelne Bits** in bestimmten **Hardware-Registern** manipuliert werden.
 - 9 Die Interaktion mit der Umwelt (hier: LED einschalten) erfolgt ebenfalls über die **Manipulation einzelner Bits** in Hardware-Registern.
 - 12 Es erfolgt **keine Rückkehr** zum Betriebssystem (wohin auch?). Die Endlosschleife stellt sicher, dass `main()` nicht terminiert.



Das zweite C-Programm – Eingabe unter Linux

- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```
#include <stdio.h>

int main(int argc, char** argv){
    printf("Press key: ");
    int key = getchar();

    printf("You pressed %c\n", key);
    return 0;
}
```

Die `getchar()`-Funktion liest ein Zeichen von der Standardeingabe (hier: Tastatur). Sie „wartet“ gegebenenfalls, bis ein Zeichen verfügbar ist. In dieser Zeit entzieht das Betriebssystem den Prozessor.



- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

```
1 #include <avr/io.h>
2
3 void main() {
4     // initialize hardware: button on port D pin 2
5     DDRD  &= ~(1<<2); // PD2 is used as input
6     PORTD |= (1<<2); // activate pull-up: PD2: high
7
8     // initialize hardware: LED on port D pin 7, active low
9     DDRD  |= (1<<7); // PD7 is used as output
10    PORTD |= (1<<7); // PD7: high --> LED is off
11
12    // wait until PD2 -> low (button is pressed)
13    while(PIND & (1<<2))
14        ;
15
16    // greet user
17    PORTD &= ~(1<<7); // PD7: low --> LED is on
18
19    // wait forever
20    while(1)
21        ;
22 }
```



- Benutzerinteraktion mit SPiCboard zeilenweise erläutert
 - 5 Wie die LED ist der Taster mit einem **digitalen IO-Pin** des μ -Controllers verbunden. Hier konfigurieren wir Pin 2 von Port D als **Eingang** durch **Löschen** des entsprechenden Bits im Register `DDRD`.
 - 6 Durch **Setzen** von Bit 2 im Register `PORTD` wird der interne Pull-Up-Widerstand (hochohmig) aktiviert, über den V_{CC} anliegt \rightsquigarrow PD2 = *high*.
 - 13 **Aktive Warteschleife:** Wartet auf Tastendruck, d. h. solange PD2 (Bit 2 im Register `PIND`) *high* ist. Ein Tasterdruck zieht PD2 auf Masse \rightsquigarrow Bit 2 im Register `PIND` wird *low* und die Schleife verlassen.



Zum Vergleich: Benutzerinteraktion als Java-Programm

Eingabe als „typisches“
Java-Programm
(**objektorientiert, grafisch**)

```
1 import java.lang.System;
2 import javax.swing.*;
3 import java.awt.event.*;
4
5 public class Input implements ActionListener {
6     private JFrame frame;
7
8     public static void main(String[] args) {
9         // create input, frame and button objects
10        Input input = new Input();
11        input.frame = new JFrame("Java-Programm");
12        JButton button = new JButton("Klick mich");
13
14        // add button to frame
15        input.frame.add(button);
16        input.frame.setSize(400, 400);
17        input.frame.setVisible(true);
18
19        // register input as listener of button events
20        button.addActionListener(input);
21    }
22
23    public void actionPerformed(ActionEvent e) {
24        System.out.println("Knopfdruck!");
25        System.exit(0);
26    }
27 }
```



- Das Programm ist mit der C-Variante nicht unmittelbar vergleichbar
 - Es verwendet das in Java übliche (und Ihnen bekannte) **objektorientierte Paradigma**.
 - Dieser Unterschied soll hier verdeutlicht werden.

- Benutzerinteraktion in Java zeilenweise erläutert
 - 5 Um Interaktionsereignisse zu empfangen, implementiert die Klasse `Input` ein entsprechendes **Interface**.
 - 10 Das Programmverhalten ist implementiert durch eine Menge von **Objekten** (`frame`, `button`, `input`), die hier bei der Initialisierung erzeugt werden.
 - 20 Das erzeugte `button`-Objekt schickt nun seine Nachrichten an das `input`-Objekt.
 - 23 Der Knopfdruck wird durch eine `actionPerformed()`-Nachricht (Methodenaufruf) signalisiert.



Ein erstes Fazit: Von Java → C (Syntax)

- **Syntaktisch** sind Java und C sich sehr ähnlich
(Syntax: „Wie sehen **gültige** Programme der Sprache aus?“)
- C-Syntax war Vorbild bei der Entwicklung von Java
↪ Viele Sprachelemente sind ähnlich oder identisch verwendbar
 - Blöcke, Schleifen, Bedingungen, Anweisungen, Literale
 - Werden in den folgenden Kapiteln noch im Detail behandelt
- Wesentliche Sprachelemente aus Java gibt es in C jedoch **nicht**
 - Klassen, Pakete, Objekte, Ausnahmen (Exceptions), . . .



Ein erstes Fazit: Von Java → C (Idiomatik)

- **Idiomatisch** gibt es sehr große Unterschiede (Idiomatik: „Wie sehen **übliche** Programme der Sprache aus?“)
- **Java: Objektorientiertes Paradigma**
 - Zentrale Frage: Aus welchen **Dingen** besteht das Problem?
 - Gliederung der Problemlösung in **Klassen** und **Objekte**
 - Hierarchiebildung durch **Vererbung** und **Aggregation**
 - Programmablauf durch Interaktion zwischen **Objekten**
 - Wiederverwendung durch umfangreiche **Klassenbibliothek**
- **C: Imperatives Paradigma**
 - Zentrale Frage: Aus welchen **Aktivitäten** besteht das Problem?
 - Gliederung der Problemlösung in **Funktionen** und **Variablen**
 - Hierarchiebildung durch Untergliederung in **Teilfunktionen**
 - Programmablauf durch Aufrufe zwischen **Funktionen**
 - Wiederverwendung durch **Funktionsbibliotheken**



Ein erstes Fazit: Von Java → C (Philosophie)

- **Philosophisch** gibt es ebenfalls erhebliche Unterschiede (Philosophie: „Grundlegende Ideen und Konzepte der Sprache“)
- **Java:** Sicherheit und Portabilität durch **Maschinenferne**
 - Übersetzung für **virtuelle Maschine** (JVM)
 - **Umfangreiche** Überprüfung von Programmfehlern zur Laufzeit
 - Bereichsüberschreitungen, Division durch 0, ...
 - **Problemnahes** Speichermodell
 - Nur typsichere Speicherzugriffe, automatische Bereinigung zur Laufzeit
- **C:** Effizienz und Leichtgewichtigkeit durch **Maschinennähe**
 - Übersetzung für **konkrete Hardwarearchitektur**
 - **Keine** Überprüfung von Programmfehlern zur Laufzeit
 - Einige Fehler werden vom Betriebssystem abgefangen – **falls vorhanden**
 - **Maschinennahes** Speichermodell
 - Direkter Speicherzugriff durch **Zeiger**
 - Grobgranularer Zugriffsschutz und automatische Bereinigung (auf Prozessebene) durch das Betriebssystem – **falls vorhanden**



C \mapsto Maschinennähe \mapsto μ C-Programmierung

Die **Maschinennähe** von C zeigt sich insbesondere auch bei der μ -Controller-Programmierung!

- Es läuft nur ein Programm
 - Wird bei RESET direkt aus dem Flash-Speicher gestartet
 - Muss zunächst die Hardware initialisieren
 - Darf nie terminieren (z. B. durch Endlosschleife in `main()`)
- Die Problemlösung ist maschinennah implementiert
 - Direkte Manipulation von einzelnen Bits in Hardwareregistern
 - Detailliertes Wissen über die elektrische Verschaltung erforderlich
 - Keine Unterstützung durch Betriebssystem (wie etwa Linux)
 - Allgemein geringes Abstraktionsniveau \leadsto fehleranfällig, aufwändig

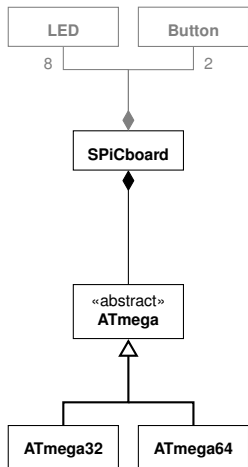
Ansatz: Mehr Abstraktion durch **problemorientierte Bibliotheken**



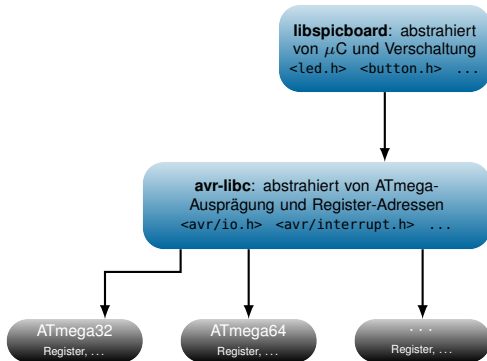
Abstraktion durch Softwareschichten: SPiCboard

↑ Problemnähe
↓ Maschinennähe

Hardwareansicht



Softwareschichten



Abstraktion durch Softwareschichten: *LED* → *on* im Vergleich

↑ Problemnähe

Programm läuft nur auf dem **SPiCboard**. Es verwendet Funktionen (wie `sb_set_led()`) und Konstanten (wie `RED0`) der **libspicboard**, welche die konkrete Verschaltung von LEDs, Tastern, usw. mit dem μC repräsentieren:

```
#include <led.h>
...
sb_set_led(RED0);
```

Programm läuft auf **jedem** μC der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

```
#include <avr/io.h>
...
DDRD |= (1<<7);
PORTD &= ~(1<<7);
```

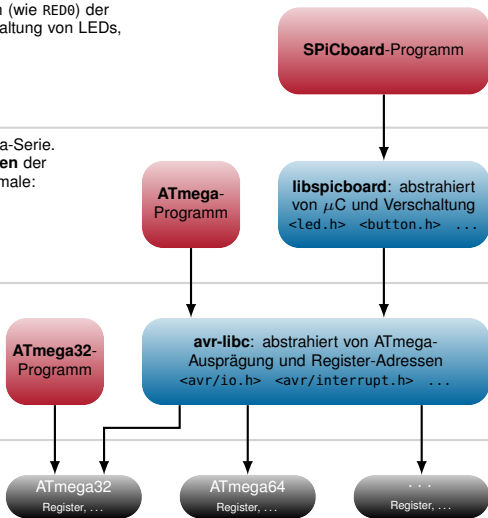
Programm läuft nur auf **ATmega32**. Es verwendet **ATmega32-spezifische** Registeradressen (wie `0x12`) und Merkmale:

```
...
(*(unsigned char*)(0x11)) |= (1<<7);
(*(unsigned char*)(0x12)) &= ~(1<<7);
```

Ziel: Schalte LED `RED0` auf SPiCboard an:



↓ Maschinennähe



Bisher: Entwicklung mit avr-libc

```
#include <avr/io.h>

void main() {
    // initialize hardware

    // button0 on PD2
    DDRD  &= ~(1<<2);
    PORTD |= (1<<2);
    // LED on PD7
    DDRD  |= (1<<7);
    PORTD |= (1<<7);

    // wait until PD2: low --> (button0 pressed)
    while(PIND & (1<<2)) {
    }

    // greet user (red LED)
    PORTD &= ~(1<<7); // PD7: low --> LED is on

    // wait forever
    while(1) {
    }
}
```

(vgl. ↔ [3-8](#))

Nun: Entwicklung mit libspicboard

```
#include <led.h>
#include <button.h>

void main() {

    // wait until Button0 is pressed
    while(sb_button_getState(BUTTON0)
        != BTNPPRESSED) {
    }

    // greet user
    sb_led_on(RED0);

    // wait forever
    while(1){
    }
}
```

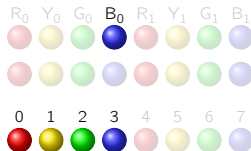
- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
 - Setze Bit 7 in PORTD
↳ `sb_set_led(RED0)`
 - Lese Bit 2 in PORTD
↳ `sb_button_getState(BUTTON0)`



■ Ausgabe-Abstraktionen (Auswahl)

■ LED-Modul (`#include <led.h>`)

- LED einschalten: `sb_set_led(BLUE0)` \rightsquigarrow
- LED ausschalten: `sb_clear_led(BLUE0)` \rightsquigarrow
- Alle LEDs ein-/ausschalten:
`sb_set_all_leds(0x0f)` \rightsquigarrow



■ 7-Seg-Modul (`#include <7seg.h>`)

- Ganzzahl $n \in \{-9 \dots 99\}$ ausgeben:
`sb_7seg_showNumber(47)` \rightsquigarrow



■ Eingabe-Abstraktionen (Auswahl)

■ Button-Modul (`#include <button.h>`)

- Button-Zustand abfragen:
`sb_button_getState(BUTTON0)` \mapsto `{BTNPRESSED, BTNRELEASED}`

■ ADC-Modul (`#include <adc.h>`)

- Potentiometer-Stellwert abfragen:
`sb_adc_read(POTI)` \mapsto `{0...1023}`

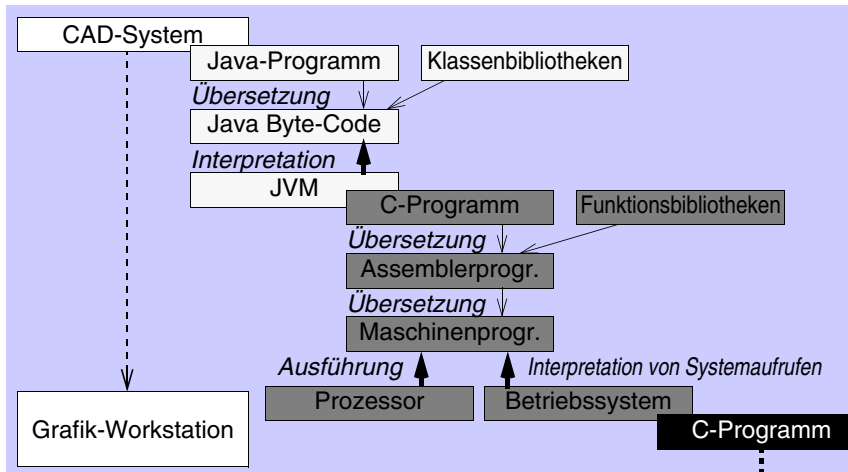


Softwareschichten im Allgemeinen

↑ Problemnähe

↓ Maschinennähe

Diskrepanz: Anwendungsproblem \longleftrightarrow Abläufe auf der Hardware



Ziel: Ausführbarer Maschinencode



- **Anwendersicht:** Umgebung zum Starten, Kontrollieren und Kombinieren von Anwendungen
 - Shell, grafische Benutzeroberfläche
 - z. B. bash, Windows
 - Datenaustausch zwischen Anwendungen und Anwendern
 - z. B. über Dateien
- **Anwendungssicht:** Funktionsbibliothek mit Abstraktionen zur Vereinfachung der Softwareentwicklung
 - Generische Ein-/Ausgabe von Daten
 - z. B. auf Drucker, serielle Schnittstelle, in Datei
 - Permanentspeicherung und Übertragung von Daten
 - z. B. durch Dateisystem, über TCP/IP-Sockets
 - Verwaltung von Speicher und anderen Betriebsmitteln
 - z. B. CPU-Zeit



- **Systemsicht:** Softwareschicht zum Multiplexen der Hardware (\leftrightarrow Mehrbenutzerbetrieb)
 - Parallele Abarbeitung von Programminstanzen durch **Prozesskonzept**
 - Virtueller Speicher \leftrightarrow eigener 32-/64-Bit-Adressraum
 - Virtueller Prozessor \leftrightarrow wird transparent zugeteilt und entzogen
 - Virtuelle Ein-/Ausgabe-Geräte \leftrightarrow umlenkbar in Datei, Socket, ...
 - Isolation von Programminstanzen durch **Prozesskonzept**
 - Automatische Speicherbereinigung bei Prozessende
 - Erkennung/Vermeidung von Speicherzugriffen auf fremde Prozesse
 - **Partieller Schutz** vor schwereren Programmierfehlern
 - Erkennung *einiger* ungültiger Speicherzugriffe (z. B. Zugriff auf Adresse 0)
 - Erkennung *einiger* ungültiger Operationen (z. B. $\text{div}/0$)

μC -Programmierung ohne Betriebssystemplattform \rightsquigarrow **kein Schutz**

- Ein Betriebssystem schützt **weit weniger** vor Programmierfehlern als z. B. Java.
- Selbst darauf müssen wir jedoch bei der μC -Programmierung i. a. **verzichten**.
- Bei 8/16-Bit- μC fehlt i. a. die für Schutz erforderliche **Hardware-Unterstützung**.