

Grundlagen der Systemnahen Programmierung in C (GSPiC)

Teil C Systemnahe Softwareentwicklung

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

<http://www4.cs.fau.de/Lehre/SS16/V-GSPiC>



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation



- Softwareentwurf: Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
 - Ziel: Zerlegung des Problems in beherrschbare Einheiten
- Es gibt eine Vielzahl von Softwareentwurfs-Methoden
 - Objektorientierter Entwurf [↔ GDI, 01-01]
 - Stand der Kunst
 - Dekomposition in Klassen und Objekte
 - An Programmiersprachen wie C++ oder Java ausgelegt
 - Top-Down-Entwurf / **Funktionale Dekomposition**
 - Bis Mitte der 80er Jahre fast ausschließlich verwendet
 - Dekomposition in Funktionen und Funktionsaufrufe
 - An Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert

Systemnahe Software wird oft (noch) mit **Funktionaler Dekomposition** entworfen und entwickelt.



Beispiel-Projekt: Eine Wetterstation

■ Typisches eingebettetes System

■ Mehrere Sensoren

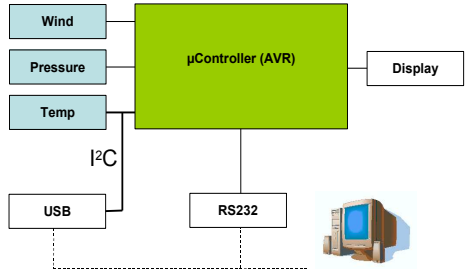
- Wind
- Luftdruck
- Temperatur

■ Mehrere Aktoren (hier: Ausgabegeräte)

- LCD-Anzeige
- PC über RS232
- PC über USB

■ Sensoren und Aktoren an den μC angebunden über verschiedene Bussysteme

- I²C
- RS232



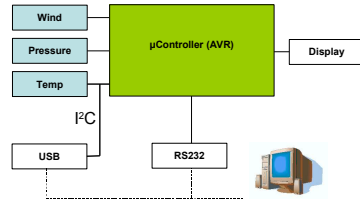
Wie sieht die **funktionale Dekomposition** der Software aus?



Funktionale Dekomposition: Beispiel

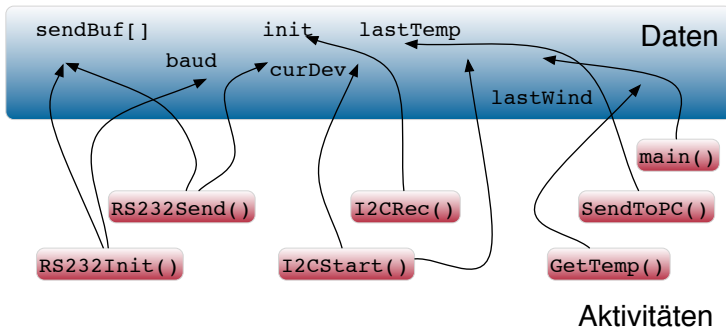
Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.1.1 I²C-Datenübertragung initiieren
 - 1.1.2 Daten vom I²C-Bus lesen
 - 1.2 Drucksensor lesen
 - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
 - 3.1 Daten über RS232 versenden
 - 3.1.1 Baudrate und Parität festlegen (einmalig)
 - 3.1.2 Daten schreiben
 - 3.2 LCD-Display aktualisieren
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der **Aktivitäten**, nicht jedoch die die Struktur der **Daten**
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten \rightsquigarrow mangelhafte Trennung der Belange



- Erzielte Gliederung betrachtet nur die Struktur der **Aktivitäten**, nicht jedoch die die Struktur der **Daten**
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten → mangelhafte Trennung der Belange

Prinzip der **Trennung der Belange**

Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

Trennung der Belange (*Separation of Concerns*) ist ein **Fundamentalprinzip** der Informatik (wie auch jeder anderen Ingenieursdisziplin).



Zugriff auf Daten (Variablen)

■ Variablen haben

↪ 10-1

- Sichtbarkeit (*Scope*) „Wer kann auf die Variable zugreifen?“
- Lebensdauer „Wie lange steht der Speicher zur Verfügung?“

■ Wird festgelegt durch Position (Pos) und Speicherklasse (SK)

Pos	SK	↪	Sichtbarkeit	Lebensdauer
Lokal	<i>keine</i> , auto static		Definition → Blockende Definition → Blockende	Definition → Blockende Programmstart → Programmende
Global	<i>keine</i> static		unbeschränkt modulweit	Programmstart → Programmende Programmstart → Programmende

```
int a = 0;           // a: global
static int b = 47;   // b: local to module

void f() {
    auto int a = b;   // a: local to function (auto optional)
                    // destroyed at end of block
    static int c = 11; // c: local to function, not destroyed
}
```



- Sichtbarkeit und Lebensdauer sollten **restriktiv** ausgelegt werden
 - Sichtbarkeit so **beschränkt wie möglich!**
 - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
 - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
 - Lebensdauer so **kurz wie möglich**
 - Speicherplatz sparen
 - Insbesondere wichtig auf μ -Controller-Plattformen

↪ 1-4

Konsequenz: Globale Variablen vermeiden!

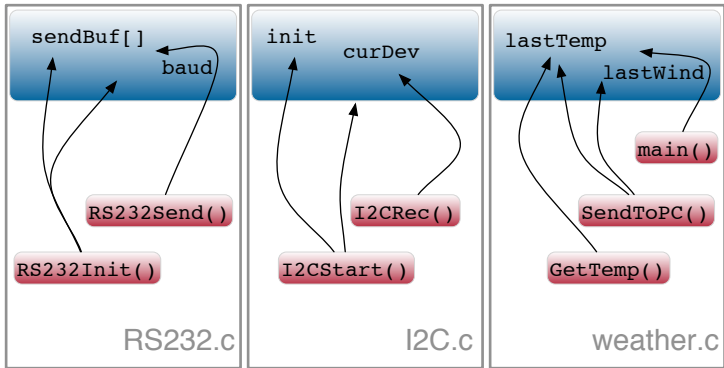
- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

Regel: Variablen erhalten stets die
geringstmögliche Sichtbarkeit und Lebensdauer



Lösung: Modularisierung

- Separation jeweils zusammengehöriger **Daten** und **Funktionen** in übergeordnete Einheiten \rightsquigarrow **Module**



Was ist ein Modul?

- **Modul** := (*<Menge von Funktionen>*, *<Menge von Daten>*, *<Schnittstelle>*) (↦ „**class**“ in Java)
- Module sind größere Programmbausteine ↔ 9-1
 - Problemorientierte Zusammenfassung von Funktionen und Daten
↪ Trennung der Belange
 - Ermöglichen die einfache Wiederverwendung von Komponenten
 - Ermöglichen den einfachen Austausch von Komponenten
 - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)
↪ Zugriff erfolgt ausschließlich über die Modulschnittstelle

Modul ↦ Abstraktion

↔ 4-1

- Die Schnittstelle eines Moduls **abstrahiert**
 - Von der tatsächlichen Implementierung der Funktionen
 - Von der internen Darstellung und Verwendung von Daten



- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein **idiomatisch** (über **Konventionen**) realisiert ↪ 3–13
 - Modulschnittstelle ↪ .h-Datei (enthält Deklarationen ↪ 9–7)
 - Modulimplementierung ↪ .c-Datei (enthält Definitionen ↪ 9–3)
 - Modulverwendung ↪ `#include <Modul.h>`

```
void RS232Init( uint16_t br );  
void RS232Send( char ch );  
...
```

RS232.h: Schnittstelle / Vertrag (öffentl.)
Deklaration der bereitgestellten Funktionen (und ggf. Daten)

```
#include <RS232.h>  
static uint16_t baud = 2400;  
static char sendBuf[16];  
...  
void RS232Init( uint16_t br ) {  
    ...  
    baud = br;  
}  
void RS232Send( char ch ) {  
    sendBuf[...] = ch;  
    ...  
}
```

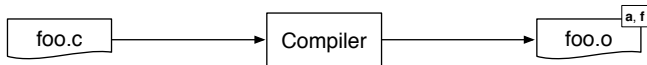
RS232.c: Implementierung (nicht öffentl.)
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (static)

Inklusion der eigenen Schnittstelle stellt sicher, dass der Vertrag eingehalten wird



- Ein C-Modul **exportiert** eine Menge von definierten **Symbolen**
 - Alle Funktionen und globalen Variablen (↪ „**public**“ in Java)
 - Export kann mit **static** unterbunden werden (↪ „**private**“ in Java)
(↪ Einschränkung der Sichtbarkeit ↪ 12-5)
- Export erfolgt beim Übersetzungsvorgang (.c-Datei → .o-Datei)



Quelldatei (foo.c)

```
uint16_t a;  
// public  
static uint16_t b;  
// private  
  
void f(void)           // public  
{ ... }  
static void g(int)     // private  
{ ... }
```

Objektdatei (foo.o)

Symbole **a** und **f** werden exportiert.

Symbole **b** und **g** sind **static** definiert und werden deshalb nicht exportiert.



- Ein C-Modul **importiert** eine Menge nicht-definierter **Symbole**
 - Funktionen und globale Variablen, die verwendet werden, im Modul selber jedoch nicht definiert sind
 - Werden beim Übersetzen als **unaufgelöst** markiert

Quelldatei (**bar.c**)

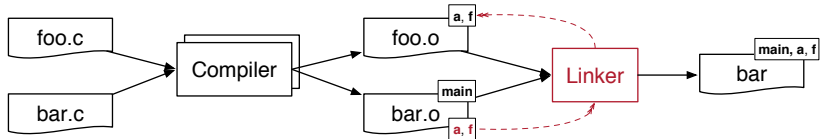
```
extern uint16_t a;  
// declare  
void f(void);      // declare  
  
void main() {      // public  
    a = 0x4711;    // use  
    f();           // use  
}
```

Objektdatei (**bar.o**)

Symbol **main** wird exportiert.
Symbole **a** und **f** sind **unaufgelöst**.



- Die eigentliche Auflösung erfolgt durch den **Linker**



Linken ist **nicht typsicher!**

- Typinformationen sind in Objektdateien nicht mehr vorhanden
- Auflösung durch den Linker erfolgt **ausschließlich** über die **Symbolnamen** (Bezeichner)
- ~> Typsicherheit muss beim **Übersetzen** sichergestellt werden
- ~> Einheitliche Deklarationen durch gemeinsame Header-Datei

- Elemente aus fremden Modulen müssen deklariert werden

- Funktionen durch normale Deklaration

↔ 9-7

```
void f(void);
```

- Globale Variablen durch **extern**

```
extern uint16_t a;
```

Das **extern** unterscheidet eine Variablendeklaration von einer Variablendefinition.

- Die Deklarationen erfolgen sinnvollerweise in einer **Header-Datei**, die von der Modulentwicklerin bereitgestellt wird

- Schnittstelle des Moduls

(↪ „**interface**“ in Java)

- Exportierte Funktionen des Moduls
- Exportierte globale Variablen des Moduls
- Modulspezifische Konstanten, Typen, Makros
- Verwendung durch Inklusion

(↪ „**import**“ in Java)

- Wird **auch vom Modul inkludiert**, um Übereinstimmung von Deklarationen und Definitionen sicher zu stellen

(↪ „**implements**“ in Java)



Modulschnittstelle: foo.h

```
// foo.h
#ifndef _F00_H
#define _F00_H

// declarations
extern uint16_t a;
void f(void);

#endif // _F00_H
```

Modulimplementierung foo.c

```
// foo.c
#include <foo.h>

// definitions
uint16_t a;
void f(void){
    ...
}
```

Modulverwendung bar.c

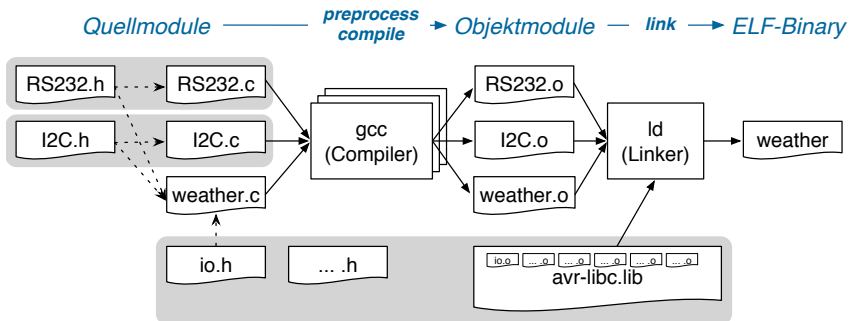
(vergleiche ↪ 12-11)

```
// bar.c
extern uint16_t a;
void f(void);
#include <foo.h>

void main() {
    a = 0x4711;
    f();
}
```



Zurück zum Beispiel: Wetterstation



- Jedes Modul besteht aus Header- und Implementierungsdatei(en)
 - .h-Datei definiert die Schnittstelle
 - .c-Datei implementiert die Schnittstelle, inkludiert .h-Datei, um sicherzustellen, dass Deklaration und Definition übereinstimmen
- Modulverwendung durch Inkludieren der modulspezifischen .h-Datei
- Das Ganze funktioniert entsprechend bei Bibliotheken



- Prinzip der Trennung der Belange \leadsto Modularisierung
 - Wiederverwendung und Austausch wohldefinierter Komponenten
 - Verbergen von Implementierungsdetails
- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern **idiomatisch** durch Konventionen realisiert
 - Modulschnittstelle \mapsto .h-Datei (enthält Deklarationen)
 - Modulimplementierung \mapsto .c-Datei (enthält Definitionen)
 - Modulverwendung \mapsto `#include <Modul.h>`
 - **private** Symbole \mapsto als `static` definieren
- Die eigentliche Zusammenfügung erfolgt durch den **Linker**
 - Auflösung erfolgt ausschließlich über Symbolnamen
 - \leadsto **Linken ist nicht typsicher!**
 - Typsicherheit muss beim Übersetzen sichergestellt werden
 - \leadsto durch gemeinsame Header-Datei



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation



Einordnung: Zeiger (*Pointer*)

- **Literal:** 'a'

Darstellung eines Wertes

'a' \equiv 0110 0001

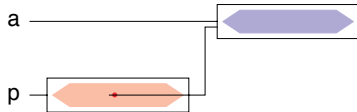
- **Variable:** `char a;`

Behälter für einen Wert



- **Zeiger-Variable:** `char *p = &a;`

Behälter für eine Referenz
auf eine Variable



Zeiger (*Pointer*)

- Eine Zeigervariable (*Pointer*) enthält als Wert die **Adresse** einer anderen Variablen
 - Ein Zeiger verweist auf eine Variable (im Speicher)
 - Über die Adresse kann man **indirekt** auf die Zielvariable (ihren Speicher) zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
 - Funktionen können Variablen des Aufrufers verändern (call-by-reference) ↪ 9-5
 - Speicher lässt sich direkt ansprechen
 - Effizientere Programme ↪ 3-14
- Aber auch viele Probleme!
 - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
 - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

„Effizienz durch
Maschinennähe“

↪ 3-14



Definition von Zeigervariablen

- **Zeigervariable** := Behälter für Verweise (\mapsto Adresse)
- Syntax (Definition): $Typ * Bezeichner ;$
- Beispiel

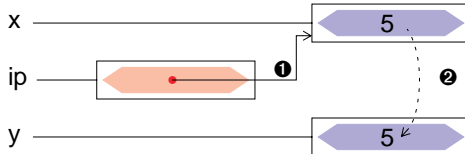
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ❶
```

```
y = *ip; ❷
```



Adress- und Verweisoperatoren

- **Adressoperator:** **&x** Der unäre **&**-Operator liefert die **Referenz** (\mapsto Adresse im Speicher) der Variablen **x**.
- **Verweisoperator:** ***y** Der unäre *****-Operator liefert die **Zielvariable** (\mapsto Speicherzelle / Behälter), auf die der Zeiger **y** verweist (Dereferenzierung).
- **Es gilt:** **(*(&x)) \equiv x** Der Verweisoperator ist die Umkehroperation des Adressoperators.

Achtung: Verwirrungsgefahr (***Ich seh überall Sterne* **)

Das *****-Symbol hat in C verschiedene Bedeutungen, **je nach Kontext**

1. Multiplikation (binär): **x * y** in Ausdrücken
2. Typmodifizierer: **uint8_t *p1, *p2** in Definitionen und
 typedef char* CPTR Deklarationen
3. Verweis (unär): **x = *p1** in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung

\leadsto ***** wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.



Zeiger als Funktionsargumente

- Parameter werden in C immer *by-value* übergeben ↔ 9-5
 - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
 - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern
- Das gilt auch für Zeiger (Verweise) [↔ GDI, 14-01-01]
 - Aufgerufene Funktion erhält eine Kopie des Adressverweises
 - Mit Hilfe des *-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

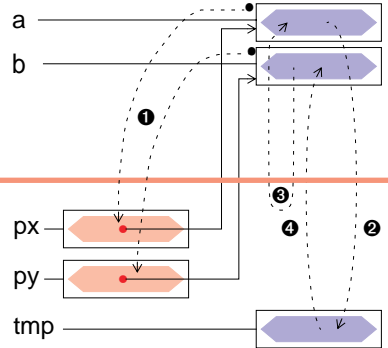
↪ **Call-by-reference**



■ Beispiel (Gesamtüberblick)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b); ❶  
    ...  
}
```

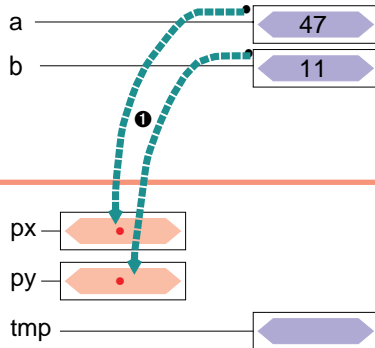
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ❷  
    *px = *py; ❸  
    *py = tmp; ❹  
}
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b); ❶
```

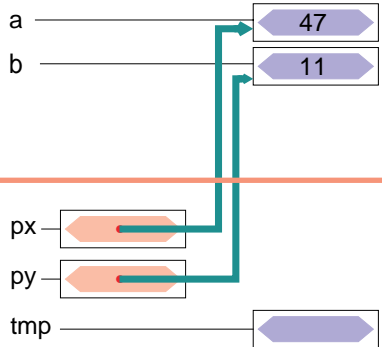
```
void swap (int *px, int *py)  
{  
    int tmp;
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

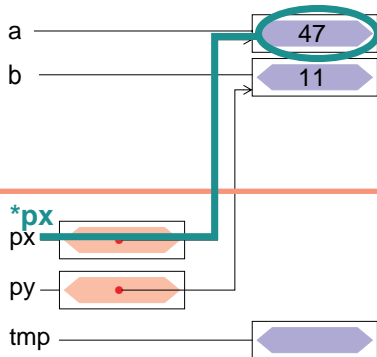
```
void swap (int *px, int *py)  
{  
    int tmp;  
    ...  
}
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
}
```



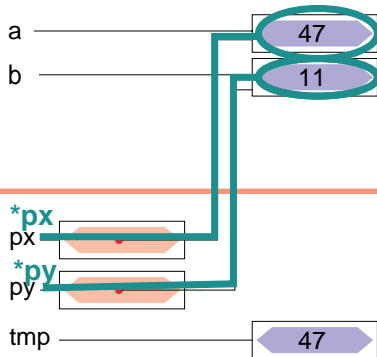
■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

```
void swap (int *px, int *py)  
{  
    int tmp;
```

```
    tmp = *px; ②
```

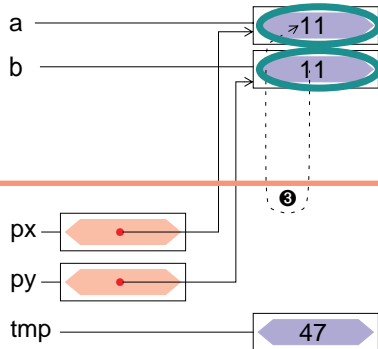
```
    *px = *py; ③
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

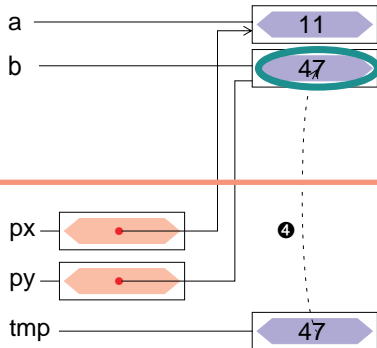
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
    *py = tmp; ④  
}
```



- **Feldvariable** := Behälter für eine Reihe von Werten desselben Typs
- Syntax (Definition): *Typ Bezeichner [IntAusdruck] ;*
 - *Typ* Typ der Werte [=Java]
 - *Bezeichner* Name der Feldvariablen [=Java]
 - *IntAusdruck* **Konstanter** Ganzzahl-Ausdruck, definiert die Feldgröße (\rightarrow Anzahl der Elemente). [\neq Java]
Ab **C99** darf *IntAusdruck* bei **auto**-Feldern auch **variabel** (d. h. beliebig, aber fest) sein.
- Beispiele:

```
static uint8_t LEDs[ 8*2 ];      // constant, fixed array size

void f( int n ) {
    auto char a[ NUM_LEDS * 2];  // constant, fixed array size
    auto char b[ n ];            // C99: variable, fixed array size
}
```



Feldinitialisierung

- Wie andere Variablen auch, kann ein Feld bei Definition eine **initiale Wertzuweisung** erhalten

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };  
int prim[5]      = { 1, 2, 3, 5, 7 };
```

- Werden zu wenig Initialisierungselemente angegeben, so werden die restlichen Elemente **mit 0 initialisiert**

```
uint8_t LEDs[4] = { RED0 };      // => { RED0, 0, 0, 0 }  
int prim[5]     = { 1, 2, 3 };  // => { 1, 2, 3, 0, 0 }
```

- Wird die explizite Dimensionierung ausgelassen, so bestimmt die **Anzahl** der Initialisierungselemente die Feldgröße

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
int prim[]     = { 1, 2, 3, 5, 7 };
```



■ Syntax: *Feld* [*IntAusdruck*]

[=Java]

■ Wobei $0 \leq \text{IntAusdruck} < n$ für $n = \text{Feldgröße}$

■ **Achtung:** Feldindex wird nicht überprüft
 ~ häufige Fehlerquelle in C-Programmen

[≠Java]

■ Beispiel

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };
```

```
LEDs[ 3 ] = BLUE1;
```

```
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( LEDs[ i ] );  
}
```

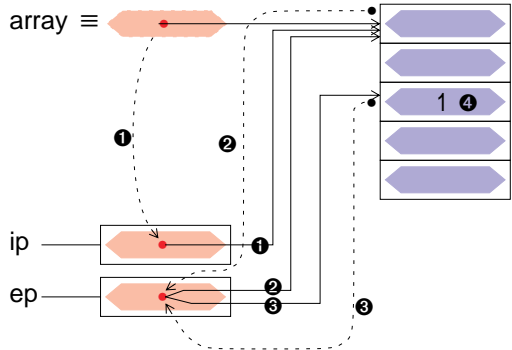
```
LEDs[ 4 ] = GREEN1;    // UNDEFINED!!!
```



Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \rightsquigarrow Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \rightsquigarrow Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

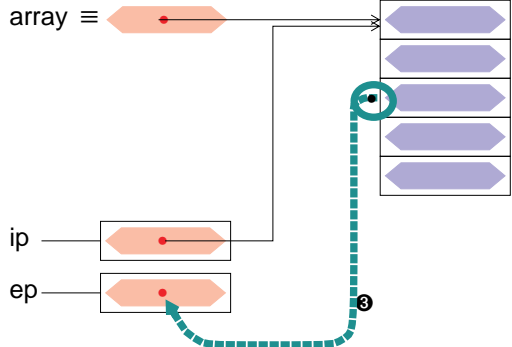
```
int array[5];
```

```
int *ip = array; ①
```

```
int *ep;
```

```
ep = &array[0]; ②
```

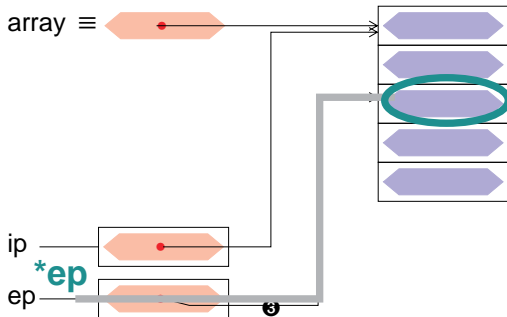
```
ep = &array[2]; ③
```



Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \rightsquigarrow Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



Zeiger sind Felder

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array` \equiv `&array[0]`
- Diese Beziehung gilt in beide Richtungen: `*array` \equiv `array[0]`
 - Ein Zeiger kann wie ein Feld verwendet werden
 - Insbesondere kann der `[]`-Operator angewandt werden ↪ 13-9
- Beispiel (vgl. ↪ 13-9)

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
  
LEDs[ 3 ]      = BLUE1;  
uint8_t *p     = LEDs;  
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( p[ i ] );  
}
```

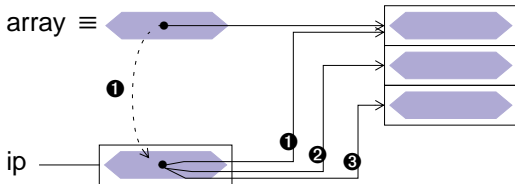


Rechnen mit Zeigern

- Im Unterschied zu einem Feldbezeichner ist eine *Zeigervariable* ein Behälter \rightsquigarrow Ihr Wert ist veränderbar
- Neben einfachen Zuweisungen ist dabei auch **Arithmetik** möglich

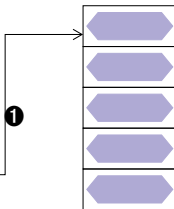
```
int array[3];  
int *ip = array; ❶
```

```
ip++; ❷  
ip++; ❸
```



```
int array[5];  
ip = array; ❶
```

ip



$(ip+3) \equiv \&ip[3]$

Bei der Zeigerarithmetik wird immer die Größe des Objekttyps mit berücksichtigt.



■ Arithmetische Operationen

- ++ Prä-/Postinkrement
↪ Verschieben auf das nächste Objekt
- Prä-/Postdekrement
↪ Verschieben auf das vorangegangene Objekt
- +, - Addition / Subtraktion eines `int`-Wertes
↪ Ergebniszeiger ist verschoben um n Objekte
 - Subtraktion zweier Zeiger
↪ Anzahl der Objekte n zwischen beiden Zeigern (Distanz)

■ Vergleichsoperationen: $<$, $<=$, $==$, $>=$, $>$, $!=$

↪ 7-3

- ↪ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen



Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C **jede** Feldoperation auf eine äquivalente Zeigeroperation abbilden.
- Für `int i, array[N], *ip = array;` mit $0 \leq i < N$ gilt:

```
array    ≡ &array[0]    ≡ ip          ≡ &ip[0]
*array   ≡ array[0]     ≡ *ip         ≡ ip[0]
*(array + i) ≡ array[i] ≡ *(ip + i) ≡ ip[i]
          array++ ≠ ip++
          Fehler: array ist konstant!
```

- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.
Der Feldbezeichner kann aber **nicht verändert** werden.



Felder als Funktionsparameter

- Felder werden in C **immer** als Zeiger übergeben

[=Java]

↪ *Call-by-reference*

```
static uint8_t LEDs[] = {RED0, YELLOW1};

void enlight( uint8_t *array, unsigned n ) {
    for( unsigned i = 0; i < n; ++i )
        sb_led_on( array[i] );
}

void main() {
    enlight( LEDs, 2 );
    uint8_t moreLEDs[] = {YELLOW0, BLUE0, BLUE1};
    enlight( moreLEDs, 3);
}
```



- Informationen über die Feldgröße gehen dabei verloren!
 - Die Feldgröße muss explizit als Parameter mit übergeben werden
 - In manchen Fällen kann sie auch in der Funktion berechnet werden (z. B. bei Strings durch Suche nach dem abschließenden **NUL**-Zeichen)

13-Zeiger: 2016-04-09



- Felder werden in C **immer** als Zeiger übergeben [=Java]

→ *Call-by-reference*

- Wird der Parameter als **const** deklariert, so kann die Funktion die Feldelemente **nicht verändern** → Guter Stil! [≠Java]

```
void enlight( const uint8_t *array, unsigned n ) {  
    ...  
}
```

- Um anzuzeigen, dass ein Feld (und kein „Zeiger auf Variable“) erwartet wird, ist auch folgende **äquivalente Syntax** möglich:

```
void enlight( const uint8_t array[], unsigned n ) {  
    ...  
}
```

- **Achtung:** Das gilt so nur bei Deklaration eines Funktionsparameters
- Bei Variablendefinitionen hat **array[]** eine **völlig andere** Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, → 13-8)



- Die Funktion `int strlen(const char *)` aus der Standardbibliothek liefert die Anzahl der Zeichen im übergebenen String

```
void main() {  
    ...  
    const char *string = "hallo"; // string is array of char  
    sb_7seg_showNumber( strlen(string) );  
    ...  
}
```



Dabei gilt: "hallo" \equiv  \leftrightarrow 6-13

- Implementierungsvarianten

Variante 1: Feld-Syntax

```
int strlen( const char s[] ) {  
    int n=0;  
    while( s[n] != 0 )  
        n++;  
    return n;  
}
```

Variante 2: Zeiger-Syntax

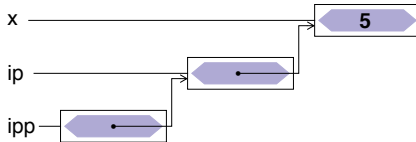
```
int strlen( const char *s ) {  
    const char *end = s;  
    while( *end )  
        end++;  
    return end - s;  
}
```



Zeiger auf Zeiger

- Ein Zeiger kann auch auf eine Zeigervariable verweisen

```
int x = 5;  
int *ip = &x;  
  
int **ipp = &ip;  
/* → **ipp = 5 */
```



- Wird vor allem bei der Parameterübergabe an Funktionen benötigt
 - Zeigerparameter *call-by-reference* übergeben
(z. B. `swap()`-Funktion für Zeiger)
 - Ein Feld von Zeigern übergeben



Zeiger auf Funktionen

- Ein Zeiger kann auch auf eine Funktion verweisen
 - Damit lassen sich Funktionen an Funktionen übergeben
 - Funktionen höherer Ordnung
- Beispiel

```
// invokes job() every second
void doPeriodically( void (*job)(void) ) {
    while( 1 ) {
        job();           // invoke job
        for( volatile uint16_t i = 0; i < 0xffff; ++i )
            ;             // wait a second
    }
}

void blink( void ) {
    sb_led_toggle( RED0 );
}

void main() {
    doPeriodically( blink ); // pass blink() as parameter
}
```



- Syntax (Definition): `Typ (* Bezeichner)(FormaleParamopt);`
(sehr ähnlich zur Syntax von Funktionsdeklarationen) ↪ 9-3

- *Typ* Rückgabebetyp der *Funktionen*, auf die dieser Zeiger verweisen kann
- *Bezeichner* Name des *Funktionszeigers*
- *FormaleParam_{opt}* Formale Parameter der *Funktionen*, auf die dieser Zeiger verweisen kann: *Typ₁, ..., Typ_n*

- Ein Funktionszeiger wird genau wie eine Funktion verwendet

- Aufruf mit `Bezeichner (TatParam)` ↪ 9-4
- Adress- (&) und Verweisoperator (*) werden nicht benötigt ↪ 13-4
- Ein Funktionsbezeichner ist ein konstanter Funktionszeiger

```
void blink( uint8_t which ) { sb_led_toggle( which ); }

void main() {
    void (*myfun)(uint8_t); // myfun is pointer to function
    myfun = blink;          // blink is constant pointer to function
    myfun( RED0 );          // invoke blink() via function pointer
    blink( RED0 );          // invoke blink()
}
```



- Funktionszeiger werden oft für **Rückruffunktionen** (*Callbacks*) zur Zustellung asynchroner Ereignisse verwendet (→ „Listener“ in Java)

```
// Example: asynchronous button events with libspicboard
#include <avr/interrupt.h>           // for sei()
#include <7seg.h>                     // for sb_7seg_showNumber()
#include <button.h>                   // for button stuff

// callback handler for button events (invoked on interrupt level)
void onButton( BUTTON b, BUTTONEVENT e ) {
    static int8_t count = 1;
    sb_7seg_showNumber( count++ ); // show no of button presses
    if( count > 99 ) count = 1;    // reset at 100
}

void main() {
    sb_button_registerListener(      // register callback
        BUTTON0, BTNPPRESSED,      //   for this button and events
        onButton                    //   invoke this function
    );
    sei();                           // enable interrupts (necessary!)
    while( 1 ) ;                     // wait forever
}
```



- Ein Zeiger verweist auf eine Variable im Speicher
 - Möglichkeit des **indirekten** Zugriffs auf den Wert
 - Grundlage für die Implementierung von *call-by-reference* in C
 - Grundlage für die Implementierung von Feldern
 - Wichtiges Element der **Maschinennähe** von C
 - **Häufigste Fehlerursache in C-Programmen**
- Die syntaktischen Möglichkeiten sind vielfältig (und verwirrend)
 - Typmodifizierer *, Adressoperator &, Verweisoperator *
 - Zeigerarithmetik mit +, -, ++ und --
 - syntaktische Äquivalenz zu Feldern ([] Operator)
- Zeiger können auch auf Funktionen verweisen
 - Übergeben von Funktionen an Funktionen
 - Prinzip der Rückruffunktion



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

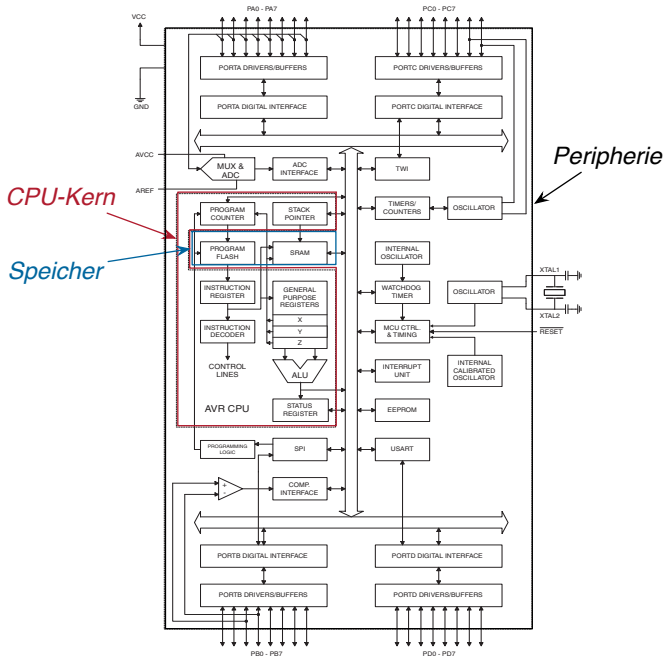


Was ist ein μ -Controller?

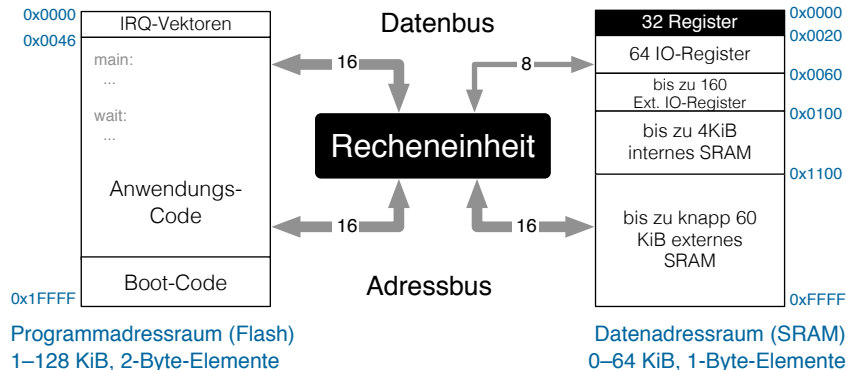
- **μ -Controller** := Prozessor + Speicher + Peripherie
 - Faktisch ein Ein-Chip-Computersystem \rightarrow SoC (*System-on-a-Chip*)
 - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher \leadsto kostengünstiges Systemdesign
- Wesentliches Merkmal ist die (reichlich) enthaltene Peripherie
 - Timer/Counter (Zeiten/Ereignisse messen und zählen)
 - Ports (digitale Ein-/Ausgabe), A/D-Wandler (analoge Eingabe)
 - PWM-Generatoren (pseudo-analoge Ausgabe)
 - Bus-Systeme: SPI, RS-232, CAN, Ethernet, MLI, I²C, ...
 - ...
- Die Abgrenzungen sind fließend: Prozessor $\longleftrightarrow \mu\text{C} \longleftrightarrow \text{SoC}$
 - AMD64-CPU's haben ebenfalls eingebaute Timer, Speicher (Caches), ...
 - Einige μC erreichen die Geschwindigkeit „großer Prozessoren“



Beispiel ATmega32: Blockschaltbild



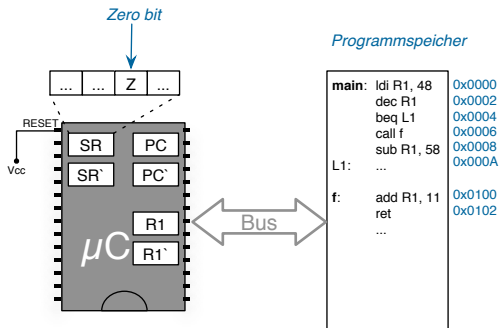
Beispiel ATmega-Familie: CPU-Architektur



- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingebündet
→ ansprechbar wie globale Variablen

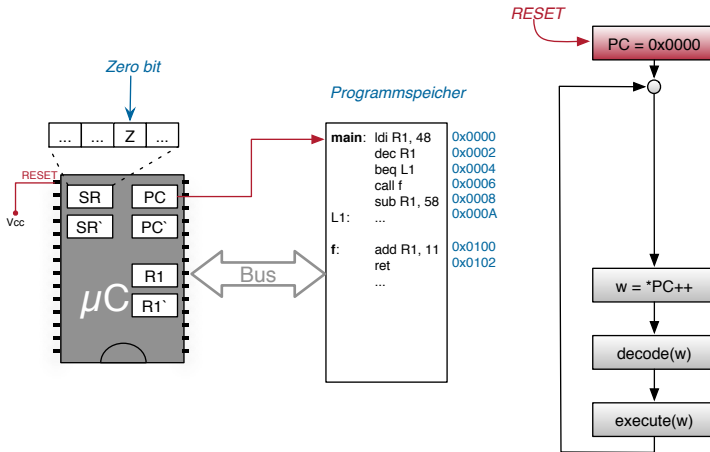
Zum Vergleich: PC basiert auf von-Neumann-Architektur [↔ GDI, 18-10] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

Wie arbeitet ein Prozessor?

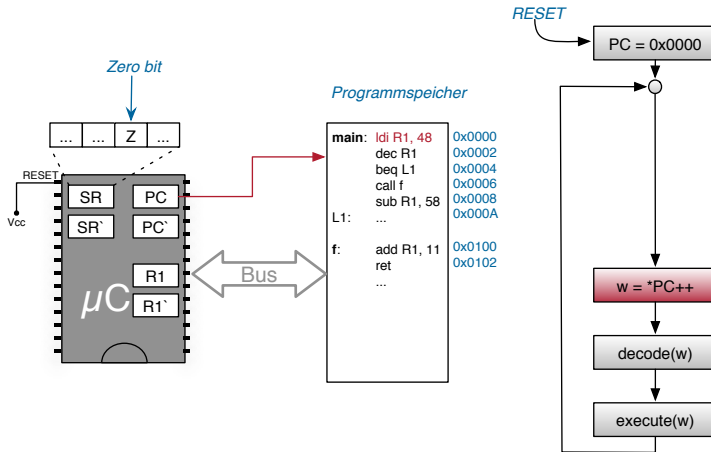


- Hier am Beispiel eines sehr einfachen Pseudoprozessors
 - Nur zwei Vielzweckregister (R1 und R2)
 - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
 - Kein Datenspeicher, kein Stapel \rightsquigarrow Programm arbeitet nur auf Registern

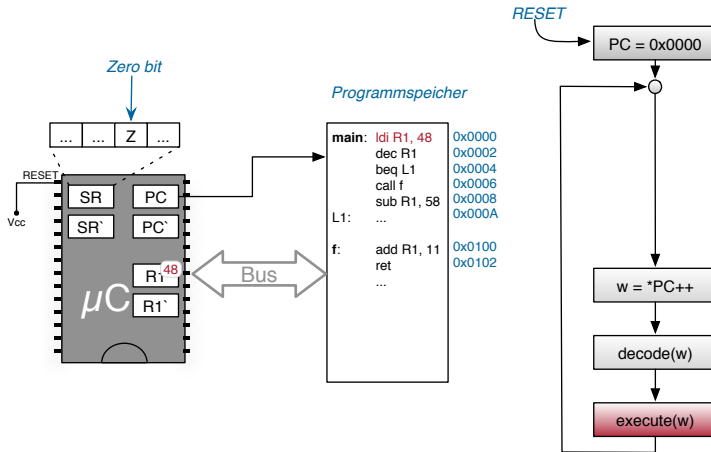
Wie arbeitet ein Prozessor?



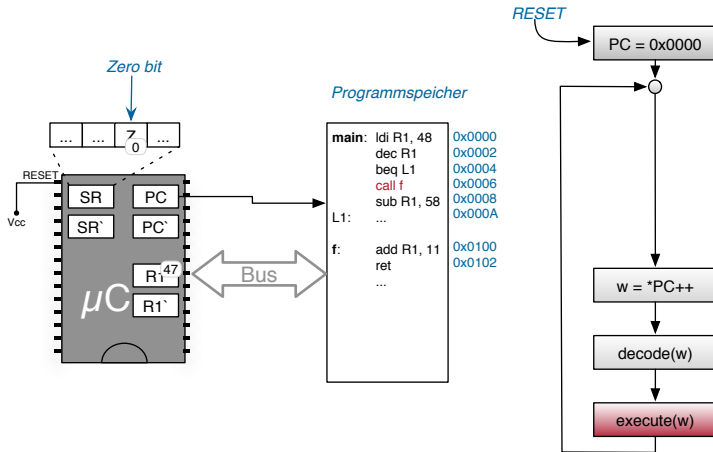
Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?

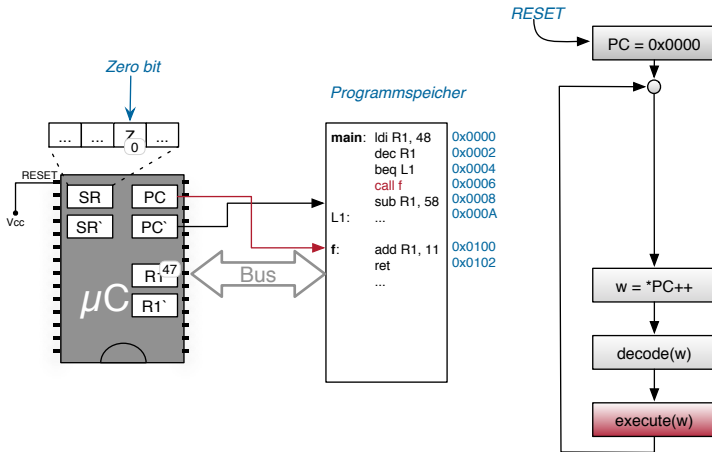


w: dec <R>
 R ← 1
 if (R = 0) Z = 1
 else Z = 0

w: beq <lab>
 if (Z) PC = lab

w: call <func>
 PC' = PC
 PC = func

Wie arbeitet ein Prozessor?

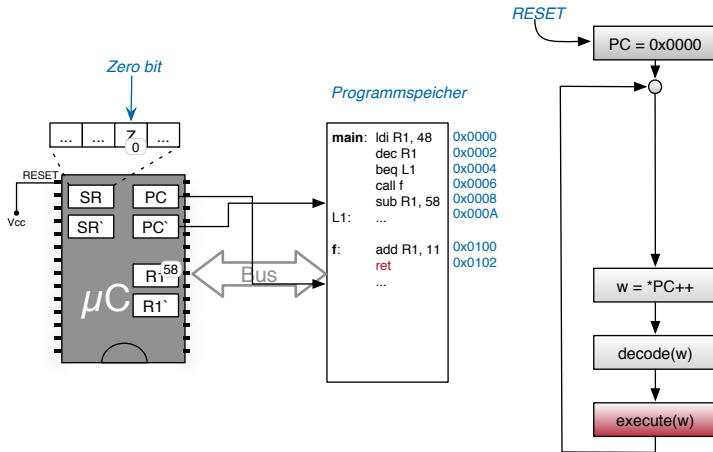


w: dec <R>
 R ← 1
 if (R == 0) Z = 1
 else Z = 0

w: beq <lab>
 if (Z) PC = lab

w: call <func>
 PC' = PC
 PC = func

Wie arbeitet ein Prozessor?



w: dec <R>
 R ← 1
 if (R == 0) Z = 1
 else Z = 0

w: beq <lab>
 if (Z) PC = lab

w: call <func>
 PC' = PC
 PC = func

w: ret
 PC = PC'

- **Peripheriegerät:** Hardwarekomponente, die sich „außerhalb“ der Zentraleinheit eines Computers befindet
 - Traditionell (PC): Tastatur, Bildschirm, ...
(→ physisch „außerhalb“)
 - Allgemeiner: Hardwarefunktionen, die nicht direkt im Befehlssatz des Prozessors abgebildet sind
(→ logisch „außerhalb“)
- Peripheriebausteine werden über **I/O-Register** angesprochen
 - Kontrollregister: Befehle an / Zustand der Peripherie wird durch **Bitmuster** kodiert (z. B. **DDRD** beim ATmega)
 - Datenregister: Dienen dem eigentlichen Datenaustausch (z. B. **PORTD**, **PIND** beim ATmega)
 - Register sind häufig für entweder nur Lesezugriffe (*read-only*) oder nur Schreibzugriffe (*write-only*) zugelassen



- Auswahl von typischen Peripheriegeräten in einem μ -Controller
 - Timer/Counter Zählregister, die mit konfigurierbarer Frequenz (Timer) oder durch externe Signale (Counter) erhöht werden und bei konfigurierbarem Zählwert einen Interrupt auslösen.
 - Watchdog-Timer Timer, der regelmäßig neu beschrieben werden muss oder sonst einen RESET auslöst („Totmannknopf“).
 - (A)synchrone serielle Schnittstelle Bausteine zur seriellen (bitweisen) Übertragung von Daten mit synchronem (z. B. RS-232) oder asynchronem (z. B. I²C) Protokoll.
 - A/D-Wandler Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V \rightarrow 10-Bit-Zahl).
 - PWM-Generatoren Bausteine zur Generierung von pulsweiten-modulierten Signalen (pseudo-analoge Ausgabe).
 - Ports Gruppen von üblicherweise 8 Anschlüssen, die auf GND oder Vcc gesetzt werden können oder deren Zustand abgefragt werden kann. \rightarrow 14–12



- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
 - Memory-mapped: Register sind in den Adressraum eingeblendet; der Zugriff erfolgt über die Speicherbefehle des Prozessors (**load**, **store**)
(Die meisten μC)
 - Port-basiert: Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle **in**- und **out**-Befehle
(x86-basierte PCs)
- Die Registeradressen stehen in der Hardware-Dokumentation

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
\$3F (\$5F)	SREG	I	T	H	S	V	N	Z	C	8
\$3E (\$5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	11
\$3D (\$5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
\$3C (\$5C)	OCR0	Timer/Counter0 Output Compare Register								86
\$12 (\$32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
\$11 (\$31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	67
\$10 (\$30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1, S. 334]

- Memory-mapped Register ermöglichen einen komfortablen Zugriff
 - Register \mapsto Speicher \mapsto Variable
 - Alle C-Operatoren stehen direkt zur Verfügung (z. B. PORTD++)
- Syntaktisch wird der Zugriff oft durch Makros erleichtert:

```
#define PORTD ( * (volatile uint8_t*)( 0x12 ) )
```

Diagramm zur Erklärung des Makros `PORTD`:

- `0x12` ist die Adresse (int).
- Der Cast `(volatile uint8_t*)` wird auf die Adresse angewendet (Cast \mapsto 7-17).
- Der Dereferenzierung `*` wird auf den Wert angewendet (Dereferenzierung \mapsto 13-4).

PORTD ist damit (syntaktisch) äquivalent zu einer `volatile uint8_t`-Variablen, die an Adresse `0x12` liegt

■ Beispiel

```
#define PORTD (*(volatile uint8_t*)(0x12))

PORTD |= (1<<7);           // set D.7
uint8_t *pReg = &PORTD;    // get pointer to PORTD
*pReg &= ~(1<<7);          // use pointer to clear D.7
```



Registerzugriff und Nebenläufigkeit

- Peripheriegeräte arbeiten **nebenläufig** zur Software
~> Wert in einem Hardwareregister kann sich **jederzeit ändern**
- Dies widerspricht einer Annahme des Compilers
 - Variablenzugriffe erfolgen **nur** durch die aktuell ausgeführte Funktion
~> Variablen können in Registern zwischengespeichert werden

```
// C code
#define PIND (*(uint8_t*)(0x10))
void foo(void) {
    ...
    if( !(PIND & 0x2) ) {
        // button0 pressed
        ...
    }
    if( !(PIND & 0x4) ) {
        // button 1 pressed
        ...
    }
}
```

```
// Resulting assembly code

foo:
    lds    r24, 0x0010 // PIND->r24
    sbrc   r24, 1      // test bit 1
    rjmp   L1
    // button0 pressed
    ...
L1:
    sbrc   r24, 2      // test bit 2
    rjmp   L2
    ...
L2:
    ret
```

PIND wird nicht erneut aus dem Speicher geladen. Der Compiler nimmt an, dass der Wert in r24 aktuell ist.



Der volatile-Typmodifizierer

- **Lösung:** Variable `volatile` („*flüchtig, unbeständig*“) deklarieren
 - Compiler hält Variable nur so kurz wie möglich im Register
 - ↪ Wert wird unmittelbar vor Verwendung gelesen
 - ↪ Wert wird unmittelbar nach Veränderung zurückgeschrieben

```
// C code
#define PIND \
    (*(volatile uint8_t*)(0x10))
void foo(void) {
    ...
    if( !(PIND & 0x2) ) {
        // button0 pressed
        ...
    }

    if( !(PIND & 0x4) ) {
        // button 1 pressed
        ...
    }
}
```

```
// Resulting assembly code

foo:
    lds    r24, 0x0010 // PIND->r24
    sbrc   r24, 1      // test bit 1
    rjmp   L1
    // button0 pressed
    ...

L1:
    lds    r24, 0x0010 // PIND->r24
    sbrc   r24, 2      // test bit 2
    rjmp   L2
    ...

L2:
    ret
```

PIND ist `volatile` und wird deshalb vor dem Test erneut aus dem Speicher geladen.



- Die `volatile`-Semantik verhindert viele Code-Optimierungen (insbesondere das Entfernen von **scheinbar unnützem Code**)
- Kann ausgenutzt werden, um aktives Warten zu implementieren:

```
// C code
void wait( void ){
    for( uint16_t i = 0; i<0xffff; i++)
    }
    volatile!
```

```
// Resulting assembly code
wait:
    // compiler has optimized
    // "unneeded" loop
    ret
```

Achtung: `volatile` \mapsto \$\$\$

Die Verwendung von `volatile` verursacht erhebliche **Kosten**

- Werte können nicht mehr in Registern gehalten werden
- Viele Code-Optimierungen können nicht durchgeführt werden

Regel: `volatile` wird nur in **begründeten Fällen** verwendet



- **Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
 - Digitaler Ausgang: Bitwert \mapsto Spannungspegel an μ C-Pin
 - Digitaler Eingang: Spannungspegel an μ C-Pin \mapsto Bitwert
 - Externer Interrupt: Spannungspegel an μ C-Pin \mapsto Bitwert
(bei Pegelwechsel) \leadsto Prozessor führt Interruptprogramm aus
- Die Funktion ist üblicherweise pro Pin konfigurierbar
 - Eingang
 - Ausgang
 - Externer Interrupt (nur bei bestimmten Eingängen)
 - Alternative Funktion (Pin wird von anderem Gerät verwendet)



Beispiel ATmega32: Port/Pin-Belegung

PDIP

(XCK/T0) PB0	□ 1	40	□ PA0 (ADC0)
(T1) PB1	□ 2	39	□ PA1 (ADC1)
(INT2/AIN0) PB2	□ 3	38	□ PA2 (ADC2)
(OC0/AIN1) PB3	□ 4	37	□ PA3 (ADC3)
(SS) PB4	□ 5	36	□ PA4 (ADC4)
(MOSI) PB5	□ 6	35	□ PA5 (ADC5)
(MISO) PB6	□ 7	34	□ PA6 (ADC6)
(SCK) PB7	□ 8	33	□ PA7 (ADC7)
RESET	□ 9	32	□ AREF
VCC	□ 10	31	□ GND
GND	□ 11	30	□ AVCC
XTAL2	□ 12	29	□ PC7 (TOSC2)
XTAL1	□ 13	28	□ PC6 (TOSC1)
(RXD) PD0	□ 14	27	□ PC5 (TDI)
(TXD) PD1	□ 15	26	□ PC4 (TDO)
(INT0) PD2	□ 16	25	□ PC3 (TMS)
(INT1) PD3	□ 17	24	□ PC2 (TCK)
(OC1B) PD4	□ 18	23	□ PC1 (SDA)
(OC1A) PD5	□ 19	22	□ PC0 (SCL)
(ICP1) PD6	□ 20	21	□ PD7 (OC2)

Aus **Kostengründen** ist nahezu jeder Pin **doppelt belegt**, die Konfiguration der gewünschten Funktion erfolgt durch die **Software**.

Beim SPiCboard werden z. B. **Pins 39–40 als ADCs konfiguriert**, um Poti und Photosensor anzuschließen.

PORTA steht daher **nicht zur Verfügung**.



Beispiel ATmega32: Port-Register

- Pro Port x sind drei Register definiert (Beispiel für $x = D$)

- **DDRx**

Data Direction Register: Legt für jeden Pin i fest, ob er als Eingang (Bit $i=0$) oder als Ausgang (Bit $i=1$) verwendet wird.

7	6	5	4	3	2	1	0
DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- **PORTx**

Data Register: Ist Pin i als Ausgang konfiguriert, so legt Bit i den Pegel fest (0=GND sink, 1=Vcc source). Ist Pin i als Eingang konfiguriert, so aktiviert Bit i den internen Pull-Up-Widerstand (1=aktiv).

7	6	5	4	3	2	1	0
PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- **PINx**

Input Register: Bit i repräsentiert den Pegel an Pin i (1=high, 0=low), unabhängig von der Konfiguration als Ein-/Ausgang.

7	6	5	4	3	2	1	0
PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
R	R	R	R	R	R	R	R

Verwendungsbeispiele: \hookrightarrow 3-5 und \hookrightarrow 3-8

[1, S. 66]



Strukturen: Motivation

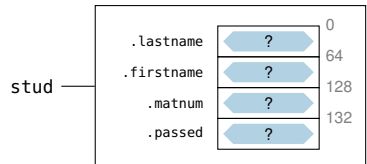
- Jeder Port wird durch *drei* globale Variablen verwaltet
 - Es wäre besser diese **zusammen zu fassen**
 - „problembezogene Abstraktionen“
 - „Trennung der Belange“
- Dies geht in C mit **Verbundtypen** (Strukturen)



```
// Structure declaration
struct Student {
    char    lastname[64];
    char    firstname[64];
    long    matnum;
    int     passed;
};

// Variable definition
struct Student stud;
```

Ein **Strukturtyp** fasst eine Menge von Daten zu einem gemeinsamen Typ zusammen.
Die Datenelemente werden **hintereinander** im Speicher abgelegt.



Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden

↪ 13-8

```
struct Student {  
    char    lastname[64];  
    char    firstname[64];  
    long    matnum;  
    int     passed;  
};
```

```
struct Student stud = { "Meier", "Hans",  
                        4711, 0 };
```

Die Initialisierer werden nur über ihre Reihenfolge, nicht über ihren Bezeichner zugewiesen.
↪ **Potentielle Fehlerquelle** bei Änderungen!

- Analog zur Definition von **enum**-Typen kann man mit **typedef** die Verwendung vereinfachen

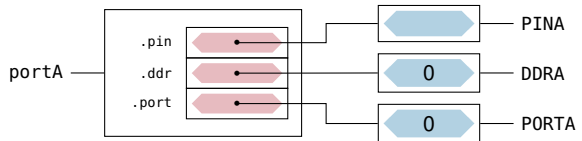
↪ 6-8

```
typedef struct {  
    volatile uint8_t *pin;  
    volatile uint8_t *ddr;  
    volatile uint8_t *port;  
} port_t;
```

```
port_t portA = { &PINA, &DDRA, &PORTA };  
port_t portD = { &PIND, &DDRD, &PORTD };
```



Strukturen: Elementzugriff



- Auf Strukturelemente wird mit dem `.`-Operator zugegriffen [\approx Java]

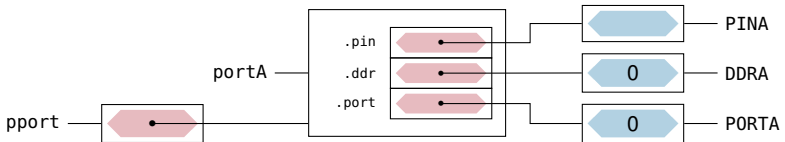
```
port_t portA = { &PINA, &DDRA, &PORTA };
```

```
*portA.port = 0;    // clear all pins  
*portA.ddr  = 0xff; // set all to input
```

Beachte: `.` hat eine höhere Priorität als `*`



Strukturen: Elementzugriff



- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

```
port_t * pport = &portA; // p --> portA

*(*pport).port = 0;      // clear all pins
*(*pport).ddr = 0xff;    // set all to output
```

- Mit dem `->`-Operator lässt sich dies vereinfachen $s \rightarrow m \equiv (*s).m$

```
port_t * pport = &portA; // p --> portA

pport->port = 0;          // clear all pins
pport->ddr = 0xff;        // set all to output
```

`->` hat **ebenfalls** eine höhere Priorität als `*`



Strukturen als Funktionsparameter

- Im Gegensatz zu Arrays werden Strukturen *by-value* übergeben

```
void initPort( port_t p ){
    *p.port = 0;          // clear all pins
    *p.ddd = 0xff;        // set all to output

    p.port = &PORTD;      // no effect, p is local variable
}

void main(){ initPort( portA ); ... }
```

- Bei größeren Strukturen wird das **sehr ineffizient**
 - Z. B. Student (↪ 14-15): Jedes mal 134 Byte allozieren und kopieren
 - Besser man übergibt einen Zeiger auf eine konstante Struktur

```
void initPort( const port_t *p ){
    *p->port = 0;          // clear all pins
    *p->ddd = 0xff;        // set all to output

    // p->port = &PORTD;  compile-time error, *p is const!
}

void main(){ initPort( &portA ); ... }
```



- Strukturelemente können auf Bit-Granularität festgelegt werden
 - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
 - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen

- Beispiel

- MCUCR **MCU Control Register:** Steuert Power-Management-Funktionen und Auslöser für externe Interrupt-Quellen INT0 und INT1. [1, S.36+69]

7	6	5	4	3	2	1	0
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

```
typedef struct {  
    uint8_t ISC0 : 2; // bit 0-1: interrupt sense control INT0  
    uint8_t ISC1 : 2; // bit 2-3: interrupt sense control INT1  
    uint8_t SM   : 3; // bit 4-6: sleep mode to enter on sleep  
    uint8_t SE   : 1; // bit 7 : sleep enable  
} MCUCR_t;
```



Unions

- In einer Struktur liegen die Elemente **hintereinander** im Speicher, in einer Union hingegen **übereinander**
 - Wert im Speicher lässt sich verschieden (Typ)-interpretieren
 - Nützlich für bitweise Typ-Casts
- Beispiel

↪ 14-15

```
void main(){
    union {
        uint16_t  val;
        uint8_t   bytes[2];
    } u;
```

```
    u.val = 0x4711;
```

```
    // show high-byte
```

```
    sb_7seg_showHexNumber( u.bytes[1] );
```

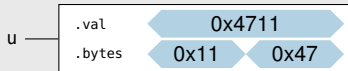
```
    ...
```

```
    // show low-byte
```

```
    sb_7seg_showHexNumber( u.bytes[0] );
```

```
    ...
```

```
}
```



47

11



Unions und Bit-Strukturen: Anwendungsbeispiel

- Unions werden oft mit Bit-Feldern kombiniert, um ein Register wahlweise „im Ganzen“ oder bitweise ansprechen zu können

```
typedef union {
    volatile uint8_t reg; // complete register
    volatile struct {
        uint8_t ISC0 : 2; // components
        uint8_t ISC1 : 2;
        uint8_t SM : 3;
        uint8_t SE : 1;
    };
} MCUCR_t;

void foo( void ) {
    MCUCR_t *mcucr = (MCUCR_t *) (0x35);
    uint8_t oldval = mcucr->reg; // save register
    ...
    mcucr->ISC0 = 2; // use register
    mcucr->SE = 1; // ...
    ...
    mcucr->reg = oldval; // restore register
}
```

