

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

Wintersemester 2011/2012

http://www4.informatik.uni-erlangen.de/Lehre/WS11/V_GSPIC

V_GSPIC_handout



Softwareentwurf

- 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
 - 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

[GDI, IV]

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

12-Module: 2012-03-27



© dl

GSPiC (Teil C, WS11)

12 Programmstruktur und Module | 12.1 Einführung

12-1

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

17 Zusammenfassung

Beispiel-Projekt: Eine Wetterstation

■ Typisches eingebettetes System

■ Mehrere Sensoren

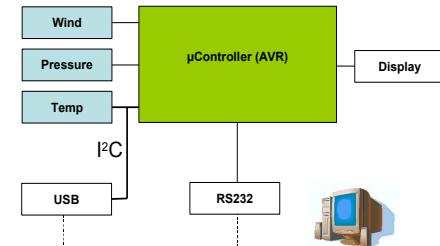
- Wind
- Luftdruck
- Temperatur

■ Mehrere Aktoren (hier: Ausgabegeräte)

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

■ Sensoren und Aktoren an den μ C angebunden über verschiedene Bussysteme

- I²C
- RS232



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

© dl

GSPiC (Teil C, WS11)

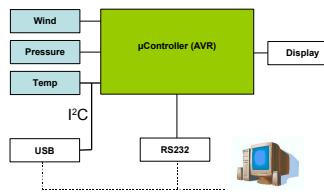
12 Programmstruktur und Module | 12.2 Funktionale Dekomposition

12-2

Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.1.1 I²C-Datenübertragung initialisieren
 - 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



12-Module: 2012-03-27

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 ↳ mangelhafte Trennung der Belange

Prinzip der Trennung der Belange

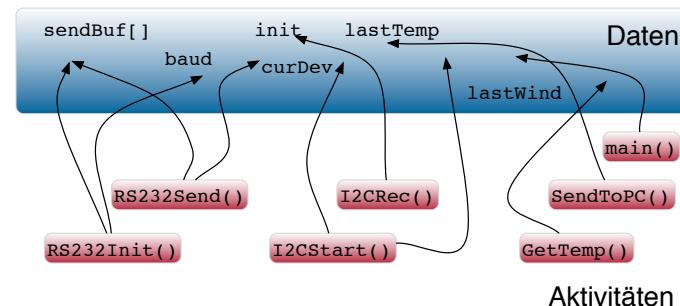
Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

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

12-Module: 2012-03-27

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 ↳ mangelhafte Trennung der Belange



12-Module: 2012-03-27

Zugriff auf Daten (Variablen)

- Variablen haben
 - 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	keine, auto static		Definition → Blockende Definition → Blockende	Definition → Blockende Programmstart → Programmende
Global	keine 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; // a: local to function (auto optional)
    // destroyed at end of block
    static int c = 11; // c: local to function, not destroyed
}
```

12-Module: 2012-03-27

Zugriff auf Daten (Variablen) (Forts.)

- 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

Konsequenz: Globale Variablen vermeiden!

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

Regel: Variablen erhalten stets die **geringstmögliche Sichtbarkeit und Lebensdauer**

12-Module: 2012-03-27



Was ist ein Modul?

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

Modul \hookrightarrow Abstraktion (\hookrightarrow 4-1)

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

12-Module: 2012-03-27



Lösung: Modularisierung

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

RS232.c

I2C.c

weather.c

12-Module: 2012-03-27

Module in C [\neq Java]

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

```
void RS232Init( uint16_t br );    RS232.h: Schnittstelle / Vertrag (öffentl.)
void RS232Send( char ch );        Deklaration der bereitgestellten
...                                Funktionen (und ggf. Daten)
```

RS232.c: Implementierung (nicht öffentl.)

```
#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;
  ...
}
```

Ggf. modulinterne Hilfsfunktionen und Daten (static)

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

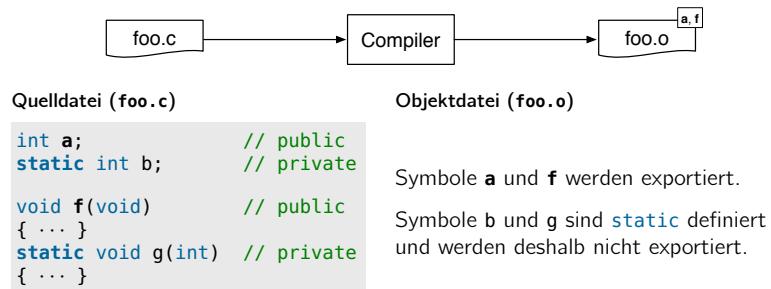
12-Module: 2012-03-27



Module in C – Export

[≠Java]

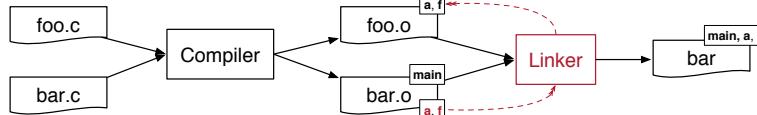
- Ein C-Modul **exportiert** eine Menge von definierten **Symbolen**
 - Alle Funktionen und globalen Variablen (→ „public“ in Java)
 - Export kann mit **static** unterbunden werden (→ „private“ in Java) (→ Einschränkung der Sichtbarkeit ↪ 12-5)
- Export erfolgt beim Übersetzungsvorgang (.c-Datei → .o-Datei)



Module in C – Import (Forts.)

[≠Java]

- Die eigentliche Auflösung erfolgt durch den **Linker** [↪ GDI, VI-158]



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



Module in C – Import

[≠Java]

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

Quelldatei (**bar.c**)

```

extern int a;          // declare
void f(void);          // declare

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

Objektdatei (**bar.o**)

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

Module in C – Header

[≠Java]

- Elemente aus fremden Modulen müssen deklariert werden
 - Funktionen durch normale Deklaration (↪ 9-7)
 - Globale Variablen durch **extern**
- Das **extern** unterscheidet eine Variablen Deklaration von einer Variablen Definition.

Das **extern** unterscheidet eine Variablen Deklaration von einer Variablen Definition.
- Die Deklarationen erfolgen sinnvollerweise in einer **Header-Datei**, die von der Modul entwicklerin 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)



Module in C – Header (Forts.)

[*Java*]

Modulschnittstelle: foo.h

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

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

#endif // _FOO_H
```

Modulimplementierung foo.c

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

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

Modulverwendung bar.c

(vergleiche ↵ 12-11)

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

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

12-Module: 2012-03-27



© dl GSPiC (Teil C, WS 11)

12 Programmstruktur und Module | 12.5 Module in C

12-Module: 2012-03-27



© dl GSPiC (Teil C, WS 11)

12 Programmstruktur und Module | 12.5 Module in C

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

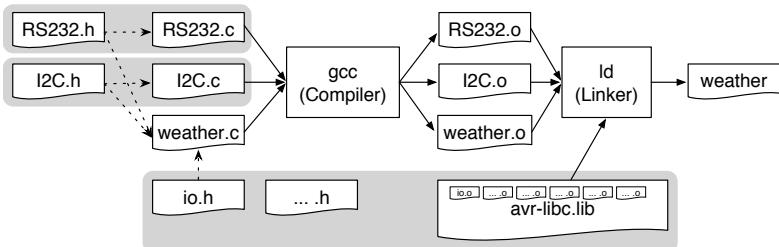
17 Zusammenfassung

V_GSPiC_handout



Zurück zum Beispiel: Wetterstation

Quellmodule — preprocess — compile —> Objektmodule — link —> ELF-Binary



- 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

© dl GSPiC (Teil C, WS 11)

12-15

Einordnung: Zeiger (*Pointer*)

■ Literal: 'a'

Darstellung eines Wertes

'a' ≡ 0110 0001

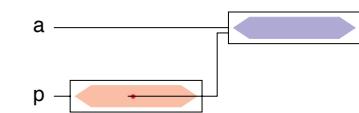


■ Variable: char a;

Behälter für einen Wert

■ Zeiger-Variable: char *p = &a;

Behälter für eine Referenz auf eine Variable



© dl GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.1 Zeiger – Einführung

13-1

13-Zeiger: 2012-01-19



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*)
 - Speicher lässt sich direkt ansprechen
 - Effizientere Programme
- „Effizienz durch Maschinennähe“
→ [3-14]
- Aber auch viele Probleme!
 - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
 - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

13-Zeiger: 2012-01-19



© dl

GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.1 Zeiger – Einführung

13-2

Adress- und Verweisoperatoren

- Adressoperator: **&x** Der unäre &-Operator liefert die **Referenz** (→ Adresse im Speicher) der Variablen **x**.
- Verweisoperator: ***y** Der unäre *-Operator liefert die **Zielvariable** (→ Speicherzelle / Behälter), auf die der Zeiger **y** verweist (Dereferenzierung).
- Es gilt: **(*(&x)) ≡ 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 Deklarationen
`typedef char* CPTR`
3. Verweis (unär): $x = *p1$ in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung

→ * wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.

13-Zeiger: 2012-01-19



© dl

GSPiC (Teil C, WS 11)

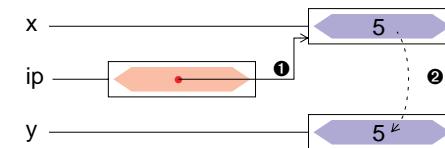
13 Zeiger und Felder | 13.2 Zeiger – Definition

13-4

Definition von Zeigervariablen

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

```
int x = 5;  
int *ip;  
int y;  
ip = &x; ①  
y = *ip; ②
```



13-Zeiger: 2012-01-19



© dl

GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.2 Zeiger – Definition

13-3

Zeiger als Funktionsargumente

- Parameter werden in C immer *by-value* übergeben
 - 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, II-89]
 - 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**

13-Zeiger: 2012-01-19



© dl

GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.3 Zeiger und Funktionen

13-5

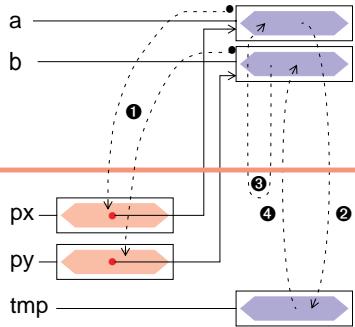
Zeiger als Funktionsargumente (Forts.)

Beispiel (Gesamtüberblick)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ①
    ...
}

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

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④
}
```



13-Zeiger: 2012-01-19



© dl GSPiC (Teil C, WS 11) 13 Zeiger und Felder | 13.3 Zeiger und Funktionen

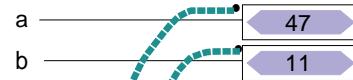
13-6

Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ①
    ...
}
```

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



13-Zeiger: 2012-01-19



© dl GSPiC (Teil C, WS 11) 13 Zeiger und Felder | 13.3 Zeiger und Funktionen

13-6

Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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



13-Zeiger: 2012-01-19



© dl GSPiC (Teil C, WS 11) 13 Zeiger und Felder | 13.3 Zeiger und Funktionen

13-6

Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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



13-Zeiger: 2012-01-19

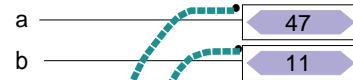


© dl GSPiC (Teil C, WS 11) 13 Zeiger und Felder | 13.3 Zeiger und Funktionen

13-6

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ①
```

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

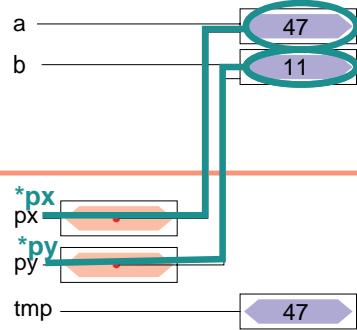


Zeiger als Funktionsargumente (Forts.)

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
```

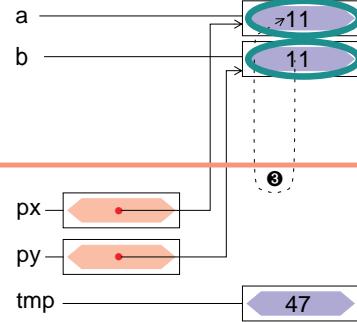


Zeiger als Funktionsargumente (Forts.)

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
```

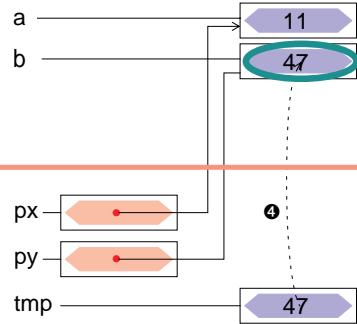


Zeiger als Funktionsargumente (Forts.)

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
}
```



Einordnung: Felder (Arrays)

[≈ Java]

- 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 (→ Anzahl der Elemente).

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 ];
}
```

[≠ Java]

[≠ Java]

[≠ Java]

[≠ Java]

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 };
```



Feldzugriff

- Syntax: *Feld [IntAusdruck]*

[=Java]

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

[≠Java]

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

- Beispiel

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

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

```
for( unit8_t i = 0; i < 4; ++i ) {
    sb_led_on( LEDs[ i ] );
}
```



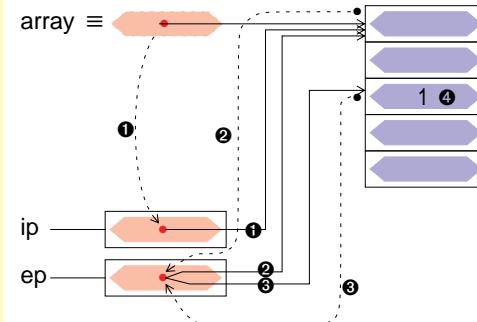
```
LEDs[ 4 ] = GREEN1; // UNDEFINED!!!
```



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 ~ Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

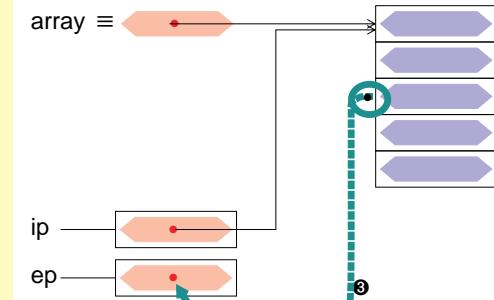
```
int array[5];
int *ip = array; ①
int *ep;
ep = &array[0]; ②
ep = &array[2]; ③
*ep = 1; ④
```



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 ~ Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

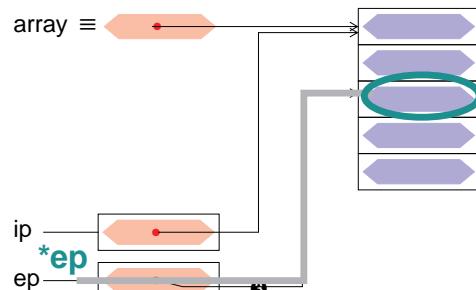
```
int array[5];
int *ip = array; ①
int *ep;
ep = &array[0]; ②
ep = &array[2]; ③
```



Felder sind Zeiger

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

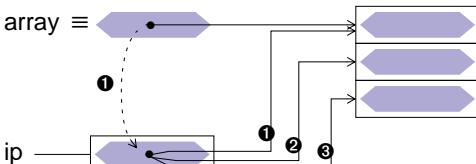
```
int array[5];
int *ip = array; ①
int *ep;
ep = &array[0]; ②
ep = &array[2]; ③
*ep = 1; ④
```



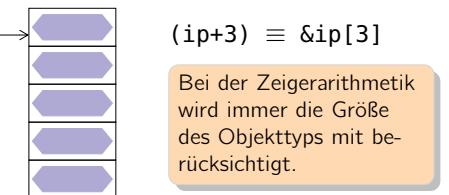
Rechnen mit Zeigern

- 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; ①
ip++; ②
ip++; ③
```



```
int array[5];
ip = array; ①
```



© dl GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.6 Zeigerarithmetik

13-12

Zeiger sind Felder

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

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };
LEDs[ 3 ] = BLUE1;
uint8_t *p = LEDs;
for( unit8_t i = 0; i < 4; ++i ) {
    sb_led_on( p[ i ] );
}
```



© dl GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.5 Syntaktische Äquivalenz

13-10

Zeigerarithmetik – Operationen

- Arithmetische Operationen
 - `++ Prä-/Postinkrement`
 - \rightsquigarrow Verschieben auf das nächste Objekt
 - `-- Prä-/Postdekrement`
 - \rightsquigarrow Verschieben auf das vorangegangene Objekt
 - `+, -` Addition / Subtraktion eines `int`-Wertes
 - \rightsquigarrow Ergebniszeiger ist verschoben um n Objekte
 - Subtraktion zweier Zeiger
 - \rightsquigarrow Anzahl der Objekte n zwischen beiden Zeigern (Distanz)
- Vergleichsoperationen: `<, <=, ==, >=, >, !=`
 - \rightsquigarrow Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen

\leftrightarrow [7-3]

© dl GSPiC (Teil C, WS 11)

13 Zeiger und Felder | 13.6 Zeigerarithmetik

13-13



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:

$$\begin{aligned} \text{array} &\equiv \&\text{array}[0] &\equiv \text{ip} &\equiv \&\text{ip}[0] \\ *\text{array} &\equiv \text{array}[0] &\equiv *\text{ip} &\equiv \text{ip}[0] \\ *(\text{array} + i) &\equiv \text{array}[i] &\equiv *(\text{ip} + i) &\equiv \text{ip}[i] \\ \text{array}++ &\not\equiv \text{ip}++ \end{aligned}$$

Fehler: array ist konstant!

- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.

Der Feldbezeichner kann aber **nicht verändert** werden.



Felder als Funktionsparameter (Forts.)

- Felder werden in C **immer** als Zeiger übergeben [=Java]
- Wird der Parameter als `const` deklariert, so kann die Funktion die Feldelemente **nicht verändern** ↪ Guter Stil!

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

- Um anzusehen, 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 Funktionparameters
- Bei Variablendefinitionen hat `array[]` eine **völlig andere** Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↪ 13-8)



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**)



Felder als Funktionsparameter (Forts.)

- 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" ≡ h a l l o ↪ 6-13

- Implementierungsvarianten

Variante 1: Feld-Syntax

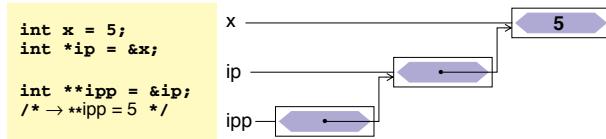
```
int strlen( const char s[] ) {
    int n=0;
    while( s[n] != 0 )
        n++;
    return n;
}
```

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



- 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 (Forts.)

- Syntax (Definition): `Typ (* Bezeichner)(FormaleParamopt);`
(sehr ähnlich zur Syntax von Funktionsdeklarationen) → [9-3]
 - Typ** Rückgabetyp 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, ..., Typn`
- 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



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
}

```

Zeiger auf Funktionen (Forts.)

- 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, BTONEVENT 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, BTNPRESSED, // for this button and events
        onButton              // invoke this function
    );
    sei();                      // enable interrupts (necessary!)
    while( 1 ) ;               // wait forever
}

```

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

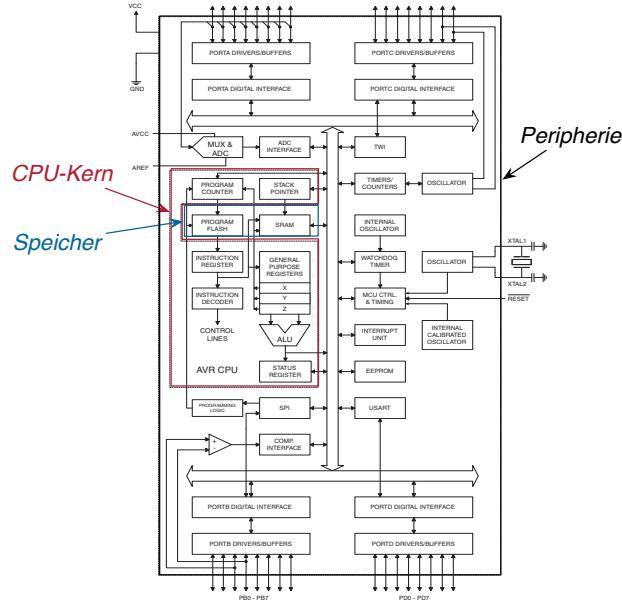
16 Speicherorganisation

17 Zusammenfassung

V_GSPIC_handout



Beispiel ATmega32: Blockschaltbild



14-MC-2012-01-19



Was ist ein µ-Controller?

- **µ-Controller** := Prozessor + Speicher + Peripherie
 - Faktisch ein Ein-Chip-Computersystem → SoC (*System-on-a-Chip*)
 - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher ↵ 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 ↔ µC ↔ SoC
 - AMD64-CPUs haben ebenfalls eingebaute Timer, Speicher (Caches), ...
 - Einige µC erreichen die Geschwindigkeit „großer Prozessoren“

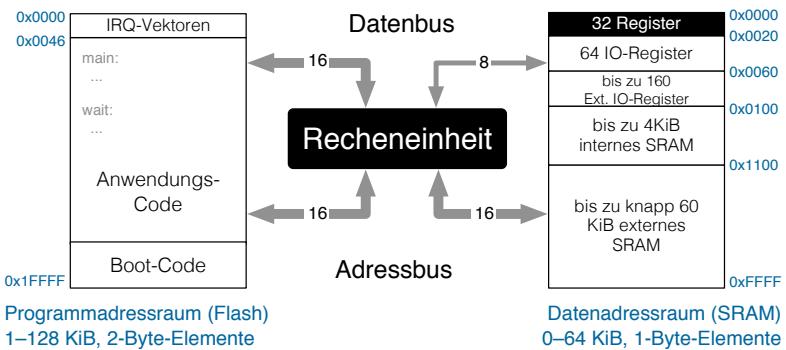
14-MC-2012-01-19



© dl GSPIC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.1 Überblick

14-1

Beispiel ATmega-Familie: CPU-Architektur



14-MC-2012-01-19



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

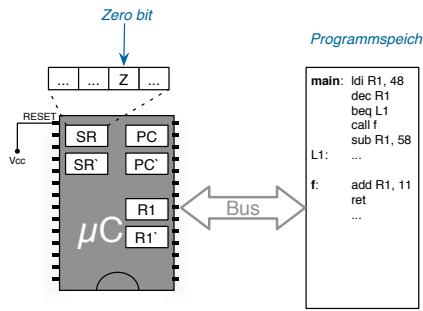
Zum Vergleich: PC basiert auf von-Neumann-Architektur [→ GDI, VI-6] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

© dl GSPIC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.2 Architektur

14-3

Wie arbeitet ein Prozessor?

14-MC-2012-01-19



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

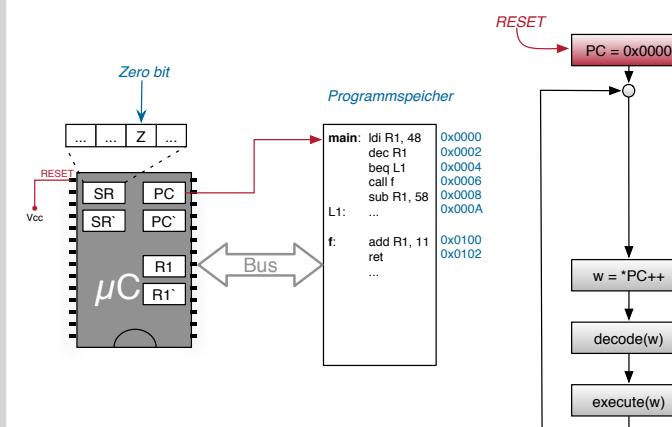


© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.2 Architektur

14-4

Wie arbeitet ein Prozessor?

14-MC-2012-01-19

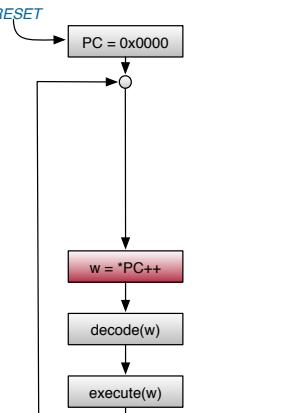
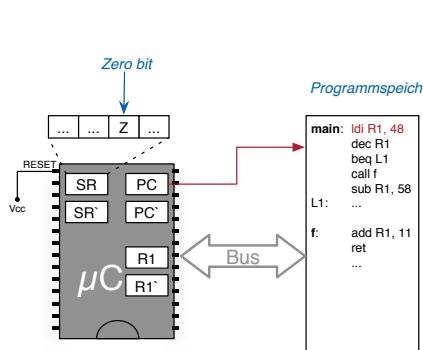


© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.2 Architektur

14-4

Wie arbeitet ein Prozessor?

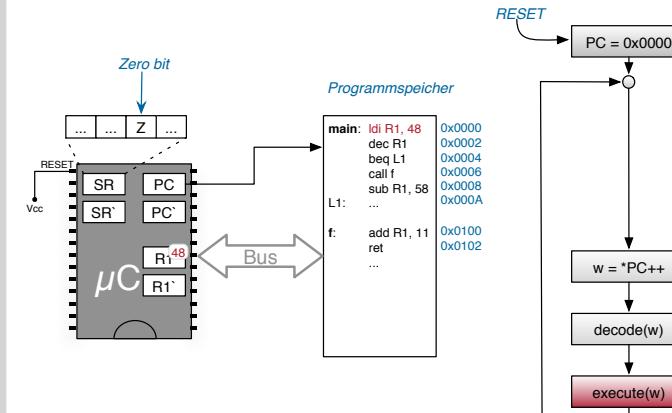
14-MC-2012-01-19



14-4

Wie arbeitet ein Prozessor?

14-MC-2012-01-19

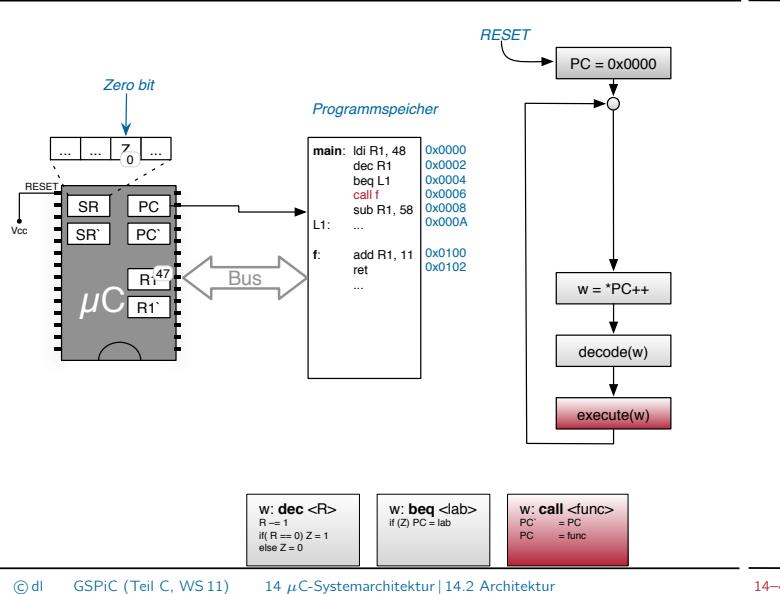


© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.2 Architektur

14-4

Wie arbeitet ein Prozessor?

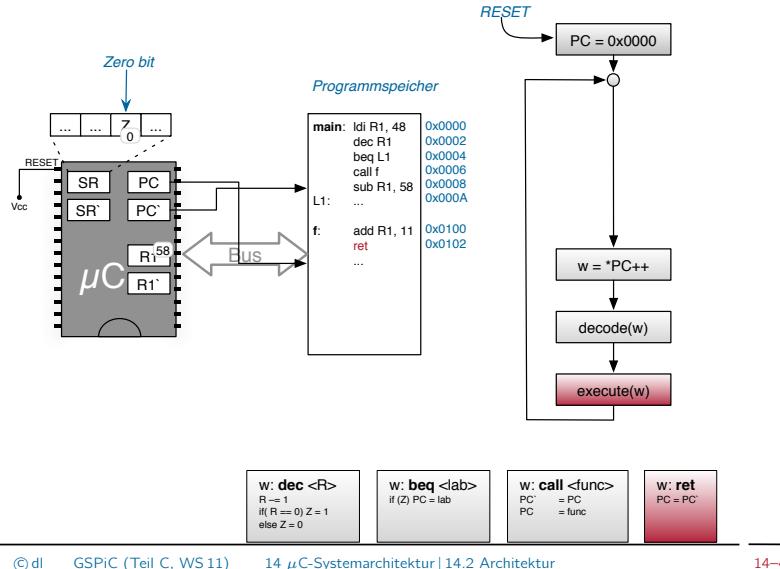
14-MC-2012-01-19



14-4

Wie arbeitet ein Prozessor?

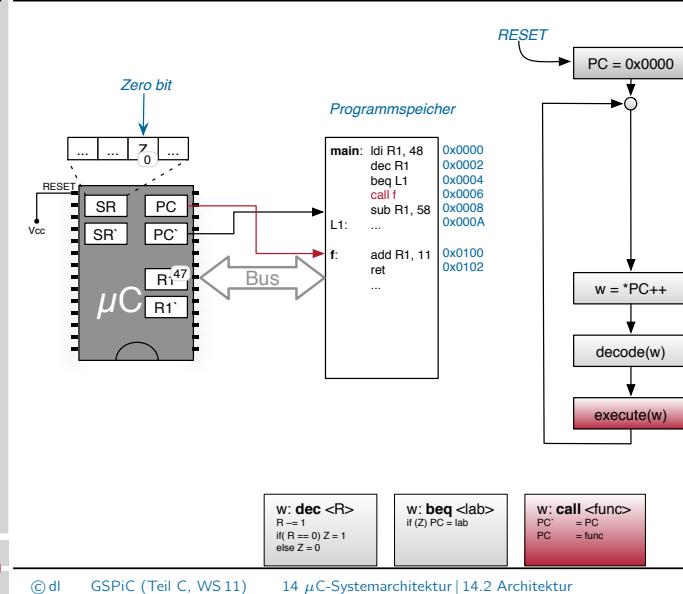
14-MC-2012-01-19



14-4

Wie arbeitet ein Prozessor?

14-MC-2012-01-19



14-4

Peripheriegeräte

- **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

14-MC-2012-01-19



14-5

Peripheriegeräte: Beispiele

- 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 Zählerwert 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 (pseude-analoge Ausgabe).
 - Ports Gruppen von üblicherweise 8 Anschlüssen, die auf GND oder V_{cc} gesetzt werden können oder deren Zustand abgefragt werden kann. \rightarrow [14-12]



Peripheriegeräte – Register (Forts.)

- Memory-mapped Register ermöglichen einen komfortablen Zugriff
 - Register \rightarrow Speicher \rightarrow 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 ) )
          ^ Adresse: int
          ^ Adresse: volatile uint8_t* (Cast  $\rightarrow$  [7-1])
          | Wert: volatile uint8_t (Dereferenzierung  $\leftrightarrow$  [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
```



Peripheriegeräte – Register

- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
 - Memory-mapped: Register sind in den Adressraum eingebettet; der Zugriff erfolgt über die Speicherbefehle des Prozessors (**load**, **store**)
 - Port-basiert: (x86-basierte PCs) Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle **in**- und **out**-Befehle
- 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
\$37 (\$3F)	SREG	I	T	H	S	N	Z	C		8
\$38 (\$3E)	SPH	—	—	—	—	SP11	SP10	SP9	SP8	11
\$39 (\$3D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
\$3C (\$3C)	OCRD	Timer/Counter Output Compare Register								
\$12 (\$32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
\$11 (\$31)	DORD	DD07	DD06	DD05	DD04	DD03	DD02	DD01	DD00	67
\$10 (\$30)	PIN0	PIN07	PIN06	PIN05	PIN04	PIN03	PIN02	PIN01	PIN00	68

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

<pre>// C code #define PIND (*(uint8_t*)(0x10)) void foo(void) { ... if(!(PIND & 0x2)) { // button0 pressed ... } if(!(PIND & 0x4)) { // button 1 pressed ... } }</pre>	<pre>// Resulting assembly code foo: lds r24, 0x0010 // PIND->r24 sbrc r24, 1 // test bit 1 rjmp L1 // button0 pressed ... lds r24, 0x0010 // PIND->r24 sbrc r24, 2 // test bit 2 rjmp L2 // button 1 pressed ... ret</pre>
	<div style="border: 1px solid orange; padding: 2px;">PIND wird nicht erneut aus dem Speicher geladen. Der Compiler nimmt an, dass der Wert in r24 aktuell ist.</div>

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

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



Peripheriegeräte: Ports

- Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
 - Digitaler Ausgang: Bitwert → Spannungspegel an µC-Pin
 - Digitaler Eingang: Spannungspegel an µC-Pin → Bitwert
 - Externer Interrupt: Spannungspegel an µC-Pin → Bitwert
(bei Pegelwechsel)
→ 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)



Der volatile-Typmodifizierer (Forts.)

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

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

volatile!

wait:
// compiler has optimized
// "nonsensical" loop away
ret

Achtung: volatile → \$\$\$

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



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 33–49 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							
 - PORTx** **Data Register:** Ist Pin i als Ausgang konfiguriert, so legt Bit i den Pegel fest ($0=\text{GND sink}$, $1=\text{Vcc source}$). Ist Pin i als Eingang konfiguriert, so aktiviert Bit i den internen Pull-Up-Widerstand ($1=\text{aktiv}$).

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

 - PINx** **Input Register:** Bit i repräsentiert den Pegel an Pin i ($1=\text{high}$, $0=\text{low}$), 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: ↪ 3-5 und ↪ 3-8 [1, S. 66]

© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.5 Ports 14-14



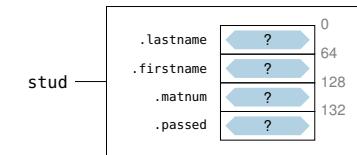
Strukturen: Motivation

- 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;
```

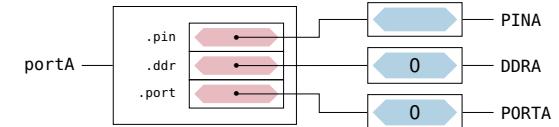
Ein Strukturtyp fasst eine Menge von Daten zu einem gemeinsamen Typ zusammen.
Die Datenelemente werden **hintereinander** im Speicher abgelegt.



Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden
 - 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
 - typedef struct {**
volatile uint8_t *pin;
volatile uint8_t *ddr;
volatile uint8_t *port;
} port_t;
 - port_t portA = { &PINA, &DDRA, &PORTA };**
 - *portA.port = 0; // clear all pins**
***portA.ddr = 0xff; // set all to input**

© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Exkurs: Verbundtypen (struct, union) 14-15

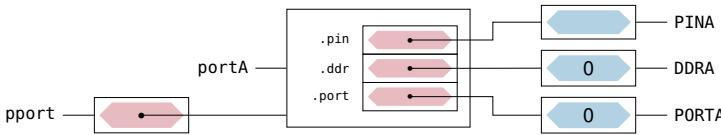


- Auf Strukturelemente wird mit dem **.**-Operator zugegriffen [≈Java]

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

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 *

14-MC-2012-01-19



© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Exkurs: Verbundtypen (struct, union) 14-18

Bit-Strukturen: Bitfelder

- 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 MCUCR 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;
```

14-MC-2012-01-19



© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Exkurs: Verbundtypen (struct, union) 14-20

Strukturen als Funktionsparameter

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

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

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

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

14-MC-2012-01-19



© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Exkurs: Verbundtypen (struct, union) 14-19

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

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

    u.val = 0x4711;
    u.bytes[1] = 0x47;
    u.bytes[0] = 0x11;

    // show high-byte
    sb_7seg_showHexNumber( u.bytes[1] );
    ...
    // show low-byte
    sb_7seg_showHexNumber( u.bytes[0] );
    ...
}
```

14-MC-2012-01-19



© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Exkurs: Verbundtypen (struct, union) 14-21

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
}
```

14-MC-2012-01-19



© dl GSPiC (Teil C, WS 11) 14 µC-Systemarchitektur | 14.6 Excurs: Verbundtypen (struct, union) 14-22

Ereignisbehandlung

- Bei einem Peripheriegerät tritt ein Ereignis (⚡) auf
 - Signal an einem Port-Pin wechselt von *low* auf *high*
 - Ein *Timer* ist abgelaufen
 - Ein A/D-Wandler hat einen neuen Wert vorliegen
 - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
 - **Polling:** Das Programm überprüft den Zustand regelmäßig und ruft ggf. eine Bearbeitungsfunktion auf.
 - **Interrupt:** Gerät „meldet“ sich beim Prozessor, der daraufhin in eine Bearbeitungsfunktion verzweigt.

15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.1 Interrupts: Einführung

15-1

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

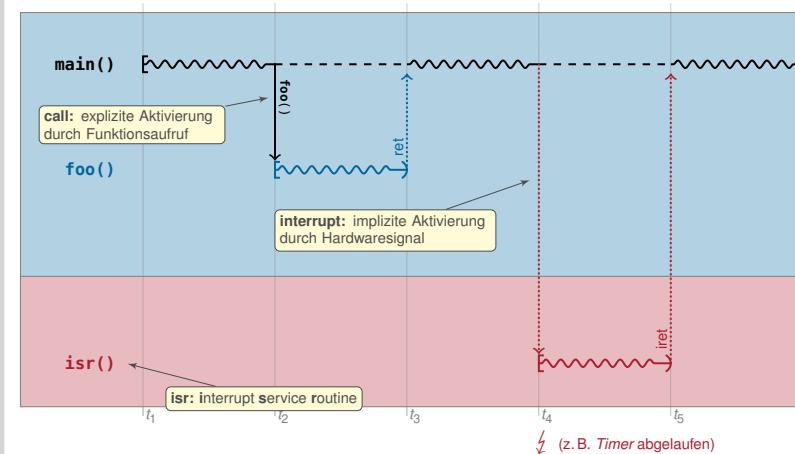
16 Speicherorganisation

17 Zusammenfassung

V_GSPiC_handout



Interrupt → Funktionsaufruf „von außen“



15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.1 Interrupts: Einführung

15-2

Polling vs. Interrupts – Vor- und Nachteile

- Polling (→ „Periodisches / zeitgesteuertes System“)
 - Ereignisbearbeitung erfolgt **synchron** zum Programmablauf
 - Ereigniserkennung über das Programm „verstreut“ (Trennung der Belange)
 - Hochfrequentes Pollen → hohe Prozessorlast → **hoher Energieverbrauch**
 - + Implizite Datenkonsistenz durch festen, sequentiellen Programmablauf
 - + Programmverhalten gut vorhersagbar

- Interrupts (→ „Ereignisgesteuertes System“)
 - Ereignisbearbeitung erfolgt **asynchron** zum Programmablauf
 - + Ereignisbearbeitung kann im Programmtext gut separiert werden
 - + Prozessor wird nur beansprucht, wenn Ereignis tatsächlich eintritt
 - Höhere Komplexität durch Nebenläufigkeit → Synchronisation erforderlich
 - Programmverhalten **schwer vorhersagbar**

Beide Verfahren bieten spezifische Vor- und Nachteile
→ Auswahl anhand des konkreten Anwendungsszenarios

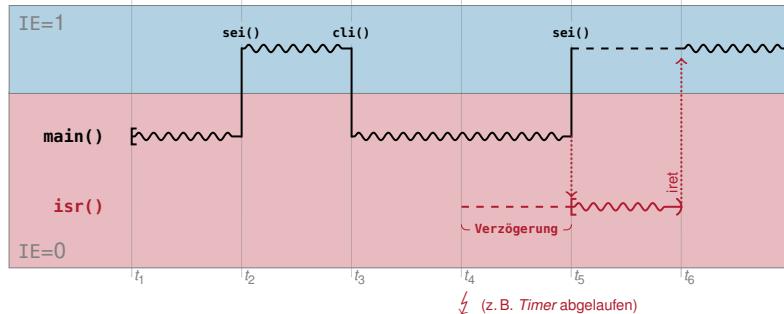
15-Irq: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.1 Interrupts: Einführung

15-3

Interruptsperren: Beispiel



- t₁ Zu Beginn von main() sind IRQs gesperrt (IE=0)
- t₂, t₃ Mit sei() / cli() werden IRQs freigegeben (IE=1) / erneut gesperrt
- t₄ ↳ aber IE=0 → Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t₅ main() gibt IRQs frei (IE=1) → gepufferter IRQ „schlägt durch“
- t₅–t₆ Während der ISR-Bearbeitung sind die IRQs gesperrt (IE=0)
- t₆ Unterbrochenes main() wird fortgesetzt

15-Irq: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

15-5

Interruptsperren

- Zustellung von Interrupts kann softwareseitig **gesperrt** werden
 - Wird benötigt zur **Synchronisation** mit ISRs
 - Einzelne ISR: Bit in gerätespezifischem Steuerregister
 - Alle ISRs: Bit (**IE**, *Interrupt Enable*) im Statusregister der CPU

- Auflaufende IRQs werden (üblicherweise) gepuffert
 - Maximal einer pro Quelle!
 - Bei längeren Sperrzeiten können IRQs verloren gehen!

- Das **IE**-Bit wird beeinflusst durch:
 - Prozessor-Befehle: cli: IE←0 (*clear interrupt*, IRQs gesperrt)
sei: IE←1 (*set interrupt*, IRQs erlaubt)
 - Nach einem RESET: IE=0 → IRQs sind zu Beginn des Hauptprogramms gesperrt
 - Bei Betreten einer ISR: IE=0 → IRQs sind während der Interruptbearbeitung gesperrt

IRQ → *Interrupt ReQuest*

15-Irq: 2011-09-28

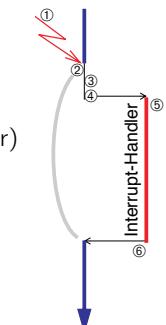


© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

15-4

Ablauf eines Interrupts – Überblick

- ① Gerät signalisiert Interrupt
 - Anwendungsprogramm wird „unmittelbar“ (vor dem nächsten Maschinenbefehl mit IE=1) unterbrochen
- ② Die Zustellung weiterer Interrupts wird gesperrt (IE=0)
 - Zwischenzeitlich auflaufende Interrupts werden gepuffert (maximal einer pro Quelle!)
- ③ Registerinhalte werden gesichert (z. B. im Datenspeicher)
 - PC und Statusregister automatisch von der Hardware
 - Vielzweckregister müssen oft manuell gesichert werden
- ④ Aufzurufende ISR (Interrupt-Handler) wird ermittelt
- ⑤ ISR wird ausgeführt
- ⑥ ISR terminiert mit einem „return from interrupt“-Befehl
 - Registerinhalte werden restauriert
 - Zustellung von Interrupts wird freigegeben (IE=1)
 - Das Anwendungsprogramm wird fortgesetzt



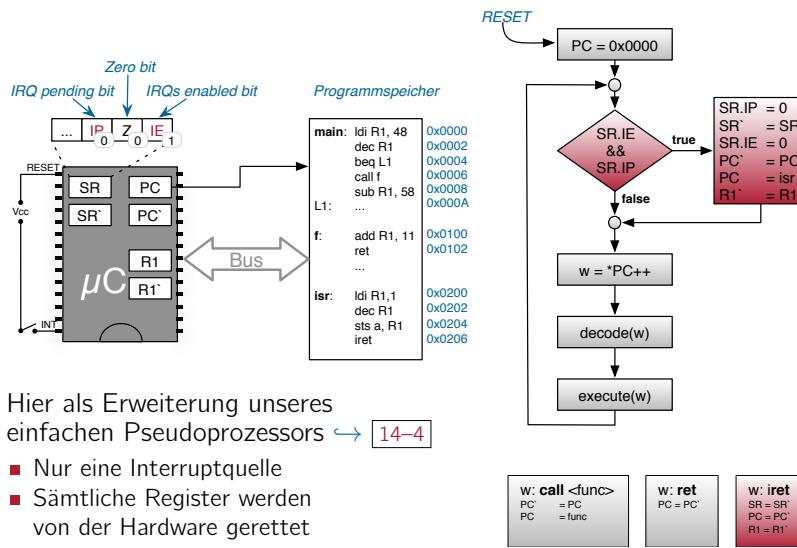
15-Irq: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

15-6

Ablauf eines Interrupts – Details

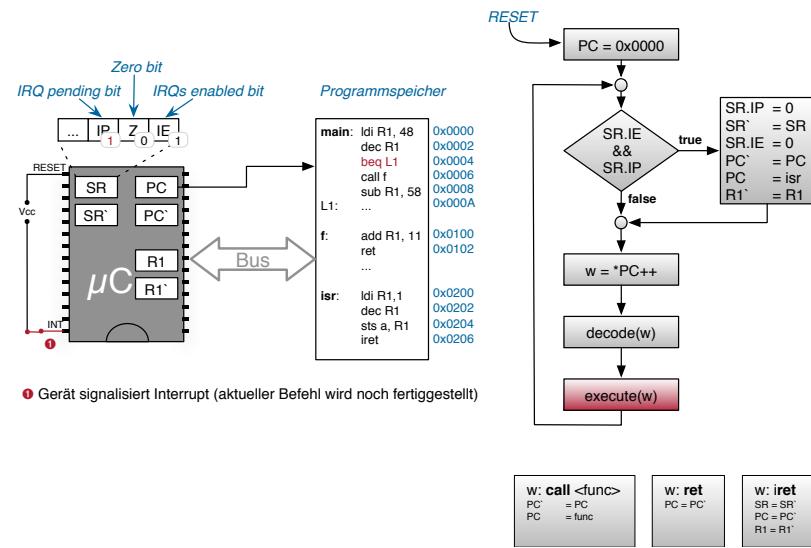


Hier als Erweiterung unseres einfachen Pseudoprozessors ↗

- Nur eine Interruptquelle
- Sämtliche Register werden von der Hardware gerettet

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

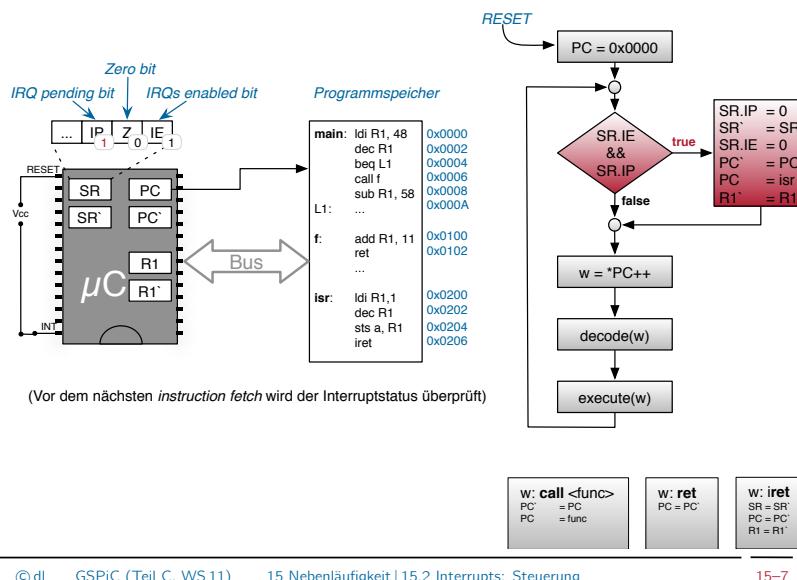
Ablauf eines Interrupts – Details



- ① Gerät signalisiert Interrupt (aktueller Befehl wird noch fertiggestellt)

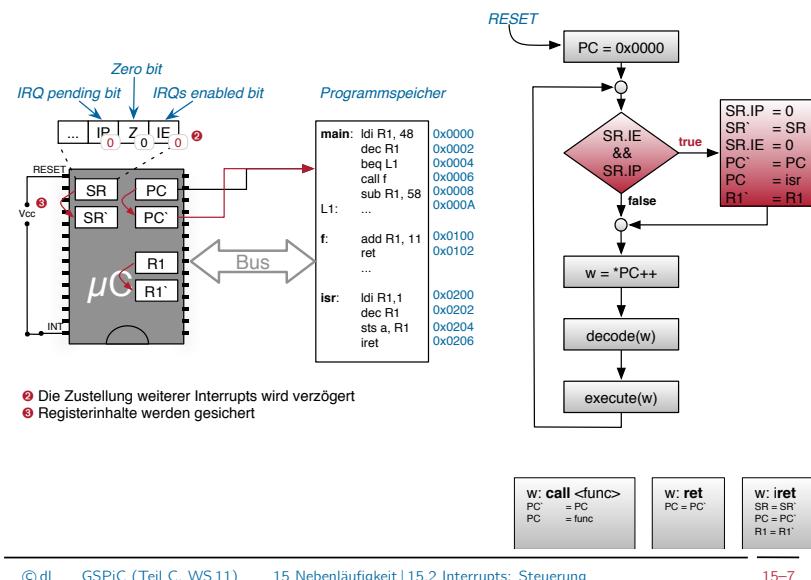
© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

Ablauf eines Interrupts – Details



(Vor dem nächsten *instruction fetch* wird der Interruptstatus überprüft)

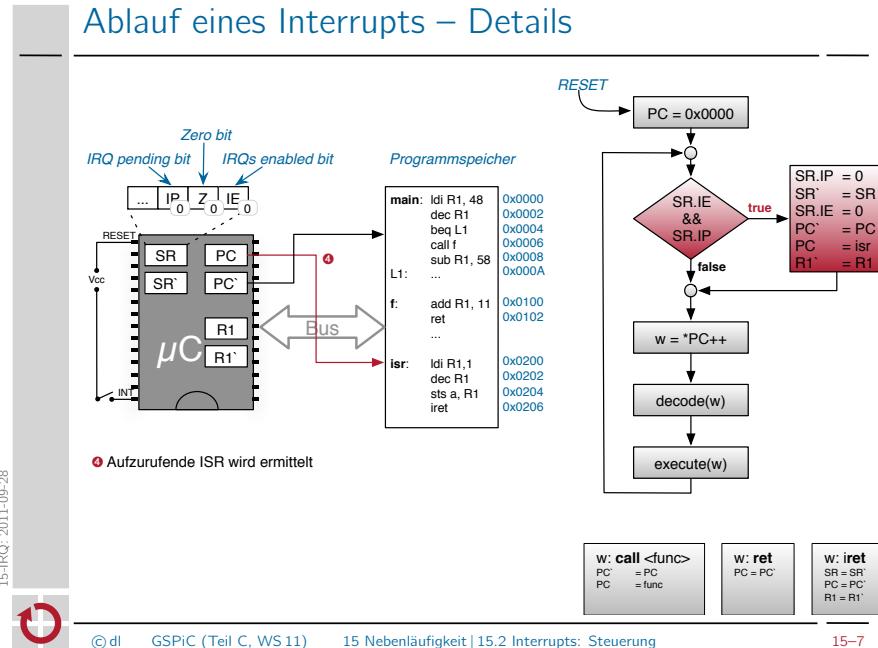
Ablauf eines Interrupts – Details



- ② Die Zustellung weiterer Interrupts wird verzögert
- ③ Registerinhalte werden gesichert

Ablauf eines Interrupts – Details

15-Irq: 2011-09-28

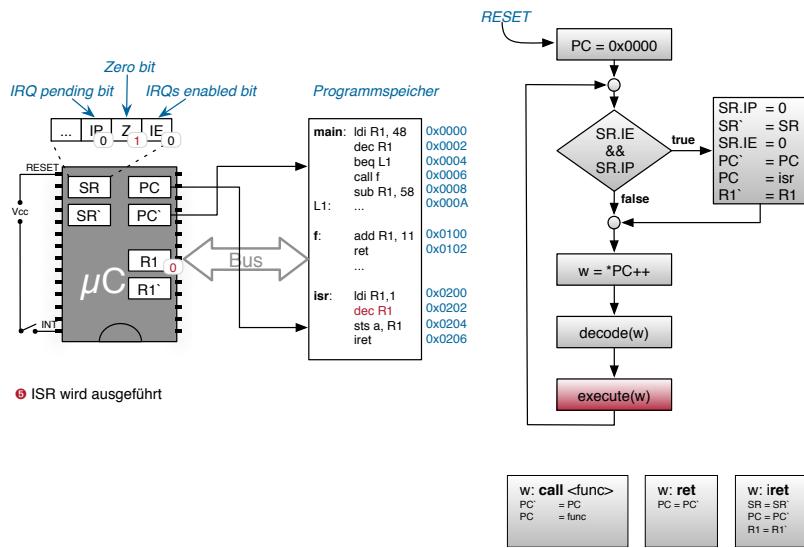


① Aufzurufende ISR wird ermittelt

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

Ablauf eines Interrupts – Details

15-Irq: 2011-09-28

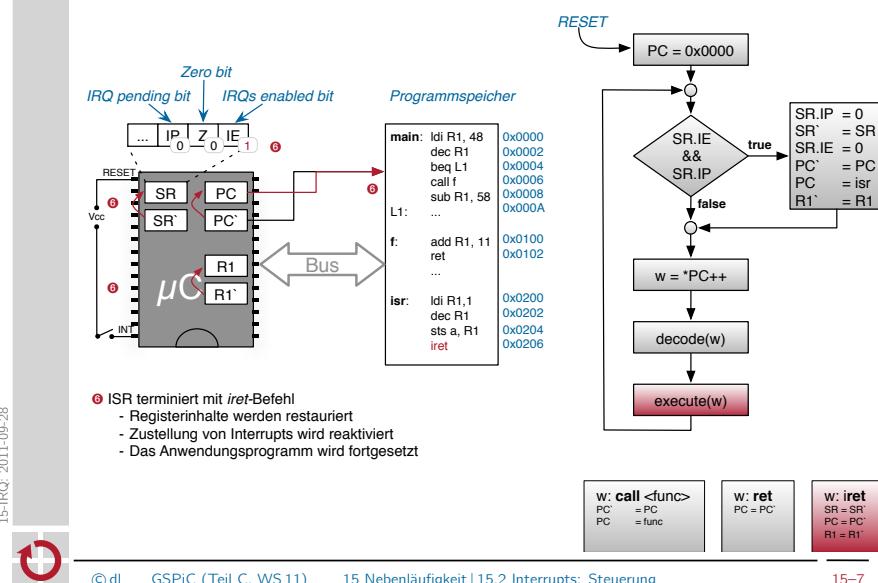


② ISR wird ausgeführt

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

Ablauf eines Interrupts – Details

15-Irq: 2011-09-28

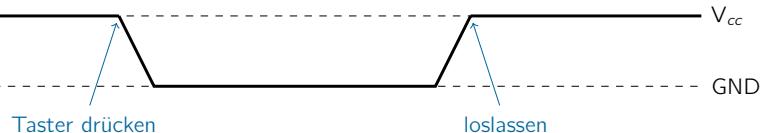


③ ISR terminiert mit iret-Befehl
 - Registerhalte werden restauriert
 - Zustellung von Interrupts wird reaktiviert
 - Das Anwendungsprogramm wird fortgesetzt

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines idealisierten Tasters (active low)



- Flankengesteuerter Interrupt

- Interrupt wird durch den Pegelwechsel (Flanke) ausgelöst
- Häufig ist konfigurierbar, welche Flanke (steigend/fallend/beide) einen Interrupt auslösen soll

- Pegelgesteueter Interrupt

- Interrupt wird immer wieder ausgelöst, so lange der Pegel anliegt

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.2 Interrupts: Steuerung

15-8

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega32
 - 21 IRQ-Quellen
 - einzeln de-/aktivierbar
 - IRQ ↼ Sprung an Vektor-Adresse
- Verschaltung SPiCboard
 - (↪ 14-14) ↪ 17-4)
 - INT0 ↪ PD2 ↪ Button0 (hardwareseitig entprellt)
 - INT1 ↪ PD3 ↪ Button1

Vector No.	Program Address ⁽¹⁾	Source	Interrupt Definition
1	\$000 ⁽¹⁾	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$002	INT0	External Interrupt Request 0
3	\$004	INT1	External Interrupt Request 1
4	\$006	INT2	External Interrupt Request 2
5	\$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	\$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	\$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	\$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	\$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	\$012	TIMER1 OVF	Timer/Counter1 Overflow
11	\$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	\$016	TIMER0 OVF	Timer/Counter0 Overflow
13	\$018	SPI_STC	Serial Transfer Complete
14	\$01A	USART_RXC	USART, Rx Complete
15	\$01C	USART_UDRE	USART Data Register Empty
16	\$01E	USART_TXC	USART, Tx Complete
17	\$020	ADC	ADC Conversion Complete
18	\$022	EE_RDY	EEPROM Ready
19	\$024	ANA_COMP	Analog Comparator
20	\$026	TWI	Two-wire Serial Interface
21	\$028	SPM_RDY	Store Program Memory Ready

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.3 Interrupts: Beispiel ATmega

15-9

Externe Interrupts: Verwendung

- **Schritt 1:** Installation der **Interrupt-Service-Routine**
 - ISR in Hochsprache ↼ Registerinhalte sichern und wiederherstellen
 - Unterstützung durch die avrlibc: Makro **ISR(SOURCE_vect)** (Modul avr/interrupt.h)

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

ISR( INT1_vect ) { // invoked for every INT1 IRQ
    static uint8_t counter = 0;
    sb_7seg_showNumber( counter++ );
    if( counter == 100 ) counter = 0;
}

void main() {
    ...
    // setup
}
```

Externe Interrupts: Register

- Steuerregister für INT0 und INT1

- **GICR**

General Interrupt Control Register: Legt fest, ob die Quellen INT*i* IRQs auslösen (Bit INT*i*=1) oder deaktiviert sind (Bit INT*i*=0) [1, S. 71]

INT1	INT0	INT2	-	-	-	IVSEL	IVCE
R/W	R/W	R/W	R	R	R	R/W	R/W

- **MCUCR**

MCU Control Register: Legt für externe Interrupts INT0 und INT1 fest, wodurch ein IRQ ausgelöst wird (Flanken-/Pegelsteuerung) [1, S. 69]

SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Jeweils zwei **Interrupt-Sense-Control-Bits** (ISC*i*0 und ISC*i*1) steuern dabei die Auslöser (Tabelle für INT1, für INT0 gilt entsprechendes):

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.3 Interrupts: Beispiel ATmega

15-10

Externe Interrupts: Verwendung (Forts.)

- **Schritt 2:** Konfigurieren der **Interrupt-Steuerung**

- Steuerregister dem Wunsch entsprechend initialisieren
- Unterstützung durch die avrlibc: Makros für Bit-Indizes (Modul avr/interrupt.h und avr/io.h)

```
...
void main() {
    DDRD &= ~(1<<PD3); // PD3: input with pull-up
    PORTD |= (1<<PD3);
    MCUCR &= ~(1<<ISC10 | 1<<ISC11); // INT1: IRQ on level=low
    GICR |= (1<<INT1); // INT1: enable
    ...
    sei(); // global IRQ enable
}
```

- **Schritt 3:** Interrupts global zulassen

- Nach Abschluss der Geräteinitialisierung
- Unterstützung durch die avrlibc: Befehl **sei()** (Modul avr/interrupt.h)

Externe Interrupts: Verwendung (Forts.)

Schritt 4: Wenn nichts zu tun, den Stromsparmodus betreten

- Die sleep-Instruktion hält die CPU an, bis ein IRQ eintrifft
 - In diesem Zustand wird nur sehr wenig Strom verbraucht
- Unterstützung durch die avrlibc (Modul `avr/sleep.h`):
 - `sleep_enable()` / `sleep_disable()`: Sleep-Modus erlauben / verbieten
 - `sleep_cpu()`: Sleep-Modus betreten



```
#include <avr/sleep.h>
...
void main() {
    ...
    sei();                                // global IRQ enable
    while(1) {
        sleep_enable();
        sleep_cpu();                      // wait for IRQ
        sleep_disable();
    }
}
```

Atmel empfiehlt die Verwendung von `sleep_enable()` und `sleep_disable()` in dieser Form, um das Risiko eines „versehentlichen“ Betreten des Sleep-Zustands (z. B. durch Programmierfehler oder Bit-Kipper in der Hardware) zu minimieren.

Nebenläufigkeit

Definition: Nebenläufigkeit

Zwei Programmausführungen A und B sind nebenläufig ($A|B$), wenn für einzelne Instruktionen a aus A und b aus B nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird (a, b oder b, a).

Nebenläufigkeit tritt auf durch

- Interrupts
 - ~ IRQs können ein Programm an „beliebiger Stelle“ unterbrechen
- Echt-parallele Abläufe (durch die Hardware)
 - ~ andere CPU / Peripherie greift „jederzeit“ auf den Speicher zu
- Quasi-parallele Abläufe (z. B. Fäden in einem Betriebssystem)
 - ~ Betriebssystem kann „jederzeit“ den Prozessor entziehen

Problem: Nebenläufige Zugriffe auf gemeinsamen Zustand

Nebenläufigkeitsprobleme

Szenario

- Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
- Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
static volatile uint16_t cars;           // photo sensor is connected
                                         // to INT2
void main() {
    while(1) {
        waitsec( 60 );
        send( cars );
        cars = 0;
    }
}
ISR(INT2_vect){
    cars++;
}
```

Wo ist hier das Problem?

- Sowohl `main()` als auch ISR **lesen und schreiben** `cars`
 - ~ Potentielle *Lost-Update*-Anomalie
- Größe der Variable `cars` **übersteigt die Registerbreite**
 - ~ Potentielle *Read-Write*-Anomalie

Nebenläufigkeitsprobleme (Forts.)

Wo sind hier die Probleme?

- Lost-Update:** Sowohl `main()` als auch ISR lesen und schreiben `cars`
- Read-Write:** Größe der Variable `cars` übersteigt die Registerbreite

Wird oft erst auf der Assemblerebene deutlich

```
void main() {
    ...
    send( cars );
    cars = 0;
}
// photosensor is connected
// to INT2
ISR(INT2_vect){
    cars++;
}

main:
...
lds r24,cars
lds r25,cars+1
rcall send
sts cars+1,__zero_reg__
sts cars,__zero_reg__
...
INT2_vect:
...
lds r24,cars      ; save regs
lds r24,cars      ; load cars.lo
lds r25,cars+1   ; load cars.hi
adiw r24,1        ; add (16 bit)
sts r25,r25      ; store cars.hi
sts cars,r24      ; store cars.lo
...
; restore regs
```



Nebenläufigkeitsprobleme: Lost-Update-Anomalie

```
main:
...
lds r24,cars lds r25,cars+1
rcall send
sts cars+1,__zero_reg__ sts cars
...
INT2_vect:
...
lds r24,cars lds r25,cars+1 adiw
...
; save regs
; restore regs
```

- Sei $\text{cars}=5$ und an **dieser Stelle** tritt der IRQ (⌚) auf
 - main hat den Wert von cars (5) bereits in Register gelesen (Register ↔ lokale Variable)
 - INT2_vect wird ausgeführt
 - Register werden gerettet
 - cars wird inkrementiert ↪ cars=6
 - Register werden wiederhergestellt
 - main übergibt den **veralteten Wert** von cars (5) an send
 - main nullt cars ↪ **1 Auto ist „verloren“ gegangen**

15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-17

Interruptsperren: Datenflussanomalien verhindern

```
void main() {
    while(1) {
        waitsec( 60 );
        cli();
        send( cars );
        cars = 0;           kritisches Gebiet
        sei();
    }
}
```

- Wo genau ist das **kritische Gebiet**?
 - Lesen von cars und Nullen von cars müssen atomar ausgeführt werden
 - Dies kann hier mit **Interruptsperren** erreicht werden
 - ISR unterbricht main, aber nie umgekehrt ↪ asymmetrische Synchronisation
 - Achtung: Interruptsperren sollten **so kurz wie möglich** sein
 - Wie lange braucht die Funktion send hier?
 - Kann man send aus dem kritischen Gebiet herausziehen?

15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-19

Nebenläufigkeitsprobleme: Read-Write-Anomalie

```
main:
...
lds r24,cars
lds r25,cars+1
rcall send
sts cars+1,__zero_reg__
sts cars,__zero_reg__
...
INT2_vect:
...
lds r24,cars lds r25,cars+1 adiw
...
; save regs
; restore regs
```

- Sei $\text{cars}=255$ und an **dieser Stelle** tritt der IRQ (⌚) auf
 - main hat bereits cars=255 Autos mit send gemeldet
 - main hat bereits das **High-Byte** von cars genutzt
 - ↪ cars=255, cars.lo=255, cars.hi=0
 - INT2_vect wird ausgeführt
 - cars wird gelesen und inkrementiert, **Überlauf ins High-Byte**
 - ↪ cars=256, cars.lo=0, cars.hi=1
 - main nullt das **Low-Byte** von cars
 - ↪ cars=256, cars.lo=0, cars.hi=1
 - ↪ Beim nächsten send werden **255 Autos zu viel gemeldet**

15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-18

Nebenläufigkeitsprobleme (Forts.)

- Szenario, Teil 2 (Funktion waitsec())
 - Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
 - Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
void waitsec( uint8_t sec ) {
    ...
    // setup timer
    sleep_enable();
    event = 0;
    while( !event ) { // wait for event
        sleep_cpu(); // until next irq
    }
    sleep_disable();
}
```

```
static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

- Wo ist hier das Problem?
 - Test, ob nichts zu tun ist**, gefolgt von **Schlafen, bis etwas zu tun ist**
 - ↪ Potentielle *Lost-Wakeup*-Anomalie

15-IRQ: 2011-09-28



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-20

Nebenläufigkeitsprobleme: Lost-Wakeup-Anomalie

```

void waitsec( uint8_t sec ) {
    ...
    sleep_enable();           // setup timer
    event = 0;
    while( !event ) {        ←
        sleep_cpu();          ←
    }
    sleep_disable();
}

static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}

```

- Angenommen, an **dieser Stelle** tritt der Timer-IRQ (⚡) auf
 - `waitsec` hat bereits festgestellt, dass `event` nicht gesetzt ist
 - `ISR` wird ausgeführt → `event` wird gesetzt
 - Obwohl `event` gesetzt ist, wird der **Schlafzustand** betreten
→ Falls kein weiterer IRQ kommt, **Dornrösenschlaf**



15-IRQ:2011-09-28

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

17 Zusammenfassung

V_GSPiC_handout



© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-21

Lost-Wakeup: Dornrösenschlaf verhindern

```

1 void waitsec( uint8_t sec ) {
2     ...
3     sleep_enable();           // setup timer
4     event = 0;
5     cli();                   ←
6     while( !event ) {        ←
7         sei();                kritisches Gebiet
8         sleep_cpu();          ←
9         cli();                 ←
10    }
11    sei();                  ←
12    sleep_disable();         ←
13
}

static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}

```

- Wo genau ist das **kritische Gebiet**?
 - Test auf Vorbedingung und Betreten des Schlafzustands
(Kann man *das* durch Interruptsperrn absichern?)
 - Problem: Vor `sleep_cpu()` müssen IRQs freigegeben werden!
 - Funktioniert dank spezieller Hardwareunterstützung:
→ Befehlssequenz `sei`, `sleep` wird von der CPU **atomar** ausgeführt

15-IRQ:2011-09-28

© dl GSPiC (Teil C, WS 11) 15 Nebenläufigkeit | 15.4 Nebenläufigkeit und Wettlaufsituationen

15-22

Speicherorganisation

```

int a;                      // a: global, uninitialized
int b = 1;                   // b: global, initialized
const int c = 2;             // c: global, const
                            Wo kommt der
                            Speicher für diese
                            Variablen her?

void main() {
    static int s = 3;         // s: local, static, initialized
    int x, y;                // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}

```

- Statische Allokation** – Reservierung beim Übersetzen / Linken
 - Betrifft globale und modullokale Variablen, sowie den Code
 - Allokation durch Platzierung in einer **Sektion**

.text – enthält den Programmcode
.bss – enthält alle uninitializeden / mit 0 initialisierten Variablen
.data – enthält alle initialisierten Variablen
.rodata – enthält alle initialisierten unveränderlichen Variablen

main()
a
b,s
c

- Dynamische Allokation** – Reservierung zur Laufzeit
 - Betrifft lokale Variablen und explizit angeforderten Speicher
 - Stack – enthält alle **aktuell gültigen** lokalen Variablen
 - Heap – enthält explizit mit `malloc()` angeforderte Speicherbereiche

x,y,p
*p

16-Speicher 2012-01-19

© dl GSPiC (Teil C, WS 11) 16 Speicherorganisation | 16.1 Einführung

16-1



Speicherorganisation auf einem μ C

16-Speicher: 2012-01-19



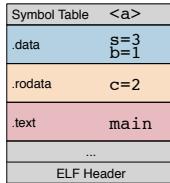
© dl GSPiC (Teil C, WS 11)

```
int a;           // a: global, uninitialized
int b = 1;       // b: global, initialized
const int c = 2; // c: global, const

void main() {
    static int s = 3;      // s: local, static, initialized
    int x, y;             // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm



ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der .bss-Sektion landen ebenfalls in .rodata.

Speicherorganisation auf einem μ C

16-Speicher: 2012-01-19



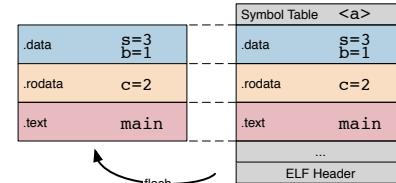
© dl GSPiC (Teil C, WS 11)

```
int a;           // a: global, uninitialized
int b = 1;       // b: global, initialized
const int c = 2; // c: global, const

void main() {
    static int s = 3;      // s: local, static, initialized
    int x, y;             // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm



ELF-Binary

Zur Installation auf dem μ C werden .text und .rodata in den Flash-Speicher des μ C geladen.

16 Speicherorganisation | 16.2 ... auf einem μ -Controller 16-2

Speicherorganisation auf einem μ C

16-Speicher: 2012-01-19



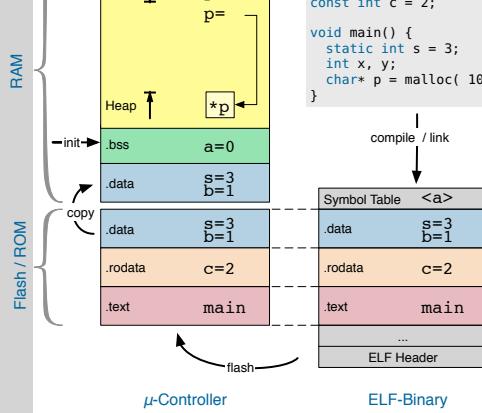
© dl GSPiC (Teil C, WS 11)

```
int a;           // a: global, uninitialized
int b = 1;       // b: global, initialized
const int c = 2; // c: global, const

void main() {
    static int s = 3;      // s: local, static, initialized
    int x, y;             // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm



ELF-Binary

Beim Systemstart wird das .bss-Segment im RAM angelegt und mit 0 initialisiert, das .data-Segment wird aus dem Flash ins RAM kopiert. Das verbleibende RAM wird für den Stack und (falls vorhanden) den Heap verwendet.

Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall \leftrightarrow [14-3]), so werden konstante Variablen ebenfalls in .data abgelegt (und belegen zur Laufzeit RAM).

16 Speicherorganisation | 16.2 ... auf einem μ -Controller 16-2

Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
 - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
 - `void* malloc(size_t n)` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
 - `void free(void* pmem)` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

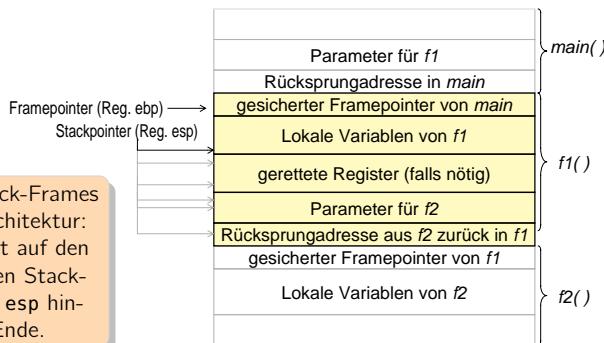
```
#include <stdlib.h>
int* intArray( uint16_t n ) { // alloc int[n] array
    return (int*) malloc( n * sizeof int );
}

void main() {
    int* array = intArray(100); // alloc memory for 100 ints
    if( array ) {
        ...
        array[99] = 4711;
        free( array ); // free allocated block (** IMPORTANT! **)
    }
}
```

16 Speicherorganisation | 16.3 Dynamische Speicherallokation: Heap 16-3

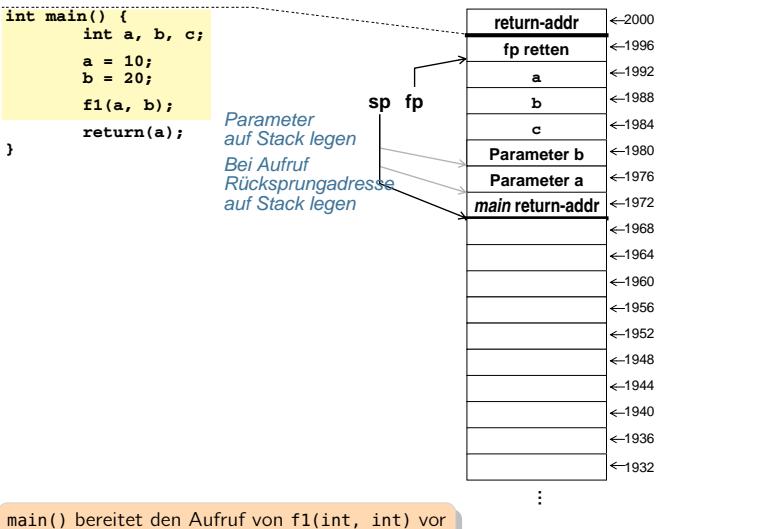
Dynamische Speicherallokation: Stack [→ GDI, V-50]

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem **Stack** (Stapel, Keller) verwaltet
 - Prozessorregister [e]sp zeigt immer auf den nächsten freien Eintrag
 - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von **Stack-Frames**



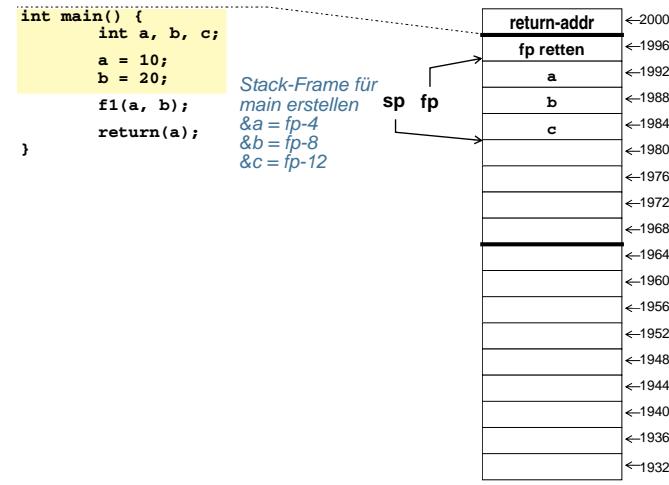
© dl GSPiC (Teil C, WS 11) 16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack 16-4

Stack-Aufbau bei Funktionsaufrufen



© dl GSPiC (Teil C, WS 11) 16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack 16-5

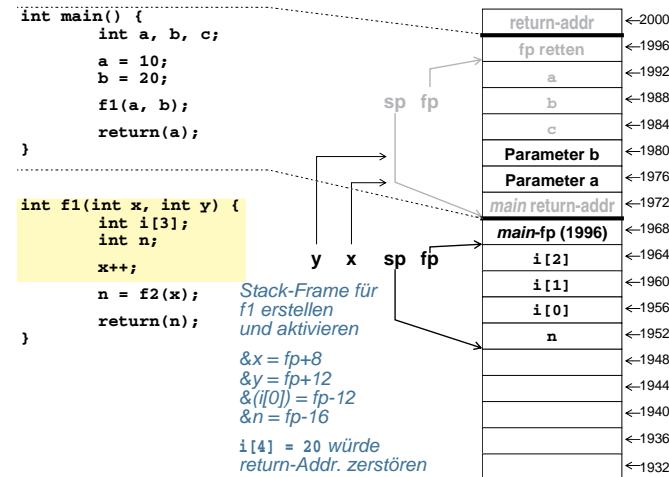
Stack-Aufbau bei Funktionsaufrufen



Beispiel hier für 32-Bit-Architektur (4-Byte ints), main() wurde soeben betreten

© dl GSPiC (Teil C, WS 11) 16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack 16-5

Stack-Aufbau bei Funktionsaufrufen

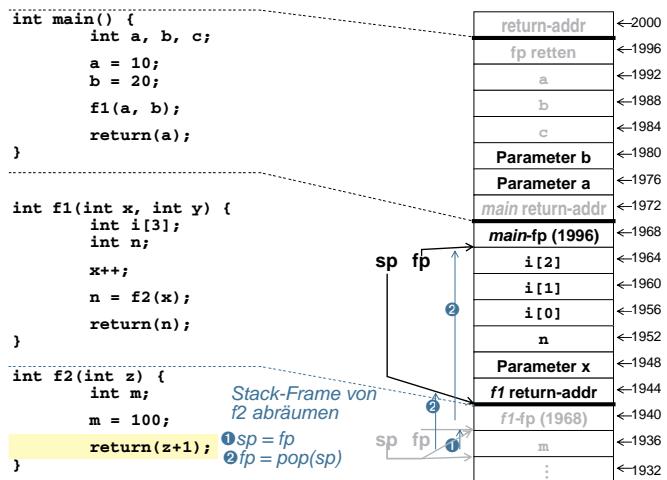


f1() wurde soeben betreten

© dl GSPiC (Teil C, WS 11) 16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack 16-5

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2012-01-19



f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

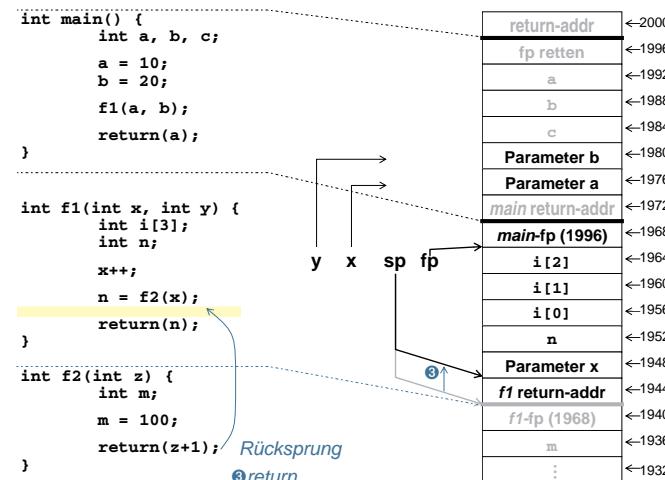
© dl GSPiC (Teil C, WS 11)

16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack

16-5

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2012-01-19



f2() wird verlassen

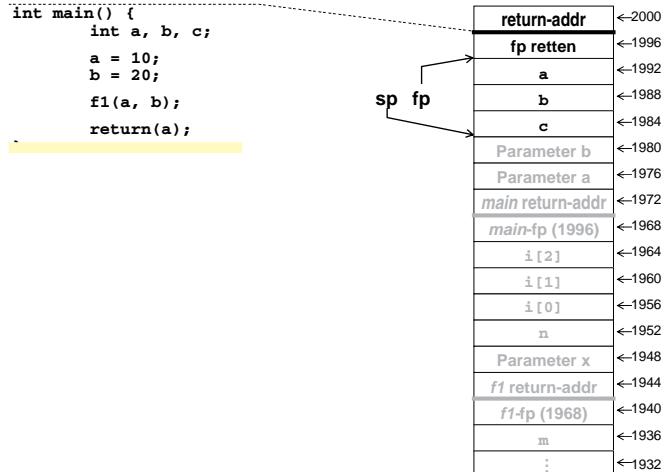
© dl GSPiC (Teil C, WS 11)

16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack

16-5

Stack-Aufbau bei Funktionsaufrufen

16-Speicher: 2012-01-19



zurück in main()

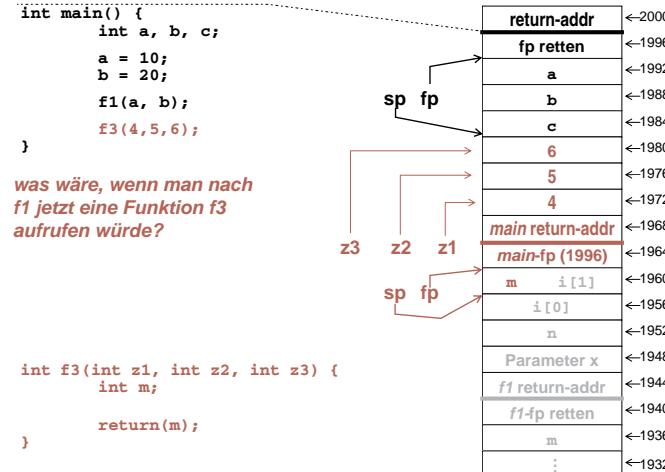
© dl GSPiC (Teil C, WS 11)

16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack

16-Speicher: 2012-01-19



Stack-Aufbau bei Funktionsaufrufen



m wird nicht initialisiert ↵ „erbt“ alten Wert vom Stapel

© dl GSPiC (Teil C, WS 11)

16 Speicherorganisation | 16.4 Dynamische Speicherallokation: Stack

16-5

Statische versus dynamische Allokation

- Bei der **μ C-Entwicklung** wird **statische Allokation** bevorzugt
 - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
 - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ [17-3])
- ```
lohmann@faui48a:~$ size sections.avr
text data bss dec hex filename
682 10 6 698 2ba sections.avr
```
- Sektionsgrößen des Programms von ↪ [16-1]
- ~ Speicher möglichst durch **static**-Variablen anfordern
    - Regel der geringstmöglichen Sichtbarkeit beachten ↪ [12-6]
    - Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden
  - Ein Heap ist **verhältnismäßig teuer** ~ wird möglichst vermieden
    - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
    - Speicherbedarf zur Laufzeit schlecht abschätzbar
    - Risiko von Programmierfehlern und Speicherlecks



## Lernziele

- **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  $\mu$ -Controller ( $\mu$ C)
  - SPiCboard-Lehrentwicklungsplattform mit ATmega- $\mu$ C
  - **Praktische Erfahrungen** in hardwarenaher Softwareentwicklung machen
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen für die Entwicklung systemnaher Software
  - Die Sprache C verstehen und einschätzen können
  - Umgang mit Nebenläufigkeit und Hardwareshäne



## Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14  $\mu$ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

## 17 Zusammenfassung



## Motivation: GSPiC – Stoffauswahl und Konzept

- **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
- **Herausforderung:** Umfang der Veranstaltung (nur 2,5 ECTS)
  - Für Vorlesung und Übung eigentlich zu wenig
  - Veranstaltung soll trotzdem einen **hohen praktischen Anteil** haben
- **Ansatz:** Konzentration auf die Domäne  $\mu$ -Controller
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI)





## Motivation: Die ATmega- $\mu$ C-Familie (8-Bit)

| Type       | Flash   | SRAM   | IO | Timer 8/16 | UART | I <sup>2</sup> 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 „Verschwendungen“  $\rightsquigarrow$  signifikant höhere Stückzahlkosten

## 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 $\mu$ C-Systemarchitektur

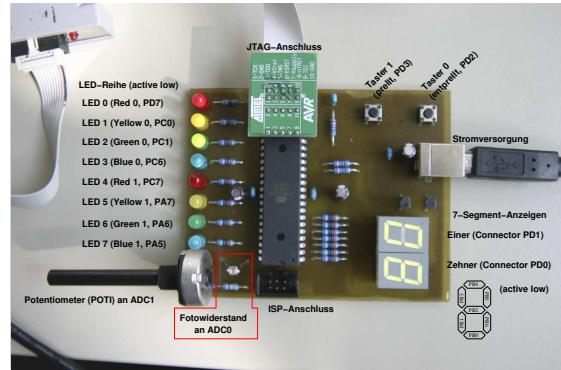
#### 15 Nebenläufigkeit

#### 16 Speicherorganisation

#### 17 Zusammenfassung

## Übungsplattform: Das SPiCboard

- ATmega32- $\mu$ C
- JTAG-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor



- Ausleihe zur Übungsbearbeitung möglich
- Oder noch besser  $\rightarrow$  selber Löten

## Teil B: Einführung in C

### Das erste C-Programm für einen $\mu$ -Controller

#### „Hello World“ für AVR-ATmega (SP)

```
#include <avr/io.h>
void main() {
```

Abstraktion durch Softwareschichten: Vollständiges Beispiel

#### Integertypen: Größe und Wertebereich [≠ Java]

##### Die interne Darstellung (Bitbreite) ist implementierungsabhängig

|           | Datentyp-Breite in Bit |            |         |         |
|-----------|------------------------|------------|---------|---------|
|           | Java                   | C-Standard | gcc/A32 | gcc/A64 |
| char      | 16                     | $\geq 8$   | 8       | 8       |
| short     | 16                     | $\geq 16$  | 16      | 16      |
| int       | 32                     | $\geq 16$  | 32      | 32      |
| long      | 32                     | $\geq 32$  | 32      | 64      |
| long long | -                      | $\geq 64$  | 64      | 64      |

#### Der Wertebereich berechnet sich aus der Bitbreite

- signed  $\rightarrow -(2^{Bits-1} - 1) \rightarrow +(2^{Bits-1} - 1)$
- unsigned  $\rightarrow 0 \rightarrow +(2^{Bits} - 1)$

Hier zeigt sich die C-Philosophie: Effizienz durch Maschinennähe [3-14]  
Die interne Repräsentation der Integertypen ist definiert durch die **Hardware** (Registerbreite, Busbreite, etc.). Das führt im Ergebnis zu **effizientem Code**.

Nun: Entwicklung mit libspicboard

```
#include <led.h>
#include <button.h>
void main() {
 // wait until Button0 is pressed
 while(sb.button.getState(BUTTON0) != BTNPRESSED) {
 // greet user
 sb.led.on(RED0);
 // wait forever
 while(1){
 }
 }
}
```

Hardwareinitialisierung entfällt  
Programm einfacher und verständlicher durch problem-spezifische Abstraktionen  
- Setzt z.B. den Pin PORTD auf HIGH  
- Liest Bit 0 von PORTD  
--- Ab button.getState(BUTTON0)

Abstraktion | 4.1 Funktionsbibliotheken 4-3

## Teil B: Einführung in C

## Eigenarten von C

## Teil C: Systemnahe Softwareentwicklung

## Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Daten nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf den Datenstruktur

### Module in C – Import (Forts.)

[# Java]

**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

In den Linker [→ GDI, VI-158]

steien nicht mehr vorhanden  
ausschließlich  
er)  
en sichergestellt werden  
emeinsame Header-Datei

Module | 12.5 Module in C | 12-12

12-Module | 2013-10-11  
13-Zeiger | 2013-10-26

13-Zeiger und Felder | 13.1 Zeiger – Einführung | 13-1

© dli GSPiC (Teil C, WS11) | 17 Zusammenfassung | 17.2 Was haben wir gemacht? | 17-9

## Veranstaltungsüberblick

## Teil A: Konzept und Organisation

# 1 Einführung

## Teil B: Einführung in C

3 Java Versus C – Erste Beispiele

5 Sprachüberblick

## 6 Einfache Datentypen

## 7 Operatoren und Ausdrücke

8 Kontrollstrukturen

© dl GSPiC (Teil C, WS 11) 17 Zusammenfassung | 17.2 Was haben wir gemacht

9 Funktionen

10 Variablen

## 11 Präprozessor

Teil C: Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felde

14 µC-Systemarchitektur

## 15 Nebenläufigkeit

16 Speicherorganisation

| 17 Zusammenfassung

# Teil C: Systemnahe Softwareentwicklung

## Peripheriegeräte – Register (Forts.)

- Memory-mapped Register ermöglichen
  - Register  $\mapsto$  Speicher  $\mapsto$  Variable
  - Alle C-Operatoren stehen direkt zur Verfügung

Nebenläufigkeitsprobleme: *Read-Write-Anomalie*

## Speicherorganisation auf einem $\mu$ C

Quellprogramm

ELF-Binary

```
2-vect:
... ; save regs
ds r24,cars
... ; restore regs
```

Überlauf der IRQ (#) auf  
send gemeldet

rs genüllt  
=0

Überlauf ins High-Byte

Autos zu viel gemeldet

Überläufigkeit und Wettlaufsituationen

15-18

© dI

GSPiC (Teil C, WS11)

16 Speicherorganisation | 16.2 ... auf einem  $\mu$ -Controller

16-2

17 Zusammenfassung | 17.2 Was haben wir gemacht?

17-10

| KW | Mo     | Di     | Mi     | Do     | Fr     | Themen     | Seiten im Skript                                              |
|----|--------|--------|--------|--------|--------|------------|---------------------------------------------------------------|
| 42 | 17.10. | 18.10. | 19.10. | 20.10. | 21.10. | VL 1       | Einführung, Organisation, Java nach C                         |
| 43 | 24.10. | 25.10. | 26.10. | 27.10. | 28.10. | VL 2       | Abstraktion, Sprachüberblick, Datentypen                      |
| 44 | 31.10. | 01.11. | 02.11. | 03.11. | 04.11. | A1 (Blink) | Variablen, Ausdrücke, Kontrollstrukturen, Funktionen          |
| 45 | 07.11. | 08.11. | 09.11. | 10.11. | 11.11. | A2 (Snake) | Funktionen, Variablen, Präprozessor, Programmstruktur, Module |
| 46 | 14.11. | 15.11. | 16.11. | 17.11. | 18.11. |            |                                                               |
| 47 | 21.11. | 22.11. | 23.11. | 24.11. | 25.11. | A3 (Spiel) | Zeiger                                                        |
| 48 | 28.11. | 29.11. | 30.11. | 01.12. | 02.12. | VL 6       | Funktionszeiger, Mikrocontroller-Systemarchitektur, volatile  |
| 49 | 05.12. | 06.12. | 07.12. | 08.12. | 09.12. | A4 (LED)   |                                                               |
| 50 | 12.12. | 13.12. | 14.12. | 15.12. | 16.12. |            |                                                               |

17-Zusammenfassung: 2012-01-19



© dl GSPiC (Teil C, WS 11) 17 Zusammenfassung | 17.3 Was kommt noch?

17-11

| KW | Mo     | Di     | Mi     | Do     | Fr     | Themen       | Seiten im Skript            |
|----|--------|--------|--------|--------|--------|--------------|-----------------------------|
| 51 | 17.12. | 18.12. | 19.12. | 20.12. | 21.12. | Weihnachten  |                             |
| 52 | 26.12. | 27.12. | 28.12. | 29.12. | 30.12. | Weihnachten  |                             |
| 01 | 02.01. | 03.01. | 04.01. | 05.01. | 06.01. | Weihnachten  |                             |
| 02 | 09.01. | 10.01. | 11.01. | 12.01. | 13.01. | VL 7         | Interrupts, Nebenläufigkeit |
| 03 | 16.01. | 17.01. | 18.01. | 19.01. | 20.01. | A5 (Ampel)   | VL 8                        |
| 04 | 23.01. | 24.01. | 25.01. | 26.01. | 27.01. |              |                             |
| 05 | 30.01. | 01.02. | 02.02. | 03.02. | 04.02. | Wiederholung |                             |
| 06 | 06.02. | 07.02. | 08.02. | 09.02. | 10.02. | VL 9         | Fragestunde                 |

17-Zusammenfassung: 2012-01-19

Siehe [http://www4.informatik.uni-erlangen.de/Lehre/WS11/V\\_GSPiC](http://www4.informatik.uni-erlangen.de/Lehre/WS11/V_GSPiC)

17-11

## Klausur

- Prüfung (Klausur)
  - Termin: 28. März (Zeit noch nicht bekannt)
  - Dauer: 60 min
  - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe
- Zur Vorbereitung stehen alte Klausuren zur Verfügung
  - Werden in den letzten beiden Übungswochen behandelt
  - Können im Forum diskutiert werden
  - Extra Fragestunde am 26. März
- Klausurnote → 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 → +1% der möglichen KP
    - ~ 100% der möglichen ÜP → +10% der möglichen KP

17-Zusammenfassung: 2012-01-19



© dl GSPiC (Teil C, WS 11) 17 Zusammenfassung | 17.3 Was kommt noch?

17-12

## Evaluation

Bitte

Bitte

Bitte

Bitte

Bitte

Bitte

Bitte



Bitte mitmachen :-)

17-Zusammenfassung: 2012-01-19



© dl GSPiC (Teil C, WS 11) 17 Zusammenfassung | 17.3 Was kommt noch?

17-13