

Grundlagen der Systemnahen Programmierung in C (GSPiC)

Teil C Systemnahe Softwareentwicklung

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2017

http://www4.cs.fau.de/Lehre/SS17/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 [↔ 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.

12-Module: 2017-05-10



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

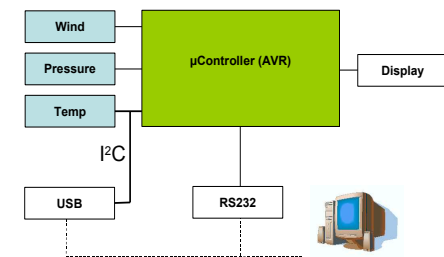
16 Speicherorganisation

V_GSPiC_handout



Beispiel-Projekt: Eine Wetterstation

- Typisches eingebettetes System
 - Mehrere Sensoren
 - Wind
 - Luftdruck
 - Temperatur
 - Mehrere Aktuatoren (hier: Ausgabegeräte)
 - LCD-Anzeige
 - PC über RS232
 - PC über USB
 - Sensoren und Aktuatoren an den μ C angebunden über verschiedene Bussysteme
 - I²C
 - RS232



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

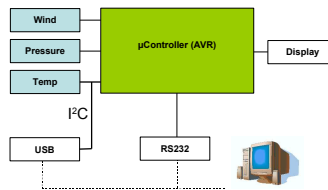
12-Module: 2017-05-10



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

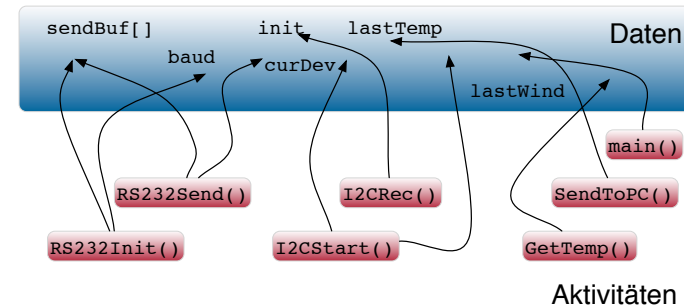


12-Module: 2017-05-10



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: 2017-05-10



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: 2017-05-10



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	→	Definition → Blockende	Definition → Blockende
	static	→	Definition → Blockende	Programmstart → Programmende
Global	<i>keine</i>	→	unbeschränkt	Programmstart → Programmende
	static	→	modulweit	Programmstart → Programmende

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

void f(void) {
    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
}
```

12-Module: 2017-05-10



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

↔ 1-4

Konsequenz: Globale Variablen vermeiden!

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

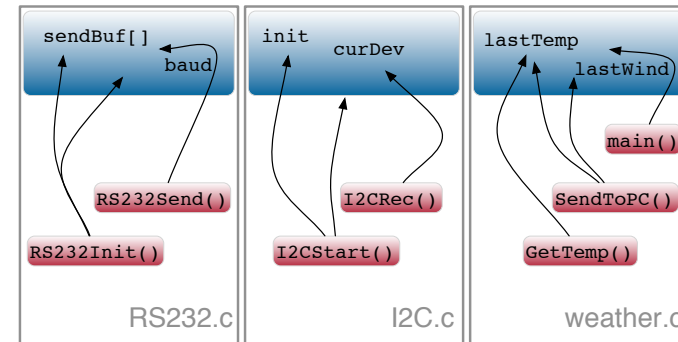
Regel: Variablen erhalten stets die **geringstmögliche Sichtbarkeit und Lebensdauer**

12-Module: 2017-05-10



Lösung: Modularisierung

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



12-Module: 2017-05-10



Was ist ein Modul?

- **Modul** := (<Menge von Funktionen>, \rightarrow „class“ in Java),
<Menge von Daten>,
<Schnittstelle>)
- Module sind größere Programmbausteine \rightarrow 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

↔ 4-1

Modul \rightarrow Abstraktion

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

12-Module: 2017-05-10



Module in C

[\neq Java]

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

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

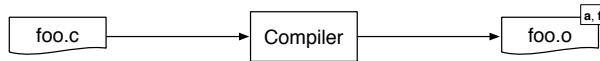
RS232.c: Implementierung (nicht öffentl.)
Definition der bereitgestellten Funktionen (und ggf. Daten)
Ggf. modulinterne Hilfsfunktionen und Daten (static)
Inklusion der eigenen Schnittstelle stellt sicher, dass der Vertrag eingehalten wird

12-Module: 2017-05-10



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)



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.

12-Module: 2017-05-10



Module in C – Import [≠Java]

- 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)

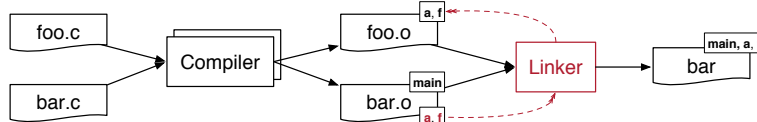
Symbol **main** wird exportiert.
Symbole **a** und **f** sind **unaufgelöst**.

12-Module: 2017-05-10



Module in C – Import (Forts.) [≠Java]

- 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

12-Module: 2017-05-10



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 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)

12-Module: 2017-05-10



Module in C – Header (Forts.)

[≠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)

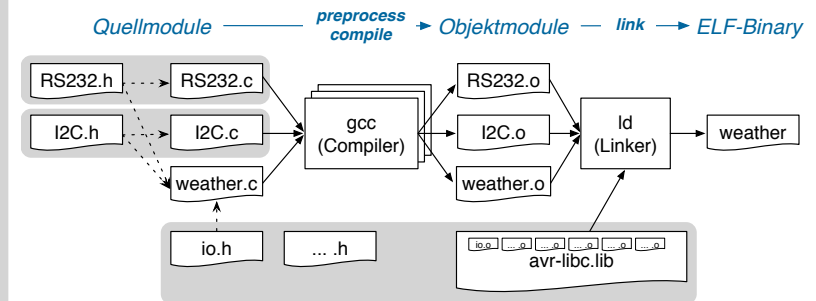
```
// bar.c
extern uint16_t a;
void f(void);
#include <foo.h>

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

12-Module: 2017-05-10



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

12-Module: 2017-05-10



Zusammenfassung

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

12-Module: 2017-05-10



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

V_GSPiC_handout

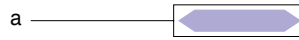


Einordnung: Zeiger (Pointer)

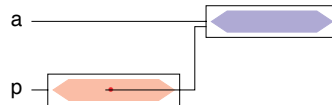
- **Literal: 'a'**
Darstellung eines Wertes

'a' ≡ 0110 0001

- **Variable: char a;**
Behälter für einen Wert



- **Zeiger-Variable: char *p = &a;**
Behälter für eine Referenz auf eine Variable



13-Zeiger: 2017-05-10



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
- 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

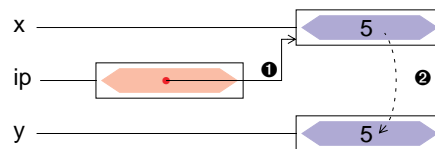
13-Zeiger: 2017-05-10



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: 2017-05-10



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: 2017-05-10



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

13-Zeiger: 2017-05-10



Zeiger als Funktionsargumente (Forts.)

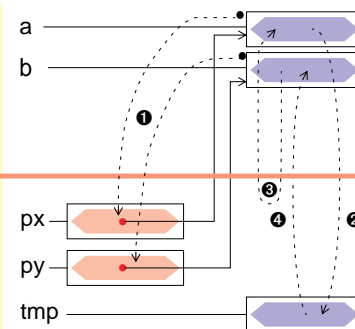
- Beispiel (Gesamtüberblick)

```
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
}

```



13-Zeiger: 2017-05-10

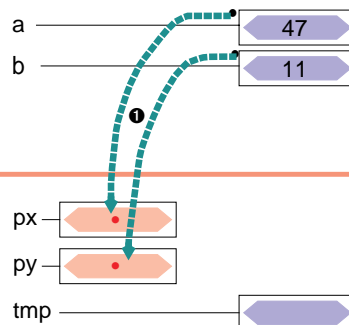


Zeiger als Funktionsargumente (Forts.)

- 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;
}

```

13-Zeiger: 2017-05-10

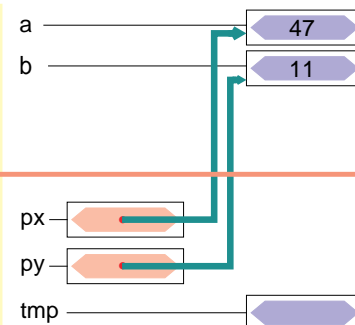


Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

```



13-Zeiger: 2017-05-10



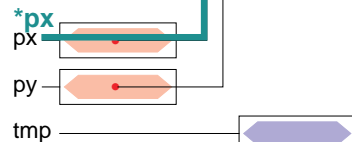
Zeiger als Funktionsargumente (Forts.)

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
```



13-Zeiger: 2017-05-10



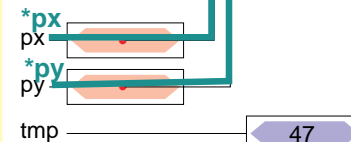
Zeiger als Funktionsargumente (Forts.)

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
```



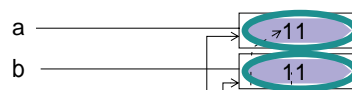
13-Zeiger: 2017-05-10



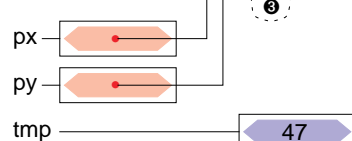
Zeiger als Funktionsargumente (Forts.)

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
```



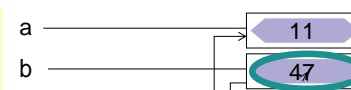
13-Zeiger: 2017-05-10



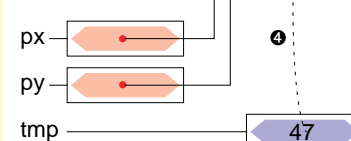
Zeiger als Funktionsargumente (Forts.)

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
}
```



13-Zeiger: 2017-05-10



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). [≠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
}
```

13-Zeiger: 2017-05-10



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 };
```

13-Zeiger: 2017-05-10



Feldzugriff

- Syntax: `Feld [IntAusdruck]` [=Java]
 - Wobei $0 \leq \text{IntAusdruck} < n$ für $n = \text{Feldgröße}$
 - **Achtung:** Feldindex wird nicht überprüft [≠Java]
→ häufige Fehlerquelle in C-Programmen

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!!!
```



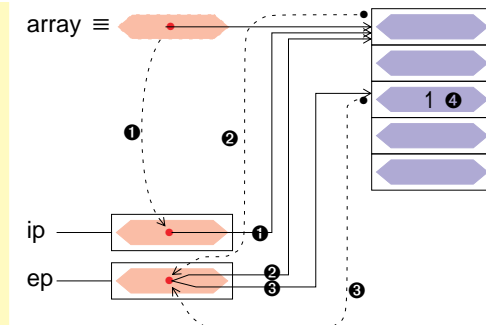
13-Zeiger: 2017-05-10



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 ~ 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
```



13-Zeiger: 2017-05-10



Felder sind Zeiger

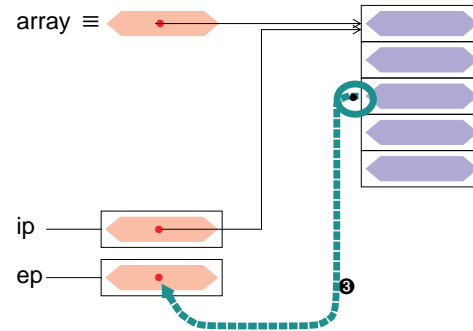
- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array ≡ &array[0]`
 - Ein Alias – kein Behälter ~ Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



Felder sind Zeiger

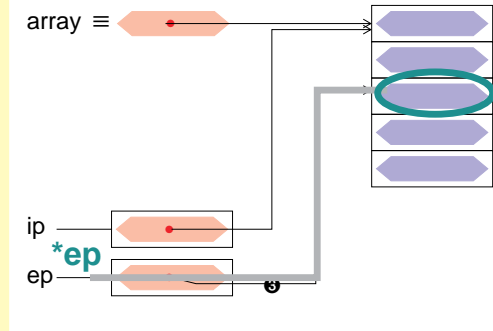
- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array ≡ &array[0]`
 - Ein Alias – kein Behälter ~ Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



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 ↔ 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]);
}
```

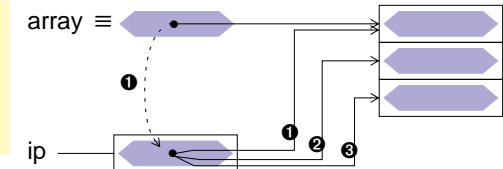


Rechnen mit Zeigern

- Im Unterschied zu einem Feldbezeichner ist eine **Zeigervariable** ein Behälter ~ 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) ≡ &ip[3]`

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

Zeigerarithmetik – Operationen

- 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

13-Zeiger: 2017-05-10



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.

13-Zeiger: 2017-05-10



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);
}
```

R_0 Y_0 B_0 Y_1 B_1
- 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: 2017-05-10



Felder als Funktionsparameter (Forts.)

- 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** [≠Java] → Guter Stil!

```
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)

13-Zeiger: 2017-05-10



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 \0` ↔ `6-13`

- Implementierungsvarianten

Variante 1: Feld-Syntax

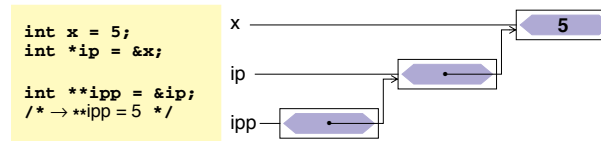
```
int strlen(const char s[]) {
    int n = 0;
    while (s[n] != '\0')
        n++;
    return n;
}
```

Variante 2: Zeiger-Syntax

```
int strlen(const char *s) {
    const char *end = s;
    while (*end != '\0')
        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

- 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.)

- Syntax (Definition): `Typ (*Bezeichner)(FormaleParamopt);`
(sehr ähnlich zur Syntax von Funktionsdeklarationen) ↔ `9-3`
 - `Typ` Rückgabtyp der Funktionen, auf die dieser Zeiger verweisen kann
 - `Bezeichner` Name des Funktionszeigers
 - `FormaleParamopt` 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

```
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()
}
```

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

13-Zeiger: 2017-05-10



Zusammenfassung

- 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

13-Zeiger: 2017-05-10



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

V_GSPiC_handout



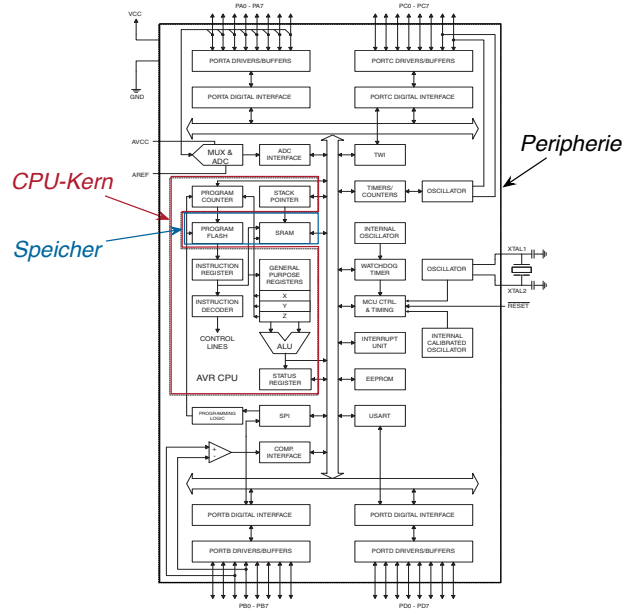
Was ist ein μ -Controller?

- **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 \longleftrightarrow μ C \longleftrightarrow SoC
 - AMD64-CPU's haben ebenfalls eingebaute Timer, Speicher (Caches), ...
 - Einige μ C erreichen die Geschwindigkeit „großer Prozessoren“

14-MC: 2017-05-30



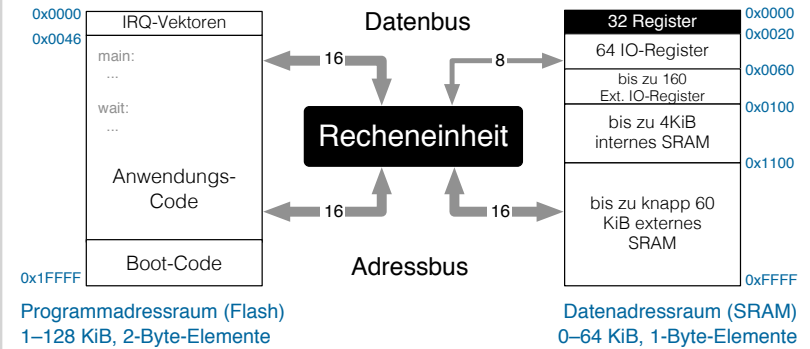
Beispiel ATmega32: Blockschaltbild



14-MC: 2017-05-30



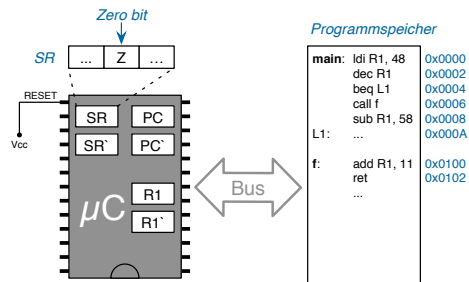
Beispiel ATmega-Familie: CPU-Architektur



14-MC: 2017-05-30



Wie arbeitet ein Prozessor?

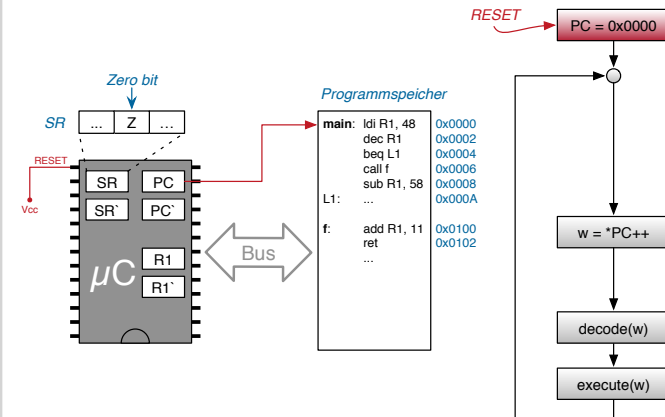


- Hier am Beispiel eines sehr einfachen Pseudoprozessors
 - Nur zwei Vielweckregister (R1 und R2)
 - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
 - Kein Datenspeicher, kein Stapel ~ Programm arbeitet nur auf Registern

14-MC: 2017-05-30



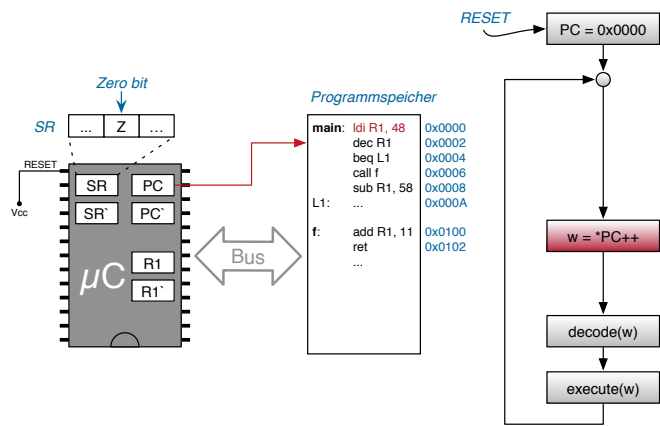
Wie arbeitet ein Prozessor?



14-MC: 2017-05-30



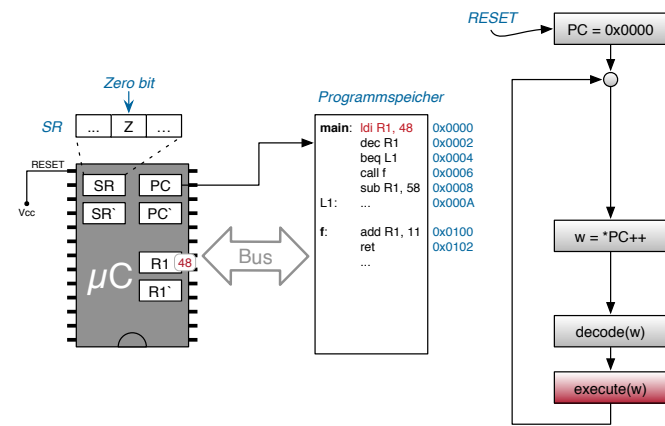
Wie arbeitet ein Prozessor?



14-MC: 2017-05-30



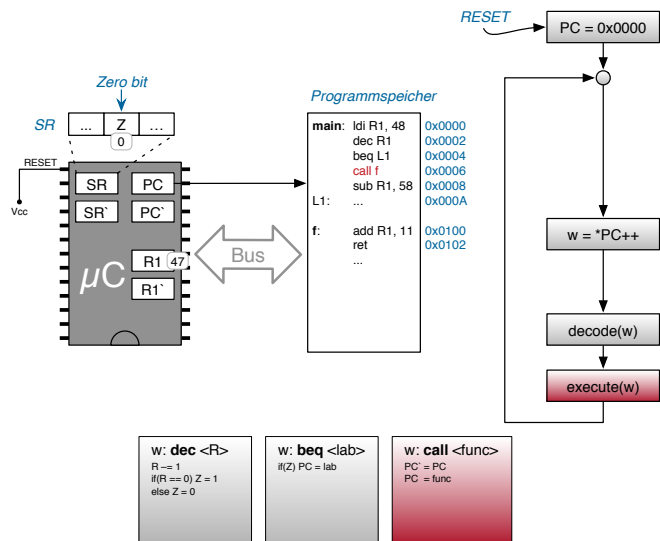
Wie arbeitet ein Prozessor?



14-MC: 2017-05-30



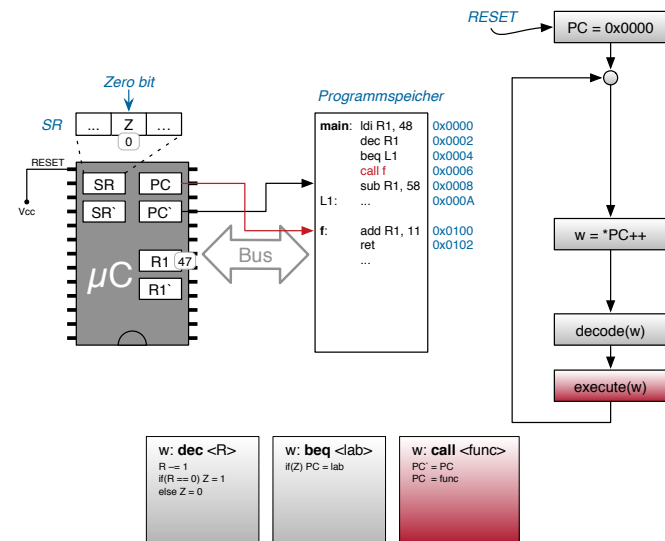
Wie arbeitet ein Prozessor?



14-MC: 2017-05-30



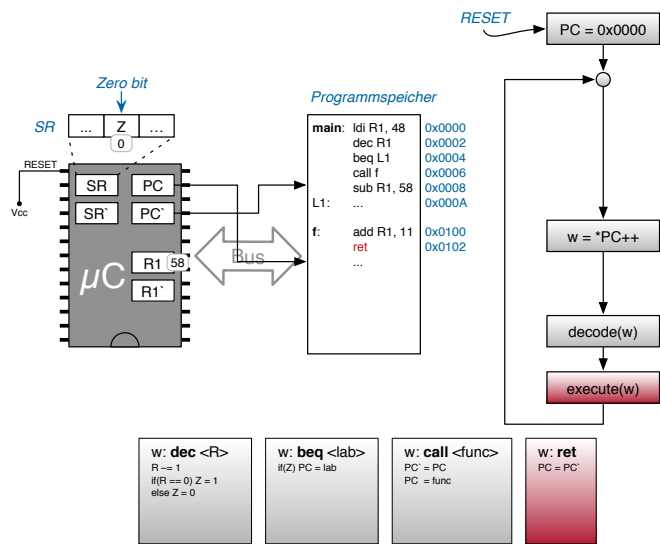
Wie arbeitet ein Prozessor?



14-MC: 2017-05-30



Wie arbeitet ein Prozessor?



14-MC: 2017-05-30

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: 2017-05-30

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 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 → 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. → 14-12

14-MC: 2017-05-30

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: 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
\$3F (\$F)	SREG	1	T	H	S	V	N	Z	C	8
\$3E (\$E)	SPL	-	-	-	-	SP11	SP10	SP9	SP8	11
\$3D (\$D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
\$3C (\$C)	OCR0	Timer/Counter0 Output Compare Register								86
\$12 (\$2)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
\$11 (\$1)	DDRD	DD07	DD06	DD05	DD04	DD03	DD02	DD01	DD00	67
\$10 (\$0)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1, S. 327]

14-MC: 2017-05-30

Peripheriegeräte – Register (Forts.)

- Memory-mapped Register ermöglichen einen komfortablen Zugriff
 - Register → Speicher → 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 *(Casting → 7-10)
 Wert: volatile uint8_t (Dereferenzierung → 13-14)

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

<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 ... L1: sbrc r24, 2 // test bit 2 rjmp L2 ... L2: ret</pre>
--	---

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

<pre>// C code #define PIND \ (*(volatile 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 ... L1: lds r24, 0x0010 // PIND->r24 sbrc r24, 2 // test bit 2 rjmp L2 ... L2: ret</pre>
---	---

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

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:

<pre>// C code void wait(void) { for (uint16_t i = 0; i < 0xffff; i++); }</pre>	<pre>// Resulting assembly code wait: // compiler has optimized // "unneded" loop ret</pre>
--	---

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

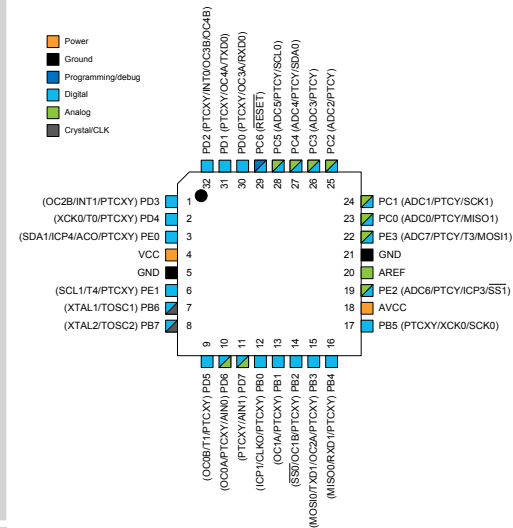
Peripheriegeräte: Ports

- **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)

14-MC: 2017-05-30



Beispiel ATmega328PB: Port/Pin-Belegung



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

Beim **SPiCboard** werden z. B. **Pins 23–24** als **ADCs** konfiguriert, um Poti und Photosensor anzuschließen. PORTC steht daher für diese Pins **nicht zur Verfügung**.

14-MC: 2017-05-30



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
DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
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 ($i=high$, $0=low$), unabhängig von der Konfiguration als Ein-/Ausgang.

7	6	5	4	3	2	1	0
PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
- Verwendungsbeispiele: \leftrightarrow 3-5 und \leftrightarrow 3-8 [1, S. 92]

14-MC: 2017-05-30



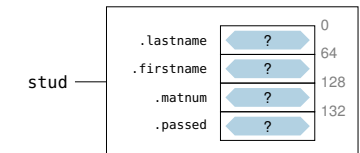
Strukturen: Motivation

- Jeder Port wird durch **drei** globale Variablen verwaltet
 - Es wäre besser diese **zusammen zu fassen**
 - „problembezogene Abstraktionen“ \leftrightarrow 4-1
 - „Trennung der Belange“ \leftrightarrow 12-4
- 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.



14-MC: 2017-05-30



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

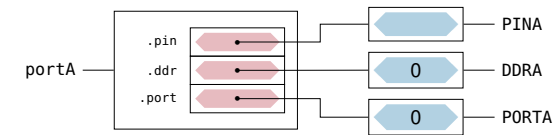
```
typedef struct {
    volatile uint8_t *pin;
    volatile uint8_t *ddr;
    volatile uint8_t *port;
} port_t;
```

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

14-MC: 2017-05-30



Strukturen: Elementzugriff



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

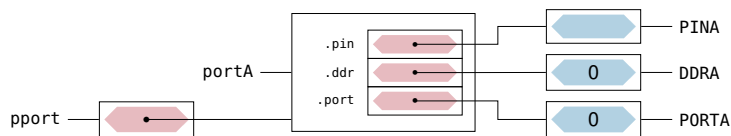
```
port_t portA = { &PINA, &DDRA, &PORTA };
*portA.port = 0; // clear all pins
*portA.ddr = 0xff; // set all to input
```

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

14-MC: 2017-05-30



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->m** ≡ **(*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: 2017-05-30



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

14-MC: 2017-05-30

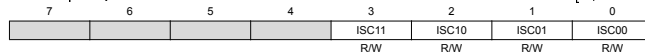


Bit-Strukturen: Bitfelder

- 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

- EICRA** **External Interrupt Control Register A** Steuert Auslöser für externe Interrupt-Quellen INT0 und INT1. [1, S. 83]



```
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 reserved : 4; // bit 4-7: reserved for future use
} EICRA_t;
```

14-MC: 2017-05-30



Unions

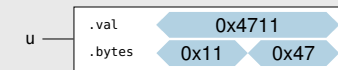
- In einer Struktur liegen die Elemente **hintereinander** im Speicher, in einer Union hingegen **übereinander**
 - Wert im Speicher lässt sich verschieden (Typ-)interpretieren
 - Nützlich für bitweise Typ-Casts
- Beispiel

↔ 14-15

```
void main(void) {
    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]);
    ...
}
```



14-MC: 2017-05-30



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 reserved : 4;
    };
} EICRA_t;

void foo(void) {
    EICRA_t *eicra = (EICRA_t *) 0x69;
    uint8_t oldval = eicra->reg; // save register
    ...
    eicra->ISC0 = 2; // use register
    eicra->ISC1 = 1; // ...
    ...
    eicra->reg = oldval; // restore register
}
```

14-MC: 2017-05-30



Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 µC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

V_GSPiC_handout



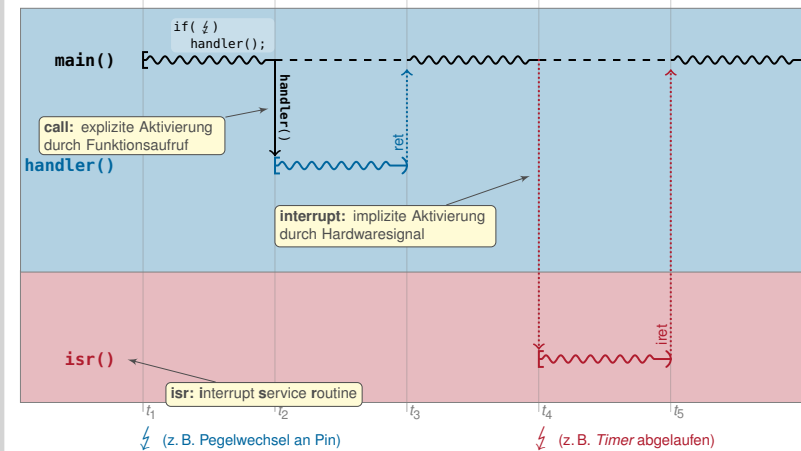
Ereignisbehandlung

- Bei einem **Peripheriegerät** tritt ein Ereignis (⚡) auf → 14-5
 - 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: 2017-05-30



Interrupt → Funktionsaufruf „von außen“



15-IRQ: 2017-05-30



Polling vs. Interrupts – Vor- und Nachteile

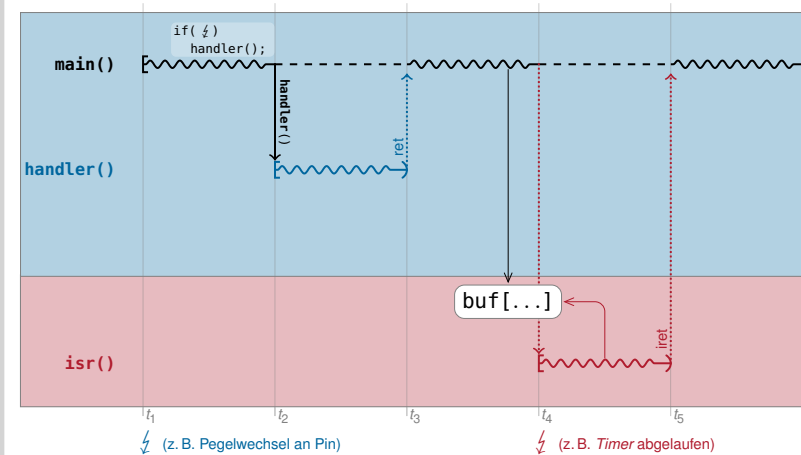
- Polling** (→ „Taktgesteuertes System“)
 - Ereignisbearbeitung erfolgt **synchron** zum Programmablauf
 - Ereigniserkennung über das Programm „verstreut“ (Trennung der Belange)
 - „Verschwendung“ von Prozessorzeit (falls anderweitig verwendbar)
 - Hochfrequentes Pollen \rightsquigarrow hohe Prozessorlast \rightsquigarrow **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 \rightsquigarrow Synchronisation erforderlich
 - Programmverhalten **schwer vorhersagbar**

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

15-IRQ: 2017-05-30



Interrupt → unvorhersagbarer Aufruf „von außen“



15-IRQ: 2017-05-30



Interruptsperrn

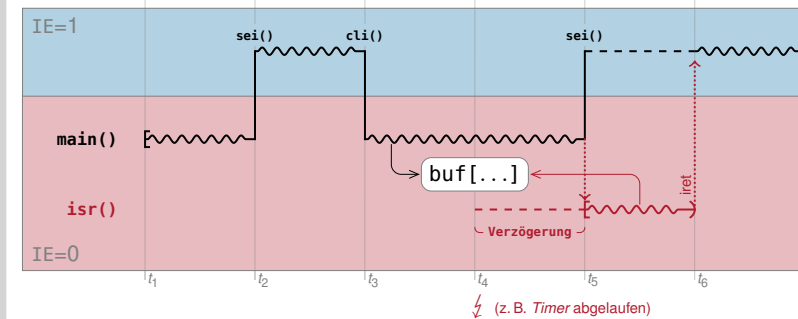
- Zustellung von Interrupts kann softwareseitig **gesperrt** werden
 - Wird benötigt zur **Synchronisation** mit ISRs
 - Einzelne ISR: Bit in gerätespezifischem Steuerregister
 - Alle ISRs: Bit (**IE**, *Interrupt Enable*) im Statusregister der CPU
- Auflaufende IRQs werden (üblicherweise) gepuffert
 - 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: 2017-05-30



Interruptsperrn: Beispiel



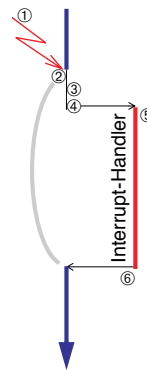
- t_1 Zu Beginn von `main()` sind IRQs gesperrt (`IE=0`)
- t_2, t_3 Mit `sei()` / `cli()` werden IRQs freigegeben (`IE=1`) / erneut gesperrt
- t_4 ⚡ aber `IE=0` ~ Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t_5 `main()` gibt IRQs frei (`IE=1`) ~ gepufferter IRQ „schlägt durch“
- t_5-t_6 Während der ISR-Bearbeitung sind die IRQs gesperrt (`IE=0`)
- t_6 Unterbrochenes `main()` wird fortgesetzt

15-IRQ: 2017-05-30



Ablauf eines Interrupts – Überblick

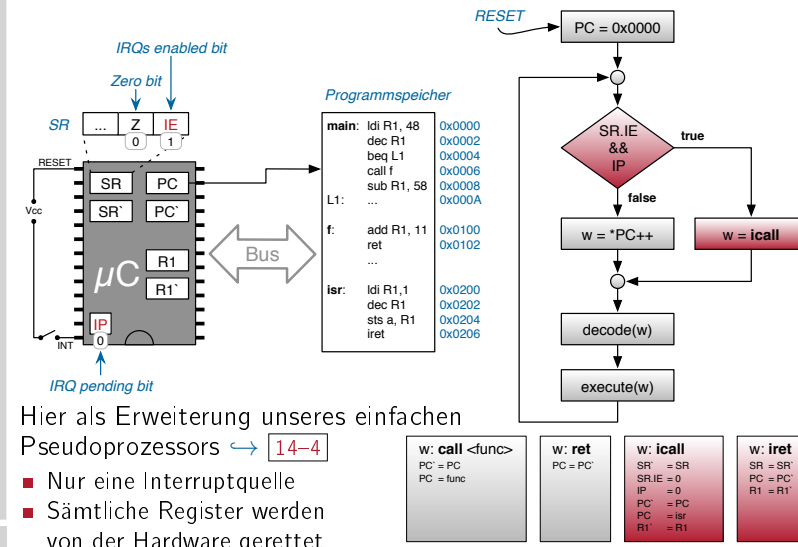
- Gerät signalisiert Interrupt
 - Anwendungsprogramm wird „unmittelbar“ (vor dem nächsten Maschinenbefehl mit `IE=1`) unterbrochen
- Die Zustellung weiterer Interrupts wird gesperrt (`IE=0`)
 - Zwischenzeitlich auflaufende Interrupts werden gepuffert (maximal einer pro Quelle!)
- Registerinhalte werden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



15-IRQ: 2017-05-30



Ablauf eines Interrupts – Details



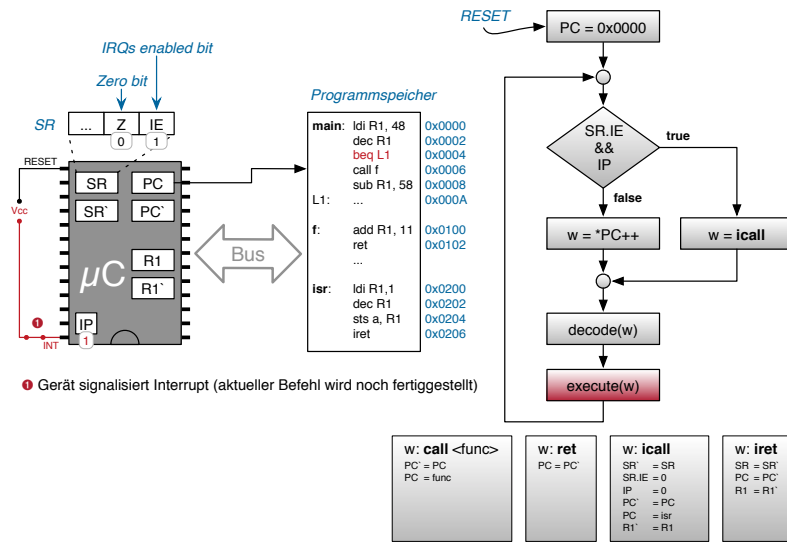
- Hier als Erweiterung unseres einfachen Pseudoprozessors ↔ 14-4

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

15-IRQ: 2017-05-30

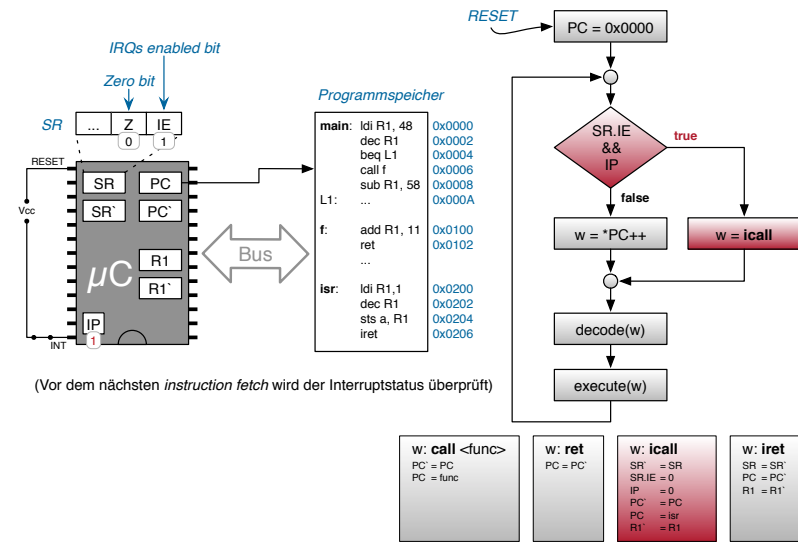


Ablauf eines Interrupts – Details



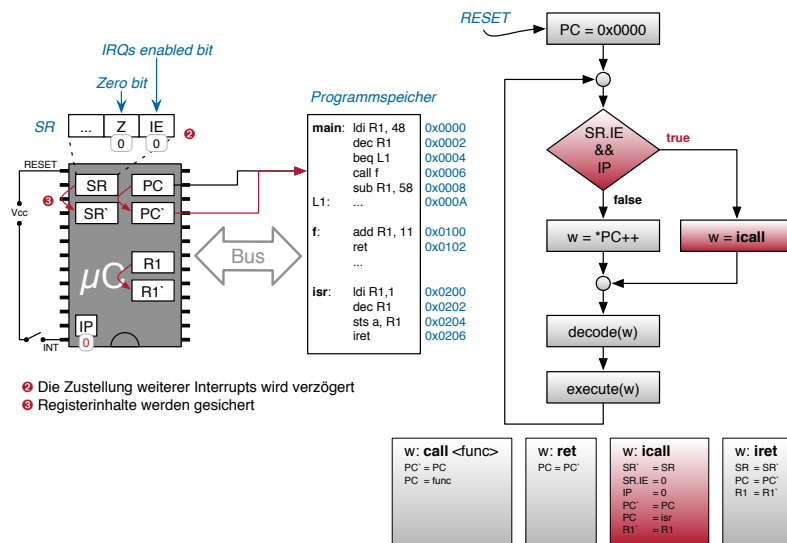
15-IRQ: 2017-05-30

Ablauf eines Interrupts – Details



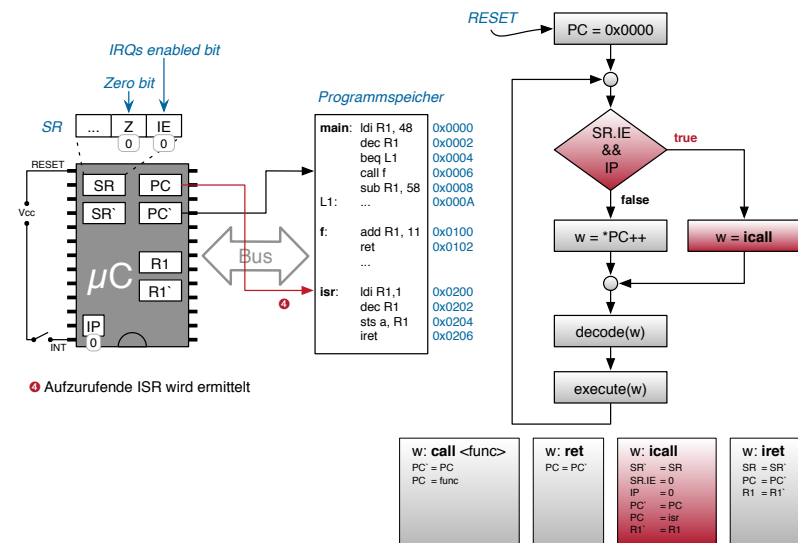
15-IRQ: 2017-05-30

Ablauf eines Interrupts – Details



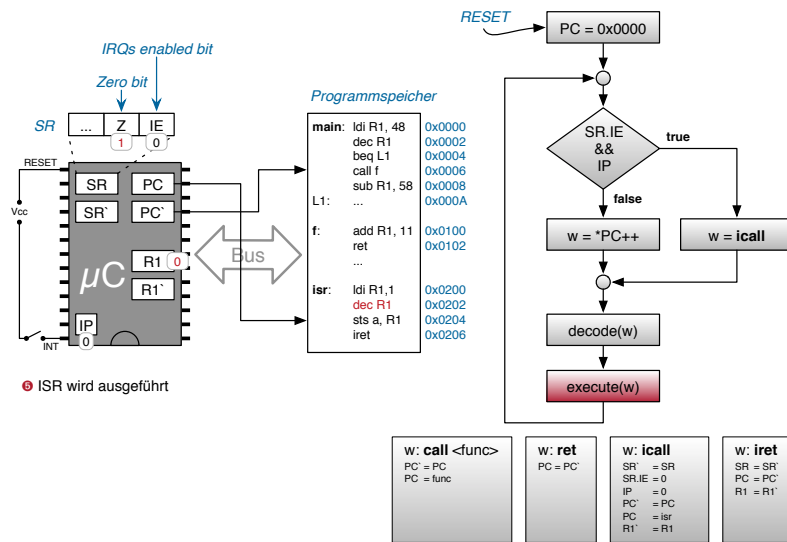
15-IRQ: 2017-05-30

Ablauf eines Interrupts – Details



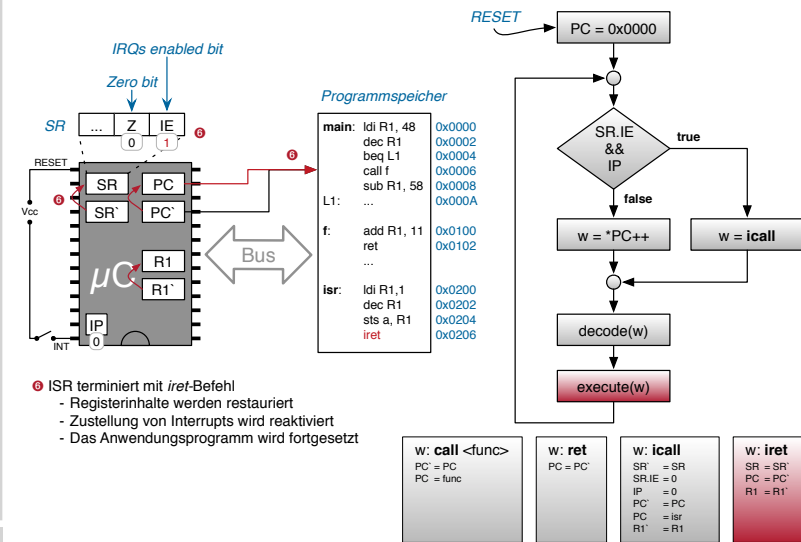
15-IRQ: 2017-05-30

Ablauf eines Interrupts – Details



15-IRQ: 2017-05-30

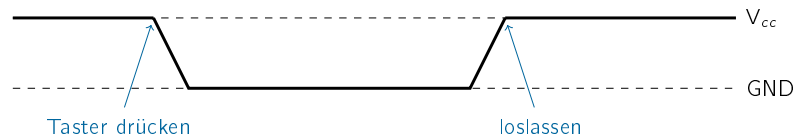
Ablauf eines Interrupts – Details



15-IRQ: 2017-05-30

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

15-IRQ: 2017-05-30

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega328PB (IRQ \mapsto *Interrupt ReQuest*)

- 45 IRQ-Quellen
- einzeln de-/aktivierbar
- IRQ \leadsto Sprung an Vektor-Adresse

[1, S. 78]

- Verschaltung SPiCboard (\leftrightarrow 14-14 \leftrightarrow 2-4)

- INT0 \mapsto PD2 \mapsto Button0 (hardwareseitig entprellt)
- INT1 \mapsto PD3 \mapsto Button1

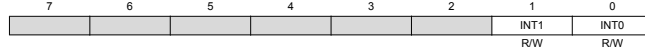
Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPB	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPB	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPB	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI0_STC	SPI1 Serial Transfer Complete
19	0x0024	USART0_RX	USART0 Rx Complete
20	0x0026	USART0_UDRE	USART0, Data Register Empty
21	0x0028	USART0_TX	USART0, Tx Complete
22	0x002A	ADC	ADC Conversion Complete

15-IRQ: 2017-05-30

Externe Interrupts: Register

Steuerregister für INT0 und INT1

- **EIMSK** **External Interrupt Mask Register:** Legt fest, ob die Quellen INT i IRQs auslösen (Bit INT i =1) oder deaktiviert sind (Bit INT i =0) [1, S. 84]



- **EICRA** **External Interrupt Control Register A:** Legt für externe Interrupts INT0 und INT1 fest, wodurch ein IRQ ausgelöst wird (Flanken-/Pegelsteuerung) [1, S. 83]



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.

15-IRQ: 2017-05-30



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

15-IRQ: 2017-05-30



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(void) {
    DDRD &= ~(1<<PD3); // PD3: input with pull-up
    PORTD |= (1<<PD3);
    EICRA &= ~(1<<ISC10 | 1<<ISC11); // INT1: IRQ on level=low
    EIMSK |= (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`)

15-IRQ: 2017-05-30



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(void) {
    ...
    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.

15-IRQ: 2017-05-30



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

- Sei `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**

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:
...           ; save regs
lds r24,cars
lds r25,cars+1
adiw r24,1
sts cars+1,r25
sts cars,r24
...           ; restore regs
    
```

- Sei 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 genullt
 ~ 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**

Interruptsperrn: Datenflussanomalien verhindern

```

void main(void) {
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 **Interruptsperrn** erreicht werden
 - ISR unterbricht main, aber nie umgekehrt ~ asymmetrische Synchronisation
 - Achtung: Interruptsperrn sollten **so kurz wie möglich** sein
 - Wie lange braucht die Funktion send hier?
 - Kann man send aus dem kritischen Gebiet herausziehen?

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

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

```

void waitsec(uint8_t sec) {
...           // setup timer
sleep_enable();
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, **Dornröschenschlaf**



Lost-Wakeup: Dornröschenschlaf verhindern

```
1 void waitsec(uint8_t sec) {
2   ... // setup timer
3   sleep_enable();
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 Interruptsperrern absichern?)
 - Problem: Vor `sleep_cpu()` müssen IRQs freigegeben werden!
 - Funktioniert dank spezieller Hardwareunterstützung:
↳ Befehlssequenz `sei`, `sleep` wird von der CPU **atomar** ausgeführt

Zusammenfassung

- Interruptbearbeitung erfolgt **asynchron** zum Programmablauf
 - Unerwartet ~ Zustandssicherung im Interrupt-Handler erforderlich
 - Quelle von Nebenläufigkeit ~ **Synchronisation erforderlich**
- Synchronisationsmaßnahmen
 - Gemeinsame Zustandsvariablen als **volatile** deklarieren (immer)
 - Zustellung von Interrupts sperren: `cli`, `sei` (bei nichtatomaren Zugriffen, die mehr als einen Maschinenbefehl erfordern)
 - Bei **längeren Sperrzeiten können IRQs verloren gehen!**
- Nebenläufigkeit durch Interrupts ist eine **sehr große Fehlerquelle**
 - *Lost-Update* und *Lost-Wakeup* Probleme
 - indeterministisch ~ durch Testen schwer zu fassen
- Wichtig zur Beherrschbarkeit: **Modularisierung** ↔ 12-7
 - Interrupthandler und Zugriffsfunktionen auf gemeinsamen Zustand (**static** Variablen!) in eigenem Modul kapseln.

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

Speicherorganisation

```
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)
}
```

Wo kommt der Speicher für diese Variablen her?

- **Statische Allokation** – Reservierung beim Übersetzen / Linken
 - Betrifft alle globalen/statischen Variablen, sowie den Code ↔ 12-5
 - Allokation durch Platzierung in einer **Sektion**
 - `.text` – enthält den Programmcode main()
 - `.bss` – enthält alle mit 0 initialisierten Variablen a
 - `.data` – enthält alle mit anderen Werten initialisierten Variablen b,s
 - `.rodata` – enthält alle unveränderlichen Variablen c
- **Dynamische Allokation** – Reservierung zur Laufzeit
 - Betrifft lokale auto-Variablen und explizit angeforderten Speicher
 - `Stack` – enthält alle **aktuell lebendigen** auto-Variablen x,y,p
 - `Heap` – enthält explizit mit `malloc()` angeforderte Speicherbereiche *p

Speicherorganisation auf einem μC

```

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

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

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 der Symboltabelle.

20-Speicher: 2017-02-17



Speicherorganisation auf einem μC

```

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

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

ELF-Binary

Zur Installation auf dem μC werden .text und .[ro]data in den Flash-Speicher des μC geladen.

Flash / ROM

$\mu\text{-Controller}$

20-Speicher: 2017-02-17



Speicherorganisation auf einem μC

```

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

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

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.

RAM

Flash / ROM

$\mu\text{-Controller}$

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

20-Speicher: 2017-02-17



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 );
}

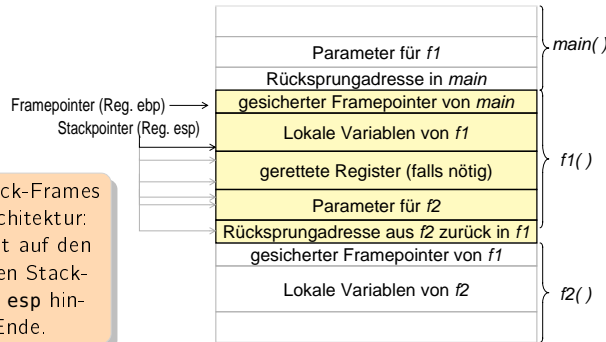
void main() {
    int* array = intArray(100); // alloc memory for 100 ints
    if( array ) { // malloc() returns NULL on failure
        ... // if succeeded, use array
        array[99] = 4711;
        ...
        free( array ); // free allocated block (** IMPORTANT! **)
    }
}
    
```

20-Speicher: 2017-02-17



Dynamische Speicherallokation: Stack

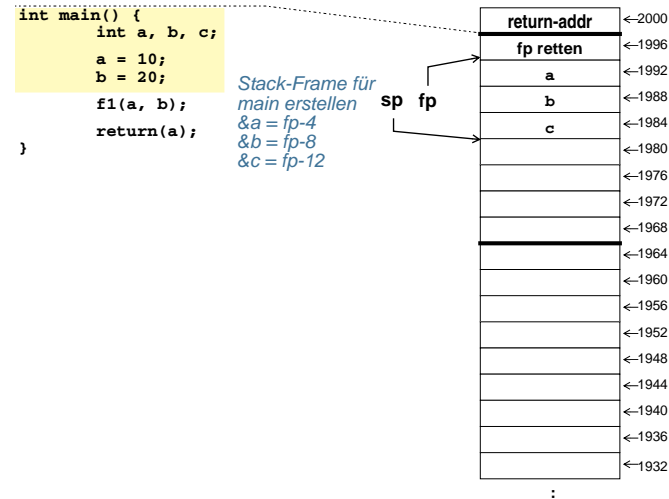
- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem **Stack** (Stapel, Keller) verwaltet
 - Prozessorregister [e]sp zeigt immer auf den nächsten freien Eintrag
 - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von **Stack-Frames**



Aufbau eines Stack-Frames auf der IA-32-Architektur: Register ebp zeigt auf den Beginn des aktiven Stack-Frames; Register esp hinter das aktuelle Ende.

20-Speicher: 2017-02-17

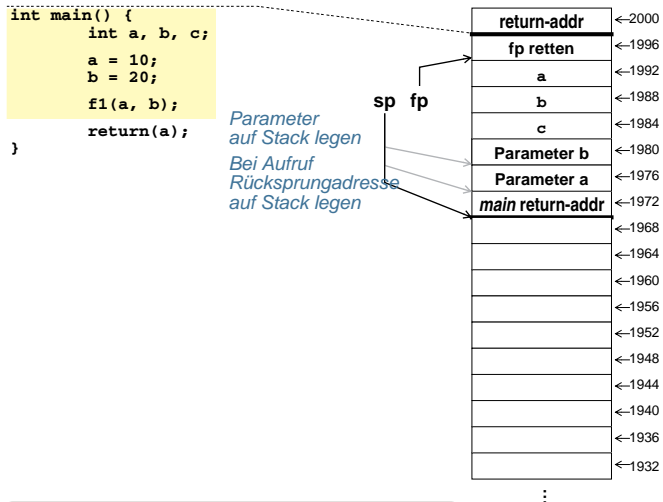
Stack-Aufbau bei Funktionsaufrufen



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

20-Speicher: 2017-02-17

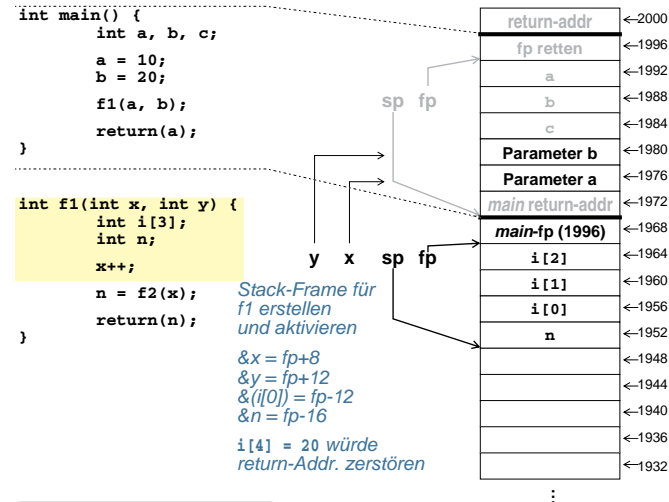
Stack-Aufbau bei Funktionsaufrufen



main() bereitet den Aufruf von f1(int, int) vor

20-Speicher: 2017-02-17

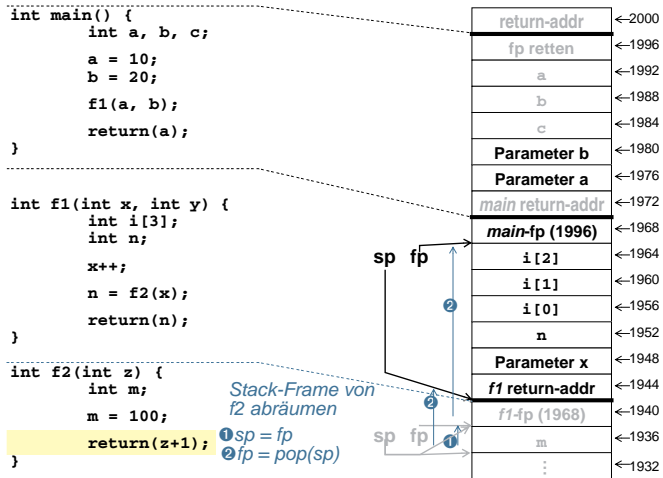
Stack-Aufbau bei Funktionsaufrufen



f1() wurde soeben betreten

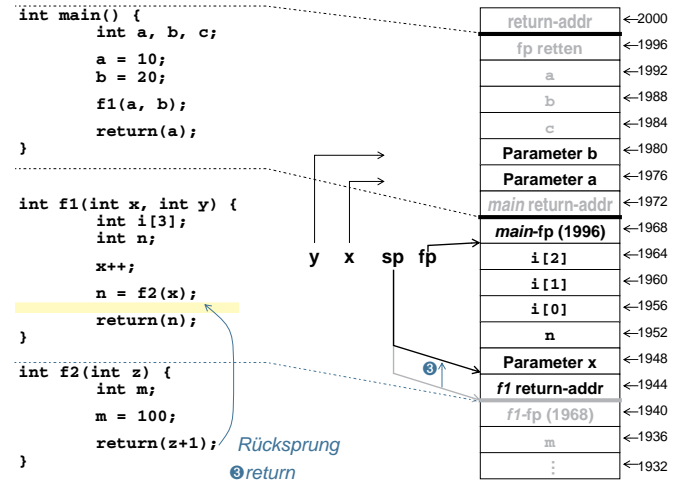
20-Speicher: 2017-02-17

Stack-Aufbau bei Funktionsaufrufen



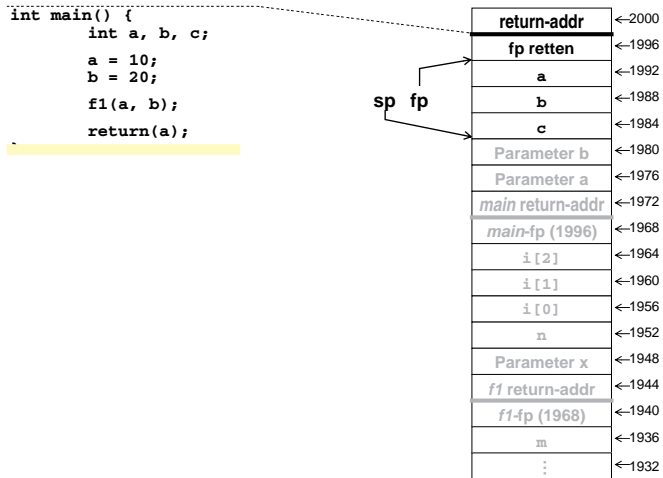
f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

Stack-Aufbau bei Funktionsaufrufen



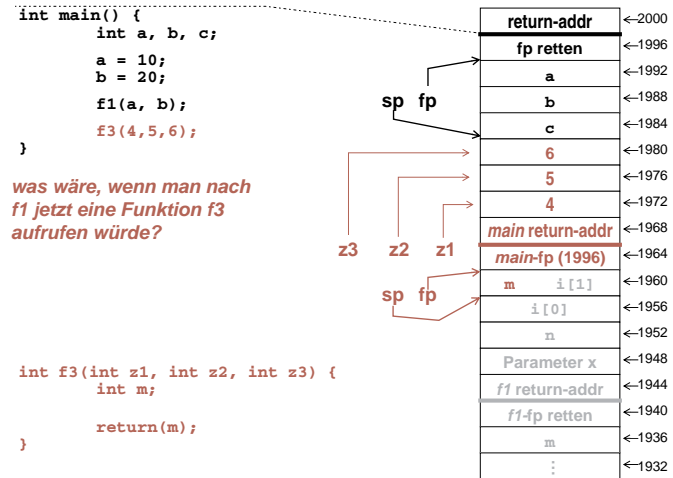
f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen



zurück in main()

Stack-Aufbau bei Funktionsaufrufen



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

Statische versus dynamische Allokation

- Bei der **μC-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! ↔ 1-4)

```
lohmann@faii48a:~$ 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

