

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μ C-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

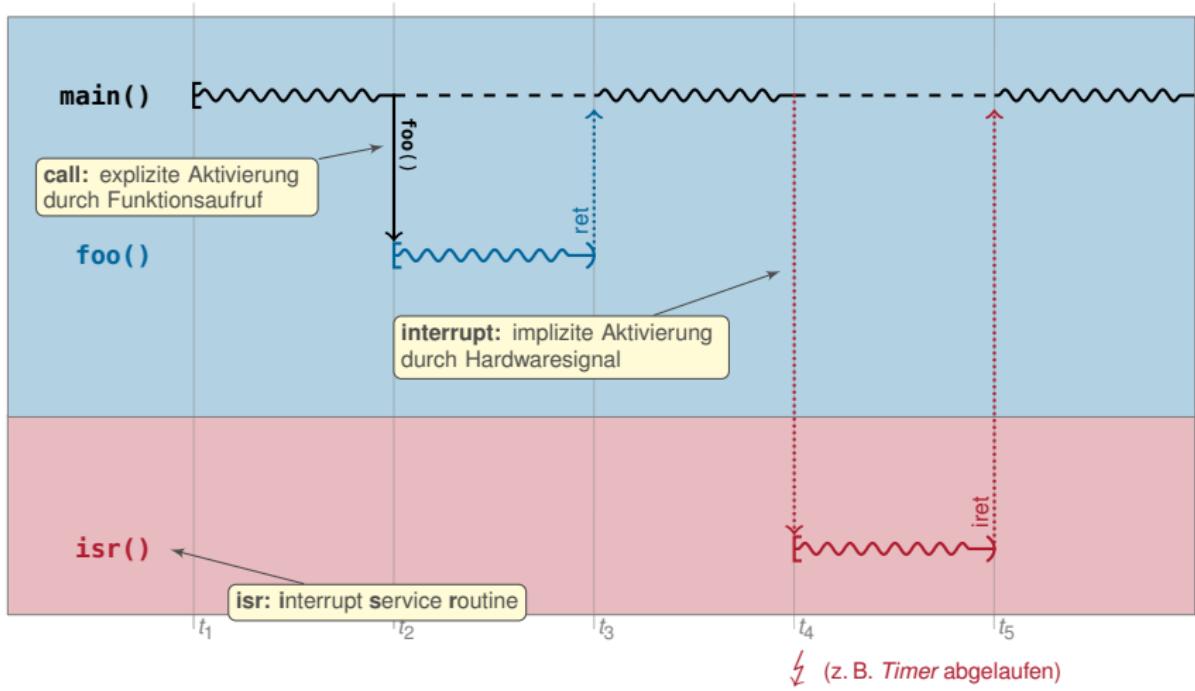


- Bei einem Peripheriegerät tritt ein Ereignis () auf
 - Signal an einem Port-Pin wechselt von *low* auf *high*
 - Ein *Timer* ist abgelaufen
 - Ein A/D-Wandler hat einen neuen Wert vorliegen
 - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
 - **Polling:**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

↪ 14–5



Interrupt \mapsto Funktionsaufruf „von außen“



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



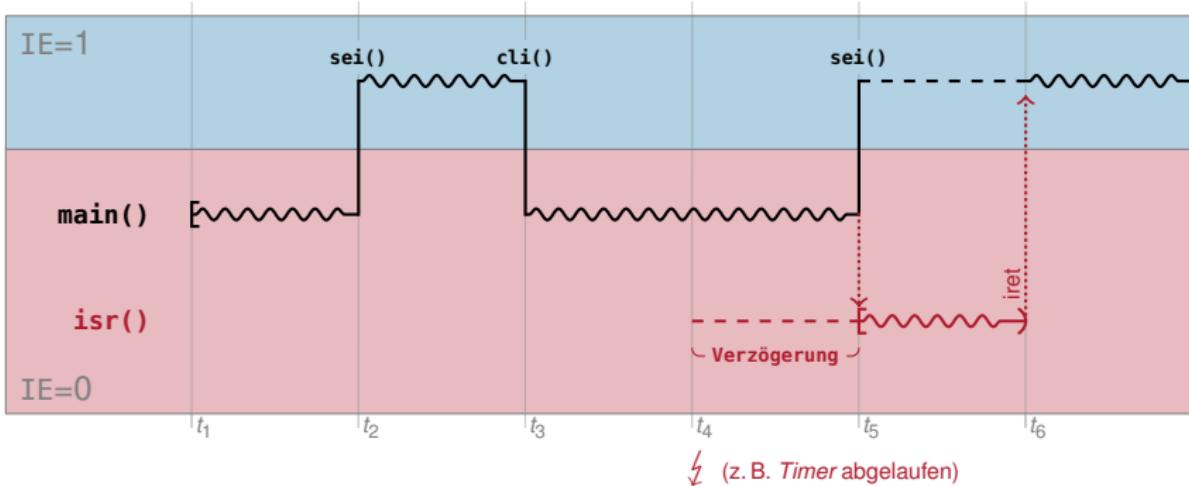
Interruptsperren

- 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`: $IE \leftarrow 0$ (*clear interrupt*, IRQs gesperrt)
 `sei`: $IE \leftarrow 1$ (*set interrupt*, IRQs erlaubt)
 - Nach einem RESET: $IE=0 \rightsquigarrow$ IRQs sind zu Beginn des Hauptprogramms gesperrt
 - Bei Betreten einer ISR: $IE=0 \rightsquigarrow$ IRQs sind während der Interruptbearbeitung gesperrt

IRQ \leftrightarrow *Interrupt Request*



Interruptsperren: Beispiel

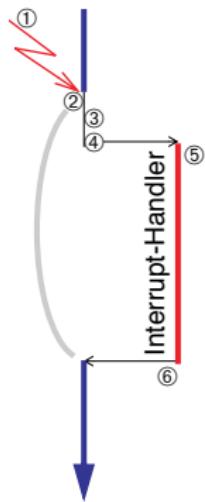


- t_1 Zu Beginn von $\text{main}()$ sind IRQs gesperrt ($\text{IE}=0$)
- t_2, t_3 Mit $\text{sei}()$ / $\text{cli}()$ werden IRQs freigegeben ($\text{IE}=1$) / erneut gesperrt
- t_4 ↳ aber $\text{IE}=0 \rightsquigarrow$ Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t_5 $\text{main}()$ gibt IRQs frei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

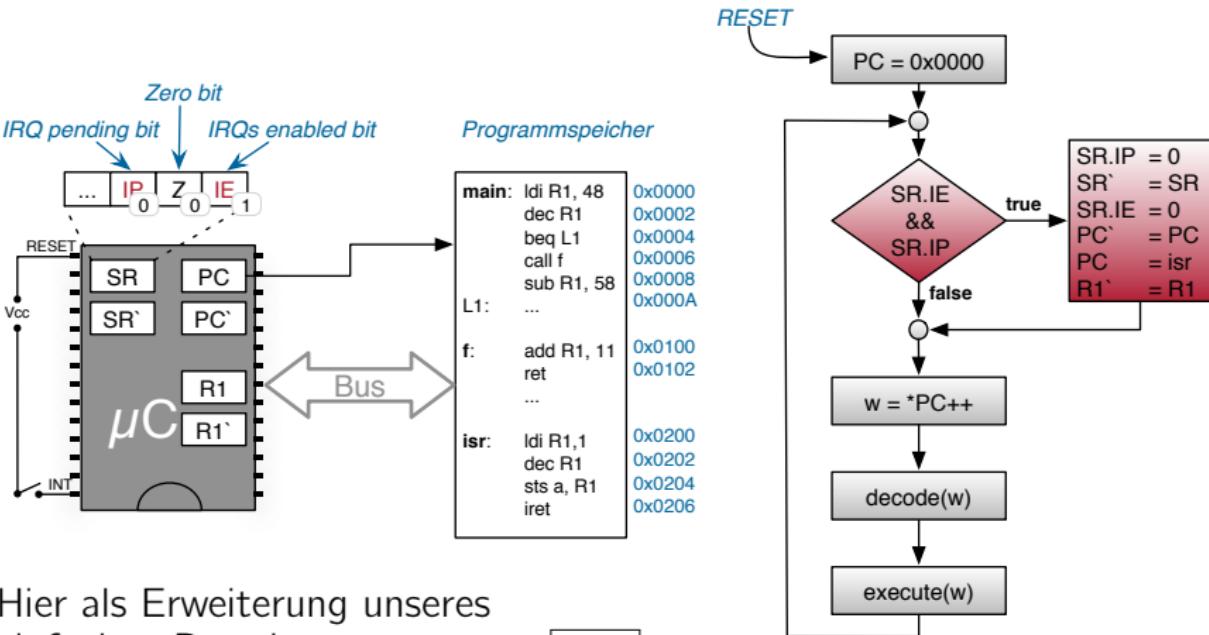


Ablauf eines Interrupts – Überblick

- 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



Ablauf eines Interrupts – Details



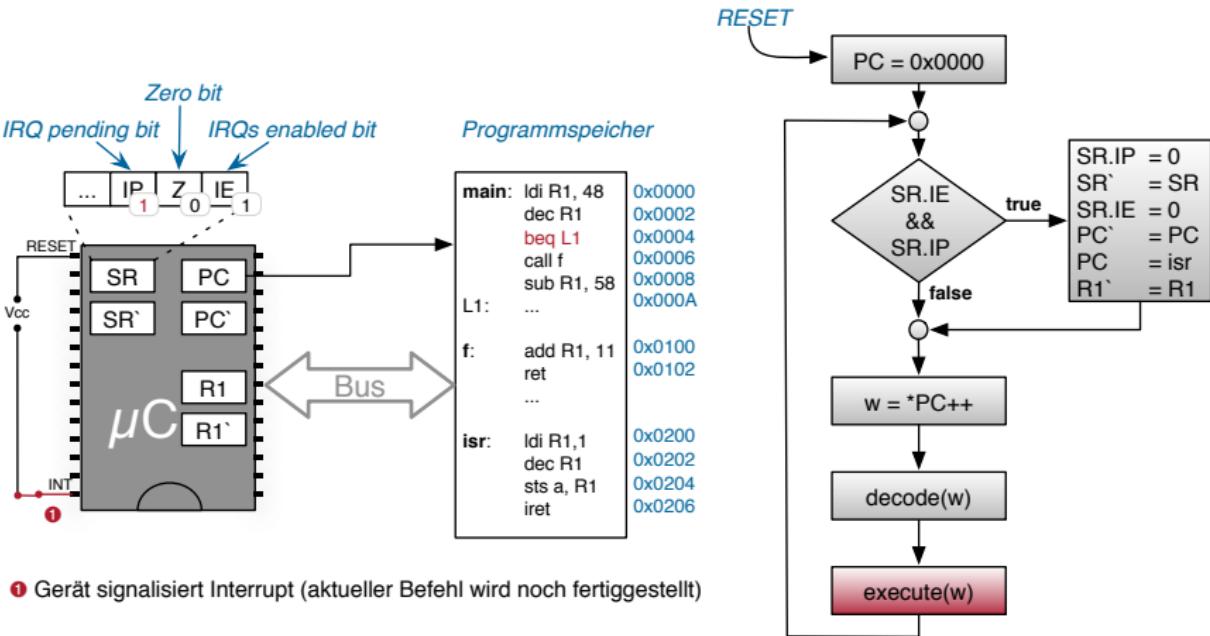
- Hier als Erweiterung unseres einfachen Pseudoprozessors ↪ 14–4
 - Nur eine Interruptquelle
 - Sämtliche Register werden von der Hardware gerettet

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

w: **ret**
PC = PC'
R1 = R1'

w: **iret**
SR = SR'
PC = PC'
R1 = R1'

Ablauf eines Interrupts – Details



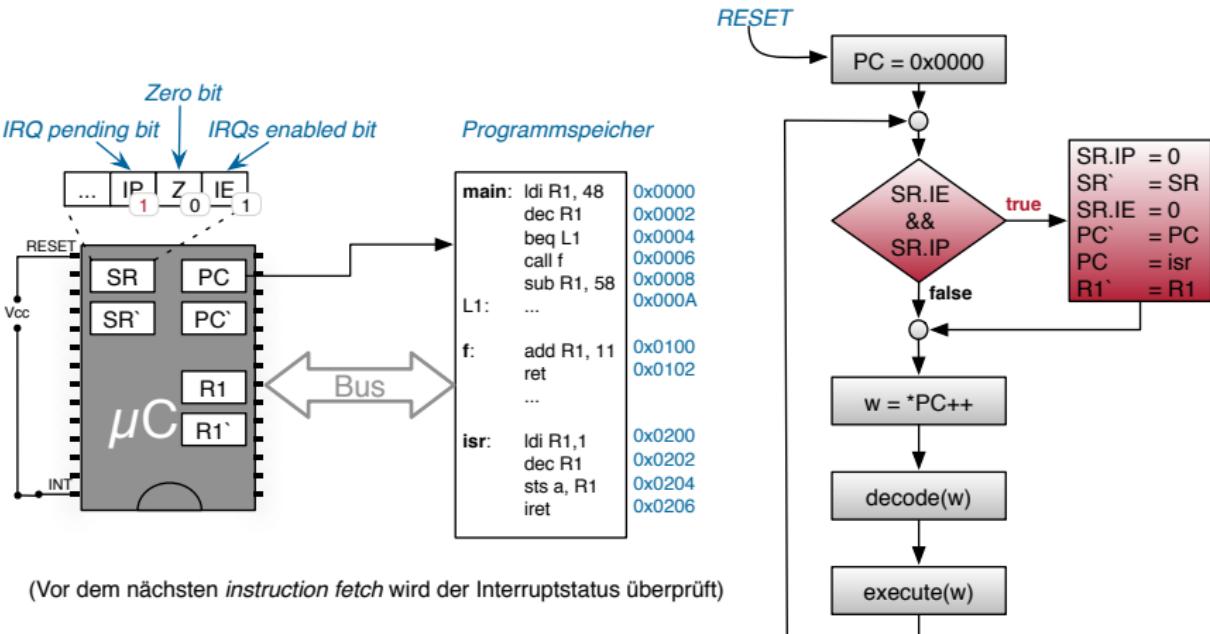
w: call <func>
PC' = PC
PC = func

w: ret
PC = PC'
SR = SR'

w: iret
R1 = R1'



Ablauf eines Interrupts – Details



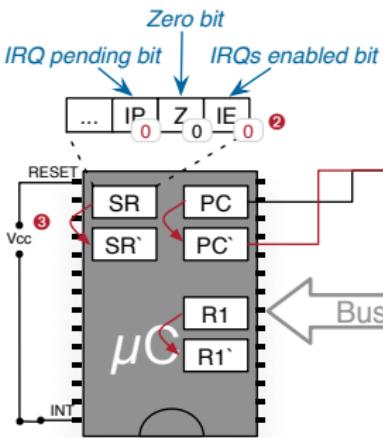
w: call <func>
PC' = PC
PC = func

w: ret
PC = PC'
R1 = R1'

w: iret
SR = SR'
PC = PC'
R1 = R1'

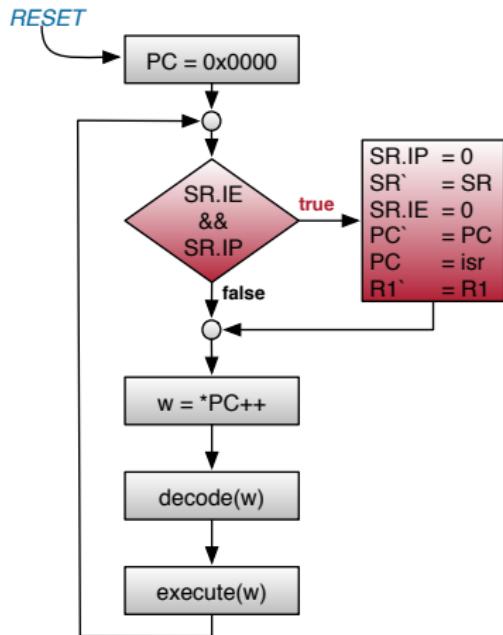


Ablauf eines Interrupts – Details



Programmspeicher

```
main: ldi R1, 48  
      dec R1  
      beq L1  
      call f  
      sub R1, 58  
      ...  
L1:  
f:   add R1, 11  
      ret  
      ...  
isr: ldi R1, 1  
      dec R1  
      sts a, R1  
      iret
```



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

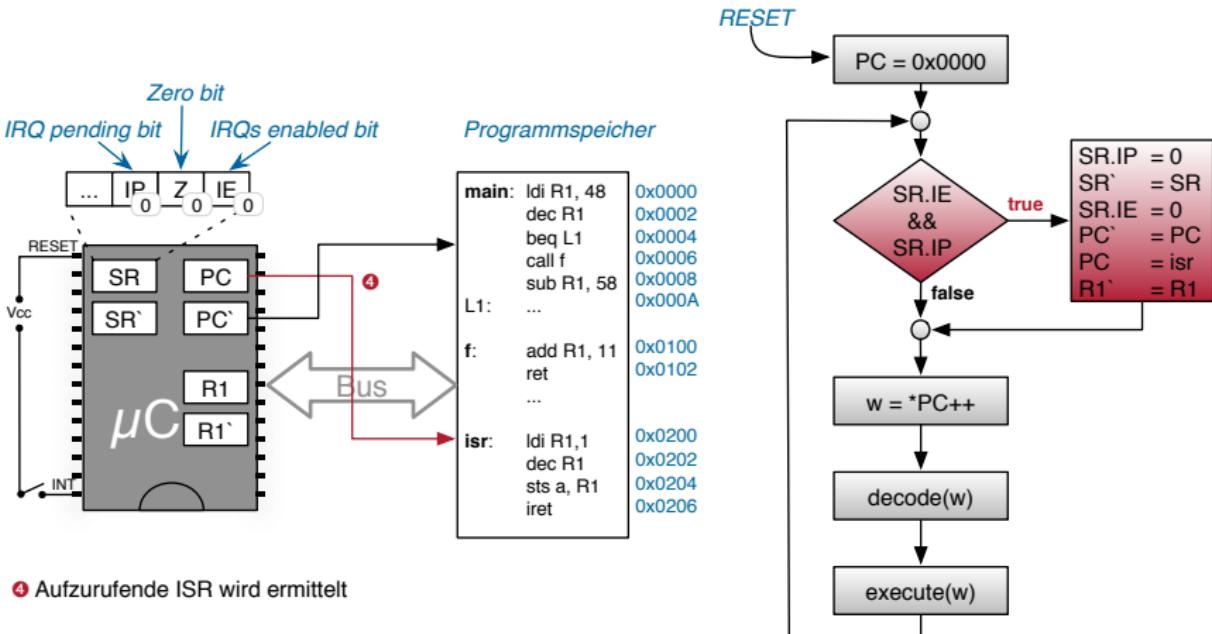
w: call <func>
 $PC' = PC$
 $PC = func$

w: ret
 $PC = PC'$

w: iret
 $SR = SR'$
 $PC = PC'$
 $R1 = R1'$



Ablauf eines Interrupts – Details

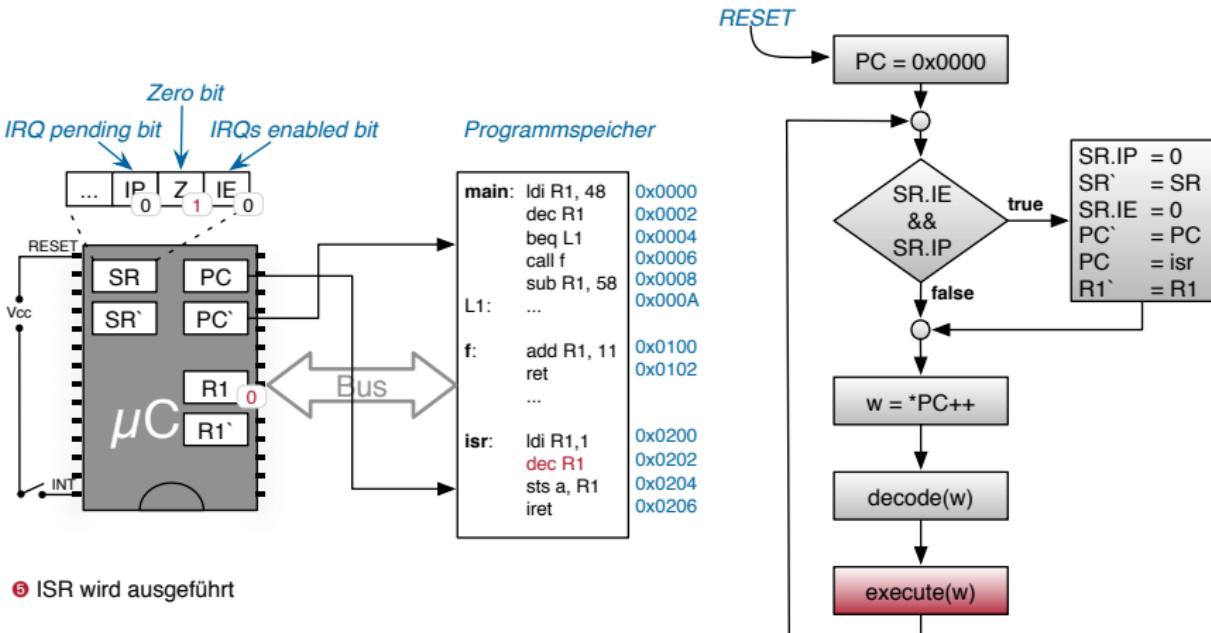


w: call <func>
PC' = PC
PC = func

w: ret
PC = PC'
R1 = R1'

w: iret
SR = SR'
PC = PC'
R1 = R1'

Ablauf eines Interrupts – Details



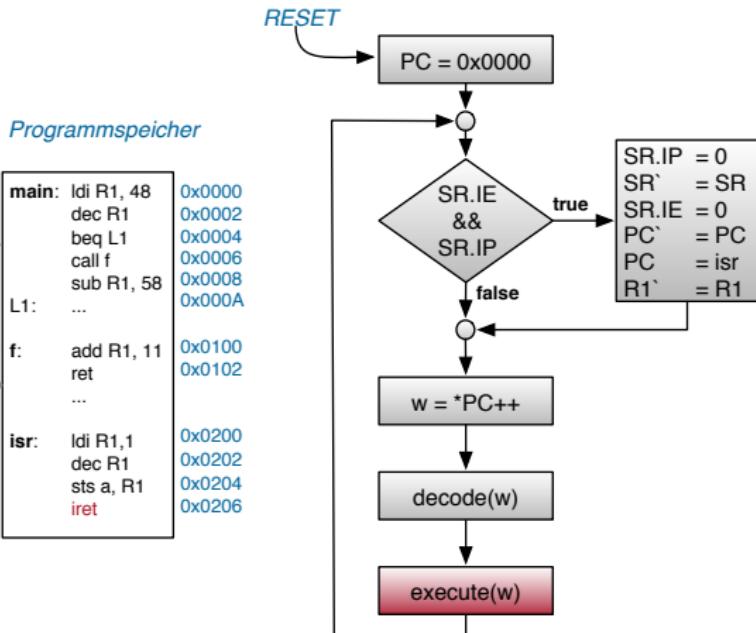
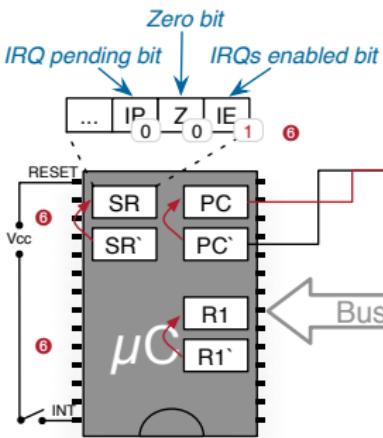
⑤ ISR wird ausgeführt

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

w: ret
PC = PC'
R1 = R1'

w: iret
SR = SR'
PC = PC'
R1 = R1'

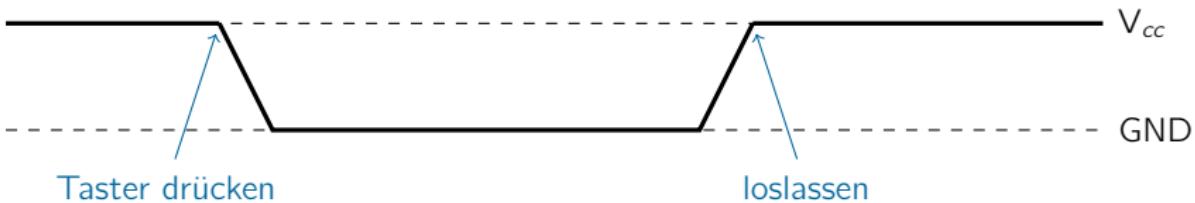
Ablauf eines Interrupts – Details



- ⑥ ISR terminiert mit *iret*-Befehl
- Registerinhalte werden restauriert
 - Zustellung von Interrupts wird reaktiviert
 - Das Anwendungsprogramm wird fortgesetzt

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



Interruptsteuerung beim AVR ATmega

■ IRQ-Quellen beim ATmega32

- 21 IRQ-Quellen
- einzeln de-/aktivierbar
- IRQ \rightsquigarrow Sprung an Vektor-Adresse

(IRQ \mapsto *Interrupt ReQuest*)

[1, S. 45]

■ Verschaltung SPIBoard

(\hookrightarrow 14-14 \hookrightarrow 2-4)

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

Vector No.	Program Address ⁽²⁾	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



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]

7	6	5	4	3	2	1	0
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]

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

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.



- **Schritt 1:** Installation der **Interrupt-Service-Routine**
 - ISR in Hochsprache \rightsquigarrow 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
}
```



■ 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)



■ 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.



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;

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

```
// photo sensor is connected
// to INT2

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



- 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:  
    ... ; 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 $\text{cars}=5$ und an **dieser Stelle** tritt der IRQ () auf
 - main hat den Wert von cars (5) bereits in Register gelesen (Register \mapsto lokale Variable)
 - INT2_vect wird ausgeführt
 - Register werden gerettet
 - cars wird inkrementiert $\leadsto \text{cars}=6$
 - Register werden wiederhergestellt
 - main übergibt den **veralteten Wert** von cars (5) an send
 - main nullt cars \leadsto **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 $\text{cars} = 255$ und an **dieser Stelle** tritt der IRQ (**⚡**) auf
 - main hat bereits $\text{cars} = 255$ Autos mit send gemeldet
 - main hat bereits das **High-Byte** von cars genullt
 - ~ $\text{cars} = 255$, $\text{cars.lo} = 255$, $\text{cars.hi} = 0$
 - INT2_vect wird ausgeführt
 - ~ cars wird gelesen und inkrementiert, **Überlauf ins High-Byte**
 - ~ $\text{cars} = 256$, $\text{cars.lo} = 0$, $\text{cars.hi} = 1$
 - main nullt das **Low-Byte** von cars
 - ~ $\text{cars} = 256$, $\text{cars.lo} = 0$, $\text{cars.hi} = 1$
 - ~ Beim nächsten send werden **255 Autos zu viel gemeldet**



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?



- 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, **Dornrösenschlaf**



Lost-Wakeup: Dornrösenschenschlaf 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 Interruptsperren absichern?)
- Problem: Vor `sleep_cpu()` müssen IRQs freigegeben werden!
- Funktioniert dank spezieller Hardwareunterstützung:
 ~ Befehlssequenz `sei`, `sleep` wird von der CPU **atomar** ausgeführt



- 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 μ C

```
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.



Speicherorganisation auf einem μ C

```
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

Flash / ROM

.data	s=3 b=1
.rodata	c=2
.text	main

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

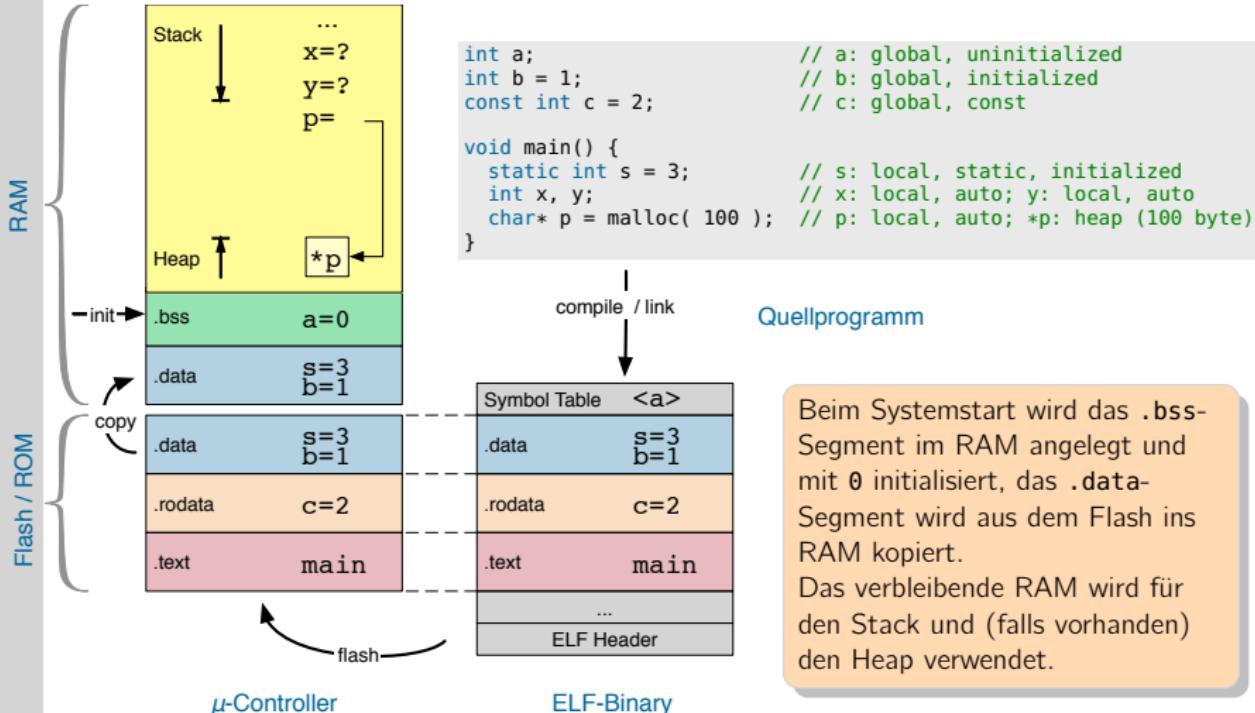
Zur Installation auf dem μ C werden .text und .[ro]data in den Flash-Speicher des μ C geladen.

flash

μ -Controller

ELF-Binary

Speicherorganisation auf einem μ C



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

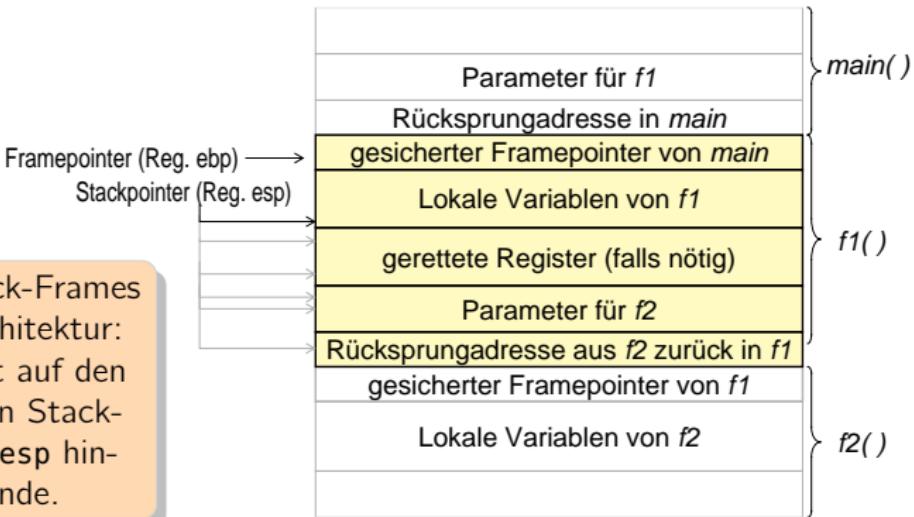
Dynamische Speicherallokation: Heap

- **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! **)
    }
}
```

- 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**

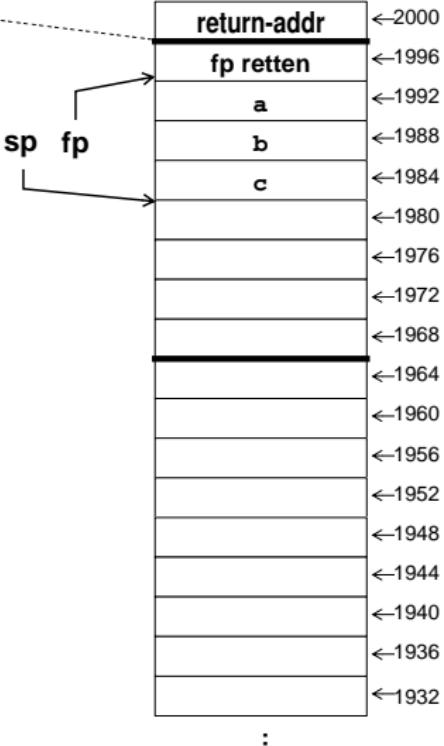
Aufbau eines Stack-Frames auf der IA-32-Architektur:
Register ebp zeigt auf den Beginn des aktiven Stack-Frames; Register esp hinter das aktuelle Ende.



Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

Stack-Frame für
main erstellen
 $\&a = fp-4$
 $\&b = fp-8$
 $\&c = fp-12$



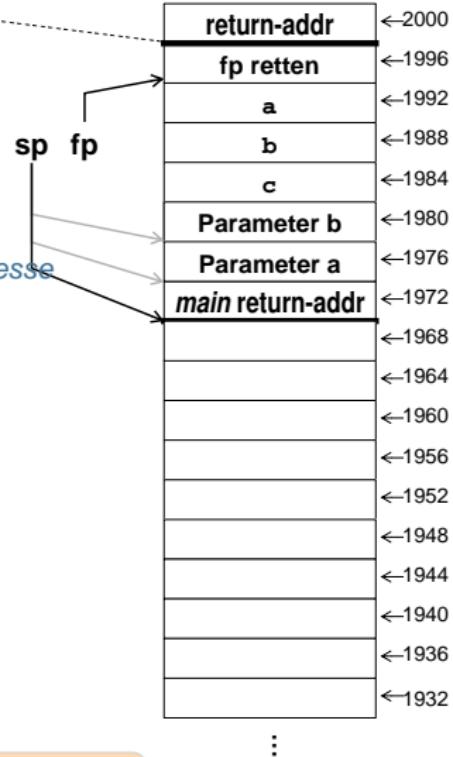
Beispiel hier für 32-Bit-Architektur (4-Byte `ints`), `main()` wurde soeben betreten



Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

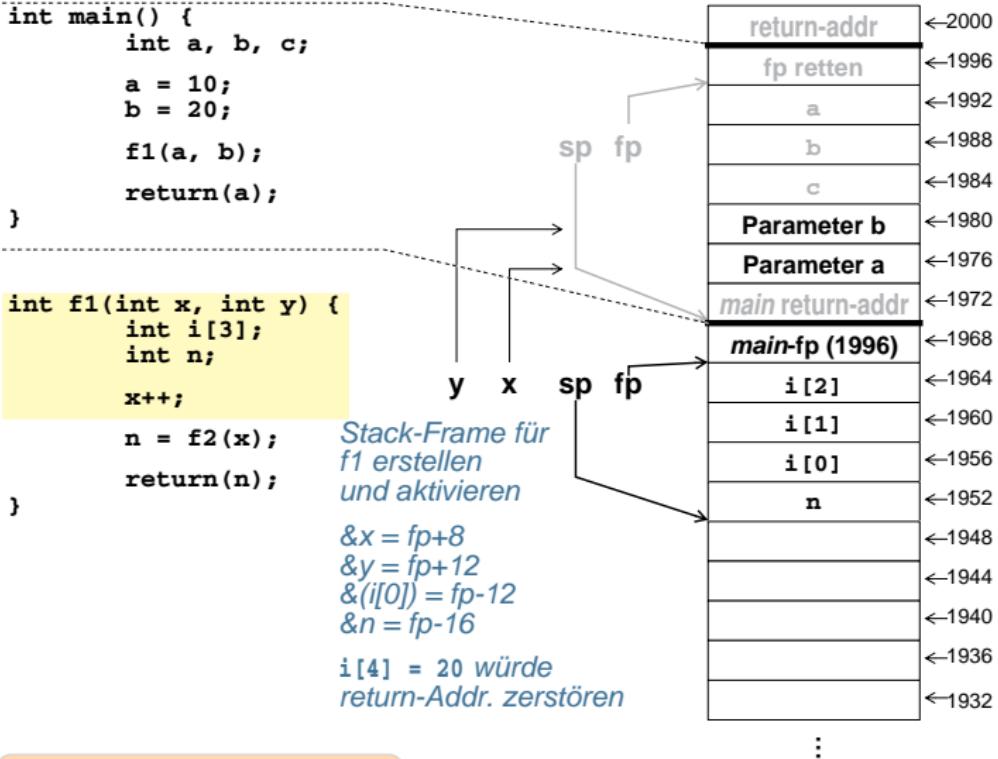
Parameter
auf Stack legen
Bei Aufruf
Rückprungadresse
auf Stack legen



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



Stack-Aufbau bei Funktionsaufrufen



Stack-Aufbau bei Funktionsaufrufen

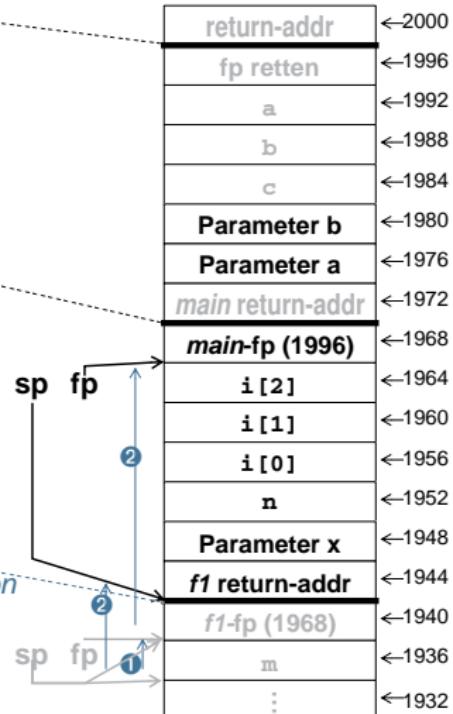
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return(n);  
}
```

```
int f2(int z) {  
    int m;  
    m = 100;  
    return(z+1);  
}
```

Stack-Frame von
f2 abräumen

① $sp = fp$
② $fp = pop(sp)$



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

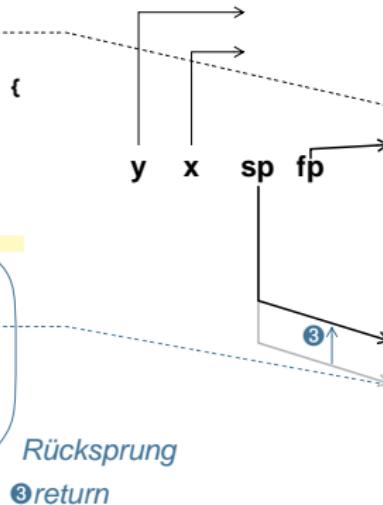


Stack-Aufbau bei Funktionsaufrufen

```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```

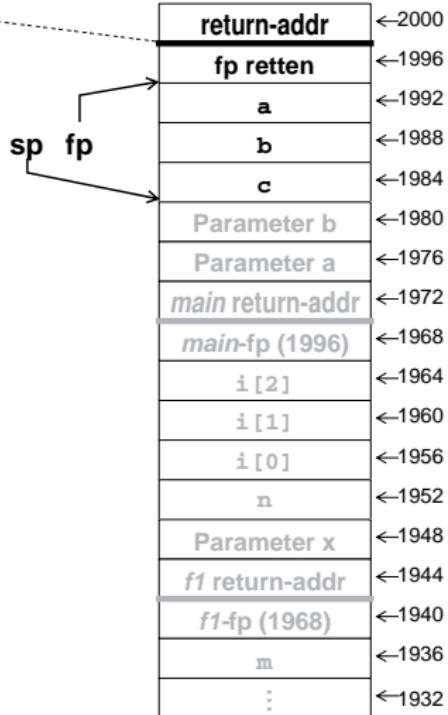
```
int f2(int z) {
    int m;
    m = 100;
    return(z+1);
}
```



f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```



zurück in main()

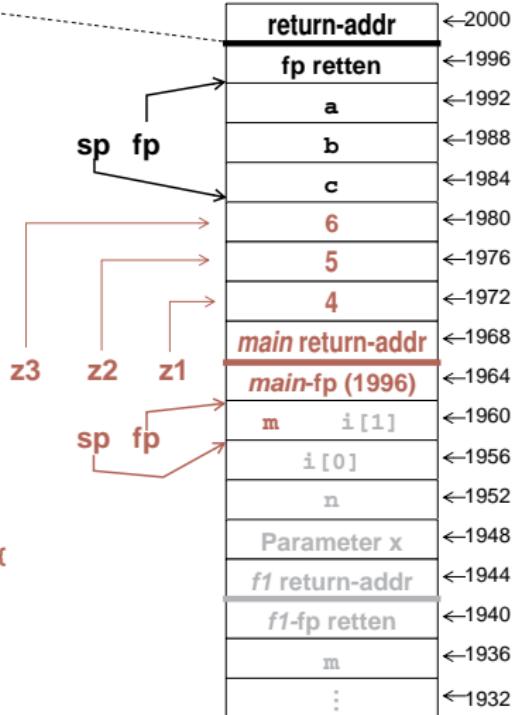


Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    f3(4,5,6);  
}
```

was wäre, wenn man nach
f1 jetzt eine Funktion f3
aufrufen würde?

```
int f3(int z1, int z2, int z3) {  
    int m;  
  
    return(m);  
}
```



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



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! ↪ **1-3**)

```
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

