

# Systemnahe Programmierung in C (SPiC)

## 18 Unterbrechungen

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2022

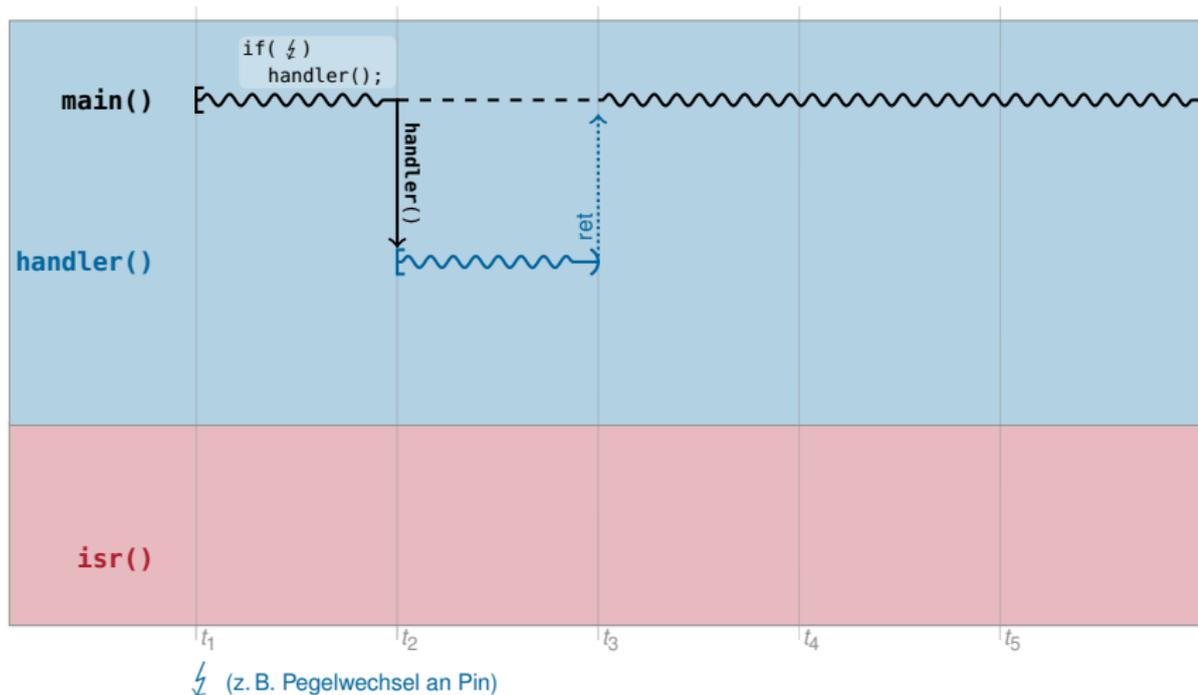
<http://sys.cs.fau.de/lehre/SS22/spic>



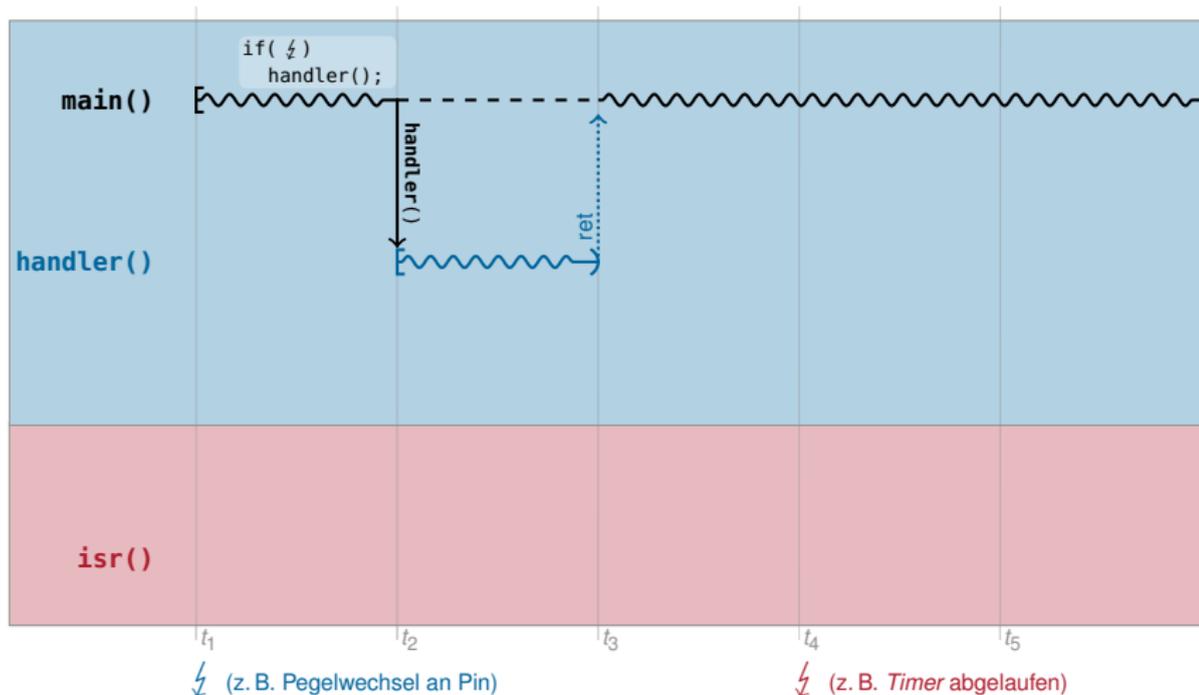
- Bei einem **Peripheriegerät** tritt ein Ereignis (⚡) auf ↔ 17-3
  - Signal an einem Port-Pin wechselt von *low* auf *high*
  - Ein *Timer* ist abgelaufen
  - Ein A/D-Wandler hat einen neuen Wert vorliegen
  - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
  - **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.



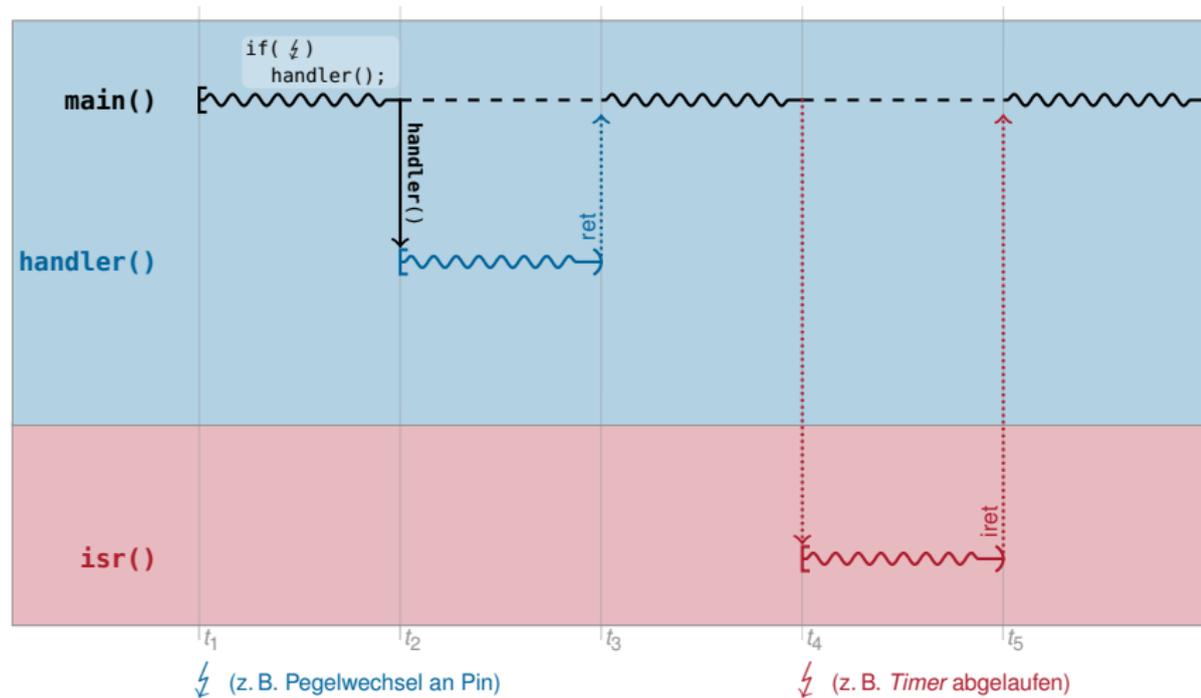
# Interrupt $\mapsto$ Funktionsaufruf „von außen“



# Interrupt $\mapsto$ Funktionsaufruf „von außen“



# Interrupt $\mapsto$ Funktionsaufruf „von außen“



# 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  $\leadsto$  hohe Prozessorlast  $\leadsto$  **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  $\leadsto$  Synchronisation erforderlich
    - Programmverhalten **schwer vorhersagbar**



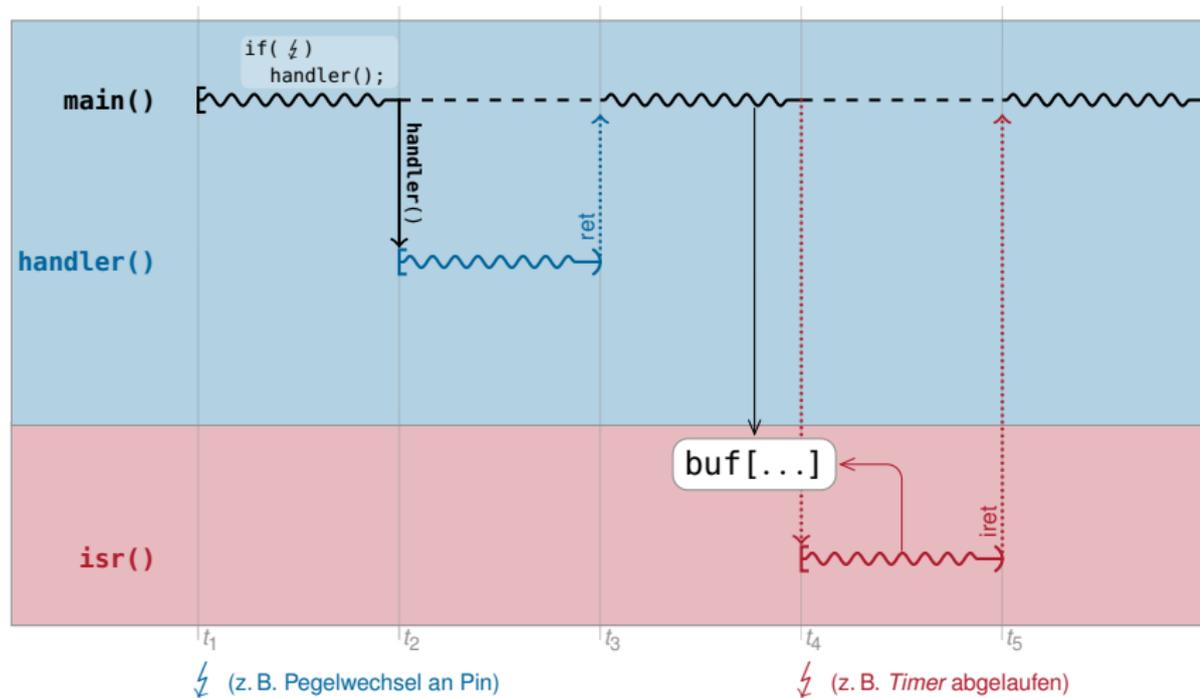
# Polling vs. Interrupts – Vor- und Nachteile

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

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



# Interrupt $\mapsto$ unvorhersagbarer Aufruf „von außen“



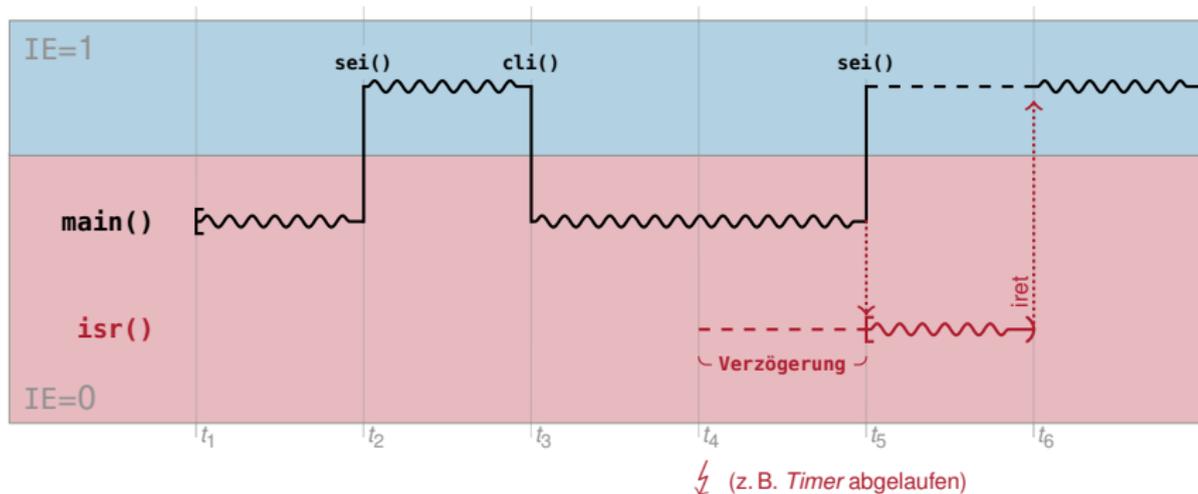
# Interruptsperrern

- 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  $\mapsto$  *Interrupt ReQuest*



# Interruptsperrren: 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  $IE=0 \leadsto$  Bearbeitung ist unterdrückt, IRQ wird gepuffert

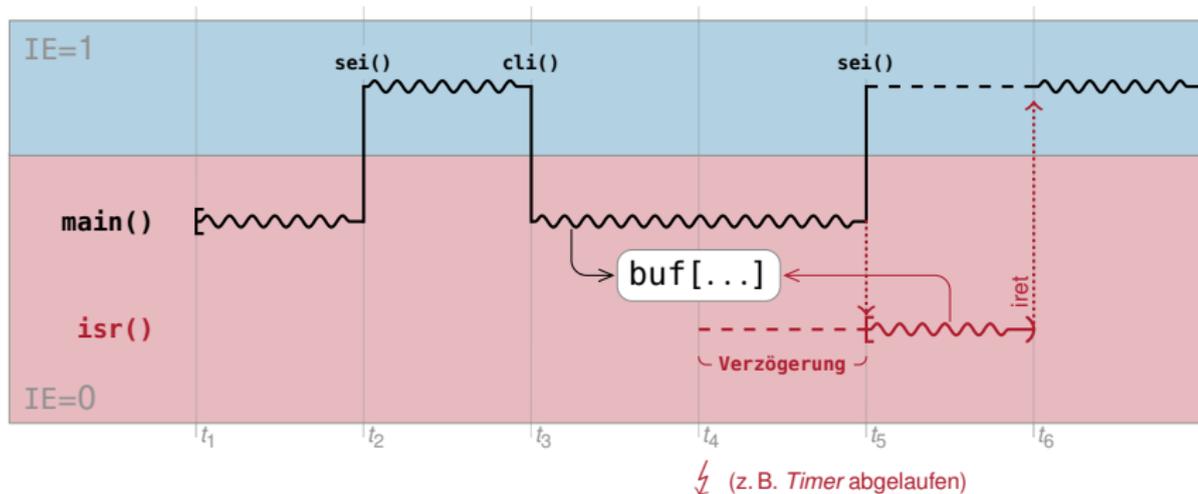
$t_5$  `main()` gibt IRQs frei ( $IE=1$ )  $\leadsto$  gepufferter IRQ „schlägt durch“

$t_5-t_6$  Während der ISR-Bearbeitung sind die IRQs gesperrt ( $IE=0$ )

$t_6$  Unterbrochenes `main()` wird fortgesetzt



# 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`  $\leadsto$  Bearbeitung ist unterdrückt, IRQ wird gepuffert

$t_5$  `main()` gibt IRQs frei (`IE=1`)  $\leadsto$  gepufferter IRQ „schlägt durch“

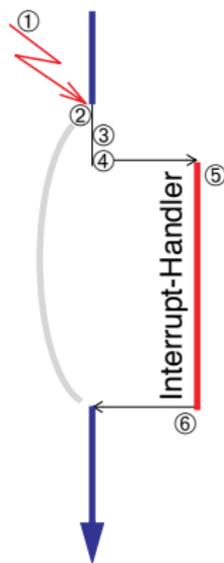
$t_5-t_6$  Während der ISR-Bearbeitung sind die IRQs gesperrt (`IE=0`)

$t_6$  Unterbrochenes `main()` wird fortgesetzt

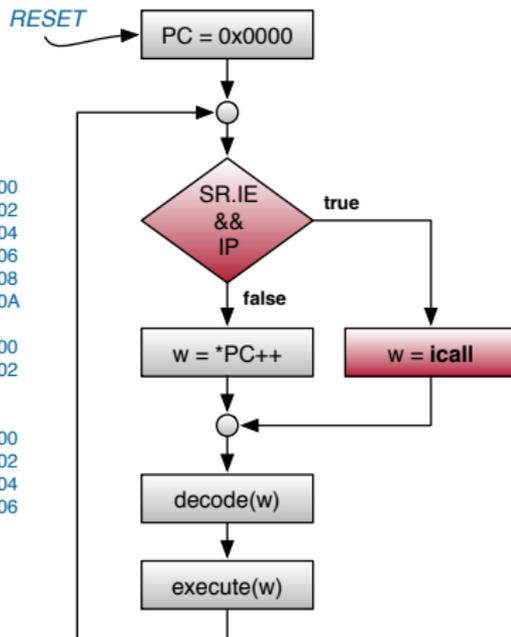
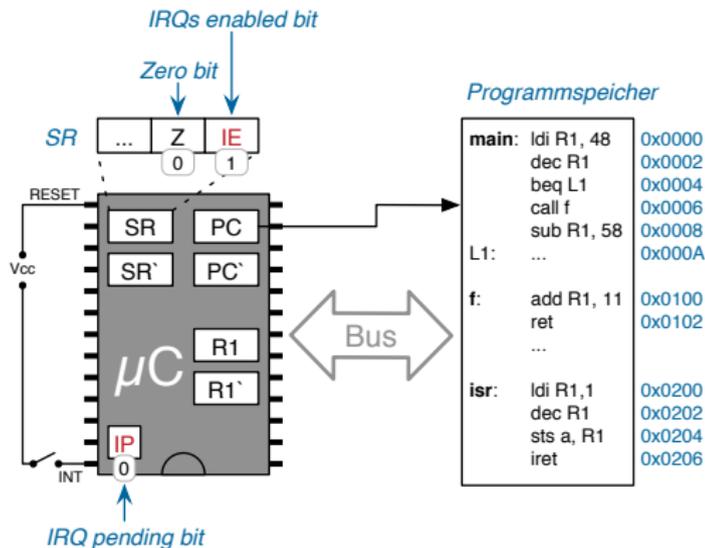


# Ablauf eines Interrupts – Überblick

- 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 Stapel)
  - PC und Statusregister automatisch von der Hardware
  - Vielzweckregister üblicherweise manuell in der ISR
- 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  $\leftrightarrow$  16-3

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

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

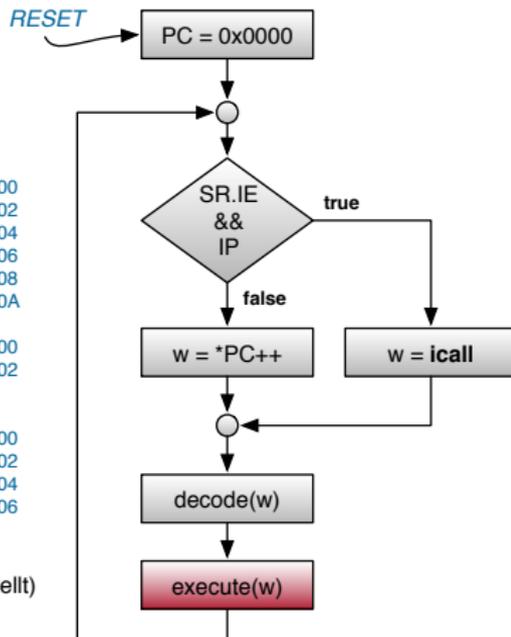
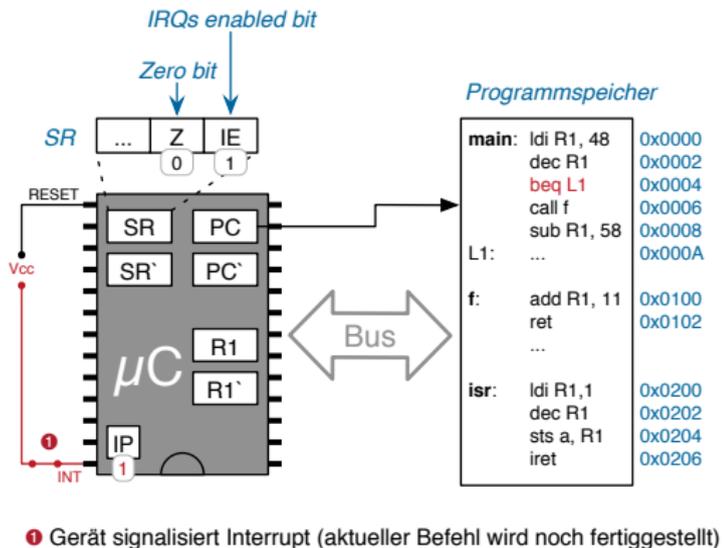
**w: ret**  
 PC = PC'

**w: icall**  
 SR' = SR  
 SR.IE = 0  
 IP = 0  
 PC' = PC  
 PC = isr  
 R1' = R1

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



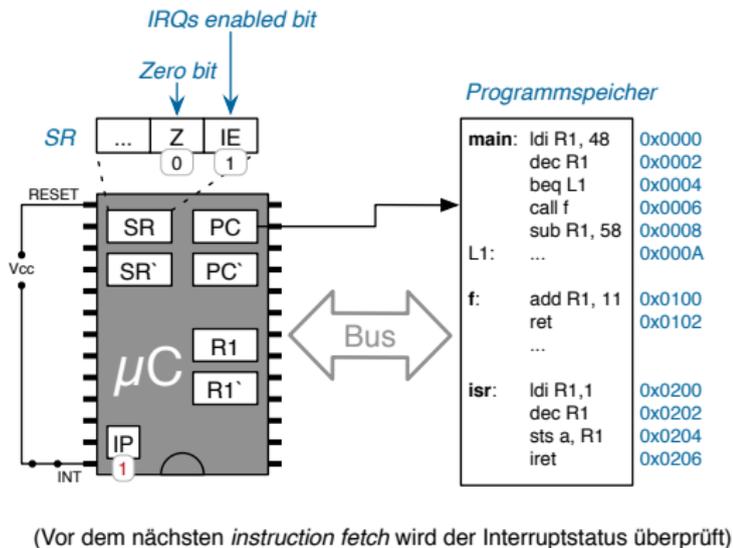
# Ablauf eines Interrupts – Details



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



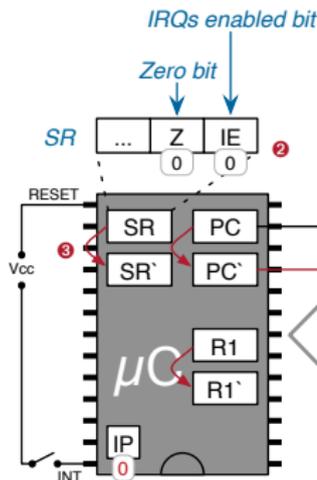
# Ablauf eines Interrupts – Details



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Ablauf eines Interrupts – Details



## Programmspeicher

```

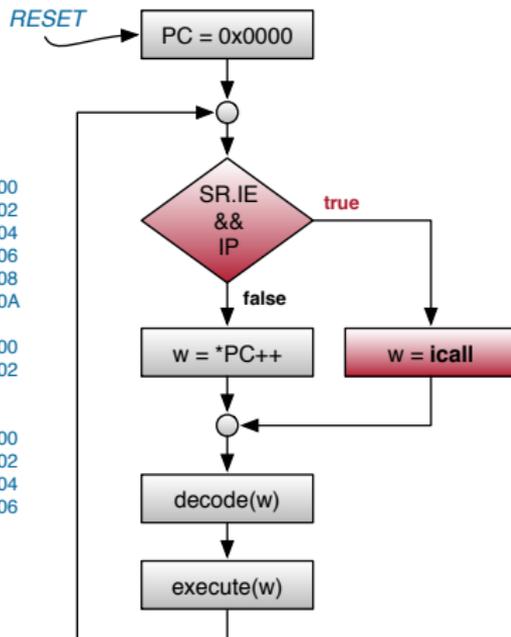
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

f:    add R1, 11 0x0100
      ret      0x0102
      ...

isr:  ldi R1, 1  0x0200
      dec R1    0x0202
      sts a, R1 0x0204
      ired     0x0206
    
```

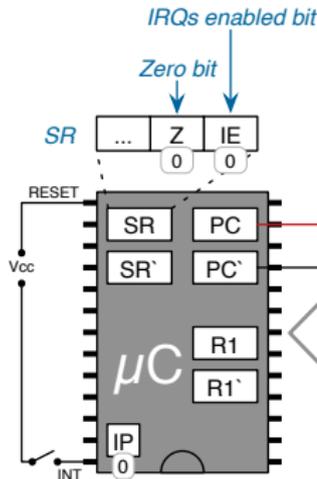
- ② Die Zustellung weiterer Interrupts wird verzögert
- ③ Registerinhalte werden gesichert



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Ablauf eines Interrupts – Details



## Programmspeicher

```

main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

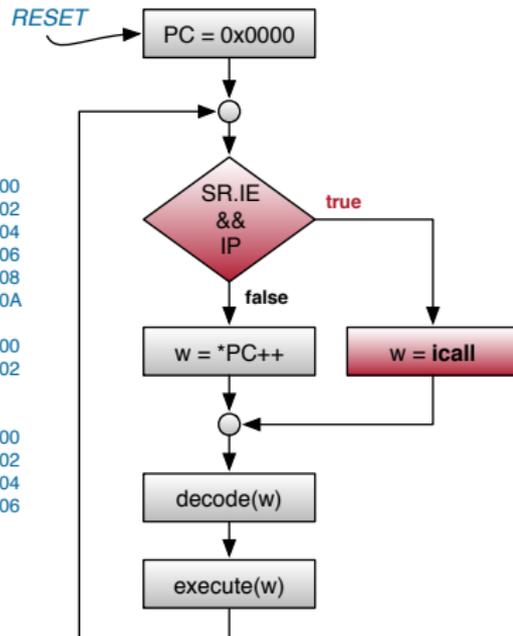
L1:   ...

f:    add R1, 11 0x0100
      ret       0x0102

...

ISR: ldi R1, 1 0x0200
      dec R1   0x0202
      sts a, R1 0x0204
      ired    0x0206
    
```

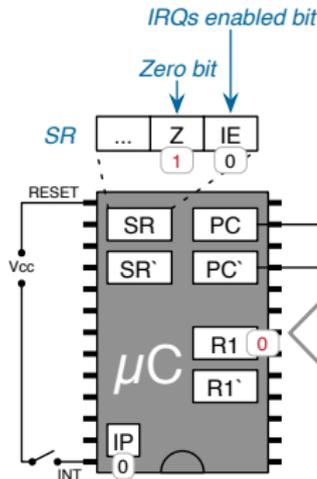
4 Aufzurufende ISR wird ermittelt



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Ablauf eines Interrupts – Details



ISR wird ausgeführt

## Programmspeicher

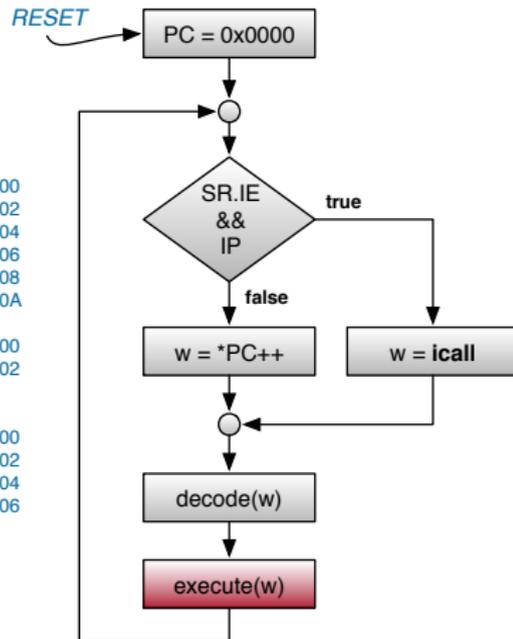
```

main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

f:    add R1, 11 0x0100
      ret       0x0102
      ...

isr:  ldi R1, 1  0x0200
      dec R1    0x0202
      sts a, R1 0x0204
      iret     0x0206
    
```



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

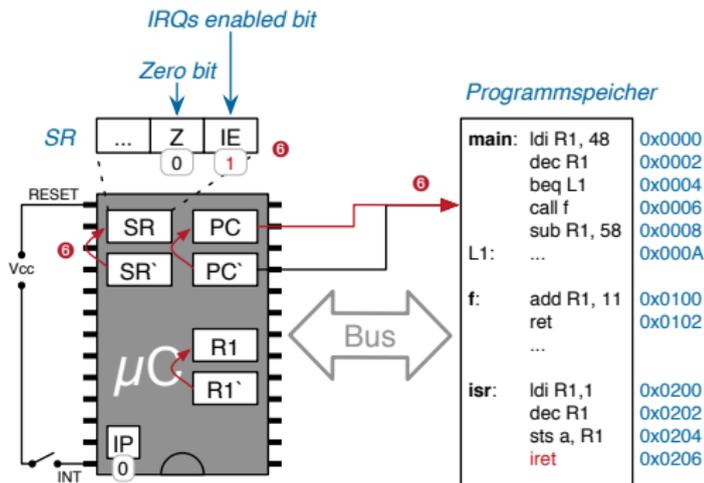
w: **ret**  
PC = PC'

w: **icall**  
SR' = SR  
SR.IE = 0  
IP = 0  
PC' = PC  
PC = isr  
R1' = R1

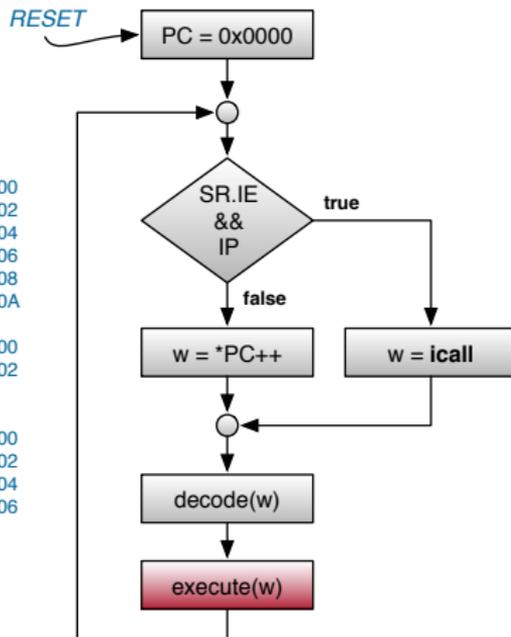
w: **iret**  
SR = SR'  
PC = PC'  
R1 = R1'



# Ablauf eines Interrupts – Details



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



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--

