

System-Level Programming

18 Interrupts

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

<http://sys.cs.fau.de/lehre/ss24>

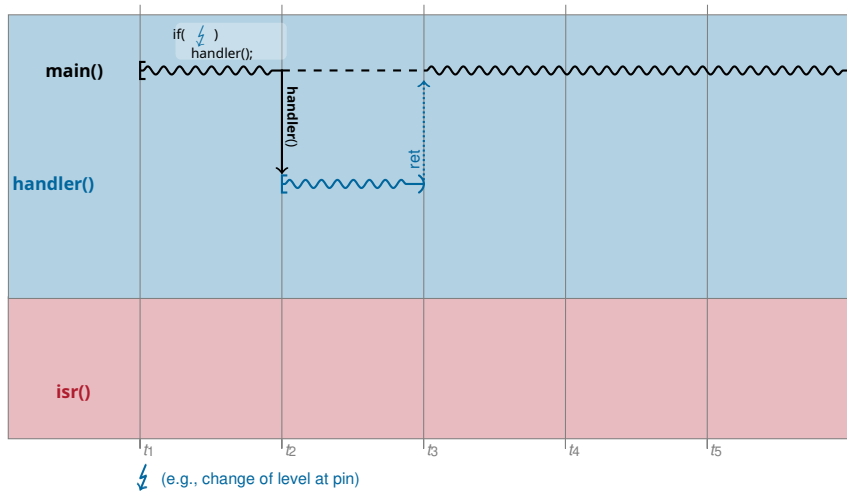


Interrupt Handling

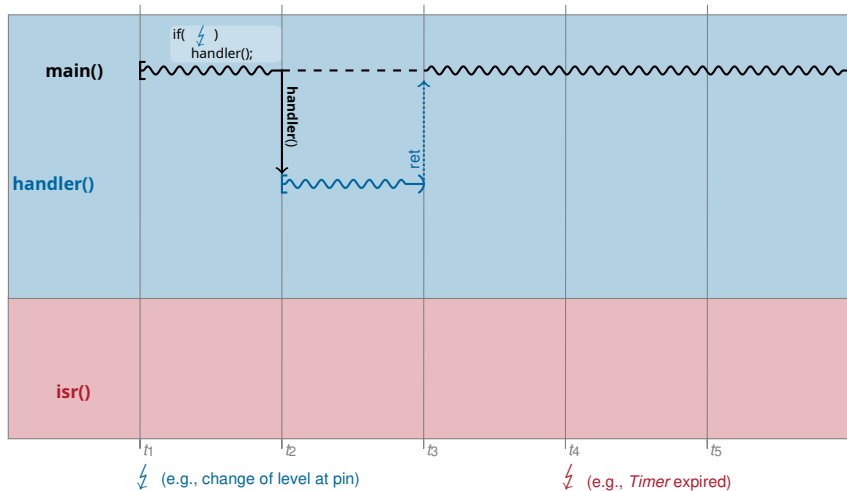
- An interrupt (⚡) occurs when a **peripheral device** signals ↔ 17-3
 - a level change at a port pin *low* to *high*
 - the expiration of a *timer*
 - the completion of an A/D conversion (new value available)
 - ...
- How is the program notified about the (concurrent) event?
- Two alternative procedures
 - **Polling:** The **program** regularly checks a state and calls a handler function if necessary.
 - **Interrupt:** Device “notifies” the **processor**; subsequently, the processor branches into a handler function.



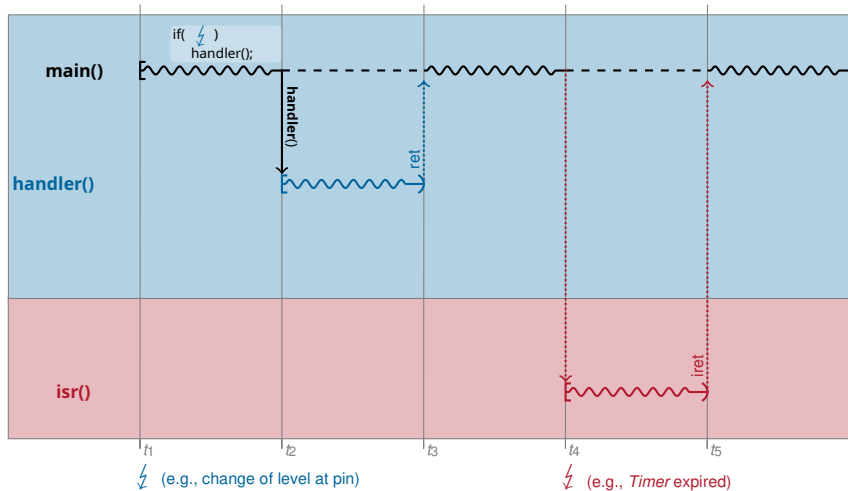
Interrupt \mapsto Function Call "from Outside"



Interrupt \mapsto Function Call "from Outside"



Interrupt \mapsto Function Call “from Outside”



Polling vs. Interrupts – (Dis-)Advantages

- Polling (→ “time-triggered system”)
 - Processing of events **synchronously** to the program flow
 - Detection of events scattered everywhere (missing separation of concerns)
 - Wasting processing resources (if usable for other things)
 - High polling frequency \leadsto high processor load \leadsto **high energy consumption**
 - + Implicit consistency in data by sequential program flow
 - + Program behaviour easily predictable
- Interrupts (→ “event-triggered system”)
 - Processing of events **asynchronously** to the program flow
 - + Event handlers can be easily separated in the source code
 - + Processor is only triggered when an event occurs
 - Higher complexity by concurrency \leadsto synchronisation required
 - Program behavior **unpredictable**



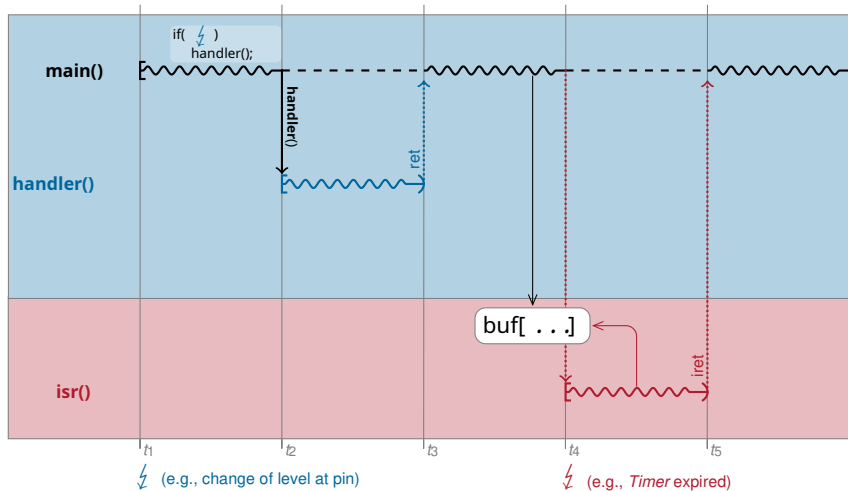
Polling vs. Interrupts – (Dis-)Advantages

- Polling (↳ “time-triggered system”)
 - Processing of events **synchronously** to the program flow
 - Detection of events scattered everywhere (missing separation of concerns)
 - Wasting processing resources (if usable for other things)
 - High polling frequency \leadsto high processor load \leadsto **high energy consumption**
 - + Implicit consistency in data by sequential program flow
 - + Program behaviour easily predictable
- Interrupts (↳ “event-triggered system”)
 - Processing of events **asynchronous** to the program flow
 - + Event handlers can be easily separated in the source code
 - + Processor is only triggered when an event occurs
 - Higher complexity by concurrency \leadsto synchronisation required
 - Program behavior **unpredictable**

Both methods provide specific (dis-)advantages
 \leadsto Which one to choose depends on concrete scenario



Interrupt \mapsto Unpredictable Call "from Outside"



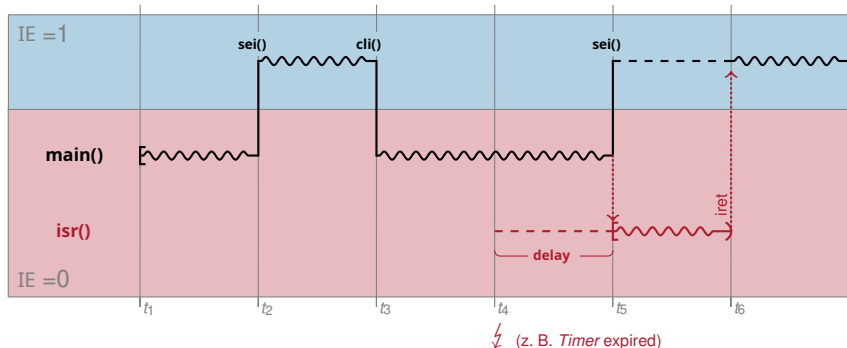
Disabling Interrupts

- Notification about new interrupts can be **disabled** by software
 - Used for **synchronisation** with ISRs
 - Single ISR: Bit in device-specific control register
 - All ISRs: Bit (**IE**, *Interrupt Enable*) in a status register of CPU
- Pending IRQs are (usually) buffered
 - At most one per source!
 - **During longer disabled time spans, IRQs can get lost!**
- The **IE** bit is affected by:
 - Processor instructions: `cli: IE ← 0` (*clear interrupt*, IRQs disabled)
`sei: IE ← 1` (*set interrupt*, IRQs enabled)
 - After a RESET: **IE=0** \rightsquigarrow IRQs are always disabled at the begin of the main program
 - When entering an ISR: **IE=0** \rightsquigarrow IRQs are disabled during handling of other interrupts

IRQ \mapsto *Interrupt ReQuest*



Interrupt Blocking: Example



t_1 At the begin of `main()`, all IRQs are disabled ($IE=0$)

t_2, t_3 With `sei()` / `cli()` IRQs can be enabled ($IE=1$) / disabled

t_4 ⚡ but $IE=0 \rightsquigarrow$ handling is blocked, IRQ is buffered

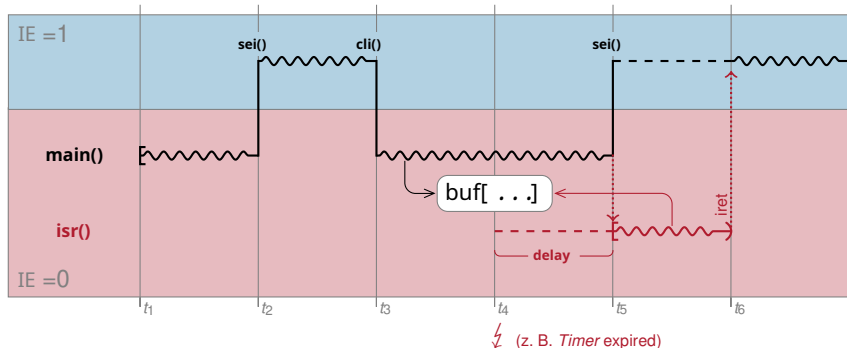
t_5 `main()` unblocks IRQs ($IE=1$) \rightsquigarrow buffered IRQ “gets through”

t_5-t_6 During handling of the ISR, all IRQs are blocked again ($IE=0$)

t_6 Interrupted `main()` is resumed



Interrupt Blocking: Example

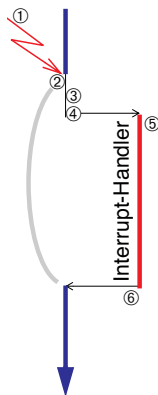


- t_1 At the begin of `main()`, all IRQs are disabled ($IE=0$)
- t_2, t_3 With `sei()` / `cli()` IRQs can be enabled ($IE=1$) / disabled
- t_4 ⚡ but $IE=0 \rightsquigarrow$ handling is blocked, IRQ is buffered
- t_5 `main()` unblocks IRQs ($IE=1$) \rightsquigarrow buffered IRQ “gets through”
- t_5-t_6 During handling of the ISR, all IRQs are blocked again ($IE=0$)
- t_6 Interrupted `main()` is resumed

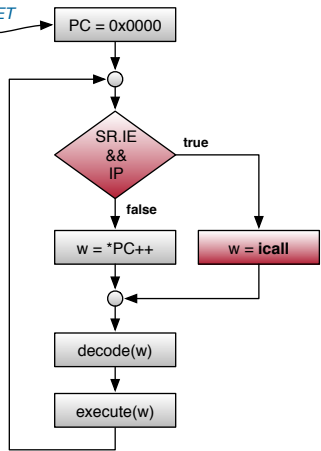
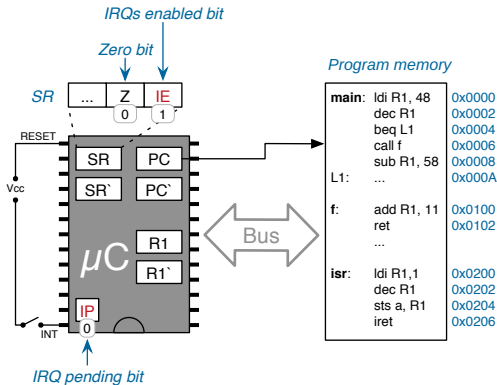


Procedure of an Interrupt – Overview

- 1 Device signals an interrupt
 - Current program is “immediately” interrupted (prior to the next machine instruction, with $IE=1$)
- 2 Notification of further interrupts is blocked ($IE=0$)
 - Interrupts that occur during this time are buffered (at most once per source!)
- 3 Content of registers is stored (e. g., on the stack)
 - PC and status registers automatically by the hardware
 - Multi-purpose registers usually manually in the ISR
- 4 Determination of to be called ISR (interrupt handler)
- 5 ISR gets executed
- 6 ISR terminates with “return from interrupt” instruction
 - Content of registers is restored
 - Notification of interrupts again unblocked/enabled ($IE=1$)
 - Program gets resumed



Procedure of an Interrupt – Details



■ Here as an extension of the simplified pseudo processor ↔ **16-3**

- Only one source for interrupts
- All registers are saved by the hardware

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

w: ret
 PC = PC'

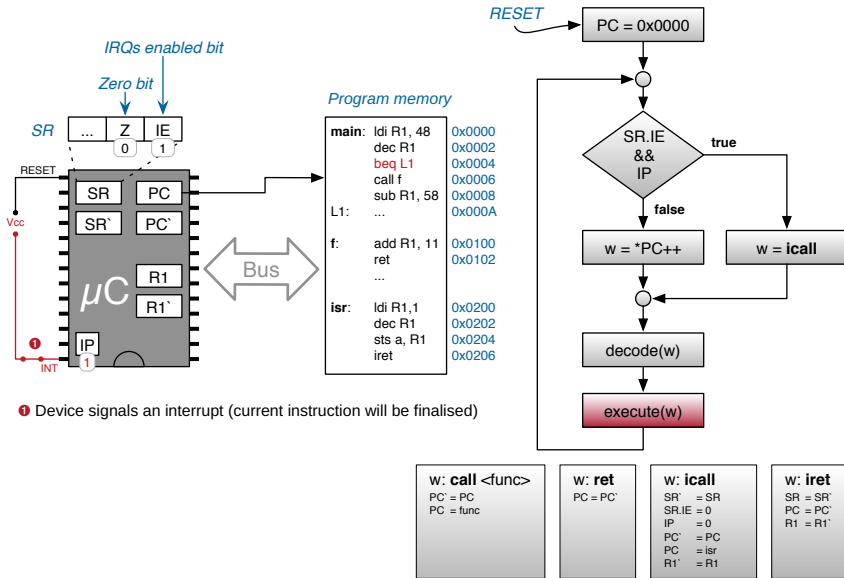
w: icall
 SR' = SR
 SR.IE = 0
 IP = 0
 PC' = PC
 PC = isr
 R1' = R1

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

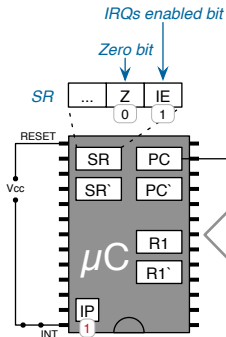
18-IRQ-Konzept_en



Procedure of an Interrupt – Details



Procedure of an Interrupt – Details

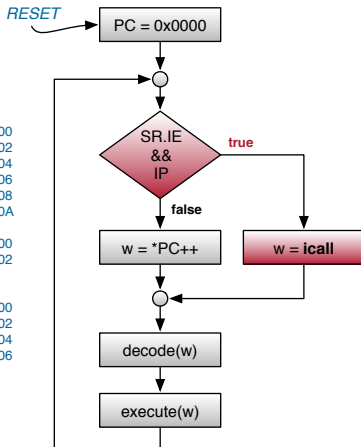


Program memory

```

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
    
```

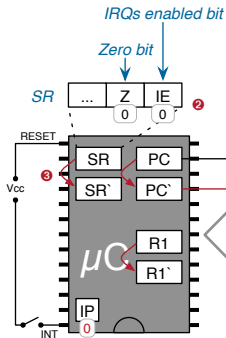
(Before the next *instruction fetch* the interrupt state is checked)



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'
--	---------------------------	--	--



Procedure of an Interrupt – Details



Program memory

```

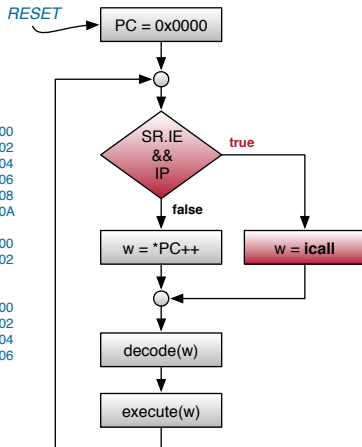
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

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

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

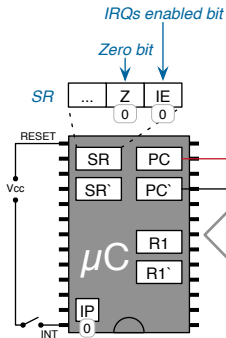
- ② The delivery of further interrupts is stalled
- ③ Register contents get saved



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'
--	---------------------------	--	--



Procedure of an Interrupt – Details

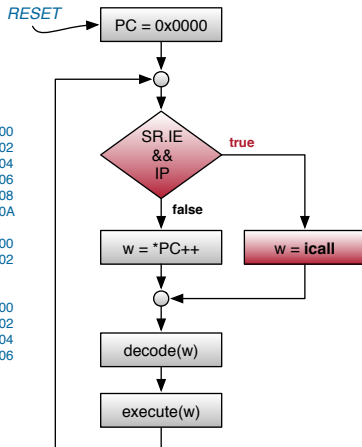


4 ISR to be called is determined

Program memory

```

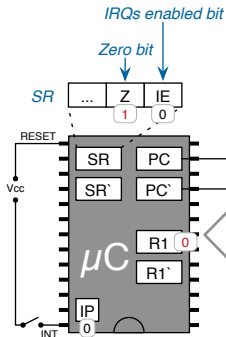
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A
L1:   ...
f:    add R1, 11 0x0100
      ret       0x0102
      ...
isr:  ldi R1, 1  0x0200
      dec R1    0x0202
      sts a, R1 0x0204
      ired     0x0206
    
```



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'
--	---------------------------	--	--



Procedure of an Interrupt – Details



⑥ ISR gets executed

Program memory

```

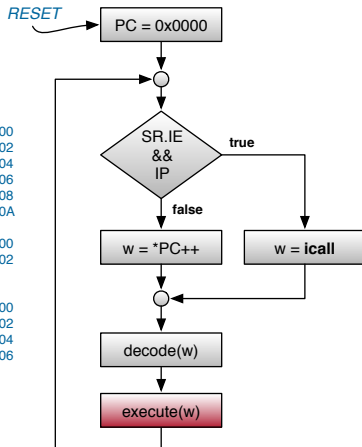
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

f:    add R1, 11 0x0100
      ret      0x0102

...

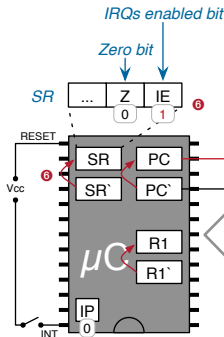
isr:  ldi R1, 1  0x0200
      dec R1    0x0202
      sts a, R1 0x0204
      ired     0x0206
    
```



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'
--	---------------------------	--	--



Procedure of an Interrupt – Details



Program memory

```

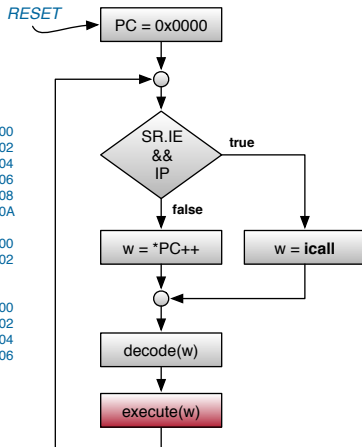
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

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

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

- ⑥ ISR terminates with *ired*-instruction
- Register contents are restored
 - Delivery of interrupts is reactivated
 - Program is resumed



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'
--	---------------------------	--	--

