System-Level Programming

18 Interrupts

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

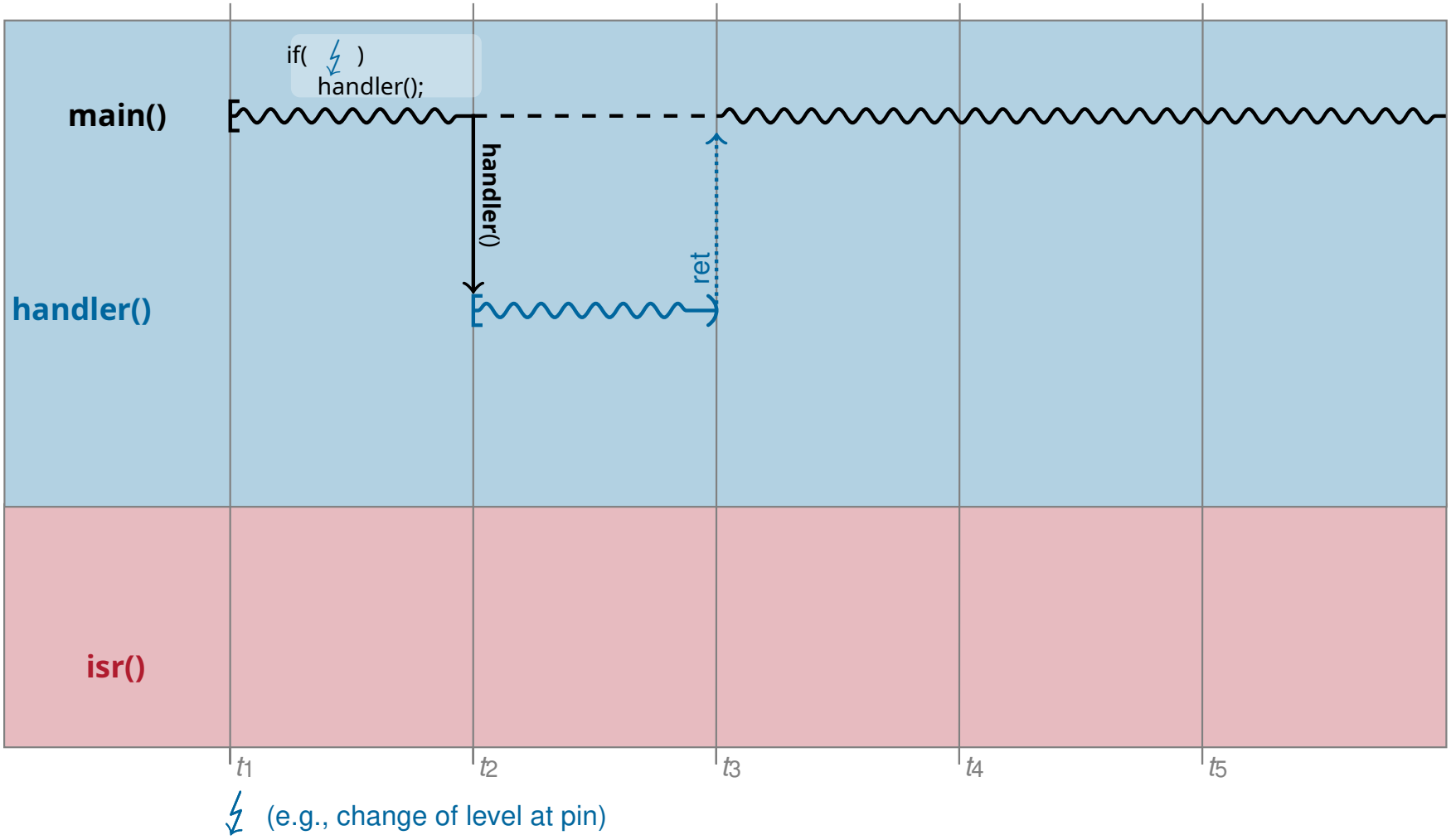
Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

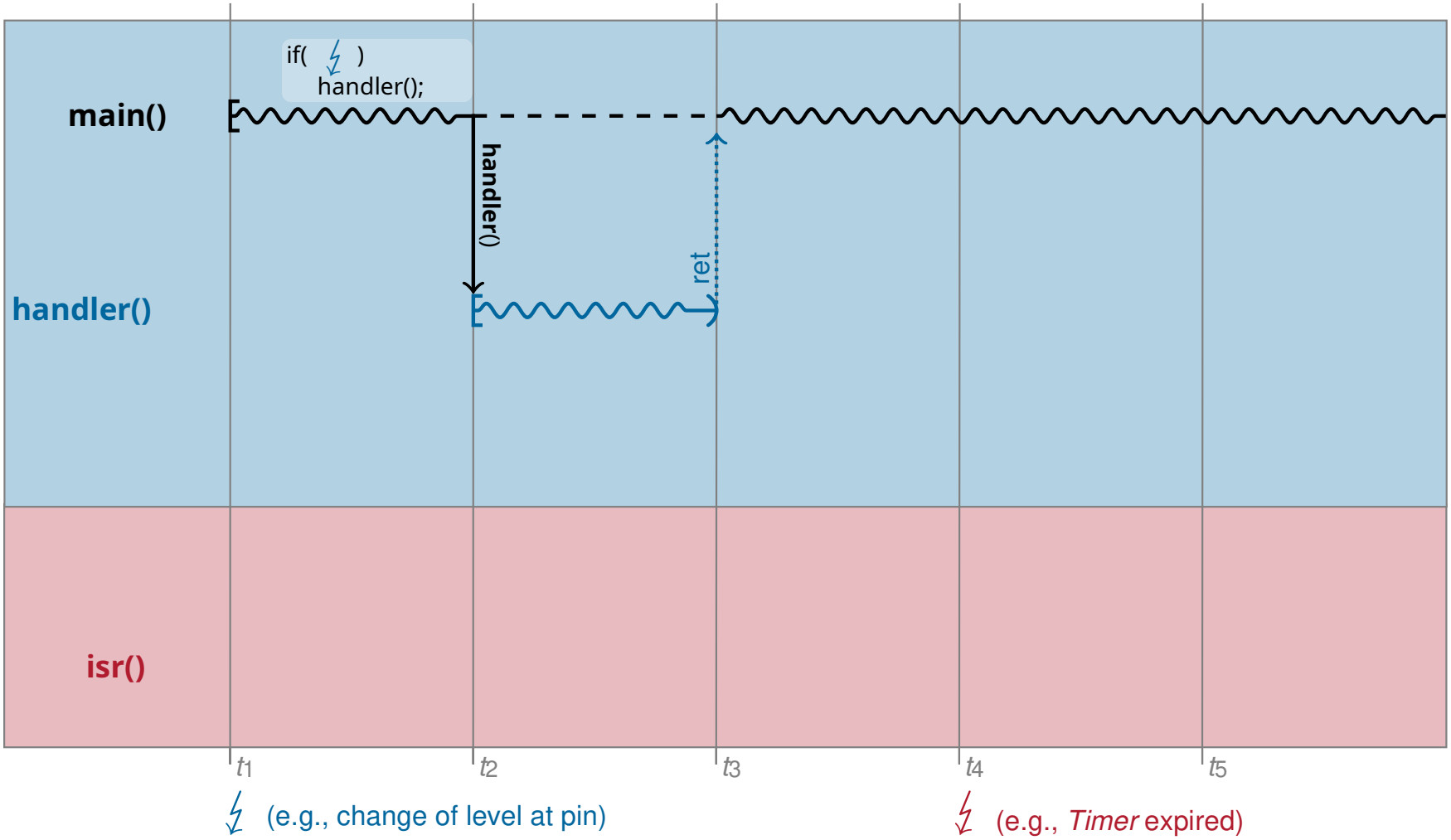
Interrupt Handling

- 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.

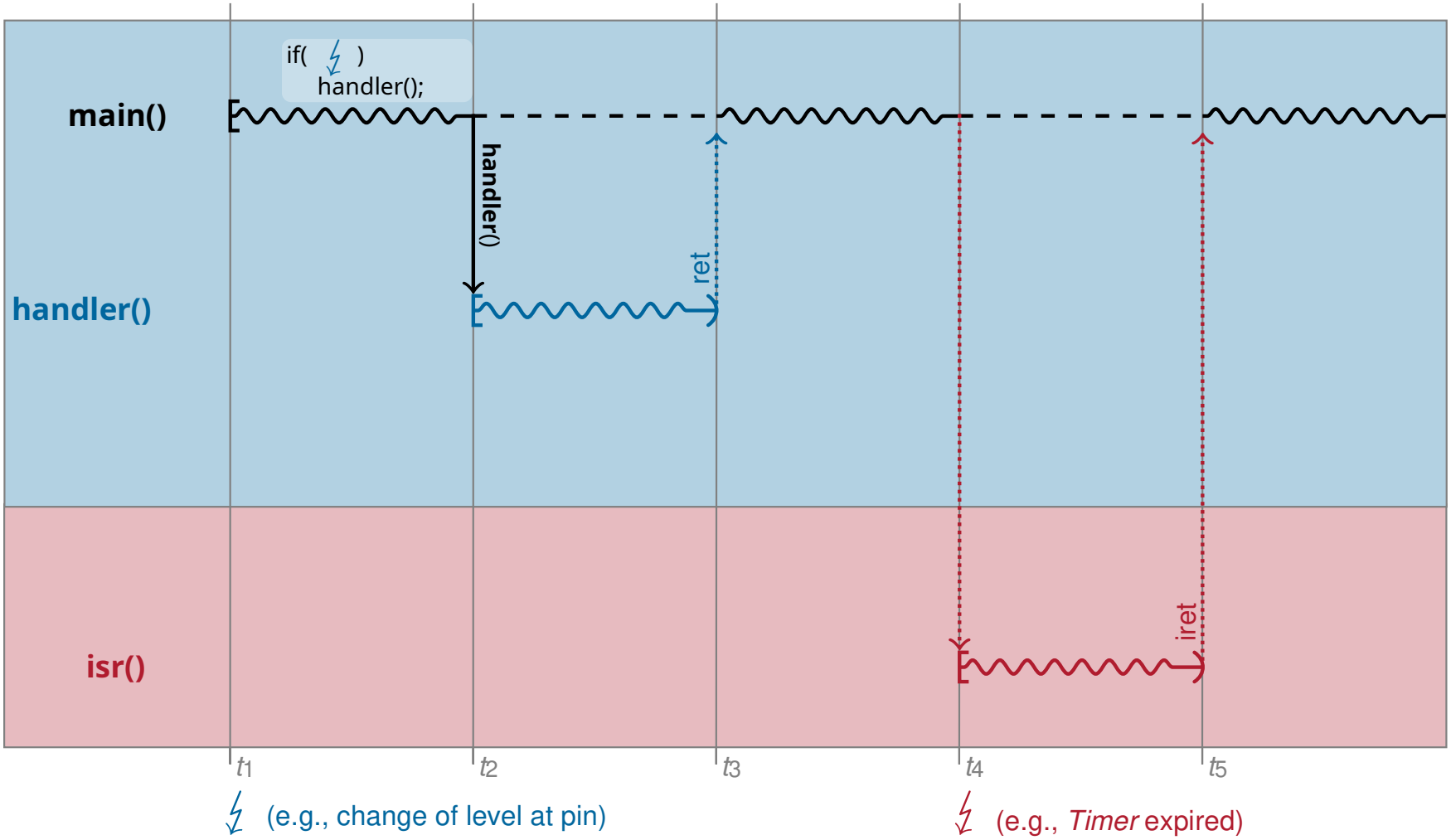
Interrupt ↦ Function Call “from Outside”



Interrupt ↦ Function Call “from Outside”



Interrupt ↦ Function Call “from Outside”



Polling vs. Interrupts – (Dis-)Advantages

- Polling ($\mapsto$ "polling-based 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 predictable
- Interrupts ($\mapsto$ "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

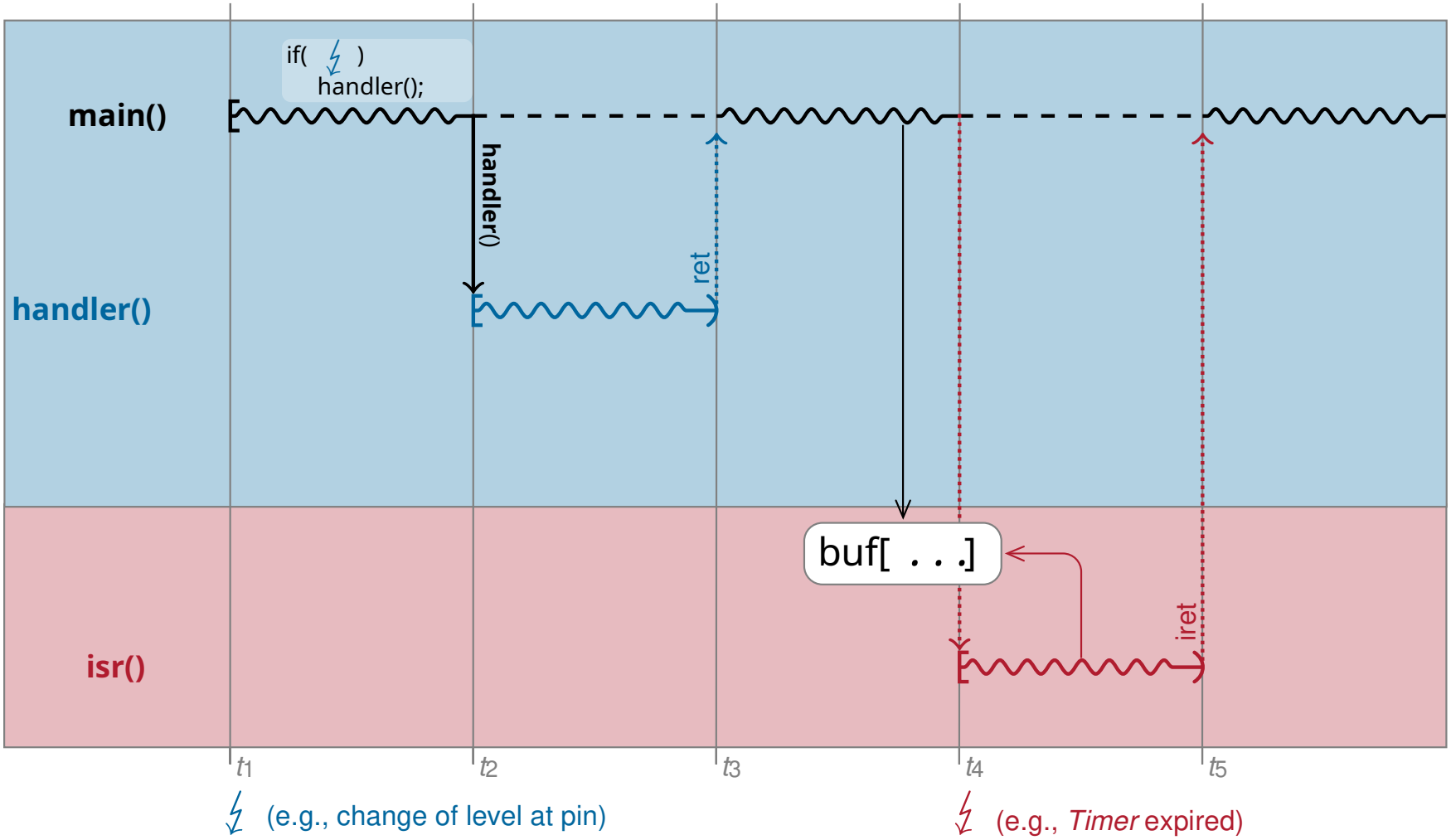
Polling vs. Interrupts – (Dis-)Advantages

- Polling ($\mapsto$ "polling-based 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 predictable

- Interrupts ($\mapsto$ "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 ↦ Unpredictable Call “from Outside”



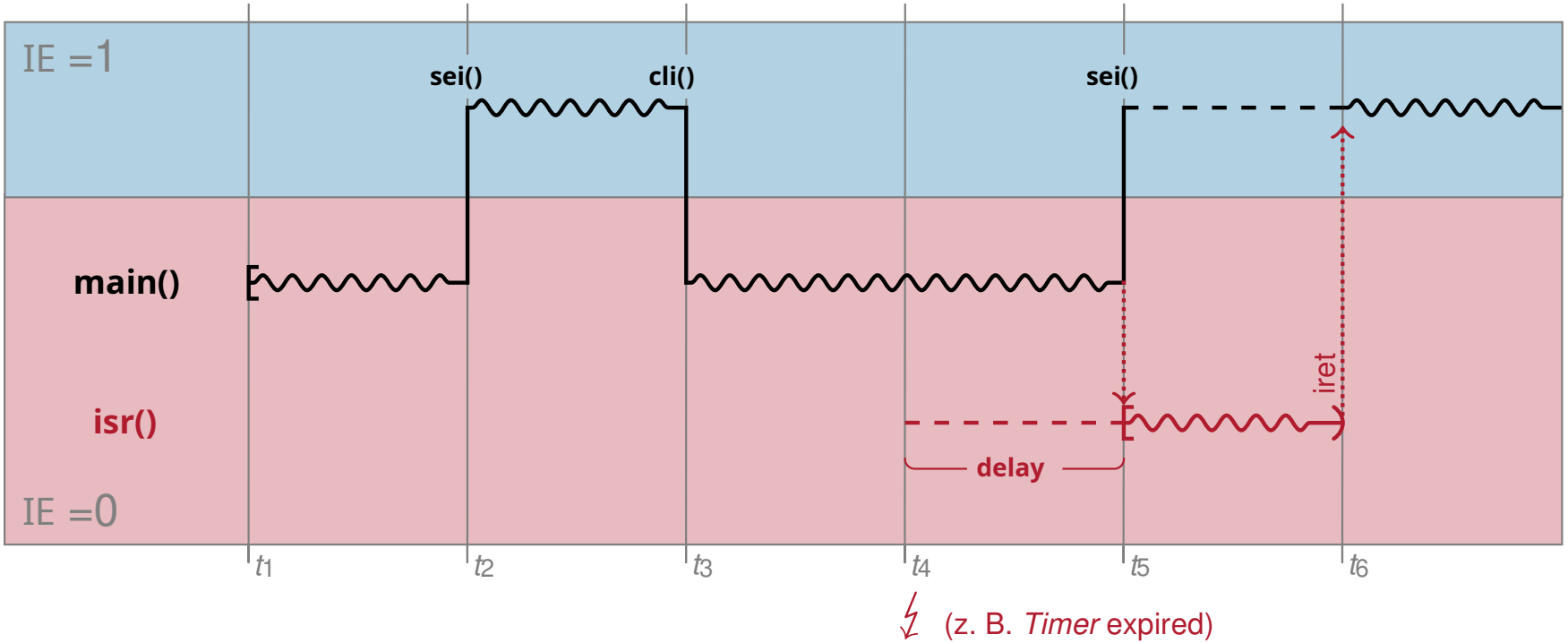
Disabling Interrupts

- 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 interrupt (per source)!
  - During longer disabled time spans, IRQs can be missed!

IRQ $\mapsto$ *Interrupt ReQuest*

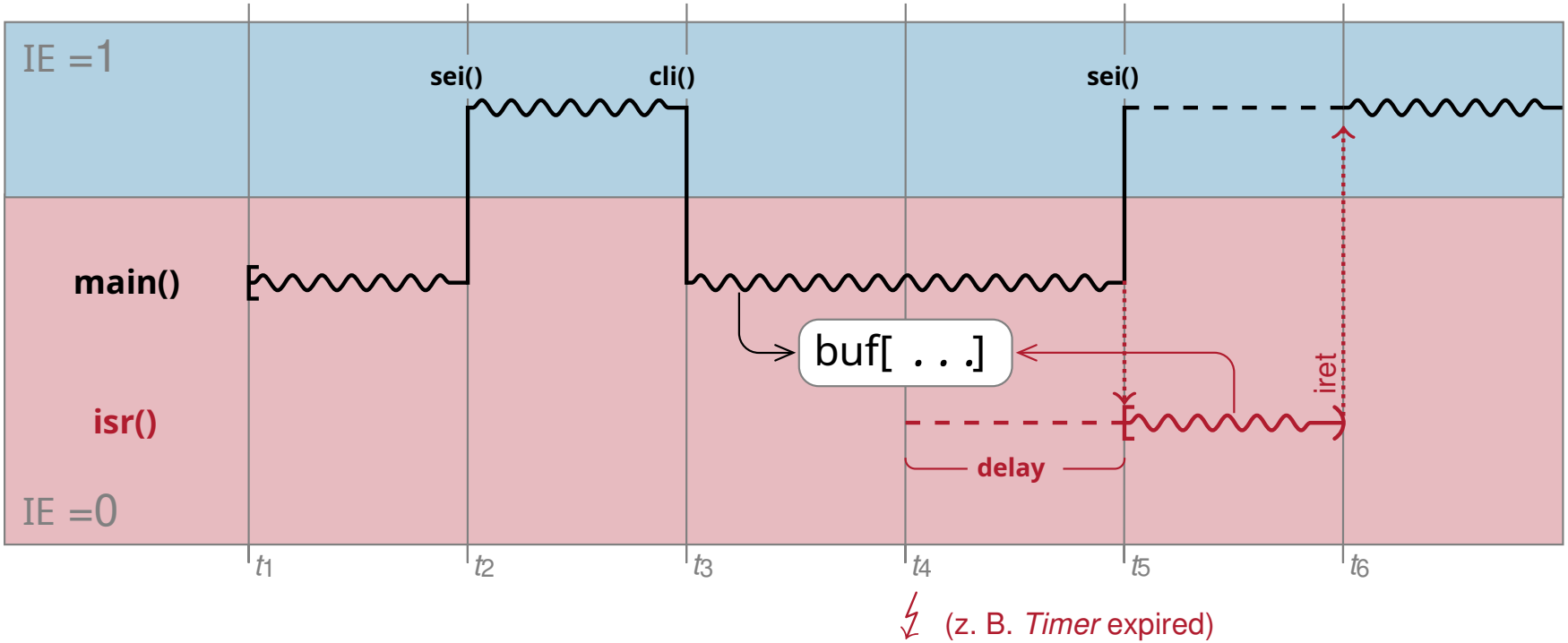
- The `IE` bit is affected by:
  - processor instructions: `cli`: `IE`$\leftarrow$0 (*clear interrupt*, IRQs disabled)
    `sei`: `IE`$\leftarrow$1 (*set interrupt*, IRQs enabled)
  - after a RESET: `IE`=0 $\leadsto$ IRQs are always disabled at the begin of the main program
  - when entering an ISR: `IE`=0 $\leadsto$ IRQs are disabled during handling of other interrupts

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 ⇝ handling is blocked, IRQ is buffered
- t_5 `main()` unblocks IRQs (`IE`=1) ⇝ buffered IRQ is executed
- t_5–t_6 During handling of the ISR, all IRQs are blocked again (`IE`=0)
- t_6 Interrupted `main()` is resumed

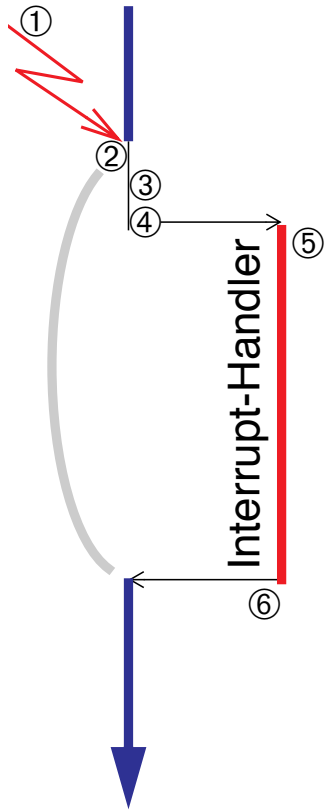
Interrupt Blocking: Example



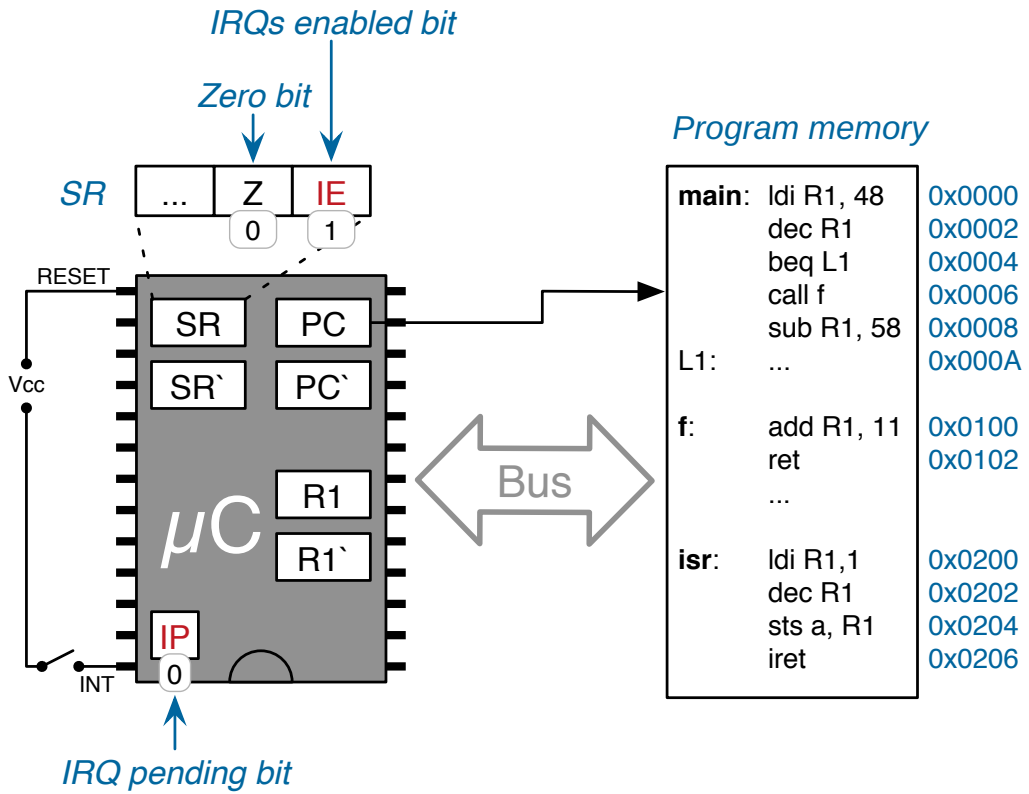
- t_1 At the begin of `main()`, all IRQs are disabled (`IE`=0)
- t_2, t_3 With `sei()` / `cli()` IRQs can be enabled (`IE`=1) / disabled
- t_4 ↯ but `IE`=0 ⇝ handling is blocked, IRQ is buffered
- t_5 `main()` unblocks IRQs (`IE`=1) ⇝ buffered IRQ is executed
- t_5–t_6 During handling of the ISR, all IRQs are blocked again (`IE`=0)
- t_6 Interrupted `main()` is resumed

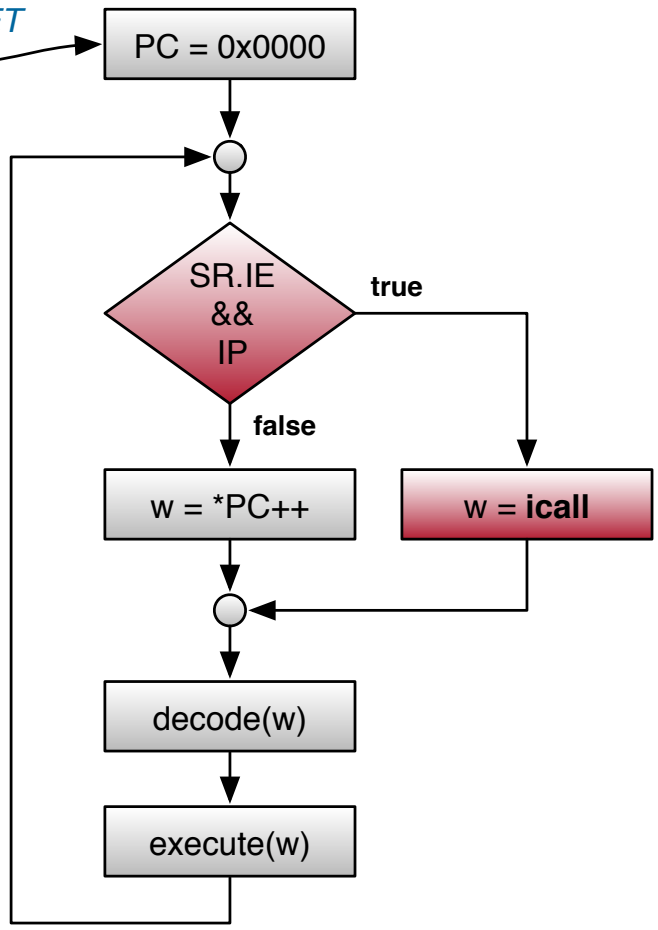
Procedure of an Interrupt – Overview

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 is executed
6. ISR terminates with "return from interrupt" instruction
   - Content of registers is restored
   - Notification of interrupts again unblocked/enabled (`IE`=1)
   - Program is 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

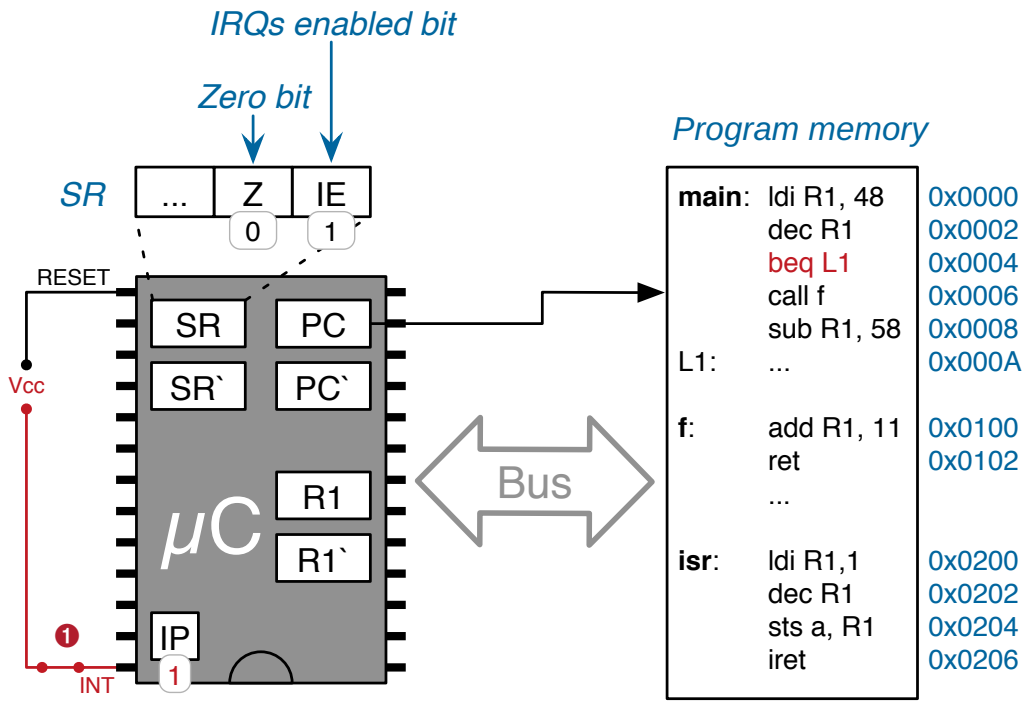
w: call <func>
PC` = PC
PC = func

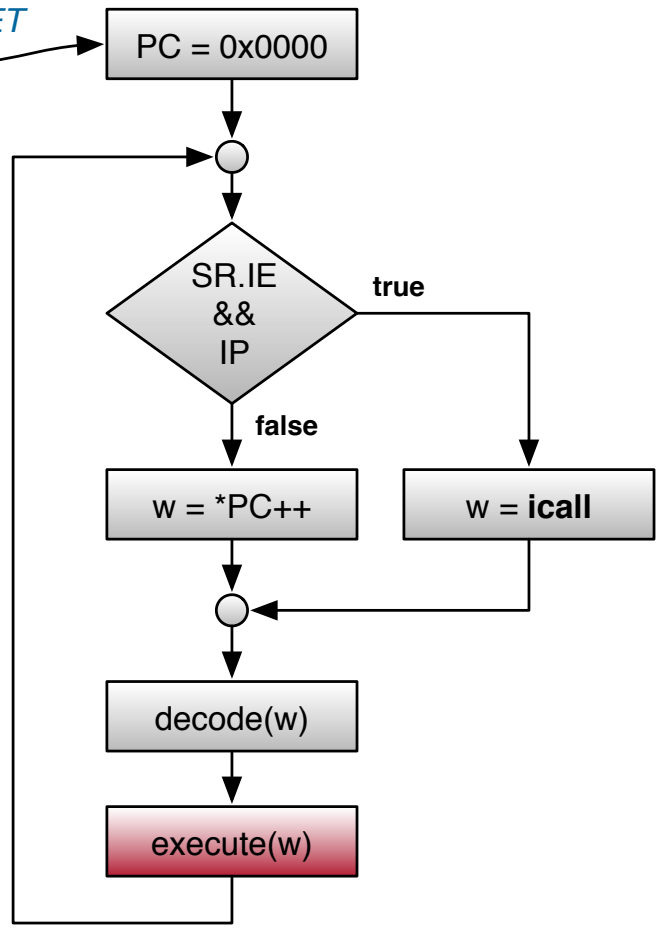
w: ret
PC = PC`

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

w: iret
SR = SR`
PC = PC`
R1 = R1`

Procedure of an Interrupt – Details

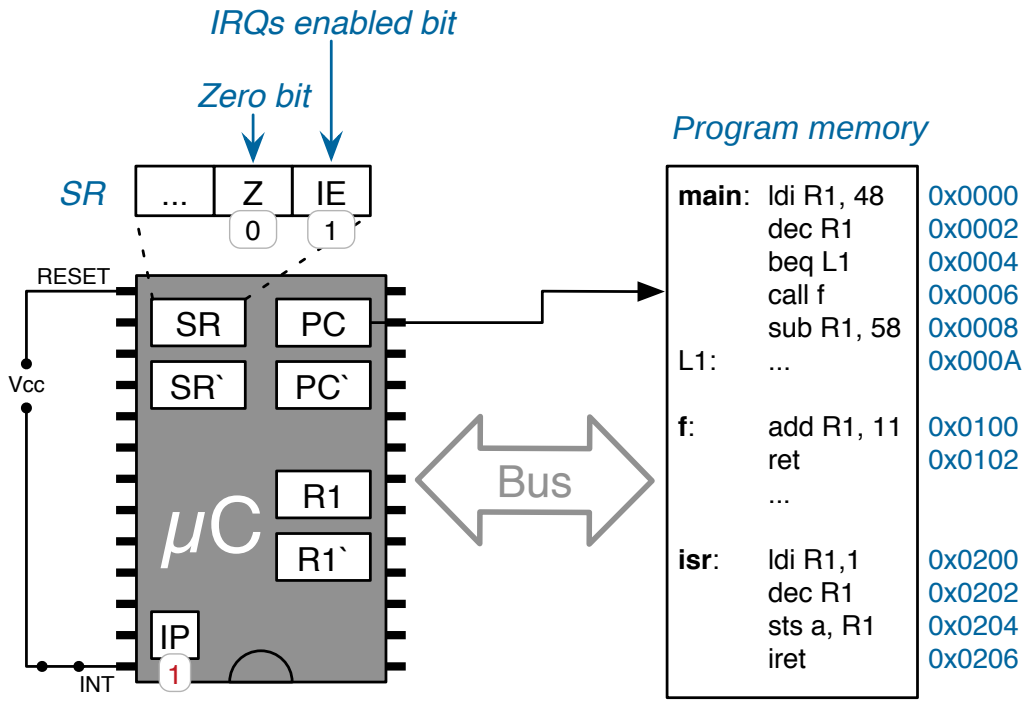


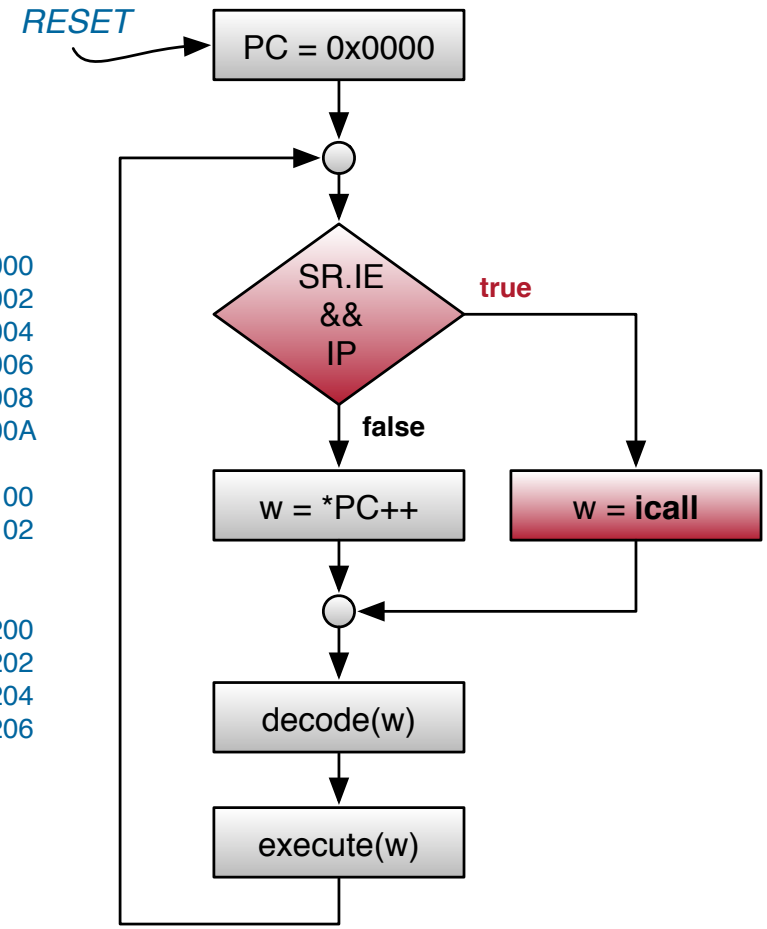


❶ Device signals an interrupt (current instruction will be finalised)

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Procedure of an Interrupt – Details

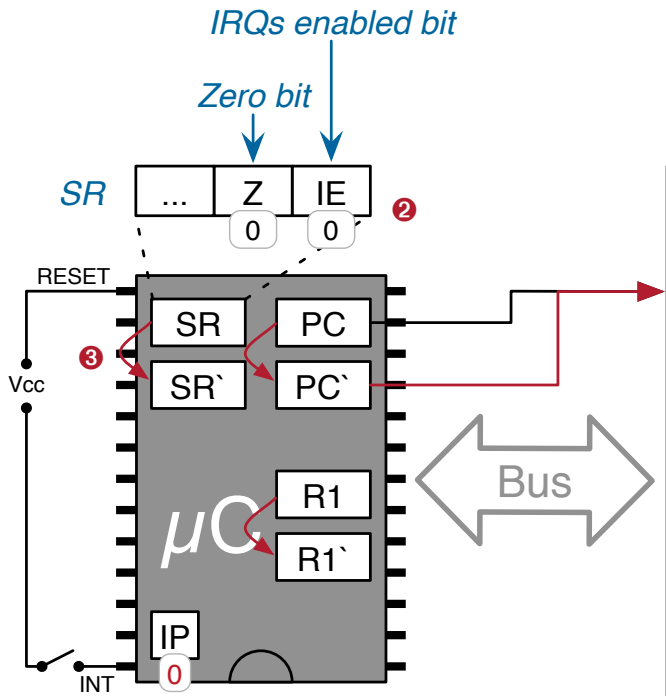


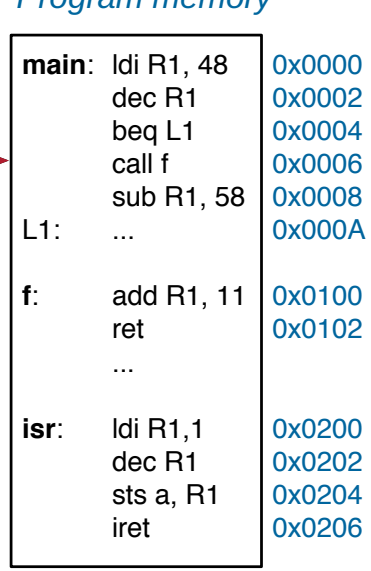


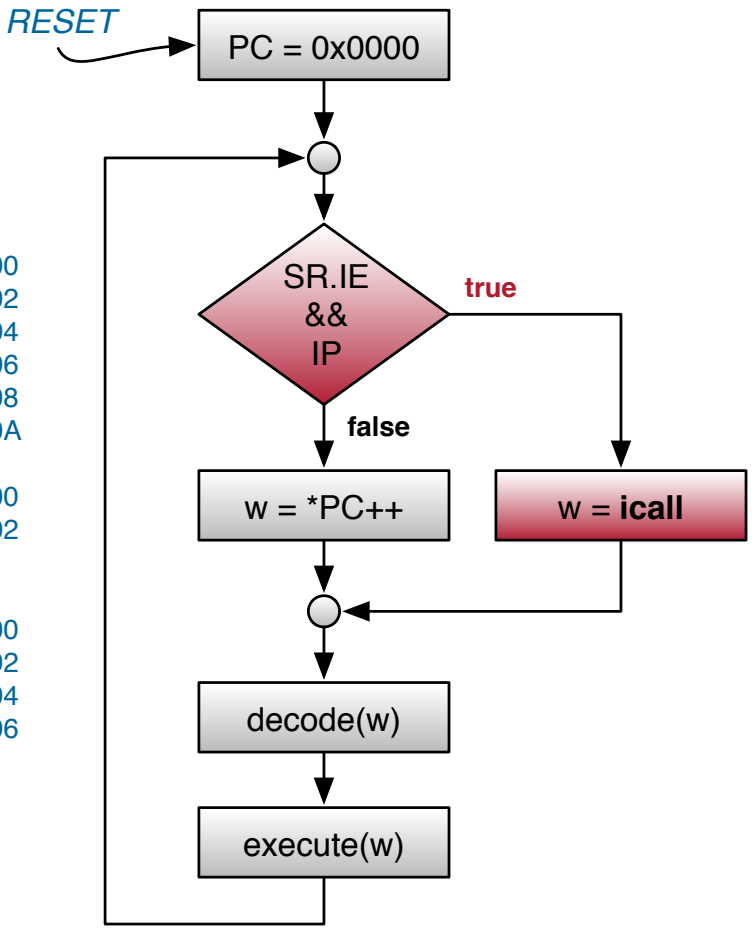
(Before the next *instruction fetch* the interrupt state is checked)

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Procedure of an Interrupt – Details







❷ The delivery of further interrupts is stalled
❸ Register contents get saved

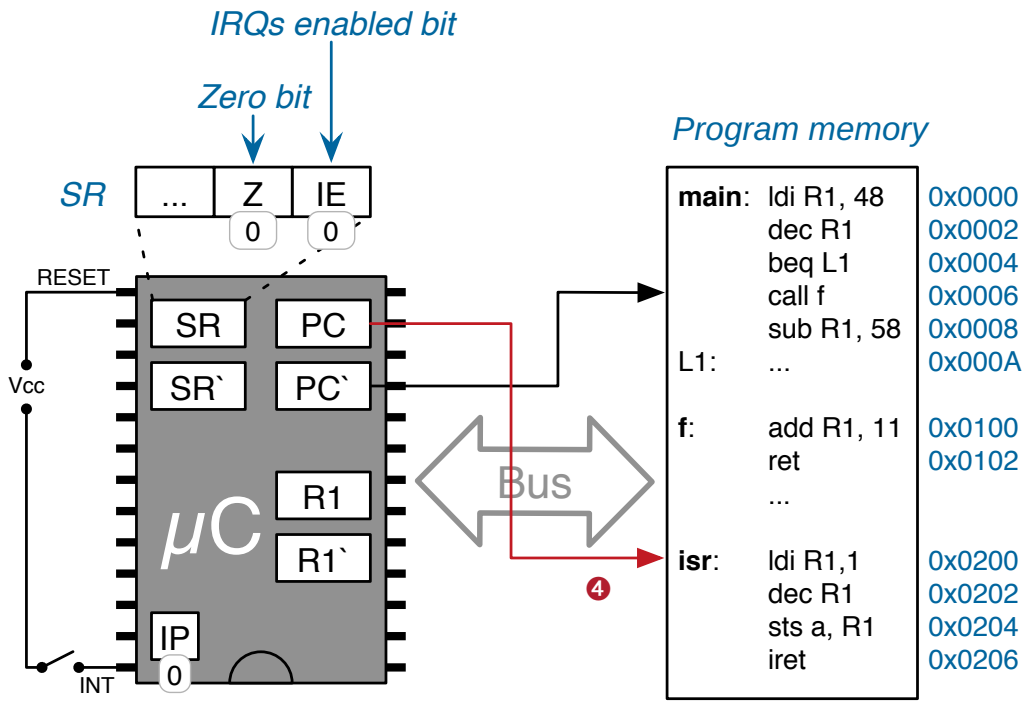
w: call <func>
PC` = PC
PC = func

w: ret
PC = PC`

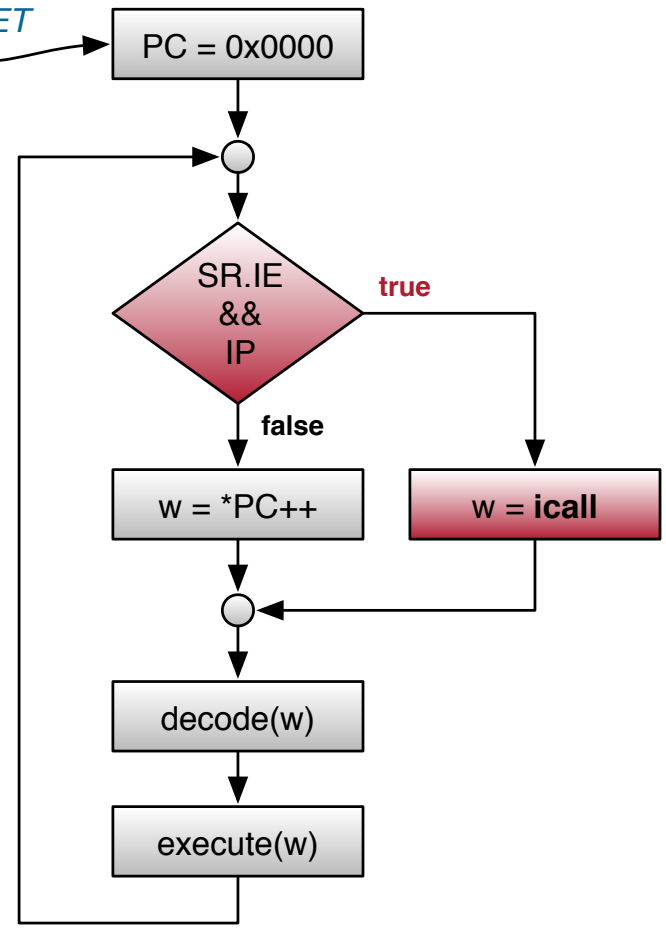
w: icall
SR` = SR
SR.IE = 0
IP = 0
PC` = PC
PC = isr
R1` = R1

w: iret
SR = SR`
PC = PC`
R1 = R1`

Procedure of an Interrupt – Details

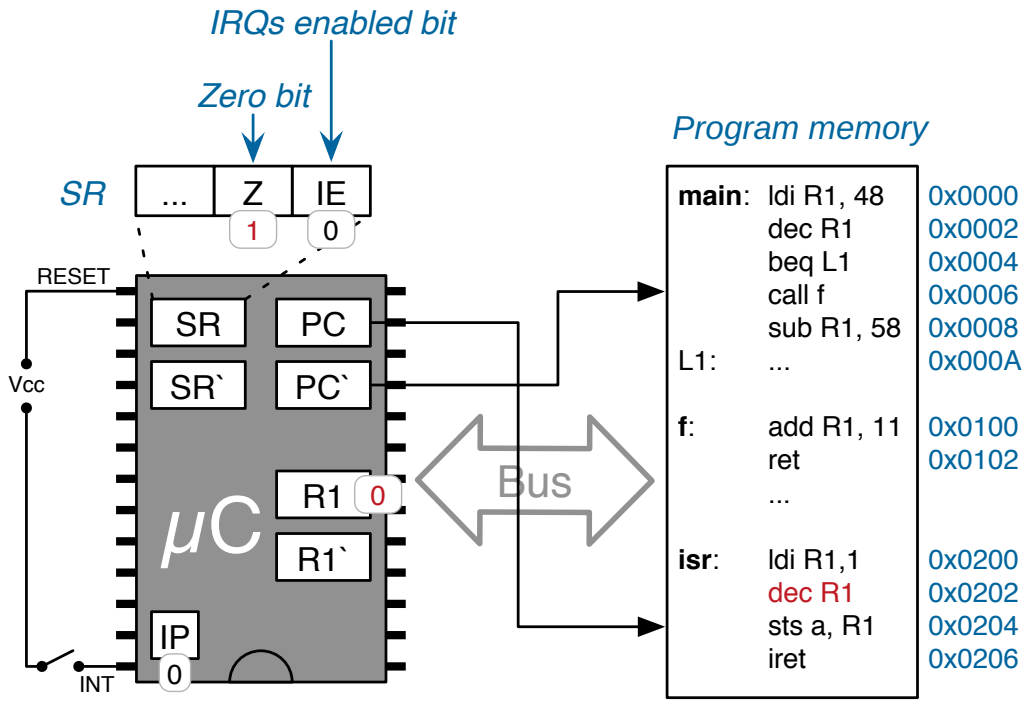


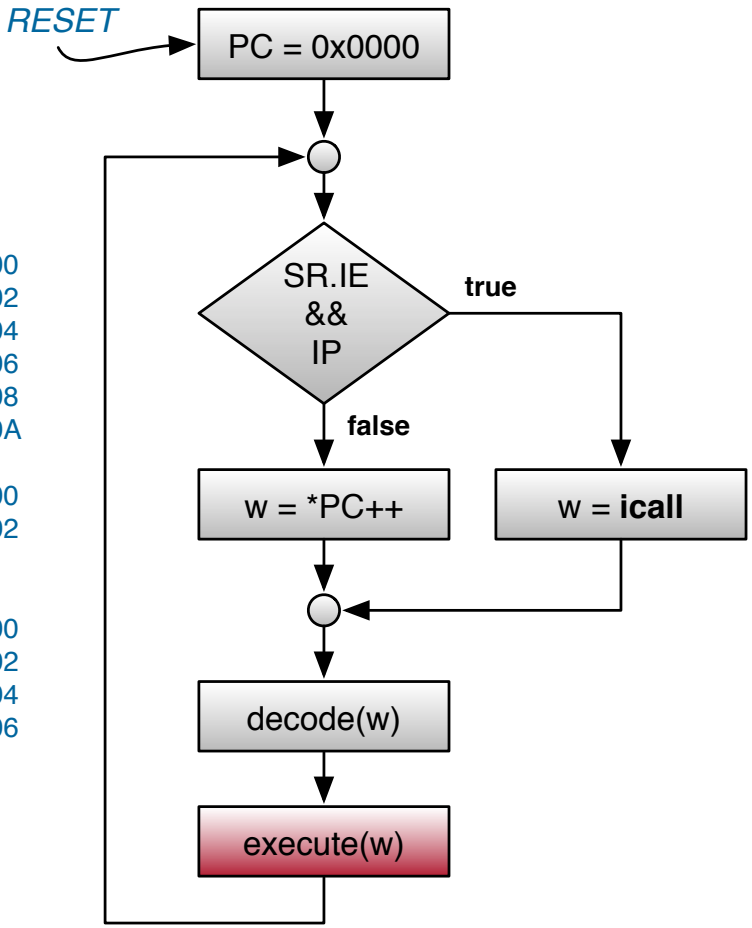
❹ ISR to be called is determined



w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Procedure of an Interrupt – Details

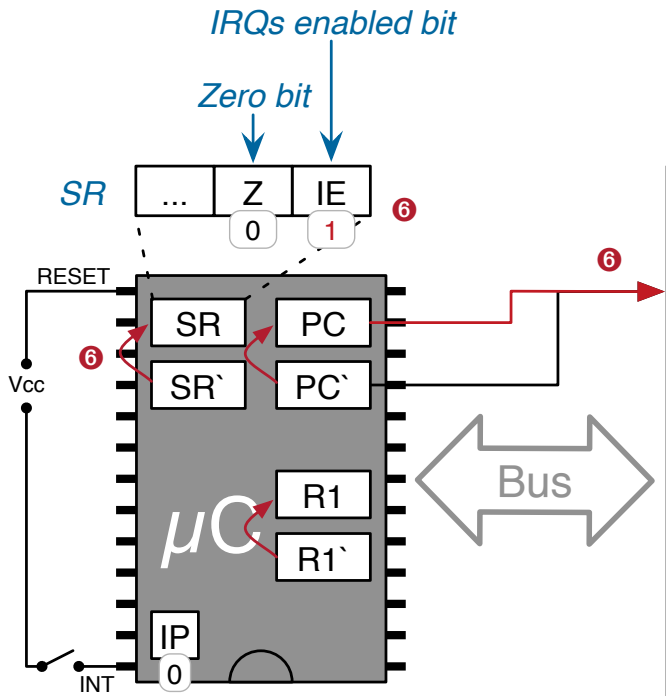


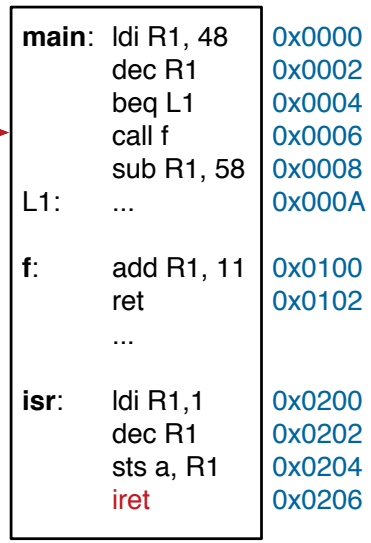


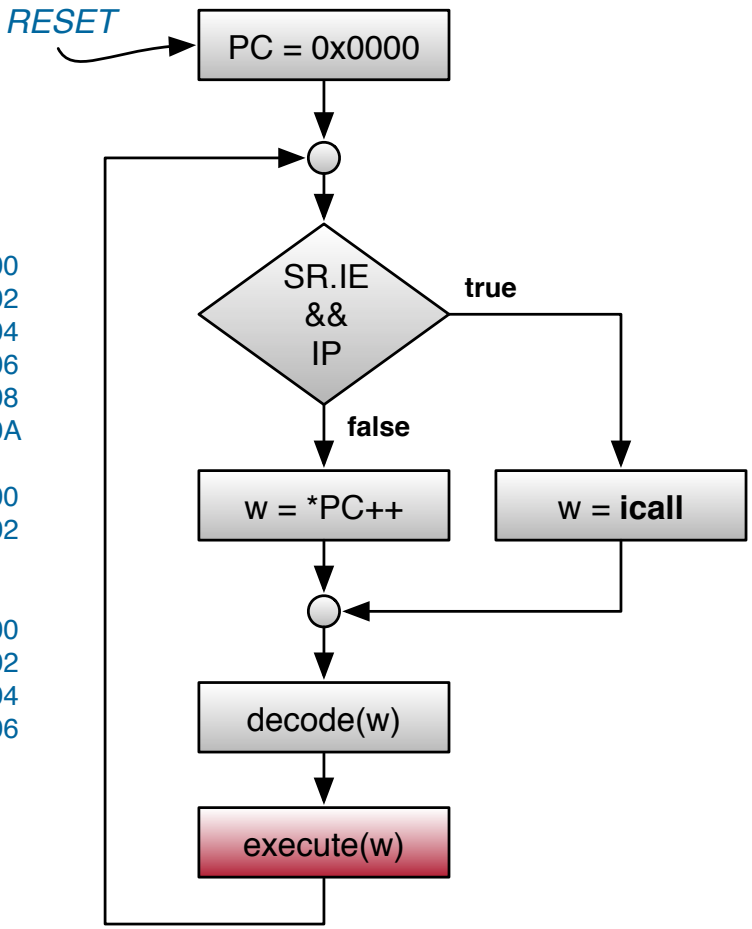
❺ ISR gets executed

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Procedure of an Interrupt – Details







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

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`