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 (IBAction) occurs when a peripheral device signals
- 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 $\rightarrow$ Function Call “from Outside”

```
main()
if()
handler();
handler()
isr()
(e.g., change of level at pin)
```

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$

$\downarrow$

$\uparrow$
Interrupt $\mapsto$ Function Call “from Outside”

```c
main()
if()
handler();

handler()

isr()

\( e.g., \) change of level at pin
\( e.g., \) Timer expired
```

\( t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \)

© kls System-Level Programming (ST 24) 18 Interrupts – Interrupts: Introduction 18–2
Interrupt $\iff$ Function Call “from Outside”

main()

if( )

handler();

handler()

isr()

(e.g., change of level at pin)

(e.g., Timer expired)

© kls System-Level Programming (ST 24)
18 Interrupts – Interrupts: Introduction 18–2
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 ↗ high processor load ↗ 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 ↗ synchronisation required
  - Program behavior unpredictable
Polling vs. Interrupts – (Dis-)Advantages

Polling

- 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

- 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”

- **main()**
- **handler()**
- **isr()**

- $t_1$ (e.g., change of level at pin)
- $t_2$ (e.g., change of level at pin)
- $t_3$ (e.g., change of level at pin)
- $t_4$ (e.g., Timer expired)
- $t_5$ (e.g., Timer expired)

```
if()
handler();
handler();
```
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 ↼ IRQs are always disabled at the begin of the main program

- When entering an ISR: IE = 0 ↼ IRQs are disabled during handling of other interrupts
Interrupt Blocking: Example

At the begin of `main()`, all IRQs are disabled (IE=0) (t₁)
With `sei() / cli()` IRQs can be enabled (IE=1) / disabled (t₂, t₃)
but IE=0 → handling is blocked, IRQ is buffered (t₄)
`main()` unblocks IRQs (IE=1) → buffered IRQ “gets through” (t₅)
During handling of the ISR, all IRQs are blocked again (IE=0) (t₅–t₆)
Interrupted `main()` is resumed (t₆)

(z. B. Timer expired)
**Interrupt Blocking: Example**

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

With `sei()` / `cli()` IRQs can be enabled (IE=1) / disabled.

But IE=0 ✗ handling is blocked, IRQ is buffered.

`main()` unblocks IRQs (IE=1) ✗ buffered IRQ “gets through”

During handling of the ISR, all IRQs are blocked again (IE=0).

Interrupted `main()` is resumed.

(z. B. Timer expired)
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
Here as an extension of the simplified pseudo processor

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

**Procedure of an Interrupt – Details**

- **PIRQs enabled bit**
- **Zero bit**
- **IRQ pending bit**

**Program memory**

- **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

© klsw  System-Level Programming (ST 24)  18 Interrupts – Controlling Interrupts
Procedure of an Interrupt – Details

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

Main:
```assembly
ldi R1, 48
dec R1
beq L1
call f
sub R1, 58
```

L1:
```assembly
...
```

F:
```assembly
add R1, 11
ret
...
```

ISR:
```assembly
ldi R1, 1
dec R1
sts a, R1
iret
```

Result:
- **PC**: 0x0000
- **SR.IE**: true
- **IP**: 0
- **w**: *PC++
- **execute(w)**
- **decode(w)**
- **ret**
- **icall**
- **iret**

SR` = SR
SR.IE = 0
IP = 0
PC` = PC
PC` = PC`
Procedure of an Interrupt – Details

**Program memory**

```
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
```

- **PC**
- **SR**
- **R1**
- **PC'**
- **SR'**
- **R1'**

**VRQs enabled bit**

**Zero bit**

**RESET**

**PC = 0x0000**

**SR.IE && IP**

false

w = *PC++

w = icall

decode(w)

execute(w)

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'

(Before the next *instruction fetch* the interrupt state is checked)
Procedure of an Interrupt – Details

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

Program memory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC 0x0000</th>
<th>PC 0x0002</th>
<th>PC 0x0004</th>
<th>PC 0x0006</th>
<th>PC 0x0008</th>
<th>PC 0x000A</th>
</tr>
</thead>
<tbody>
<tr>
<td>main: ldi R1, 48</td>
<td>0x0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec R1</td>
<td>0x0002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq L1</td>
<td>0x0004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call f</td>
<td>0x0006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub R1, 58</td>
<td>0x0008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1: ...</td>
<td>0x000A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f: add R1, 11</td>
<td></td>
<td>0x0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td>0x0102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isr: ldi R1, 1</td>
<td>0x0200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec R1</td>
<td>0x0202</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sts a, R1</td>
<td>0x0204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iret</td>
<td>0x0206</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- PC = 0x0000
- PC = isr
- R1` = R1

© klsy System-Level Programming (ST 24) 18 Interrupts – Controlling Interrupts
Procedure of an Interrupt – Details

**Interrupts Enable Bit**: Enables or disables interrupts.

**Zero Bit**: Used for branch instructions.

**Program Memory**: Stores instructions that need to be executed.

**PC**: Program Counter.

**SR**: Status Register.

**R1**: Register 1.

**Vcc**: Voltage Supply.

**µC**: Microcontroller.

**Bus**: Data transfer medium.

**ISR to be called is determined**: Indicates the function to be called.

**ICall**: Instruction to call another function.

**IREt**: Instruction to return from a function call.

**RESET**: System reset signal.

**PC = 0x0000**: Initial program counter value.

**Main Code**:
- ldi R1, 48
- dec R1
- beq L1
- call f
- sub R1, 58
- L1: ...
- f: add R1, 11
- ret

**ISR Code**: ldi R1, 1
- dec R1
- sts a, R1
- iret

**Flowchart**:
1. **PC = 0x0000**
2. **SR.IE && IP**
3. **SR` = SR**
4. **SR.IE = 0**
5. **IP = 0**
6. **PC` = PC**
7. **R1` = R1**
8. **ICall**: SR = SR`
9. **PC = PC**
10. **PC = PC`**
11. **0x0000**
12. **0x0002**
13. **0x0004**
14. **0x0006**
15. **0x0008**
16. **0x000A**
17. **0x0100**
18. **0x0102**
19. **0x0200**
20. **0x0202**
21. **0x0204**
22. **0x0206**

**Decode**: Decodes the instruction.

**Execute**: Executes the decoded instruction.

**ISR to be called is determined**: Determines which ISR to call.
Procedure of an Interrupt – Details

**Program memory**

```
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
```

**Flowchart**

- **RESET**
  - PC = 0x0000
  - SR.IE && IP
    - true
      - w = *PC++
    - false
      - w = icall
  - decode(w)
  - execute(w)

**ISR gets executed**

© klsw  System-Level Programming (ST 24)  18 Interrupts – Controlling Interrupts
ISR terminates with `iret`-instruction
- Register contents are restored
- Delivery of interrupts is reactivated
- Program is resumed