Exercises in System Level Programming (SLP) – Summer Term 2025

Exercise 5

Maxim Ritter von Onciul
Eva Dengler

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Interrupts

Interrupts

- Procedure of an interrupt (see 18-7):
  0. Hardware sets required flag
  1. If interrupts are enabled and the interrupt is not masked, the interrupt controller interrupts the current execution
  2. Further interrupts are disabled
  3. Current program position is saved
  4. Address of the handler is read from the interrupt vector table and is then jumped to
  5. The interrupt handler is executed
  6. At the end of the interrupt handler, the instruction “return from interrupt” returns to the interrupted program and then re-enables the interrupts

Implementation of Interrupt Handlers

- For every interrupt, one bit for storing its state is available
- May lead to lost interrupts: An interrupt occurs during...
  - the execution of an interrupt handler (interrupts too fast)
  - disabled interrupts section (for synchronization of critical sections)
- This problem cannot be prevented in general

$\rightsquigarrow$ Risk minimization: Interrupt handler shall be as short as possible
  - Avoid any kind of loops and function calls
  - Do not use any blocking function (ADC/serial interface!)

Interrupts on the AVR

- Timer
- Serial interface
- ADC (analog digital converter)
- External interrupts by level changes at certain I/O pins
  - Choice of level- or edge-triggered
  - Depend on the interrupt source

  ⇒ ATmega328PB: 2 sources at the pins PD2 (`INT0`) and PD3 (`INT1`)

  ⇒ `BUTTON0` at PD2

  ⇒ `BUTTON1` at PD3
- More details in the ATmega328PB data sheet

(Re-)Enabling Interrupts

- Interrupts can be enabled and disabled by special machine instructions
- The library avr-libc provides useful macros:
  `#include <avr/interrupt.h>`
  - `sei()` (set interrupt flag): enables interrupts (delayed by one instruction)
  - `cli()` (clear interrupt flag): disables all interrupts (immediately)
- Upon entering an interrupt handler, all interrupts are disabled automatically and re-enabled again as soon as the handler is exited
- `sei()` should never be called from inside an interrupt handler
  - Potentially infinitely nested interrupt handlers
  - Possibility of a stack overflow
- At the start of the µC, interrupts are disabled by default

Configuring Interrupts

- Interrupt sense control (ISC) bits of the ATmega328PB are located at the external interrupt control register A (EICRA)
- Position of the ISC-bits inside the register defined by macros

Interrupt INT0		Interrupt on	Interrupt INT1	
ISC01	ISC00		ISC11	ISC10
0	0	low level	0	0
0	1	either edge	0	1
1	0	falling edge	1	0
1	1	rising edge	1	1

- Example: Configuring INT1 of the ATmega328PB for a falling edge

```
/* the ISC-bits are located in the EICRA */
EICRA &= ~(1 << ISC10); // deleting ISC10
EICRA |= (1 << ISC11);  // setting  ISC11
```

(Un-)Masking Interrupts

- Single interrupts can be enabled (= unmasked) individually
  - ATmega328PB: External interrupt mask register (`EIMSK`)
- The bit positions inside of the register are defined by macros `INTn`
- A set bit enables the corresponding interrupt
- Example: Enabling the external interrupt `INT1`

```
EIMSK |= (1 << INT1); // Unmask the external interrupt INT1
```

Interrupt Handler

- Registering an interrupt handler is implemented by the C library
- Macro `ISR` (interrupt service routine) used for defining a handler function (`#include <avr/interrupt.h>`)
- Parameter: Desired vector
  - Available vectors: Refer to avr-libc documentation for `avr/interrupt.h`
  - Example: `INT1_vect` for external interrupt `INT1`
- Example: Implement handler for `INT1`

```c
#include <avr/interrupt.h>

static volatile uint16_t counter = 0;

ISR(INT1_vect) {
  counter++;
}
```

Synchronization

Keyword volatile

- When an interrupt occurs, `event = 1` is set
- Active waiting loop waits until `event != 0`
- Compiler detects that `event` is not changed within the loop
  - ⇒ the value of `event` is only loaded once from memory into a processor register
  - ⇒ endless loop
- `volatile` enforces that the variable is loaded from memory before every access

```
static uint8_t event = 0;
ISR(INT0_vect) {
    event = 1;
}

void main(void) {
    while(1) {
        while(event == 0) { /* wait for event */ }
        // handle event [...]
    }
}
```

Usage of volatile

- Missing `volatile` can lead to unexpected program execution
- Unnecessary use of `volatile` prevents certain compiler optimizations
- Correct use of `volatile` is task of the programmer!

⇝ Use `volatile` as rarely as possible but as often as required

Lost Update

- Counting button presses that have to be processed
  - Incremented in the interrupt handler
  - Decremented in the main program to start the processing

```c
static volatile uint8_t counter = 0;
ISR(INT0_vect) {
  counter++;
}

void main(void) {
  while(1) {
    if(counter > 0) {

      counter--;

      // handle pressed button
      // [...]
    }
  }
}
```

Lost Update

Main program

```
01 ; C instruction: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interrupt handler

```
05 ; C instruction: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Line	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—
6	5	4	5
7	5	4	6
8	6	4	6
4	4	4	—

16-Bit Access (Read Write)

- Concurrent use of 16 bit values (read write)
  - Incrementing in the interrupt handler
  - Reading in the main program

```c
static volatile uint16_t counter = 0;
ISR(INT0_vect) {
  counter++;
}

void main(void) {
  if(counter > 300) {
    sb_led_on(YELLOW0);
  } else {
    sb_led_off(YELLOW0);
  }

  // [...]
}
```

16-Bit Access (Read Write)

Main program

```
01 ; C instruction: if(counter>300)
02 lds  r22, counter
03 lds  r23, counter+1
04 cpi  r22, 0x2D
05 sbci r23, 0x01
```

Interrupt handler

```
07 ; C instruction: counter++;
08 lds r24, counter
09 lds r25, counter+1
10 adiw r24,1
11 sts counter+1, r25
12 sts counter, r24
```

Line	counter	r22 & r23	r24 & r25
—	`0x00ff`	—	—
2	`0x00ff`	`0x??ff`	—
8+9	`0x00ff`	`0x??ff`	`0x00ff`
10	`0x00ff`	`0x??ff`	`0x0100`
11+12	`0x0100`	`0x??ff`	`0x0100`
3	`0x0100`	`0x01ff`	—

⇒ In lines 4+5, the comparison uses `0x01ff` (= 511) instead of `0x0100` (= 256). The comparison yields `true` and the LED is switched on.

Disabling the Handling of Interrupts on the AVR

- Many more concurrency problems are possible
  - Non-atomic modification of shared data
  - Analysis of the problem by the application programmer
  - Choice of suitable synchronization primitives
- Solution here: Mutual exclusion by disabling interrupts
  - Disable all interrupts: `cli()` and `sei()`
  - Disabling single interrupts (`EIMSK`-register)
- Problem: Interrupts can be lost during a critical section

⇒ Critical sections have to be as short as possible

Lost Update

- How can a lost update be prevented?

```
static volatile uint8_t counter = 0;
ISR(INT0_vect) {
  counter++;
}

void main(void) {
  while(1) {
    if(counter > 0) {
      cli();
      counter--;
      sei();
      // handle pressed button
      // [...]
    }
  }
}
```

16-Bit Access (Read Write)

- How can a read-write anomaly be prevented?

```
static volatile uint16_t counter = 0;
ISR(INT0_vect) {
  counter++;
}

void main(void) {


  cli();
  if(counter > 300) {
    sei();
    sb_led_on(YELLOW0);
  } else {
    sei();
    sb_led_off(YELLOW0);
  }

  // [...]
}
```

Power-Saving Modes

Power-Saving Modes of AVR Processors

- AVR-based devices are often powered by batteries (e.g. remotes)
- Saving energy can drastically extend the life span
- AVR processors support multiple power-saving modes
    - Deactivating functional units
    - Different “depths” of sleep
    - Only active functional units can wake up the CPU
- Default mode: Idle
    - CPU clock is stopped
    - No more memory accesses
    - Hardware (timer, external interrupts, ADC, etc.) are still active
- Documentation in ATmega328PB data sheet

Usage of the Sleep Modes

- Support from the avr-libc: (`#include <avr/sleep.h>`)
  - `sleep_enable()` - enables the sleep mode
  - `sleep_cpu()` - enters the sleep mode
  - `sleep_disable()` - disables the sleep mode
  - `set_sleep_mode(uint8_t mode)` - configures the used mode
- Documentation of `avr/sleep.h` in avr-libc documentation

```
#include <avr/sleep.h>

set_sleep_mode(SLEEP_MODE_IDLE); // use idle mode
sleep_enable();   // activate sleep mode
sleep_cpu();      // enter sleep mode
sleep_disable();  // recommended: deactivate sleep mode
    ↪ afterwards
```

Lost Wakeup

- Sleeping beauty (german: *Dornröschenschlaf*)
  - ⇒ **Problem:** There is exactly one interrupt
  - ⇒ **Solution:** Disable interrupts during the critical area

Main program

```
01 sleep_enable();
02 event = 0;
03
04 cli();
05 while(!event) {
06     sei(); ↯ Interrupt ↯
07     sleep_cpu();
08     cli();
09 }
10 sei();
11
12 sleep_disable();
```

Interrupt handler

```
01 ISR(TIMER1_COMPA_vect) {
02     event = 1;
03 }
```

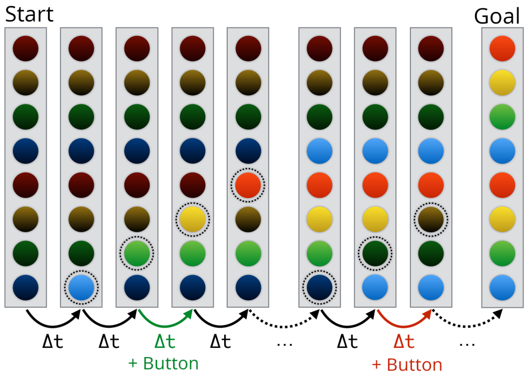
⇒ What if the interrupt occurs between lines 6 and 7?

⇒ **Solution:** `sei()` is executed atomically with next line

Assignment: Dexterity Game

Assignment: Dexterity Game (1)

- Game cursor moves over the LED strip and inverts (toggles) the state of the LED
- LED state is retained if the button is pressed
- Goal: Switch on all LEDs



Assignment: Dexterity Game (2)

- After each level, a winning sequence is displayed via the LEDs

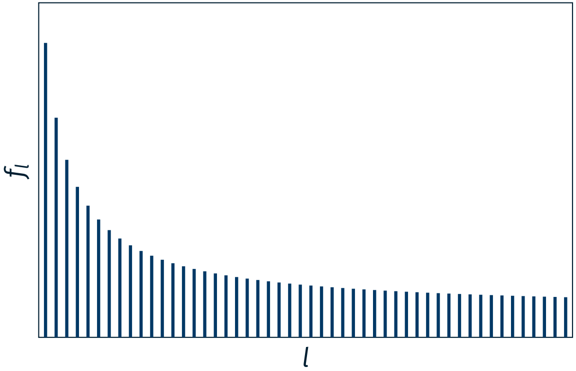
```
void main(void) {
  // Initialisation
  // [...]

  while(1) {
    // starting level
    // [...]

    // show win sequence
    // [...]

    // update level
    // [...]
  }
}
```

Detect a Button Press

- Goals:
  - Edge detection in hardware
  - Handle events using interrupts
  - **No** use of the `libspicboard`
- Details:
  - `BUTTON0` is wired to PD2
  - Configure PD2 as input (with activated pull-up resistor)
  - PD2 is input of `INT0`
  - Which level/edge has to be configured for the interrupt?
  - How does a minimal interrupt handler for this assignment look like?

Difficulty

- Speed of the game determines its difficulty
  - ⇒ Passive waiting with the `timer` module of the `libspicboard`
- Difficulty increases with each level l
- Speed converges to a maximum
  - ⇒ Series of waiting times: $f_l = \frac{a}{l} + b$ (a and b are constants)



Hands-On: Simple interrupt counter

Screencast: `https://www.video.uni-erlangen.de/clip/id/17231`

Hands-on: Simple Interrupt Counter

- Counting activations of `BUTTON0` (`PD2`)
- Detect activation with the help of interrupts
- Output the current counter value using the 7-segment display
- Enter a CPU sleeping state whenever the **value is even**
- “Standby” LED switched on during the sleep mode (`BLUE0`)
- Hints:
  - Detection of the activation **without** the `libspicboard`
  - `PD2/BUTTON0` is the input of `INT0`
  - Interrupt on a falling edge:
    - `EICRA(ISC00) = 0`
    - `EICRA(ISC01) = 1`
  - 7-segment display needs regular interrupts to display values