Exercises in System Level Programming (SLP) – Summer Term 2024

Exercise 8

Maximilian Ott

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

Presentation Assignment 4
Hands-on: Coffee Machine

Hands-on: Coffee Machine (1)

- Learning goals:
  - Finite state machines
  - Timers and alarms
  - Interrupts & sleep modes
Wiring:

- Pump & heating: Port D, Pin 5 (active-low)
- Button: INT0 an Port D, Pin 2 (active-low)
- Sensor: INT1 an Port D, Pin 3 (water: high; no water: low)
- State LED:
  - BLUE0: STANDBY
  - GREEN0: ACTIVE
  - RED0: NO_WATER

**STANDBY**
- Machine is switched off
- Pump and heating are off
- User can start making coffee by pressing the button
- Initial state

**ACTIVE**
- Machine is switched on
- Pump and heating are on
- Water tank is not empty
- User can stop the machine by pressing the button

**NO_WATER**
- Coffee machine shows that not enough water is in the tank
- Pump and heating are off
- Time period: 2 seconds
Hints:

- Pressed button & change of water level by interrupts
- State LED: `void setLEDState(state_t state)`
- Waiting phases can be implemented using the single-shot alarms
- During waiting phases always enter a power-saving mode

### DDRx
Configuration of pin i of port x as in-/output

- Bit \( i = 1 \) → Pin i as output
- Bit \( i = 0 \) → Pin i as input

### PORTx
Mode of operation depends on DDRx:

- If pin i is configured as output, then bit i in the PORTx register controls whether a high level or a low level has to be generated at pin i
  - Bit \( i = 1 \) → high level at pin i
  - Bit \( i = 0 \) → low level at pin i
- If pin i is configured as input, then the internal pull-up resistor can be activated
  - Bit \( i = 1 \) → pull-up resistor at pin i (level is pulled high)
  - Bit \( i = 0 \) → pin i configured as tri-state

### PINx
Bit i returns the current level of pin i at port x (read only)
Hands-on: Coffee Machine (4)

- 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

<table>
<thead>
<tr>
<th>Interrupt INTO ISCØ1</th>
<th>ISCØ0</th>
<th>Interrupt on low level</th>
<th>Interrupt INTO ISC11</th>
<th>ISC10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>low level</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>either edge</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>falling edge</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rising edge</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- ATmega328PB: External interrupt mask register (EIMSK)
- The position of the bits in this register is also defined by macros INTn

Hands-on: Ticker
In Depth: Strings

- char: Single character (e.g. 'a')
- String: Array of chars (e.g. "Hello")
- C: Last char of a string: '\0'
  ⇒ Memory requirement: strlen(s) + 1

```
01 char s[] = "World\n";
02 char c = s[0];
03 c = s[4];
04 char *s2 = s + 2;
05 c = s2[1];
```

... Stack ↓

```
0x0906
0x0907
0x0908
0x090a
0x0909
0x090c
0x090d
0x090f
0x0910
0x0911
... 0x0911
```

In Depth: Strings

- char: Single character (e.g. 'a')
- String: Array of chars (e.g. "Hello")
- C: Last char of a string: '\0'
  ⇒ Memory requirement: strlen(s) + 1

```
01 char s[] = "World\n";
02 char c = s[0];
03 c = s[4];
04 char *s2 = s + 2;
05 c = s2[1];
```

... Stack ↓

```
0x0906
0x0907
0x0908
0x090a
0x0909
0x090c
0x090d
0x090f
0x0910
0x0911
... 0x0911
```

In Depth: Strings

- char: Single character (e.g. 'a')
- String: Array of chars (e.g. "Hello")
- C: Last char of a string: '\0'
  ⇒ Memory requirement: strlen(s) + 1

```
01 char s[] = "World\n";
02 char c = s[0];
03 c = s[4];
04 char *s2 = s + 2;
05 c = s2[1];
```

... Stack ↓

```
0x0906
0x0907
0x0908
0x090a
0x0909
0x090c
0x090d
0x090f
0x0910
0x0911
... 0x0911
```
In Depth: Strings

- **char**: Single character (e.g. 'a')
- **String**: Array of chars (e.g. "Hello")
- **C**: Last char of a string: '\0'
  
  ⇒ Memory requirement: strlen(s) + 1

```
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```

Stack ↓

```
... 0x0911
s[6] '\0' 0x0910
s[5] '\n' 0x090f
s[4] 'd' 0x090e
s[3] 'l' 0x090d
s[2] 'r' 0x090c
s[1] 'o' 0x090b
s[0] 'W' 0x090a
C 'W' 0x0909
```

```
... 0x0908
0x0907
0x0906
... 0x0911
```

In Depth: Strings

- **char**: Single character (e.g. 'a')
- **String**: Array of chars (e.g. "Hello")
- **C**: Last char of a string: '\0'

  ⇒ Memory requirement: strlen(s) + 1

```
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```

Stack ↓

```
... 0x0911
s[6] '\0' 0x0910
s[5] '\n' 0x090f
s[4] 'd' 0x090e
s[3] 'l' 0x090d
s[2] 'r' 0x090c
s[1] 'o' 0x090b
s[0] 'W' 0x090a
C 'W' 0x0909
```

```
... 0x0908
0x0907
0x0906
... 0x0911
```
In Depth: Strings

- **char**: Single character (e.g. 'a')
- **String**: Array of chars (e.g. "Hello")
- **C**: Last char of a string: '\0'

⇒ Memory requirement: strlen(s) + 1
Hands-on: Ticker

- Functionality:
  Displaying a text step-by-step on the 7-segment display

- Learning goals:
  - Strings in C
  - Pointers & pointer arithmetic
  - Alarms & sleep modes

- Procedure:
  - Recurring alarms with TIMER0
  - Combining the current substring
  - Output via the 7-segment display
  - During waiting phases, the microcontroller has to enter a sleep mode (passive waiting)

Hands-on: Ticker – Determine Substrings

```c
const char *string = "HELLO SPIC";
const char *current = string;

// current[0] == 'H' && current[1] == 'E'
++current;

// current[0] == 'E' && current[1] == 'L'

// [...

// current[0] == '\0', current[1] == ??

current = string;
```
const char *string = "HELLO SPIC";
const char *current = string;
// current[0] == 'H' & current[1] == 'E'
++current;
// current[0] == 'E' & current[1] == 'L'
// [...]
// current[0] == '\0', current[1] == ??
current = string;
Hands-on: Ticker – Determine Substrings

```c
const char *string = "HELLO SPIC";
const char *current = string;
// current[0] == 'H' && current[1] == 'E'
++current;
// current[0] == 'E' && current[1] == 'L'
// [...]
// current[0] == '\0', current[1] == ?
current = string;
```

![Diagram of string and current pointers]

Hands-on: Ticker – Determine Substrings
Hands-on: Ticker – Determine Substrings

```c
const char *string = "HELLO SPIC";
const char *current = string;
// current[0] == 'H' && current[1] == 'E'
++current;
// current[0] == 'E' && current[1] == 'L'
// [...]
// current[0] == '\0', current[1] == ??
current = string;
```

```
string → H E L L O S P I C \0

...?

C \0
current
```

Hands-on: Ticker – Determine Substrings
const char *string = "HELLO SPIC";
const char *current = string;
// current[0] == 'H' && current[1] == 'E'
++current;
// current[0] == 'E' && current[1] == 'L'
// [...]
// current[0] == '\0', current[1] == ??
current = string;
Hands-on: Ticker – Determine Substrings

01 const char *string = "HELLO SPIC";
02 const char *current = string;
03 // current[0] == 'H' && current[1] == 'E'
04 ++current;
05 // current[0] == 'E' && current[1] == 'L'
06 // [...]
07 // current[0] == '\0', current[1] == ?? ☇
08 current = string;