FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

182.694 Microcontroller VU

FAKULTÄT FÜR !NFORMATIK

Manfred Schwarz

WS 2018

Featuring Today:
Assembler Programming

Weekly Training Objective

- Already done
  - 1.2 Board test †
  - 2.1.1 Assembler demo program †
  - 2.1.2 Makefile †
  - 2.2.1 Logical operations ∗
- This week
  - 2.2.2 Input with floating pins ∗
  - 2.2.4 Monoflop buttons
  - 2.2.5 Digital I/O
  - 2.4.1 precompiled LCD ∗
- Until Exam
  - 2.2.3 LED Rain ∗
  - 2.2.9 LED curtain ∗
  - 2.4.2 Calling conventions I
  - 2.4.3 Calling conventions II

Assembler Programming

- Assembler is always very “device specific”
  $\rightarrow$ AVR-Assembler
- Start with basic AVR Assembler
- Followed by “advanced” examples

Why Assembler?

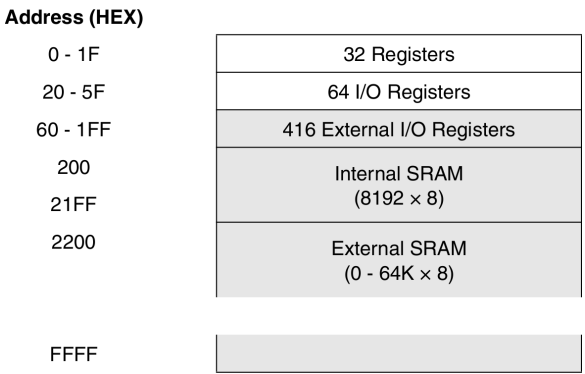
- see how all programs you write “really” end up
- to understand the CPU architecture better
- to understand where speed improvements may be possible
- to realize there is no big secret behind it

“Features” of Assembler

- Assembler is basically a 1–1 mapping to machine code
- Assembly language is human readable
- No high-level language constructs, e.g., if or while
- No nested expressions.
  e.g., you cannot write add (mult 3,2), 1

RAM

- General Purpose Register and IO-Register mapped into SRAM address range
- ATmega1280 . . . 8 kB SRAM
- Multiple addressing modes
  - Register Direct
  - Data Direct
  - Data Indirect (Pointer)
  - Powerful displacement, pre-decrement, and post-increment modes!



Assembler Basics

GPR instructions

```
ldi r0,0x0F
ldi r2,0x0F
add r2,r0
...
```

Assembler Basics

I/O Register example

- DDRx: defines if a physical pin of the controller is used as in or out
- PORTx: stores the logic values that currently being outputted on the physical pins
- PINx: to read the values on the pins of Portx, you read the values that are in this register

Note: x ranges from $A - L$ for our uC.

Bit operations

- Used for Digital I/O (set or clear port pins)
- Example: PA0 drives high-active LED. Turn that LED on.

```
sbi DDRA,  DDA0
sbi PORTA, PA0
```

Assembler Basics

Bit operations

- Used for Digital I/O (set or clear port pins)
- Example: PA0 drives high-active LED. Turn that LED on.

```
sbi DDRA,  DDA0
sbi PORTA, PA0
```

```
;better?
sbi PORTA, PA0
sbi DDRA,  DDA0
```

Assembler Basics

Bit operations

- Used for Digital I/O (set or clear port pins)
- Example: PA0 drives high-active LED. Turn that LED on.

```
sbi DDRA,  DDA0
sbi PORTA, PA0
```

```
;better?
sbi PORTA, PA0
sbi DDRA,  DDA0
```

- switching a MC-Pin from * to Output:
  Mostly better to first change PORT register and then the DDR.
  Avoids glitches!

Assembler Basics

Example

- PA7 connected with button against ground
- Objective: Read button value

```
sbi PORTA, PA7
cbi DDRA,  DDA7
in  r16,   PORTA
```

Assembler Basics

Example

- PA7 connected with button against ground
- Objective: Read button value

```
sbi PORTA, PA7
cbi DDRA,  DDA7
in  r16,   PORTA
```

- Common mistake!
  - reading PINx gives real input value
  - reading PORTx gives pull-up/output status

Assembler Basics

Example

- PA7 connected with button against ground
- Objective: Read button value

```
sbi PORTA, PA7
cbi DDRA,  DDA7
in  r16,   PINA
```

- Common mistake!
  - reading PINx gives real input value
  - reading PORTx gives pull-up/output status

Assembler Basics

Example

- PA7 connected with button against ground
- Objective: Read button value

```
sbi PORTA, PA7
cbi DDRA,  DDA7
in  r16,   PINA
```

```
;much better!
cbi DDRA,  DDA7
sbi PORTA, PA7
in  r16,   PINA
```

- Common mistake!
  - reading PINx gives real input value
  - reading PORTx gives pull-up/output status
- switching a MC-Pin from * to Input:
  Mostly better first to change DDR register. Avoids glitches!

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off.

```
cbi PORTA, PA0
sbi PORTA, PA1
sbi PORTA, PA2
cbi PORTA, PA3
sbi DDRA,  DDA0
sbi DDRA,  DDA1
sbi DDRA,  DDA2
sbi DDRA,  DDA3
```

- Works, but we can do better!

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off.

```
ldi temp,  0x06 ;0000 0110
out PORTA, temp
ldi temp,  0x0F ;0000 1111
out DDRA,  temp
```

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off.

```
ldi temp,  0x06 ;0000 0110
out PORTA, temp
ldi temp,  0x0F ;0000 1111
out DDRA,  temp
```

- Works, but we are overwriting unused bits!

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off.

```
ldi temp,  0x06 ;0000 0110
out PORTA, temp
ldi temp,  0x0F ;0000 1111
out DDRA,  temp
```

- Works, but we are overwriting unused bits! $\rightarrow$ unused?
- Consider the previous example: PA7 is configured as an input with pull-up

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off.

```
ldi temp,  0x06 ;0000 0110
out PORTA, temp
ldi temp,  0x0F ;0000 1111
out DDRA,  temp
```

- Works, but we are overwriting unused bits! $\rightarrow$ unused?
- Consider the previous example: PA7 is configured as an input with pull-up
- Now it is an input without pull-up!
- Not really readable (which bits are set if PORTA = 0xCA?)

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off without changing other PINs.

```
in  temp,  PORTA
ori temp,  (1<<PA1)|(1<<PA2)
out PORTA, temp

in  temp, DDRA
ori temp, (1<<DDA0)|(1<<DDA1)|(1<<DDA2)|(1<<DDA3)
out DDRA, temp
```

Instead of `(1<<PA1)` one can use `_BV(PA1)`.

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off without changing other PINs.

```
in  temp,  PORTA
ori temp,  (1<<PA1)|(1<<PA2)
out PORTA, temp

in  temp, DDRA
ori temp, (1<<DDA0)|(1<<DDA1)|(1<<DDA2)|(1<<DDA3)
out DDRA, temp
```

Nearly correct!

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off without changing other PINs.

```
in   temp,  PORTA
ori  temp,  (1<<PA1)|(1<<PA2)
andi temp,  (0<<PA0)&(0<<PA3)
out  PORTA, temp

in  temp, DDRA
ori temp, (1<<DDA0)|(1<<DDA1)|(1<<DDA2)|(1<<DDA3)
out DDRA, temp
```

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off without changing other PINs.

```
in   temp,  PORTA
ori  temp,  (1<<PA1)|(1<<PA2)
andi temp,  (0<<PA0)&(0<<PA3)
out  PORTA, temp

in  temp, DDRA
ori temp, (1<<DDA0)|(1<<DDA1)|(1<<DDA2)|(1<<DDA3)
out DDRA, temp
```

Not correct!

Assembler Basics

Another example

PA3:0 are connected to LED3:0.
Turn on LED1 and LED2 and turn the other ones off without changing other PINs.

```
in   temp,  PORTA
ori  temp,  (1<<PA1)|(1<<PA2)
andi temp,  ~((1<<PA0)|(1<<PA3))
out  PORTA, temp

in  temp, DDRA
ori temp, (1<<DDA0)|(1<<DDA1)|(1<<DDA2)|(1<<DDA3)
out DDRA, temp
```

Correct!

Assembler Basics

Procedure is called RMW

- Read
- Modify
- Write

Assembler Basics

Procedure is called RMW

- Read
- Modify
- Write

- Should always be used!
  - Interrupts, Timer, ADC, ...
  - Assembler, C, ...

Assembler Basics

Procedure is called RMW

- Read
- Modify
- Write

- Should always be used!
  - Interrupts, Timer, ADC, ...
  - Assembler, C, ...
- Not explicitly checked in the first exam, but ...

Assembler Basics

Procedure is called RMW

- Read
- Modify
- Write

- Should always be used!
  - Interrupts, Timer, ADC, ...
  - Assembler, C, ...
- Not explicitly checked in the first exam, but ...
- ... in the second and the make-up exam we will check that no bits are unnecessarily changed!

Pull-Ups

Why do we even use them?

Why don't we just connect the push-button to VCC instead of ground, and sense a pressed button as high instead of low?

Pull-Ups

Why do we even use them?

Why don't we just connect the push-button to VCC instead of ground, and sense a pressed button as high instead of low?

Electrical Characterisitics

The ATmega1280 has a absolute maximum rating of 40 mA per I/O pin. Thus, 0.2 W is the maximum allowed load on a pin!
There is also an overall maximum (200 mA) for all pins!

Pull-Ups

Electrical Characterisitics

The ATmega1280 has a absolute maximum rating of 40 mA per I/O pin. Thus, 0.2 W is the maximum allowed load on a pin!
There is also an overall maximum (200 mA) for all pins!

Drive large loads

If you have to drive loads above the limit, use the port to enable a transistor to drive the load.

Pull-Ups

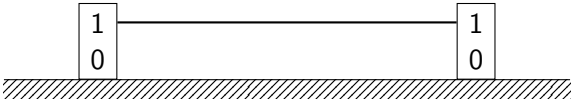
Electrical Characterisitics

The ATmega1280 has a absolute maximum rating of 40 mA per I/O pin. Thus, 0.2 W is the maximum allowed load on a pin!
There is also an overall maximum (200 mA) for all pins!

Internal Pull-Ups are weak

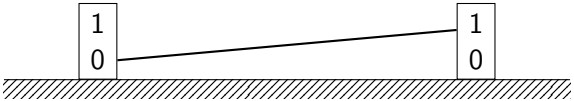
This is by design, to prevent the current from exceeding the maximum rating.

Analogy: Single Line and Ground



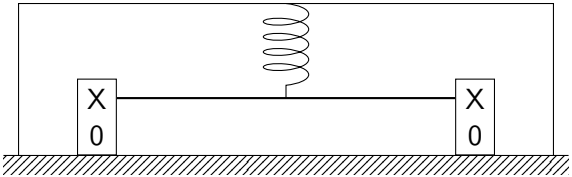
Fixing the levers to a common plate.

Analogy: Single Line and Ground



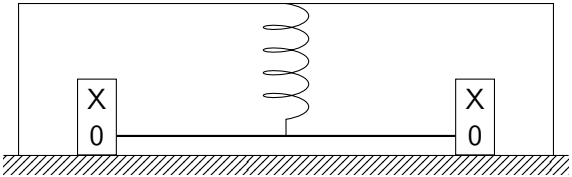
Fixing the levers to a common plate.
$\Rightarrow$ large current flowing and no detection of change!

Analogy: Single Line, Ground and Pull-Up



A weak spring keeps the bar in the high state (=weak/recessive state).

Analogy: Single Line, Ground and Pull-Up



A weak spring keeps the bar in the high state (=weak/recessive state).

Questions

Connect two output pins

- Both are configured as output
- one is set to high
- the other to low

Questions

Connect two output pins

- Both are configured as output
- one is set to high
- the other to low
- Short circuit!

Questions

Connect two output pins

- Both are configured as output
- one is set to high
- the other to low
- Short circuit!

Connect two inputs pins

- Both are configured as input
- one has the internal pull-up enabled
- the other one has not.

Questions

Connect two output pins

- Both are configured as output
- one is set to high
- the other to low
- Short circuit!

Connect two inputs pins

- Both are configured as input
- one has the internal pull-up enabled
- the other one has not.
- Both read high.

Questions

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one has not.

Questions

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one has not.
- Both read high.

Questions

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one has not.
- Both read high.

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one an external pull-down enabled.

Questions

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one has not.
- Both read high.

Connect two inputs pins

- Both are configured as input
- one has an external pull-up enabled
- the other one an external pull-down enabled.
- Both read their value? (Short circuit!)

Attention!

Parallel resistors reduce the cumulative resistance!

Thus connecting multiple pull-up/down resistors to one pin, e.g., incorrect usage of a matrix keypad, may lead to a violation of the maximum current! Check the lecture notes Sec. 5.2 on how this should be done.

Warning

There will be point deductions in the applications if you require a setup which causes shorts / conflicting drivers!
Draw a schematic, for yourself, to check if there are problems!

SBI vs. SBR

SBI: Set Bit in I/O Register (already heard)

- e.g., `sbi PORTA, PA7 ;sets bit 7 in PORTA register`
- only works in the first 32 I/O Registers (most timer registers are above)

SBI vs. SBR

SBI: Set Bit in I/O Register (already heard)

- e.g., `sbi PORTA, PA7 ;sets bit 7 in PORTA register`
- only works in the first 32 I/O Registers (most timer registers are above)

SBR: Set Bits in Register

- works on (upper 16) General Purpose Registers (r16-r31)
- e.g., `sbr r16, 7 ;set bit 7 in register 16`

SBI vs. SBR

SBI: Set Bit in I/O Register (already heard)

- e.g., `sbi PORTA, PA7 ;sets bit 7 in PORTA register`
- only works in the first 32 I/O Registers (most timer registers are above)

SBR: Set Bits in Register

- works on (upper 16) General Purpose Registers (r16-r31)
- e.g., `sbr r16, 7 ;set bit 7 in register 16`

SBI vs. SBR

SBI: Set Bit in I/O Register (already heard)

- e.g., `sbi PORTA, PA7 ;sets bit 7 in PORTA register`
- only works in the first 32 I/O Registers (most timer registers are above)

SBR: Set Bits in Register

- works on (upper 16) General Purpose Registers (r16-r31)
- e.g., `sbr r16, 7 ;set bits 2:0 in register 16`

- second argument is a bitmask: 0x07 $\rightarrow$ 0b0000 0111
- takes over all "ones" in the bitmask to the target register
- other option to achieve this?

SBI vs. SBR

What about ori?

- `ori r16, 7`
  does it do the same as `sbr r16, 7`?

SBI vs. SBR

What about ori?

- `ori r16, 7`
  does it do the same as `sbr r16, 7`?
- solution: compare the opcodes (AVR Instruction Set):
  `sbr: 0110 KKKK dddd KKKK`
  `ori: 0110 KKKK dddd KKKK`

Other Assembler Stuff

CBR: Clear Bits in Registers

works like sbr but clears all bits where the bitmasks is 1
`cbr r16, 0x05` → `andi r16, (0xFF − 0x05)`

LSL: Logical Shift Left

shifts all bits in register one place to the left. Bit 0 is cleared. Implementation in the AVR core: `add rd, rd` (add without carry)

Other Assembler Stuff

Many of these 'tricks' can be found in the Instruction Set

- `ser` (set all bits in register) is implemented as `ldi` with "hardcoded" value 0xFF.
- `clr Rd` (clear register) is implemented as `eor Rd, Rd`
- `ld Rd, Z` (indirect load from data space) is implemented as `ldd Rd, Z+q` (indirect load with displacement) with q$= 0$

"More than 8-bit" – Operations

- `A = r17:16, B = r19:18`
- 16-bit addition (`A` $\leftarrow$ `A+B`)

```
add r16, r18 ;r16 + r18
adc r17, r19 ;r17 + r19 + C
```

- 16-bit subtraction (`A` $\leftarrow$ `A-B`)

```
sub r16, r18 ;r16 - r18
sbc r17, r19 ;r17 - r19 - C
```

- 8-bit multiplication $\rightarrow$ 16-bit result

```
mul r16, r17 ; r1:r0 ← r16 × r17
```

- Accessing 16-bit registers (be aware!)

Examples – if

Given

```
if(r17==2)
  r18 = 0;
else
  r18 = 1;
```

With `r17` and `r18` being CPU registers.

Examples – if

Solution

```
    cpi r17, 2    ; compare r17 with 2
    brne else     ; if (!zero_flag) => else
    ldi r18, 0    ; r18 = 0
    rjmp end      ; => end
else:
    ldi r18, 1    ; r18 = 1
end:
```

Examples – while

Given

```
while (r17 < 20)
  r17++;
```

With `r17` being a CPU register.

Examples – while

Solution

```
while:
    cpi r17, 20    ; r17 - 20
    brge end       ; if (!negative_flag) => end;
    addi r17, 1    ; r17 = r17 + 1
    jmp while      ; => while
end:
```

Control Flow

Branch

Conditional branches (jump-if-statements) have very limited range!

- BREQ label1 ; jumps to label1 if Z=1
- BRLO label2 ; jumps to label2 if C=1
- ...

Unconditional Jumps

- JMP (reaches complete memory range)
- RJMP (one cycle faster but smaller range)

Stack

Stack

- "Part" of the SRAM
- Stores:
  - Temporary data (to backup registers used in ISRs)
  - Local variables (mainly C programming)
  - Return addresses of
    - Subroutine calls
    - Interrupt Service Routines
- Grows Top-Down (starts at highest SRAM address)

Stack

Stack Pointer

- "Pointer" to the first empty stack location (AVR)
- Has to be initialized to the end of RAM
  - The ATmega1280 does this automatically
  - But it is good practice to do it; imagine you decide to implement a soft-reset feature (RAMEND is defined in .inc)
- Be very careful when changing the Stack by hand!
- There must NEVER be important data below the Stack Pointer (Interrupts)

Assembler Functions with Parameters

How to pass Parameters?

- 3 different possibilities to hand parameters to functions
- Depending on the number of parameter, some may not work

Assembler Functions with Parameters

1. via Register

fast, easy, only 32 registers available

```
ldi  r16,    'a'
call toupper     ;r16 <- toupper(r16)
out  PORTA,  r16
```

Assembler Functions with Parameters

2. via SRAM (heap)

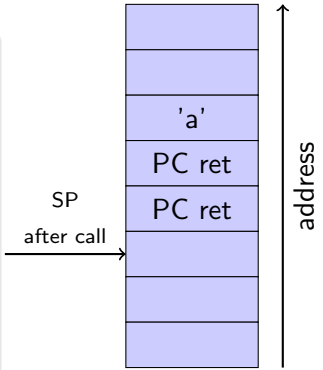
```
ldi  r16,    'a'
ldi  XL,     0x2?
ldi  XH,     0x1?
st   X,      r16
call toupper      ;toupper(*X)
ldi  XL,     0x2?
ldi  XH,     0x1?
ld   r16,    X
out  PORTA,  r16
```

Assembler Functions with Parameters

3. via Stack

- allows for variable number of parameters
  E.g., printf, is such a variadic function
- push parameters on stack before calling the function

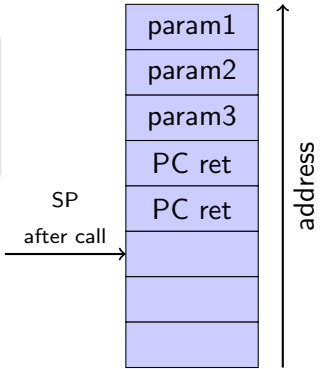
```
ldi  r16,    'a'
push r16
call toupper     ;r16 <- toupper('a')
out  PORTA,  r16
pop  r16         ; clean stack
```



Assembler Functions with Parameters

3. via Stack

what if we want something like:
`uint8_t myXOR(uint8_t x, uint8_t y)`
or `uint16_t mySquare(uint8_t x)`



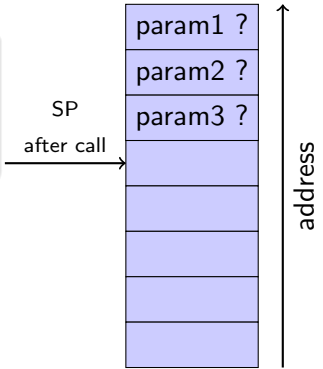
Assembler Functions with Parameters

3. via Stack

what if we want something like:

`uint8_t myXOR(uint8_t x, uint8_t y)`

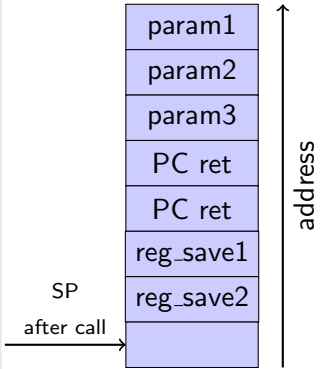
or `uint16_t mySquare(uint8_t x)`



Assembler Functions with Parameters

3. via Stack

- Stack needs to be cleaned!
  Be very very careful!
- caller-save vs. callee-save registers
  - caller-save: have to be saved/restored by caller (callee can write on them without restore)
  - callee-save: have to be saved/restored by the callee
  - callee-save is the more challenging task
- It is good to have both
  $\rightarrow$ calling conventions
- See Exercise 2.4.3
- What about interrupts?



AVR Interrupt Handling

Interrupts

- Events on Microcontroller
- Different sources (Timer, ADC, Reset, ...)
- "Interrupts" the program execution
- cannot be "predicted"

What happens when an Interrupt occurs?

- Finishing the current instruction (if multi cycle)
- Program Counter pushed on the stack / Interrupts are disabled
- Instruction at corresponding Interrupt Vector is executed (normally a jump to Interrupt Service Routine)

Interrupts

- There are no “parameters” to Interrupts
- Save all registers changed in the ISR on the stack — push
- and restore them — pop — in reversed order
- Do not forget to save the SREG!

```
ldi  r16, 0x20
cpi  r16, 0x20
breq is_equal
jmp  is_notequal
```

```
myisr:
push r16
in   r16, PORTA
inc  r16
out  PORTA, r16
pop  r16
reti
```

- Return from ISR with `reti`

Interrupts

- There are no “parameters” to Interrupts
- Save all registers changed in the ISR on the stack — push
- and restore them — pop — in reversed order
- Do not forget to save the SREG!

```
ldi  r16, 0x20
cpi  r16, 0x20
Interrupt → myisr
breq is_equal
jmp  is_notequal
```

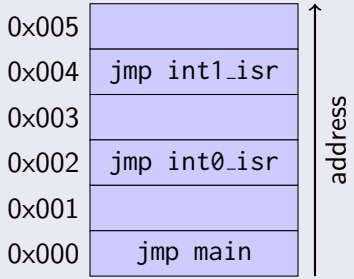
```
myisr:
push r16
in   r16, PORTA
inc  r16
out  PORTA, r16
pop  r16
reti
```

- Return from ISR with `reti`

Interrupt Vector Table

"Normally" at the beginning of Program Memory

- Reset Vector 0x000
- INT0 Vector 0x002
- INT1 Vector 0x004
- ...



Warning:

Program memory of the ATmega MCU is a 16-bit wide memory (addressed by word)

- ISR Vector Addresses are word addresses
- .org command uses byte addressing $\rightarrow$ multiply addresses by 2

Interact with the environment

Polling vs. Interrupts

- Timing more predictable.
- Prevention of missing an event
- Enter sleep mode $\Rightarrow$ conserve energy

Timed events

Small example application

- Increment PORTA every 50 kHz i.e. every 20 µs
- Every 256th increment perform some very sophisticated computation (SC), e.g., busy loop for 40 µs

Taking the easy road → let's count up and it will work!

With polling

- In an infinity loop increment a variable
- on compare-match perform the action (increment PORTA).

Will it work? Simple answer: NO!

Taking the easy road → let's count up and it will work!

With polling

- In an infinity loop increment a variable
- on compare-match perform the action (increment PORTA).

Will it work? Simple answer: NO!

Taking the easy road → let's count up and it will work!

With polling

- In an infinity loop increment a variable
- on compare-match perform the action (increment PORTA).

Will it work? Simple answer: NO!

Why not?

The timing may work, but

- high energy consumption
- integration of additional functionality
- how can a non-constant running time of the SC be handled?

Use a Timer

- A timer is a simple counter which counts a the uC frequency.
- The count frequency can be changed by a prescaler.
- By default starts at 0 and counts to a maximum value. When the maximum is reached it reset to 0 and starts again.
- The default behaviour can be modified in various ways.

Using a timer; but without interrupt

Which timer value?

- We use the Overflow flag
- thus we need an offset to `TCNT0`'s maximum value (`0xFF`).
- We decided to use a prescaler value of 8

$$\frac{16\,\text{MHz}}{8} = 2\,\text{MHz}$$

$$\frac{20\,\mu\text{s}}{\frac{1}{2\,\text{MHz}}} = \frac{20\,\mu\text{s}}{500\,\text{ns}} = 40$$

$$255 - 40 + 1 = 216 = \texttt{0xD8}$$

Using a timer; but without interrupt

```
.equ temp, 0x10
.equ clrt, 0x11
.section .text
.global main
.org 0x0000
  rjmp   main
main:
  ; initialize stack pointer
  ldi    temp, lo8(RAMEND)
  out    SPL,  temp
  ldi    temp, hi8(RAMEND)
  out    SPH,  temp

  ; setup PORTA
  ldi    temp,  0xFF
  out    DDRA,  temp
  out    PORTA, temp

  ; configure Timer0
  ldi    temp,  0x00
  out    TCCR0A, temp
  ldi    temp,  0xD8
  out    TCNT0,  temp
  ldi    clrt, (1<<TOIE0)
  ; start clock
  ldi    temp, (1<<CS01)
  out    TCCR0B, temp
```

```
infinite_loop:
  ; check if timer has overrun
  in     temp, TIFR0
  andi   temp, (1<<TOIE0)
  breq   no_ov_occured
ov_occured:
  ; reset interrupt flag
  out    TIFR0, clrt
  ; reset Timer
  ldi    temp, 0xD8
  out    TCNT0, temp
  ; increment port
  in     temp, PORTA
  inc    temp
  out    PORTA, temp
  brne   no_overflow

  ; SC
overflow:
  ldi  r18, 255
loopers:
  dec  r19
  brne loopers

no_overflow:
no_ov_occured:
  rjmp   infinite_loop
```

Using a timer; but without interrupt

Observation

Unstable frequency.

This is due to

- the imperfect alignment of the occurrence of the interrupt and the check if it has occured, and
- the fact that TCNT is constantly incrementing when the timer is running, and we are changing it at some point.

This leads to either the value we wanted, or a 'few' increments more.

Using a timer with Output-Compare-Match, still no interrupt

Which timer value?

- We use the Output-Compare Match and use a prescaler value of 8
- We use the same calculation as before.
- Notice that is important to substract 1 from 40 to account for the increment from 0 to 1.

$$\frac{20\,\mu s}{\frac{1}{16\,MHz/8}} = \frac{20\,\mu s}{500\,ns} = 40$$

$$OCR0A = 40 - 1$$

- You can also use the formula on page 214 of the ATmega1280 manual.
- Note: the frequency of this formula is for the signal "generated" by the interrupt. Thus we double the frequency to get the interrupt frequency!

Using a timer with Output-Compare-Match, still no interrupt

```
.equ temp, 0x10
.equ clrt, 0x11

.section .text
.global main
.org 0x0000
  rjmp   main
main:
  ; initialize stack pointer
  ldi    temp, lo8(RAMEND)
  out    SPL,  temp
  ldi    temp, hi8(RAMEND)
  out    SPH,  temp

  ; setup PORTA
  ldi    temp,  0xFF
  out    DDRA,  temp
  out    PORTA, temp

  ; configure Timer0
  ldi    temp,  (1<<WGM01)
  out    TCCR0A, temp
  ldi    temp,  0x27
  out    OCR0A,  temp
  ldi    temp,  0x00
  out    TCNT0,  temp
  ldi    clrt, (1<<OCF0A)
  ; start clock
  ldi    temp, (1<<CS01)
  out    TCCR0B, temp

infinite_loop:
  ; check if OC-Interrupt has occured
  in     temp, TIFR0
  andi   temp, (1<<OCF0A)
  breq   no_ov_occured
ov_occured:
  out    TIFR0, clrt
  in     temp, PORTA
  inc    temp
  out    PORTA, temp
  brne   no_overflow

  ; SC
overflow:
  ldi  r18, 255
loopers:
  dec  r19
  brne loopers

no_overflow:
no_ov_occured:
  rjmp   infinite_loop
```

Using a timer with Output-Compare-Match, still no interrupt

Observation

Frequency more stable.
Incorrect period on overflow/SC!

Using a timer with Output-Compare-Match Interrupt

```
.equ temp, 0x10

.section .text
.global main
.org 0x0000
  rjmp    main
.org OC0Aaddr*2
  rjmp    ov_occured
main:
  ; initialize stack pointer
  ldi     temp, lo8(RAMEND)
  out     SPL, temp
  ldi     temp, hi8(RAMEND)
  out     SPH, temp

  ; setup PORTA
  ldi     temp, 0xFF
  out     DDRA, temp
  out     PORTA, temp

  ; configure timer
  ldi     temp, (1<<WGM01)
  out     TCCR0A, temp
  ldi     temp, 0x27
  out     OCR0A, temp
  ldi     temp, 0x00
  out     TCNT0, temp
  ldi     temp, (1<<OCIE0A)
  sts     TIMSK0, temp
  ; start clock
  ldi     temp, (1<<CS01)
  out     TCCR0B, temp

  sei
infinite_loop:
  rjmp    infinite_loop

ov_occured:
  in      temp, PORTA
  inc     temp
  out     PORTA, temp
  brne    no_overflow

  ; SC
overflow:
  ldi  r18, 255
loopers:
  dec  r19
  brne loopers

no_overflow:
  reti
```

Using a timer with Output-Compare-Match Interrupt

Observation

Frequency stable.
Still incorrect period on overflow/SC!

Timer with OC-Match Interrupt, non-blocking ISR

```
.equ temp, 0x10

.section .text
.global main
.org 0x0000
  rjmp    main
.org OC0Aaddr*2
  rjmp    ov_occured
main:
  ; initialize stack pointer
  ldi     temp, lo8(RAMEND)
  out     SPL, temp
  ldi     temp, hi8(RAMEND)
  out     SPH, temp

  ; setup PORTA
  ldi     temp, 0xFF
  out     DDRA, temp
  out     PORTA, temp

  ; configure timer
  ldi     temp, (1<<WGM01)
  out     TCCR0A, temp
  ldi     temp, 0x27
  out     OCR0A, temp
  ldi     temp, 0x00
  out     TCNT0, temp
  ldi     temp, (1<<OCIE0A)
  sts     TIMSK0, temp
  ; start clock
  ldi     temp, (1<<CS01)
  out     TCCR0B, temp

  sei
infinite_loop:
  rjmp    infinite_loop

ov_occured:
  in      temp, PORTA
  inc     temp
  out     PORTA, temp
  brne    no_overflow
  sei

  ; SC
overflow:
  ldi   r18, 255
loopers:
  dec   r19
  brne loopers

no_overflow:
  reti
```

Timer with OC-Interrupt, non-blocking ISR

Observation

Behaviour as specified.

Do we need the infinity loop?

Timer with OC-Interrupt, non-blocking ISR, and sleep mode

```
.section .text
.global main
.org 0x0000
  rjmp    main
.org OC0Aaddr*2
  rjmp    ov_occured
main:
  ; initialize stack pointer
  ldi     temp, lo8(RAMEND)
  out     SPL, temp
  ldi     temp, hi8(RAMEND)
  out     SPH, temp
  ; setup PORTA
  ldi     temp, 0xFF
  out     DDRA, temp
  out     PORTA, temp
  ; configure timer
  ldi     temp, (1<<WGM01)
  out     TCCR0A, temp
  ldi     temp, 0x27
  out     OCR0A, temp
  ldi     temp, 0x00
  out     TCNT0, temp
  ldi   temp, (1<<OCIE0A)
  sts   TIMSK0, temp
  ; start clock
  ldi   temp, (1<<CS01)
  out   TCCR0B, temp

  sei
infinite_loop:
  ; goto sleep
  cli
  ldi temp, (1<<SE)
  out SMCR, temp
  sei
  sleep
  rjmp    infinite_loop

ov_occured:
  in      temp, PORTA
  inc     temp
  out     PORTA, temp
  brne    no_overflow
  sei

  ; SC
overflow:
  ldi   r18, 255
loopers:
  dec   r19
  brne loopers

no_overflow:
  reti
```

Timer with OC-Interrupt, non-blocking ISR, and sleep mode

Observation

Behaviour as specified.
Lower Temperature $\rightarrow$ lower energy consumption!

Note

If only the frequency generated by the LSB would be required (for output), then use the port toggle feature of the OCR-module! This does not require an ISR call, and thus will also work when interrupts are currently disabled, e.g., by extended SC.

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)
- What 'features' of the MC do you need?
  - Outputs/Inputs
  - ADC/Timer/...

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)
- What 'features' of the MC do you need?
  - Outputs/Inputs
  - ADC/Timer/...
- How should they interact? Are interrupts needed?

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)
- What 'features' of the MC do you need?
  - Outputs/Inputs
  - ADC/Timer/...
- How should they interact? Are interrupts needed?
- Consider the Control/Data-Flow (Petri-Net, state machine, structograms, flow chart, ...)

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)
- What 'features' of the MC do you need?
  - Outputs/Inputs
  - ADC/Timer/...
- How should they interact? Are interrupts needed?
- Consider the Control/Data-Flow (Petri-Net, state machine, structograms, flow chart, ...)
- Modularize (Functions)

How-to start programing a Microcontroller application (in Assembler)

- Think about what you want to do (obviously)
- What 'features' of the MC do you need?
  - Outputs/Inputs
  - ADC/Timer/...
- How should they interact? Are interrupts needed?
- Consider the Control/Data-Flow (Petri-Net, state machine, structograms, flow chart, ...)
- Modularize (Functions)
- Implement and test the modules.
  Use a consistent, and clean, programming style!

Assembler Guidelines

Peak at the compiler output

- `gcc -S code.c` create an assembler file.
- this can be a good source of negative examples.
- depending on optimizer parameters, this can be very verbose.

Assembler Guidelines

Debug systematically

- it is nearly impossible to code assembler “blindly”, i.e., without continuous testing.
- debug only small code blocks simultaneously.
- use LEDs to display current state (registers) while debugging.

Assembler Guidelines

Be redundant (sometimes)

- assembler code is very susceptible to hard-to-see mistakes.
- e.g., create redundant labels just to clarify the control flow:

```
    cpi r16, 1
    breq equals_one
not_equals_one:
    ...
equals_one:
    ...
```

- label not_equals_one is redundant here, but documents the control flow.

Questions?