

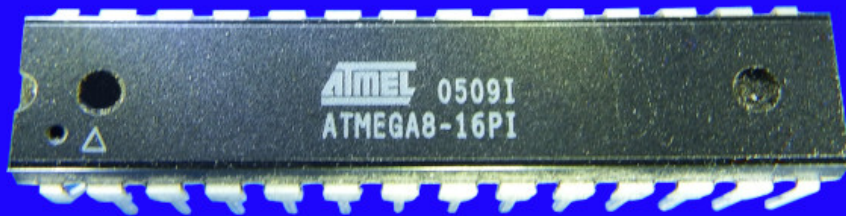
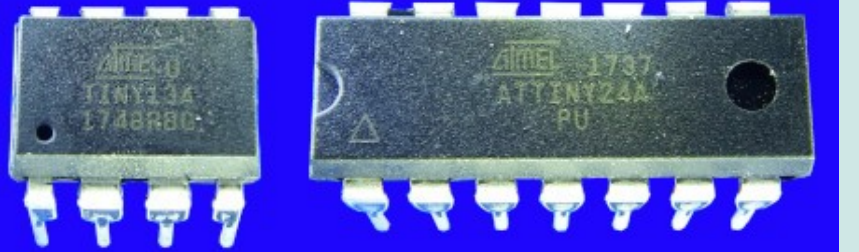


AVR Microcontrollers

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

AVR Microcontrollers



Microcontrollers are Integrated Circuits (ICs). They can have 6, 8, 14, 20, 28, 40, 64 or 100 pins.

AVRs are available in more than 460 different types. So they can be tailored to very different hardware needs.

They are equipped with lots of internal hardware, which can be involved when needed by programming them. This is why such controllers are extremely versatile and can be used for thousands of purposes.

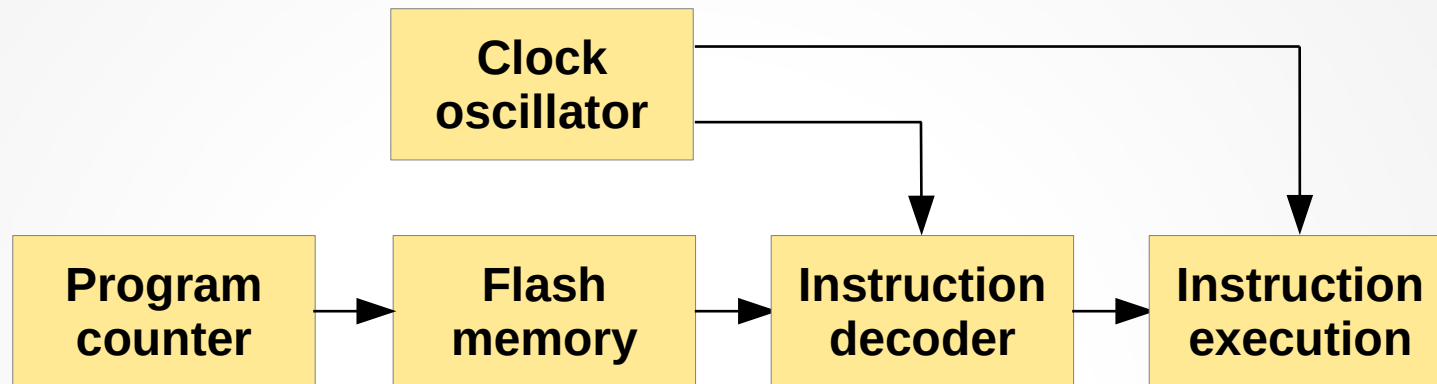
Their main features are:

- 1) Very small power consumption (a few mA).
- 2) Very fast reaction to external events (a few μ s).
- 3) Flash memory, static RAM and durable EEPROM on board.



Program execution in AVR

- The program of the AVR is located in the flash memory as 16-bit binary words.



- Driven by a clock oscillator the program counter addresses the next instruction to be executed, a decoder splits this 16-bit-word into individual steps and the instruction executor (called Central Processing Unit = CPU) executes these.
- AVR uses pre-fetch: while the CPU executes the last instruction, the decoder already decodes the next instruction word. If the last execution did not change the program counter, the next instruction can be executed immediately. Most instructions need only ONE clock oscillator cycle to be executed. This nearly doubles the execution speed (MIPS, Mega-Instructions Per Second) on the same clock frequency.

Unique: Plenty registers in AVR

- AVR has 32 registers on board (ATtiny4/5/9/10: 16).
- Each register provides 8 bit storage space (256 single bits).
- Setting a register to a constant number is achieved by the following assembler source code (text on green background is assembler):

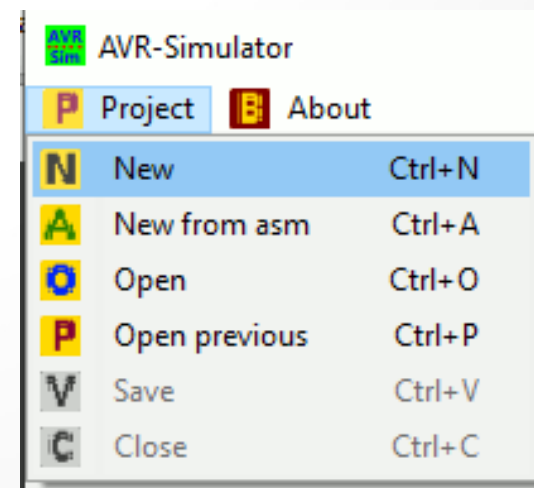
; Loading a number to a register in assembler source code format
LDI R16, 85 ; Write the decimal number 85 to register number 16
LDI R16, 0B01010101 ; Write the binary number 01010101 to register 16
LDI R16, 0X55 ; Write the hexadecimal number 55 to register 16

- The 32 registers can be used as source and as target of instructions.
- The following assembler source code
 - loads decimal 15 to register R16,
 - loads decimal 55 to register R17,
 - adds both numbers and writes the result to R17.

; Loading and adding two numbers
LDI R16, 15 ; Write the decimal number 15 to register number 16
LDI R17, 55 ; Write the decimal number 01010101 to register 17
ADD R17, R16 ; Add the content of R16 to R17 and write the result to R17

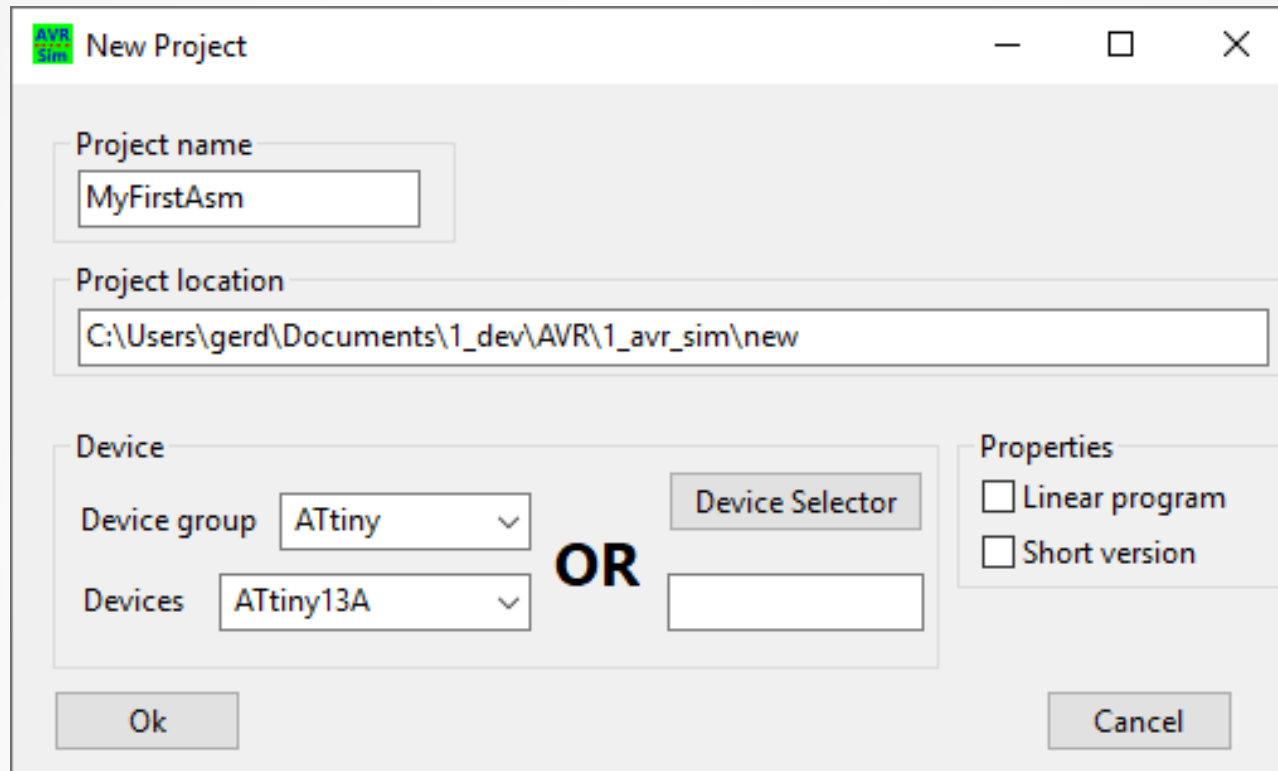
Simulating assembler code

- The three instructions produce a result. But which result?
- To see it, we have to simulate an AVR on the PC or laptop.
- We can use avr_sim for that. It is available as Windows- or Linux-64 executable at http://www.avr-asm-tutorial.net/avr_sim/index_en.html.
- After starting it it asks for a folder that assembler projects will be located in. Choose a convenient folder in your harddrive for that.
- In the Project menu select New.



A new assembler project

- This window opens:



The screenshot shows the 'New Project' dialog box in AVR Sim. The window has a title bar with the AVR Sim logo and standard minimize, maximize, and close buttons. The dialog is divided into several sections:

- Project name:** A text box containing 'MyFirstAsm'.
- Project location:** A text box containing 'C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\new'.
- Device:** This section contains two rows of dropdown menus. The first row has 'Device group' set to 'ATtiny'. The second row has 'Devices' set to 'ATtiny13A'. To the right of these dropdowns is a large 'OR' text and an empty text box.
- Device Selector:** A button located to the right of the 'Device group' dropdown.
- Properties:** A section on the right with two checkboxes: 'Linear program' and 'Short version', both of which are currently unchecked.
- Buttons:** At the bottom, there are 'Ok' and 'Cancel' buttons.

- Unselect „Interrupts“ and „Comprehensive“ in the properties field.
- Fill in the information like here.
- Press „Ok“ when done.

Package selection

- Select a package form for the ATtiny13.
- Press „Ok“.

Device selection

Currently selected device
Name Ni= Np=

Previously selected device
Name Ni= Np=

Select from available packages

☒ 8-pin-PDIP_SOIC
☐ 20-pin-QFN
☐ 10-pin-QFN

Show package

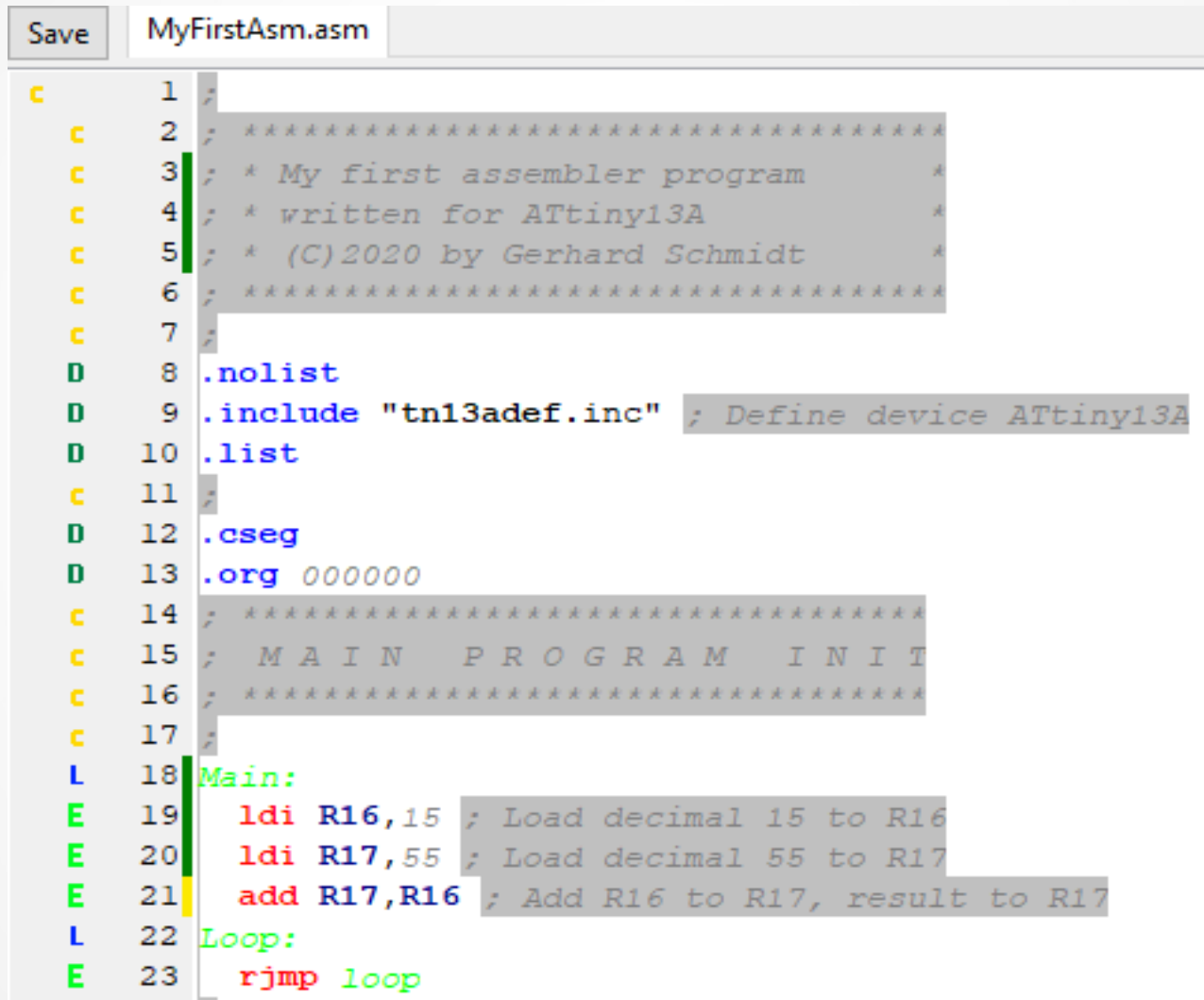
Device selector
Group

AT90S1200
AT90S2313
AT90S2323
AT90S2333
AT90S2343
AT90S4414
AT90S4433
AT90S4434
AT90S8515
AT90S8535
AT90C8534
AT86RF401
ATtiny4
ATtiny5
ATtiny9
ATtiny10
ATtiny11

Ok Press ok when done! Cancel

Editing the asm source code

- Fill in your three code lines for adding after „Main:“.



```
Save MyFirstAsm.asm
1 ;
2 ; *****
3 ; * My first assembler program *
4 ; * written for ATtiny13A *
5 ; * (C)2020 by Gerhard Schmidt *
6 ; *****
7 ;
8 .nolist
9 .include "tn13adef.inc" ; Define device ATtiny13A
10 .list
11 ;
12 .cseg
13 .org 000000
14 ; *****
15 ; M A I N   P R O G R A M   I N I T
16 ; *****
17 ;
18 Main:
19     ldi R16,15 ; Load decimal 15 to R16
20     ldi R17,55 ; Load decimal 55 to R17
21     add R17,R16 ; Add R16 to R17, result to R17
22 Loop:
23     rjmp loop
```


Assembling and simulating

- Now press „Assemble“ and then „Simulate“.

```
15: ; (C) 2020 by Gertjan Bormio
16: ; *****
17: ;
18: ; .nolist
19: ;
20: ; .cseg
21: ; .org 000000
22: ; *****
23: ; MAIN PROGRAM INIT
24: ; *****
25: ;
26: ; Main:
27: 19: 000000 E00F ldi R16,15 ; Load decimal 15 to R16
28: 20: 000001 E317 ldi R17,55 ; Load decimal 55 to R17
29: 21: 000002 0F10 add R17,R16 ; Add R16 to R17, result
30: 22: Loop:
31: 23: 000003 CFFF rjmp loop
32: 24: ;
33: 25: ; End of source code
34: 26: ;
```

The screenshot shows the AVR simulator interface. The 'Simulation' window has a toolbar with buttons for Restart (R), Step (S), Skip (P), Run/Go (G), and Stop (X). Below the toolbar, the 'Simulation status' panel displays the following information:

- Prog counter = \$000000
- Instructions = 0
- Stackpointer = \$00000
- Watchdog = 0.00000%
- Clock frequ. = 1,200,000 Hz
- Time elapsed = 0.00 ns
- Stop watch = 0.00 ns

To the right of the status panel is the 'SREG' register table:

I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

Below the SREG table are two input fields: 'Update status Instructions' with the value '1000' and 'Step Delay ms' with the value '10'.

Below these fields is the 'Register' table:

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

At the bottom of the window is the 'Messages' panel, which shows the message '\$0000: Starting'.

- The simulation window now shows
 - the program counter (always starts at address zero),
 - the number of executed instructions (none so far),
 - the clock frequency of the AVR (1.2 MHz by default),
- The blue arrow in the editor window left points to the line in the assembler listing, where the first executable instruction is located. This is the instruction that will be executed next (when STEP will be clicked).

Stepping through the code

- Now press „Step“ in the simulation window.

The screenshot shows the AVR Simulator window with the following components:

- Simulation status:**
 - Prog counter = \$000001
 - Instructions = 1
 - Stackpointer = \$0000
 - Watchdog = 0.00000%
 - Clock frequ. = 1,200,000 Hz
 - Time elapsed = 833.333 ns
 - Stop watch = 833.333 ns
 - Sleep share = 0.00000%
- SREG:**

I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0
- Update status Instructions:** 1000
- Step Delay ms:** 10
- Register:**

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	0F	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00
- Messages:**

\$0000: Starting
- Show internal hardware:**
 - ☐ Ports
 - ☐ Timers/counters
 - ☐ WDT
 - ☐ ADC
 - ☐ Scope
 - ☐ EEPROM

- The program counter has advanced.
- The number of instructions is 1.
- The time elapsed is 833.333 ns.
- The register R16 is now 0X0F, which is decimal 15.

Hints for better understanding

- The binaries produced by the assembler can be seen in the listing (E00F at address 0, E317 at 1, etc.). This is what the controller finally gets, it is in its pure form in the text file MyFirstAsm.hex.
- The LDI in the source code means „Load Immediate“ and only the assembler understands it (the controller doesn't). These abbreviations are called „Mnemonics“, simple to remember. ADD is another mnemonic.
- AVR's know more than 100 different instructions, all have at least one, a few have two different mnemonics.
- The first parameter behind the instruction mnemonic is always the target register of the instruction, where the result is written to.

Simulation visualizes execution

- If you press „Step“ two further times, you'll see the result of adding 15 and 55 in Register R17.
- The elapsed time is still only in the microseconds range: very fast.
- Thanks to simulating we see what will be going on inside the microcontroller.
- We'll use this tool later on whenever we need to understand more complex instructions.

Basic rules in assembler

- Use a simple ASCII text editor to generate source code, no formatting informations are allowed.
- Anything behind a „;“ on a source code line are comments and are ignored by the assembler.
- Upper case and lower case letters are all converted to upper case during assembling. Use upper and lower cases for your own convenience.
- Mnemonics and parameters have to be separated by blanks or tab characters.
- The second parameter has to be separated with a comma („LDI R16,55“).
- Labels are jump targets and have to be immediately followed by a „:“ character (see Lecture 2 for an example).

Why assembler and not C?

- **Assembler translates each line of the source code directly to its binary representation. It can be seen directly, what the controller does with it.**
- **C is in-transparent: no such direct connection between source code and executed instructions. One line of C source code can cause thousands of single instructions and the programmer has no chance to understand what is really going on inside.**
- **In assembler you'll learn exactly how the internal hardware works. No hidden stuff between you and the hardware.**
- **In C you can program without even knowing something about the controller: the compiler is between you and the controller, and you have no chance to understand.**

Questions and Tasks for Lecture 1

Question 1-1: Nearly all source code examples use Register # 16. Why not Register # 0?

(Hint: Change the source code of one example to use R0 instead of R16 and try to understand the resulting error message when assembling.)

Try to find out which types of instructions can only use R16 and above (Use the Instruction Set Manual for 8-bit-AVRs provided by Microchip for that).

Questions and Tasks for Lecture 1, Continued

Question 1-2: Which range of integer numbers can be handled

- a) in 8-bits,**
- b) in 16 bits,**
- c) in 24 bits, and**
- d) in 32 bits.**

(Hint: Under Windows use the provided calculator in the „Programmer“ mode and switch to the Hexadecimal input mode.)



Questions and Tasks for Lecture 1, Continued

Task 1-3: Find out how many registers the following CPUs provided and at which maximum clock frequencies they ran:

a) ZUSE Z4

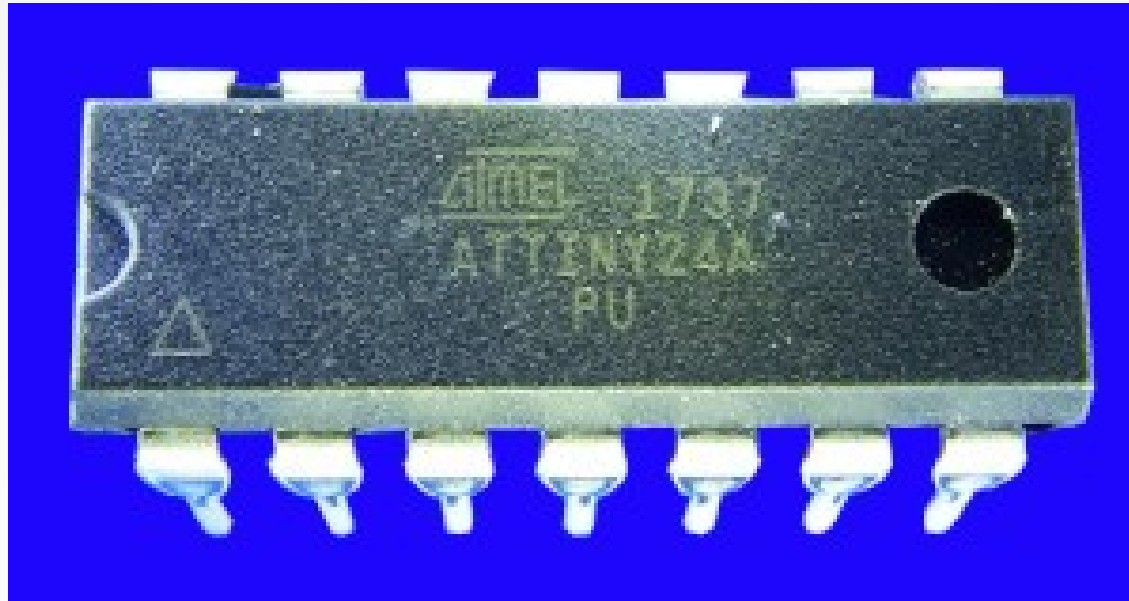
b) PIC8

c) 8086

d) Z80

e) 6502

Compare that with an ATtiny13 and ATtiny24 (use Microchips Data Books for these).



Lecture 2: Manipulating pins

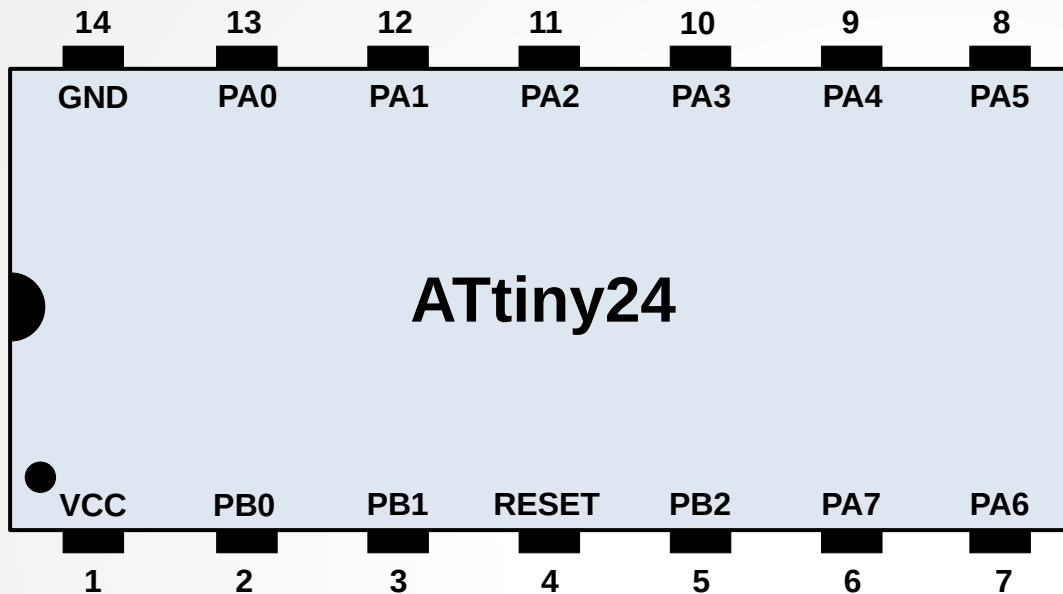
Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

How to manipulate pins

- The pins of an ATtiny24 as seen from above:



Pins 1 and 14 are for the supply voltage: VCC: +2.7 to +5.5 Volt, GND: 0 Volt

Pin 4: RESET, LOW stops the controller, LOW to HIGH restarts the program at address 0, should be tied with a resistor of 10 k Ω to VCC

- The ATtiny24 has eight pins called port A: PA0 to PA7
- Additionally it has three pins of a port B: PB0 to PB2
- Each of those 11 pins can either be an input or an output pin
- If a pin is input, a Pull-Up resistor of 50 k Ω can be switched on to tie the pin to VCC (if no external connection pulls it LOW)

Making a port to be output

- Switching a port to be output is simple, just set its direction bit to ONE.
- The direction bits are located in a memory space called „port registers“, named data direction register „DDRA“ and „DDRB“ at the addresses 0x1A and 0x17:

22. Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C
0x3E (0x5E)	SPH	–	–	–	–	–	–	SP9	SP8
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x3C (0x5C)	OCR0B	Timer/Counter0 – Output Compare Register B							
0x3B (0x5B)	GIMSK	–	INT0	PCIE1	PCIE0	–	–	–	–
0x3A (0x5A)	GIFR	–	INTF0	PCIF1	PCIF0	–	–	–	–
0x39 (0x59)	TIMSK0	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0
0x38 (0x58)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0



0x1B (0x3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
0x1A (0x3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
0x19 (0x39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
0x18 (0x38)	PORTB	–	–	–	–	PORTB3	PORTB2	PORTB1	PORTB0
0x17 (0x37)	DDRB	–	–	–	–	DDB3	DDB2	DDB1	DDB0
0x16 (0x36)	PINB	–	–	–	–	PINB3	PINB2	PINB1	PINB0

Making a port to be output

- To make the whole port A output, just do the following:

; Making port A to be output

LDI R16, 0xFF ; All bits in register 16 to one

OUT 0x1A, R16 ; Write the content of R16 to port register 0x1A

- Generate a new project in the simulator, select the ATtiny24 or ATtiny24A and add these three lines behind the Main: line.
- The same does the following:

; Making port A to be output

SER R16 ; All bits in register 16 to one

OUT DDRA, R16 ; Write the content of R16 to port register DDRA

- SER („Set register“) is just an additional mnemonic. The binary produced by the assembler is the same (compare the two binary codes in the listing).
- The use of DDRA as symbol name simplifies the code. The symbol DDRA translates to 0x1A in the assembler. It is defined in the header file tn24def.inc, which is included on top of the source code.

Manipulating single pins

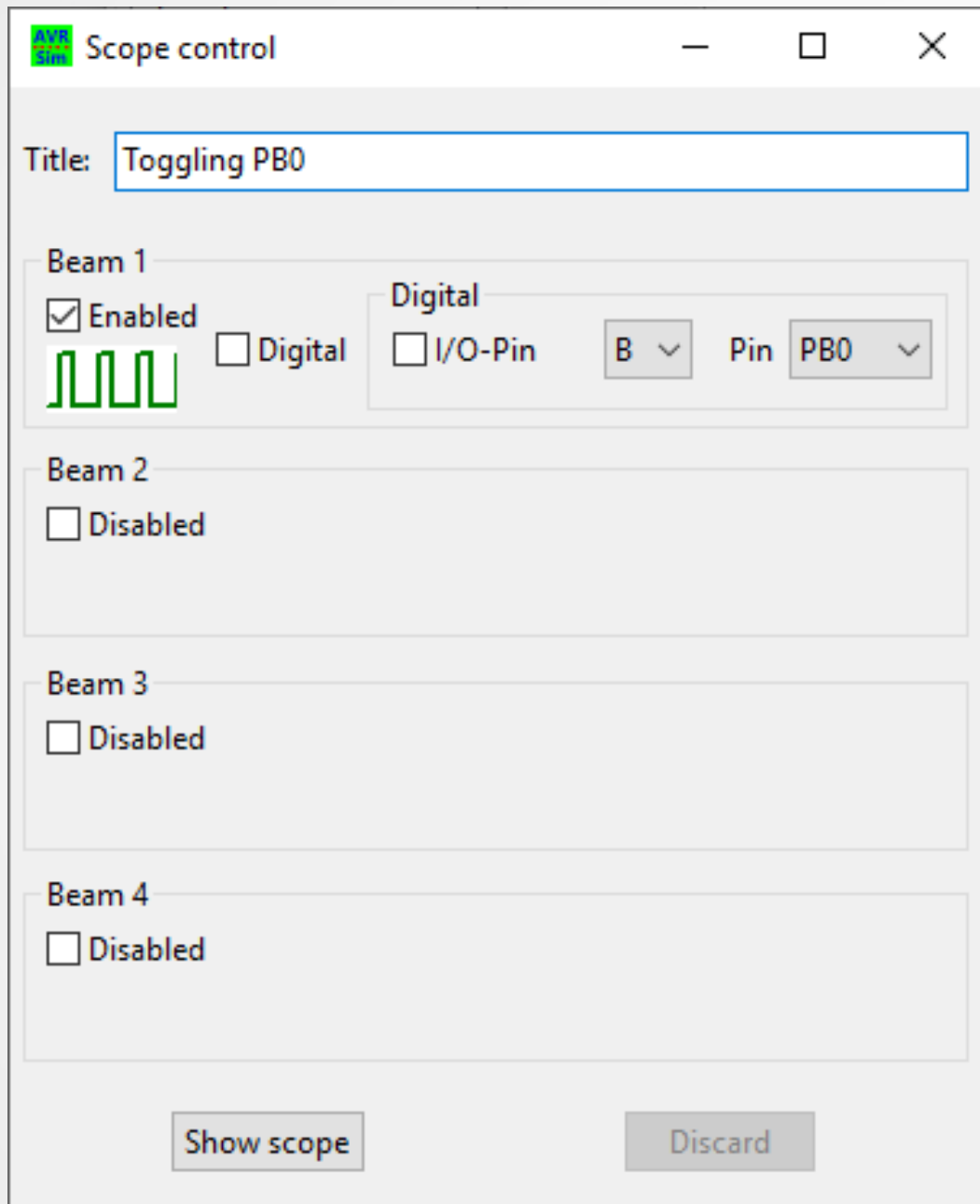
- **If you need to set or clear a single output pin use the following code:**

```
; Making portpin PB0 output and setting it HIGH  
SBI DDRB, DDB0 ; Set the direction bit of PB0 high  
SBI PORTB, PORTB0 ; Set the output port pin PB0 HIGH
```

- In the simulator, in the „View hardware“ section click the „Ports“ field to view the ports of the controller. In the port window use „Previous“ and „Next“ to switch between port A and port B.
- The following switches the port pin PB0 on and off and generates a rectangle on PB0.

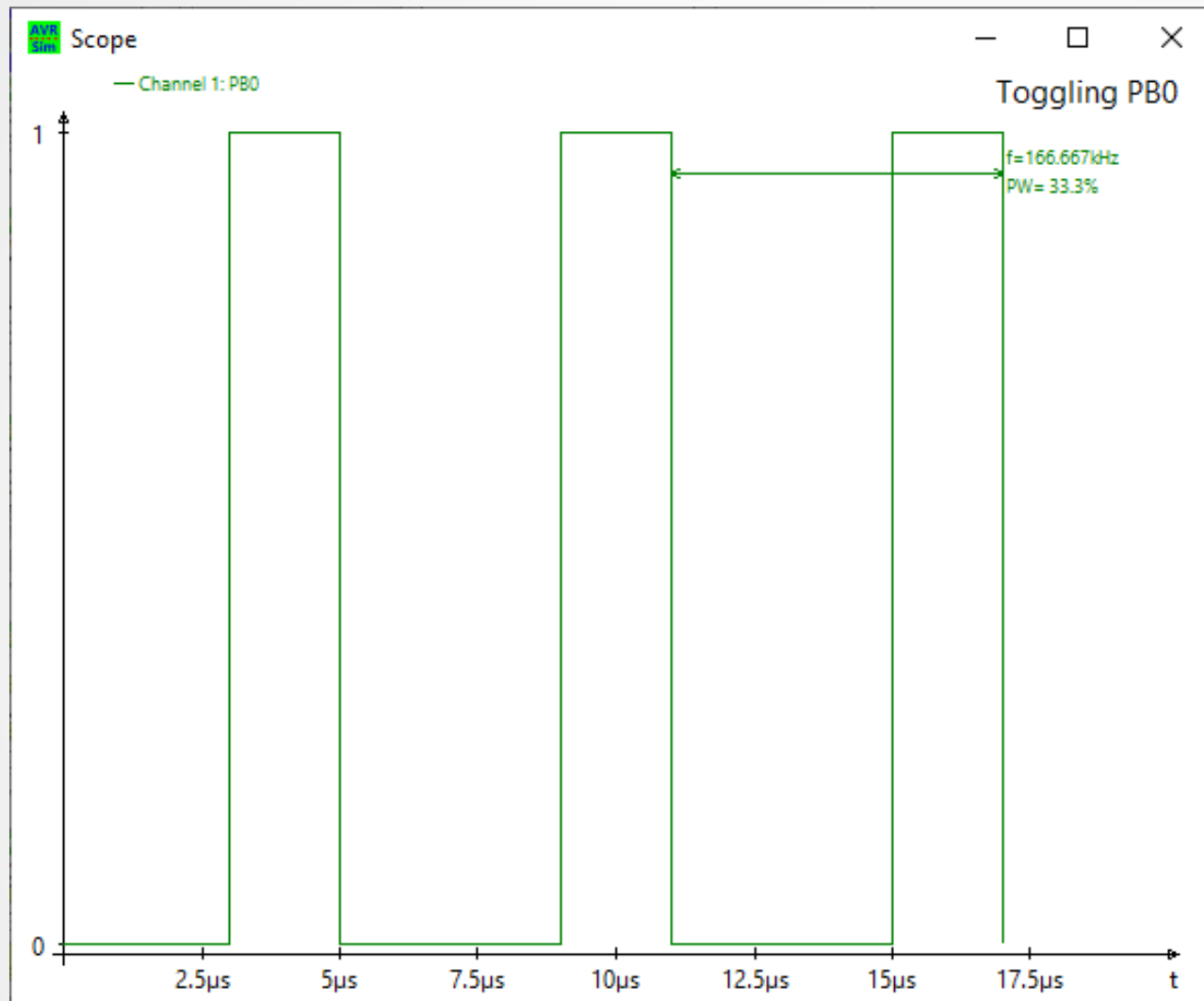
```
; Toggling portpin PB0  
SBI DDRB, DDB0 ; Set the direction bit of PB0 high  
Toggle:  
SBI PORTB, PORTB0 ; Set the output port pin PB0 to HIGH  
CBI PORTB, PORTB0 ; Clear the output port pin PB0 to LOW  
RJMP Toggle ; Jump back to the label Toggle
```

Scope view of the toggling pin



- In the simulation window in the section „View hardware“ enable the field „Scope“.
- Click „Disabled“, select „Digital“ and I/O-Pin, „Port B“ and „PB0“. Fill in a title and click „Show scope“.

Scope display of the toggling



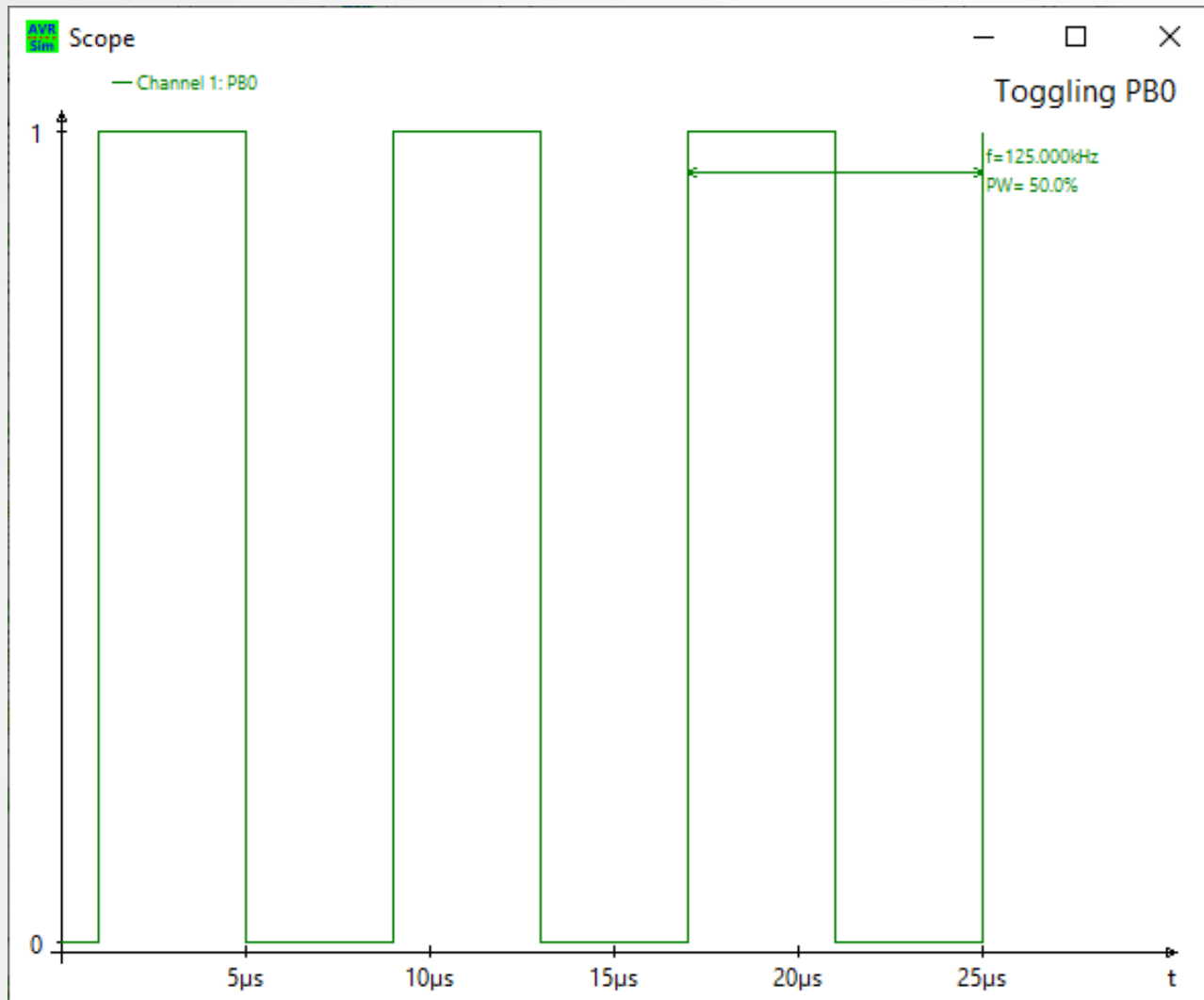
- If you step through the code, you'll see the rectangle to the left.
- The frequency is 167 kHz, the pulse width is 33.3%.
- The reason for this is that the set instruction is immediately followed by the clear instruction, while the RJMP is between clear and the next set.

No operation (NOP)

- The delay introduced by the RJMP has to be compensated to reach a pulse width of 50%.
- An instruction that simply consumes time but does nothing else is NOP. It consumes one clock cycle.
- The RJMP changes the program counter. So the pre-fetch of the controller fails and the RJMP consumes two clock cycles.
- SBI and CBI also consume two clock cycles each because the content of DDRB has to be read and the DDB0 bit has to be set and written back to DDRB.
- To compensate the RJMP we add two NOP instructions.

```
; Toggling portpin PB0 with 50% pulse width
    SBI DDRB, DDB0 ; Set the direction bit of PB0 high
Toggle:
    SBI PORTB, PORTB0 ; Consumes two clock cycles
    NOP ; Consumes one clock cycle
    NOP ; Consumes one clock cycle
    CBI PORTB, PORTB0 ; Consumes two clock cycles
    RJMP Toggle ; Consumes two clock cycles
```

The ideal rectangle



- Now the rectangle has exactly 50% pulse width.
- Because the whole execution cycle is eight clock cycles long and as the ATtiny24 is clocked with 1 MHz by default, the frequency is $1 \text{ MHz}/8 = 125 \text{ kHz}$.

Prolonging the cycle

- We can add further NOPs to prolong the cycle.

```
; Toggling portpin PB0 at 100 kHz with 50% pulse width  
SBI DDRB, DDB0 ; Set the direction bit of PB0 high
```

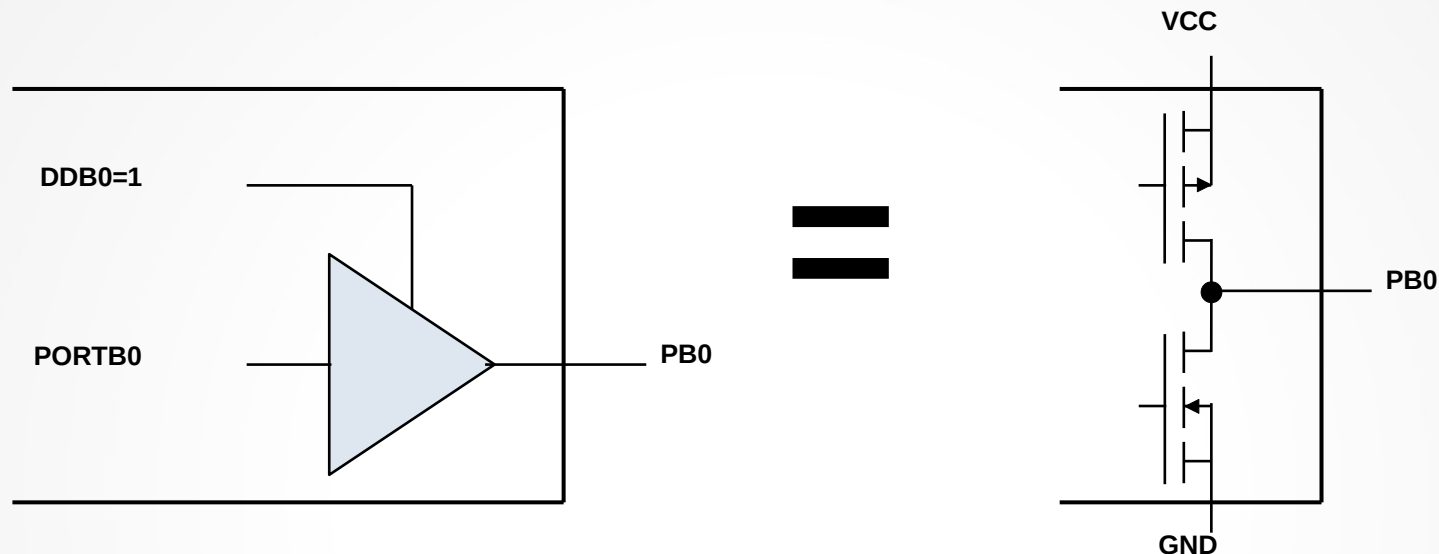
Toggle:

```
SBI PORTB, PORTB0 ; Consumes two clock cycles  
NOP ; Consumes one clock cycle  
NOP ; Consumes one clock cycle  
NOP ; Add another clock cycle during HIGH  
CBI PORTB, PORTB0 ; Consumes two clock cycles  
NOP ; And another clock cycle during LOW  
RJMP Toggle ; Consumes two clock cycles
```

- As now the whole cycle is 10 clock cycles long, the frequency is $1 \text{ MHz} / 10 = 100 \text{ kHz}$.
- We can add further NOPs to prolong the cycle. But the effect is rather limited and we cannot reach really small frequencies.
- Very low frequencies require longer delay times.

Properties of output pins

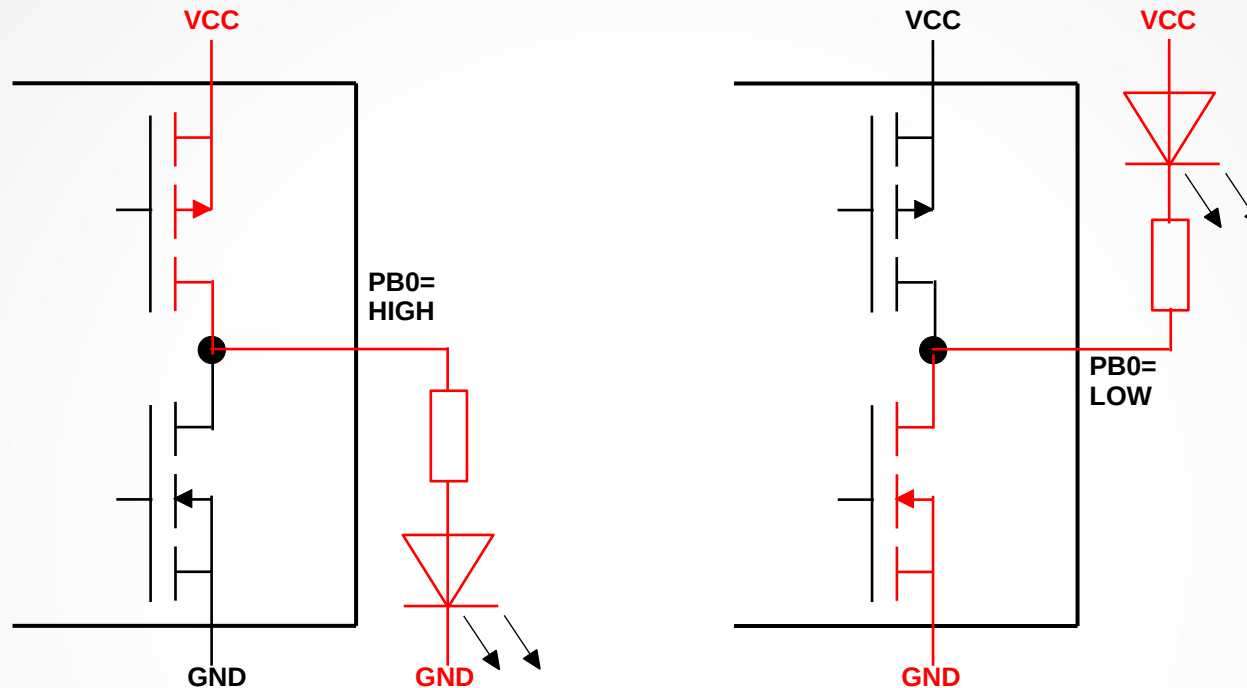
- The output pins are designed as follows:



- The DD-bit switches the output driver on/off.
- If the DD-bit is on, the output follows the state of the PORT bit.
- The output pins are driven by two transistors.

Outputs driving High and Low

- The driving transistors can provide current in both directions:



- If the output pin is high, the upper transistor connects the output pin to VCC. An external component can **source** current to GND.
- If the output pin is low, the lower transistor connects the output pin to GND. An external component can **sink** current from VCC through the output pin.

Further properties of outputs

- **The outputs are protected against short circuits.**
- **They can sink or source up to 40 mA.**
- **Voltage losses are between 0.5 to 0.7 Volt at 10 mA.**

Pins as inputs

- If the DD bit is low, the pin serves as input.
- If the respective port bit is high, a pull-up resistor of around 50 k Ω forces the input pin high, unless externally pulled low.
- The current state of the pin can be read in the PIN port register.

```
; Reading port PINB  
IN R16, PINB ; Reading all input pins in port B  
; Reading portpin PB0 and jumping over the next instruction if HIGH  
SBIS PINB,PINB0 ; Read PINB0 and jump if set (HIGH)  
RJMP PinIsLow  
; Reading portpin PB0 and jumping over the next instruction if LOW  
SBIC PINB,PINB0 ; Read PINB0 and jump if clear (LOW)  
RJMP PinIsHigh
```

- In most of the AVR's writing a ONE to the PIN bit toggles the respective PORT bit.

Conclusions

- **Internal hardware can be configured by setting or clearing a few bits in port registers.**
- **All properties of the internal hardware can be changed at any time.**
- **Pins of AVR can be used as output as well as inputs.**
- **The state of input pins can be used to branch and execute depending program parts.**

Questions and Tasks for Lecture 2

Question 2-1: In the register summary are four portpins listed for PORTB. Where is the fourth portpin? Why is it not displayed in the schematic?

(Hints: Use the Device's Data Book for the complete schematic and to find out why PB3 is not listed here.)



Questions and Tasks for Lecture 2, Continued

Question 2-2: Here the ATtiny24 is shown and discussed. There is another controller named ATtiny24A. What are the differences between those two types in respect to

- a) Memory sizes (Flash, SRAM, EEPROM),
- b) Operating voltages,
- c) Clock frequencies,
- d) Electrical characteristics.

(Hint: Use the Data Books for these two types provided by Microchip)



Questions and Tasks for Lecture 2, Continued

Question 2-3: There are LEDs available that can light in two different colors (e.g. red and green), but they only have two pins. What can be done to attach such a LED

a) to two output pins, or

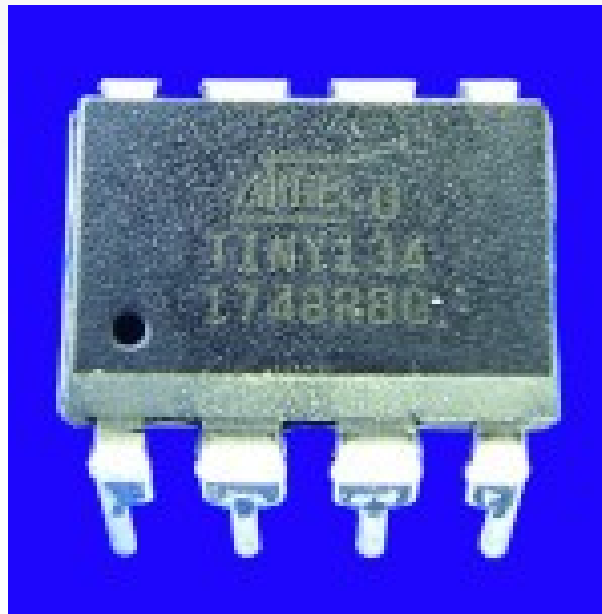
b) to a single output pin

of an AVR at 5 Volt operating voltage?

(Hint for b: The forward voltage of LEDs is approximately 2 Volt. Also consider the voltage drops when current is sourced/sinked!)

Bonus question: What would be necessary to have the LED lighting with 20% intensity in red and 80% intensity in green?





Lecture 3: Counting

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

The registers

- The AVR has 32 registers, named R0 to R31. Use the `.DEF` directive to give them a more meaningful name and use that in the source code:

```
; Renaming a register
.DEF rMyReg = R16
    LDI rMyReg, 'A' ; Load register MyReg with the ASCII character A
    INC rMyReg ; Register MyReg plus one, result is a 'B'
```

- Registers can hold any type of 8-bit information and data: numbers from 0 to 255 (hexadecimal: 0xFF), characters or just eight single bits. No type definition is required (if it's ASCII: just keep this in mind for yourself), no restrictions apply when using this data for any purpose (copy, add, subtract, multiply etc.).
- Special functions of registers are:
 - ➔ R0 is the target of the Load from Program Memory instruction `LPM` (without any parameters).
 - ➔ R1:R0 is the 16-bit register pair where the result of hardware multiplications with `MULT` is written to (R1 is MSB, R0 is LSB).
 - ➔ R0 to R15 cannot be targets of instructions that require an 8-bit constant (Load Immediate `LDI`, Set or Clear Bit in Register `SBR/CBR`, SUBtract Immediate `SUBI`, AND Immediate `ANDI`, OR Immediate `ORI`). R16 to R31 can do that.

The registers

- **Special functions of registers (continued):**
 - ➔ **R27:R26 = XH:XL = X, R29:R28 = YH:YL = Y, R31:R30 = ZH:ZL = Z** are 16-bit register pairs that can be used as address pointers to **LoaD LD** or **STore ST** registers to **SRAM**.
 - ➔ These three pairs, as well as the register pair **R25:R24** (no special name), can be added 16-bit-wise with a constant **ADd Immediate Word ADIW** or **SuBtracted Immediate Word SBIW**.
 - ➔ **R29:R28 = Y** and **R31:R30 = Z** can temporarily add a displacement to **LoaD Displaced LDD** or **STore Displaced STD** bytes.
- Therefore and because of source code transparency and overview the placing of the data to the appropriate registers is vital for assembler. No other (higher level) language requires this thorough register planning. The result is more effective binary code and increased elegancy.

The register landscape

- The landscape of registers in AVRs therefore looks like this:

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	01	02	03	04	05	06	07
R8	08	09	0A	0B	0C	0D	0E	0F
R16	10	11	12	13	14	15	16	17
R24	18	19	1A	1B	1C	1D	1E	1F

#	Unrestricted	#	LPM	#	16-bit pointer
#	No immediate	#	MULT	#	ADIW SBIW

- Keep this mind when placing data to registers. And always place them with a .DEF directive, so you can change the .DEF associations to registers easily.
- In PICs there is only one register, so this choice is unique for AVR.

Counting with the controller

- The plenty registers of an AVR can be used to count up or down.

; Counting a register up

CLR R16 ; Set register number 16 to zero

CountUp:

INC R16 ; Count the register content up

RJMP CountUp ; Jump back to the label CountUp

- We'll simulate this to see what is going on inside. Start a new project in the simulator named „Count“, select an ATtiny24.
- Add the above lines behind the „Main:“ line and assemble.
- Either step through your program, until you reach R16=255. Or:



With Run/Go or Ctrl-G see how the time advances.

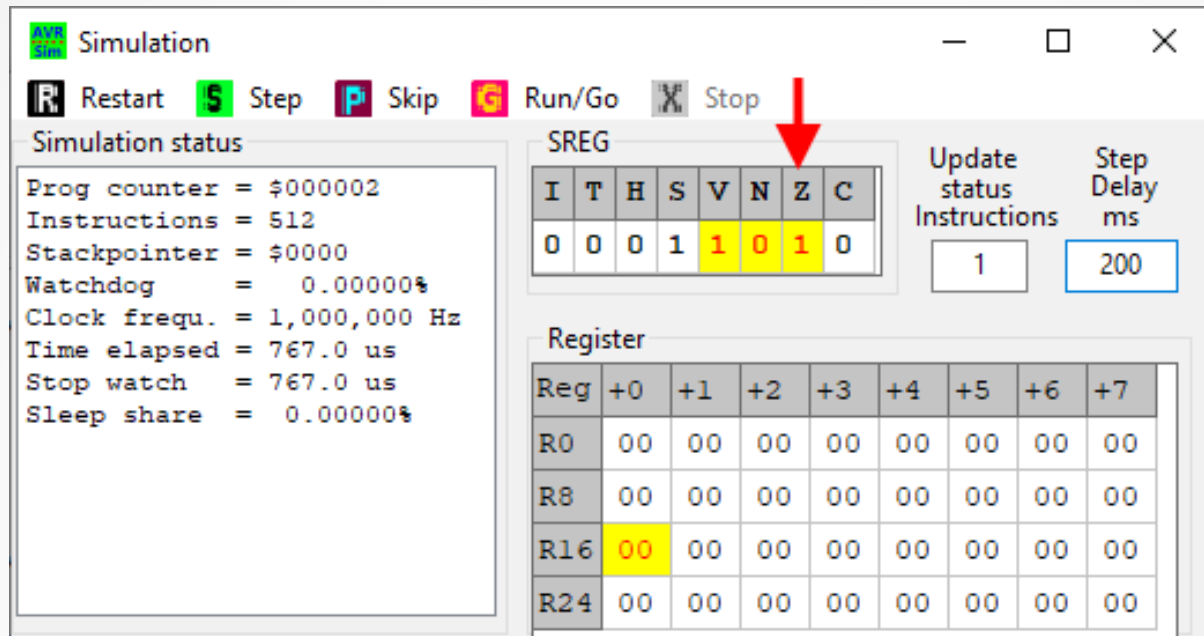
- CLR needs 1 μ s.
 - Each INC needs 1 μ s.
 - Each RJMP needs 2 μ s.
- To increase execution speed remove one zero from the 10,000.

Execution times of the loop

- If R16 reaches 0xF8 press „Stop“ or Ctrl-X.
- Then press „Step“ until R16 reaches 0xFF.
- The initial CLR needed one clock cycle, all the INCs needed 255 and all the RJMPs needed $2 \times 255 = 510$ clock cycles. Altogether 766 clock cycles. At 1 MHz that means 766 μ s.
- What we learn from that:
 - 1) All execution times for all instructions have exactly predictable durations.
 - 2) No hidden delay times or unknown factors affect this.
 - 3) That's what you get with assembler language exclusively.
- Where to know these cycles from? See the instruction list [here](#), column „Clk“ or ATMEL's Instruction Set Manual.

Status register flags

- With the next „Step“ the counter reaches zero and restarts at zero.



- The CPU has set the Z flag (Z = Zero) in its status register (SREG) to signal that the INC ended with Zero in the register.
- The Z flag can be used to branch conditionally: branching if the Z flag is set or clear.
- There are eight flags in SREG, six of them are effected by the CPU. Which? See the instruction list [here](#) or in ATMEL's Instruction Set Manual.

Conditional branching

- The conditional branching depending from the Z flag is done with BREQ or BRNE (Mnemonics for BRanch if EQal and BRanch if Not Equal). See other branches in the device's databook or [here](#).
- These can be used to construct loops: repeating counting until zero will be reached.

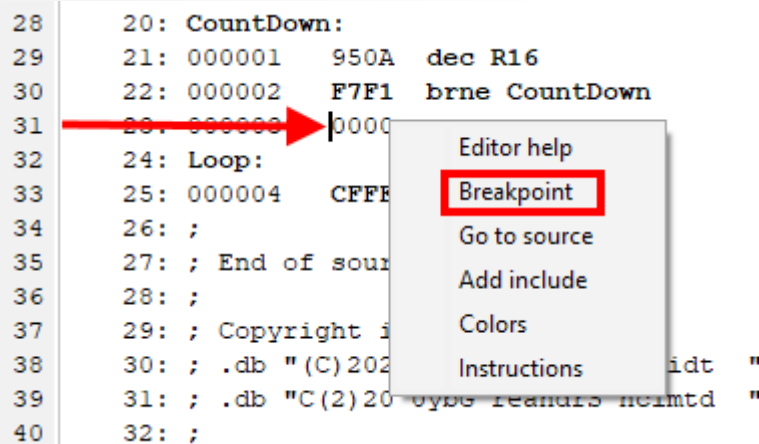
```
; Counting a register down until it is zero
    LDI R16,10 ; Set register number 16 to ten
CountDown:
    DEC R16 ; Count the register content one down (DECrease)
    BRNE CountDown ; Jump back to the label CountDown if not zero
    NOP ; Do nothing
```

- The method of repeating „Step“ until the counter reaches zero can be simplified by setting a Breakpoint.
- Breakpoints are executable lines of code, where execution stops before the code in this line is executed.

Breakpoints

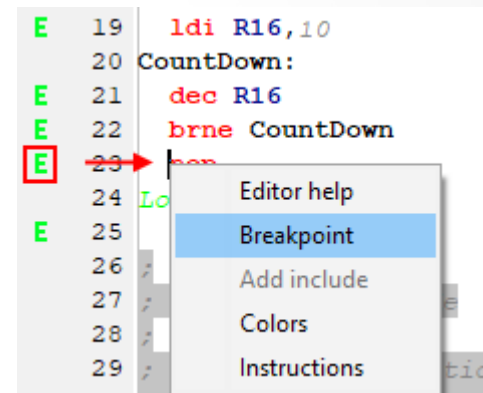
- To set a breakpoint, first assemble the source code.

In the listing place the cursor into the line where execution should stop. Then right-click and select „Breakpoint“ from the context menu.



```
28      20: Countdown:
29      21: 00000001 950A dec R16
30      22: 00000002 F7F1 brne Countdown
31      23: 00000000 0000
32      24: Loop:
33      25: 00000004 CFFF
34      26: ;
35      27: ; End of source
36      28: ;
37      29: ; Copyright (C) 2002
38      30: ; .db "(C) 2002"
39      31: ; .db "(C) 2002"
40      32: ;
```

In the source code place the cursor into the line where execution should stop. Make sure that this line has a green E. Then right-click and select „Breakpoint“ from the context menu.



```
E 19      ldi R16,10
E 20      Countdown:
E 21      dec R16
E 22      brne Countdown
E 23      Loop:
E 24      ;
E 25      ; End of source
E 26      ;
E 27      ; Copyright (C) 2002
E 28      ; .db "(C) 2002"
E 29      ; .db "(C) 2002"
```

- The breakpoint is now marked with a red „B“ in the listing as well as in the source code.
- Whenever this breakpoint will be executed as next instruction, it will stop the running simulation.

Loop execution ended

- When simulating with Run/Go the simulator stops exactly at the breakpoint and the color of the execution arrow turns from blue to red.

28	20: Countdown:
29	21: 000001 950A dec R16
30	22: 000002 F7F1 brne Countdown
→ 31	23: 000003 0000 nop

Simulation status

```
Prog counter = $000003
Instructions = 21
Stackpointer = $0000
Watchdog      = 0.000000%
Clock frequ.  = 1,000,000 Hz
Time elapsed  = 30.0 us
Stop watch    = 30.0 us
Sleep share   = 0.000000%
```

- The time required for that loop can be read from the lines „Time Elapsed“ or „Stop watch“ in the simulation status window.
- It is
 - one clock cycle for LDI, plus
 - nine loop executions ending with the Z flag not set = one for DEC and two for BRNE, plus
 - the last loop with one for DEC and BRNE.
- Together 30 clock cycles have been elapsed.
- The formula is:
$$\text{Clock cycles} = 1 + 3 * (N - 1) + 2 = 3 * N$$

Instruction duration and flags

- The durations of the instructions can be read from three documents:
 - From the instruction set manual: [Link](#)
 - From the datasheet for each controller (Chapter Instruction Set Summary): [Link ATtiny24](#)

23. Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	$Rd \leftarrow Rd + Rr$	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,H	1
...					
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	Z,N,V	1
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	Z,N,V	1
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \cdot Rd$	Z,N,V	1
CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z,N,V	1
SER	Rd	Set Register	$Rd \leftarrow 0xFF$	None	1
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	$PC \leftarrow PC + k + 1$	None	2

- The instruction list [here](#) also provides this information.

Nested loops

- If you need longer times for loops: loops can be nested.

```
; Defining loop counts  
.equ cOuter = 200 ; the outer loop count  
.equ cInner = 100 ; the inner loop count  
;  
; Defining register names  
.def rOuter = R16 ; defining the register for the outer loop  
.def rInner = R17 ; defining the register for the inner loop  
;  
; Starting the outer loop  
LDI rOuter,cOuter ; Set register rOuter to the cOuter count value  
LoopOuter:  
LDI rInner,cInner ; Set register rInner to the cInner count value  
LoopInner:  
DEC rInner ; Decrease inner loop counter  
BRNE LoopInner ; Jump back to the inner label if not zero  
DEC rOuter ; Decrease outer loop counter  
BRNE LoopOuter ; Jump back to the outer label if not zero  
; Long loop done  
NOP ; For setting breakpoints
```


Assembler directives

- The source code now uses so-called directives:
 - All directives start with a dot character „.“.
 - They are meant for the assembler only and produce no executable code.
 - .EQU name = value defines a symbol constant with a name and sets it to the value given or calculated (calculation rules **here**).
 - .DEF name = Rn defines a symbol name for the register Rn (n = 0 .. 31).
- The advantage of using symbols are:
 - The names are easier to remember than register numbers (rInner and rOuter remind yourself of the purpose they are used for).
 - Changing the two constants or placing the registers somewhere else is easier than having to go through the complete source code.

Execution time of nested loop

- The outer loop executes one inner loop.
- So first the execution time of the inner loop is:

```
; Inner loop
    LDI rInner,cInner ; One clock cycle
LoopInner:
    DEC rInner ; One clock cycle
    BRNE LoopInner ; Two clock cycles when jumping, one if not
```

- The formula for the inner loop is:
$$\text{Clock cycles} = 1 + 3 * (cInner - 1) + 2 = 3 * cInner$$
- The outer loop is like that:

```
; Outer loop
    LDI rOuter,cOuter ; One clock cycle
LoopOuter:
    ; (Inner loop), 3 * cInner clock cycles
    DEC rOuter ; One clock cycle
    BRNE LoopOuter ; Two clock cycles when jumping, one if not
```

- Its formula is:
$$\text{Clock cycles} = 1 + (cOuter - 1) * (3 * cInner + 3) + 3 * cInner + 2$$

The nested loop, continued

- $\text{Clock cycles} = 1 + (\text{cOuter} - 1) * (3 * \text{cInner} + 3) + 3 * \text{cInner} + 2$
- $\text{Clock cycles} = 1 + \text{cOuter} * (3 * \text{cInner} + 3) - (3 * \text{cInner} + 3) + 3 * \text{cInner} + 2$
- $\text{Clock cycles} = 1 + 3 * \text{cOuter} * \text{cInner} + 3 * \text{cOuter} - 3 * \text{cInner} - 3 + 3 * \text{cInner} + 2$
- $\text{Clock cycles} = 1 + 3 * \text{cOuter} * \text{cInner} + 3 * \text{cOuter} - 1$
- $\text{Clock cycles} = 3 * \text{cOuter} * \text{cInner} + 3 * \text{cOuter}$
- For $\text{cOuter} = 200$ and $\text{cInner} = 100$:
 $\text{Clock cycles} = 3 * 200 * 100 + 3 * 200 = 60,600$
@1 MHz: 60.6 ms
- At maximum: $\text{cOuter} = 256$, $\text{cInner} = 256$
 $\text{Clock cycles} = 3 * 256 * 256 + 3 * 256 = 197,376$
@1 MHz: 197 ms
(Do not expect that `LDI rOuter,cOuter` assembles correct if you `.EQU cOuter=256`.
What else to correct that error?).
- With that, we can expand the two-stage nested loop to a three-stage nested loop and achieve the desired 500,000 clock cycles delay.

The next lecture: the seconds blink

- Now with one ATtiny24 (14-pin) we have constructed the following:
 - An oscillator with 1 MHz (with the internal RC oscillator of 8 MHz, by default divided by 8),
 - Two 8 bit counters counting down from selectable initial values down to zero.
- One ATtiny24 therefore replaces the following CMOS ICs here:
 - one 4093 as oscillator, 14 pins,
 - four 4516 4-bit presetable binary down-counters, $4 * 16 = 64$ pins.
- A good argument why a controller is the better choice. And:
- Why learning assembler is so much better than C: you'll keep anything under your own control and no compiler interferes with your calculations and places unknown and unnecessary instructions into your execution code.

Questions and Tasks for Lecture 3

Question 3-1: What happens if your program does not jump back

a) in the simulator, or

b) inside the controller?

(Hint for b: The flash memory CANNOT be empty, it is always 0xFFFF if cells are not programmed.)

Bonus question: What makes the difference between a hardware reset to 0x0000 and a simple jump to address 0x0000? (Hint: Try to simulate that by setting some register values in between. A hardware reset in the simulator is forced with the Restart menu entry! A jump to 0x0000 can be done with RJMP 0).

Questions and Tasks for Lecture 3, Continued

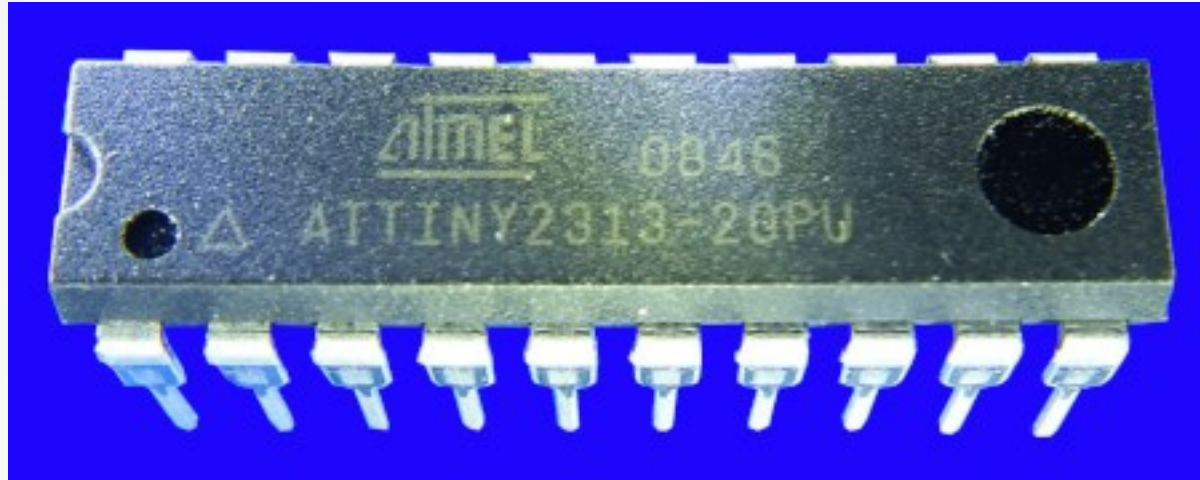
Task 3-2: Go through the instruction set summary and find out if there are INC and DEC instructions for 16 bits. What are their limitations or use conditions?

Questions and Tasks for Lecture 3, Continued

Task 3-3: Find the differences between DEC R16 and SUBI R16,1 in respect to

- a) their execution time, and**
- b) the affected flags in the status register.**

Bonus question: There is no ADDI instruction listed in the instruction set. How can you then increase a register's content with the same flag settings like in SUBI? (Try to simulate your solution and see if the flags Z and C are set/cleared correct.)



Lecture 4: The second blinker

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

The seconds blinker

- From the lecture 3 we know that blinking a LED in one second rhythm requires a 24 bit down counter.
- Blinking would require the following:
 - A LED attached to one of the pins of the ATtiny24 (preferably PB0) via a current-limiting resistor, either to GND (sourcing current) or to the operating voltage (sinking current).
 - The portpin has to be set as output pin.
 - The output has to be set (HIGH). Then a period of 500,000 clock cycles has to be absolved.
 - The output has to be cleared (LOW). Again, a period of 500,000 clock cycles has to be absolved.
 - The program then has to return to the HIGH-setting code.
- That implies the following: the delay period has to be executed two times. This can be resolved in three ways:
 1. We can write identical delay code in between the HIGH and the LOW period. This is the most simple way to solve this.
 2. We can write the code once, store it as macro code and place the macro call twice between the two I/O instructions.
 3. We can write the code once, store it as a subroutine and call this same code twice.
- Method 1 is trivial. The latter two methods are demonstrated here.
- Finally we learn how the assembled code is written to the controller.

The 24-bit loop

- The source code for the 24-bit down-count loop is of course as follows:

```
; Defining loop counts for 499,998 cycles
.equ cOuter = 15 ; the outer loop count
.equ cMiddle = 55 ; the middle loop count
.equ cInner = 201 ; the inner loop count
; Defining register names
.def rOuter = R16 ; defining the register for the outer loop
.def rMiddle = R17 ; defining the register for the middle loop
.def rInner = R18 ; defining the register for the inner loop
; Starting the outer loop
    LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
    LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value
LoopMiddle:
    LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
    DEC rInner ; Decrease inner loop counter
    BRNE LoopInner ; Jump back to the inner label if not zero
    DEC rMiddle ; Decrease middle loop counter
    BRNE LoopMiddle ; Jump back to the middle label if not zero
    DEC rOuter ; Decrease outer loop counter
    BRNE LoopOuter ; Jump back to the outer label if not zero
; Very long loop done
```

Execution times

- The clock cycles of the inner loop are again $CC_i = 3 * c_{Inner}$.
- The clock cycles of the middle loop are:
 $CC_m = 1 + (c_{Middle} - 1) * (CC_i + 3) + CC_i + 2$
 $CC_m = 1 + c_{Middle} * CC_i + 3 * c_{Middle} - CC_i - 3 + CC_i + 2$
 $CC_m = c_{Middle} * CC_i + 3 * c_{Middle}$
 $CC_m = 3 * c_{Middle} * c_{Inner} + 3 * c_{Middle}$
- The clock cycles of the outer loop are:
 $CC_o = 1 + (c_{Outer} - 1) * (CC_m + 3) + CC_m + 2$
 $CC_o = 1 + c_{Outer} * CC_m + 3 * c_{Outer} - CC_m - 3 + CC_m + 2$
 $CC_o = c_{Outer} * CC_m + 3 * c_{Outer}$
 $CC_o = c_{Outer} * (3 * c_{Middle} * c_{Inner} + 3 * c_{Middle}) + 3 * c_{Outer}$
 $CC_o = 3 * c_{Outer} * c_{Middle} * c_{Inner} + 3 * c_{Outer} * c_{Middle} + 3 * c_{Outer}$
 $CC_o = 3 * c_{Outer} * (c_{Middle} * c_{Inner} + c_{Middle} + 1)$
- An optimal combination to achieve around 499,998 clock cycles is:
 $c_{Outer} = 15 ; c_{Middle} = 55 ; c_{Inner} = 201 ; CC_o = 499,995$

Delay loop as macro

- The following makes a macro for the loop code:

```
.macro delay
; Starting the outer loop
    LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
    LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value
LoopMiddle:
    LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
    DEC rInner ; Decrease inner loop counter
    BRNE LoopInner ; Jump back to the inner label if not zero
    DEC rMiddle ; Decrease middle loop counter
    BRNE LoopMiddle ; Jump back to the middle label if not zero
    DEC rOuter ; Decrease outer loop counter
    BRNE LoopOuter ; Jump back to the outer label if not zero
; Very long loop done
.endmacro
```

- When the assembler sees the directive `.macro` he grabs the following text until a line saying `„.endmacro“` or `„.endm“` occurs.
- If the macro's name (`„delay“`) is found instead of a mnemonic this text is inserted into the source code as if it were placed there – and assembled.

Calling the macro

- **Calling the macro is simple:**

```
; Defining loop counts for 499,998 cycles
.equ cOuter = 15 ; the outer loop count
.equ cMiddle = 55 ; the middle loop count
.equ cInner = 201 ; the inner loop count
; Defining register names
.def rOuter = R16 ; defining the register for the outer loop
.def rMiddle = R17 ; defining the register for the middle loop
.def rInner = R18 ; defining the register for the inner loop
; (Place the macro delay here)
    sbi DDRB,PORTB0 ; PB0 as output
LedLoop:
    sbi PORTB,PORTB0 ; Output pin high
    DELAY ; Insert the macro content here
    cbi PORTB,PORTB0 ; Output pin low
    DELAY ; Insert the macro content the second time
    RJMP LedLoop
```

- **Note that the macro code is inserted twice, which consumes flash memory (of which the ATtiny24 has plenty of).**
- **Unfortunately avr_sim cannot set and handle breakpoints inside macros, so you cannot measure execution times separately.**

Where to get cOuter etc. from?

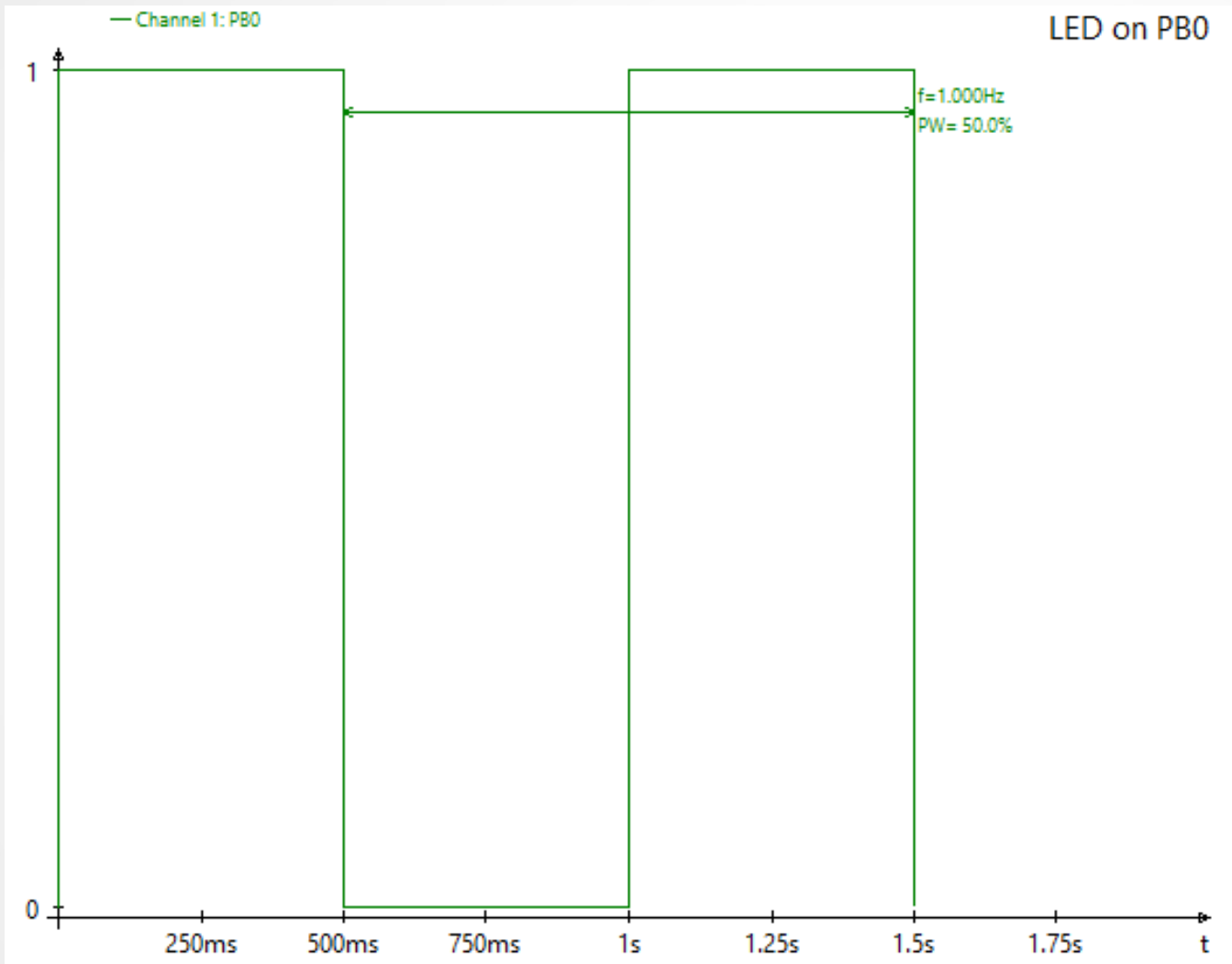
- **Where to get those constants from? I used MS-Excel and some VBA code:**

```
Private Sub CommandButton1_Click()  
Dim cOut As Long, cMid As Long, cInn As Long  
Dim Should As Long, Value As Long, Delta As Long, DeltaMin As Long  
Dim Out As Long, Mid As Long, Inn As Long, Val As Long  
Should = Cells(3, 1)  
DeltaMin = 9999999  
For cOut = 0 To 255  
    For cMid = 0 To 255  
        For cInn = 0 To 255  
            Value = 3 * cOut * cMid * cInn + 3 * cOut * cMid + 3 * cOut  
            Delta = Abs(Value - Should)  
            If Delta < DeltaMin Then  
                Val = Value  
                DeltaMin = Delta  
                Out = cOut  
                Mid = cMid  
                Inn = cInn  
            End If  
        Next cInn  
    Next cMid  
Next cOut  
Cells(3, 2) = Out  
Cells(3, 3) = Mid  
Cells(3, 4) = Inn  
Cells(3, 5) = DeltaMin  
Cells(3, 6) = Val  
End Sub
```

- Note that there are many more combinations that yield approximately the target value.
- 68 combinations are within a distance of +/- 4 to 500,000.
- 84 are within a distance of +/- 5.

The second blinker – as simulated

- This is what you get from the source code within the simulator:



- A rather exact signal frequency of 1.000 Hz - But in reality?
- The internal RC oscillator has $\pm 3\%$ accuracy at 3.3 Volt operating voltage, but $\pm 10\%$ at 5 Volt.
- So your real controller will produce something in between 0.9 and 1.1 Hz by default.
- The difference cannot be seen, but rather can be measured.
- If you need it more exact: clock your ATtiny24 with a crystal or a crystal oscillator – see the next pages.

Subroutines

- The second method of using the same source code twice (or more times) and writing it once is to formulate it as subroutine. Calling it is simple:

```
; Calling a subroutine  
RCALL MySub  
; Continue here if MySub has finished execution
```

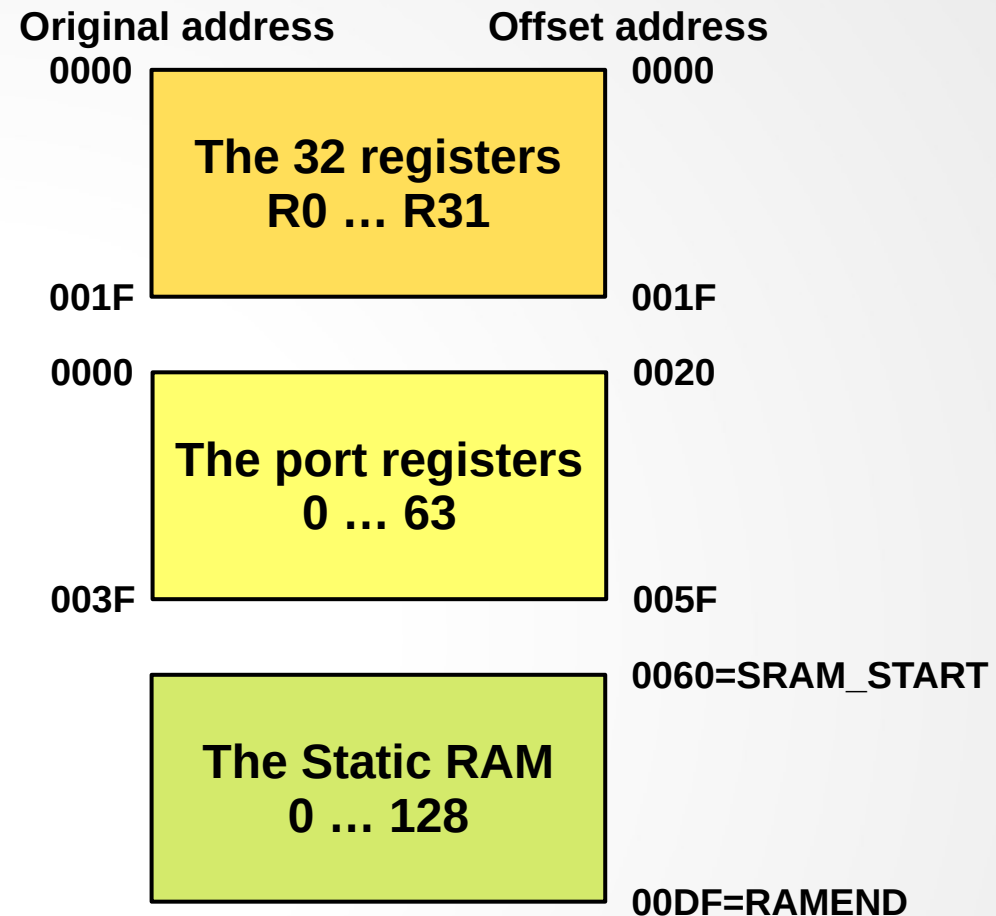
- Subroutines start with a label (where they can be called) and end with RET (for RETurn):

```
; My subroutine  
MySub: ; Label for calling the subroutine  
; The subroutine does something here  
RET ; Control goes back where it was called from
```

- The subroutine has to know where it was called from, it's return address. The place where this is stored is called the „Stack“.
- The stack is a memory space at the end of the Static RAM (SRAM) of the controller. The def.inc file provides a constant called „RAMEND“ for this. In the ATtiny24 RAMEND is 0x00DF (decimal 223).

Memory organization of AVR

- This demonstrates how the memory space in AVR is organized in an ATtiny24.
- Not shown are
 - the flash memory where the binary code is stored, starting at 0000,
 - the EEPROM memory where durable data can be stored, also starting at 0000.
- The other memories are stapled:
 - The 32 registers from 0000 to 001F appear on their original addresses.
 - The 64 port registers, originally addressed from 0000 to 003F, appear at 0020 etc.
 - The static RAM starts at SRAM_START and ends at RAMEND.



- This organization allows to access all memory components of the AVR at any time and from every binary code location. Memory is completely unprotected, and it is up to the program writer to preserve needed information from being changed in an uncontrolled way.

Initiating the stack

- Initiating the stack should be the first thing on top of the executable code.
- It differs a little bit depending from the size of SRAM memory that the device has:
 - Devices with 256 bytes SRAM or less (like the ATtiny24) have one port register for the stack pointer. It is named SPL (for Stack Pointer Low).

; Init stack for devices with small SRAM

LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16

OUT SPL,R16 ; Write R16 to the LSB of the stack pointer

- Devices with more than 256 bytes STAM have an additional port register named SPH (H for High). It holds the Most Significant Byte (MSB) of the stack pointer.

; Init stack for devices with large SRAM

LDI R16, High(RAMEND) ; Load the MSB of RAMEND to R16

OUT SPH,R16 ; Write R16 to the MSB of the stack pointer

LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16

OUT SPL,R16 ; Write R16 to the LSB of the stack pointer

- From now on the stack is ready for use, and calling addresses can be stored there. The whole program for 0.5-seceonds-delay-counting looks like this:

Use of the stack for address storage

```
; Init the stack for the ATtiny24 device  
    LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16  
    OUT SPL,R16 ; Write R16 to the LSB of the stack pointer  
; Make PB0 output  
    SBI DDRB, DDB0 ; Portpin PB0 as output  
; The second loop blinks  
Second: ; Label for jump back  
    SBI PORTB, PORTB0 ; Make port pin PB0 high  
    RCALL Delay ; Call the subroutine DELAY  
    CBI PORTB, PORTB0 ; Make port pin PB0 low  
    RCALL Delay ; Again call the subroutine DELAY  
    RJMP Second ; Restart the second loop  
; (Continued on the next page)
```

- The difference to the macro code above are the two instructions **RCALL**. This is a single-word instruction, but consumes three clock cycles.

- It does the following:
 - It throws the LSB of the program counter PC onto the current stack position and decreases the Stack Pointer by one.
 - Then it throws the MSB of the PC onto this next stack position and again decreases the Stack Pointer by one.
 - It loads the relative address of the subroutine „Delay“ onto the PC.
 - The next instruction executed is the first instruction of the subroutine.

The subroutine

```
; The subroutine DELAY  
Delay: ; Label for calling the subroutine  
; Starting the outer loop  
    LDI rOuter,cOuter ; Set register rOuter to the cOuter count value  
LoopOuter:  
    LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value  
LoopMiddle:  
    LDI rInner,cInner ; Set register rInner to the cInner count value  
LoopInner:  
    DEC rInner ; Decrease inner loop counter  
    BRNE LoopInner ; Jump back to the inner label if not zero  
    DEC rMiddle ; Decrease middle loop counter  
    BRNE LoopMiddle ; Jump back to the middle label if not zero  
    DEC rOuter ; Decrease outer loop counter  
    BRNE LoopOuter ; Jump back to the outer label if not zero  
; Very long loop done  
    RET ; RETurn to the caller
```

- Even though this looks pretty much like the macro from above, it is working completely different:
 - It ends with a RET instruction.
 - It is only once coded (and not twice like the macro).

Demonstration with the simulator

- Assemble the program and start simulation.
- In the View hardware section click on „SRAM“.
- Click „Step“ twice and see the changed stackpointer entry in the Status window.
- Then click „Step“ twice again and the cursor is now on the RCALL.
- With the next „Step“ three things happen:
 1. The highest two bytes of the SRAM display the calling address, that has been thrown onto the stack.
 2. The stackpointer in the Status window is now by 2 positions below its initial value.
 3. The next executable instruction is LDI rOuter, cOuter in the delay subroutine.
- In order to not having to wait a half second, click „Skip“ until the RET instruction is executed next.
- When RET is executed, the following happens:
 1. The PC is loaded with the address on top of the stack, the next executable instruction is the one that follows the first RCALL instruction.
 2. The stackpointer is at its initial value.
 3. The SRAM content remains unchanged.
- Note that the RET consumes four clock cycles, so don't write subroutines just for fun, if your program has to perform very fast.

Conclusions

- **There are always many different modes that a task can be resolved with.**
- **As with all programming the arousing twist is optimization: does the source code what is required (functionality and correctness), is it easy to understand (readability, understandability) and is it nice and aesthetic (elegancy)? (Spaghetti code is always the opposite of the latter two, even it works correct.)**
- **Assembler offers much more optimization opportunities than Higher-Level languages because of its „Anything is allowed, be your own master“ attitude.**
- **The counter-side of this freedom is:**
 - 1.You can easily produce spaghetti code, no built-in conventions or safety functions hinder you.**
 - 2.You are solely responsible for where you place data and variables to and to protect those. No mechanism warns you, if you overwrite data needed later on.**

Questions and tasks in Lecture 4

Task 4-1: Write a program that counts three registers up until they reach a target value. Use the assembler functions BYTE1, BYTE2 and BYTE3 to derive the three bytes from the constant:

```
; Deriving single bytes from a constant  
.equ constant = 1234567 ; = hexadecimal 0x12D687  
.equ b1 = BYTE1(constant) ; Yields 0x87  
.equ b2 = BYTE2(constant) ; Yields 0xD6  
.equ b3 = BYTE3(constant) ; Yields 0x12
```

Use the instruction CPI (ComPare with Immediate) and the Z flag to find out, if the value has been reached. Try if you can derive a formula for the clock cycles and verify with the simulator if your formula is working correct and exact.

(Note: This task is solvable!)

Questions and tasks in Lecture 4 - Continued

Task 4-2: Try to change the code so that the three registers down-count the initial constant value.

Hint: Do not use the instruction DEC (for DECrease) because it does not affect the overflow flag C (for Carry). Use the instructions SUBI (for SUBtract Immediate) and BRCC (BRanch on Carry flag Clear) for conditional jumps.

Try to derive the formula for the number of clock cycles.

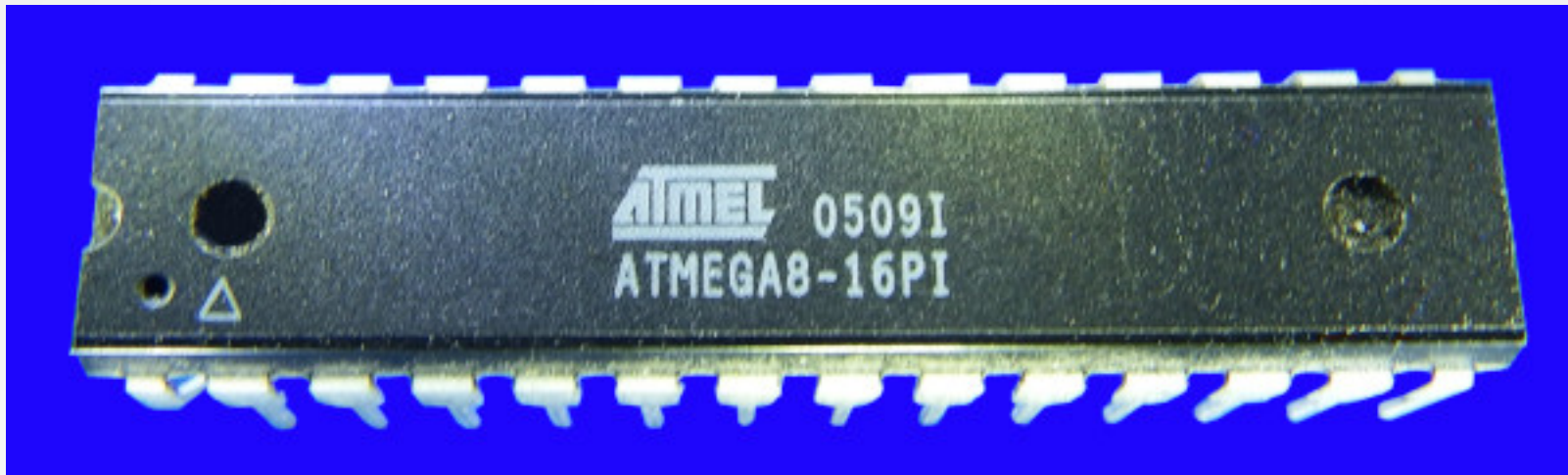
Another hint: The formula to derive the consumed clock cycles for that might get a little complicated, so give up if unsuccessful for longer than two hours. Genius approaches are welcome!

Questions and tasks in Lecture 4 - Continued

Task 4-3: If your crystal (next lecture) would have a frequency of 4 MHz, what would be the constants for

- a) cOuter, cMiddle and cInner for the method first introduced here,**
- b) for the up-counting, and**
- c) for the down-counting.**

Try verification with avr_sim, by inputting the clock frequency and disabling the CLKDIV8 fuse (but use a very fast computer for that, place breakpoints, let avr_sim work over night and disable the operating system's sleep features!)



Lecture 5: Practical application

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

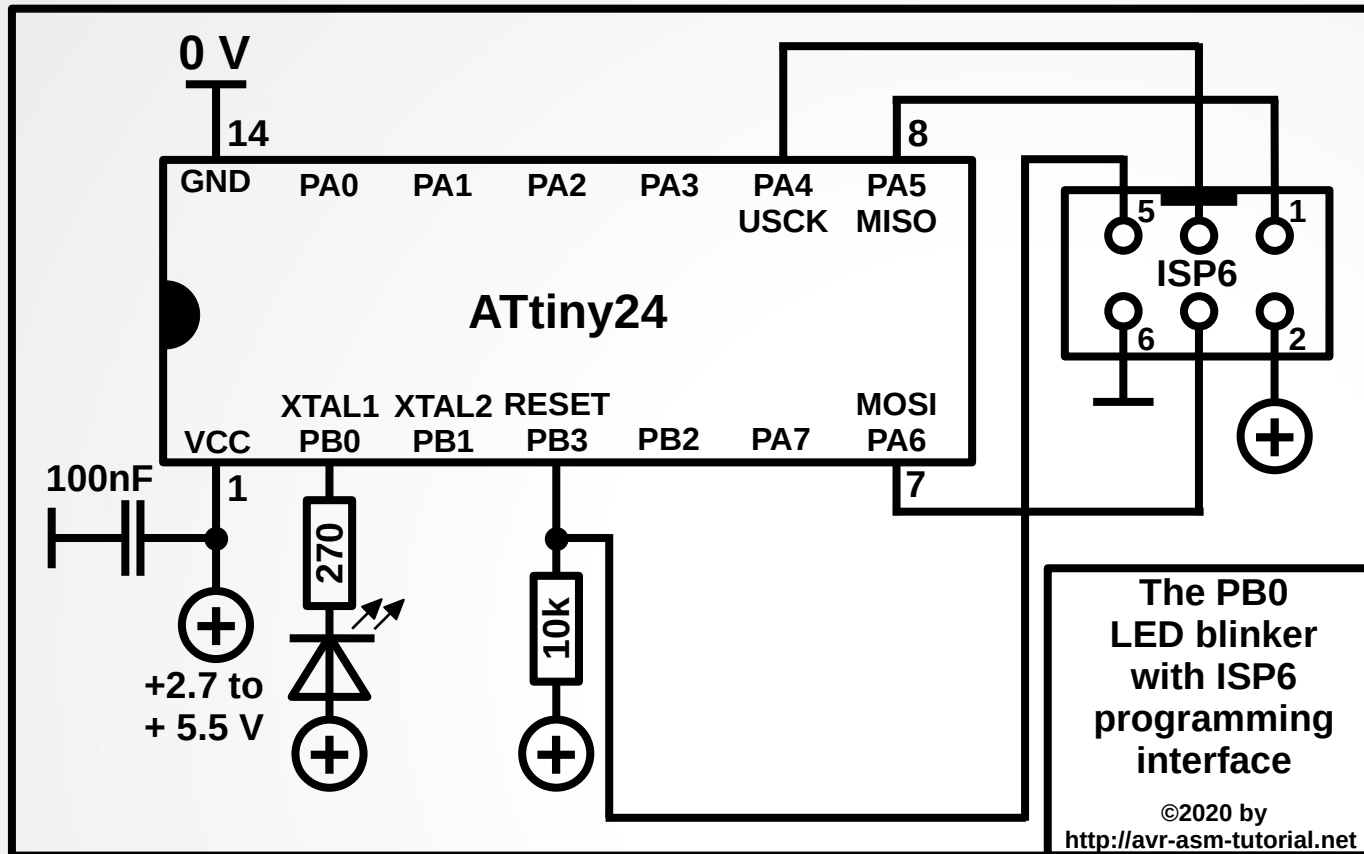
The ISP programming interface

- **Writing binary code to the controller requires access to its internals.**
- **A convenient method to gain access is the In-System-Programming interface of all AVR.**
- **ISP (In-System-Programming) works via a standardized 6-pin plug. Via this the programmer hardware can access the internals.**
- **A second programming method works with +12 V on the RESET pin (High-voltage serial programming, HV programming). This is not used here.**

Practical applications

- **With this lecture you will learn how the controller is programmed and practically does for what your source code has taught him to do.**
- **You will need**
 - a) **a hardware programmer (I use a Diamex ProgS2, but there are several others in the market),**
 - b) **a program that can write hex files to the programmer (I use ATMEL's Studio 4.19, an alternative is AvrDude),**
 - c) **some electronic components (a breadboard and wires, a 14-pin IC socket, a 6-pin box connector, two 6-pin IDC sockets plus 10 cm 6-pole flat cable, a resistor of 10 k Ω , a resistor of 270 Ω , a red standard LED and a 100nF ceramic capacitor).**
- **First get familiar with the breadboard and its internal wiring.**

The schematic of the PB0 blinker



- This is the schematic.
- The ATtiny24 is shown from the top.
- The ceramic capacitor on pin 1 suppresses rectangles from the CMOS internals on the +5V line.
- The LED is connected as sink load with a current-limiting resistor.
- The 10kΩ ties the RESET pin to +5V.

- The In-System-Programming interface ISP6 is connected with USCK, MISO, MOSI, RESET and the operating voltage.
- The programmer delivers +5V, if so enabled. If not: connect a 4.5V or 3.7V batterie to the GND- and +-connections.

The ISP programming interface

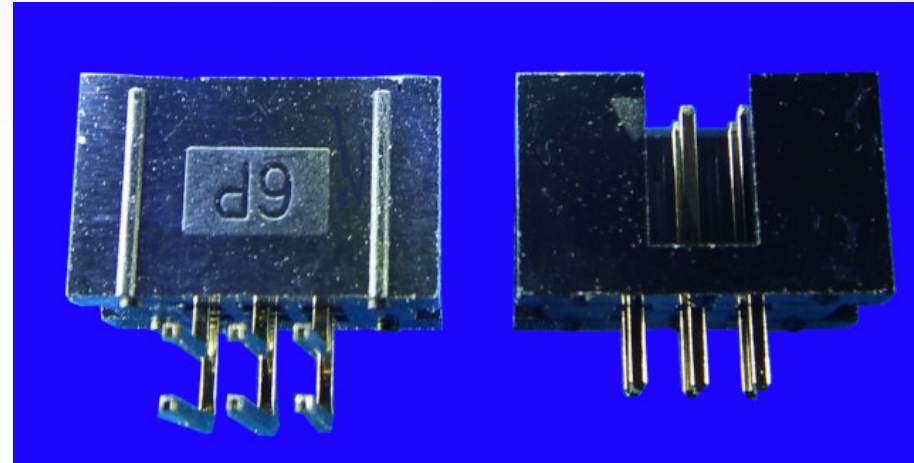
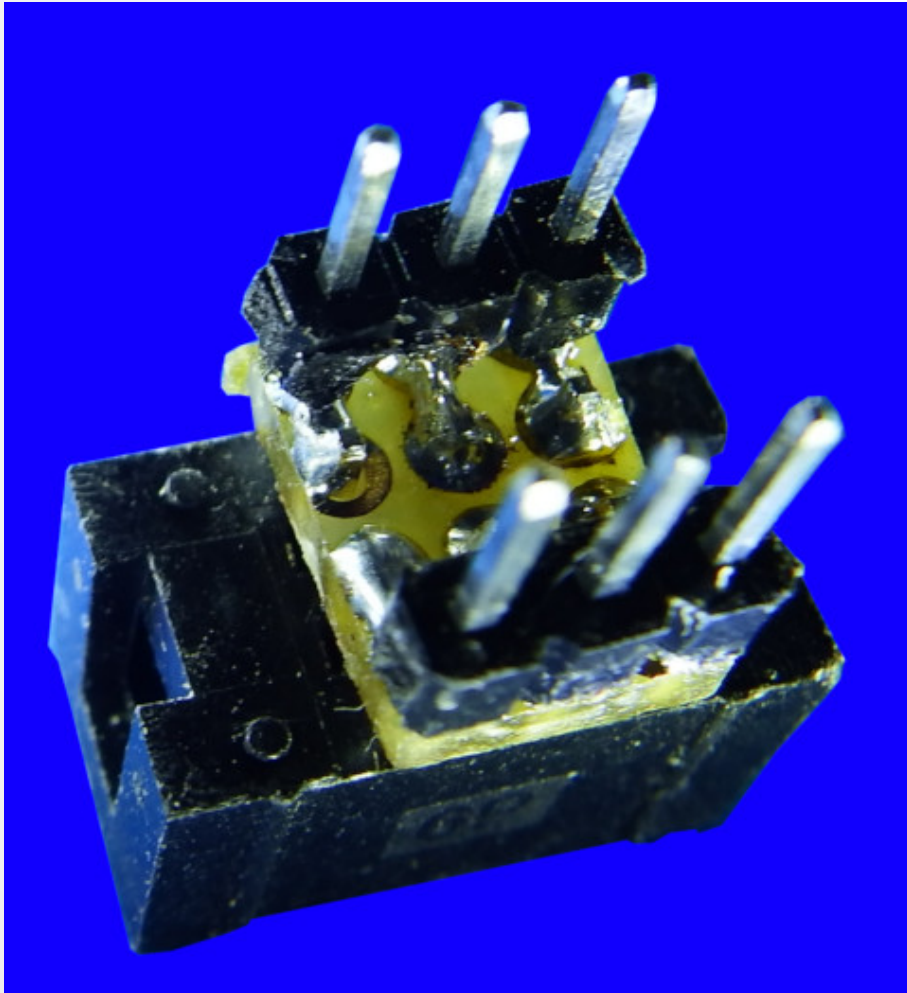
- **Via a 6-pin plug the following signals allow access with programmer hardware:**
 1. **The RESET pin (pin 4 of the ATtiny24, pin 5 of the ISP6 plug) is tied LOW, which initiates the serial communication between the programmer and the controller.**
 2. **The SCK or USCK pin (pin 8 of the ATtiny24, pin 3 of the ISP6) clocks the device's communication.**
 3. **MOSI (pin 7 of the ATtiny24, pin 4 of the ISP6) outputs serial data (from the programmer to the device).**
 4. **MISO (pin 8 of the ATtiny24, pin 1 of the ISP6) inputs serial data (from the device to the programmer).**
- **When programming is completed, the RESET pin is disconnected, the 10k pulls it high and the controller immediately starts at address 0000.**

Advantages of ISP

- **The device can be programmed with the standard operating voltages, no extra voltage supplies are necessary.**
- **The device can remain in its place, no removal or disassembling of the device is necessary.**
- **Via the 6-pin ISP plug the whole system can be sourced (e.g. with the 5V from the USB plug), if the programmer enables that.**
- **Quick and reliable re-programming of the controller can be achieved with very low effort and nearly no extra hardware.**

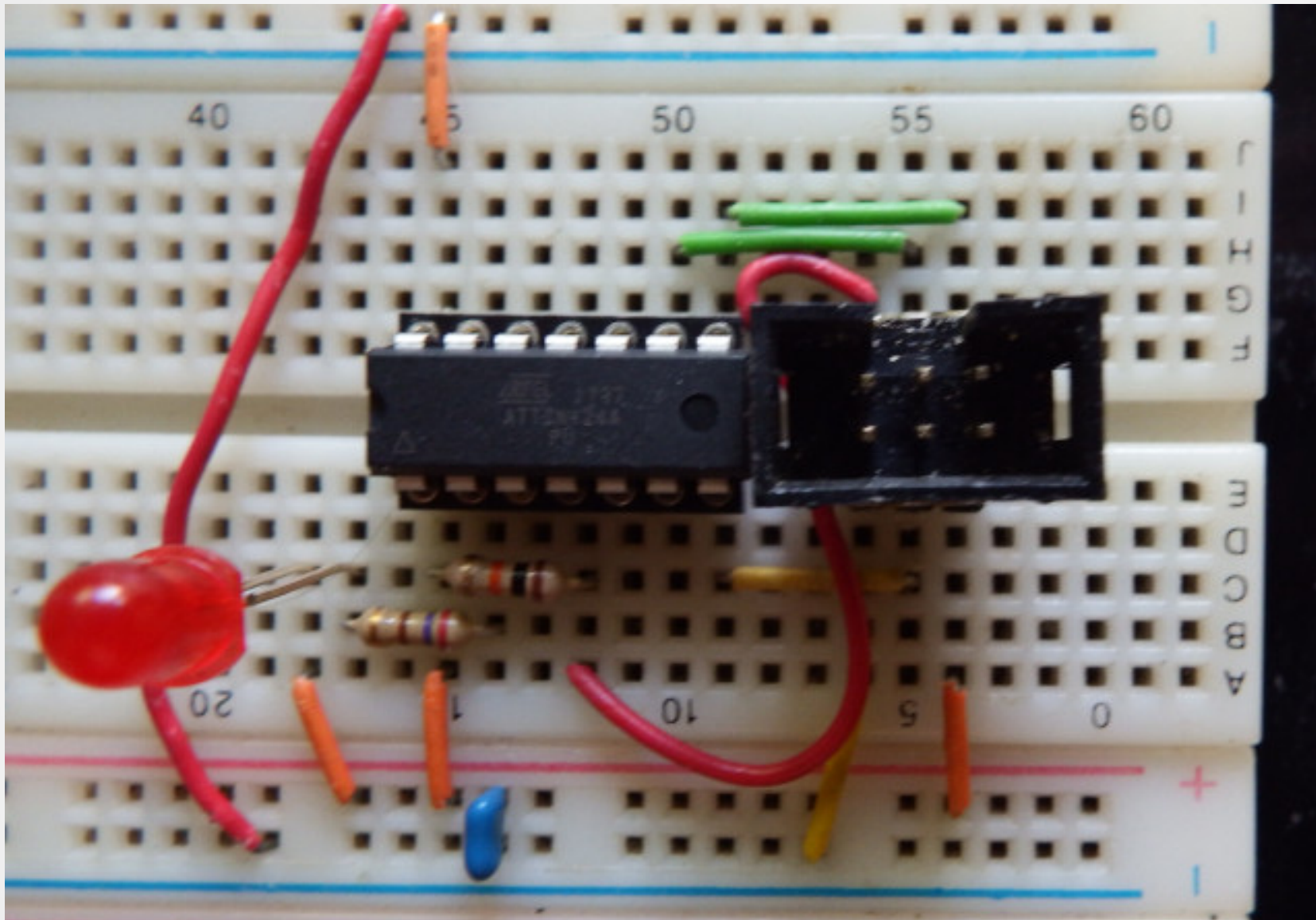
The ISP6 connector

- These are the 6-pin box connectors. We need only a straight one.



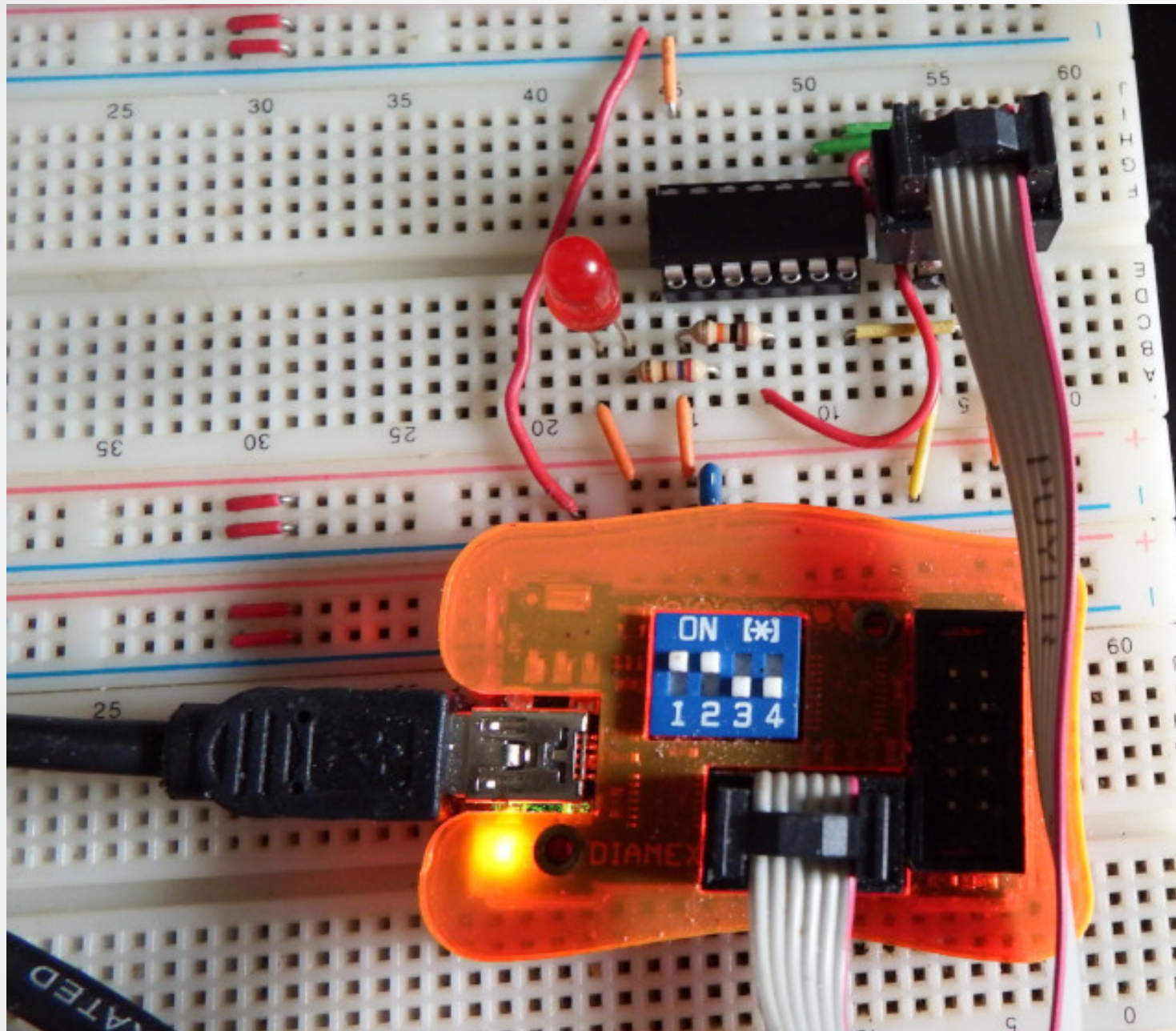
- In order to fit this to the breadboard we need to solder this to a small 12-hole PCB and add two 3-pin plugs, like shown here.

The breadboard programmer



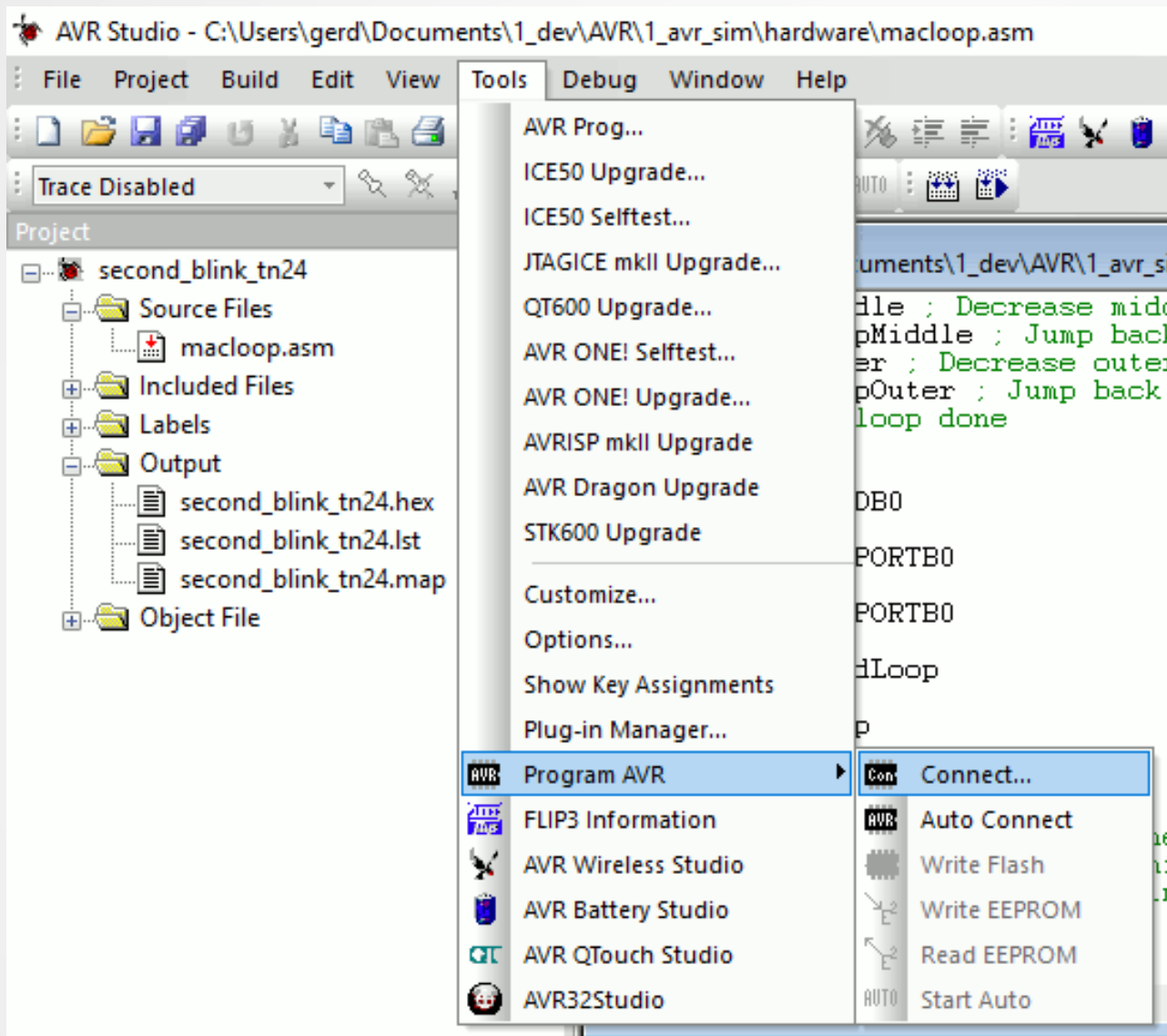
- This is all on the bread-board.
- The upper and lower (-) line are to be connected.
- The ISP6 delivers the operating voltage.

Connecting the programmer



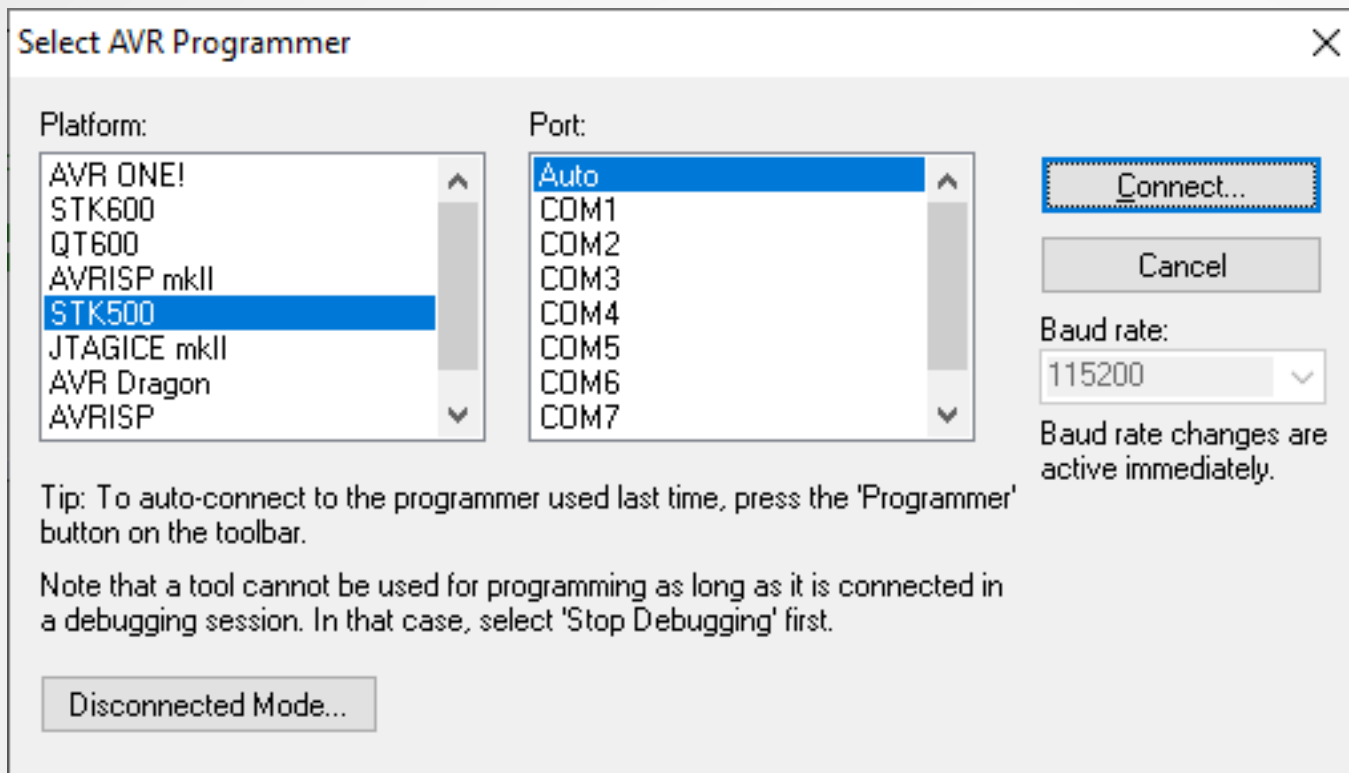
- The programmer (here a Diamex ProgS2) is connected with a 6-pin flat cable with the ISP6 plug.
- This allows programming of the ATtiny24 on the board.
- The two switches provide the 5V through the ISP6 line.

Programming with the Studio



- If you use the Studio 4.19 (like shown here) you'll have to download it, to install it and to install the Jungo USB driver from ATMEL Norway (under Win10 the one for Studio 7, not the one that comes with Studio 4.19, disable driver signatures).
- In the Tools menu select Program AVR and connect to your programmer.

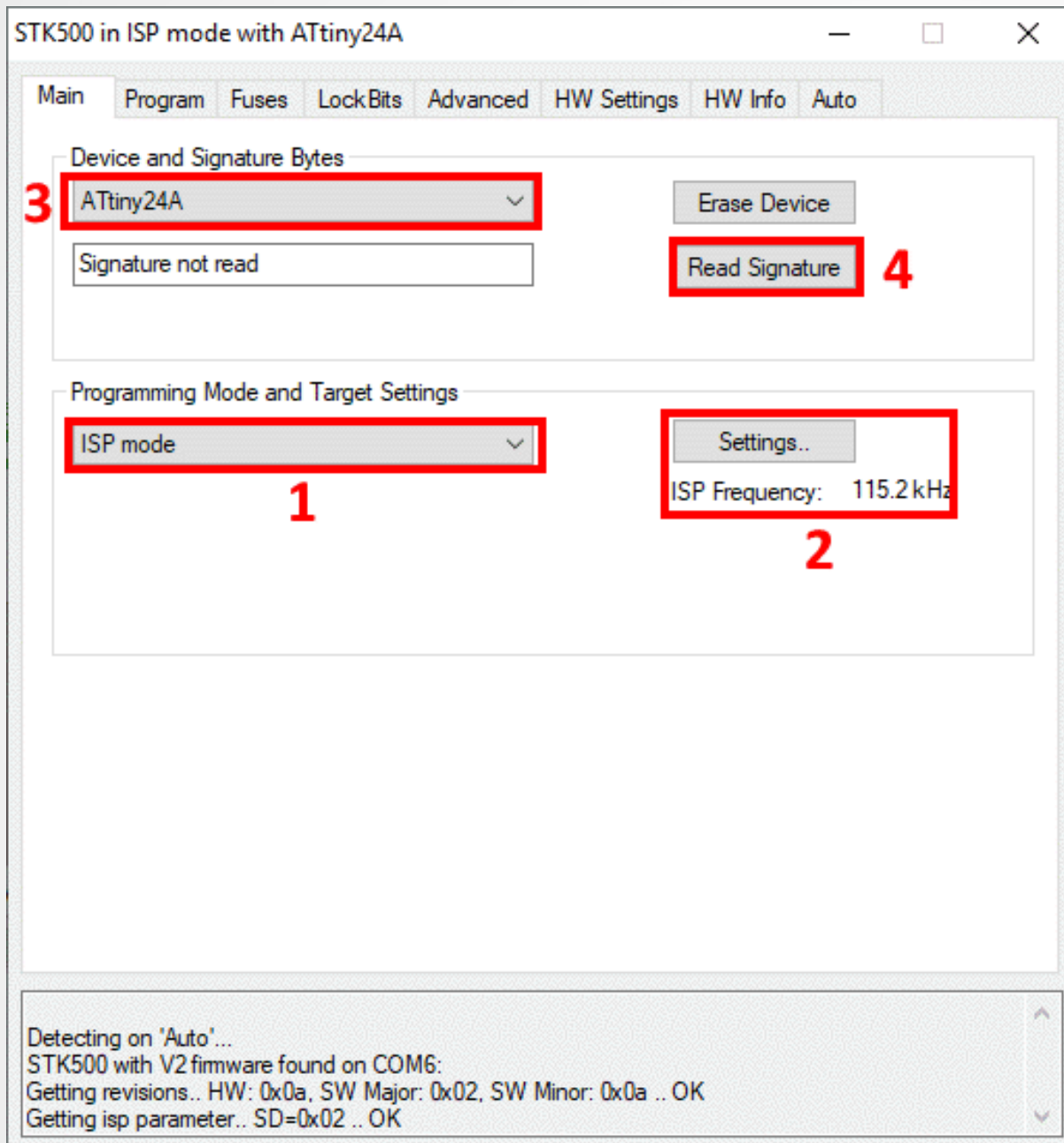
Identifying the programmer



- Select the mode that the programmer is working with.
- Most programmers work like an STK 500 board and have a USB-to-Serial converter.

- Select the mode that the programmer is working with.
- Most programmers work like an STK 500 board and have a USB-to-Serial converter.
- If your Windows installs the Serial converter to a COM interface beyond COM8, first replace it by a lower COM number.

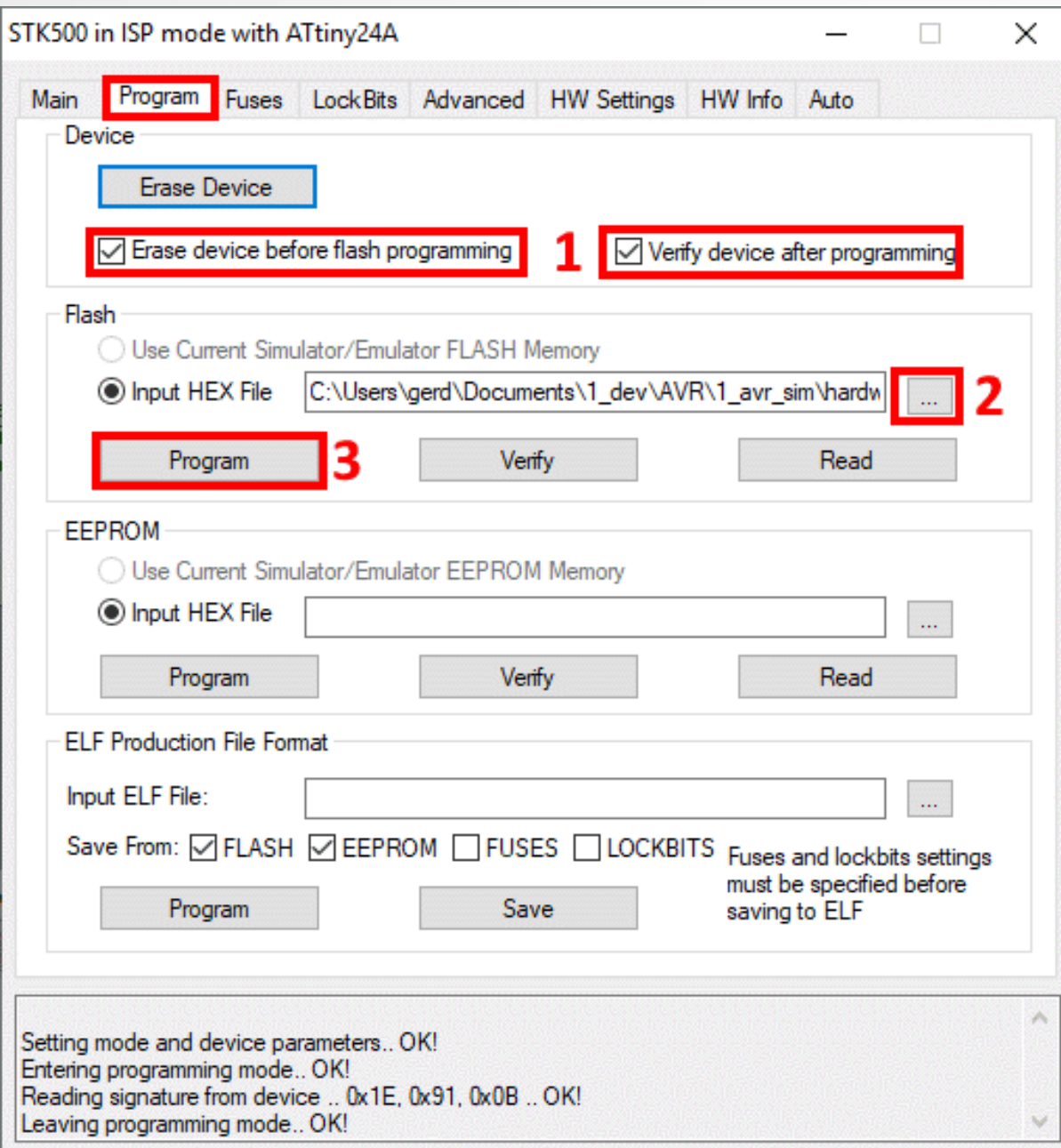
Setting ISP mode



- In the Main tab, make sure that the ISP mode is selected. If not select this from the dropdown list.
- Next ensure that the ISP frequency is lower than $\frac{1}{4}$ of the device's clock frequency (1 MHz). If not click Settings and select another frequency.
- Next set the device type and select it from the dropdown list.
- At last click Read Signature. The result should match.

Programming the device

- Then move to the **Program** tab.
- First make sure that the two settings are correct.
- Then select the hex file that holds your program (this can either be produced by the Studio with the menu Build or with avr_sim).
- If correct, press the **Program** button and view the messages on the window. It should display correct verification.



The LED is blinking

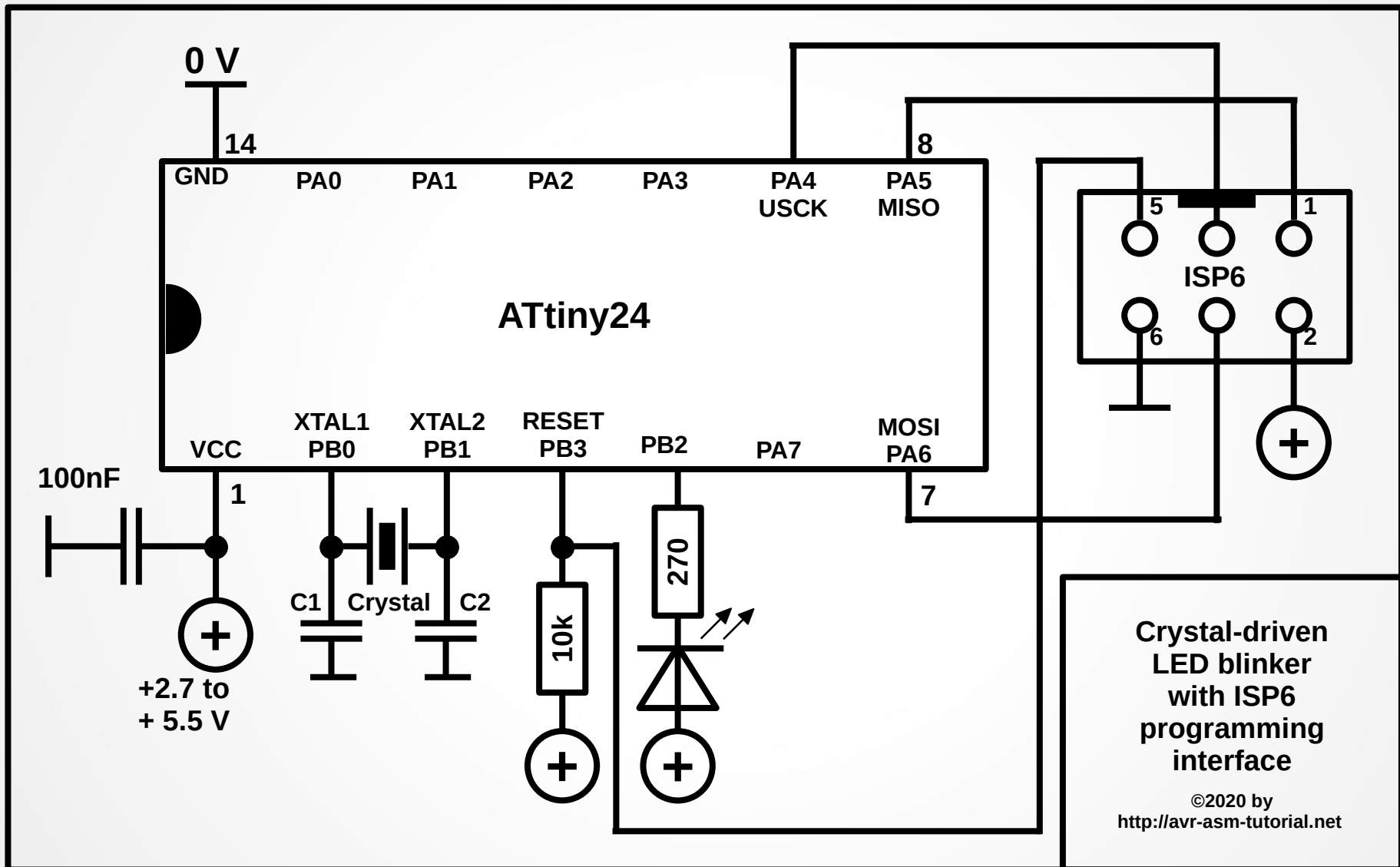
- **Immediately after the verification ended, the LED should blink in a seconds rhythm.**
- **If not: there are many different errors that can be made, such as:**
 - **false wiring of the ISP6 plug (the programmer reports errors when trying to connect),**
 - **the ATtiny24 was installed in reverse direction (the USB of the computer reports over-current),**
 - **missing operating voltage or out of voltage range,**
 - **a defective LED.**

A high-precision LED blinker

- **The internal RC oscillator in the ATtiny24 is not very reliable: it has an error rate of 3% and more.**
- **If you need the seconds signal to be more exact, you'll have to clock the controller with a crystal.**
- **The following shows how to do that and teaches you about the fuses of the controller.**

Exact clocking – with a crystal

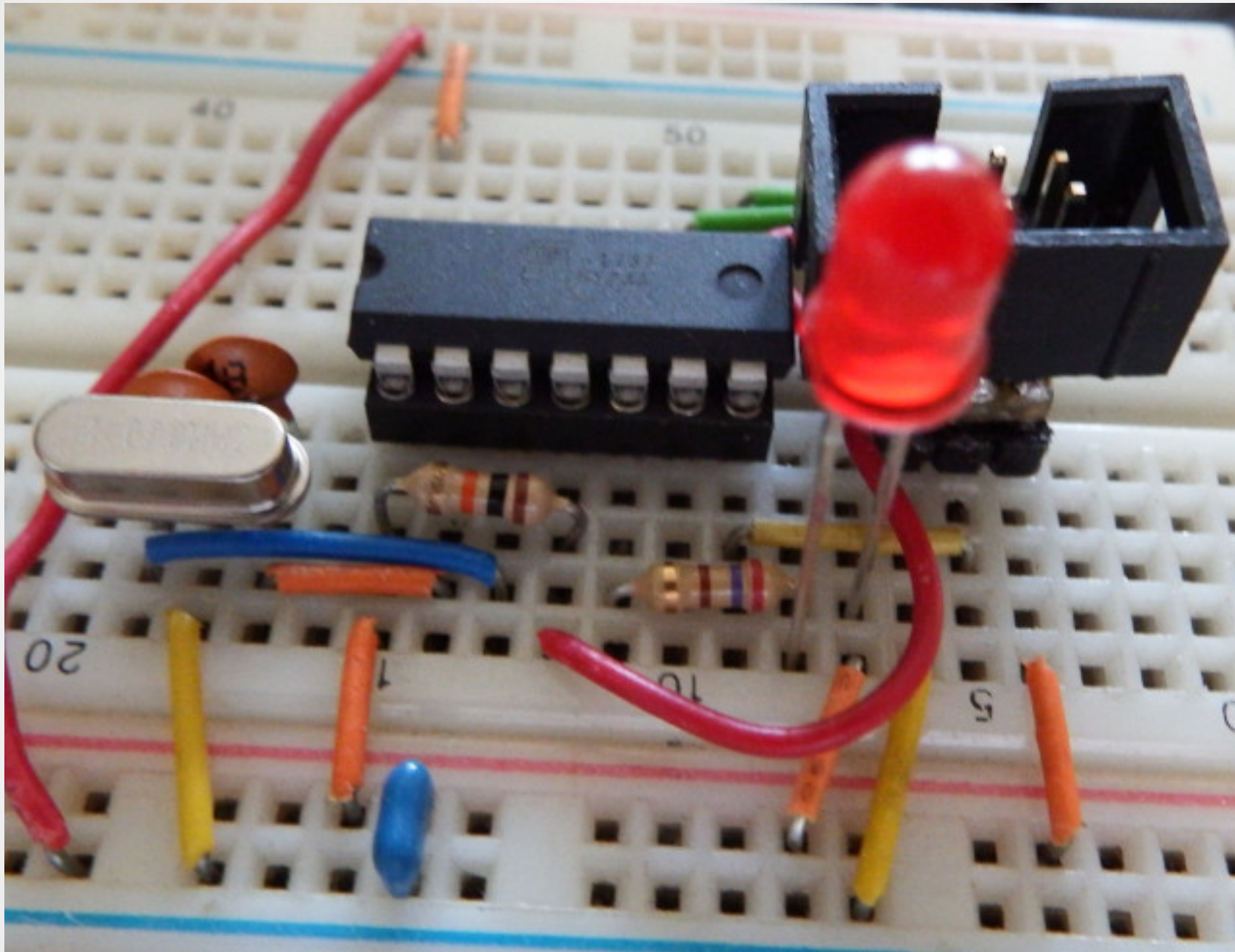
- This is the crystal-driven ATtiny24:



Some hints on the schematic

- A general rule: each pin of a controller can serve multiple functions. Not GND and VCC, but most others.
- An example: PB3 serves by default as RESET input and is required to program the flash memory of the chip and to change internal fuses via ISP.
- Here, PB0 and PB1 change their function: they are part of an internal crystal oscillator. XTAL1 is input and XTAL2 is output. But: PB0 and PB1 can not be used any more (so the LED is relocated to PB2).
- The generated oscillator signal then clocks the ATtiny24. It runs as exact as the crystal is (for a 4 MHz crystal: 20 to 30 ppm = 4 MHz +/- 80 to 120 Hz).
- This change can be (and has to be) done by changing fuses: internal switches that change the hardware of the controller.

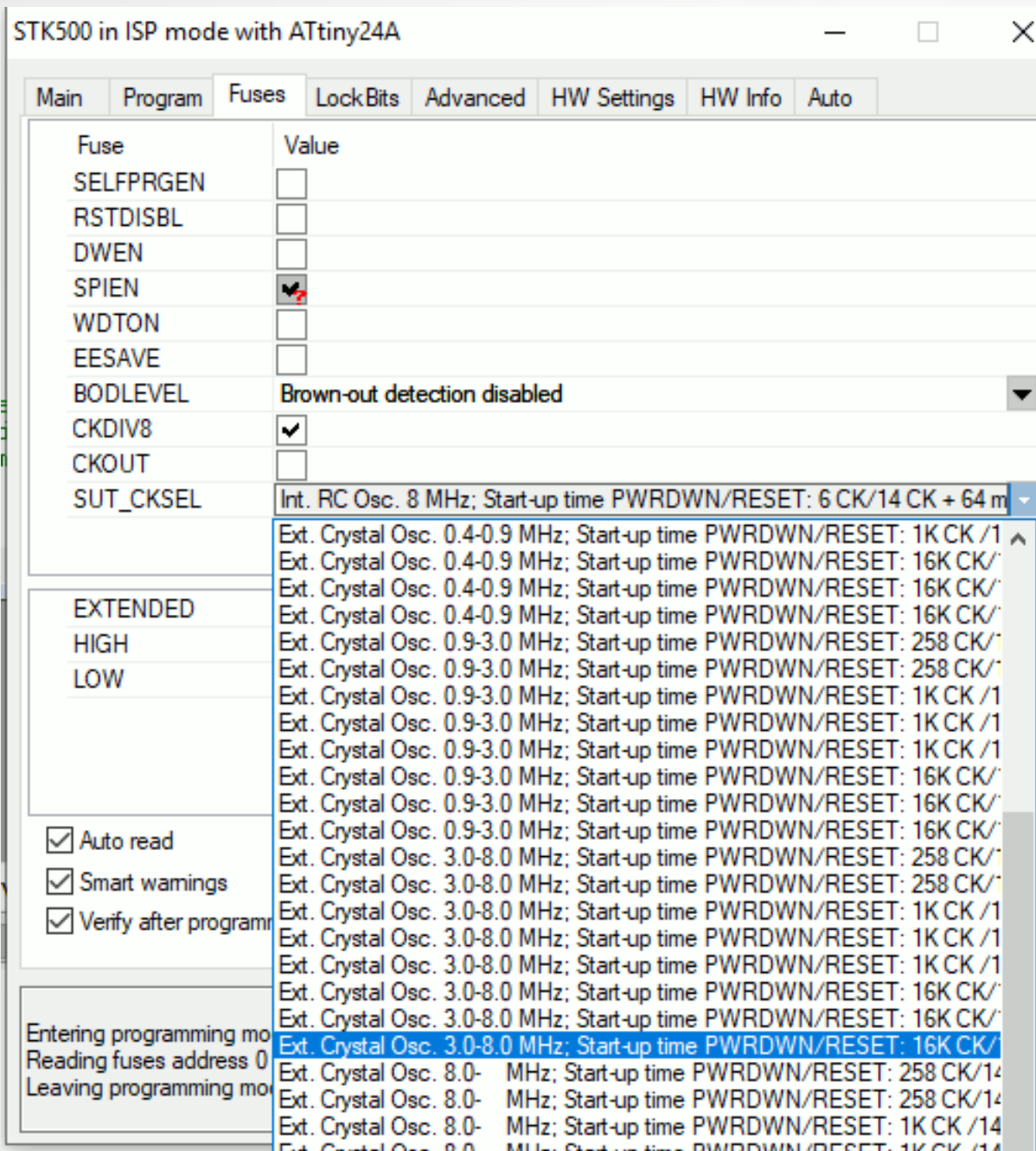
The xtal driven LED blinker



The crystal has 4 MHz, the two capacitors 18pF.

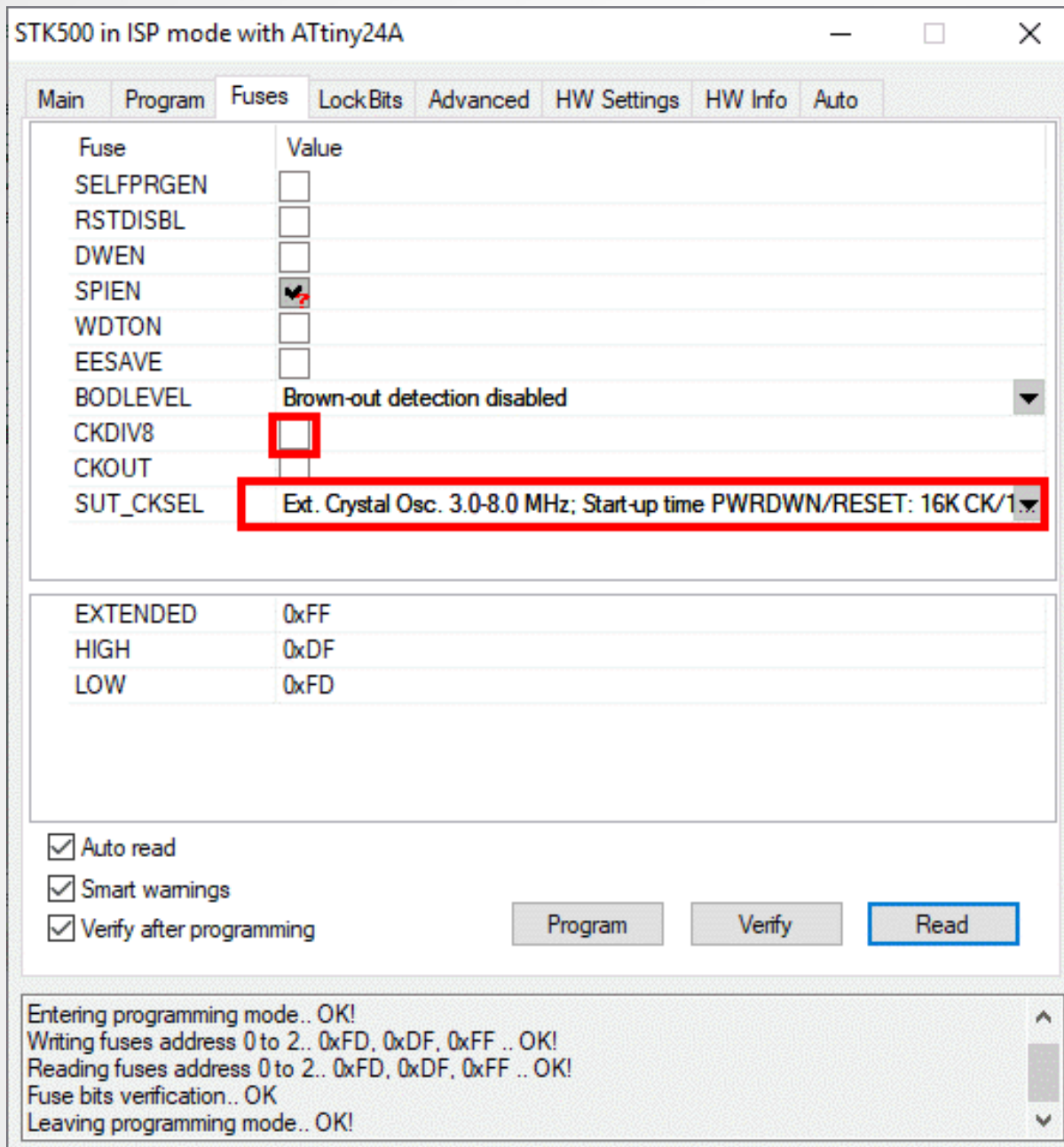
The LED and its current limiting resistor is now connected to PB2.

Changing the clock fuse



- To change the clock fuse, change to the Fuses tab.
- From the dropdown list in SUT_CKSEL select Ext. Crystal Osc. 3.0-8.0 MHz.
- Choose the longest duration from the different start-up times.
- Before pressing Program see the next page.

Clearing the CLKDIV8 fuse



- The crystal oscillator should now appear in the SU_CKSEL field.
- Then clear the CKDIV8 fuse. If that would be forgotten, the controller would run with 1/8th of the crystal's frequency, $4 \text{ MHz} / 8 = 500 \text{ kHz}$.
- Now you can press Program.
- What would happen, if the crystal is not installed? Already verification would fail. In that case either repair the error or attach an RC oscillator with 1 to 4 MHz to XTAL1 and clear the fuse.

AVRs are extremely flexible

- **With their clever designed hardware AVRs can**
 - **be tailored to any electronic system,**
 - **exchange dozens of CMOS/TTL devices in a system by one single IC,**
 - **be used in small and very small quantities at an extremely low price (from less than 1 € up to 6 € per piece),**
 - **be programmed within the system, easing debugging and functional changes,**
 - **ideally be used by amateurs as well as on an industrial scale.**

Questions and tasks in Lecture 5

Question 5-1: The instructions PUSH register throws the register content onto the stack and POP register recalls the last value from the stack.

What happens with a POP when the PUSHed values are exhausted (Which value comes back? Which position has the stack pointer?).

Bonus question: What happens if you PUSH a value to an uninitialized stack (the stack pointer is at zero after a RESET).



Questions and tasks in Lecture 5 - Continued

Task 5-2: Design a schematic and program source code that blinks a 2-pin red/green LED with 0.9 seconds in green and 0.1 seconds in red.

Hint: Forward voltages for such LEDs are around 2.0 Volt. Think about using Zener diodes.

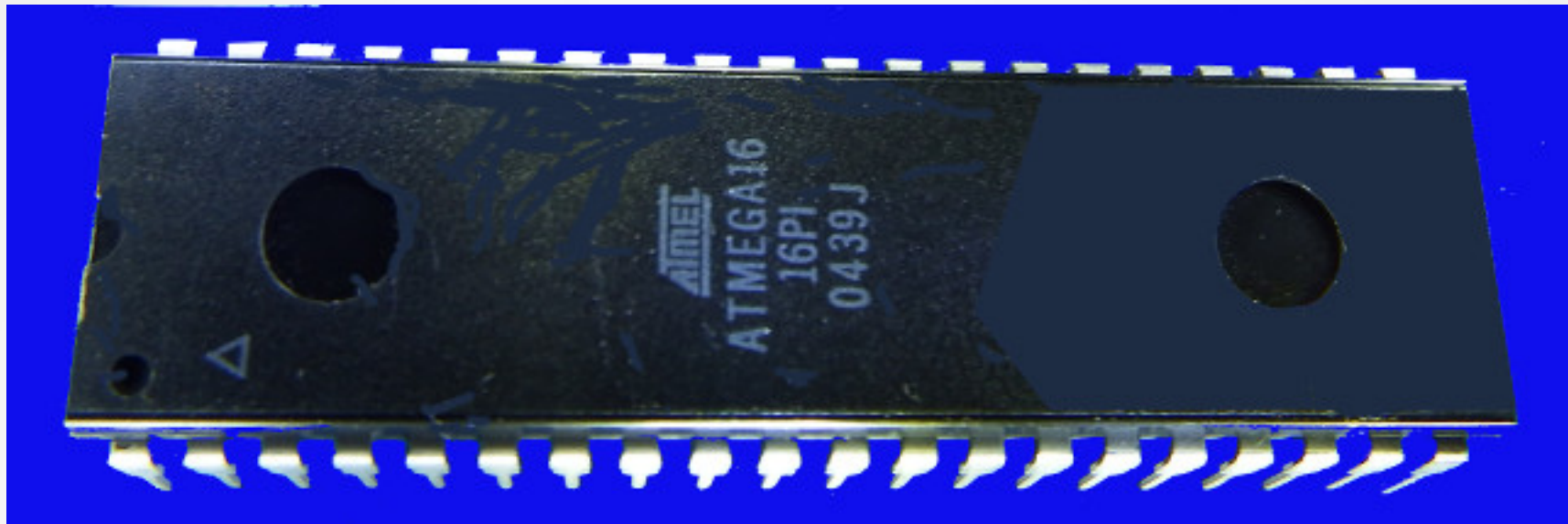


Questions and tasks in Lecture 5 - Continued

Task 5-3: Design a schematic and write a program that outputs four bits in binary format on PA0 to PA3 (ones: LED on, zeroes: LED off), then waits for a second and counts up.

Bonus question: What happens if 0x10 is reached after 16 seconds? Is it necessary to restart the counter?





Lecture 6: Timer/Counter

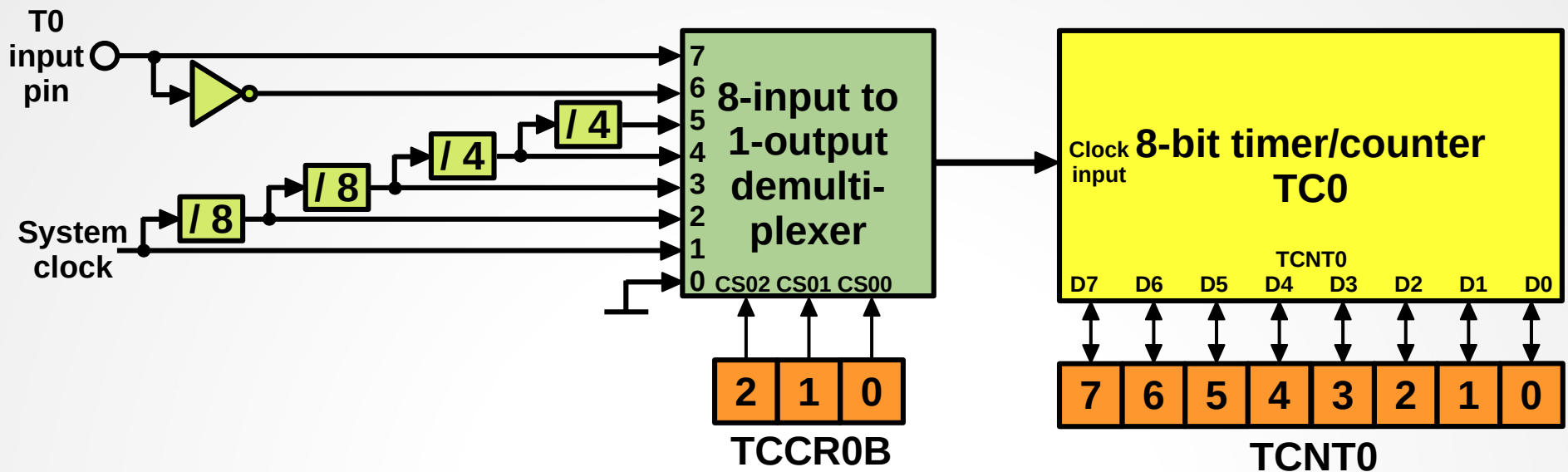
Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Timers/Counters in AVR

- **Doing timing tasks with counting loops is not very effective: the AVR can do nothing else than counting and counting can better be done with timers.**
- **All AVR**s have at least one timer on board that can do that.

8-bit timer/counter with prescaler



- 8-bit timers/counters count from 0 up to 255, then restart.
- They are switched on by setting bits 0 to 2 in the port register TCCR0B to a non-zero value and off by setting 0 to.
- The timer can be clocked by the prescaled system clock (divided by 1, 8, 64, 256 or 1,024), the counter can be clocked by positive or negative pulses on the T0 input pin (pin 10 in ATtiny24's PDIP package).

Achievable TC frequencies

- The following timer clock frequencies can be achieved with different prescaler values:

Prescaler	System clock Internal RC oscillator			System clock crystal oscillator		
	1 MHz	4 MHz	8 MHz	2.048 MHz	3.2768 MHz	4.194304 MHz
0	Timer off					
1	1 MHz	4 MHz	8 MHz	2.048 MHz	3.2768 MHz	4.194304 MHz
8	125 kHz	500 kHz	1 MHz	256 kHz	409.6 kHz	524.288 kHz
64	15.625 kHz	62.5 kHz	125 kHz	32 kHz	51.2 kHz	65.536 kHz
256	3.90625 kHz	15.625 kHz	31.25 kHz	8 kHz	12.8 kHz	16.384 kHz
1,024	976.563 Hz	3.90625 kHz	7.8125 kHz	2 kHz	3.2 kHz	4.096 kHz

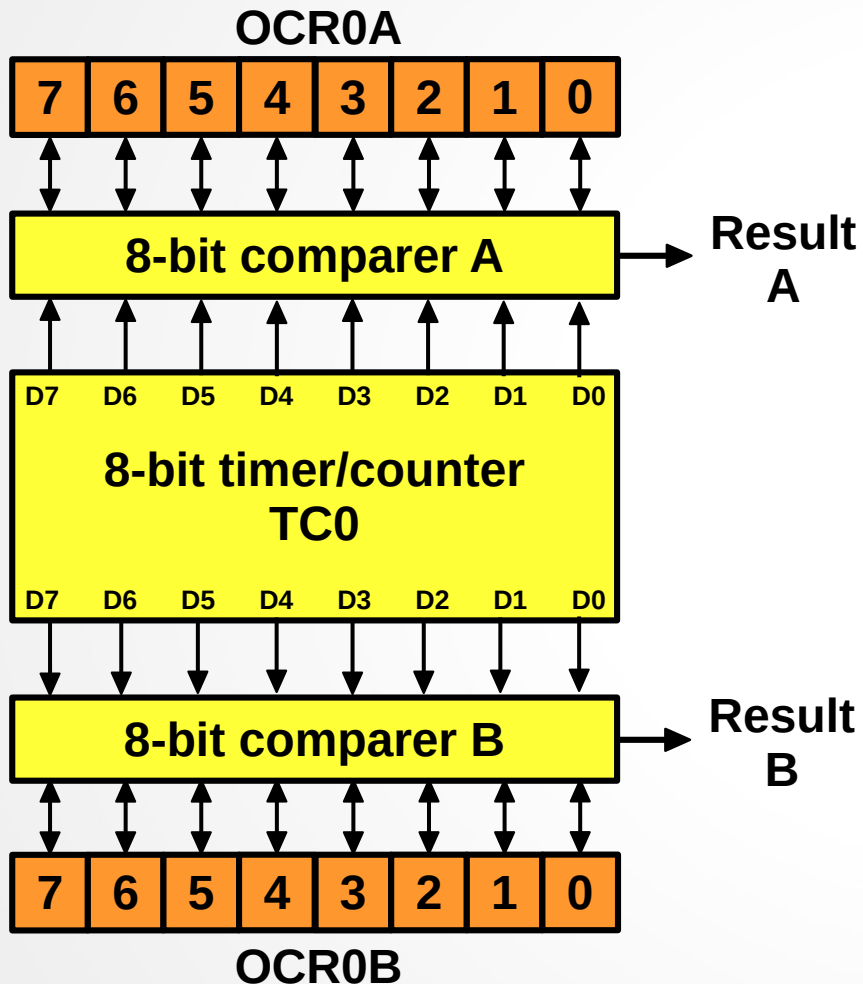
Achievable frequencies TC overflow

- The timer overflows after 256 TC clock cycles. The following shows the frequencies at which this happens:

Prescaler	System clock Internal RC oscillator			System clock crystal oscillator		
	1 MHz	4 MHz	8 MHz	2.048 MHz	3.2768 MHz	4.194304 MHz
0	Timer off					
1	3.90625 kHz	15.625 kHz	31.250 kHz	8 kHz	12.8 kHz	16.384 kHz
8	488.28125 Hz	1.953125 kHz	3.90625 kHz	1 kHz	1.6 kHz	2.048 kHz
64	61.03515625 Hz	244.140625 Hz	488.28125 Hz	125 Hz	200 Hz	256 Hz
256	15.2587890625 Hz	61,03515625 Hz	122.0703125 Hz	31.25 Hz	50 Hz	64 Hz
1,024	3.814697265625 Hz	15.2587890625 Hz	30.517578125 Hz	7.8125 Hz	12.5 Hz	16 Hz

Comparers

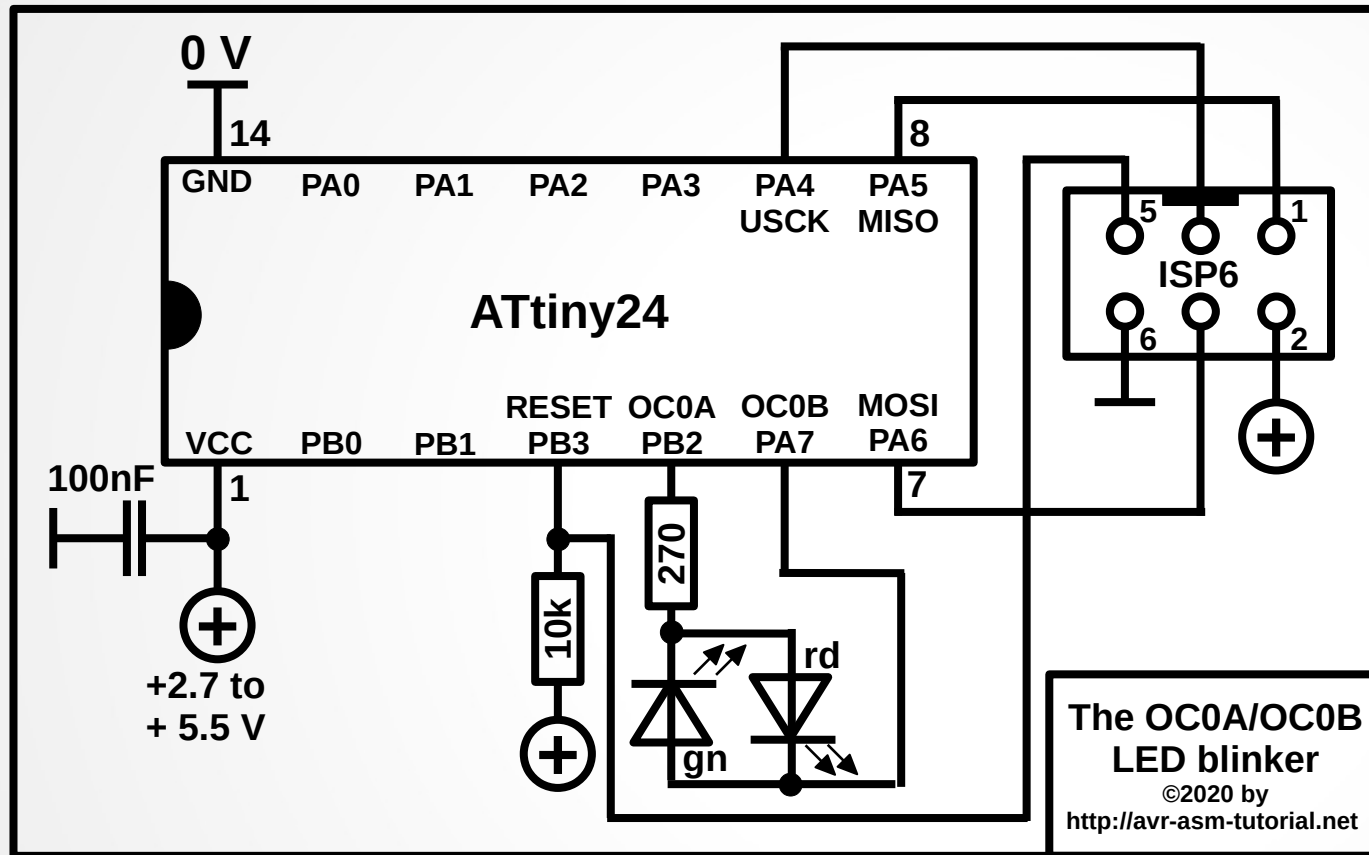
- Most timers/counters have two comparers on board. Those compare the timer/counter's state continuously with compare match values that were written to the comparer's port register.



- In the timer/counter clock cycle following the compare match, one or two output pins can either be set, be cleared or be toggled.
- Most timers/counters have two comparers on board. Those compare the timer/counter's state continuously with compare match values that were written to the comparer port registers.

The hardware of the toggles

- Build the following hardware with a red/green Duo-LED:



Toggling OC0A and OC0B

- The following source code
 - sets the comparer port registers to 255,
 - enables toggling of the OC0A and OC0B output pins, and
 - starts TC0 with a prescaler value of 1,024.

```
; Timer toggles OC0A and OC0B
; Configure pins as output (switching output driver on)
SBI DDRB, DDB2 ; The OC0A output pin is PB2, set as output
SBI DDRA, DDA7 ; The OC0B output pin is PA7, set as output
SBI PORTA, PORTA7 ; Set OC0B = PA7 high
; Set output pins to toggle mode
LDI R16, (1<<COM0A0) | (1<<COM0B0) ; Toggle output pins A and B
OUT TCCR0A,R16 ; to TC0 control register A
; Start TC0 in normal mode with its prescaler
LDI R16, (1<<CS02) | (1<<CS00) ; Prescaler to 1,024
OUT TCCR0B, R16 ; to TC0 control register B
; Endless loop
Loop:
    RJMP Loop
```

Hints to the source code

- Please refer to the device's data book, chapter 11.9 for the port register descriptions.
- The formulation $(1 \ll \text{COM0A0}) \mid (1 \ll \text{COM0B0})$ works as follows:
 1. The assembler picks the 1 and shifts it left (\ll) for COM0A0 times. As COM0A0 is 6 (taken from the def.inc file), the 1 is shifted 6 times to the left and 0b0100,0000 results for the first bracket.
 2. Then the assembler picks the other 1 and shifts it left (\ll) for COM0B0 times. COM0B0 is 4, the result is 0b0001,0000 for the second bracket.
 3. The \mid means „Binary OR“ and combines both with or'ing them bit-by-bit. The result is 0b0101,0000.
 4. The reason why this formulation is used is that COM0A0 is simpler to remember than 0b0100,0000 and the position of the two bits is always correct, even if moved to somewhere else inside this port register.

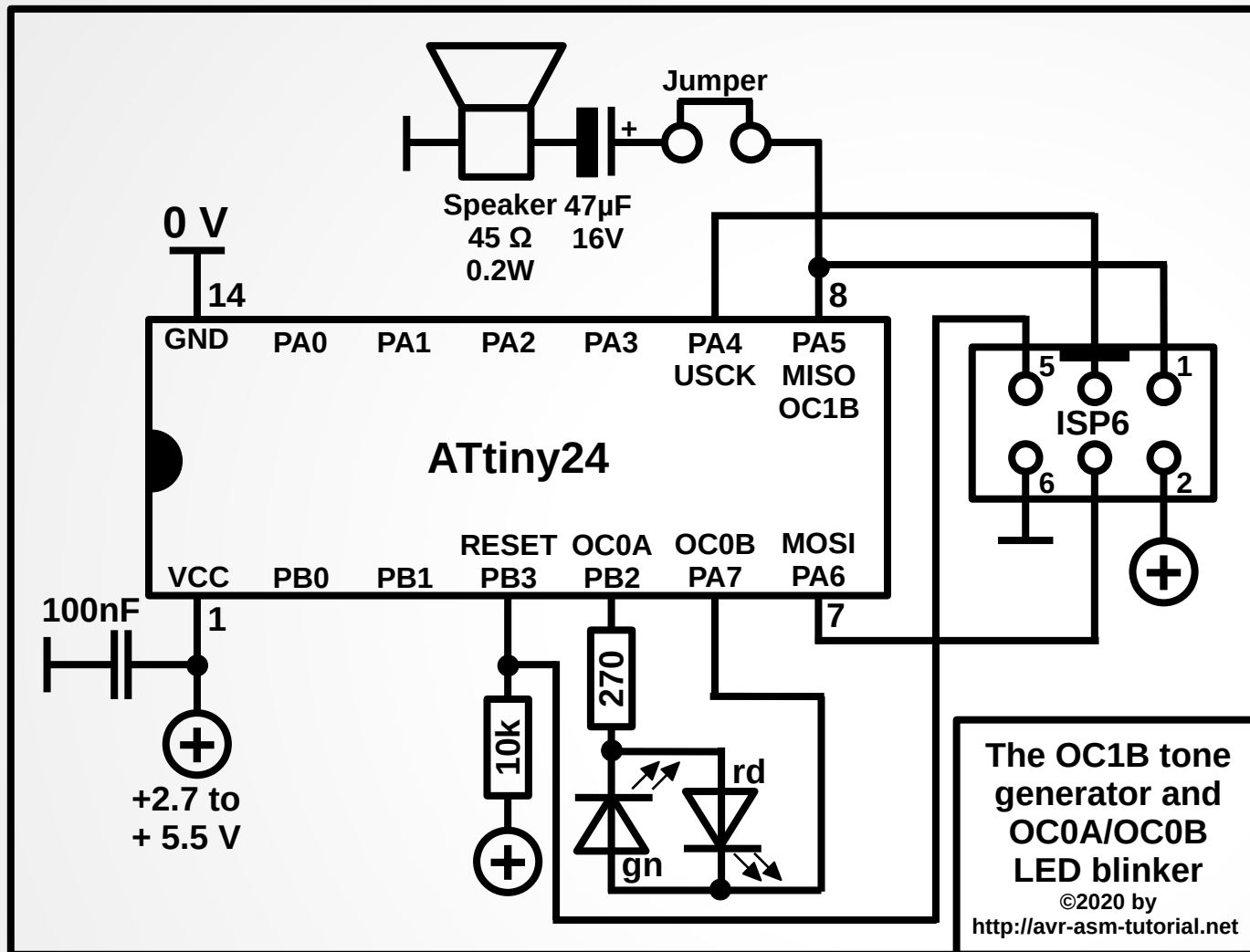
Further hints on the source code

- **The reason why PA7 on start-up is set is as follows:**
 - **The port register PB2 (=OC0A) is cleared on RESET.**
 - **The same with PA7 (=OC0B). As both are zero on start-up, the LED would be off.**
 - **As both toggle their PORT state after 256 TC clocks, they both are set to one, the LED still will be off.**
 - **By setting the PA7 port bit, the state of this is always the opposite of PB2, and the LED is always on. The two I/O-pins reverse the voltage of the LED on every toggle.**

The 16-bit counter TC1

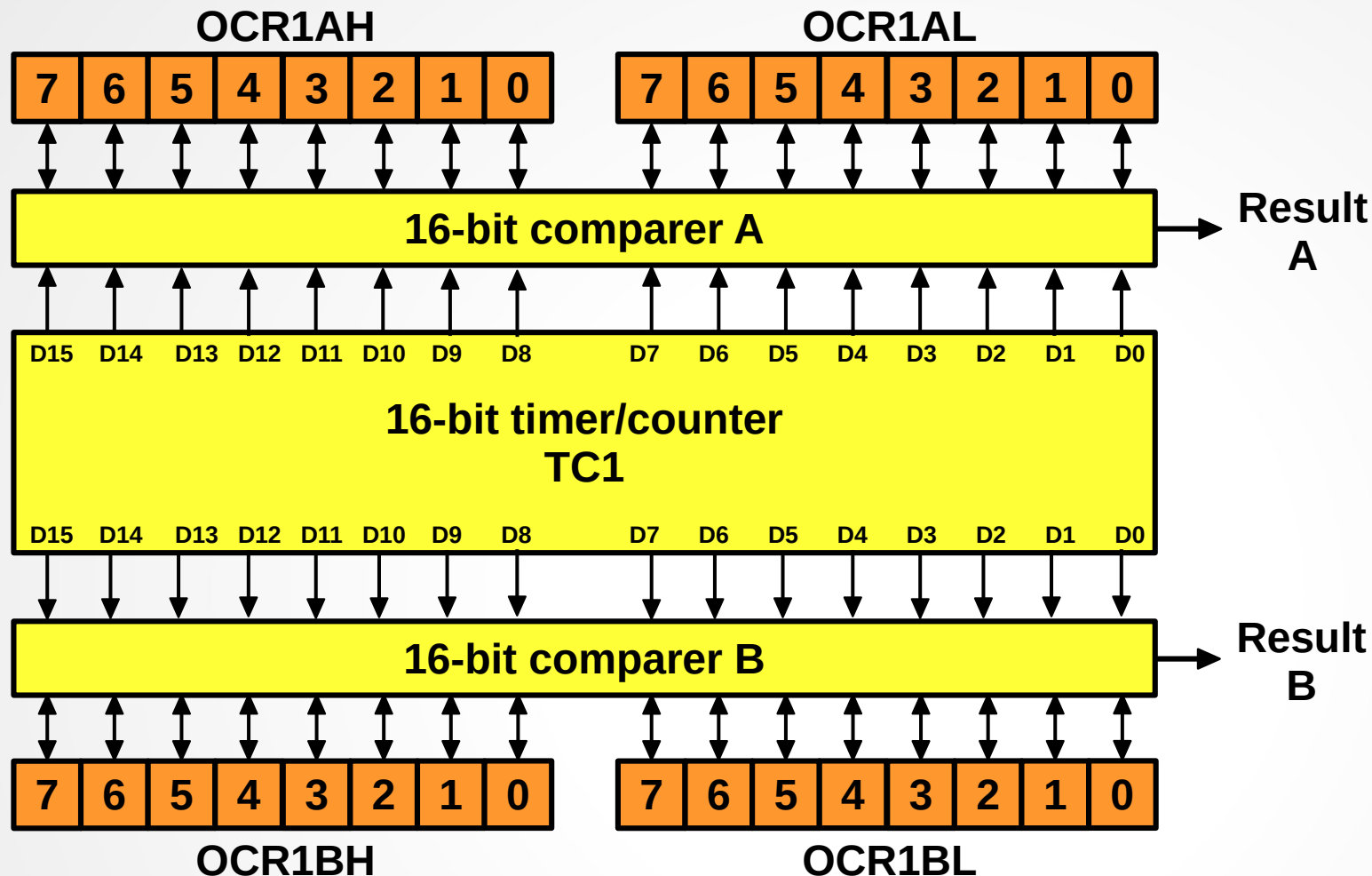
- The ATtiny24 has a second counter on board, which can count with 16 bits up to 65,535.
- 16-bit counters are a little bit different in handling, this is why we have to learn some new stuff here.
- In this section we work with the TC in a second mode: CTC. That means: the TC counter is cleared (Clear Timer on Compare, restarts at zero) in the clock cycle following the compare match.
- This allows to generate a wider variety of frequencies.

The hardware



- This is the combined tone generator and blinker.
- The tone is generated with TC1's OC1B pin.
- To avoid interference with the MISO function, a jumper disconnect the speaker during programming.

16-bit timers



- 16-bit timers have all count and compare port registers in 2 bytes each: Low and High.
- The two bytes have to be written/read in two single operations.
- To ensure that the high and low byte belong to the same counter state, reading and writing use an interim buffer where the first byte is stored until the second byte is written or read.

Assessing 16-bit port registers

- During read access, the low byte must be read first. This stores the associated high byte in the temporary register, which can be read next.

; Reading the 16-bit counter

IN R16, TCNT1L ; Read low byte first, this stores the high byte
IN R17, TCNT1H ; Then the high byte

- During write access, the high byte must be written first. This stores the high byte in the temporary register.

; Writing to the 16-bit counter

LDI R16, High(count) ; The high byte
OUT TCNT1H, R16 ; Write high byte first, this stores the high byte in the interim
LDI R16, Low(count) ; The low byte next
OUT TCNT1L, R16 ; Then the low byte, the high byte taken from interim

- Note that the writing of the 16 bits simultaneously takes place only after the low byte has been written.

CTC mode of the timer

- The following shows the CTC mode of the timer. CTC modes can be used in 8- and in 16-bit timers/counters.
- CTC modes use a compare register (normally A) to restart the timer in the next following timer clock cycle that follows compare match.
- If your counter shall only run from 0 to 5, set its compare register to 5. When the sixth counter pulse occurs, the counter will restart at zero.
- The number of active counter clock cycles is always CompareMatchA plus one.
- The CTC mode requires that the WGM bits in the control registers A and B are set to the appropriate combination.

Timer modes TC1

Table 12-5. Waveform Generation Modes

Mode	WGM 13:10	Mode of Operation	TOP	Update of OCR1X at	TOV1 Flag Set on
0	0000	Normal	0xFFFF	Immediate	MAX
1	0001	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0010	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0011	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0100	CTC (Clear Timer on Compare)	OCR1A	Immediate	MAX
5	0101	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0110	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0111	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1000	PWM, Phase & Freq. Correct	ICR1	BOTTOM	BOTTOM
9	1001	PWM, Phase & Freq. Correct	OCR1A	BOTTOM	BOTTOM
10	1010	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1011	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1100	CTC (Clear Timer on Compare)	ICR1	Immediate	MAX
13	1101	(Reserved)	–	–	–
14	1110	Fast PWM	ICR1	TOP	TOP
15	1111	Fast PWM	OCR1A	TOP	TOP

- These are all available timer modes for TC1.
- The CTC modes are marked red.

Setting the WGM bits

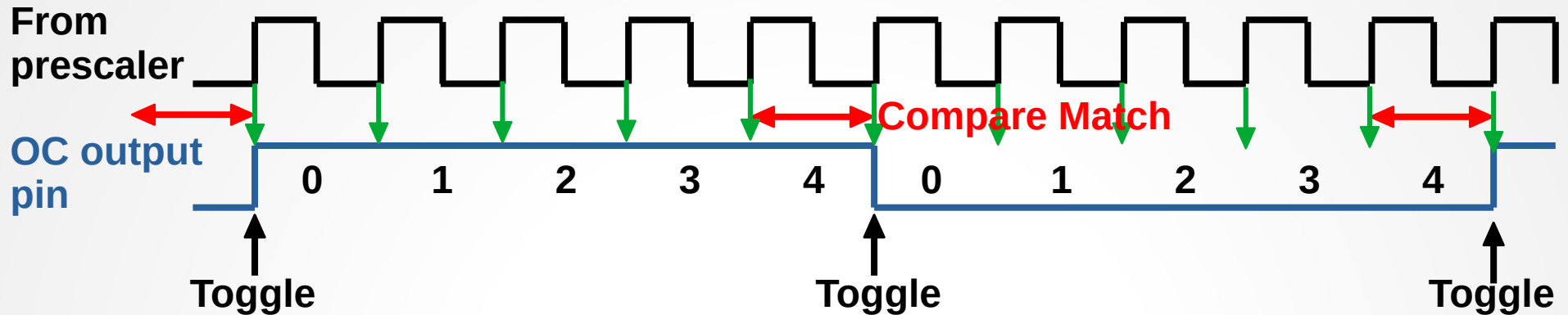
- The WGM bits are located in the two control port registers TCCR1A and 1B:

Bit	7	6	5	4	3	2	1	0	
0x2F (0x4F)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- So WGM10 and WGM11 have to be written together with the COM bits, while WGM12 and WGM13 have to be written together with the clock selection bits CS.

Generating tones

- The CTC mode and the OC pin works like this:



- The system clock, divided by the prescaler, advances the counter.
- When the counter reaches the compare value and the next counting pulse comes in, the OC pin toggles and the counter is cleared.
- The whole OC pin cycle requires two toggle events.

The frequency generated

- Because of this, the generated frequency is:
$$f = f_{\text{system}} / d_{\text{prescaler}} / (d_{\text{compare}} + 1) / 2$$
- For 1 MHz, a prescaler of 1 and a compare match value of 1,000 the generated frequency is 500 Hz.
- The lowest frequency at 1 MHz with a prescaler of 1 is 7.6 Hz, the highest is 500 kHz.
- So a prescaler of 1 allows generation of the complete audible frequency range.

Questions and tasks in Lecture 6

Question 6-1: At which frequency does the red/green LED blink at 1 MHz and at 4 MHz clock? (Hint: do not forget that one cycle has two phases – one forward red and one backward green!)

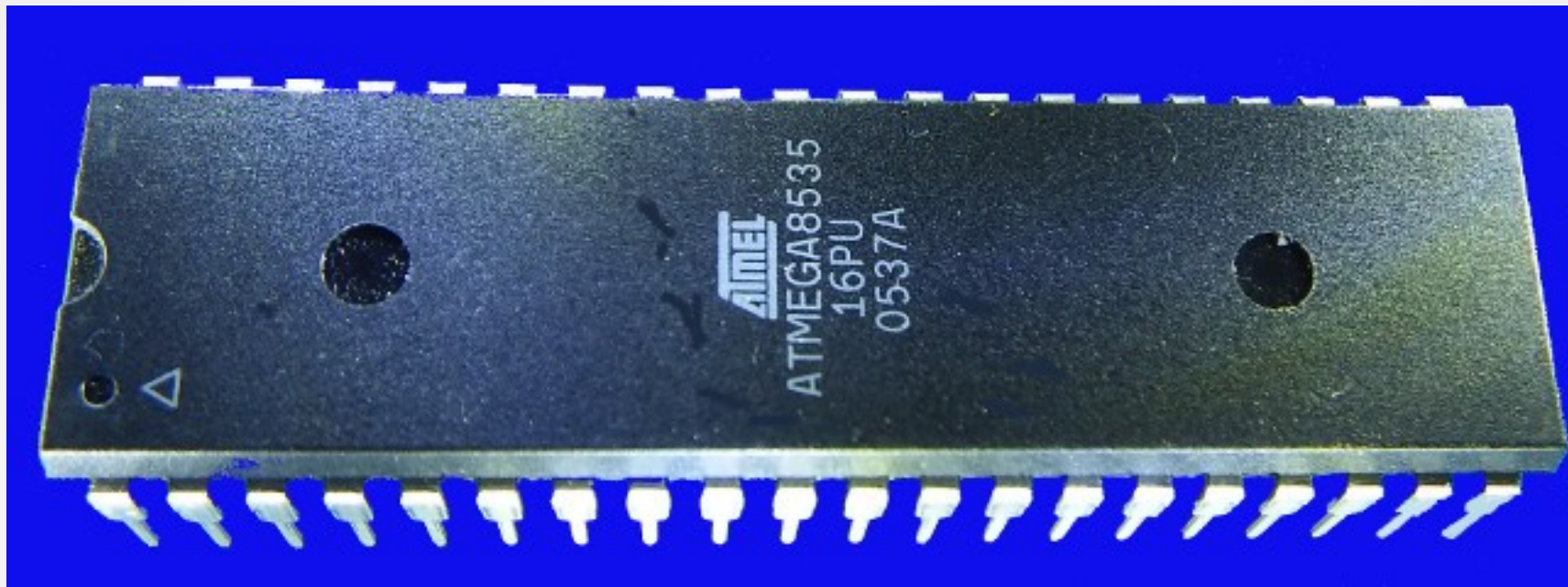
Bonus question: At which clock frequency has the controller to be operated to get a 1 Hz blink frequency? How can this be made electronically (search for „External clock“ in the handbook)? And at which ISP clock frequency has the controller to be programmed then? Does your hard- and software allow this (test this!)?



Questions and tasks in Lecture 6 - Continued

Task 6-2: Write a program that generates a pre-defined frequency on the speaker (calculate the compare match value from the given frequency in Milli-Hertz) and that blinks in a second rhythm. When the LED is on, change the tone to be exactly one octave higher. Test your program on the breadboard and with the simulator (by enabling the timer view).

Bonus question: What are the compare match values for the gambit tones a to g (440, 493.883, 523.251, 587.330, 659.255, 698.456 and 783.991 Hz) and how accurate can those be generated with an xtal-driven ATtiny24 at 4 MHz? Is a crystal with 4.194304 MHz a better choice in this case? Why?



Lecture 7: Interrupts

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Interrupts

- When the timer/counter overflows or reaches compare match A or B it would be a good idea to have a mechanism that interrupts normal program execution, so the controller can react on this event and do the necessary steps.
- After doing these steps the controller shall continue to work as if nothing has happened in between.
- This mechanism is implemented in all AVR's. It requires the following:
 1. The stack has to store the execution address when interrupted.
 2. The controller has to have a place where it jumps to when interrupted.
 3. Further interrupts have to be blocked and their execution has to be delayed, but not simply forgotten.
 4. As the AVR's have only one status register, its content has to be saved and restored before ending the interrupt service routine.

Interrupt enable and stack

- The status register has a flag called I that enables reaction to interrupt flags:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- To enable this set I to one:

```
; Enable general interrupt flag
SEI ; Set I bit in SREG
; CLI ; The opposite: clear the I bit and stop interrupt execution
```

- As the stack is needed to store the return address, stack has to be initiated before interrupts can be allowed:

```
; Initiate the stack
; LDI R16, HIGH(RAMEND) ; In larger devices with more than 256 byte SRAM
; OUT SPH, R16 ; Set MSB stackpointer
LDI R16, LOW(RAMEND) ; In all devices
OUT SPL, R16 ; Set LSB stackpointer
```

Interrupt vectors

- The places where all interrupts jump to are the interrupt vectors.
- ATtiny24 has 17 different vectors, others have different ones.
- Int vectors are located from address 0000 on.

Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Capture Event
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B
9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	EE_RDY	EEPROM Ready
16	0x000F	USI_STR	USI START
17	0x0010	USI_OVF	USI Overflow

Filling the int vector jump table

```
; The interrupt vector jump table of an ATtiny24  
.CSEG ; Beginning of the code segment  
.ORG 0x0000 ; Starts at address 0  
    RJMP Main ; On reset jump to the init routines  
    RETI ; INT0 interrupt, not used  
    RETI ; PCINT0 interrupt, not used  
    RETI ; PCINT1 interrupt, not used  
    RETI ; WDT interrupt, not used  
    RETI ; TC1-CAPT interrupt, not used  
    RETI ; TC1-COMP-A interrupt, not used  
    RETI ; TC1-COMP-B interrupt, not used  
    RETI ; TC1-OVF interrupt, not used  
    RETI ; TC0-COMP-A interrupt, not used  
    RETI ; TC0-COMP-B interrupt, not used  
    RJMP TC0ISR ; TC0-OVF interrupt, used  
    RETI ; ANACOMP interrupt, not used  
    RETI ; ADC interrupt, not used  
    RETI ; EERDY interrupt, not used  
    RETI ; USI-STR interrupt, not used  
    RETI ; USI-OVF interrupt, not used
```

- To ensure that these vectors are really placed at address 0000, the two directives **.CSEG** (for code segment) and **.ORG** (for origin) ensure that no code is placed before the vectors.
- The vector consists of **RJMP** instructions and **RETI** instructions.
- The **RJMP Main** is executed on any hardware reset.
- The **TC0 overflow** interrupt jumps to a interrupt service routine.
- All other vectors are inactive and return from the interrupt (**RETI**).

The interrupt service routine

- The place where the TC0-OVF interrupt jumps to has to be written to a location outside the vectors, e. g. like this:

```
; The interrupt service routine of the TC0-OVF  
Tc0Isr:  
    SBI PINB, PINB2 ; Toggle the PB2 pin  
    RETI ; Return from interrupt
```

- The SBI on the input port register PINB toggles the pin PB2.
- The RETI instruction does two things:
 - It sets the I flag in the SREG to one again. I was cleared when the interrupt began to block further interrupts.
 - It returns to the program address where the interrupt occurred, which is taken from the stack.
- It is recommendable to fill all unused interrupt vectors with RETI to have all interrupts finalized in a controlled manner.

Priority rules

- If two or more interrupt conditions occur at exactly the same time:
 1. The interrupt scheme is prioritized. The higher the respective interrupt is listed in the jump vector list, the earlier is it executed. The highest priority always have the external interrupts.
 2. If an interrupt vector is executed, its interrupt flag is automatically cleared.
 3. If an interrupt is not yet executed because a higher-ranked interrupt is executed first, its interrupt flag still remains set.
 4. Immediately after the higher-ranked interrupt is terminated with RETI, the pending lower-ranked interrupt is executed.
 5. To avoid interrupt overruns (a set interrupt flag is set again) interrupt-service-routines shall always be short enough to avoid such overruns of lower-ranked interrupts.

The status register during interrupts

- The status register is only once implemented. If the interrupt service routine uses instructions that alter SREG flags, their initial state has to be restored.
- The SREG can be stored in a register (here R15).

```
.def rSreg = R15 ; Place to store SREG during interrupts
; Interrupt service routine TC0 overflow
Tc0Isr:
    IN rSreg, SREG ; Store the status register's content
    DEC R16 ; Decrease R16 (instruction changes flags)
    BRNE Tc0IsrRet
    SBI PINB,PINB2 ; Toggle PB2
Tc0IsrRet:
    OUT SREG, rSreg ; Restore previous content of SREG
    RETI ; Return from interrupt
```

- As interrupts occur asynchronous to the normal program flow, it has to be ensured that SREG flags are not changed uncontrolled by any interrupt service routine, otherwise unpredictable conditions can result.

Advantages of interrupts

- **The advantages of using interrupts for programming controllers are:**
 - **Interrupts are very fast, they react within 7 clock cycles (a few micro-seconds) on events (e.g. on external interrupts).**
 - **Lots of different tasks can be programmed to run simultaneously and without delaying other time-critical tasks.**
 - **Interrupts do not waste time and program code, even not for complex timing tasks, the CPU can do other tasks in that time.**
 - **Interrupts save operating current: the controller can go to sleep in idle mode, and only wakes up if needed (search for SLEEP in the handbook).**
- **Interrupt Service Routines have to be short (never use timing loops inside such a routine). If you need longer actions, use flags.**

Timer in PWM modes

- **Beside normal-mode and CTC-mode all timer in AVR's have also PWM modes.**
- **In PWM modes the timer counts up (in fast PWM mode) or up and then down (in phase-correct PWM mode).**
- **In PWM modes the OC pins work differently:**
 1. **At the beginning of the PWM cycle the OC pin is either set (normal PWM) or cleared (reverse PWM).**
 2. **When the compare match occurs, the OC pin is either cleared (in normal PWM) or set (in reverse PWM).**
- **That produces a signal with a controlled pulse-width on the pin.**
- **With compare match = 0 a signal of one pulse duration is produced, with compare match = 255 (in an 8-bit timer), the signal is fully on.**

Timer in PWM modes - TOP=255

- Beside normal-mode and CTC-mode all timer in AVR's have also PWM modes.
- In PWM modes the timer counts up (in fast PWM mode) or up and then down (in phase-correct PWM mode).
- In PWM modes the OC pins work differently:
 1. At the beginning of the PWM cycle the OC pin is either set (normal PWM) or cleared (reverse PWM).
 2. When the compare match occurs, the OC pin is either cleared (in normal PWM) or set (in reverse PWM).
- That produces a signal with a controlled pulse-width on the pin.
- With compare match = 0 a signal of one pulse duration is produced, with compare match = 255 (in an 8-bit timer), the signal is fully on.

Timer in PWM modes - TOP=255

- The 8-bit TC0 modes are controlled with the WGM bits:

Table 11-8. Waveform Generation Mode Bit Description

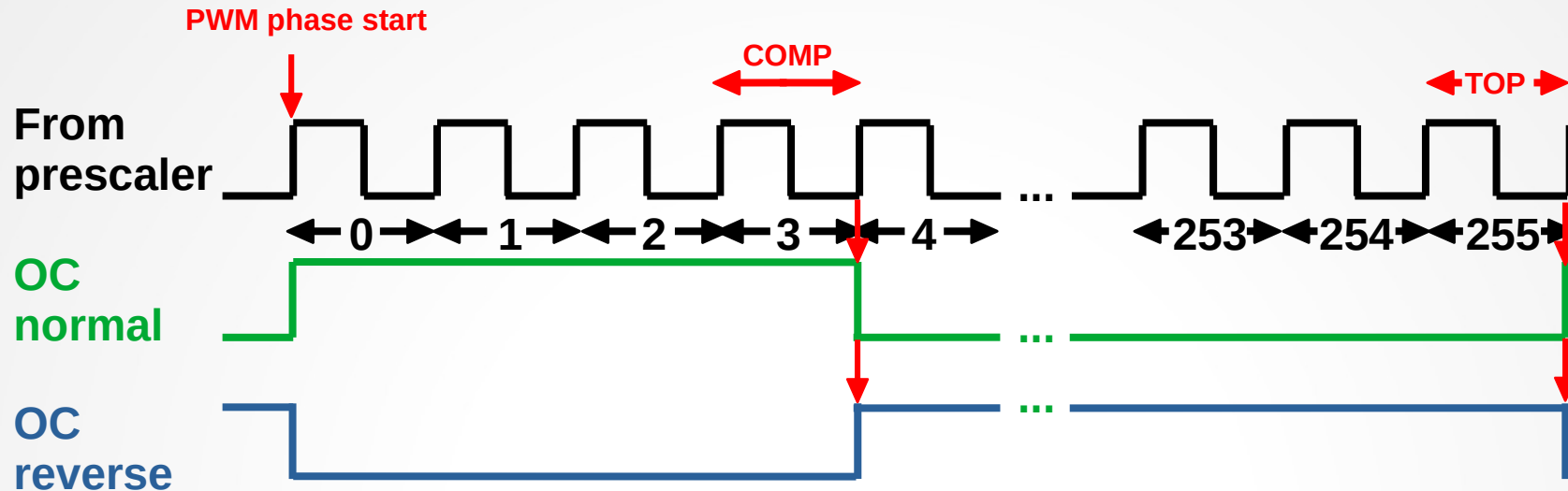
Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on ⁽¹⁾
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	—	—	—
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	—	—	—
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Note: 1. MAX = 0xFF
BOTTOM = 0x00

- There are two Fast PWM modes: one with 0xFF and one with OCR0A as TOP values.

Timer in PWM modes - TOP=255

- The fast PWM mode with TOP=255 works as follows:



- At TCNT0=0 (Phase start) the OC pin is set (OC normal) or cleared (OC reverse).
- On the next counting pulse after compare match has occurred, the OC pin is cleared (OC normal) or set (OC reverse).
- That changes only when the TOP value (255) is reached: on the next counting pulse following TOP match the PWM restarts at 0 and the OC pins are set or cleared.

Timer in PWM modes - TOP=255

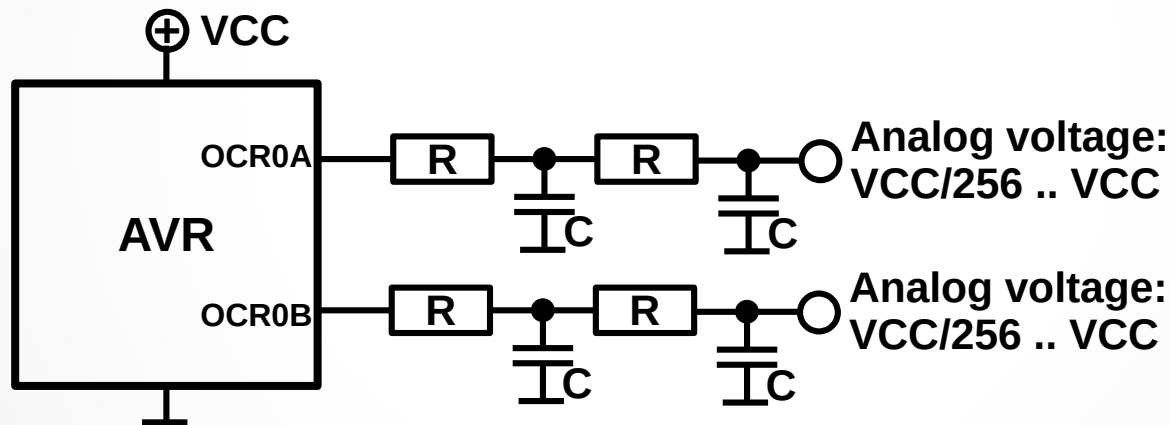
- In PWM modes changed compare values are not immediately applied.
- Writing compare values writes those to an interim storage instead of the compare match port register.
- Only after a new PWM cycle starts at 0 the stored new compare match value is actually written to the compare port register.
- That mechanism ensures that in each PWM cycle exactly one compare match will occur, even if the new compare match value is below the current count: it is only applied at the next PWM cycle start.
- The column „Update OCRx“ in the mode table shows that.
- In fast PWM with TOP=255 all three interrupts (overflow, compare match A and compare match B) can be used.

Timer in PWM modes - TOP=OCRA

- In fast PWM mode with TOP in OCRA the TOP value can be set to any value to achieve a shorter PWM period (e. g. 128, 64 or 10).
- This increases the PWM frequency, but limits the compare match range, too.
- Changes to OCRA also come only into effect on PWM cycle start.
- As OCRA is used, only OCRB can be used as PWM channel.
- If OCRA is below 255, the overflow interrupt condition will never be reached, use the compare match A and/or B interrupt instead.

Practical applications of fast PWM

- Fast PWM can be used for the following:
 - a) for LED dimming: the LED is only on in the active phase of the PWM, so the time over which the LED lights can be reduced from full down to $1 / 256 = 0.39\%$ of the time.
 - b) for conversion of a digital value to an analog voltage (use an RC filter to reduce the ripple of the PWM voltage):



- c) for motor power regulation (switched power): use a power transistor to achieve higher power consumption, avoid humming by a higher-than-audible PWM frequency.

Conclusions timers and PWM

- **With a controller producing signals with a timer in the different modes and with interrupts you can:**
 - a) generate exact signals and control those with interrupts,**
 - b) avoid lengthy counting sequences if you need exact timing,**
 - c) with a few lines of fast, lean and effective source code resolve any timing and signal task,**
 - d) replace a lot of CMOS or analog electronics by one single controller.**

Questions and tasks in Lecture 7

Task 7-1: Write a program that blinks a LED with a crystal clock at 4 MHz in an exact one-second rhythm using TC0 in CTC mode and its COMP-A interrupt to down-count a register. Don't forget to restart the counter register when exhausted!

Bonus question: Without counter register restart: which blink frequency would result instead of 1 Hz?



Questions and tasks in Lecture 7 - Continued

Task 7-2: Write a program with the 16-bit counter TC1 as sound generator with 1,000 Hz in CTC mode and use the COMP-A-interrupt to down-count a 16-bit double register R25:R24 to stop the sound after exactly two seconds. (Hints: 16-bit double registers can be down-counted in 16 bit mode with SBIW R24, 1 (SuBtract Immediate Word) and sound output can be stopped by clearing the pin on compare match when keeping TC1 still running).

Bonus question: What would be the COMPA values and the down-count value in R25:R24 for the whole a-to-g gambit frequencies?

Questions and tasks in Lecture 7 - Continued

Task 7-3: Write a program that provokes two interrupts at exactly the same time with the two timers TC0 and TC1 in the ATtiny24 and with the two compare A interrupts. Simulate this with `avr_sim` and see if the prioritization of interrupts function as desired within the simulator.



Lecture 8: ADC & EEPROM

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Analog Digital Conversion

- Two thirds of all AVR types have an Analog Digital Converter (ADC) on board.
- An ADC converts the voltage on an ADC pin to a 10-bit wide digital value between 0x0000 and 0x03FF.
- ADC's function like this:
 - The analog voltages is stored in a capacitor (sample&hold).
 - In a stepwise process this is compared with half the reference voltage. If it is larger, the highest bit is one and the next comparison is with $\frac{3}{4}$ of the reference voltage. If not, the highest bit is zero and the next comparison is with $\frac{1}{4}$ of the reference voltage. This is repeated 10 times.
 - On completion of this process, the ADSC bit in the AD control register port is cleared and, if so enabled, an interrupt is generated.

ADC control

- The AD control register ADCSRA port looks like this:

Bit	7	6	5	4	3	2	1	0	
0x06 (0x26)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **ADEN** switches the ADC on or off. The first conversion needs slightly longer if the ADC was off before. Switch the ADC off if you want to reduce power consumption in battery operation.
- **ADSC** starts a conversion. This bit remains high until the conversion is completed and is then cleared.
- **ADATE** starts automatic conversion with the sources shown on the next page.
- **ADIE** enables interrupts on ADC completion, **ADIF** holds the respective interrupt flag.
- **ADPS2:0** control the prescaler of ADC's conversion clock (divides the system clock by between 2 and 128).

Auto-triggered ADC

- The Autotrigger selection bits are located in port register ADCSRB. These bits control the auto-trigger source:

Table 16-7. ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match A
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

- In free running mode, the ADC restarts immediately after completing the previous conversion.
- The ADC's result is only updated if the high byte in ADCH has been read, so the low byte in ADCL has to be read first!

The multiplexed channels

- The reference voltage can be selected with the bits REFS1:0 in the ADMUX port register:

Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	REFS1	REFS0	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- REFS1:0 = 0b00 uses the VCC voltage as reference, 0b01 uses the voltage on the ADC0 input pin, 0b10 uses a built-in 1.1 Volt source.
- All ADC input pins use the same ADC, selection is done with the MUX bits in ADMUX. MUX5:0 between 0b00.0000 and 0b00.0111 select the voltages on the pins ADC0 to ADC7.
- Further MUX combinations can select differential voltages between two ADC pins and can amplify this difference by 1 or 20.
- MUX5:0 = 0b10.0010 measures the internal temperature sensor in the device.

Left-adjusted result

- The result in the port registers ADCH:ADCL can be automatically left-adjusted by setting the ADLAR bit in control port ADCSRB:

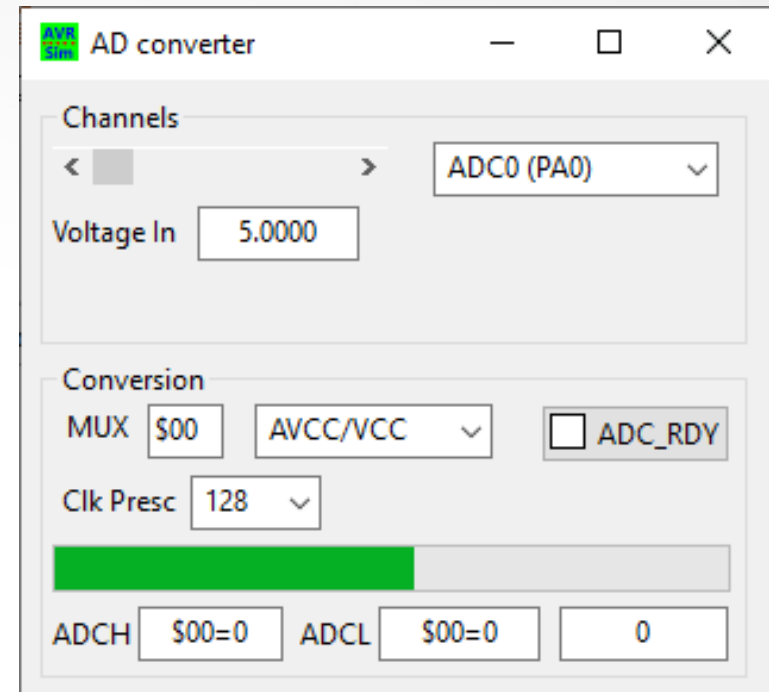
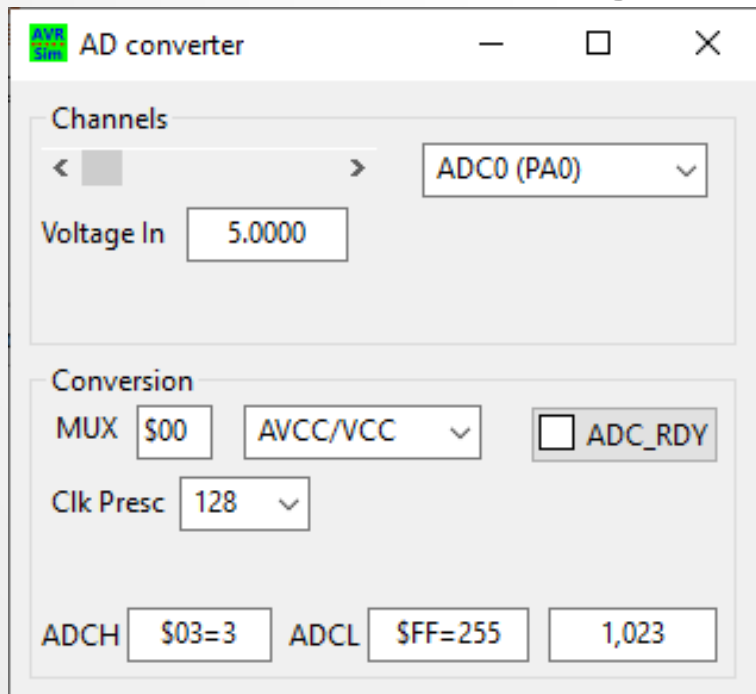
Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	BIN	ACME	–	ADLAR	–	ADTS2	ADTS1	ADTS0	ADCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- If ADLAR is set, the ADCH has the upper 8 bits of the result. If you do not need 10 bits accuracy, it is sufficient to read ADCH solely.
- A typical sequence to start an ADC conversion on the ADC0 pin with VCC as reference voltage and polling of ADSC is:

```
; AD conversion of ADC0 with ADSC polling
LDI R16, 0 ; MUX=ADC0, REFS=VCC
OUT ADMUX, R16 ; To MUX port register
LDI R16, (1<<ADEN)|(1<<ADSC)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
OUT ADCSRA, R16 ; Write this to control register A
AdcWait: ; Wait until ADSC is cleared
SBIC ADCSRA, ADSC ; Jump over next instruction if ADSC is clear
RJMP AdcWait ; ADSC not yet cleared
IN R0, ADCL ; read LSB result
IN R1, ADCH ; read MSB result
```

Simulating AD conversion

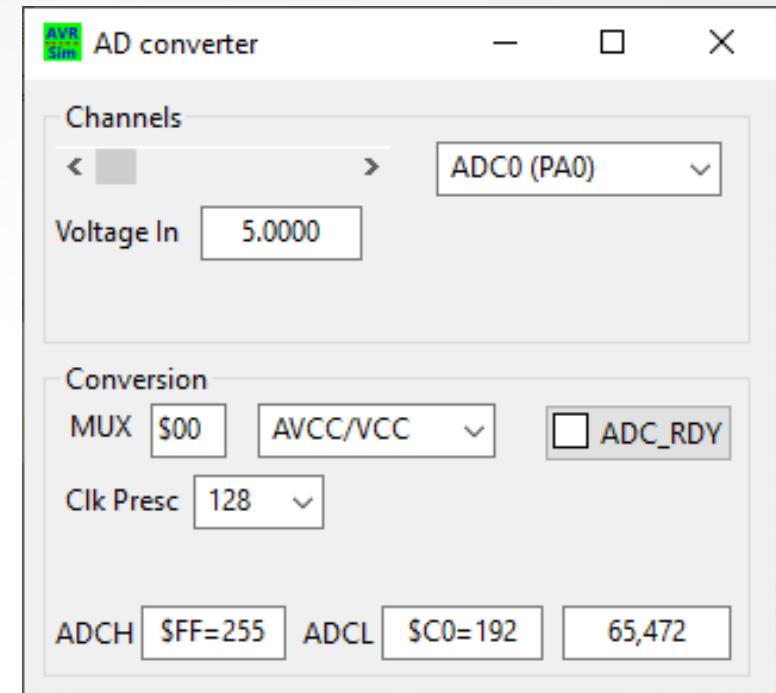
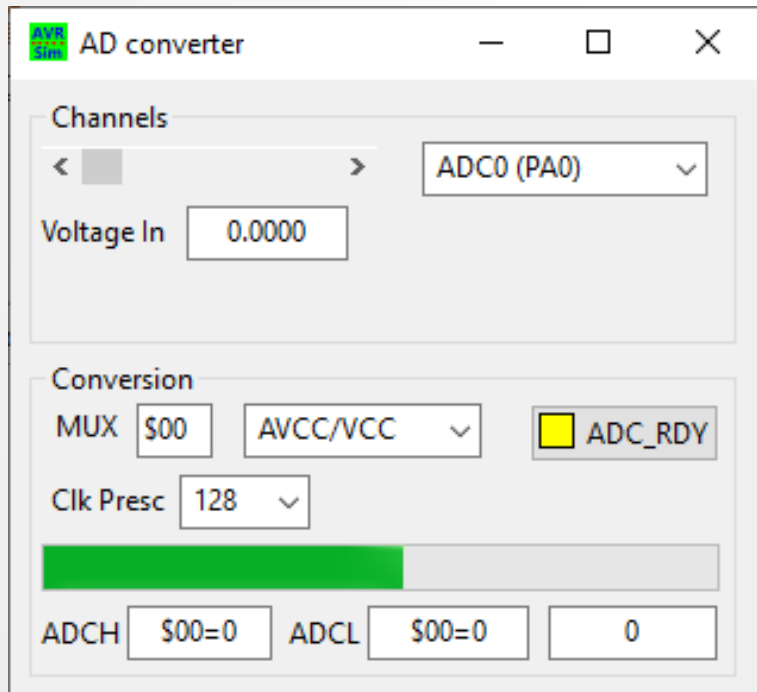
- Simulating this source code with `avr_sim` (click ADC to view the hardware) is very slow because of the clock prescaler at 128. Select Go/Run and set a breakpoint.
- Input the ADC0 voltage in the edit field before starting conversion.



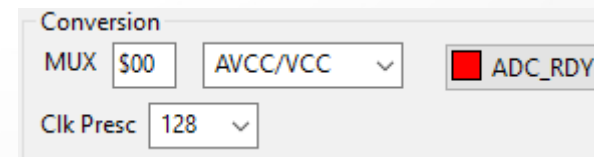
- Conversion can be followed with the progress bar.
- Conversion has completed and the result is in ADCH:ADCL as well as in R1:R0.
- The conversion lasted 1.67 ms.

Simulating ADLAR

- Simulating the source code with ADLAR set avr_sim version 2.2 does not yield the correct result. Version 2.3 (preliminary here) displays correct.
- Enabling interrupts by setting the ADIE bit, by initiating the stack, by setting the I flag and having an ISR:



- The ADC interrupt is enabled, but has not yet triggered.
- But after a while it is executed:



EEPROM

- All AVR's have EEPROM memory on board.
- EEPROM means Electrically Erasable and Programmable Read Only Memory. It is a storage type that conserves its content over many years, if not erased and/or re-programmed. It can be used to store manually adjustable settings.
- To store initial content in the EEPROM the EEPROM segment can be filled with a table:

```
; Table for the EEPROM  
.ESEG ; Switch assembly to the EEPROM segment  
.ORG 16 ; Start table at address 16  
.DB 1, 2, 3, 4, 5  
.DB "This is a null terminated text", 0
```

- The assembler will assemble that but will write the content to a file named [source-filename].eep in Intel-Hex-Format.
- The file's content can be written with a programmer to the EEPROM.

Reading EEPROM content

- Reading the content of the EEPROM is done via 3 port registers.
- First the address, from which the content is to be read, is written to EEARH:EEARL.

; EEPROM address register

LDI R16, High(EepAdr) ; Writing the EEP address, MSB

OUT EEARH, R16 ; to the high byte

LDI R16, Low(EepAdr) ; Dto., LSB

OUT EEARL, R16 ; to the low byte

- In devices with 256 bytes EEPROM or less EEARH is not implemented. If reading the content is done sequential and EEARH is not changed, it is not necessary to write it again.
- Next the read-enable-bit EERE in the control register EECR is set (SBI EECR, EERE). This halts the CPU for four cycles. The content read can be read from the data register EEDR (IN R16,EEDR).

Writing EEPROM content

- Writing a byte to EEPROM involves two stages: First the content has to be erased (this writes all bits at the location to ones), then the new data is written.
- First the address, to which the content is to be written, has to be written to EEARH:EEARL, similar to the read access. Then the byte to be written is written to the data register EEDR.
- Before a write operation can be initiated it has to be checked that previous write operations are terminated. This can be done by checking the program enable bit EEPE in the control register EECR:

```
; Wait for EEPROM write finished
```

```
EepWait:
```

```
SBIC EECR, EEPE ; Jump over next instruction if EEPE bit is clear
```

```
RJMP EepWait ; Not yet clear, continue waiting
```

- Then the Master Program Enable bit EEMPE in the control register EECR has to be written to one (SBI EECR, EEMPE).

Writing EEPROM content

- Within four clock cycles after setting EEMPE the program enable bit EEPE has to be written to one (SBI EECR, EEPE). If interrupts are enabled these have to be disabled temporarily (CLI) to ensure that this timing is not corrupted. After setting EEPE interrupts can be allowed again (SEI).
- EEPE will stay one as long as erasing and programming lasts. This lasts for 3.4 milli-seconds. In non-interrupt controlled write operations the EEPE bit can be accessed to find out, if writing is finished.
- As write operations last that long, the EE_RDY interrupt can be used to write more than one byte in a row. After starting the first EEMPE/EEPE operation the interrupt enable bit EERIE in EECR can be set, so that further write operations can be started in the Interrupt Service Routine ISR of the EE_RDY interrupt. As the EE_RDY interrupt is always triggered if the EEPE bit is clear, EERIE has to be disabled when not needed any more.

Writing EEPROM content

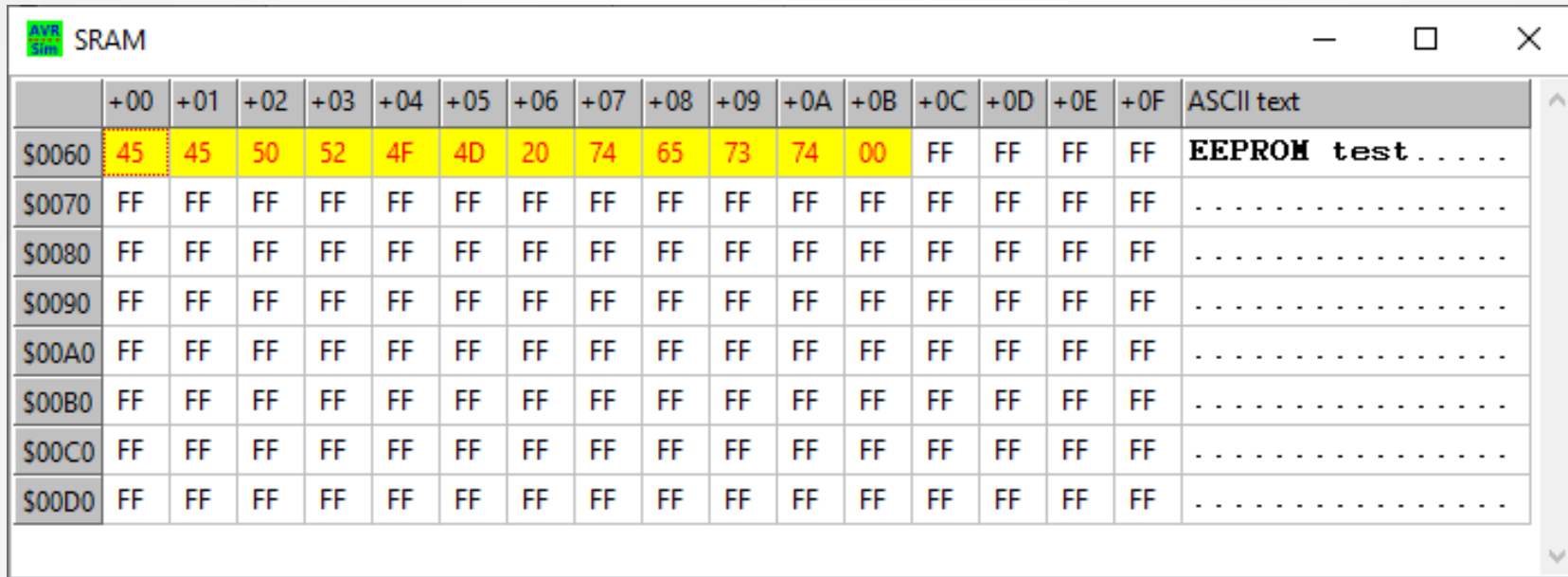
- The ISR looks like this:

```
; ISR for EEPROM ready interrupt
Eeplsr:
    IN rSreg, SREG ; Save SREG
    LD R16,X+ ; Read byte at address in XH:XL and auto-increment
    TST R16 ; Check Null character
    BREQ EeplsrEnd ; Yes, end is reached
    out EEDR, R16 ; Write to EEP data register
    ADIW ZL,1 ; Next address location in Z
    OUT EEARH, ZH ; Set EEPROM write address, MSB
    OUT EEARL, ZL ; dto., LSB
    SBI EECR, EEMPE ; Set master programming enable
    SBI EECR, EEPE ; Set programming enable bit
    RJMP EeplsrRet ; Jump to return
EeplsrEnd:
    CBI EECR, EERIE ; Disable EEP interrupt
EeplsrRet:
    OUT SREG, rSreg ; Restore SREG
    RETI
```

- The registers X points to the SRAM, Z to the EEPROM address.

Example writing EEPROM

- This example writes 10 bytes in the SRAM to the EEPROM at location 16. The following is the content of the SRAM:

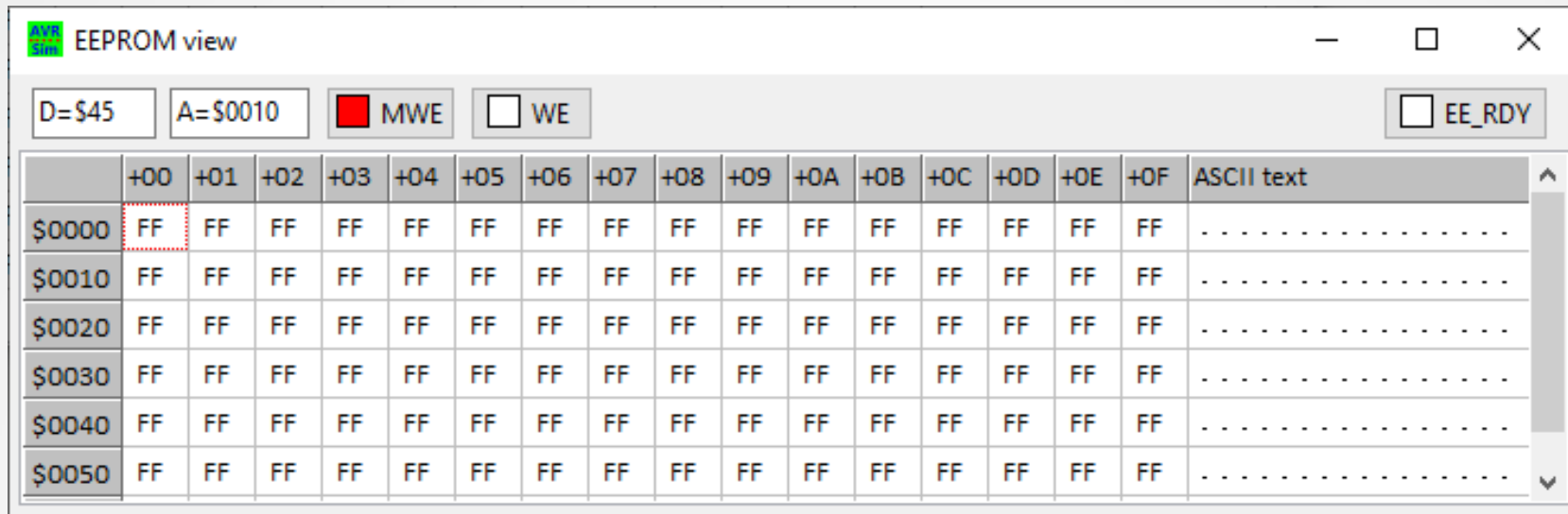


The image shows a screenshot of an 'AVR Sim' window titled 'SRAM'. It displays a memory dump with columns for addresses (+00 to +0F) and ASCII text. The first row, starting at address \$0060, contains the hexadecimal values 45, 45, 50, 52, 4F, 4D, 20, 74, 65, 73, 74, 00, followed by FF for the remaining three bytes. The corresponding ASCII text is 'EEPROM test.....'. The subsequent rows, from \$0070 to \$00D0, all contain FF for the hexadecimal values and empty space for the ASCII text.

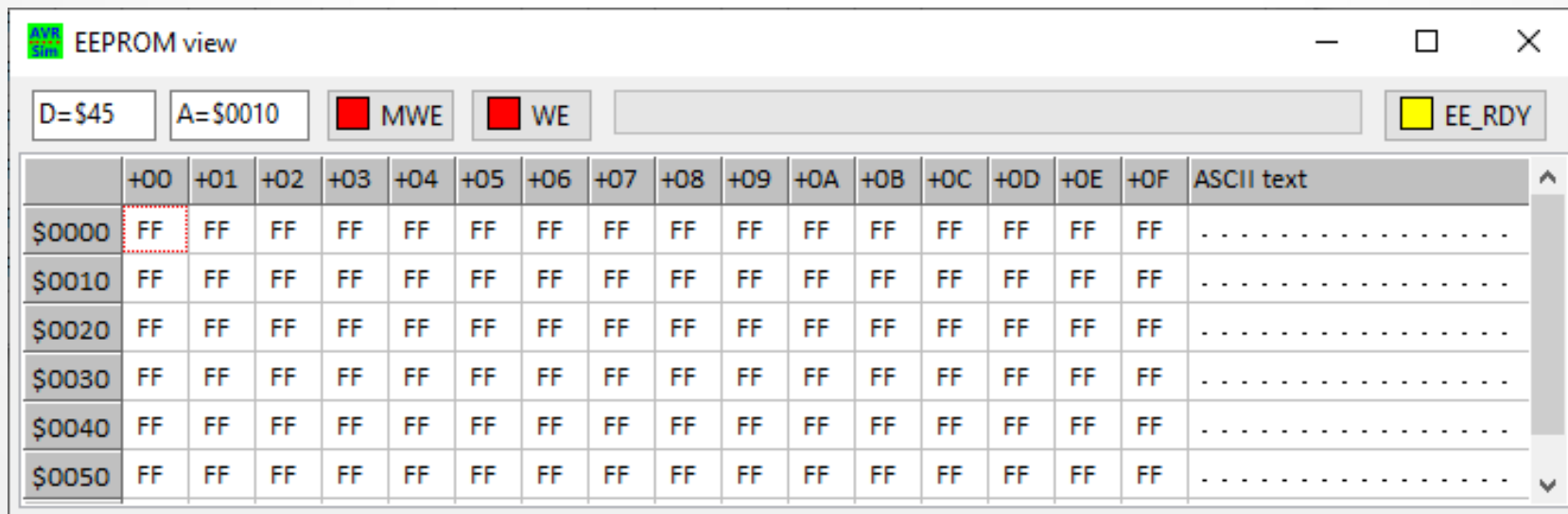
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	45	45	50	52	4F	4D	20	74	65	73	74	00	FF	FF	FF	FF	EEPROM test.....
\$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

- The data and the address are written, the master program enable bit is set.

Example writing EEPROM



- **Now the program enable and the interrupt enable bits are set:**



Example writing EEPROM

- The erasing time is over, the location 0x0010 is set to 0xFF.

[illegible]

- **Programming of the first byte is finished, the interrupt is starting**

[illegible]

Example writing EEPROM

- **The ISR has written the second byte.**

[illegible]

- **The complete text has been copied to the EEPROM, the int is off**

[illegible]

Example writing EEPROM

- The whole write process took 37.6 ms (see the stop watch).

The screenshot shows the AVR Studio simulation interface. The 'Simulation status' panel on the left displays various system metrics. The 'SREG' panel shows the status register with the 'I' (Interrupt Flag) bit set to 1. The 'Register' panel shows the contents of registers R0 through R24, with R8 containing 02 and R24 containing 6C. The 'Messages' panel shows the message '\$0000: Starting'. The 'Show internal hardware' panel at the bottom has checkboxes for Ports, Timers/counters, WDT, ADC, Scope, EEPROM, and SRAM, with EEPROM and SRAM checked.

Simulation status

Prog counter = \$000035
Instructions = 18,874
Stackpointer = \$00DF
Watchdog = 0.000000%
Clock frequ. = 1,000,000 Hz
Time elapsed = 37.726 ms
Stop watch = 37.623 ms
Sleep share = 0.000000%

SREG

I	T	H	S	V	N	Z	C
1	0	0	0	0	0	1	0

Update status Instructions: 1000
Step Delay ms: 10

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	02
R16	00	00	00	00	00	00	00	00
R24	00	00	6C	00	00	00	1A	00

Messages

\$0000: Starting

Show internal hardware

☐ Ports ☐ Timers/counters ☐ WDT ☐ ADC ☐ Scope ☒ EEPROM ☒ SRAM

- This would be a lengthy process if programmed without interrupts.
- Interrupts relieve the controller from running around in circles, waiting for completion.
- Interrupts use the controller only when really needed.

Questions and tasks in Lecture 8

Task 8-1: Write a program that measures continuously the voltage on the ADC0 pin and dims a LED on OCR0A according to that voltage.

Bonus question: What has to be done to increase the red portion with increasing voltages and to reduce the green portion on a two-color-LED?



Questions and tasks in Lecture 8 - Continued

Task 8-2: Write a program that counts the number of power-ons that the controller has had in his lifetime and blink a LED with this number.

Bonus question: Extend the program so that if the controller had more than five power-ups the blinking rhythm enhances its speed.



Questions and tasks in Lecture 8 - Continued

Task 8-3: Write the user-terrorizing software for a machine that measures the ambient temperature when first started and that fails to start if the temperature is smaller than this on every fourth start-up. Failure shall be signalled with a blinking red LED, succesful operation with a durable green LED.





Lecture 9: External interrupts

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

PC interrupts

- The INTn interrupts are on top of the interrupt vector jump list and have the highest priority.
- The ATtiny24 has only one external INT pin (PB2, pin 5 of the PDIP package), INT0.
- If you need more external interrupts: any pin can generate an interrupt. These interrupts are called PCINTn, in the ATtiny24 PCINT0 to PCINT7 are the interrupts generated on port pins PA and PCINT8 to PCINT11 on port pins PB.
- If enabled, any level change triggers a PCINT interrupt.
- To enable those external interrupts the respective PCINTn bits have to be set in the port registers PCMSK0 (PCINT0:7) or PCMSK1 (PCINT8:11). Any number of bits can be set or cleared.
- The interrupts can be enabled by setting the PCIE0 and/or PCIE1 bit in the GIMSK port register.

PC interrupts, pin identification

- Preferably only one of the PCINT0:7 and PCINT8:11 interrupts is enabled, so the attribution of the PCINT0 and PCINT1 to the pin is trivial.
- If more interrupts have to be utilized, the identification which of the pins has changed its level and triggered the PCINT0 or PCINT1 interrupt can use the Exclusive OR instruction EOR. EORing the current and the previous state of the port sets all bits that have changed.

```
.EQU PcInt0Mask = 0b01010101 ; PA0/PA2/PA4/PA6 can cause the PCINT0
.DEF rPrev = R17 ; Previous port state
.DEF rCurr = R18 ; Current port state
; PCINT0 ISR
    IN R15, SREG ; Save SREG
    IN rCurr, PINA ; Read current port A state
    ANDI rCurr, 255 - PcInt0Mask ; Clear all bits that are not masked
    MOV R16, rCurr ; Copy current state
    EOR R16, rPrev ; Exclusive OR the previous state
```

Pin identification, continued

```
SBRC R16, 0 ; Skip next instruction if bit 0 is clear
RCALL Bit0Changed
SBRC R16, 2 ; Test bit 2 changed
RCALL Bit2Changed
SBRC R16, 4 ; Test bit 4 changed
RCALL Bit4 Changed
SBRC R16, 6 ; Test bit 6 changed
RCALL Bit6Changed
MOV rPrev, rCurr ; Copy current over previous
OUT SREG, R15 ; Restore SREG
RETI ; Return from interrupt
```

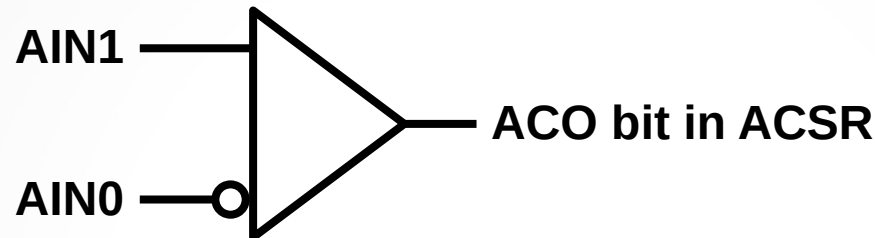
- The subroutines BitNChanged can react specifically on every changed input pin.
- If the PCINT0/PCINT1 interrupts shall react on pressed keys connected to the pins, it is sufficient to react on a low level on these pins. Skip on Bit in I/o Set SBIS can be used, followed by those RCALLs.

Suppression of key bouncing

- **Keys tend to bounce, maybe not when new but after ageing. So it is a good idea**
 - **to clear and start a timer if one of the keys has been pressed,**
 - **to disable further key reactions within the following 20 or 50 ms,**
 - **to restart the timer by clearing its TCNT port register if another active key action happens in between,**
 - **to enable key reactions again when the timer reaches its end point.**
- **Blocking needs a flag bit here. Use any bit in any register, with Set Bit in Register SBR and Clear Bit in Register CBR to set and clear this flag.**
- **If not used for other purposes, the T flag in SREG can be set with SET and cleared with CLT. Note that saving the SREG in interrupt service routines overwrites the T flag when restoring its state.**

The analog comparer

- One fifth of all AVR's have analog comparers on board. If enabled (ACD bit in ACSR is one), the analog comparer compares the analog voltages on the pins AIN1 and AIN0 and set the ACO bit in ACSR according to the result.



Bit	7	6	5	4	3	2	1	0	
0x08 (0x28)	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

- If the ACIE bit is set, an interrupt can be generated. If the ACS1:0 bits are 0b00 this is done on every level change. The ACS bits can enable interrupts only on falling edges (0b10) or on rising edges (0b11).

Serial communication

- Half of the AVR devices have serial communication hardware on board.
- Two types of serial communication are supported:
 - Synchronous interfaces such as the Two-Wire Interface (TWI) or I2C, see Universal Serial Communication (USC) in the data book,
 - Asynchronous interfaces such as Universal Asynchronous Receiver and Transmitter (UART).
- All serial interfaces can initiate diverse interrupt types.
- Use `avr_sim` („Project“, „New“, „Device-selector“) to find all AVR types that have such serial interfaces by selecting serial pins (SDA0 or UXD0).

Questions and tasks in Lecture 9

Task 9-1: Write a program that follows any changes on the INT0 pin and lights a LED if the input is low.

Bonus question: What has to be changed if the LED lights on the input high?



Questions and tasks in Lecture 9 - Continued

Task 9-2: Write a program that counts the key bounces on PB0 and display those on four LEDs attached to PA0 to PA3.

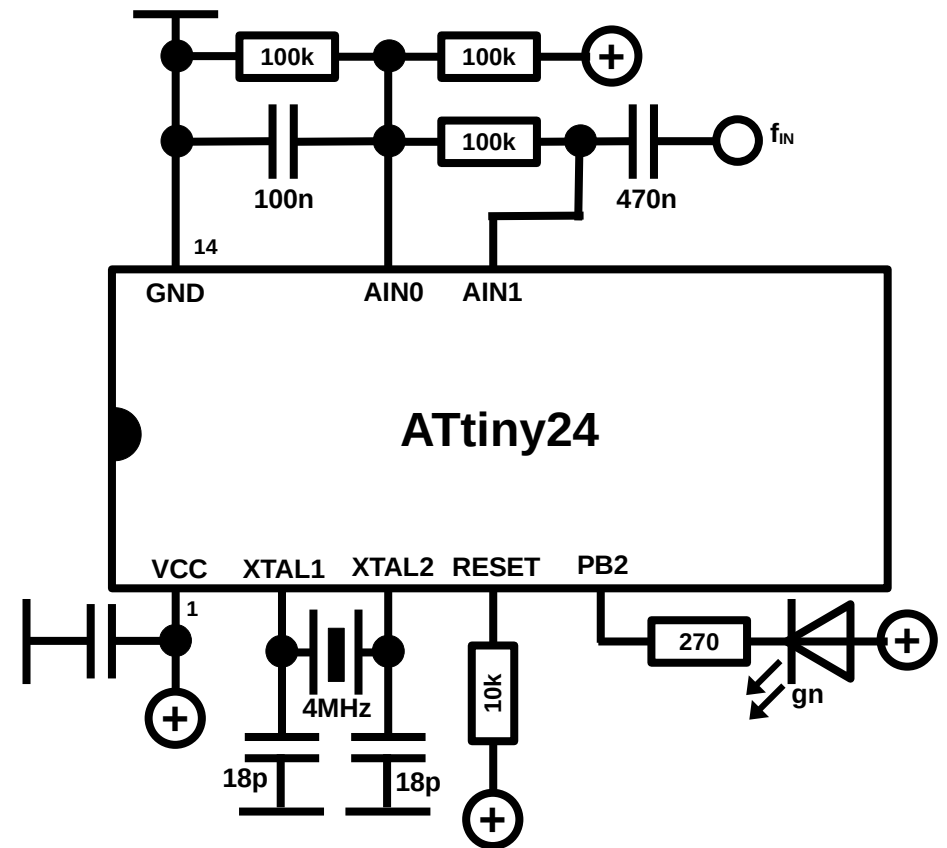
Bonus task: Clear the display if the PCINT input pin has been inactive for longer than five seconds.

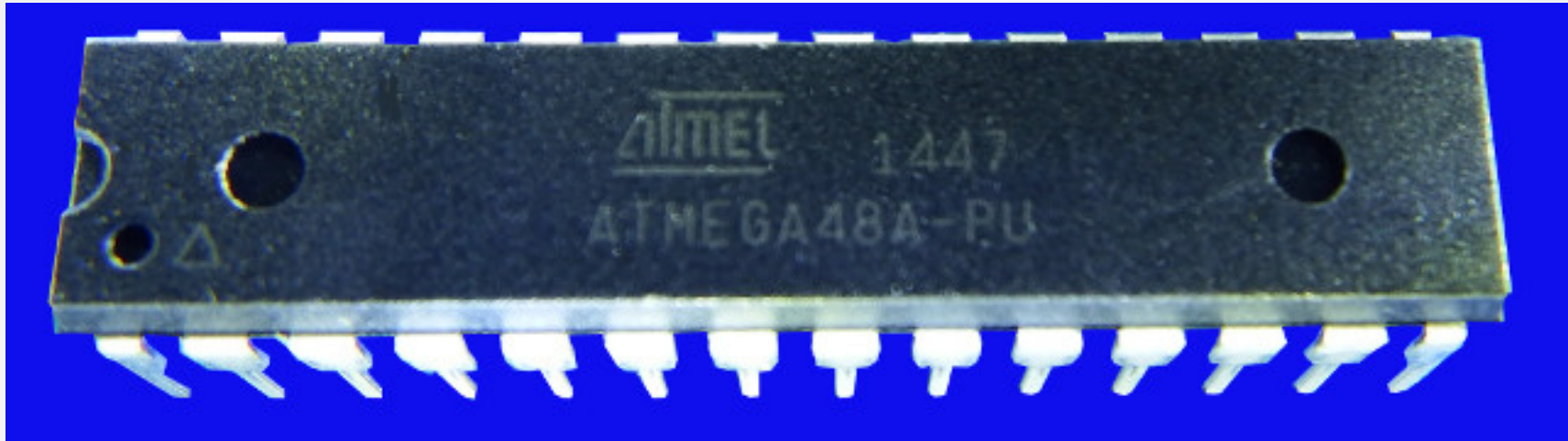


Questions and tasks in Lecture 9 - Continued

Task 9-3: Build this hardware and measure the frequency on the input with the analog comparer. If this is between 49 and 51 Hz, light the green LED.

Touch the f_{IN} connector to test the hard- and software.





Lecture 10: Assembler math

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Assembler math

- **Assembler math is special because it requires writing all routines by yourself using the available instructions that the CPU understands. While in higher-level languages this is already implemented as part of the language, assembler doesn't have this. The advantage is that your routines are tailored exactly to your needs, you do not waste memory space and unnecessary execution time.**
- **Assembler math is binary and therefore simpler than decimal math: only zeros and ones have to be considered.**
- **I'll discuss 8 and 16 bit addition and subtraction here as well as 8x8 and 16*8 bit multiplication (in software and in hardware versions) and 8/8 and 16/8 bit division. If you need other math, see if this website [here](#) provides examples.**
- **At the end I'll demonstrate 8- and 16-bit binary-to-decimal conversion.**

8-by-8 bit addition and subtraction

- Adding two 8 bit numbers is simple: the mnemonic for that is ADD.

; Adding two registers

LDI R16, 128 ; Load 128 to a register

LDI R17, 64 ; Load 64 to a second register

ADD R16, R17 ; Add the second register, result in the first register

- If the result is larger than 255, the carry flag C in SREG is set. If the result is zero (= 256) the Z flag in SREG is additionally set.
- Similarly subtraction is also simple: the mnemonic for that is SUB.

; Subtracting two registers

LDI R16, 128 ; Load 128 to a register

LDI R17, 64 ; Load 64 to a second register

SUB R16, R17 ; Subtract the second register, result in the first register

- The carry flag is set if the second number is larger than the first, and the Z flag if both are equal.

16-by-8 bit addition and subtraction

- Adding an 8 bit number to a 16 bit number involves the carry flag. The mnemonics for that are ADC and SBC (add or subtract with carry).

```
; Adding an 8-bit register to a 16 bit register
LDI R16, 128 ; Load 128 to a register as LSB
LDI R17, 0 ; Load 0 to the MSB
LDI R18, 240 ; Load 240 to the 8 bit register
ADD R16, R18 ; First add the 8 bit register to the LSB
BRCC NoCarry ; Branch if carry flag is clear
INC R17 ; Add the carry to the MSB
NoCarry:
```

- An alternative to branching is to write zero to the 8-bit register and to use the ADC instruction. This adds zero plus the carry flag to the MSB.
- Do not use CLR for writing zero to the 8-bit register, that would clear the carry flag, too.

16-by-16 bit addition and subtraction

- Adding a 16 bit number to a 16 bit number is also simple:

; Adding a 16-bit number to a 16 bit number

LDI R16, Low(1028) ; Load the LSB of 1028 to a register

LDI R17, High(1028) ; Load the MSB

LDI R18, Low(32700) ; Load the LSB of 32700 to a register

LDI R19, High(32700) ; Load the MSB

ADD R16, R18 ; First add the two LSB registers

ADC R17, R19 ; Then add the MSBs and the carry

- This can be extended to more bits: just use ADC for all higher bytes.
- Subtraction is similar: Use SUB for the LSB and SBC for all higher bytes to be subtracted with the carry.
- What if the numbers are signed? Signed numbers use bit 7 of the highest byte as sign bit: if it is zero, the number is positive, if it is one, the number is negative and the number has been subtracted from 256 (8 bit) or 65,536 (16 bit).

Addition of negative numbers

- As negative numbers are stored as their negative value (256 – number) adding two numbers, of which the second is negative, is simple:

Register	Decimal	Signed	Binary
R16	100	0x64	0b0110.0100
R17	-20	0xEC	0b1110.1100
ADD R16, R17			
R16	80	0x50	0b0101.0000

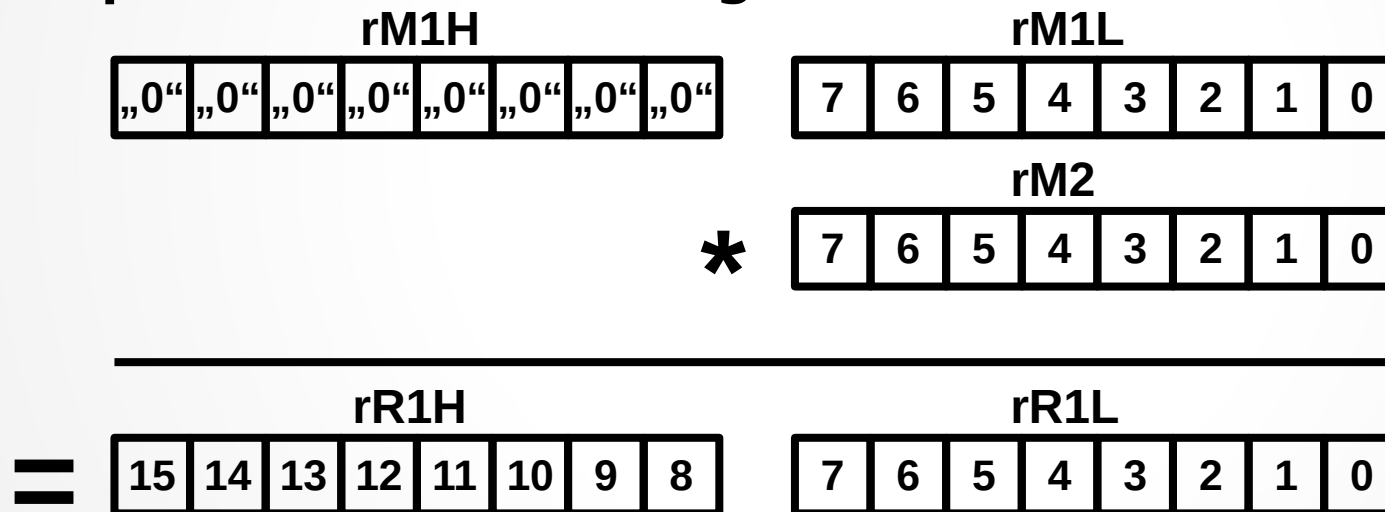
- Nothing has to be corrected.
- In assembler negative 16-bit numbers are handled like this:

```
; A negative number in Z
LDI ZH, High(-20) ; MSB of the negative number
LDI ZL, Low(-20) ; LSB of the negative number
```

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	EC	FF

Multiplication 8 x 8

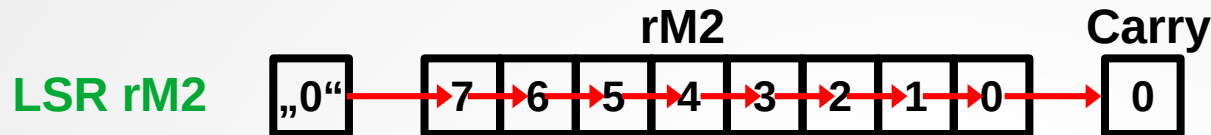
- Multiplication of 8 bits by 8 bits can yield a 16 bit result. It can be done in two ways:
 1. by software,
 2. by hardware (in ATmega and in some advanced ATtiny).
- Multiplication in software goes like this:



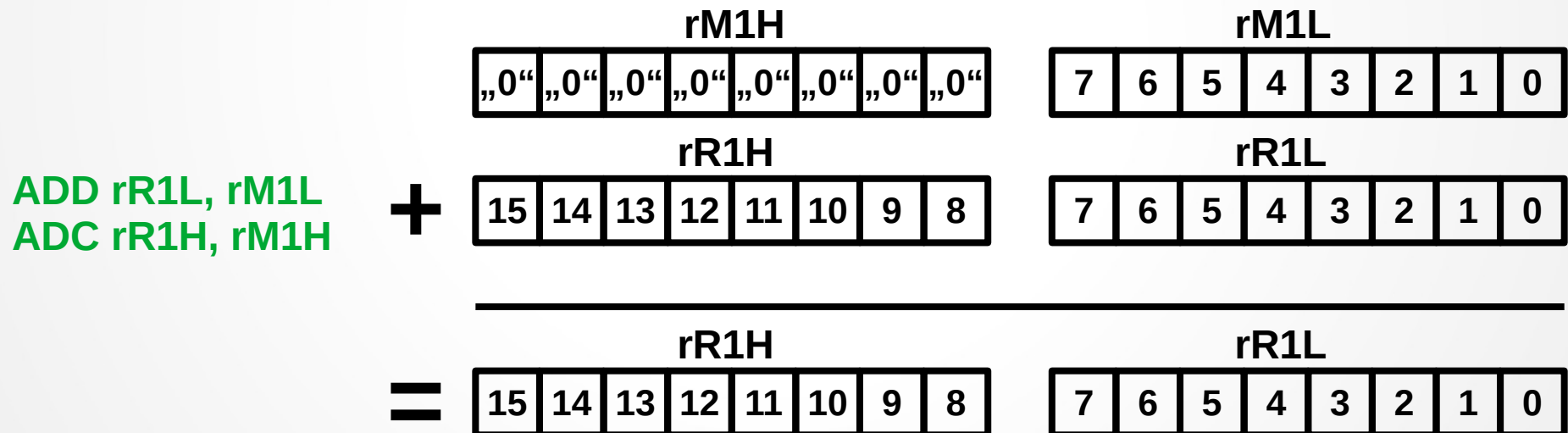
First, rM1L and rM2 are loaded with the numbers to be multiplied. rM1H as well as the result registers rR1H:rR1L are cleared.

Software multiplication 8 x 8

- In the first step rM2 is shifted right.



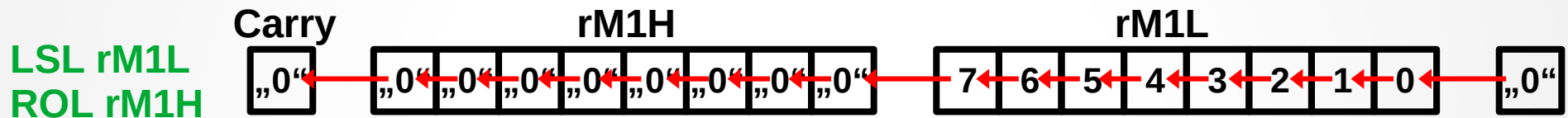
- That shifts a zero to its bit 7, shifts its bits 7 to 1 one position to the right and its former bit 0 to the carry flag.
- If the carry flag is one, it adds rM1H:rM1L to the result registers:



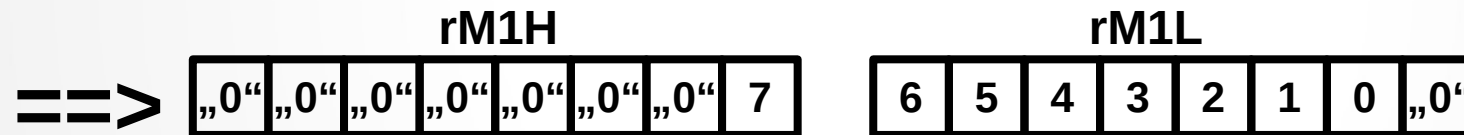
If not (BRCC), no addition is done and it is jumped over.

Software multiplication 8 x 8

- Now it is tested (TST rM2) if the multiplier is already clear. If that is the case, multiplication is done and the result is already final and correct.
- If not the multiplicator rM1H:rM1L is multiplied by two, shifting rM1L one left and rotating the carry flag into rM1H.



rM1H:rM1L now look like this:



- These steps are repeated until all ones in rM2 are treated and rM2 is empty.

Multiplication 8 x 8

- The process, as simulated with the maximum numbers 0xFF, consumes 82 μ s and the result in R1:R0 is correct:

The screenshot shows the AVR Simulator interface. The 'Simulation status' panel on the left displays the following information:

- Prog counter = \$00000E
- Instructions = 74
- Stackpointer = \$0000
- Watchdog = 0.00000%
- Clock frequ. = 1,000,000 Hz
- Time elapsed = 82.0 μ s
- Stop watch = 82.0 μ s
- Sleep share = 0.00000%

The 'SREG' panel shows the status register bits:

I	T	H	S	V	N	Z	C
0	0	1	0	0	0	1	0

The 'Register' panel shows the values of registers R0, R8, R16, and R24:

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	01	FE	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	80	7F	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

The 'Update status Instructions' field is set to 1000, and the 'Step Delay ms' field is set to 10.

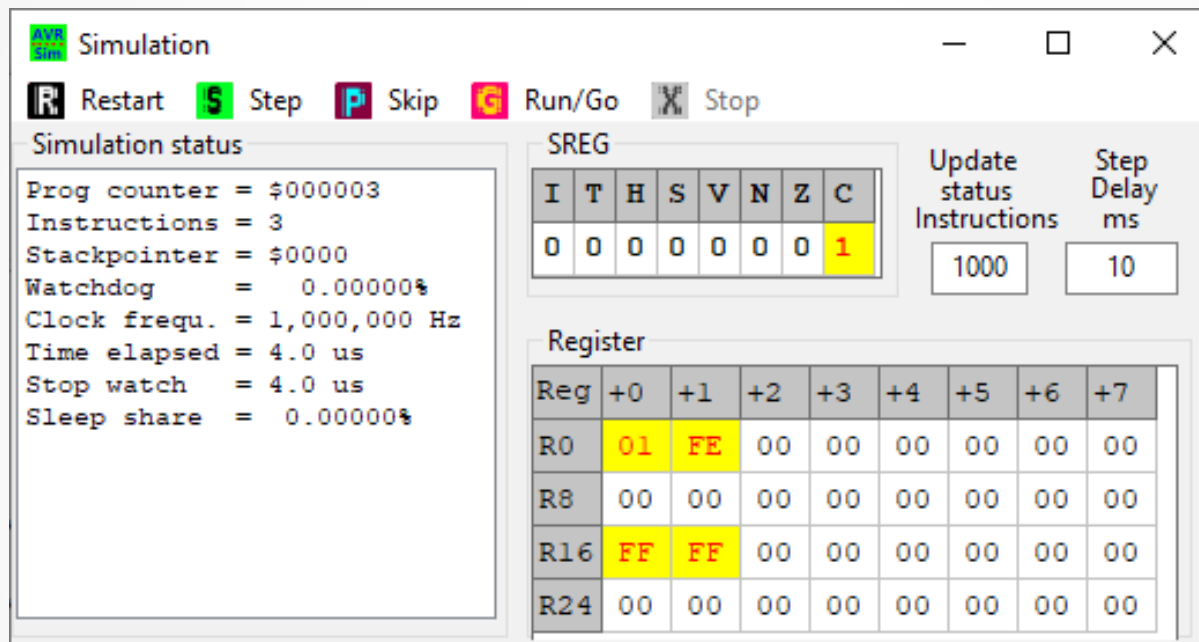
- Therefore, hardware multiplication is unnecessary unless you cannot afford this short period. So the ATtiny24, that has no hardware multiplication on board, is sufficient doing software multiplication.

Multiplication 16 x 8

- Extending this scheme to multiplying 16 bits by 8 bits is simple: just add another rM1 register to the left, such as rM1S for super, and add another result register to the left. Now addition requires an additional ADC and shifting an additional ROL. That is it.
- The same with 24 by 8 or 32 by 8 or 64 by 8. Add registers that the AVR has plenty of.
- Extending the scheme to 16 by 16 is also simple: add another rM2H to the left and add two super registers to rM1 and rR1 (the result can now have $16 + 16 = 32$ bits). Shifting the lowest bit to carry involves an additional LSR and the previous LSR is a ROL now. Addition adds two ADCs, multiplication of rM1 by two now has then two additional ROLs.
- If you have understood how the 8-by-8 multiplication works, you can do whatever multiplication.

Hardware multiplication 8 x 8

- The hardware multiplication is even simpler and faster: just add the instruction MUL rM1, rM2 (if your AVR type has that) and the result is in R1:R0.



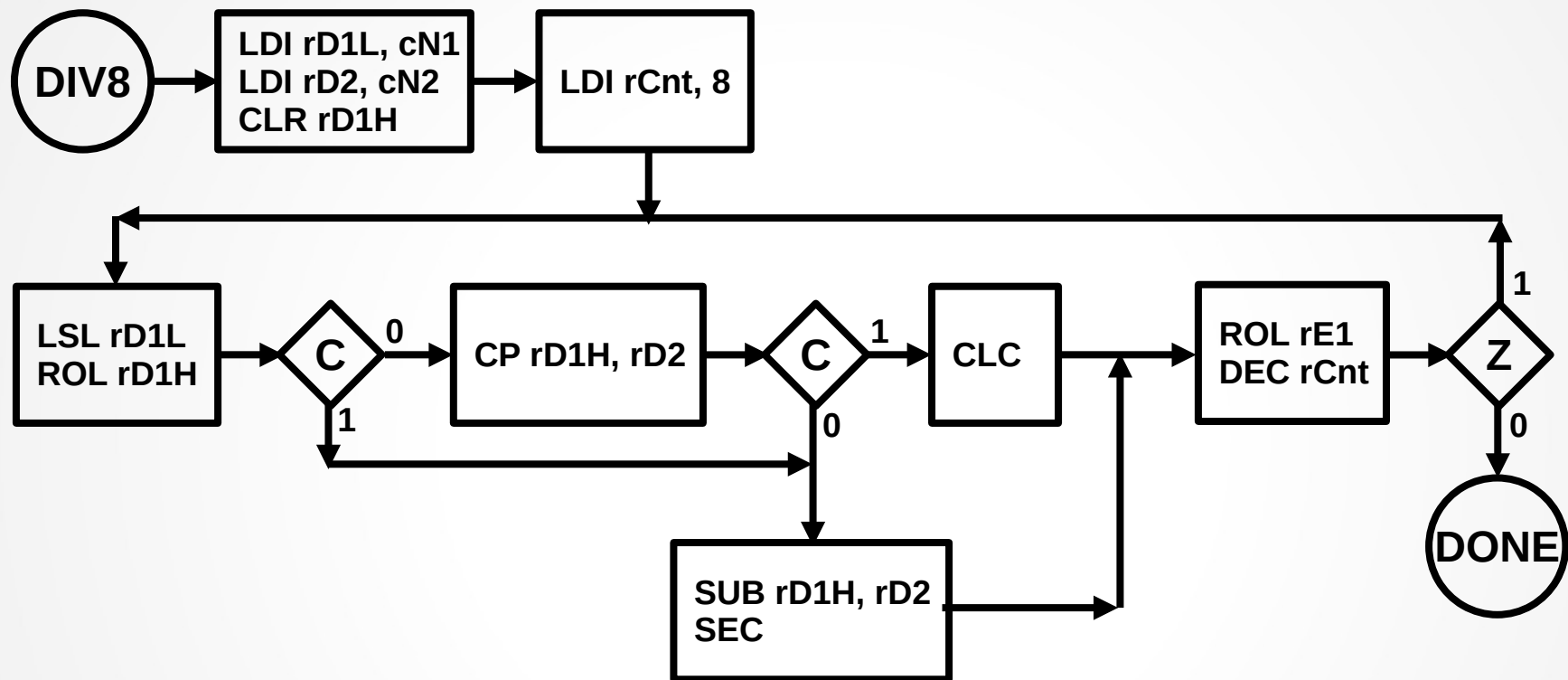
- Setting the two registers to 0xFF and MUL needs only 4 μ s, if the hardware multiplicator is used.

Hardware multiplication 16 x 8

- Extending the hardware multiplication to 16 by 8 bits is not that simple, as it requires not only one step but two.
- The 16 bit registers $rM1H:rM1L * rM2$ have to be multiplied using the following scheme:
 1. Multiply $rM1L$ with $rM2$ and copy the result in $R1:R0$ to the result registers in $rR1S:rR1H:rR1L$ to $rR1H:rR1L$ and clear $rR1S$.
 2. Multiply $rM1H$ with $rM2$ and add the result in $R1:R0$ to the result registers $rR1S:rR1H$ (the result registers by that are multiplied by 256).
- Extending that scheme to 16-by-16 involves four multiplications:
$$rM1H:rM1L * rM2H:rM2L = rM1L*rM2L + 256*rM1H*rM2L + 256* rM1L*rM2H + 65536*rM1H*rM2H$$
The multiplication with 256 and 65536 is done by just adding $R1:R0$ to the respective and correct upper registers.

Division 8 x 8

- Division is also rather simple. It starts with loading the registers and setting a counter to 8.



Then the divider registers are multiplied by 2. If the carry is one, the high byte of the divider is surely larger than the divisor: the divisor is subtracted from the high byte and a one is left-rotated into the result. If not, the high byte is compared with the divisor.

Division 8 x 8

- If the divisor is larger than the high byte (carry after compare is set), the carry is cleared and left-shifted into the result. If not, the divisor is subtracted from the high byte and a one is left-shifted into the result.
 - In all cases the counter is decremented. If he reaches zero, the division is done. If not is the cycle repeated.
 - Dividing 255 by 5 requires 99 μ s, slightly longer than multiplication. The result in R19 is 0x33, which is decimal 51 and obviously correct.
 - Note that the result is always an integer, not a fractioned number.
-
- The screenshot shows the AVR Studio simulation interface. On the left, the 'Simulation status' panel displays various system parameters:
- Prog counter = \$000010
 - Instructions = 84
 - Stackpointer = \$0000
 - Watchdog = 0.00000%
 - Clock frequ. = 1,000,000 Hz
 - Time elapsed = 99.0 us
 - Stop watch = 99.0 us
 - Sleep share = 0.00000%
- On the right, the 'SREG' register is shown with its bits: I=0, T=0, H=1, S=0, V=0, N=0, Z=1, C=0. Below it, the 'Register' file shows the contents of registers R0 through R24. Register R16 contains the value 05, and register R17 contains the value 33, representing the result of the division in hexadecimal.

[illegible]

Division of larger numbers

- **Extending division to 16 by 8 requires**
 1. **extension of the divider to three bytes,**
 2. **extension of the result to two bytes, and**
 3. **16 division steps.**
- **Extension to 16 by 16 requires additionally the extension of the divisor to two bytes.**
- **As general rules: the divider always has to be extended by the length of the divisor in bytes, the counter always has to be set to the length of the divisor in bits.**
- **Note that in division the first step is always the multiplication of the divider by two, while in multiplication this step is done as last step.**
- **If you need rounding: if the next division step yields a one, add 1.**

Conversion of a binary to hexadecimal

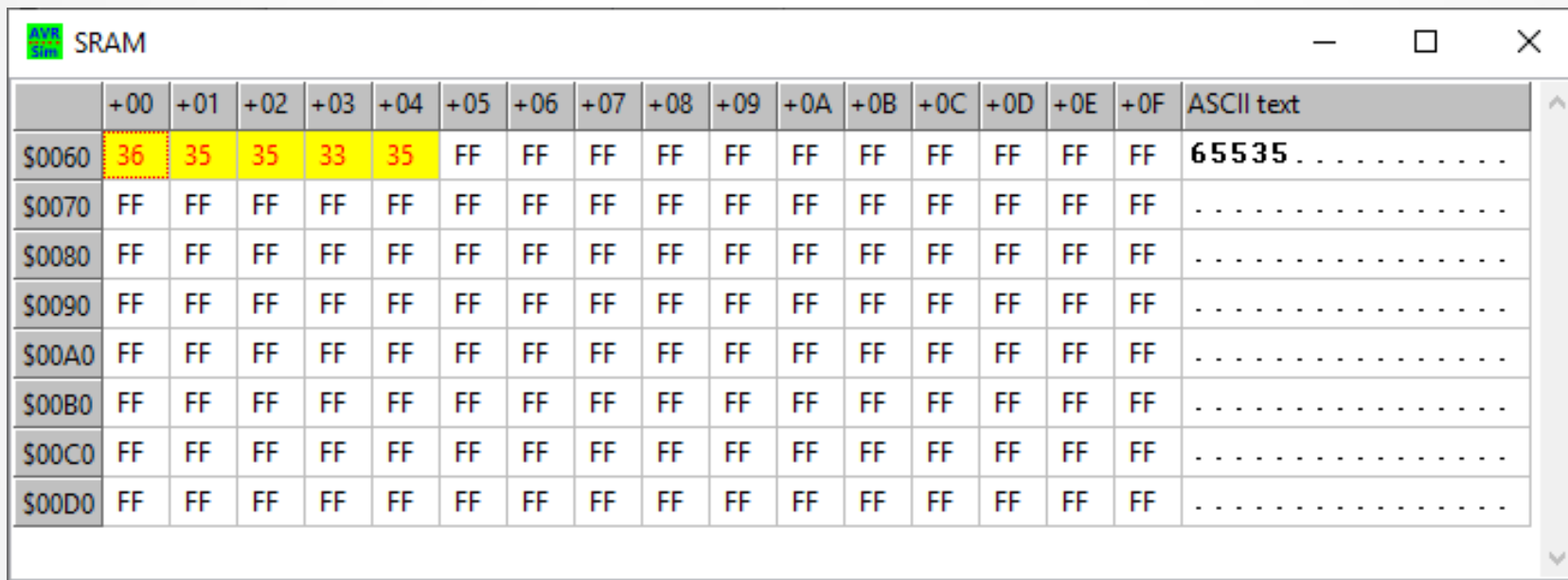
- If you need conversion of a byte to a hexadecimal format to display those (e.g. on a LCD) first convert the upper four bits (upper nibble), then the lower four bits. Use the instructions
 - **SWAP R16** to exchange both nibbles in R16,
 - **ANDI R16, 0x0F** to isolate the lower nibble in R16,
 - **SUBI R16, -'0'** to convert the binary to a hexadecimal character, and
 - **SUBI R16, -7** in case that the binary nibble is larger than '9'.
- It is convenient to **PUSH R16** to the stack, then to **SWAP** and **RCALL** the nibble conversion, then to **POP** the original and doing nibble conversion.
- If two or more bytes have to be displayed, copy those to R16 and **RCALL** the byte conversion. (See the hex conversion [here](#)).

Conversion of a binary to decimal

- If you need conversion of a byte to decimal format to display those (e.g. on a LCD) the following algorithm can be used.
- First subtract binary 100 (0x64) until a carry occurs. These are the hundreds. Add 100 to undo the last subtraction.
- Then subtract binary 10 (0x0A) until a carry occurs, which are the tens. Add 10 to undo the last subtraction.
- The remaining rest of the number is the last digit.
- If you need zero suppression to blanks set a flag (e.g. T in SREG) and exchange zeroes by blanks, as long as the flag is set. If a non-zero digit occurs clear the flag. Do not use the flag with the last digit.
- Extending the conversion to 16 bit numbers: start with binary 10,000, then with binary 1,000 and the lower ones and use two bytes for subtraction.

Conversion of binaries to decimal

- As an example: **here** is the source code of the decimal conversion program that uses a table with the decimal binaries to write the binary to SRAM.
- The picture shows the conversion of hexadecimal FFFF to decimal.



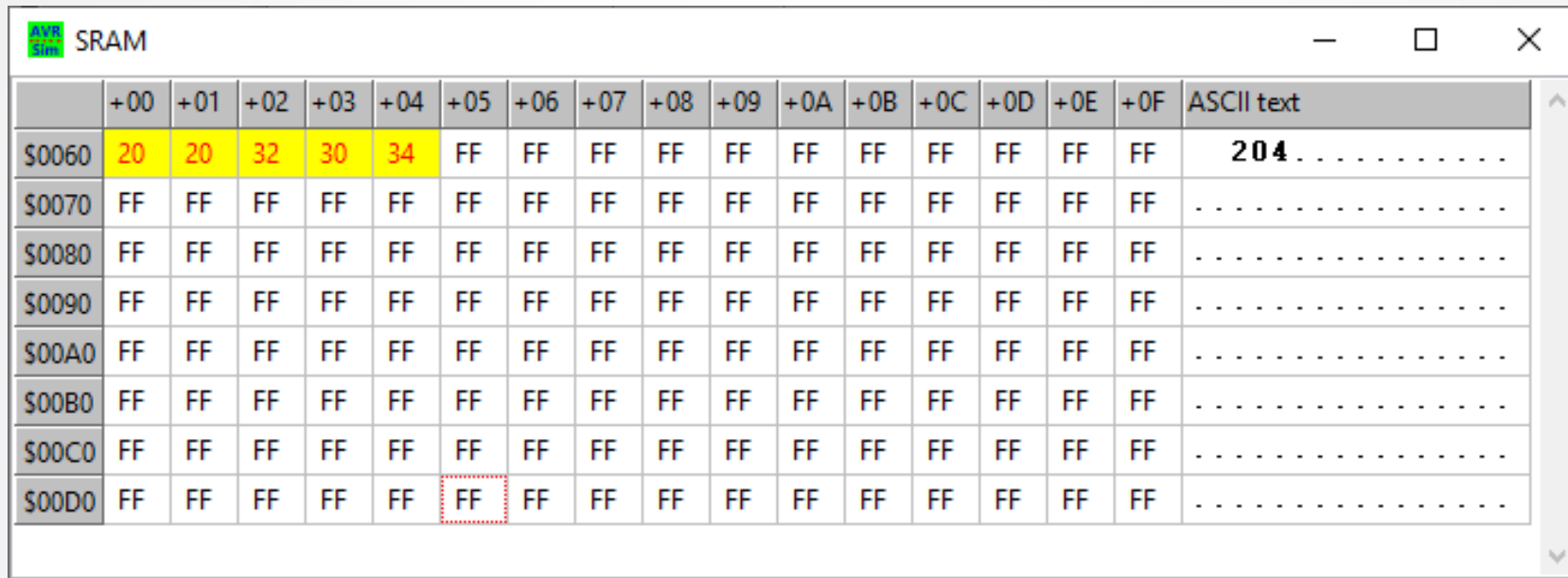
The screenshot shows a window titled "SRAM" with a table of memory addresses and their contents. The table has 17 columns: 16 columns for hexadecimal values (labeled +00 to +0F) and one column for ASCII text. The first row, at address \$0060, shows the hexadecimal values 36, 35, 35, 33, 35, followed by FF for the remaining 11 columns. The corresponding ASCII text is "65535". The other rows show FF in all 16 hexadecimal columns and empty ASCII text.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	36	35	35	33	35	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	65535
\$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

- The execution lasted 206 μ s.

Conversion of binaries to decimal

- This is the result if 0x00CC is converted. Leading zeroes are blanked, later appearing zeros are not blanked.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	20	20	32	30	34	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	204.....
\$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

- In that case conversion needed 127 μ s.

Questions and tasks in Lecture 10

Task 10-1: Write a program that sets and then adds and subtracts two 24-bit numbers.

Bonus question: How many registers would a 64-by-64 bit addition need and what would be the largest numbers than can so be added/subtracted? Can all living individuals in the world be tagged with a unique 64 bit number? Who would then get number 0 in your view?



Questions and tasks in Lecture 10 - Continued

Task 10-2: Write a program that sets and then multiplies two 16-bit numbers.

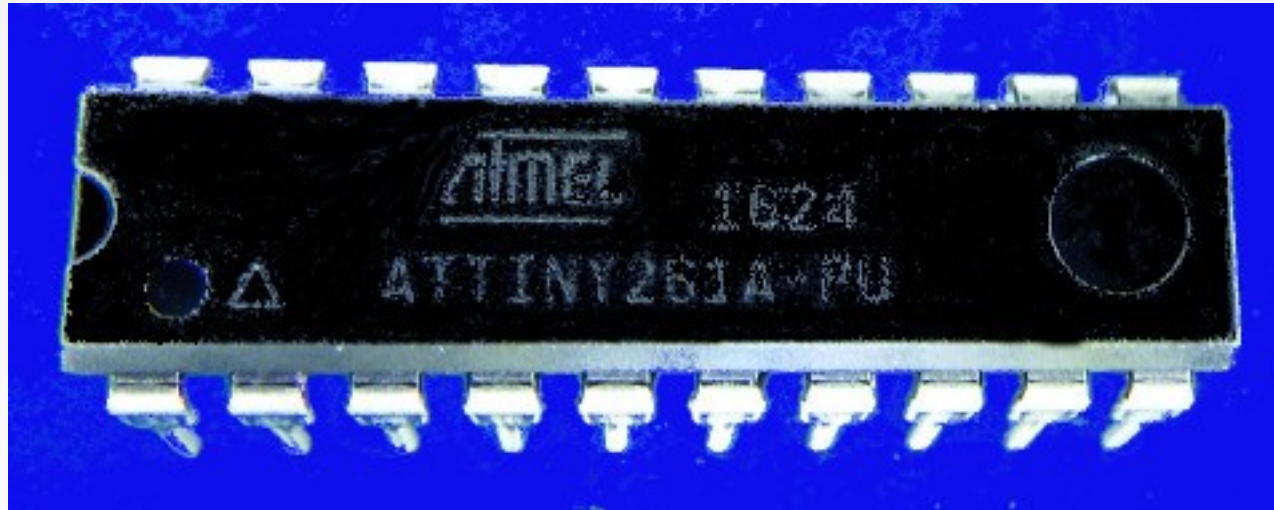
Bonus question: How many registers would be needed to multiply two 64-bit numbers? Can that be handled in the 32 registers in AVR?



Questions and tasks in Lecture 10 - Continued

Task 10-3: Convert the number 12,345,678 from binary to decimal with leading-zero-blanking.





Lecture 11: Tables and accesses

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by

Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Table access

- Tables can be used to store fixed values.
- Those can be located in the flash, in the SRAM or in the EEPROM memory.
- To store a table in the flash memory, use the following source code:

```
; A table with bytes and one with words  
MyByteTable: ; The table needs a label to be accessible  
.DB 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; A byte-wise table, bytes separated with comma  
MyWordTable: ; A label for the word table, words separated by comma  
.DW 1000, 2000, 3000, 4, 5, 6
```

- As the flash memory is organized as 16-bit words, the bytes in MyByteTable: are byte-wise packed as words. The first byte goes to the LSB, the second to the MSB, therefore the first word is 0x0201.
- The MyWordTable: is already organized in 16-bit word format, so each MSB:LSB word creates one flash memory location. In case that the number has MSB=0 (4, 5 and 6 in the example), the MSB is nevertheless stored (0x0004, 0x0005 and 0x0006).

Table access

- When reading from this table,
 - two read operations have to take place, and
 - the least significant bit of the read address decides whether the LSB or the MSB at this address shall be accessed, and
 - reading requires the double register ZH:ZL, which is R31:R30, for the address, and
 - this address has to be shifted one bit position left to gain space for the MSB/LSB access bit.
- Reading one byte from the table MyByteTable: in the flash memory:

```
; Reading the fifth byte from MyByteTable  
LDI ZH, High(2*MyByteTable + 4) ; Point to the fifth byte entry, MSB  
LDI ZL, Low(2*MyByteTable + 4) ; dto., LSB  
LPM R16, Z ; Read byte at that address to R16
```

Table access auto-inc/dec

- If LPM does not specify the register and Z (just pure LPM) it reads the byte at Z to R0.
- As accessing memory with LPM is very often sequentially:
 - LPM R16, Z+ auto-increments Z (adds 1 to Z wordwise) after reading,
 - LPM R16, -Z first subtracts one from Z and reads the byte thereafter.
- Access to calculated table addresses can be done with adding the displacement first:

; Reading the fifth byte from MyByteTable

LDI ZH, High(2*MyByteTable + 4) ; Point to the fifth byte entry, MSB

LDI ZL, Low(2*MyByteTable + 4) ; dto., LSB

ADD ZL, R16 ; Displacement in R16

LDI R16, 0 ; MSB displacement (LDI does not affect flags, CLR would also clear C)

ADC ZH, R16 ; Add carry flag to MSB

LPM R16, Z ; Read byte at that address displaced by R16

Table access word-wise

- If the table is organized wordwise: LPM has to be done twice to read LSB and MSB.
- As an example: with a table that holds addresses, access to the third address can be done as follows:

; Reading the third address from MyWordTable

LDI R16, 3 ; The displacement of the address in the address table to be copied

LDI ZH, High(2*MyWordTable) ; Point to the beginning of the table, MSB

LDI ZL, Low(2*MyWordTable) ; dto., LSB

LSL R16 ; Double the 3 in R16 because each address is 16 bits long

ADD ZL, R16 ; Adding the displacement in R16 to the base address, LSB

LDI R16, 0 ; MSB of the displacement

ADC ZH, R16 ; Add carry flag to MSB

LPM XL, Z+ ; Read byte at that address displaced to double register X, LSB

LPM XH, Z ; Read the MSB

- Another example: To convert a 16-bit binary to decimal one needs the decimals of 10,000, 1,000, 100 and 10 in binary form:

; My decimal table

DecTab:

.DW 10000, 1000, 100, 10, 0 ; The decimal table as binary words

Decimal conversion

- With that table the decimal conversion of a 16-bit binary in R1:R0 to a decimal in the SRAM goes as follows:

; Converting the binary in R1:R0 to decimal in SRAM

DecConv:

LDI XH, High(SRAM_START) ; Point X to SRAM start, MSB

LDI XL, Low(SRAM_START) ; dto., LSB

LDI ZH, High(2*DecTab) ; Point Z to decimal tab, MSB

LDI ZL, Low(2*DecTab) ; dto., LSB

DecConvDec:

LPM R2, Z+ ; Read decimal value to R2, LSB, and auto-increment Z

LPM R3, Z+ ; dto., Read MSB to R3

TST R2 ; Is the LSB zero?

BREQ DecConvFinished ; Yes, decimal conversion is complete

SER R16 ; Count how often the decimal can be subtracted, start with -1

DecConvSubtr:

INC R16 ; Count number of subtractions up

SUB R0, R2 ; Subtract the decimal binary from the LSB

SBC R1, R3 ; Subtract the MSB and the carry flag

BRCC DecConvSubtr ; No carry has occurred, continue subtraction

ADD R0, R2 ; Undo the last subtraction, LSB

ADC R1, R3 ; dto., MSB

SUBI R16, -'0' ; Add an ASCII Null (decimal 48)

ST X+, R16 ; Copy result byte to SRAM and auto-increment

RJMP DecConvDec ; Continue with the next decimal binary

Decimal conversion - continued

- Now the last digit has to be prepared:

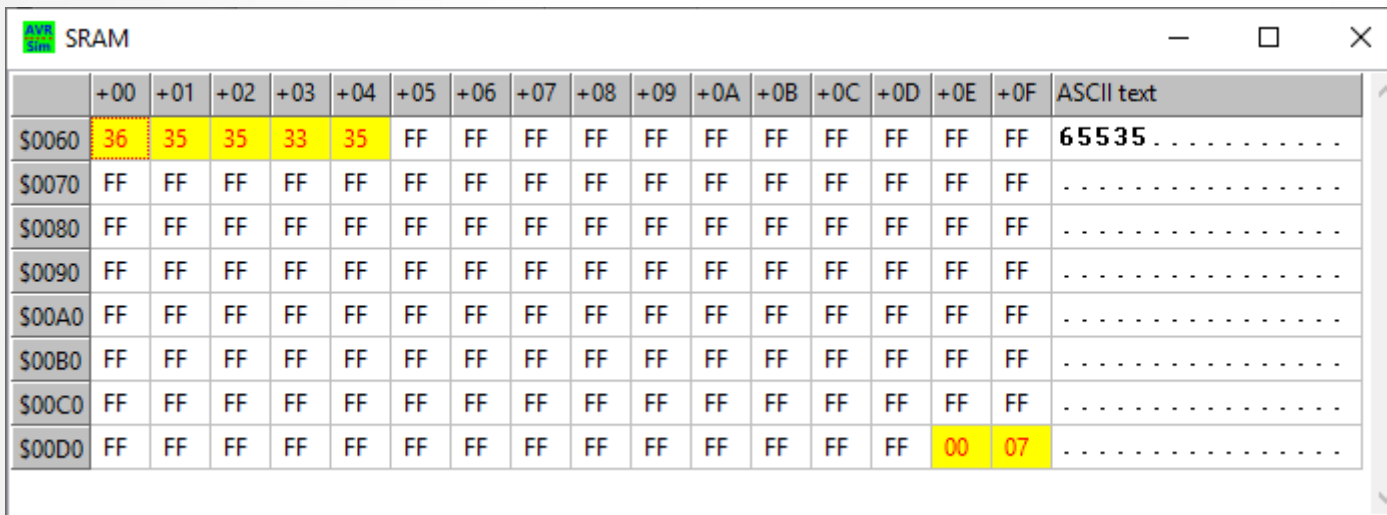
DecConvFinished:

LDI R16, '0' ; Load ASCII 0 to R16

ADD R16, R0 ; Add the last digit

ST X, R16 ; Write the last digit to SRAM

- This is what the SRAM looks like if 0xFFFF is converted to decimal.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	36	35	35	33	35	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	65535.....
\$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	07	

- As there are leading zeroes, if the number is smaller than 10,000, those have to be converted to blanks.
- The following code does this.

Decimal conversion - continued

LeadingZeros:

LDI XH, High(SRAM_START) ; Point X to the ASCII number, MSB

LDI XL, Low(SRAM_START) ; Dto., LSB

LeadingZerosChck:

LD R16, X ; Read character at position

CPI R16, '0' ; Is that a leading zero?

BRNE LeadingZerosFinished ; No, not a zero, finished

LDI R16, ' ' ; Load a blank

ST X+, R16 ; Overwrite the ASCII zero

CPI XL, SRAM_START+4 ; Is that the last digit?

BRNE LeadingZerosChck

LeadingZerosFinished:

- This is what comes out if a binary 9 is converted:

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	20	20	20	20	39	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	9
\$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
\$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	08

Double registers - specialties

- There are 3 double registers in AVR: X (R27:R26), Y (R29:R28) Z (R31:R30). With those, ST writes the content of a register to the SRAM location that X/Y/Z point to (pointer registers). LD loads the content in the SRAM to a register.
- X, Y and Z can auto-post-increment and auto-pre-decrement the pointer address.
- With ADIW XL/YL/ZL, constant a constant between 0 and 63 can be added word-wise, with SBIW XL/YL/ZL, constant subtracted.
- The register pair R25:R24 also knows ADIW and SBIW, but cannot point to locations in SRAM.
- The addresses below SRAM-START can be accessed with X/Y/Z too, there are the registers and the port registers located.

Double registers - specialties

- A special feature can be used with Y and Z: a displacement constant can be added temporarily, the instructions are STD and LDD.
- This feature is useful if you have a record-type structure in SRAM, that have identical inner structures. By changing the base address in Y/Z you can access the single bytes of the identical structure inside the record.
- Accesses to SRAM locations without pointers are STS address, register and LDS address, register. Those are two-word instructions that have the 16-bit address as second word.
- As in larger AVR's the port registers exceed 64 locations due to the many hardware, these extra port registers can be accessed by this method. So, if IN/OUT reports an error (port address above 0x3F), try if STS/LDS with the given address in the def.inc does not fail.

Conclusions

- **The AVR's offer many different methods to access flash memory, registers, port registers and SRAM.**
- **Choosing the appropriate addressing method can optimize program flow, increase the elegance of the source code and save execution time.**

Questions and tasks in Lecture 11

Task 11-1: Write a program that reads a null-terminated text from a byte table in the flash memory (inserted with `.DB "This is a text.", 0`) to the beginning of the SRAM.

Bonus Task 11-1: Write the text backwards to the SRAM. Try different methods and find out which is the most elegant.

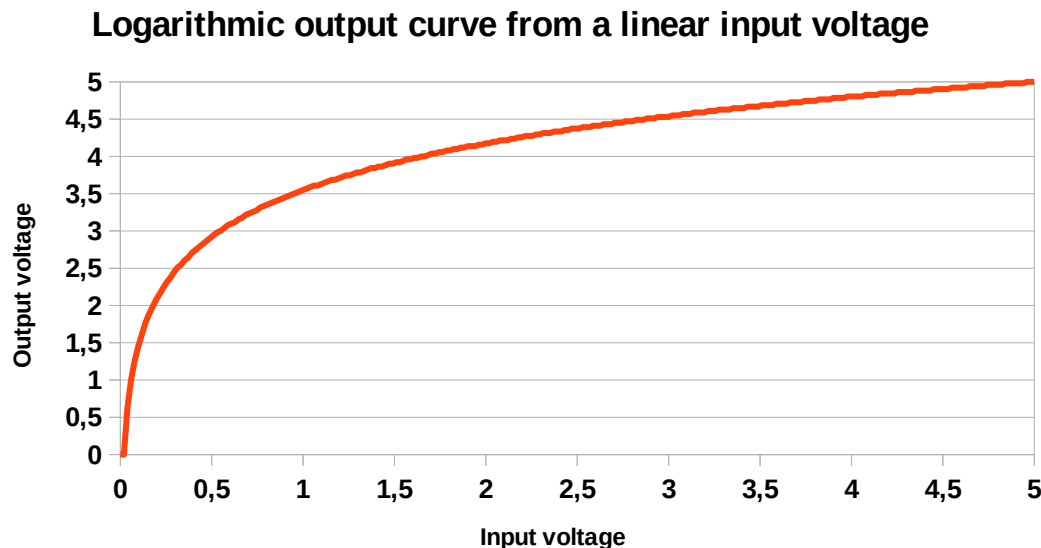
Bonus question: How long can the text be in an ATtiny24, in an ATtiny44 and in an ATtiny84 (how many characters)?



Questions and tasks in Lecture 11 - Continued

Task 11-2: Write a program that continuously

- reads an analog voltage byte-wise (with ADLAR!) every 10 ms (use DATE and a timer to start conversion, use the ADCC interrupt and the T flag),
- converts this to its logarithm with a byte-wise table, and



- writes that value to an 8-bit PWM at 100 Hz to convert it back into a voltage.

Questions and tasks in Lecture 11 - Continued

Task 11-3: Write a program that converts 32-bit binaries to decimal in the SRAM and blanks leading zeros.

