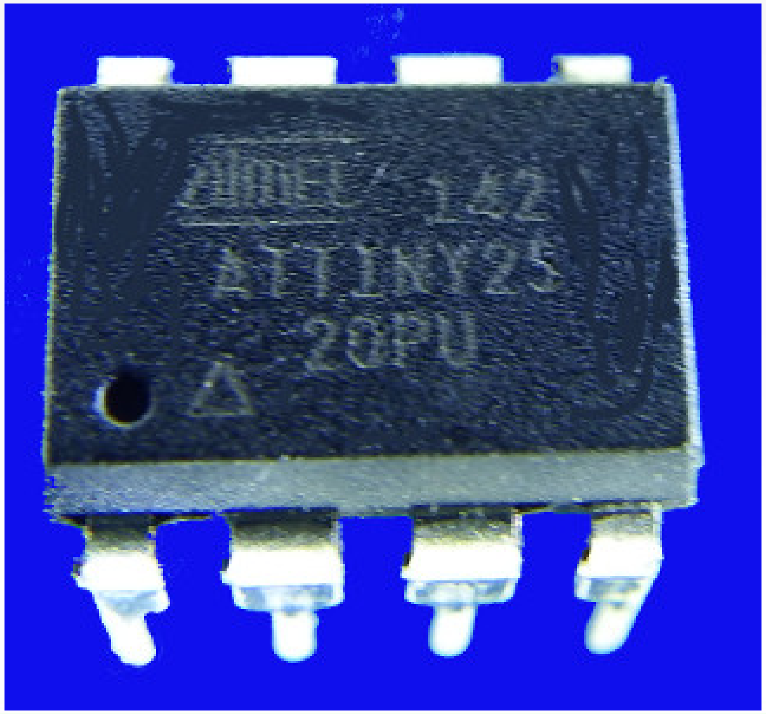


Lecture 8: ADC & EEPROM

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

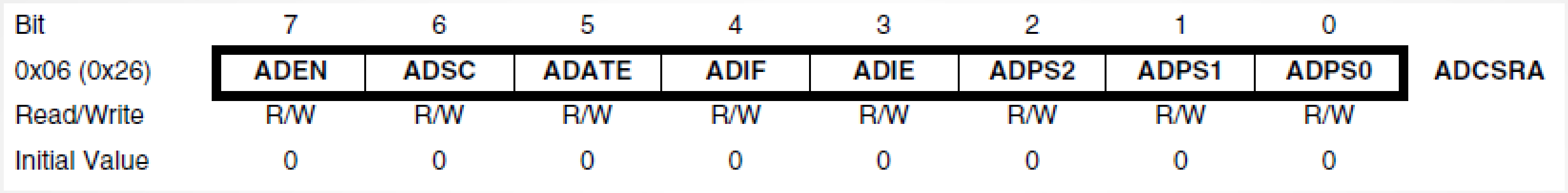
by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

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 ¾ of the reference voltage. If not, the highest bit is zero and the next comparison is with ¼ 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:**

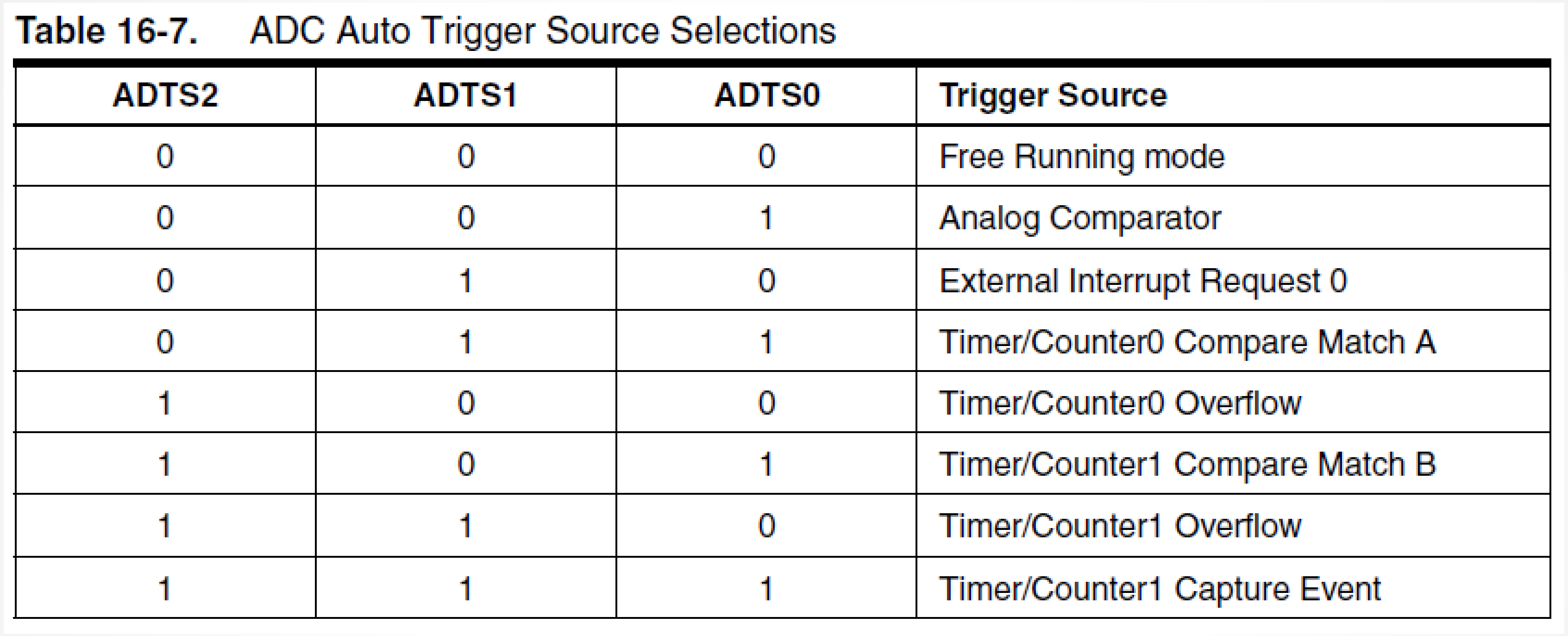


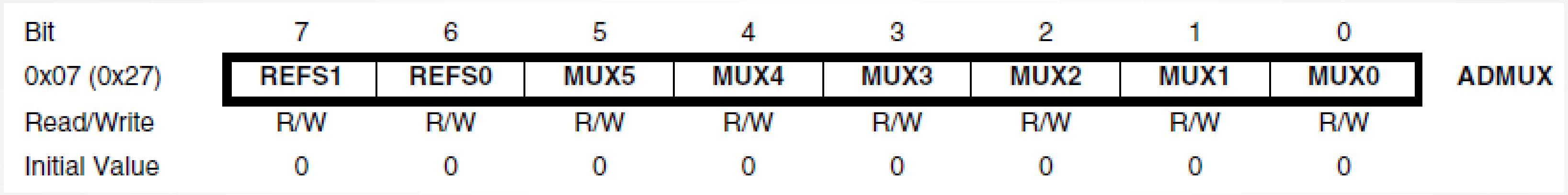
Table 16-7. ADC Auto Trigger Source Selections

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:

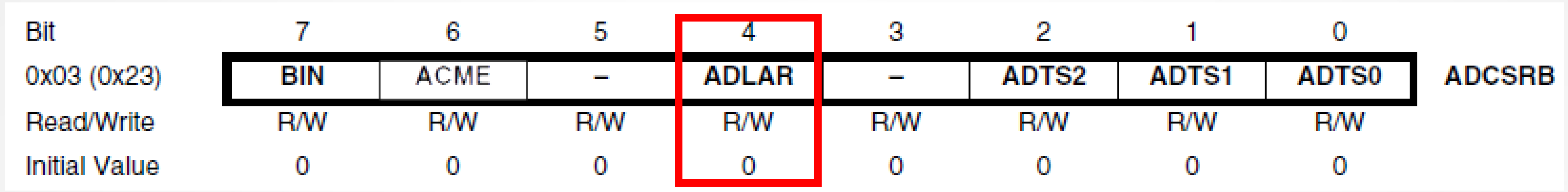


Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	REFS1	REFS0	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- 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:



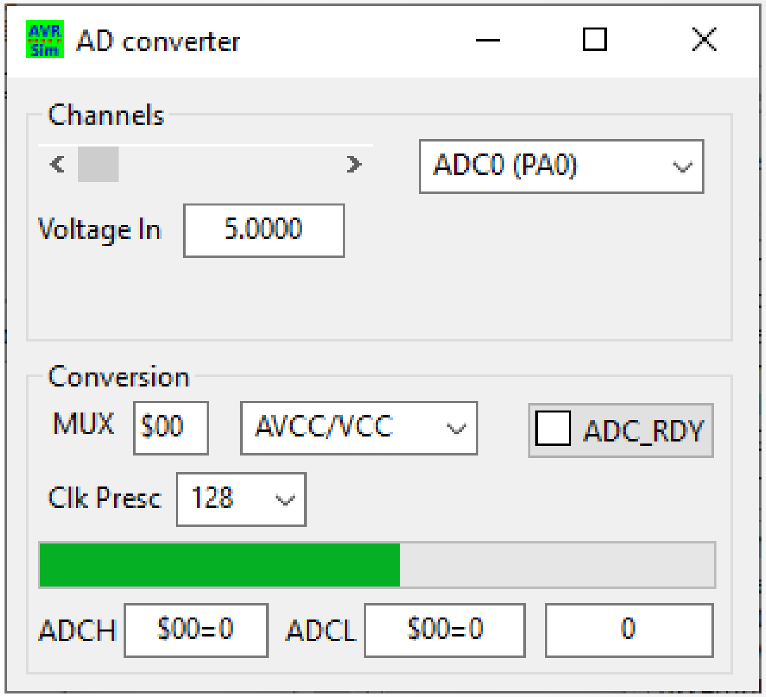
Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	BIN	ACME	–	ADLAR	–	ADTS2	ADTS1	ADTS0	ADCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- 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.**



AD converter

Channels: ADC0 (PA0)

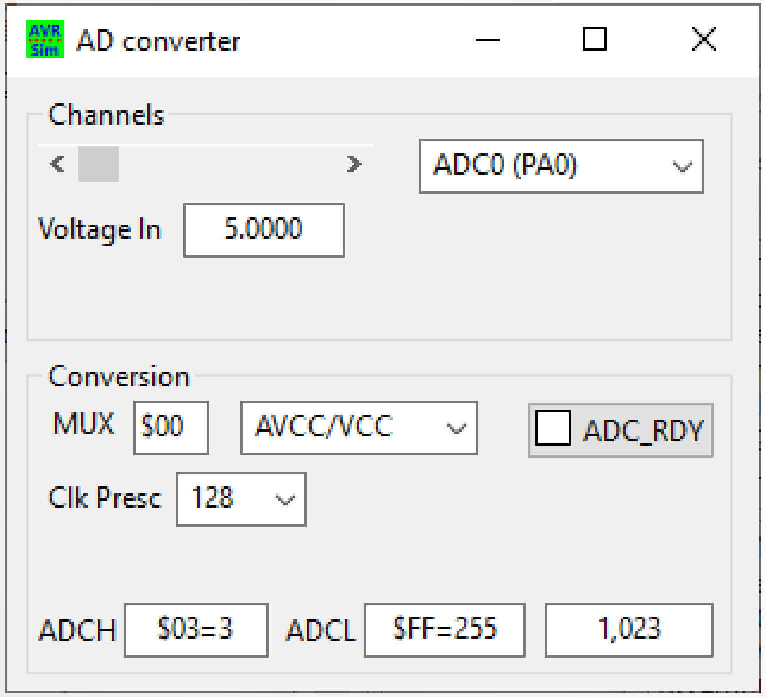
Voltage In: 5.0000

Conversion: MUX $00, AVCC/VCC, ADC_RDY

Clk Presc: 128

ADCH: $00=0 ADCL: $00=0 0

- **Conversion can be followed with the progress bar.**



AD converter

Channels: ADC0 (PA0)

Voltage In: 5.0000

Conversion: MUX $00, AVCC/VCC, ADC_RDY

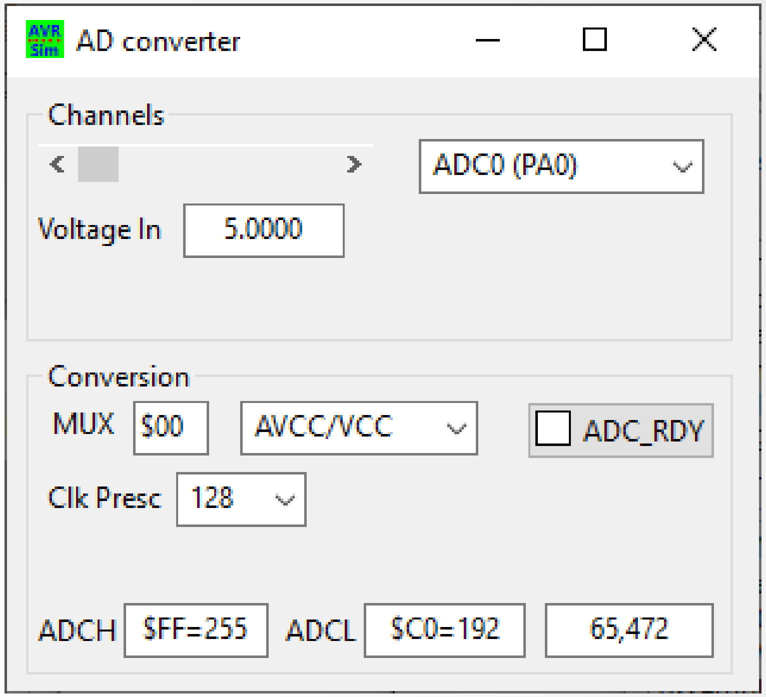
Clk Presc: 128

ADCH: $03=3 ADCL: $FF=255 1,023

- **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.



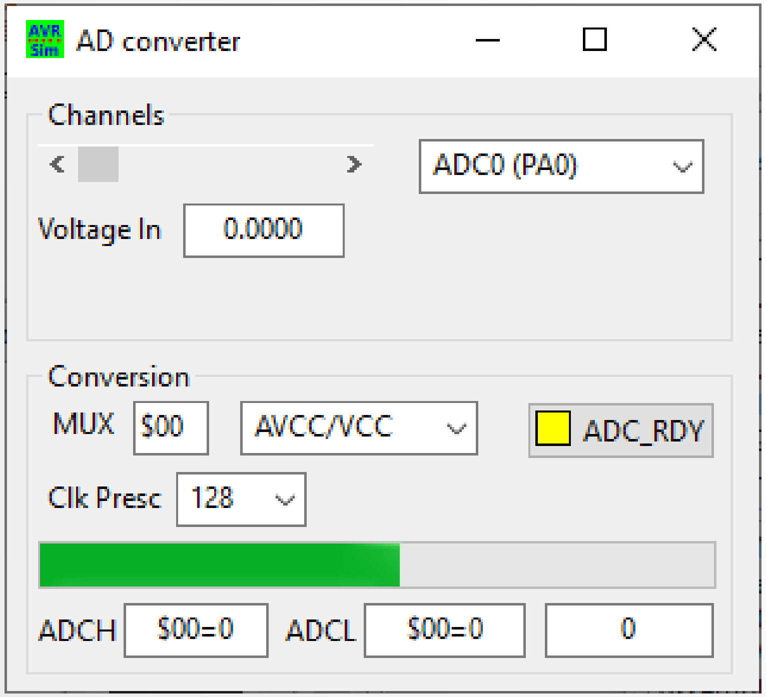
AD converter

Channels: ADC0 (PA0)
Voltage In: 5.0000

Conversion
MUX: $00 | AVCC/VCC | ADC_RDY
Clk Presc: 128

ADCH: $FF=255 | ADCL: $C0=192 | 65,472

- Enabling interrupts by setting the ADIE bit, by initiating the stack, by setting the I flag and having an ISR:



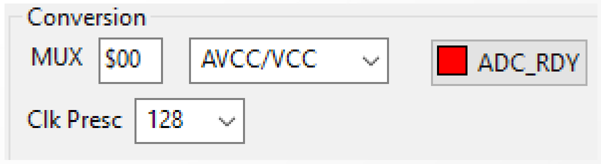
AD converter

Channels: ADC0 (PA0)
Voltage In: 0.0000

Conversion
MUX: $00 | AVCC/VCC | 🟨 ADC_RDY
Clk Presc: 128

ADCH: $00=0 | ADCL: $00=0 | 0

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



Conversion
MUX: $00 | AVCC/VCC | 🟥 ADC_RDY
Clk Presc: 128

EEPROM

- All AVRs have EEPROM memory on board.
- EEPROM means Electricly 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 prgram 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

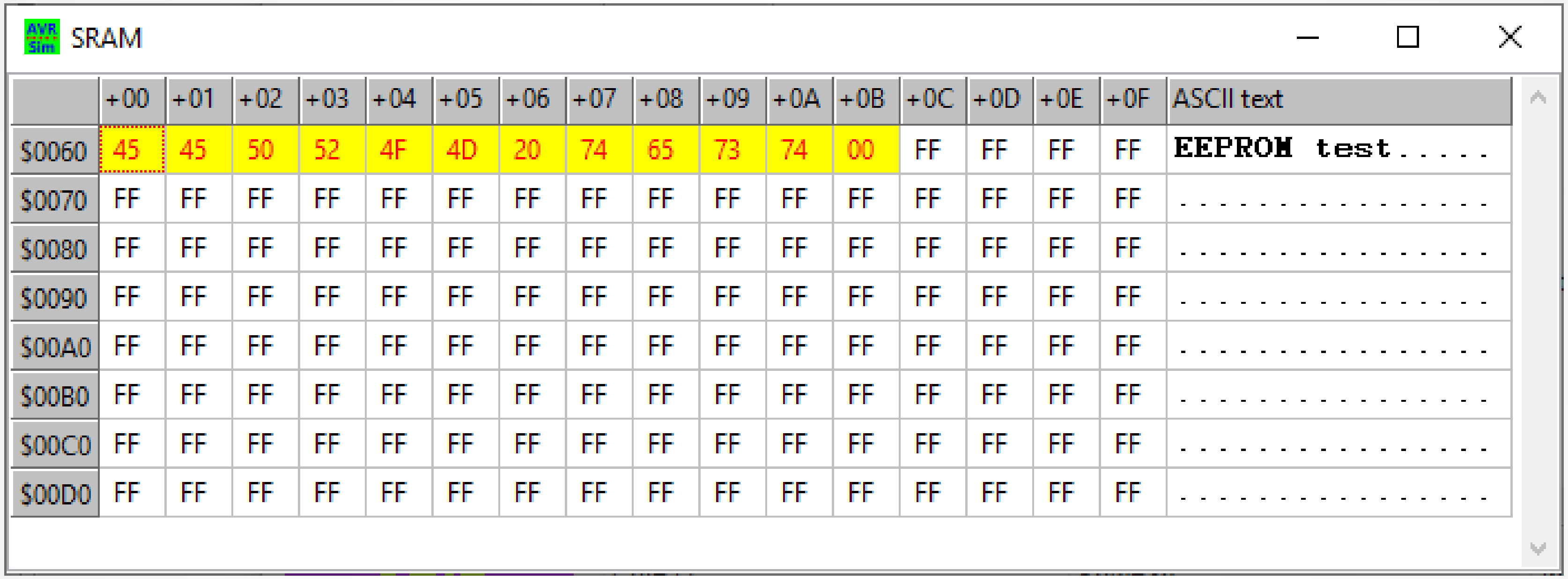
- **The ISR looks like this:**

```
; ISR for EEPROM ready interrupt
EepIsr:
  IN rSreg, SREG ; Save SREG
  LD R16,X+ ; Read byte at address in XH:XL and auto-increment
  TST R16 ; Check Null character
  BREQ EepIsrEnd ; 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 EepIsrRet ; Jump to return
EepIsrEnd:
  CBI EECR, EERIE ; Disable EEP interrupt
EepIsrRet:
  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:

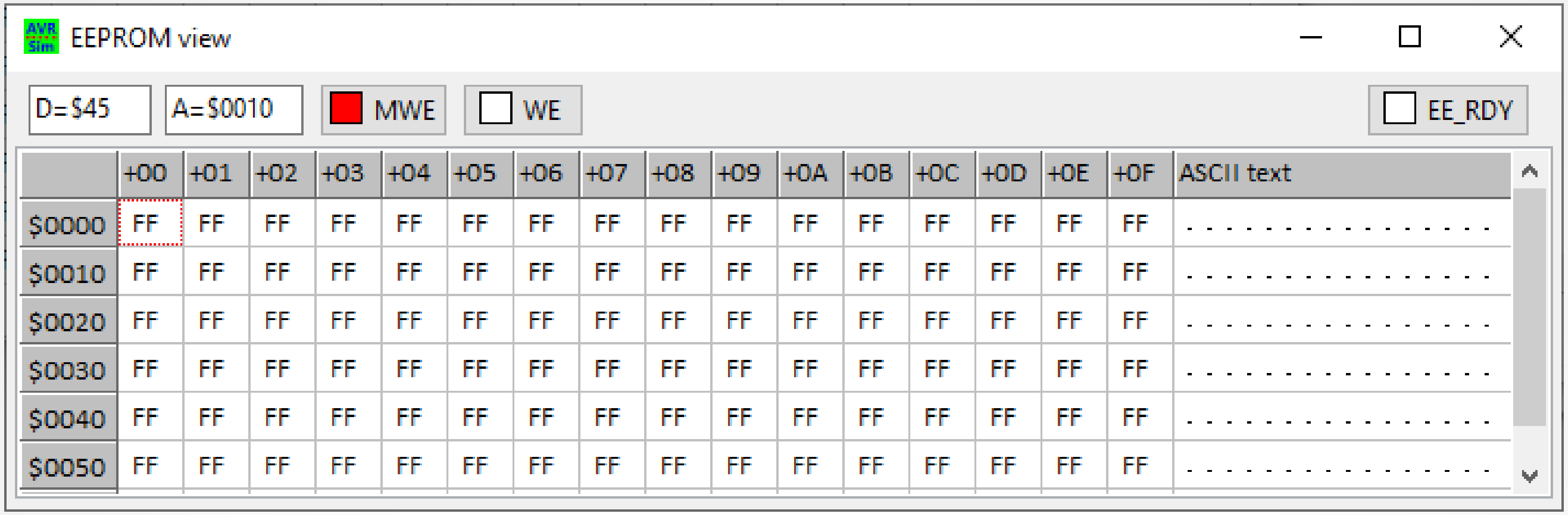


SRAM

	+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

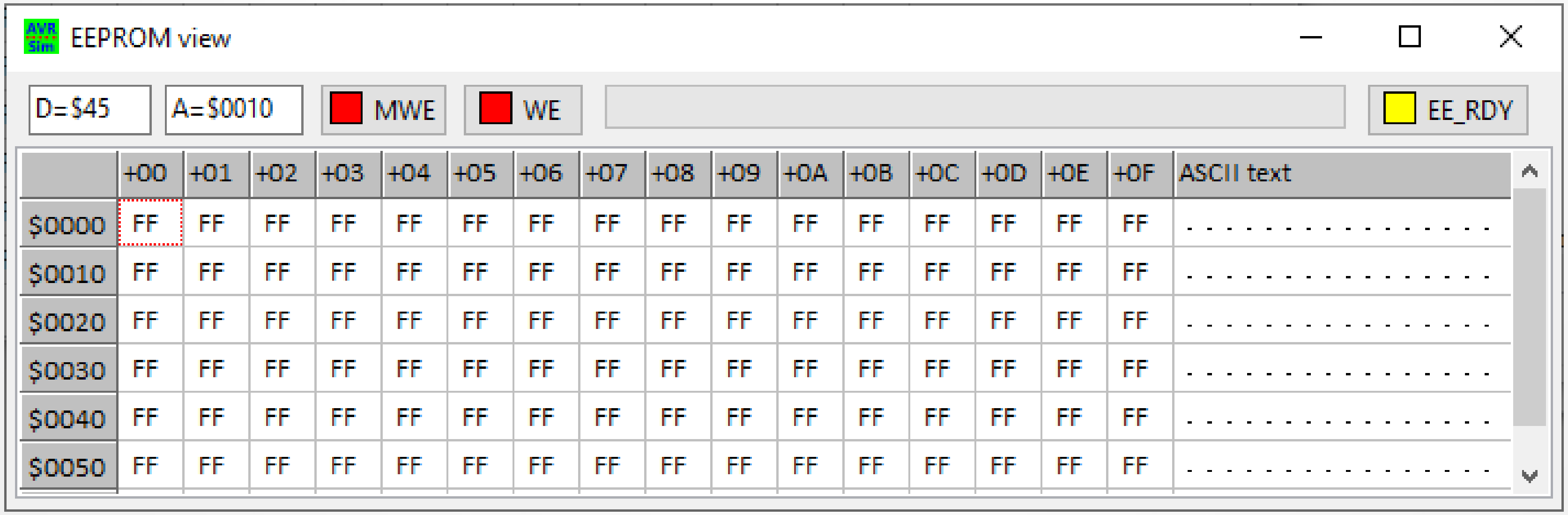


EEPROM view

D=$45 | A=$0010 | 🟥 MWE | ☐ WE | ☐ EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .

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



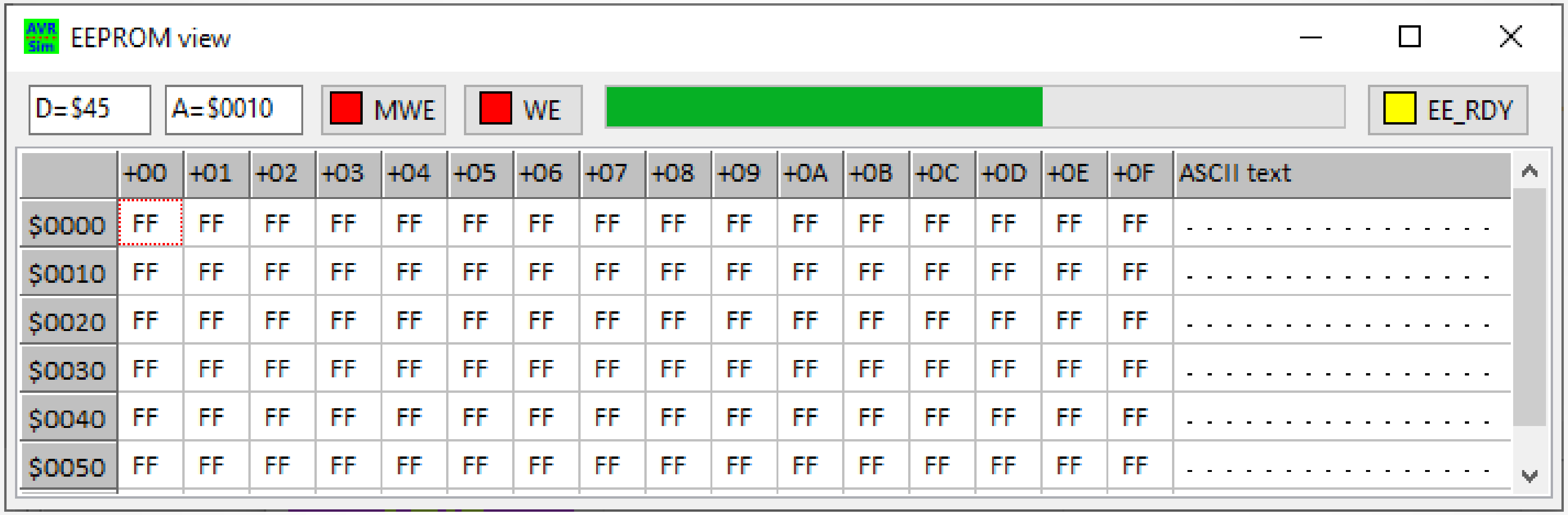
EEPROM view

D=$45 | A=$0010 | 🟥 MWE | 🟥 WE | 🟨 EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .

Example writing EEPROM

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

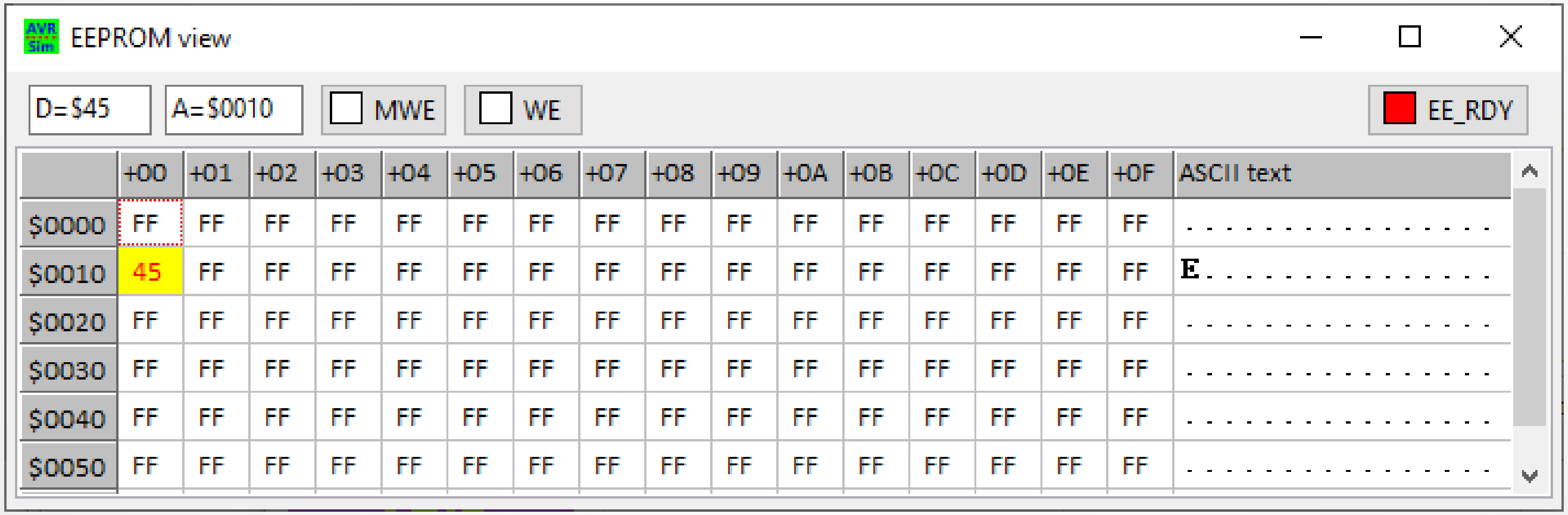


EEPROM view

D=$45 | A=$0010 | 🟥 MWE | 🟥 WE | 🟨 EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

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



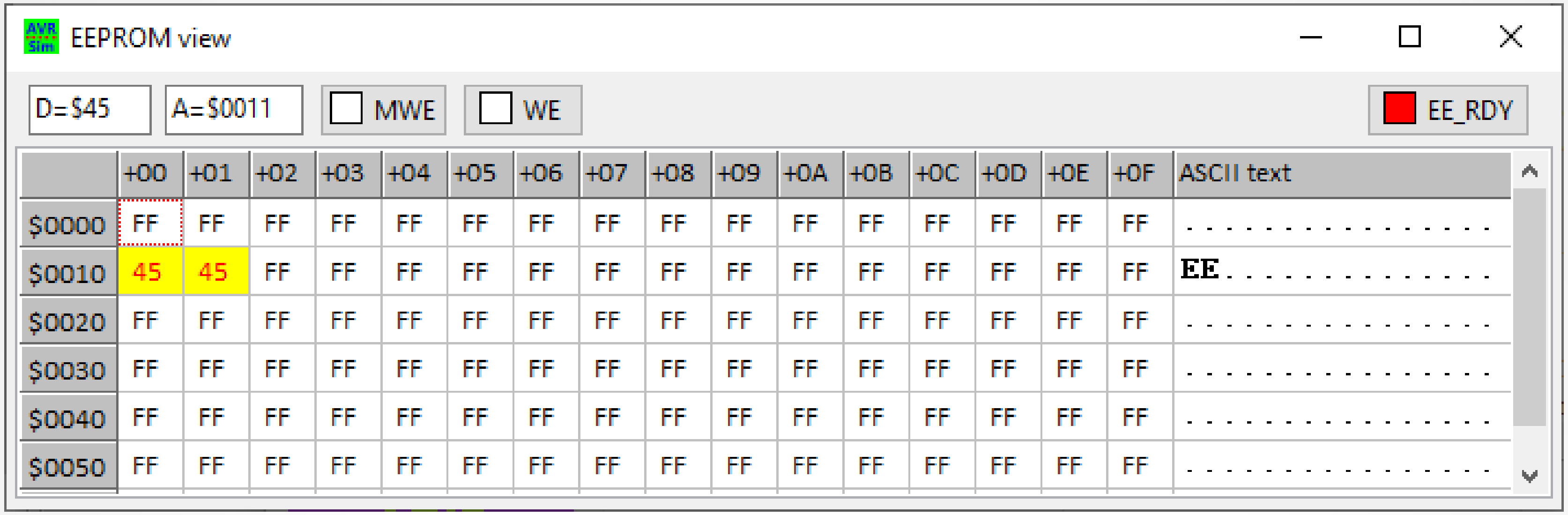
EEPROM view

D=$45 | A=$0010 | ☐ MWE | ☐ WE | 🟥 EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0010	45	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	E...............
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

Example writing EEPROM

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

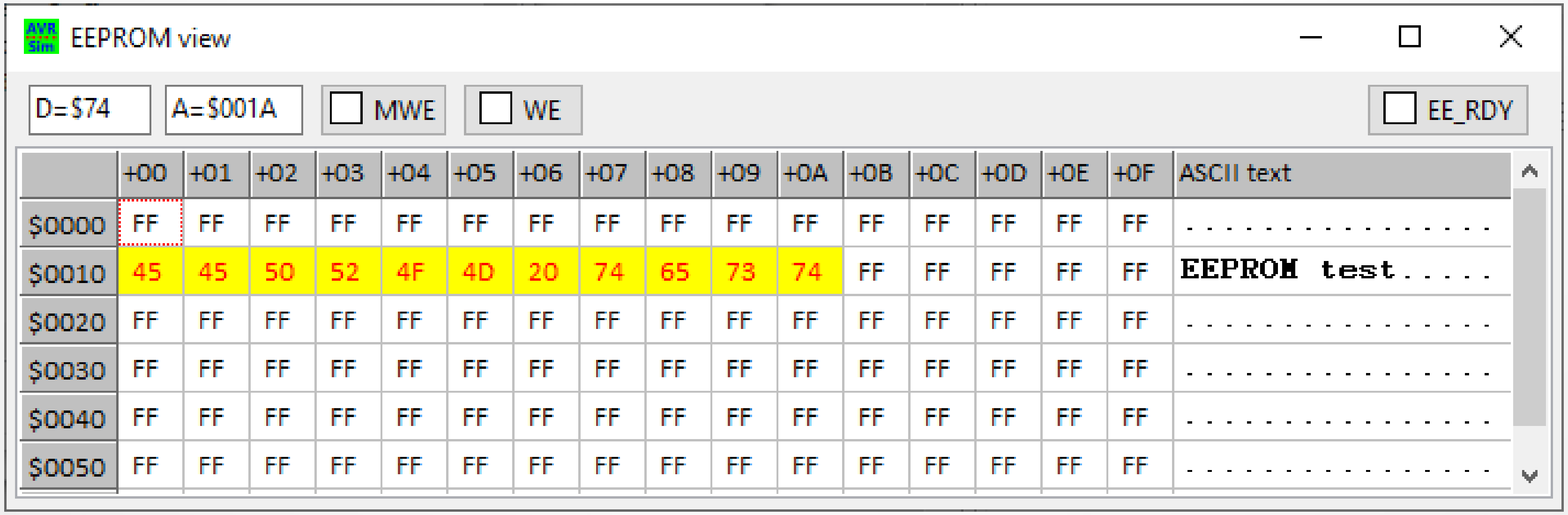


EEPROM view

D=$45 | A=$0011 | ☐ MWE | ☐ WE | 🟥 EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0010	45	45	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	EE..............
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

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



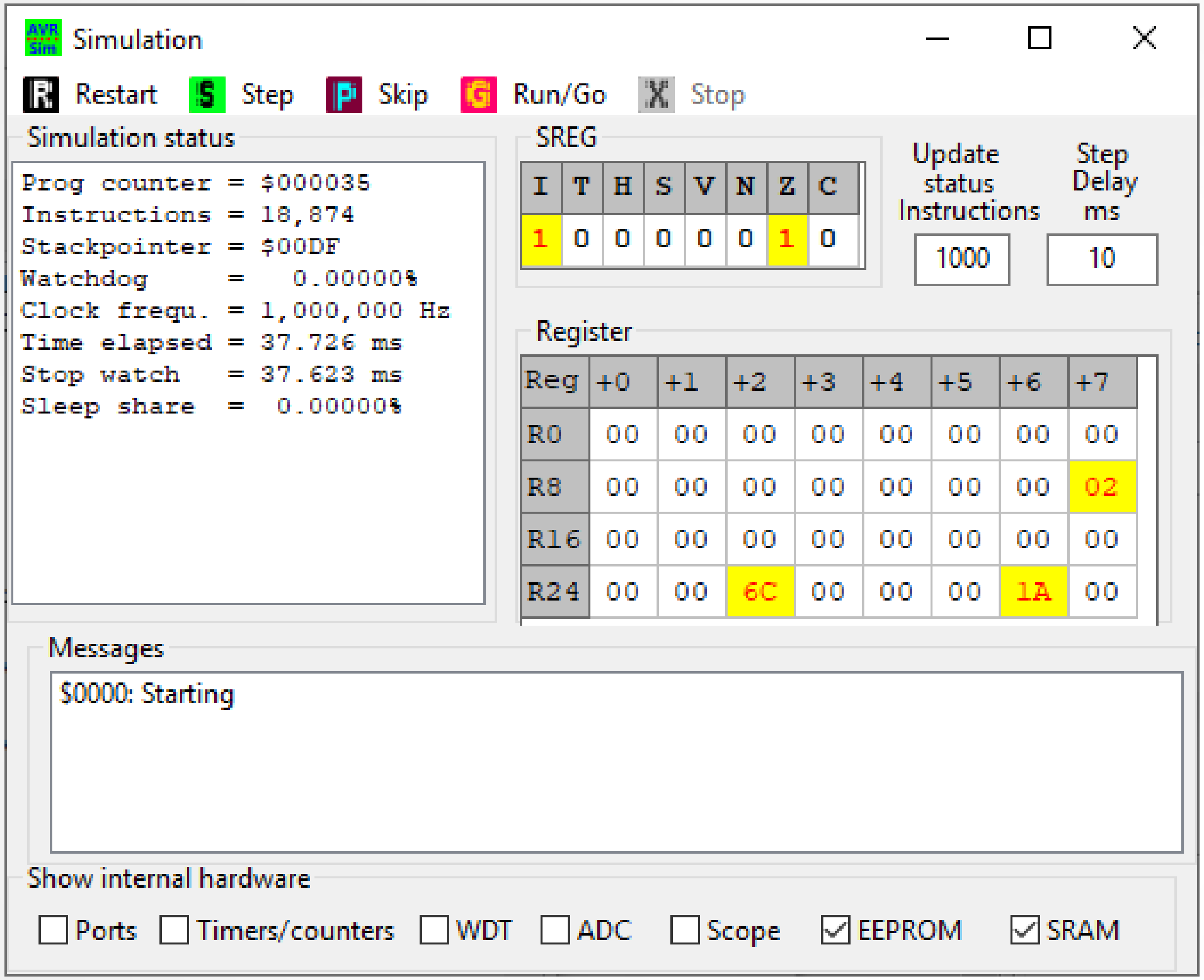
EEPROM view

D=$74 | A=$001A | ☐ MWE | ☐ WE | ☐ EE_RDY

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0010	45	45	50	52	4F	4D	20	74	65	73	74	FF	FF	FF	FF	FF	EEPROM test.....
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

Example writing EEPROM

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



Simulation status

```
Prog counter = $000035
Instructions = 18,874
Stackpointer = $00DF
Watchdog     =   0.00000%
Clock frequ. = 1,000,000 Hz
Time elapsed = 37.726 ms
Stop watch   = 37.623 ms
Sleep share  =  0.00000%
```

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 continously 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.