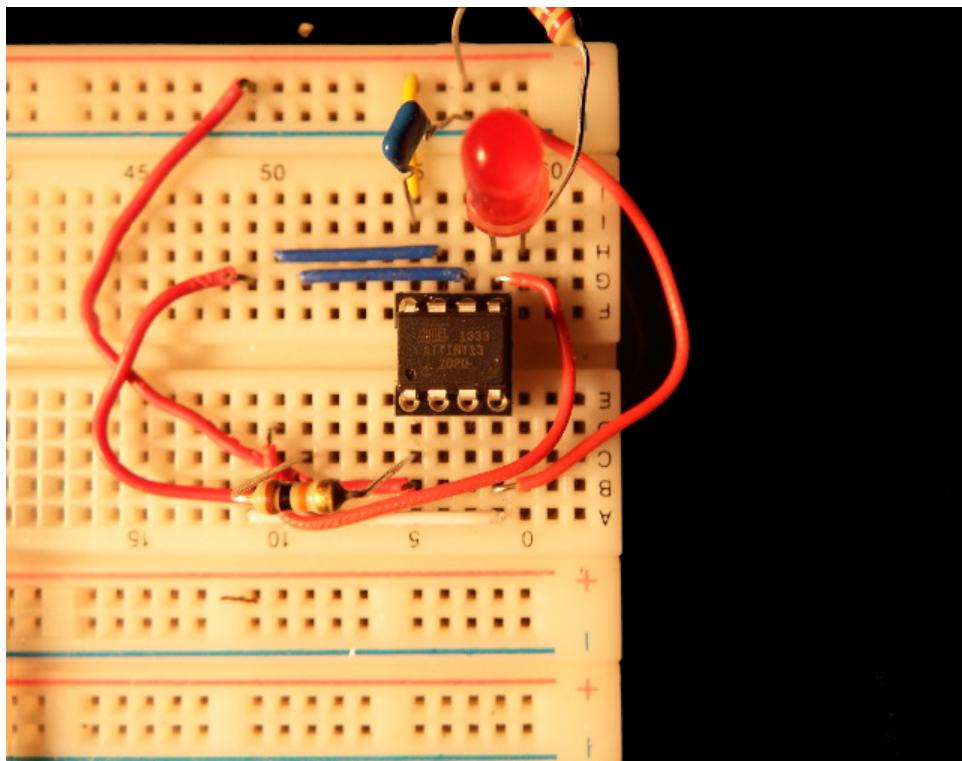


Microcontrollers for Beginners

on the basis of practical examples
with ATMEL AVR controllers in Assembler



by

Gerhard Schmidt, Kastanienallee 20, D-64289 Darmstadt
Version 2, January 04, 2018
Included in this version : Simulation with [avr_sim](#)

[**http://www.avr-asm-tutorial.net/avr_en/micro_beginner/**](http://www.avr-asm-tutorial.net/avr_en/micro_beginner/)

Preface

Overall task

This text provides 14 lectures to understand and master the hard- and software of AVR micro-controllers. By experiments on a breadboard relevant electronic, controlling and programming technologies are learned and practically tested. The necessary home-brewed software demonstrates how the hardware can be adjusted to own purposes and needs.

The conceptual approach

The following text was written with the experience in mind that learning new things is faster if a complete project task is to be solved and if this holistic task requires learning new knowledge. Task means a complete job with all necessary things to resolve the task and get the whole hard- and software working together. In this case holistic means electronics, the internals of micro-controllers and programming these internals using assembler language. Those three aspects are closely related and, as you will see, have to be fine-tuned to yield the well working result.

Programming is not learned systematically here from scratch to complete, perfect and in all detail. That is why you find the whole instruction set only as an attachment at the end, pointing to the text portion where this instruction was first used in practice and where the things that an instruction does with the controller make a practical sense. If you prefer learning „on stock“, systematically and instruction by instruction, you'll find enough tutorials in the net where this approach is taken (including my own one on [this webpage](#)).

This here takes a very different approach: to blink a LED in a slow rhythm involves knowledge on LEDs, knowledge on the internal hardware of the controller (e.g. its port drivers design) as well as understanding how and in which time the controller executes something. We have to understand counting loops, their time requirements and how to calculate all the necessary delays. And at last: how we can control all this by telling instructions to the controller that he understands and exactly follows, to do exactly what we planned for him to do. The blinking LED at the end of all this says: we understood all those things correct, translated the algorithm well into source lines and the controller does exactly follow our thoughts.

The plan behind those 14 lectures is that you understand (nearly) all the opportunities that the controller offers, and the selected tasks cover 95% of all this internals (I left serial communication, hardware multiplication and the watchdog out of the scope). If you like you can mount the components to the breadboard, assemble the offered source code (see [Attachment 5: Links to the source code files](#)), burn that into the controller chip and feel happy about the achieved result. At the end you have learned nothing about the electronics, the controller and its assembler language. You will not be able to design your own tasks, to write your own source code and to solve your own problem. So better try to understand the provided basics, too.

Another approach is to look at the most similar task in the lectures that fits nearest to your own task. And to complain by email that I did not define the task in exactly the same way that you need it („Why did you not use the ATmegaXX for that?“, „What do I have to do to get that working on an ATTiny841?“, „I need to turn two LEDs on and off softly, why didn't you use that example?“). You can chose that kind of cherry picking, but do not expect that I will solve your own problem. Rather go through the whole text and select all things for more extensive reading that might contribute to your basic understanding of things that you need to solve your task, e.g. the extensive text on interrupt vectors and interrupt processing or about the PWM mode of timers.

Those who prefer systematic learning, because this is the preferred learning method in schools and universities, can go to this [address](#) and continue there. Those who search for a specific type of information and want to use this document as reference book can go directly to Attach-

ment 6: Themes index and search there for an appropriate link.

Assembler for learning

The concept to learn electronics as well as the hard- and software of controllers does only work consequently in assembler language. Other languages put too much in between you and the hardware and how that works in detail, and integrate hundreds of single instructions into just one word (that only hopefully fits nearly to your needs). You really will not be able to understand controllers if you use Basic language, this hides anything from you that the controller really does and can do. And you will run into a lot of unnecessary problems if you use C, just because learning another language is seen as something extra and complicated if you already understand another language construction. Behind that other language construction stands another world, which in case of C is not the controller's world, but an abstract virtual construction. As an example: to design exact switching times for two or three related signals, rather simple in assembler, is a real horror in C.

Not assembler is complicated but the internal hardware of a controller is rather complicated and complex. Complexity reduction therefore can only be achieved for the price of not really understanding how that all finally works together. If you have to select between 10 different modes of a 16 bit timer so that it fits best to your external hardware you should at least understand these modes, what they do and what they are good for. In this version simulation of the internal hardware has been added: if you do not understand lengthy text descriptions, the pictures on simulated internal hardware might ease your understanding. No high level language will ease that process of understanding the timer. The two or three instructions that are necessary to bring the timer into that mode are not really complicated in assembler. And the ten or so lines for the interrupt service routine are not so, either. Do not try that in C, it will drive you crazy because your compiler follows its own undocumented rules on what to do next.

Yes, doing some multiplication and division of 40 bit binaries is, at first, rather complicated. If once understood (in later lectures), it becomes rather simple and you will not need any code libraries, that do not exactly fit to your needs and provide overdone extras that you'll never need.

Note

This text uses copies from the device data-books in several cases. The Copyright for those is with [ATMEL](#). All other graphics and pictures are (C)2016/2017 by me, if you use those it is nice and good practice to mention the source.

Necessary hardware components

You will need several components if you want to test the provided software on your own. Those are either available in local electronics shops or via internet trade. The needed components are listed lecture by lecture in [Attachment 4: Component list by lectures](#).

Feedback

I am happy about any positive, negative or neutral feedback. Please send that to info at the website avr-asm-tutorial.net.

I am not a native English speaker and translated all this from the original German version. So it might include many language errors or forgotten German original text. Please excuse that.

Changes to text

Date	Chapter	Page(s)	Changes
03.03.20	8	84	Hint on reading ADCL/ADCH of the AD converter
	10	156	Hint on available LCD include file
27.02.19	Instruction list	302 to 305	Parameter limitations added for MOVW, IN, OUT, CBI, SBI, BRBS, BLD, BST
16.07.18	Att. 3	306	Added .IFNDEF directive
24.05.18	Att. 2	301	Added limitation to RCALL instruction
	10	147	Added the LCD-RW pin to be compatible with the separate LCD module pack

Overview of the lectures

Preface.....	I
Lecture 1: An ISP programming interface.....	1
Lecture 2: Switch a LED on.....	12
Lecture 3: A LED is blinking.....	26
Lecture 4: Blinking a LED with a timer.....	35
Lecture 5: To control the intensity of a LED via PWM.....	48
Lecture 6: A LED blinks with the timer interrupt.....	55
Lecture 7: A LED blinks with a key interrupt.....	71
Lecture 8: Regulation of the intensity of a LED.....	82
Lecture 9: An audio generator with ADC, tone table, multiplication.....	101
Lecture 10: A Lcd display on an ATtiny24.....	133
Lecture 11: EEPROM with LCD on an ATtiny24.....	156
Lecture 12: IR receiver and transmitter.....	176
Lecture 13: Frequency counter and inductance meter.....	244
Lecture 14: Voltage, current and temperature meter.....	275
Conclusions from those lectures.....	295
Attachment 1: Preferred register uses.....	297
Attachment 2: Instructions in AVR-Assembler.....	298
Attachment 3: Directives in AVR assembler.....	302
Attachment 4: Component list by lectures.....	303
Attachment 5: Links to the source code files.....	305
Attachment 6: Themes index.....	306

Content

Preface.....	I
Lecture 1: An ISP programming interface.....	1
1.0 Overview.....	1
1.1 Introduction.....	1
1.2 Hardware.....	2
1.2.1 The programming device.....	2
1.2.2 The programming hardware.....	2
1.3 Components and mounting.....	4
1.3.1 Components.....	4
1.3.2 Mounting.....	5
1.4 Operation.....	7
Lecture 2: Switch a LED on.....	12
2.0 Overview.....	12
2.1 Introduction.....	12
2.1.1 LEDs.....	12
2.1.2 Controller pins as in- and output.....	13
2.1.3 Controller pins as output.....	13
2.2 Hardware.....	14
2.3 Components and mounting.....	15
2.3.1 Components.....	15
2.3.2 The hardware.....	15
2.4 Programming.....	15
2.4.1 Program storage.....	15
2.4.2 Source code.....	16
2.4.3 To assemble.....	16
2.4.4 Writing source code.....	17
2.4.5 To assemble.....	18
2.4.6 To write the hex code to the program flash storage.....	21
2.5 Simulating program execution.....	21
2.5.1 Simulating with ATMEL's Studio.....	22
2.5.2 Simulating with avr_sim.....	24
Lecture 3: A LED is blinking.....	26
3.0 Overview.....	26
3.1 Introduction.....	26
3.1.1 Execution of instructions by the controller.....	26
3.1.2 Execution times of instructions.....	27
3.2 Hardware, components and mounting.....	27
3.3 Fast blinking.....	27
3.2.1 The simple fast blinker.....	27
3.2.2 Delayed fast blinking, 8-Bit.....	30
3.2.3 Delayed fast blinking, 16 Bit.....	30
3.4 Exact second blinking.....	31
Lecture 4: Blinking a LED with a timer.....	35
4.0 Overview.....	35
4.1 Introduction to timer hardware.....	35
4.1.1 Timer.....	35
4.1.2 Timer pulse sources.....	35
4.1.3 Timer and compare match.....	36

4.1.4 Timer in CTC mode.....	<u>37</u>
4.1.5 Timer manipulation of pin outputs.....	<u>37</u>
4.2 Hardware, components and mounting.....	<u>37</u>
4.3 Timer with standard operation.....	<u>38</u>
4.3.1 Functioning.....	<u>38</u>
4.3.2 Program.....	<u>38</u>
4.3.3 Simulating timer operation in this mode.....	<u>39</u>
4.3.4 Disadvantage of that solution.....	<u>40</u>
4.4 Clock prescaler.....	<u>40</u>
4.4.1 Functioning.....	<u>40</u>
4.4.2 Program.....	<u>43</u>
4.4.3 Simulating timer operation in CTC mode.....	<u>44</u>
4.4.4 Advantages and drawbacks.....	<u>44</u>
4.5 Timer in 128kcs/s mode.....	<u>45</u>
4.5.1 Functioning.....	<u>45</u>
4.5.2 Program.....	<u>45</u>
4.5.3 Simulation of the timer operation at 128 kHz.....	<u>46</u>
4.5.4 Advantages and drawbacks.....	<u>47</u>
Lecture 5: To control the intensity of a LED via PWM.....	<u>48</u>
5.0 Overview.....	<u>48</u>
5.1 Introduction to the PWM mode of the timer.....	<u>48</u>
5.1.1 8 bit PWM.....	<u>48</u>
5.1.2 Phase correct 8 bit PWM.....	<u>50</u>
5.1.3 PWM with different resolution.....	<u>50</u>
5.2 Hardware, components and mounting.....	<u>51</u>
5.3 Fast PWM mode.....	<u>51</u>
5.3.1 Simulation of the Fast PWM mode.....	<u>52</u>
5.4 Timer in phase correct PWM mode.....	<u>54</u>
Lecture 6: A LED blinks with the timer interrupt.....	<u>55</u>
6.0 Overview.....	<u>55</u>
6.1 Introduction to interrupt programming.....	<u>55</u>
6.1.1 Interrupts.....	<u>55</u>
6.1.2 Stack storage.....	<u>56</u>
6.1.3 Interrupt vectors.....	<u>57</u>
6.1.4 The timer overflow interrupt.....	<u>58</u>
6.1.5 The compare match interrupt.....	<u>58</u>
6.1.6 Interrupts and sleep modes.....	<u>58</u>
6.2 Hardware, components, mounting.....	<u>59</u>
6.3 Timer with overflow interrupt.....	<u>59</u>
6.3.1 Task description.....	<u>59</u>
6.3.2 Steps towards solution.....	<u>59</u>
6.3.3 Program.....	<u>59</u>
6.3.4 The result.....	<u>62</u>
6.3.5 Program structuring.....	<u>62</u>
6.3.6 Simulation of the processes.....	<u>62</u>
6.3.7 Advantages and disadvantages.....	<u>66</u>
6.4 Timer with CTC interrupt.....	<u>66</u>
6.4.1 CTC selection.....	<u>66</u>
6.4.2 Program.....	<u>67</u>
6.4.3 Simulation.....	<u>69</u>

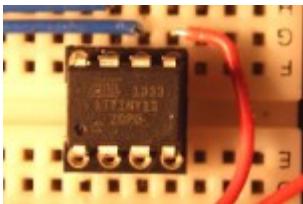
Lecture 7: A LED blinks with a key interrupt.....	<u>71</u>
7.0 Overview.....	<u>71</u>
7.1 Introduction to key operation and INT0 programming.....	<u>71</u>
7.1.1 Keys on input ports.....	<u>71</u>
7.1.2 Keys and switches are bouncing.....	<u>72</u>
7.1.3 The INT0 interrupt.....	<u>72</u>
7.2 The task to be programmed.....	<u>73</u>
7.3 Hardware, components and mounting.....	<u>74</u>
7.3.1 Hardware.....	<u>74</u>
7.3.2 Components.....	<u>74</u>
7.4 Program.....	<u>74</u>
7.4.1 Processes and flows.....	<u>74</u>
7.4.2 Flow diagrams.....	<u>75</u>
7.4.3 The Program.....	<u>76</u>
7.4.4 Simulating the processes.....	<u>78</u>
Lecture 8: Regulation of the intensity of a LED.....	<u>82</u>
8.0 Overview.....	<u>82</u>
8.1 Introduction to AD conversion.....	<u>82</u>
8.2 Introduction to the PCINT programming.....	<u>84</u>
8.3 Hardware, components and mounting.....	<u>84</u>
8.3.1 Hardware scheme.....	<u>84</u>
8.3.2 Components.....	<u>84</u>
8.3.3 Mounting.....	<u>85</u>
8.4 Intensity regulation.....	<u>85</u>
8.4.1 Task 1.....	<u>85</u>
8.4.2 Solution 1.....	<u>85</u>
8.4.3 Program 1.....	<u>85</u>
8.4.4 Simulating program execution.....	<u>88</u>
8.5 Intensity regulation with color change.....	<u>90</u>
8.5.1 Task 2.....	<u>90</u>
8.5.2 Debouncing.....	<u>91</u>
8.5.3 Program 2.....	<u>92</u>
8.6 Intensity regulation dynamically.....	<u>95</u>
8.6.1 Task 3.....	<u>95</u>
8.6.2 Solution.....	<u>95</u>
8.6.3 Program 3.....	<u>95</u>
8.7 Fast red/green change.....	<u>98</u>
8.7.1 Task 3.....	<u>98</u>
8.7.2 Solution.....	<u>99</u>
8.7.3 Program.....	<u>99</u>
Lecture 9: An audio generator with ADC, tone table, multiplication.....	<u>101</u>
9.0 Overview.....	<u>101</u>
9.1 Introduction to audio generation.....	<u>101</u>
9.2 Hardware, components and mounting.....	<u>101</u>
9.2.1 The hardware scheme.....	<u>101</u>
9.2.2 The components.....	<u>102</u>
9.2.3 Mounting.....	<u>102</u>
9.3 Regulating the tone frequency.....	<u>102</u>
9.3.1 Simple task 1.....	<u>102</u>
9.3.2 Solution.....	<u>103</u>

9.3.3 Program.....	<u>103</u>
9.3.4 Simulation of the program.....	<u>105</u>
9.4 Task 2: To play the tones of the gamut.....	<u>108</u>
9.4.1 Task.....	<u>108</u>
9.4.2 Introduction to tables.....	<u>108</u>
9.4.3 Introduction to multiplication.....	<u>112</u>
9.4.4 Programming the gamut.....	<u>114</u>
9.4.5 Debugging with avr_sim.....	<u>117</u>
9.4.6 Debugging with the Studio.....	<u>122</u>
9.5 Playing music.....	<u>124</u>
9.5.1 Task.....	<u>124</u>
9.5.2 The melody to be played.....	<u>124</u>
9.5.3 Tone duration.....	<u>124</u>
9.5.4 Tone pauses.....	<u>125</u>
9.5.5 Duration to play different notes.....	<u>126</u>
9.5.6 Processing structure.....	<u>126</u>
9.5.7 Program.....	<u>127</u>
9.5.8 Simulating execution.....	<u>130</u>
Lecture 10: A Lcd display on an ATtiny24.....	<u>133</u>
10.0 Overview.....	<u>133</u>
10.1 Introduction to LCD displays.....	<u>133</u>
10.1.1 General on LCD displays.....	<u>133</u>
10.1.2 Interfaces of LCD displays.....	<u>134</u>
10.1.3 Controlling LCD devices.....	<u>135</u>
10.1.4 Backlight of LCDs.....	<u>136</u>
10.2 Introduction to the ATtiny24.....	<u>137</u>
10.3 Hardware, components and mounting.....	<u>138</u>
10.3.1 Scheme.....	<u>138</u>
10.3.2 Components.....	<u>138</u>
10.3.3 Mounting.....	<u>139</u>
10.3.4 Alternative Mounting: The ATtiny24 LCD experimental board.....	<u>139</u>
10.4 Controlling a LCD with delay loops.....	<u>139</u>
10.4.1 Timing requirements and construction of delay loops.....	<u>139</u>
10.4.2 Task.....	<u>140</u>
10.4.3 Program.....	<u>141</u>
10.4.4 Simulating the wait routines.....	<u>144</u>
10.5 Control of the LCD in busy mode.....	<u>145</u>
10.5.1 Reading the busy flag of the LCD.....	<u>145</u>
10.5.2 Task.....	<u>145</u>
10.5.3 Program.....	<u>146</u>
10.6 To design new characters.....	<u>149</u>
10.6.1 The character generator in LCDs.....	<u>149</u>
10.6.2 Software for character design.....	<u>150</u>
10.6.3 Task.....	<u>150</u>
10.6.4 The program.....	<u>150</u>
Lecture 11: EEPROM with LCD on an ATtiny24.....	<u>156</u>
11.0 Overview.....	<u>156</u>
11.1 Introduction to EEPROM.....	<u>156</u>
11.1.1 An EEPROM as a permanent memory.....	<u>156</u>
11.1.2 To write to the EEPROM.....	<u>156</u>

11.1.3 To read EEPROM content in assembler.....	<u>157</u>
11.1.4 To write data to the EEPROM in assembler.....	<u>158</u>
11.2 Introduction to decimal conversion of binaries.....	<u>158</u>
11.2.1 The most primitive (and most lengthy) version.....	<u>158</u>
11.2.2 The improved version.....	<u>158</u>
11.2.3 The 16 bit version.....	<u>159</u>
11.3 Task, hardware, components and mounting.....	<u>160</u>
11.3.1 Task.....	<u>160</u>
11.3.2 Scheme.....	<u>160</u>
11.4 Byte counter.....	<u>160</u>
11.4.1 The LCD routines as include file.....	<u>160</u>
11.4.2 The program.....	<u>165</u>
11.4.3 Simulation.....	<u>167</u>
11.5 Word counter.....	<u>170</u>
11.5.1 Task.....	<u>170</u>
11.5.2 The program for that task.....	<u>170</u>
11.5.3 Simulation.....	<u>173</u>
Lecture 12: IR receiver and transmitter.....	<u>176</u>
12.0 Overview.....	<u>176</u>
12.1 Introduction to infrared signals.....	<u>176</u>
12.2 Introduction to conditioned assembly programming.....	<u>177</u>
12.2.1 To define conditions.....	<u>177</u>
12.2.2 If-then-else alternatives.....	<u>178</u>
12.2.3 Other helpful directives.....	<u>178</u>
12.3 Hardware, components and mounting.....	<u>179</u>
12.3.1 Schemes.....	<u>179</u>
12.3.2 Components: the IR receiver modules.....	<u>179</u>
12.3.3 Mounting.....	<u>180</u>
12.4 Measuring IR signals.....	<u>180</u>
12.4.1 Task.....	<u>180</u>
12.4.2 Start signals.....	<u>180</u>
12.4.3 End signals.....	<u>185</u>
12.4.4 Number of signals.....	<u>186</u>
12.4.5 Signal durations of data bits.....	<u>190</u>
12.4.6 Encoding of IR commands.....	<u>195</u>
12.5 An IR transmitter.....	<u>201</u>
12.5.1 The hardware for IR transmitting.....	<u>202</u>
12.5.2 Transmitter algorithms.....	<u>203</u>
12.5.3 Program for transmitting.....	<u>204</u>
12.5.4 Simulating the transmission.....	<u>208</u>
12.5.5 Analysis of the transmitted signals.....	<u>211</u>
12.6 An IR data transfer system.....	<u>211</u>
12.6.1 The IR analog data transmitter.....	<u>212</u>
12.6.2 Software.....	<u>212</u>
12.6.3 Simulation of the software with the Studio.....	<u>217</u>
12.6.3 The IR analog data receiver.....	<u>218</u>
12.7 A self-learning IR receiver with three channel relay switches.....	<u>227</u>
12.7.1 Task.....	<u>227</u>
12.7.2 Hardware, components and mounting.....	<u>227</u>
12.7.3 Program.....	<u>229</u>

12.7.4 Measured IR remote control codes.....	<u>230</u>
12.7.5 Diagnosis with the ATtiny24 version with LCD.....	<u>231</u>
12.7.6 Program.....	<u>231</u>
Lecture 13: Frequency counter and inductance meter.....	<u>244</u>
13.0 Overview.....	<u>244</u>
13.1 Introduction to frequency measuring.....	<u>244</u>
13.1.1 Digital signals and counting limits.....	<u>244</u>
13.1.2 Detecting analog signals.....	<u>245</u>
13.1.3 Measuring inductivities.....	<u>245</u>
13.2 Introduction to decimal conversion (24 and 32 bit).....	<u>249</u>
13.3 Measuring digital signals with the PCINT.....	<u>250</u>
13.3.1 Task.....	<u>250</u>
13.3.2 Hardware, mounting.....	<u>250</u>
13.3.3 Program.....	<u>250</u>
13.3.4 Examples.....	<u>255</u>
13.4 Frequency measurement with the analog comparator.....	<u>256</u>
13.4.1 Task.....	<u>256</u>
13.4.2 Hardware and components.....	<u>256</u>
13.4.3 Program.....	<u>257</u>
13.4.4 Experiences and example.....	<u>261</u>
13.5 Measuring inductance with PCINT.....	<u>262</u>
13.5.1 Task.....	<u>262</u>
13.5.2 Hardware and components.....	<u>262</u>
13.5.3 Program.....	<u>263</u>
13.5.4 Simulating program execution.....	<u>271</u>
13.5.4 Examples.....	<u>273</u>
Lecture 14: Voltage, current and temperature meter.....	<u>275</u>
14.0 Overview.....	<u>275</u>
14.1 Measuring, calculate and display of voltages.....	<u>275</u>
14.1.1 Hardware.....	<u>275</u>
14.1.2 Measuring range, measure/calculation/display issues.....	<u>276</u>
14.1.3 Program.....	<u>277</u>
14.1.4 Example.....	<u>280</u>
14.2 Measuring, calculation and display of currents.....	<u>280</u>
14.2.1 Hardware.....	<u>281</u>
14.2.2 Measuring range, measure/calculate/display issues.....	<u>281</u>
14.2.3 Program.....	<u>283</u>
14.2.4 Example.....	<u>287</u>
14.3 Measuring, calculation and display of temperatures.....	<u>287</u>
14.3.1 Hardware.....	<u>287</u>
14.3.2 Measuring range, measuring, calculation and display.....	<u>287</u>
14.3.3 Program.....	<u>288</u>
14.3.4 Example.....	<u>294</u>
Conclusions from those lectures.....	<u>295</u>
Controller internal hardware.....	<u>295</u>
External hardware.....	<u>295</u>
Program design.....	<u>296</u>
Attachment 1: Preferred register uses.....	<u>297</u>
Attachment 2: Instructions in AVR-Assembler.....	<u>298</u>
Attachment 3: Directives in AVR assembler.....	<u>302</u>

Attachment 4: Component list by lectures.....	<u>303</u>
Attachment 5: Links to the source code files.....	<u>305</u>
Attachment 6: Themes index.....	<u>306</u>



Lecture 1: An ISP programming interface

1.0 Overview

1. [Introduction](#)
2. [Hardware](#)
3. [Components and mounting](#)
4. [Operation](#)

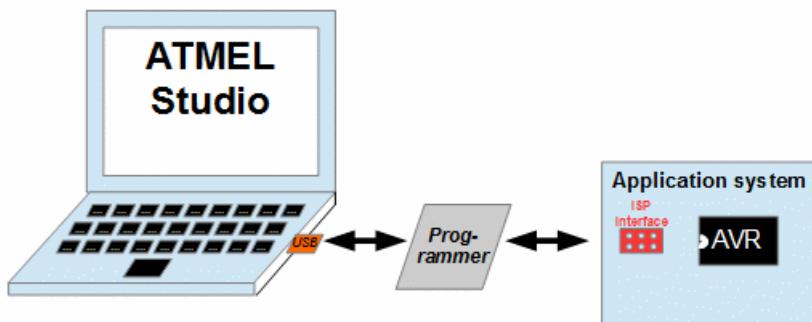
1.1 Introduction

To understand microprocessors one needs to access the processor's internal components via a programming interface. With that, one can

- transfer program code to the processor, or
- delete this program code,
- select the clock source for clocking the device,
- transfer data to the EEPROM inside,
- read internal program code and EEPROM content, and
- protect the chip against reading of its content.

Without this programming interface nothing goes, the processor sleeps and does absolutely nothing.

In these lectures programming is performed within the readily mounted electronic systems. This is called "In-System Programming" or ISP. This shows what is needed:



This has immense advantages: the final electronic system works immediately after programming, no further and unnecessary interfaces or systems are between you and the microprocessor and limit or modify your connection with the processor. Bye bye complicated cable orgies, such as with an STK500 or an Arduino. The current supply of the electronic system is not affected at all.

The ISP interface of the AVR works in all device types in the same manner, no matter if the device has six or 96 pins. If your application system does not allow to access the three port-pins used for the ISP interface just switch to the next larger device, change a few lines of your software to adapt to the new type and program this into the new device in your electronic system.

If you built in the six pin ISP plugin into your system, you can easily re-program the device. No need to unmount the processor or to solder new connections.

In this chapter we are learning to access the processor within the system and to read its device identifier. The other possibilities to change the chip's properties are demonstrated in brief.

1.2 Hardware

1.2.1 The programming device

Many different programming devices can be used. I use an ancient AVR-ISPMkII of ATMEL. This is a handy device, provides the desired six pin ISP connector and works perfectly with ATMEL's Studio. The following examples were all prepared with that. It is not available in ATMEL's store any more, so you'll have to switch to one of the many clones available.

The cheapest choice is to use a serial programmer that emulates a STK500 programming board. Communication between the STK500 and the Studio was over a RS232 serial interface, using a serial port between COM1 and COM4. The communication protocol was published by ATMEL, so can be emulated by an external board. As computers nowadays have no RS232 serial communication interfaces any more, you can use an USB-to-Serial interface cable. This installs on Windows and emulates a serial COM port over USB. But make sure it installs on a low COM port number. Please note that on whatever USB port you plug the cable to, Windows follows its own rules on which COM port it will install the serial port to. It should be anything from COM1 to COM4, so the Studio finds a STK500 there.

An emulated STK500 can program many devices but not any. The devices used here are clearly part of the STK500 choices, so can be programmed with any STK500 clone. Because my AVR ISP MK II has, in the meantime, given up I use a cheap Diamex USB-to-6pin programmer, which works fine and reliably.

1.2.2 The programming hardware

The supply of the system is performed by an AA four-pack rechargeable battery, with 1.2 V each. As AVRs are optimized to require very low current, a line power supply would be overdone. If

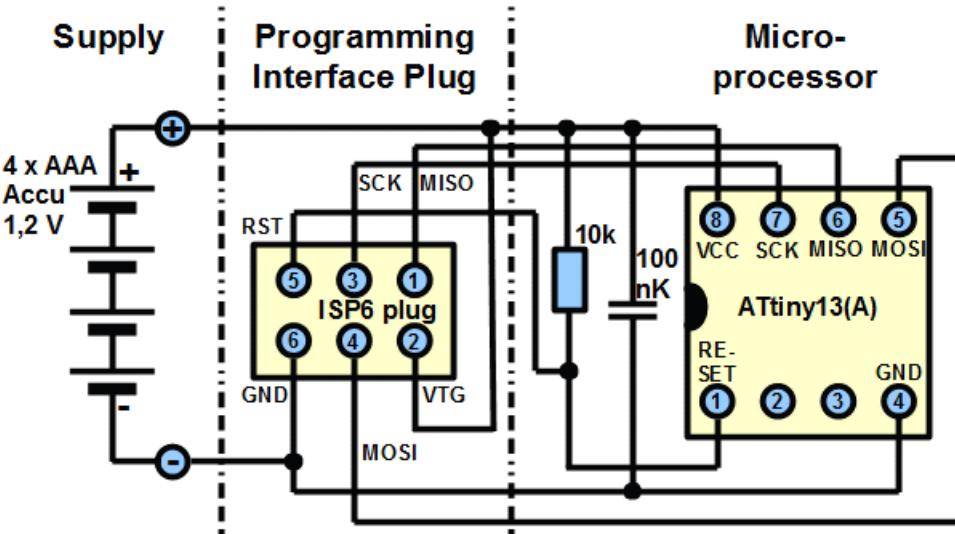
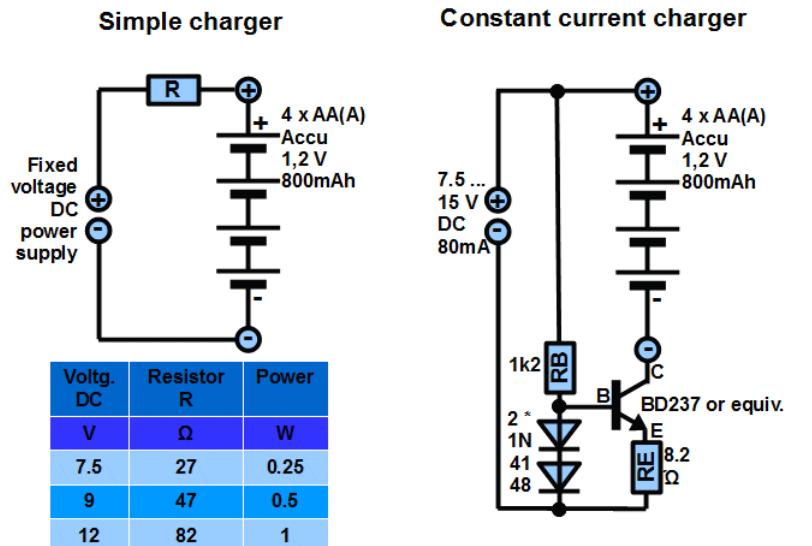
- you do not have a four-pack rechargeable battery (and you don't like the idea of buying one),
- you do not like this type of packs in general (e.g. because having four of such does not give you control over each of it),
- you are not able to charge such a package, because your charger charges 1.2V devices only, you do not have an adjustable power supply or a power transistor, two silicon diodes and two resistors to construct your own homemade charger,
- you have a power supply providing 5V and you prefer using this,

there are many practical alternatives for that as long as your supply provides between 4.5 and 5.5 V (lower voltages are not practical for the later lectures that use a 5V LCD) at a current of at least 20 mA.

Charging can of the four pack be done like shown here.

To the left, the simplest charger is shown. If the battery is completely empty, leave the resistor and the battery for ten hours on the DC source. Not much longer, though.

To the right, a more exact constant load charger is shown. It provides roughly 80 mA constant current and can be operated on any DC source between 7.5 and 15 V. Please load for not more than 10 hours.



The necessary hardware to access the AVR is shown here.

The supply of the system is performed by an AA four-pack rechargeable battery, with 1.2 V each. As AVRs are optimized to require very low current, a power supply would be overdone.

Via the three pins MOSI, MISO and SCK the communication between the programming device and the AVR is done. The AVR enters programming mode if the RESET pin (pin 1 of the ATtiny13) goes to low (which the programming device initiates via the RST line of the 6-pin ISP). Without active programming this pin is tied to the positive operating voltage with a resistor of 10 k, allowing the processor to do his work.

Between the positive and negative supply voltage, near to the processor's supply pins, a ceramic capacitor of 100 nF blocks high-frequency peaks that result from the processor's rectangle switching internals.

The pins of the processor are named with the active function that is used here. Each pin has additional names, depending from its use and function. Pins can change their function only if programmed to do so.

With this simple hardware we can communicate with the AVR and can access its internals.

1.3 Components and mounting

1.3.1 Components

1.3.1.1 The rechargeable battery pack



This is the rechargeable battery pack used. If you use batteries that are not rechargeable, use only three of those. You can use anything that provides between four and five Volts.

1.3.1.2 The ISP6 plug



The ISP plug is a two-row by three pins each male connector. For the use on a breadboard an adapter is necessary.

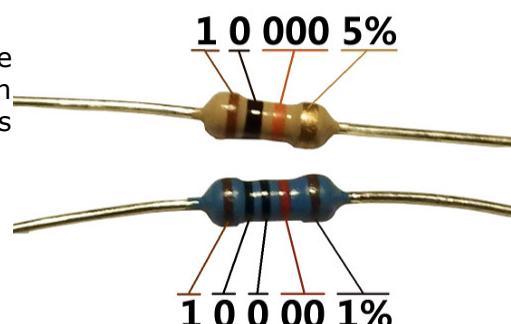
This adapter has to be self-mounted. For this a 3-by-4 wide piece of a printed circuit board with isolated eyes is cutted. On the soldering side a 2-by-3 plug connector is attached and soldered (with plugs only as deep in



the whole as necessary so that soldering is still enabled). On the other side of the board two 3-pin plug connectors are soldered in and the neighboring eyes are soldered together as shown.

1.3.1.3 The 10k resistor

Those here are two different types of resistors. In the upper part a carbon film, in the lower part a metal film resistor with 10 k is shown. The color coding of the rings is demonstrated.



1.3.1.4 The 100 nF ceramic capacitor

This here is such a capacitor.



1.3.1.5 The 8 pin socket

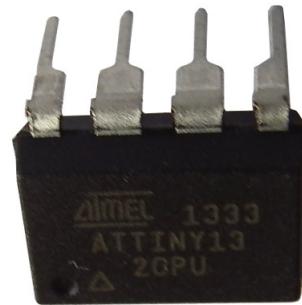
Into this socket the microprocessor can be plugged in.



1.3.1.6 The ATtiny13



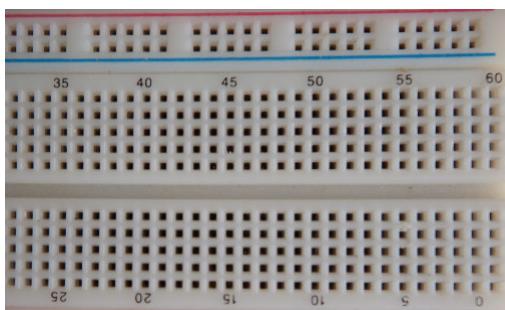
This is the microprocessor, even though it does not look alike. In the package many hardware is sleeping and can be attached to its eight legs, which are numbered in an unconventional way (left picture). At least pin 1 can be found easily, because it has several markings. First of all a small triangle is printed near this leg. Second, a small pit can be seen there. And third a small notch is on the side of the chip that holds leg #1. With those characteristics it is nearly impossible to plug in the chip the wrong way.



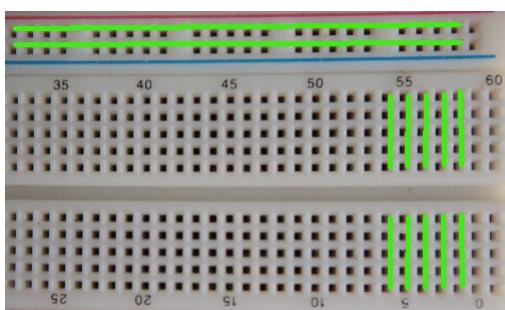
Of course, the processor does not fit into the socket above. His legs do not exactly point downwards but a little bit to the side. To correct this one can correct the four legs by pressing the chip cautiously on a hard surface and to move the four pins of each side slightly inwards. If you do not do that the risk is high that the legs are bent during pressing the chip into the socket. That means danger to the life of Mr. Professor and it is very funny to straighten those legs without loosing parts or all of them and then to move the whole chip to the recycling bin.

1.3.2 Mounting

1.3.2.1 The breadboard



This is a part of the breadboard. A breadboard is an experimental or developer system. Pins of components can directly be plugged into the holes and are contacted. The board allows to test electronic systems and the software before the whole goes into mass production (which is never the case with the systems we build in the upcoming lectures).



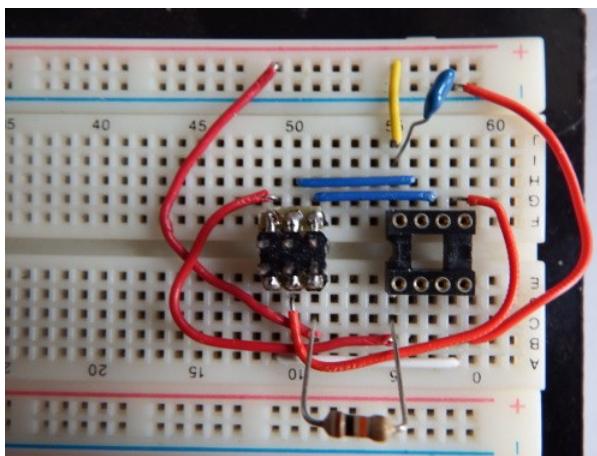
The green lines show which holes are interconnected (in the lower part of the picture only five connections are shown. The upper two rows are for the supply voltage, the lower columns for components. From that it is clear why the ISP6 plugin had to be altered: the separation line in the middle of the board lines is two rows from each side and in the component area such a two-row plug would be short-circuited.

1.3.2.2 Mounting

Mounting goes as follows:

1. Place the ISP6 plugin and the IC socket on the board.
2. Wire supply and direct connections.
3. Place and wire all other components (resistor, capacitor).
4. Check all connections between the ISP plugin and the IC socket using a multimeter in resistor measurement mode.
5. Connect the supply (rechargeable battery pack) with the supply lines of the board.
6. With the multi-meter in DC measurement mode check that pin 4 of the IC socket is connected to GND or Minus of the battery pack und pin 8 with VCC or plus. Similar check that on the ISP6 plugin on pins 6 (GND, Minus) and 2 (VTG, Plus). Check that voltage and polarity is correct.
7. Then push the ATtiny13 into the socket. Now plug the the programmer into the USB (check that the JUNGO driver is loaded correct) and connect with ISP6. The control lamp of the programmer should turn to green if the voltage is correct.

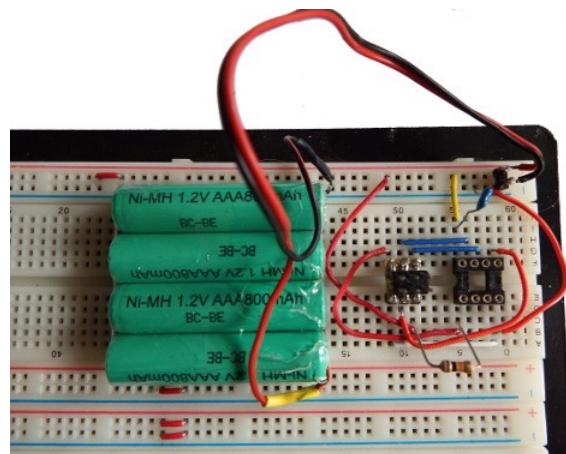
Leaving out single steps in that row (especially steps 4 and 6) ends in error messages. To debug those errors you will have to redo those steps, so why not doing those from the early beginning?



The rechargeable battery pack adds voltage, but still without the Tiny and the programmer. Now check polarities and voltages to not kill the Tiny with over-voltage or reverse polarity.

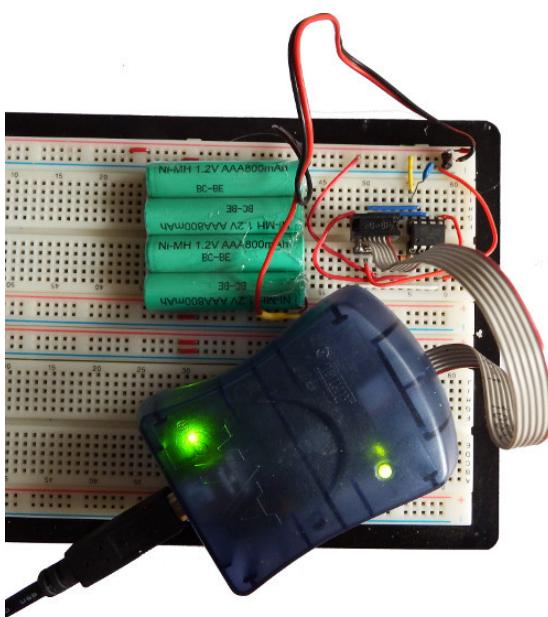
The following pictures show the stages of mounting.

So the mounting of all components looks like. The supply and the ATtiny13 are not yet in place to do the testing.



Now anything is complete and the AVR-ISPMkII is plugged in. The green LED signals that he is correctly connected to the USB plug and that the voltage on the ISP6 plugin is correct. A red LED would signal that something is wrong.

Now, what to do if you don't have and you don't get an AVR-ISPMkII? The answer is simple: take anything else that you get a programmer



- with a 6-pin standard programming interface on it to connect that with your breadboard (nearly all programmers have that, if yours has a 10-pin KANDA interface only you have to home-brew a 10- to 6-pin adapter), and
- that understands
 - a usual ATMEL USB-Interface for programming with the Studio such as an AVR dragon, or

- a serial RS232 interface that either is an STK500 or just acts like one and connects either to the Studio (either via RS232 or via an USB/RS232 interface, make sure that the interface installs on COM ports between 1 and 4) or supports any other programming software such as AVRdude, or
- an antique parallel interface programmer (make sure that your computer has such a parallel interface), together with AVRdude, or
- ...
- that supports ATTiny13(A) and ATTiny24(A), devices which are not the latest controllers and are supported by most programmers.

If you have the latest of the latest operating systems which does neither supports available programmer software nor access to a serial RS232 nor to your USB interface: consider stepping back in time and rather use something useful.

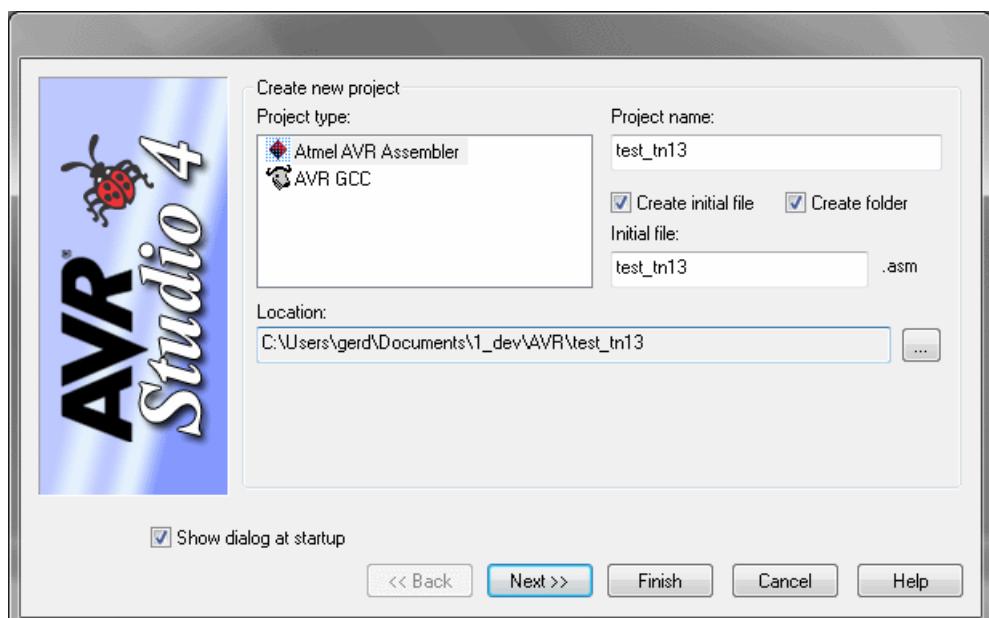
Now one can start the experiments and the access to the interior of Mister Professor.



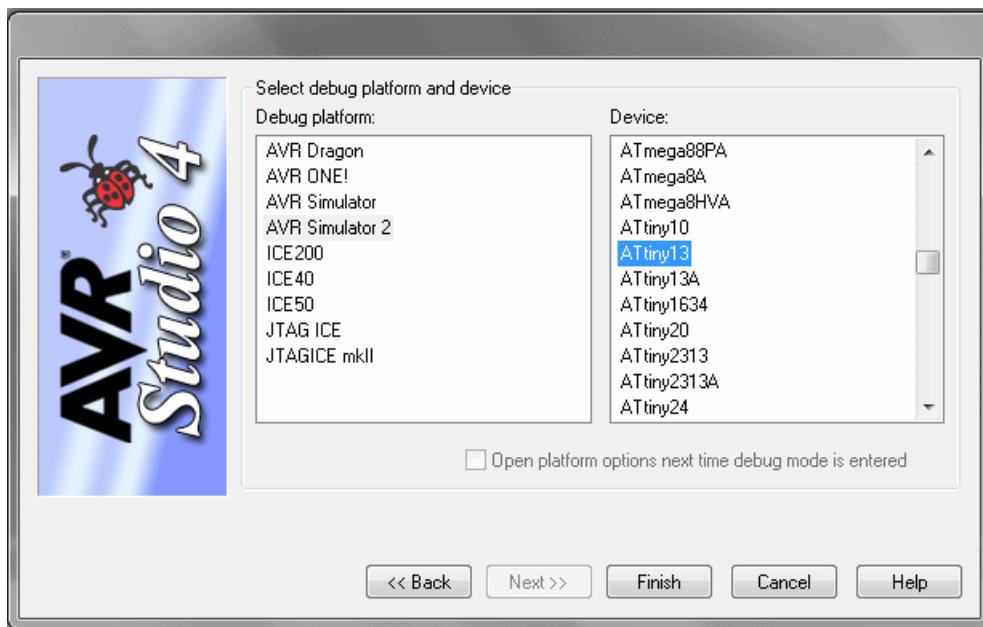
1.4 Operation

To access the interior of the professor we need some software. One gets this for free from [ATMEL](#) if we search there in the 8-bit section for "Studio", but only under Windows. Check that your hard-disk has lots of free space, because the software elephant requires well over one Gigabyte.

The following examples were produced using version 4 of the studio, an older version of that. It works similar to versions 5, 6 or 7 (at least for our purposes). Right at the beginning of a project select "Assembler".

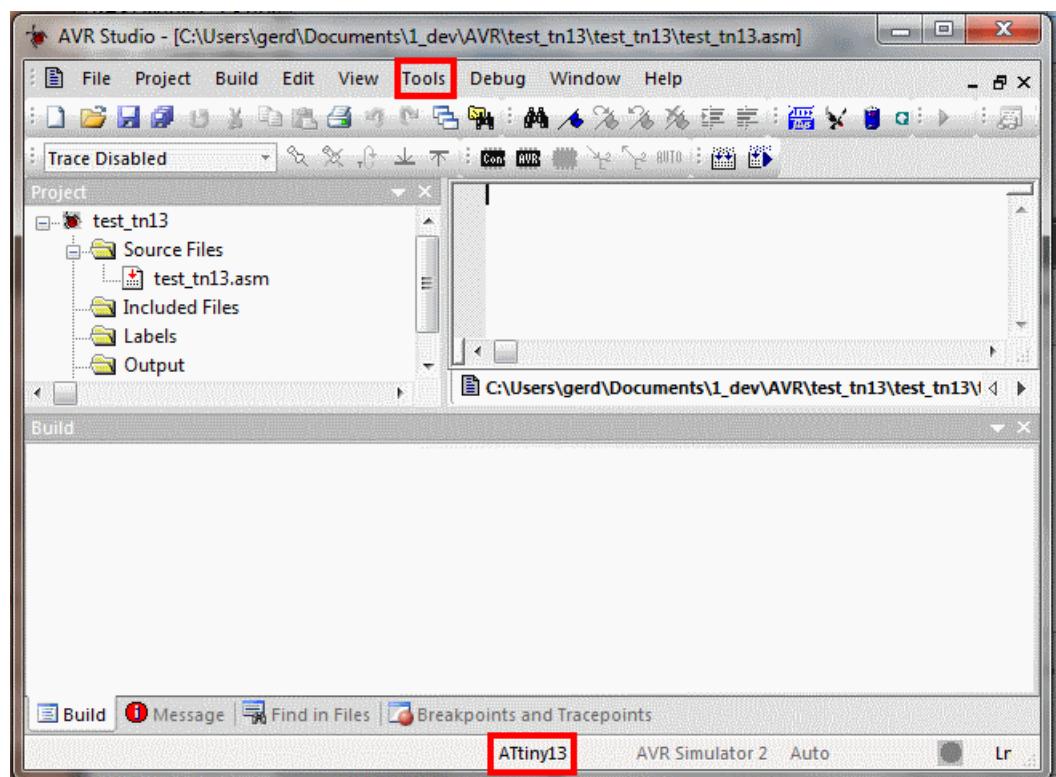


So the Studio looks right after start: it requests to generate a new project. We select an assembler project, select a meaningful name for that, let the Studio generate a source file with the same name in the selected folder and go to the Next button.



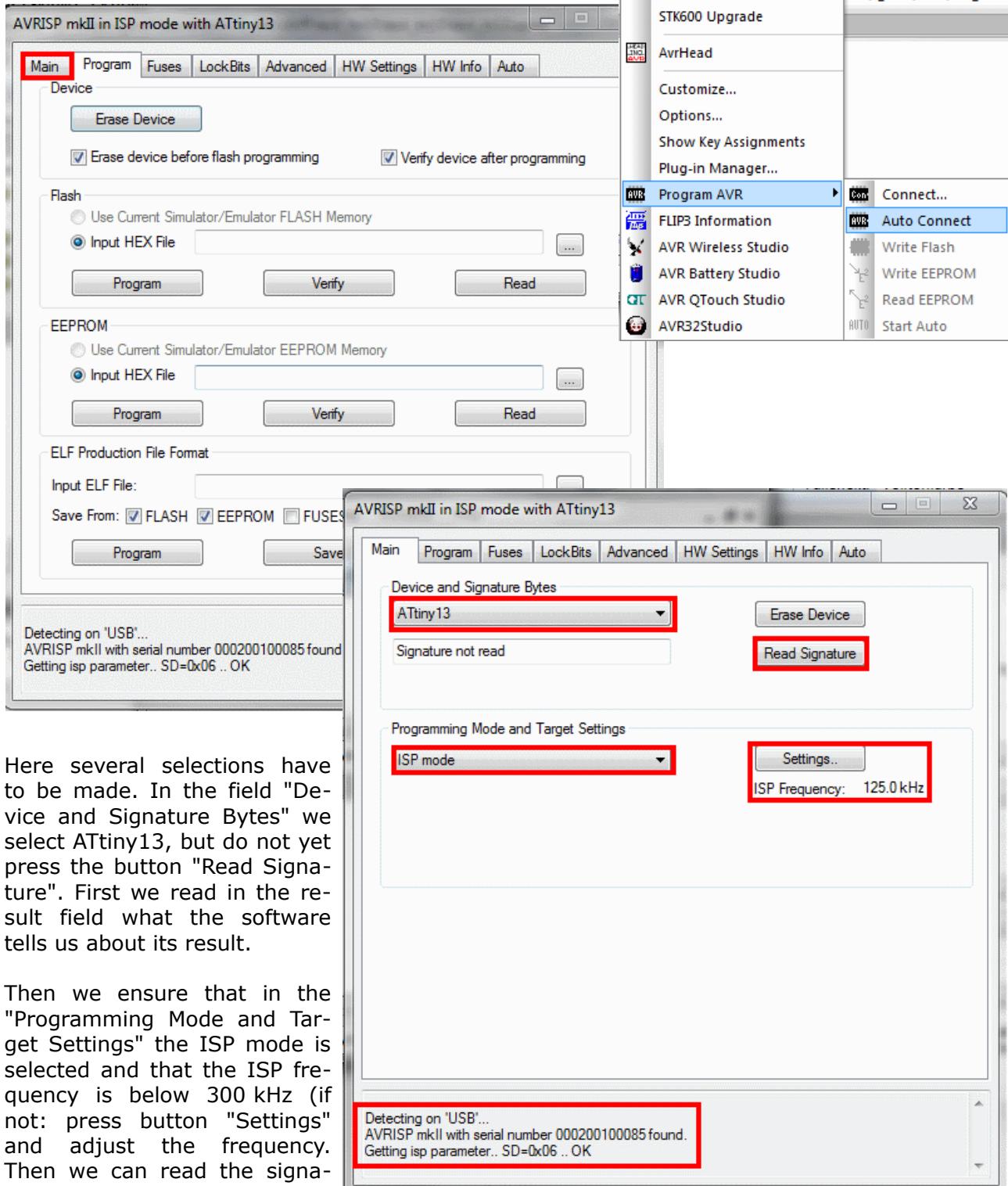
We select the simulator as platform and as device an ATTiny13.

We select "Tools" to connect the programmer interface.



After plugging the programmer to an USB jack we select within the tools menu "Program AVR" and "Auto Connect" and hope the programmer is identified automatically. If you use a STK500 clone, select a COM interface instead of a USB device.

If the USB jack is correctly identified and the programmer has told the Studio that he is an AVR-ISPmkII, the tools window opens. In that window we select the tab "Main".



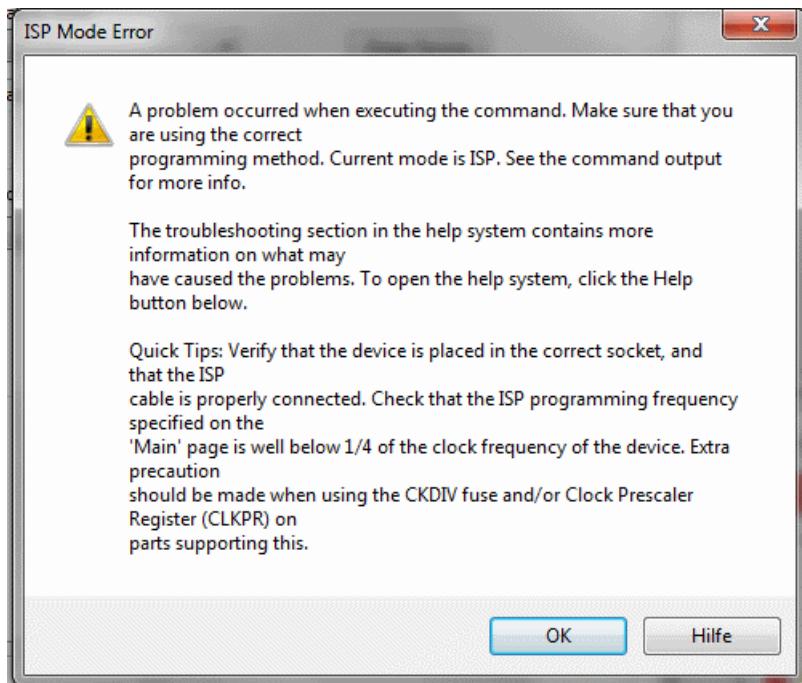
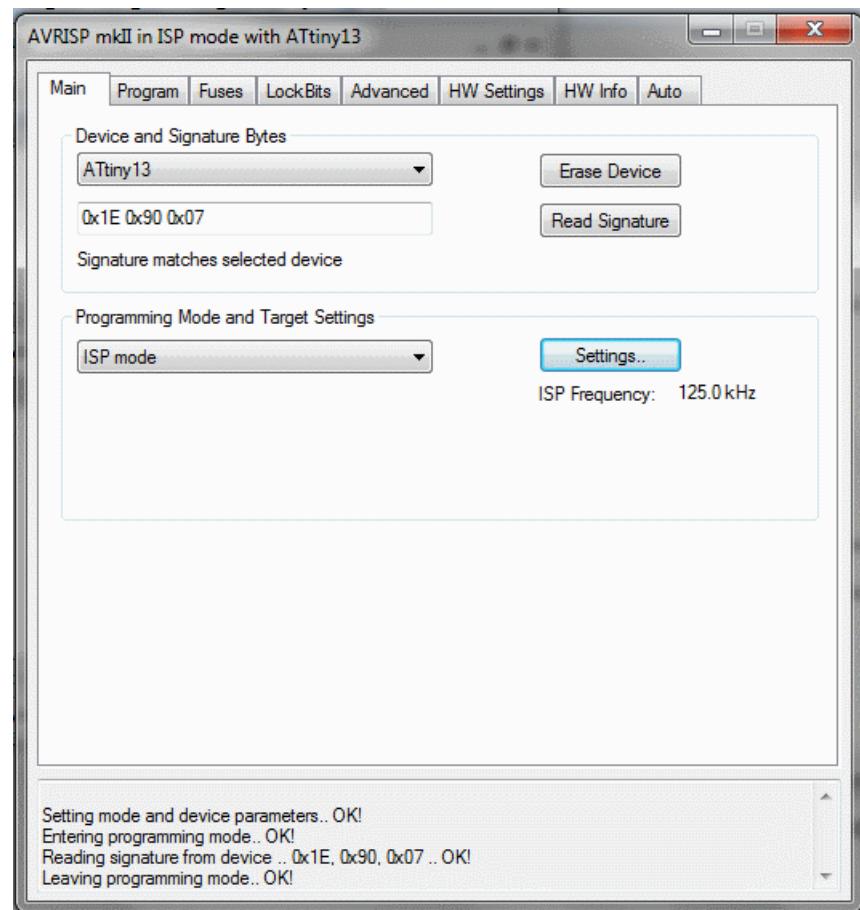
Here several selections have to be made. In the field "Device and Signature Bytes" we select ATtiny13, but do not yet press the button "Read Signature". First we read in the result field what the software tells us about its result.

Then we ensure that in the "Programming Mode and Target Settings" the ISP mode is selected and that the ISP frequency is below 300 kHz (if not: press button "Settings" and adjust the frequency. Then we can read the signa-

ture bytes of the chip.

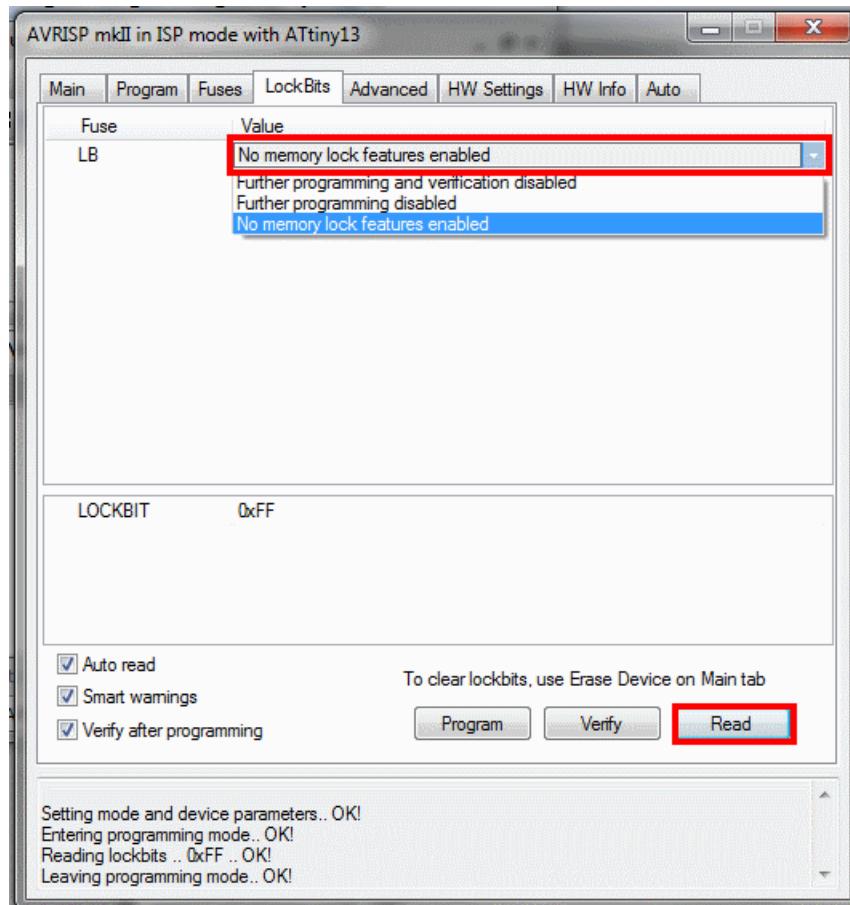
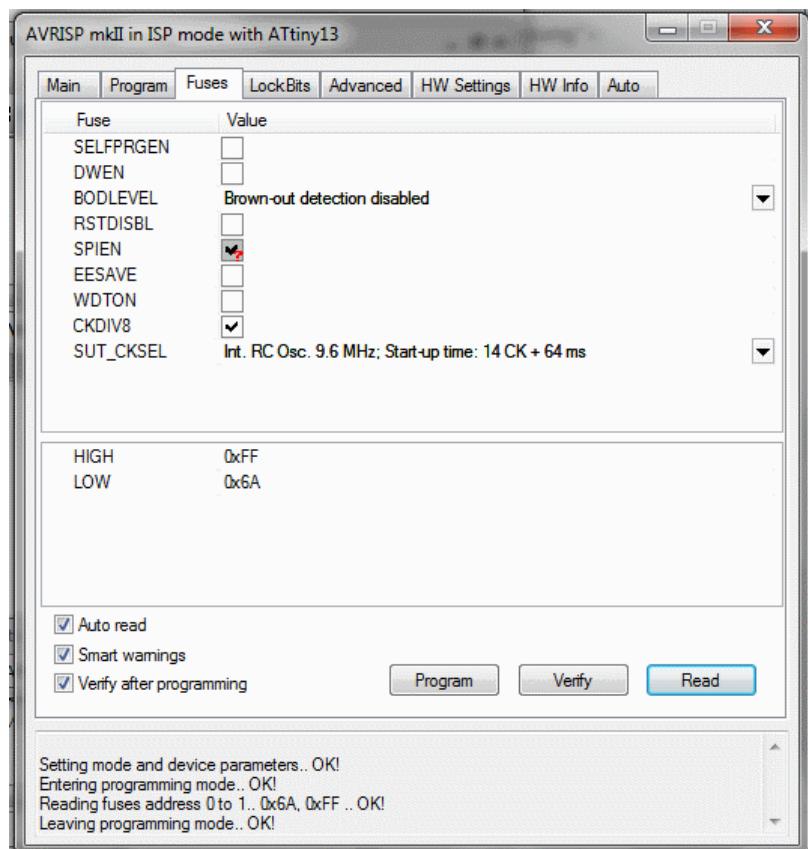
In the tab "Main" we select the ATtiny13 from the Device drop-down and press the button "Read Signature".

This is the report of the programmer. It has read the signature, compared the result with the ATtiny13 and comes to the conclusion that the signature is correct. Anything works well and all functions correct.



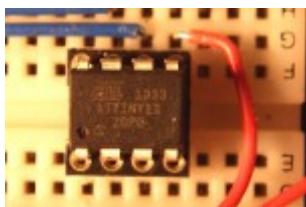
If not then this or a similar error message comes up. We then enter error message debug mode and measure voltages, connections and act like described above.

To just have a view on the fuses of Mister Professor, we select the tab "Fuses". We should not change something here because unwanted changes can have the consequence that nothing goes any more and any access to the chip vanishes. Who wants to already know at this point what the fuses do takes a view into the Device Data-book of the ATtiny13 (to download under [this address](#), 176 pages).



This here hides behind the tab "LockBits": the switches to disable reading out the flash storage. Those can only be overwritten if the whole storage is cleared (using the button in tab "Main"). Perfect software protection.

So far the internals of Mister Professor. More later on.



Lecture 2: Switch a LED on

2.0 Overview

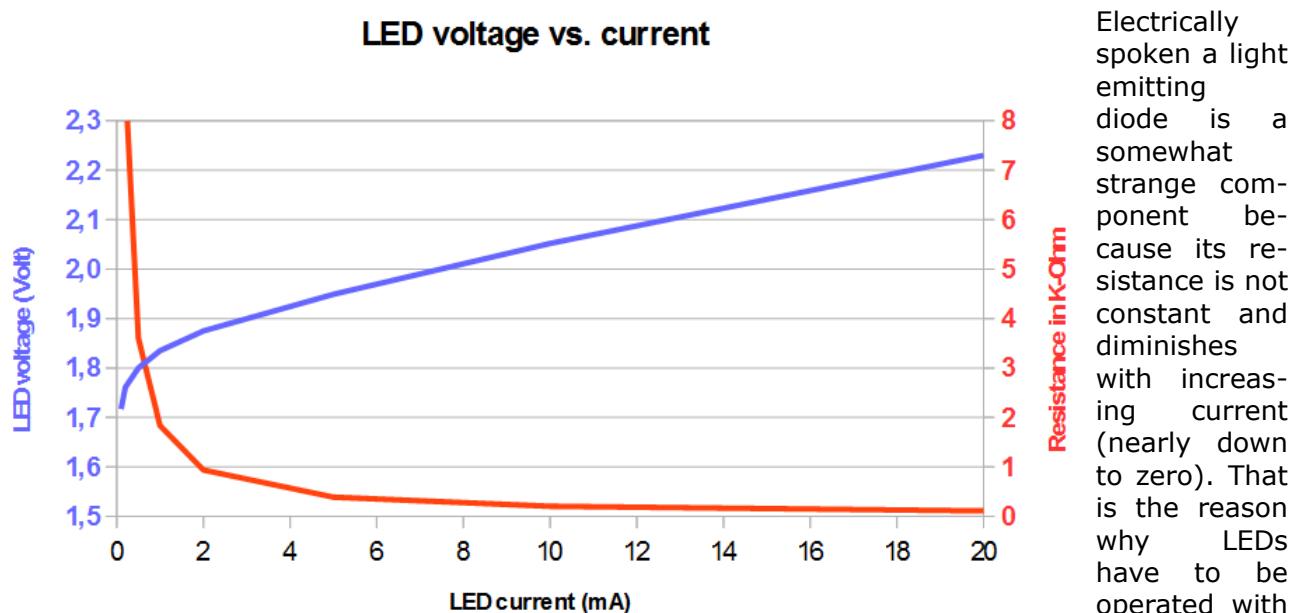
1. [Introduction](#)
2. [Components and mounting](#)
3. [Programming](#)
4. [Simulating](#)

2.1 Introduction

The first gigantic micro-controller program switches a light emitting diode LED on. We learn here how a program looks like, how it is assembled (translated to the controller's binary language), what comes out of the assembling process and how this is transferred to the controller. We learn much about LEDs and I/O pins and their manipulation.

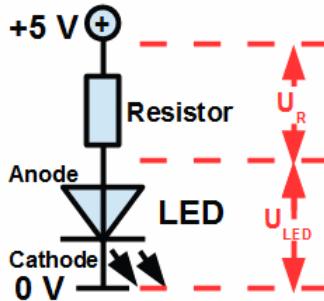
2.1.1 LEDs

LEDs send light, if current flows through them. So far so good. In a certain area "the more current the more light" applies, but this area is rather limited. There, more current does not emit more light but produces more heat. In the area with small current of a few milli-amperes (mA) the increase in emitted light is the highest, at high currents the increase is not visible any more. The human eye and its sensitivity is anything else but linear so that we cannot really see and realize "more light" and "even more light".



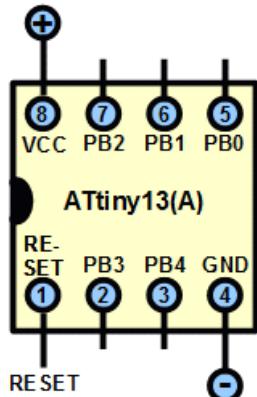
current because they are not able to regulate the current via their resistance. This limited current can be provided by a resistor or an electronic current regulator based on semiconductors.

Electrically spoken a light emitting diode is a somewhat strange component because its resistance is not constant and diminishes with increasing current (nearly down to zero). That is the reason why LEDs have to be operated with a limited current.



The value of current limiting resistors can be calculated applying Ohm's law. $R = U / I$ says that the current I increases with the voltage applied linearly. The voltage on the resistor U_R is the difference between the operating voltage (e.g. 5 V) and the voltage of the LED U_{LED} (at a given current). At 5.0 V and a LED voltage of 2.1 V the voltage on the resistor is $5.0 - 2.1 = 2.9$ V. For a current of 10 mA the resistor should be $R = 2.9 / 0.010 = 290$ Ohms.

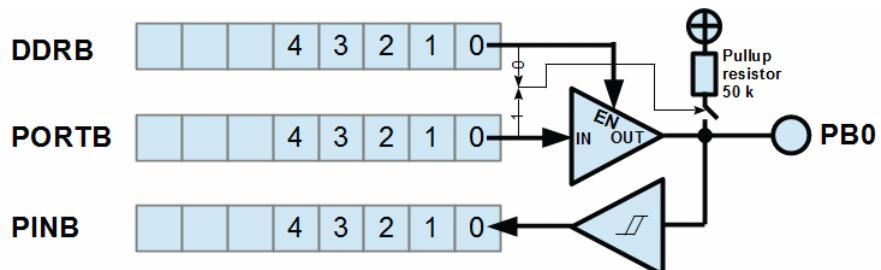
2.1.2 Controller pins as in- and output



The controller ATtiny13 has 8 pins of which five can be used as port-pins. Those are named PB0 to PB4. Each of those port pins is controlled by two bits in two internal storage places. Those storage places are named DDRB (data direction register port B) and PORTB (data output register port B). The logical state of the port pin can be read via PINB (input register B). Why the port names in the ATtiny13 start with B and not with A is ATMEL's secret. Why they named those storage places "registers" is to confuse you a little bit with other internals also named registers but those have a completely different function.

Each of the port pins can be switched to four states (in the following for external port-pin PB0):

1. Output driver off ($DDB0 = 0$), pull-up resistor off ($PORTB0 = 0$). In this mode the port-pin can be used as a high-resistance input pin. The logical state of the pin can be read in $PINB0$.



2. Output driver off ($DDB0 = 0$), pull-up resistor on ($PORTB0 = 1$). In this mode a resistor of approximately 50 k ties the voltage on the open input pin to the operating voltage of the controller. Externally a switch or push button can overwrite this and pull the input voltage down to zero volts (Ground, GND), the bit $PINB0$ reflects this level change.
3. Output driver on ($DDB0 = 1$), output bit zero ($PORTB0 = 0$). The pin now is on a very low voltage near the negative operating voltage. The pin can pull down currents of up to 50 mA, with the pin voltage slightly increasing with current.
4. Output driver on ($DDB0 = 1$), output bit one ($PORTB0 = 1$). The pin is on a high voltage slightly below the operating voltage. The pin can drive up to about 30 mA, with slightly decreasing pin voltage.

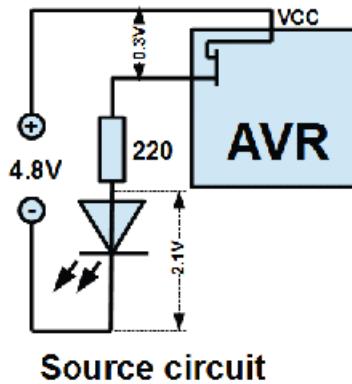
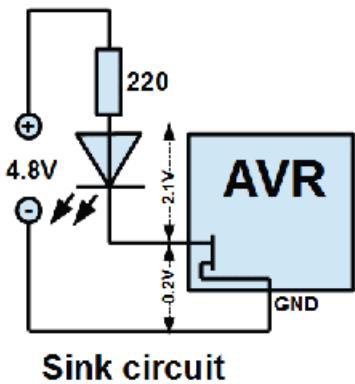
2.1.3 Controller pins as output

The task to switch on a LED with one of the controller pins can be resolved in two ways:

1. The pin is configured as output and the output bit in PORTB is set to one. In that case the LED (with its anode) and the resistor are connected to the pin and the cathode side goes to GND. The pin is high and provides current ("source").
2. The pin is configured as output and the output bit is cleared pulling the output pin to GND. Now the cathode of the LED points to the pin and the resistor is connected with

the operating voltage („sink“).

Both cases are different in that, in the first case, the output bit is high, in the second case it is low.

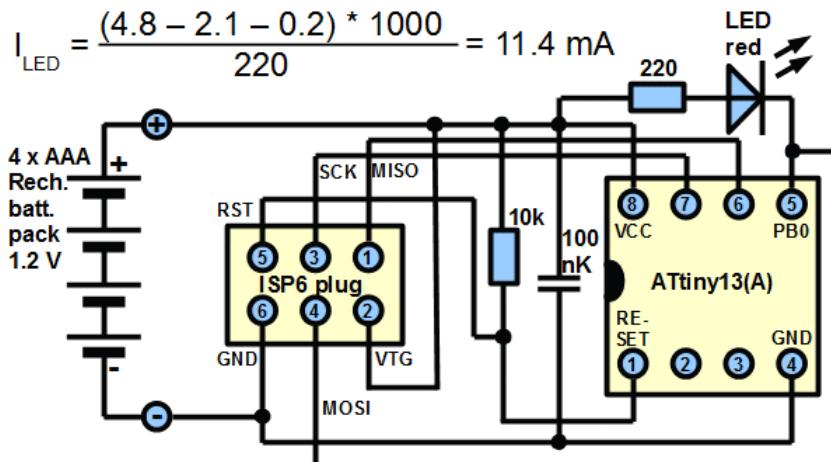


Here both opportunities are shown with the respective voltages at 4.8 V. The output driver transistors of the AVR show a slightly higher voltage difference when sourcing. At 10 mA and nearly 5 V operating voltage the difference is not very significant, but increases at higher output currents and smaller operating voltages. Please consult the electrical characteristics chapter in the device databook in those cases.

All outputs can be short-circuited without any damage. If several pins are simultaneously short-circuited, the maximum heat power of the AVR can be exceeded. If more than approximately 30 mA have to be driven, consider an external transistor driver.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Programming](#) [Simulating](#)

2.2 Hardware



To switch the LED on, the cathode of the LED is connected to PB0 and via a resistor of 220 Ohms to the positive operating voltage. The sink option is selected here: if the output pin is low the LED is on.

The formula to calculate the LED current is also given. The 4.8 V are the battery pack voltage, the 2.1 V the conducting state voltage of the LED (at approx. 10 mA current) and the 0.2 V is the AVR driver voltage in sink mode (at approximately 10 mA and 4.8 operating voltage). Larger currents make no sense because the increase in emitted light cannot be seen. Only if a larger LED is used a higher current is appropriate.

The other parts of the circuit remain the same as in the previous lecture. PB0 is used for two purposes now: for driving the LED and as MISO in the ISP programming mode. That does not cause any conflicts (higher currents might conflict with the ISP programmer!).

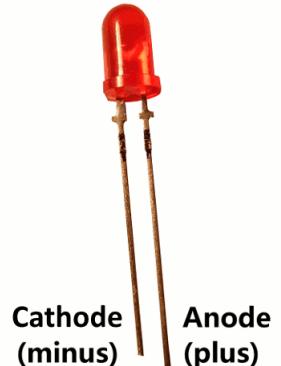
[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Programming](#) [Simulating](#)

2.3 Components and mounting

2.3.1 Components

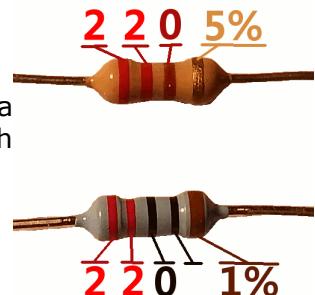
2.3.1.1 The LED

This is a 5 mm standard-LED. The longer pin is the anode. If mounted in reverse mode: the diode has a Zener voltage break-through at around 16 V, but does not emit light.

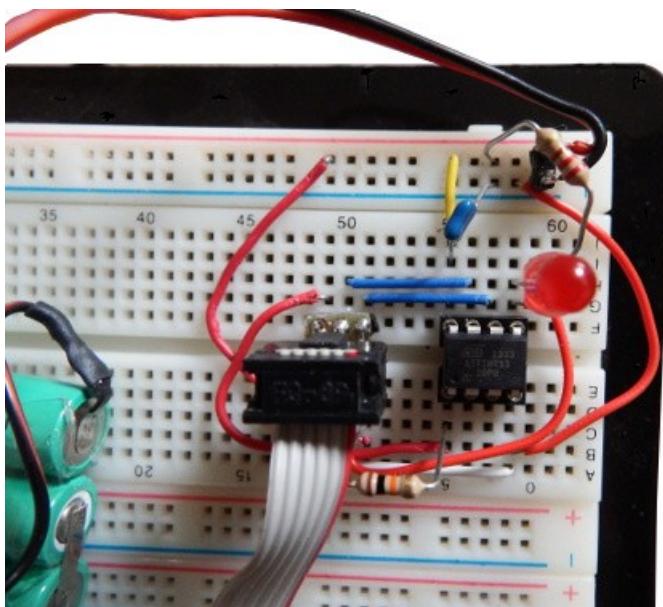


2.3.1.2 The 220 Ohm resistor

These are two different types of resistors of 220 Ohm. The upper is a carbon film resistor with 5% accuracy, the lower a metal film type with 1% tolerance.



2.3.2 The hardware



The mounting of the hardware requires that the cathode of the LED is connected with PB0 (pin 5) of the ATtiny13. The anode is tied to the neighboring empty column. From there the 220 Ohm resistor goes to the plus bar. That is it.

Even under power: nothing happens. The reason is that a native ATtiny13 switches off all his port pins. To switch those ports on, program code has to be executed.

2.4 Programming

2.4.1 Program storage

To animate the ATtiny13 he has to be programmed. In this case with the two hexadecimal words 9AB8 and 98C0. In binary language (which the controller natively speaks) this corresponds to 1001.1010.1011.1000 and 1001.1000.1100.0000. The useful place for those 32 bits, four bytes or two words is at the beginning of its program storage area (flash), on addresses 0000 and 0001. If you buy a new device (or if you erase the flash area) there is nothing useful there. As a storage cannot be empty (there is always something there), it is filled with hexadecimal FFFF in all those cells. Factually the controller reads the FFFFs, decodes those

and executes them. But with the result to do nothing. The same operation happens, if he reads a 0000 from its flash: doing nothing. This operation is called "NOP" or "No operation". NOP by the way is a mnemonic representation for the binary 0000.0000.0000.0000 or hexadecimal 0000. All things that the controller can do (execute instructions) have such a mnemonic, as we will see later on.

The program storage or flash memory of the ATtiny13 has 1024 bytes, into which 512 instruction words fit. That sounds not very large, but in assembler this is a large bunch. Our most complicated program will have several tens of instruction words, and in dozens of projects I never reached the limit of the flash. My stepper motor application had only 141 instruction words, and this has to perform complex timing, counting and AD conversion tasks in parallel. One does definitely not need more if you avoid ineffective C style programming.

2.4.2 Source code

Because binary codes such as 9AB8 and 98C0 are not easy to remember understandable representations have been defined that are easier to remember. In this language the software engineer writes the two lines

```
sbi DDRB,DDB0
cbi PORTB,PORTB0
```

into a textfile ([commented source code file here](#)). From these two lines the assembler, a translation program, generates the two instruction words 9AB8 and 98C0 for the controller.

The abbreviations sbi and cbi are not understandable either, but if you write

```
SBI: Set bit DDB0 in I/O port DDRB, the data direction bit of pin PB0 (DDB0), to one.
CBI: Clear bit PORTB0 in I/O port PORTB, the data bit of pin PB0, to zero.
```

That sounds more understandable. With that background knowledge the mnemonics SBI and CBI are memorable.

In assembler, as with the windows file-system, upper and lower letters are not discriminated. That gives us the opportunity to signal the type of symbols by our own. In the above formulation instructions are in lower letters, symbols in upper letters.

Each line in the source code (e.g. "sbi DDRB,DDB0") is exactly one instruction of the controller. Only those instructions that the central processing unit (CPU) of the controller physically masters have a representing mnemonic. This is specific for assembler: it depends completely from the abilities of the controller. While in other languages instructions can be generated and named by the author (such as subroutines or functions), no such ability is given in assembler. Each line is exactly one physical operation of the controller.

That is the reason why assembler instructions differ only minor with different dialects. What the CPU understands and handles is not so different. Each CPU can add two binary numbers. In each assembler dialect the mnemonic for this might be different, but the basic processing is virtually the same. Those that have learned AVR assembler might well switch to PIC assembler. All that is to be learned are the slightly different mnemonics (and specific abilities of the CPU). The principle, one mnemonic translates to one CPU operation, remains the same.

2.4.3 To assemble

In the code line "sbi DDRB,DDB0" the type of "sbi" stands for an instruction while "DDRB" and "DDB0" are parameters for this instruction. The first parameter is the data direction port of port B, the second is the bit position in that port to be set to one. The device data handbook for the ATtiny13 says about this:

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x18	PORTB	-	-	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x17	DDRB	-	-	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0x16	PINB	-	-	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

DDRB therefore translates to port number 17 (hexadecimal, that is 23 decimal). DDB0 translates to hexadecimal 0 or bit 0. Both parameters stand for numbers. We do not need to remember those numbers if we use the symbol names DDRB and DDB0 instead. Avoid to learn and use the numbers, because in a different device some of those numbers might be different from those in an ATTiny13.

The ports DDRB and PORTB are listed in the device data-book as follows:

DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	-	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Initial Value	0	0	0	0	0	0	0	0	

PORPB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	-	PORPB5	PORPB4	PORPB3	PORPB2	PORPB1	PORPB0	PORPB
Initial Value	0	0	0	0	0	0	0	0	

R/W means that those bits can be read (R) and written (W). The "initial Value" means that this bit will be cleared (0) or set (1) during the reset sequence of the controller.

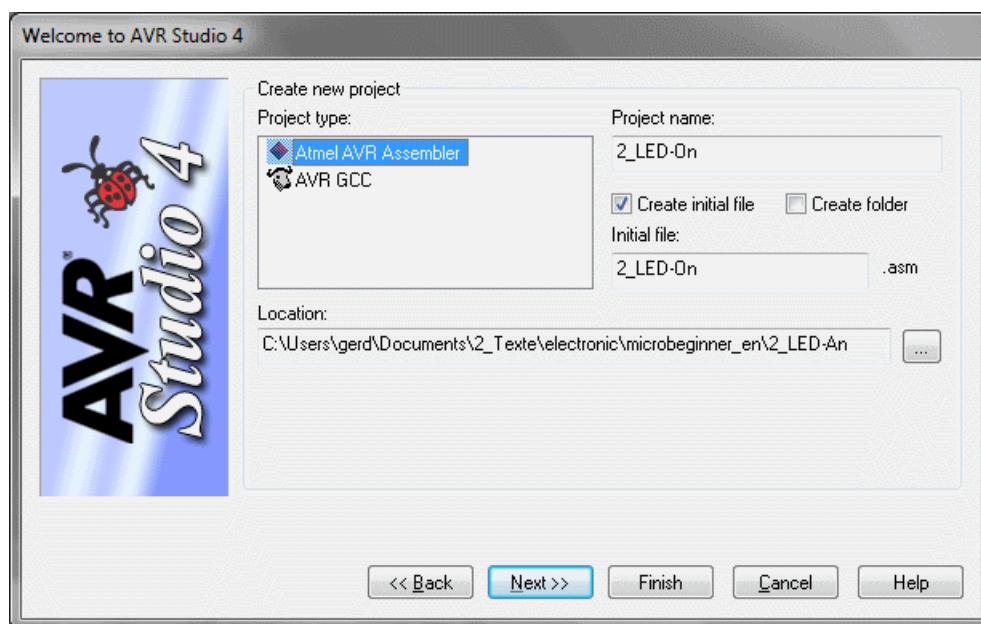
2.4.4 Writing source code

```

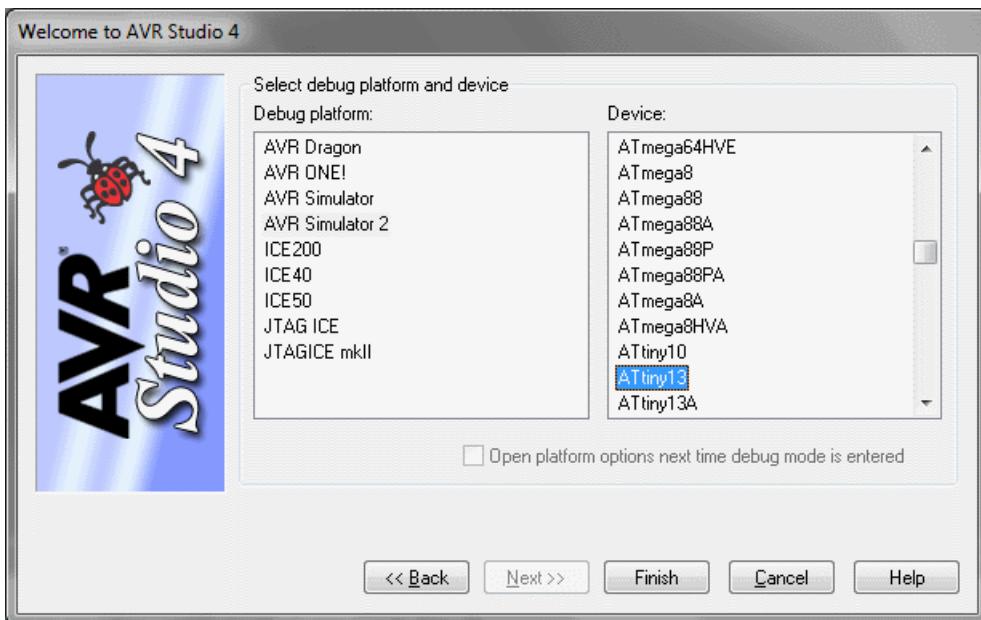
2_LED_On_notepad.asm - Editor
Datei Bearbeiten Fformat Ansicht ?
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
    sbi DDRB, DDB0
    cbi PORTB, PORTB0
<>

```

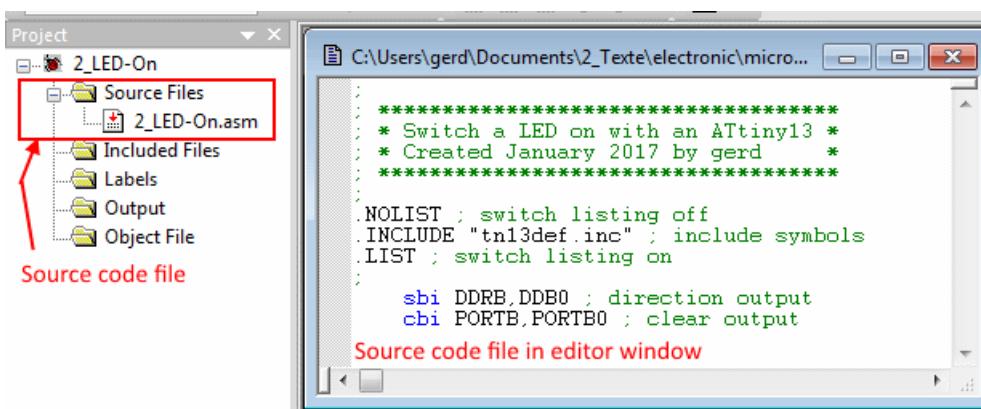
To write assembler source code one needs a simple text editor. No special software, just an editor that writes ASCII chars to a file. Such as Notepad (under windows, see picture to the left) or KWrite (under linux KDE). And name this text file "somehow.asm", to just better remember what is written in there and for which purpose. Please do not use a text-writer such as Word or OpenOffice to write source code, those do not store plain text and are confusing the assembler with formatting information.



Those who want it more comfortable write the source code in the editor window of ATMEL's studio (here: older version 4). First we open a new project, select the name of the project and the source code file and its location.



Then we select the simulation platform and the device (irrelevant in this case).



In the editor window to the right we type in the source code.

Lines with a semi-colon are comments that are ignored by the assembler. They are only useful for the human reader.

The directive `.INCLUDE "tn13def.inc"` reads in a file where all symbol representations of numbers for the device type ATTiny13 are defined, in our case the ports DDRB and PORTB and the port bits DDB0 and PORTB0. If we would skip this line, the assembler does not know these symbols and error messages would result. The combination of the directives `".NOLIST"` and `".LIST"` switches the output in the listing off and on so that the include does not produce output in the listfile. If you leave these two lines out you can see in the listing all symbols that are defined for the ATTiny13.

To ease the recognition the editor of the Studio uses different colors for instructions, parameters, directives and comments. This is called syntax-highlighting.

2.4.5 To assemble

There are many ways to assemble these source code files. Many assemblers are available, I recommend my own command line version, gavrasm. It is available [here for download](#), versions for windows and linux as well as the source code written in fpc-Pascal is available.

With gavrasm open a windows command line or a linux bash shell and type two commands. The first command, "cd path to source file", is necessary to introduce the path where the source code file is located. The second command, "[path to gavrasm]\gavrasm.exe -seb source.asm" calls gavrasm to assemble the source code file.

```
ca Eingabeaufforderung
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\gerd>cd C:\Users\gerd\Documents\2_Texte\electronic\microbeginner_en\
C:\Users\gerd\Documents\2_Texte\electronic\microbeginner_en>C:\Users\gerd\Documents\1_dev\fpc\gavrasm\w36\gavrasm.exe -seh 2_LED_On.asm
```

```
ca Eingabeaufforderung
! gavrasm gerd's AVR assembler Version 3.6 <C>2017 by DG4FAC !
+-----+
Compiling Source file: 2_LED_On.asm
-----
Pass:      1
15 lines done.
Pass 1 ok.
-----
Pass:      2
Warning 009: Include defs not necessary, using internal values!
  File: 2_LED_On.asm, Line: 8
  Source line: .INCLUDE "tn13def.inc" ; Read port definitions
15 lines done.

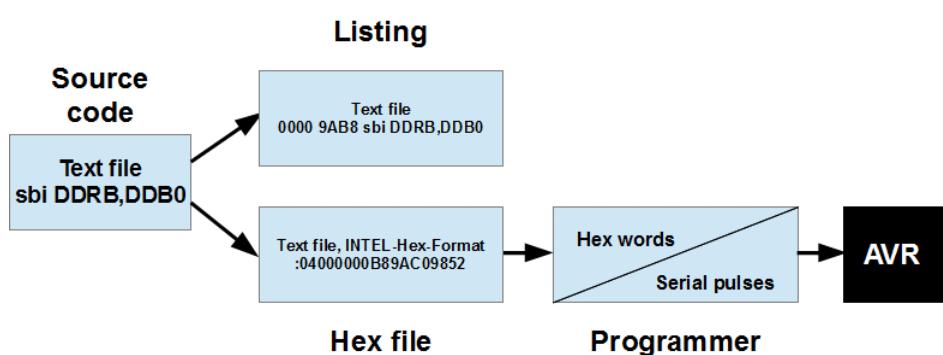
2 words code, 0 words constants, total=2 =  0.4%
One warning!
Compilation completed, no errors. Bye, bye ...
C:\Users\gerd\Documents\2_Texte\electronic\microbeginner_en\html\2_Led_On>
```

This shows the assembler at work. Finally it comes out with "No errors" and one warning. The warning is that gavrasm knows all symbols of all AVR devices internally and does not read the file "tn13def.inc". That makes

this assembler independent from the def.inc files of ATMEL, and it works under Linux or MacOS or whatever operating system. Only gavrasm provides this service. So you can finally ignore the warning.

If you have to assemble the same source code file over and over again it is useful to place these two call into a batch file that can be started by clicking on it. [This provides such an example batch file for windows.](#)

gavrasm produced two new files after successful completion: a listing and a hex file. The files are of the same name as the source but have different extensions: ".lst" and ".hex". Both are simple text files and can be viewed with any simple text editor. The hex file can be used to



program the controller's flash memory.

```

2_LED_On.lst - Editor
Datei Bearbeiten Format Ansicht ?
gavrasm Gerd's AVR assembler version 3.6 (c)2017 by DG4FAC
-----
Source file: 2_LED_On.asm
Hex file: 2_LED_On.hex
Eeprom file: 2_LED_On.eep
Compiled: 11.01.2017, 15:32:20
Pass: 2
1: ; ****
2: ; ***** Switch a LED on with an ATtiny13 ****
3: ; * (C)2017 by http://www.avr-asm-tutorial.net *
4: ; ****
5: ; ****
6: ;
7: .NOLIST ; output of listing off
10: ;
11: 000000 9AB8 sbi DDRB,DDRB0 ; set portpin PB0 as output
12: 000001 98C0 cbi PORTB,PORTB0 ; clear portpin PB0
13: ;
14: ; End of source code
15: ;

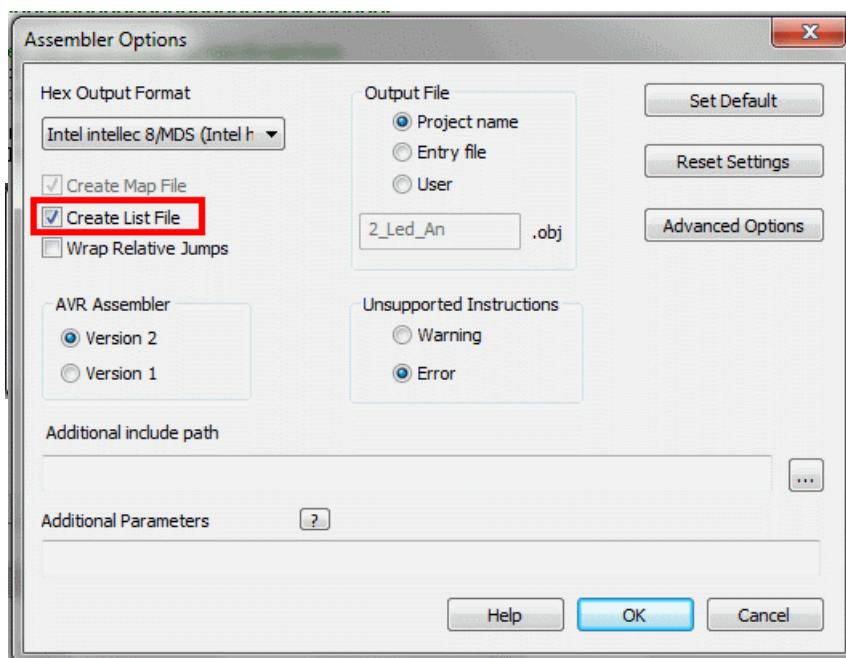
List of symbols:
Type nDef nUsed Decimalval Hexvalue Name
T 1 1 18 12 ATTINY13
No macros.

Program : 2 words.
Constants : 0 words.
Total program memory: 2 words.
Eeprom space : 0 bytes.
Data segment : 0 bytes.
Compilation completed, no errors.
Compilation endet 11.01.2017, 15:32:20

```

This is the assembler listing. It shows that the sbi instruction translates to hex 9AB8 at address 000000 and cbi to 98C0 at address 000001.

The list of symbols in the lower part of the listing (switched on with the -s parameter on the command line) says that only one symbol is defined: the ATtiny13 with a T type is defined once (column nDef), is once used (column nUsed) and has an internal value of decimal 18. Other types of symbols (registers, constants) etc. would occur here if defined and used. This symbol table is only provided by gavrasm, no other AVR assembler has that.



With the Studio the creation of the listing has to be switched actively on because it is disabled by default.

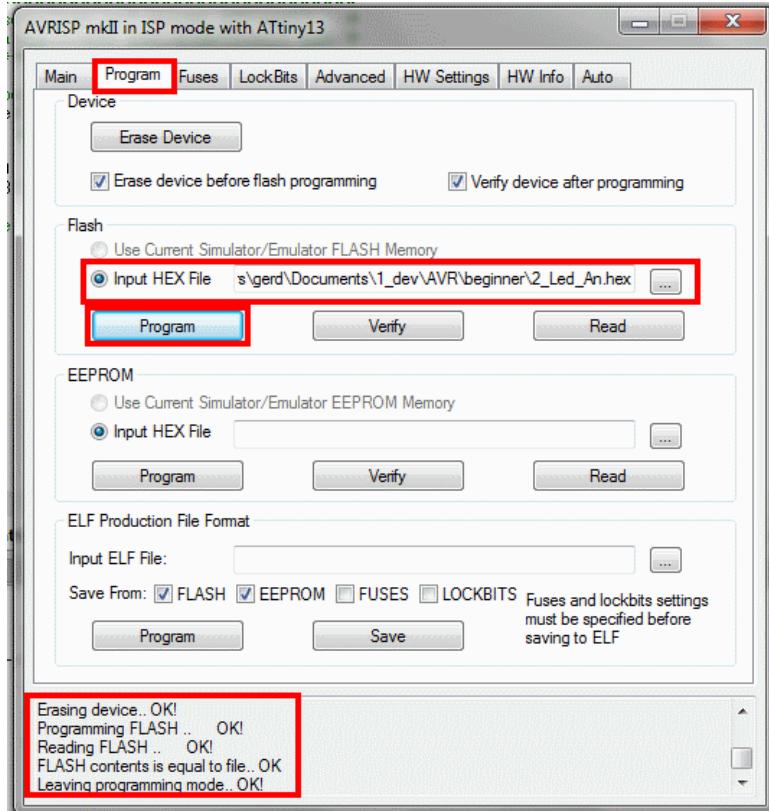
```

2_LED_On.hex - Editor
Datei Bearbeiten Format Ansicht ?
:020000020000FC
:04000000B89AC09852
:00000001FF

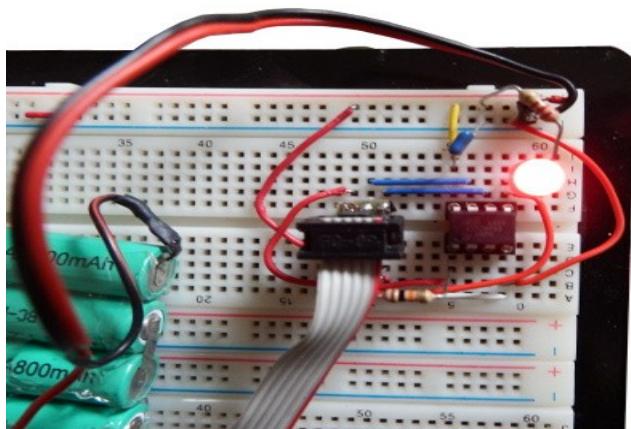
```

This is the generated hex code in readable form, in Intel hex format. This provides the naked code with addresses, byte-wise coded instruction words and a check byte on each line. We do not have to care about this file, it works with every programmer.

2.4.6 To write the hex code to the program flash storage



To transfer the hex file's content to the controller, one has to start the Studio, open the tools section and select "Autoconnect" there. In the tab "Main" we assure that the controller is accessible and of the correct device type. Then we go to the tab "Program". There we select the "Input Hex File" by clicking on the small square to the right of the input field. The button "Program" initiates the programming.



After some on and off of the LED (the programming pulses) the LED is permanently on. The two code lines finally work and do what they are supposed to do.

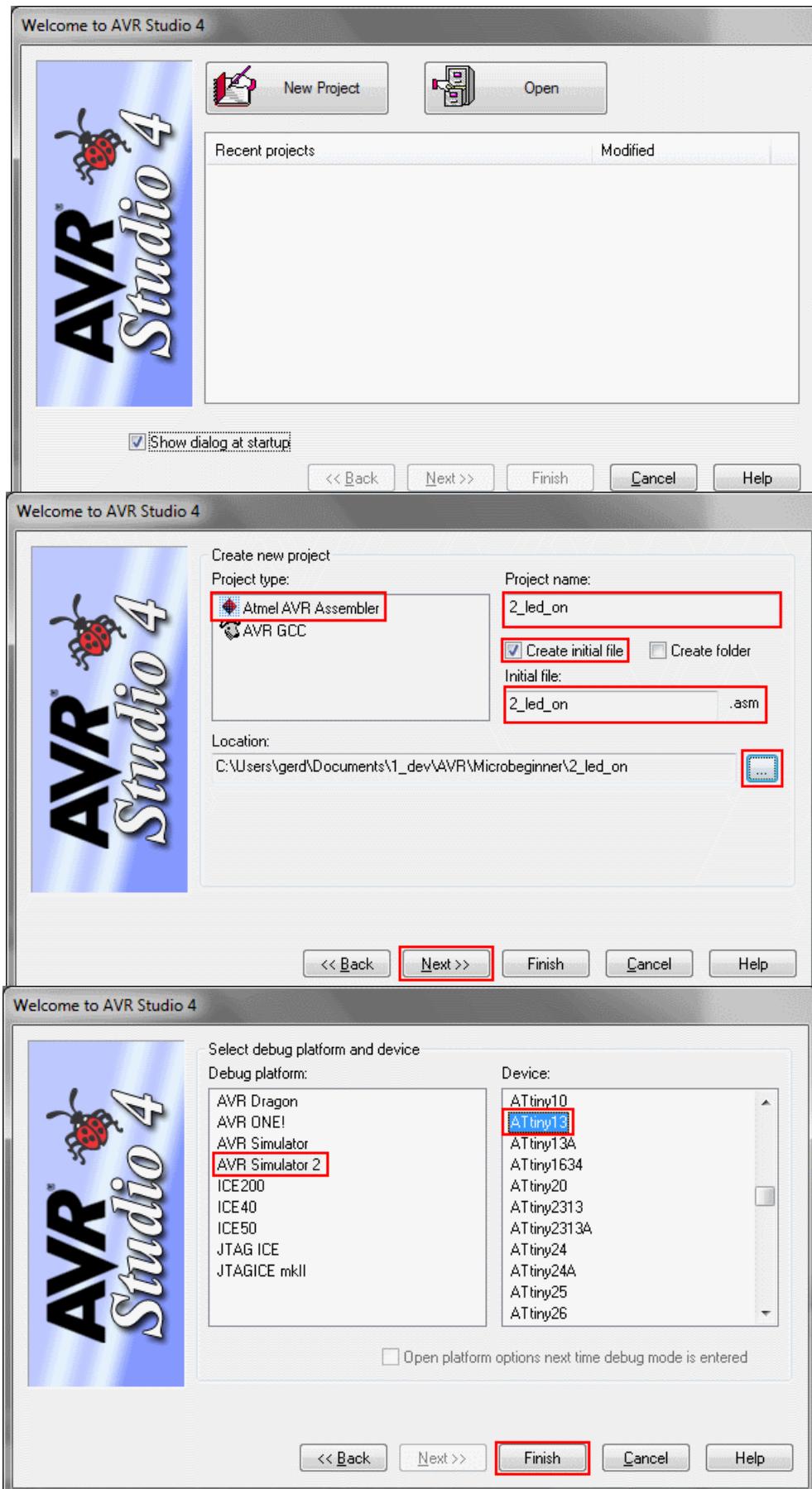
2.5 Simulating program execution

Simulation is to step through an assembler source code and to view what happens within a micro-controller in those steps. It is not a real microcontroller but a simulated one.

The following describes two opportunities to simulate program execution:

- with ATMEL's Studio, running under the Windows operating system only, and
- with the software avr_sim, running under Windows and Linux operating systems.

2.5.1 Simulating with ATMEL's Studio



ATMEL's studio can be downloaded from their [website](#). The following describes simulation with version 4, later versions changed the design but work in a similar manner.

Starting the Studio displays the window to the left. Press the NEW button to start a new project.

This opens the new project page, where you have to select AVR assembler as project language, input a project name, select a location for the project on your hard-drive and continue with the button "Next".

In the "Debug platform" section select Simulator 2, in the "Device" section ATtiny13. Click "Finish" to continue.

```

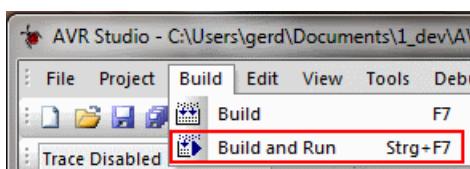
; ****
; * Switch a LED on with an ATtiny13 *
; * Created January 2017 by gerd   *
; ****

.NOLIST ; switch listing off
INCLUDE "tn13def.inc" ; include symbols
.LIST ; switch listing on

    sbi DDRB,DDB0 ; direction output
    cbi PORTB,PORTB0 ; clear output

```

In the opening source code editor type in your led-on source code.



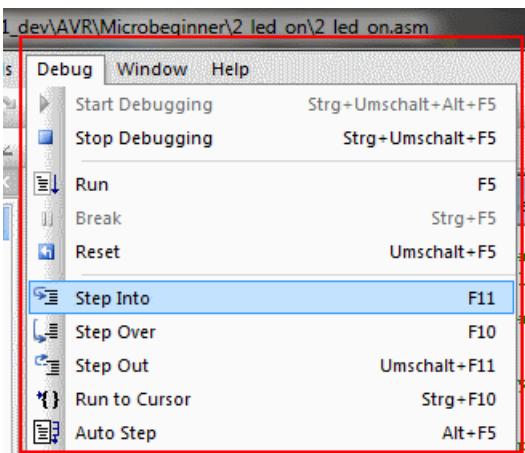
In the menu select "Build" and "Build&Run".

The simulator starts. It displays a false frequency. The program counter (yellow arrow in the editor field) points to the first executable instruction. If you click on "Port B" in the "I/O view" the location and content of the three port registers of Port B are displayed (which are all cleared by the reset).

Name	Value
Program Counter	0x000000
Stack Pointer	0x9F
Xpointer	0x00
Ypointer	0x00
Zpointer	0x0000
Cycle Counter	0
Frequency	1.0000 MHz
Stop Watch	0.00 us
SREG	<input checked="" type="checkbox"/> <input type="checkbox"/>
Registers	

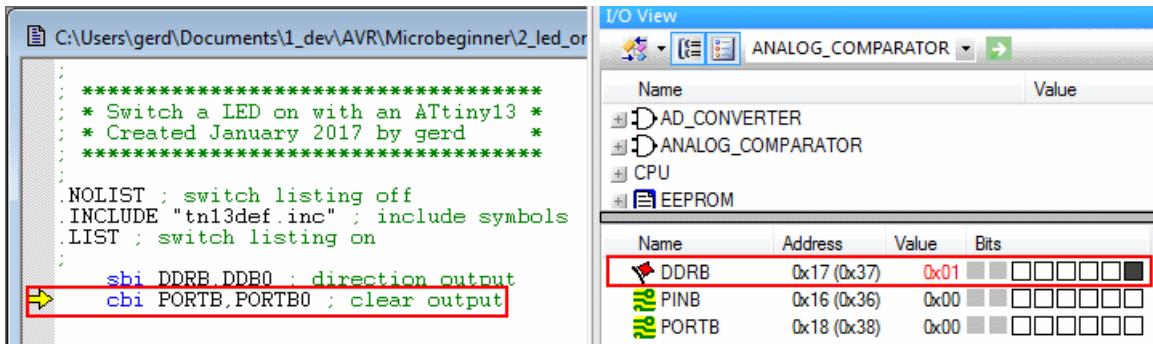
Name	Value
AD_CONVERTER	0x00
ANALOG_COMPARATOR	0x00
CPU	0x00
EPPROM	0x00
EXTERNAL_INTERRUPT	0x00
PORTB	0x00
Data Register, Port B	0x00
Data Direction Register, Port B	0x00
Input Pins, Port B	0x00
TIMER_COUNTER_0	0x00
WATCHDOG	0x00

Name	Address	Value	Bits
DDRB	0x17 (0x37)	0x00	<input type="checkbox"/>
PINB	0x16 (0x36)	0x00	<input type="checkbox"/>
PORTB	0x18 (0x38)	0x00	<input type="checkbox"/>



By clicking on "Step into" in the "Debug" menu entry one step is executed.

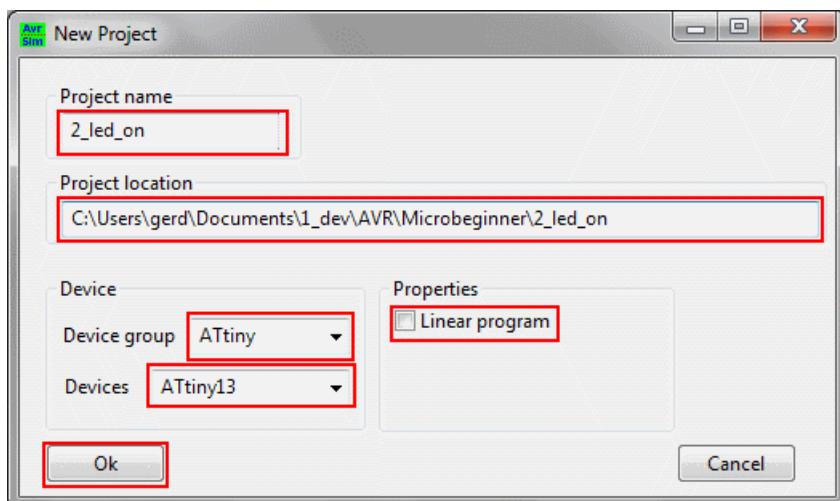
The cursor has moved one instruction further and Bit 0 of the port register DDRB has been set. The output of the pin PB0 now follows the state of the PORTB0 port register PORTB.



By this tool we can step through an assembler program and we can view the internal controller hardware in detail and see what the instructions change. Simulation therefore is a powerful tool to inspect the internal execution of an assembler program.

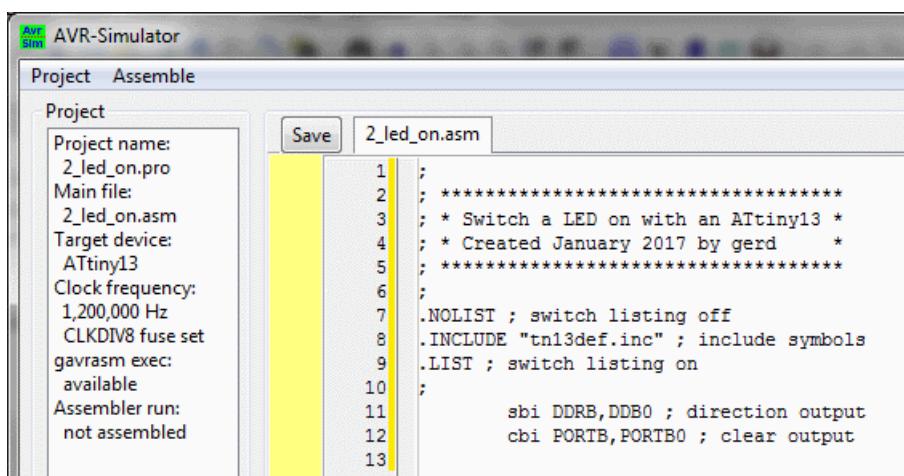
2.5.2 Simulating with avr_sim

Another simulation tool is `avr_sim`, for which the Pascal source code and Windows as well as Linux executables are available at [my website](#). Its use is described in a handbook, also available at this site.



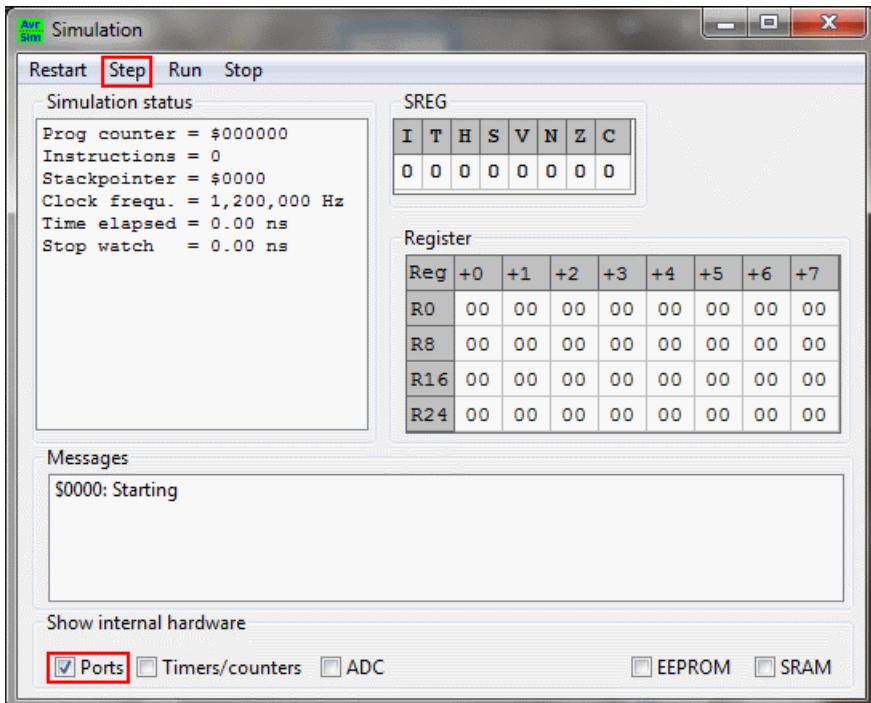
The menu entry "New" in the main menu entry "Project" in `avr_sim` opens this window. Enter your project name, disable interrupts, select the AVR type and device and click "Ok".

If a window appears that offers to select a package type, select the PDIP/SOIC package.



Delete the whole default entries and enter your source code. Save the code by clicking "Save".

Now assemble the source code by clicking "Assemble". A message box shall appear that summarizes the result of the assembly process. And a listing appears as an additional tab entry.



By selecting "Simulation" the main simulation window opens. It provides information on the running simulation process (correct values unlike ATMEL's Studio, e.g. the correct stackpointer address when not initiated, the clock frequency on which the device runs, in that case 1.2 MHz).

By clicking on the "Ports" checkbox on the simulation window the port view opens and displays the port register contents.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	0
PINB	0	0	0	0	0	0	0	0
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0

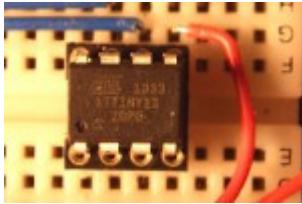
Previous B Next

By clicking on "Step" in the simulation window the first instruction is executed and the bit 0 of the data direction of port B is set. Accordingly the Port B's input register is set low for bit 0 because PORTB0 is zero.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	Lo
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0

Previous B Next

avr_sim opens some interesting opportunities to debug assembler programs.



Lecture 3: A LED is blinking

With this lecture animation starts: the LED is going on and off. First with a high frequency, then slowly in one second.

3.0 Overview

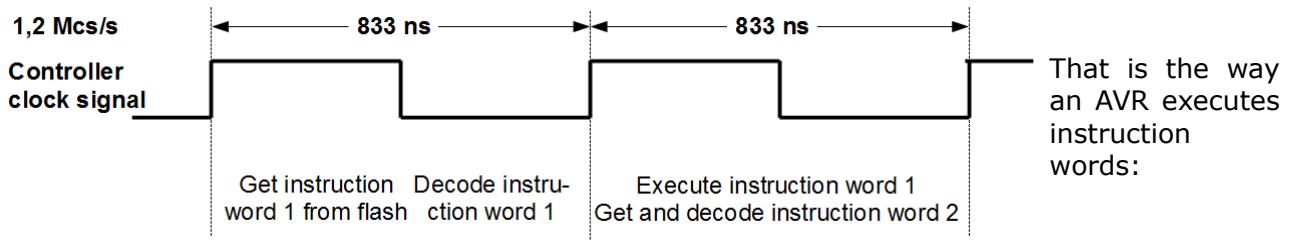
1. [Introduction](#)
2. [Hardware, components, mounting](#)
3. [Fast blinking](#)
4. [Second blinking](#)

3.1 Introduction

To blink a LED this has to be driven off and on. The basics of switching an output port has already been described in lecture 2: cbi PORTB,PORTB0 switches the LED on, sbi PORTB,PORTB0 off. That is it.

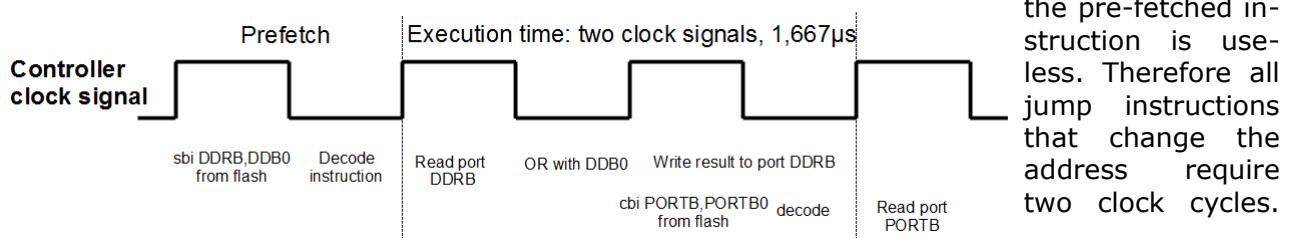
Unfortunately a controller with a clock frequency of 1.2 Mcs/s, on which the ATtiny13 is working by default, requires for those two operations only $4 / 1.200.000 \text{ seconds} = 0.000,003,33 \text{ seconds}$. This is much much too fast for the human eye to realize. The solution, to engage the controller with something else in between, is demonstrated here.

3.1.1 Execution of instructions by the controller



1. The next instruction word is read from the flash storage.
2. The instruction word is decoded into stages to be executed.
3. The instruction word is executed. During execution the next instruction word is read and decoded ("Pre-Fetch").

Actually the execution of an instruction word requires two clock cycles. But this is halved by pre-fetching the next instruction already during execution of the previous, so effective execution requires only one cycle. This works in general but not in cases when the instruction word is changing execution address (in case of jumps to somewhere else in the code). In this case



Because of the pre-fetch AVR execute double as fast than without. If one compares different controller types and their clock cycles this has to be taken into account.

Nearly all instructions of the AVR controllers are executed in a single clock cycle. But a few

ones require two (or even more). The ones that we used in the previous lecture and which we use to blink the LED, SBI and CBI belong to this rare species. This results from reading in the whole port first, then setting or clearing a single bit in that byte and rewriting the result back to the port. The execution with pre-fetch requires two clock cycles.

3.1.2 Execution times of instructions

The exception times of all instructions of an AVR are listed in the device databook in the table "[Instruction Set Summary](#)". In the column "Clocks" the number of clock cycles is listed.

21. 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
ADIW	Rd, K	Add Immediate to Word	$Rdh:Rdl \leftarrow Rdh:Rdl + K$	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	$Rd \leftarrow Rd - Rr$	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	$Rd \leftarrow Rd - K$	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	$Rd \leftarrow Rd - K - C$	Z,C,N,V,H	1
SBIW	Rd, K	Subtract Immediate from Word	$Rdh:Rdl \leftarrow Rdh:Rdl - K$	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	$Rd \leftarrow Rd \bullet Rr$	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	$Rd \leftarrow Rd \bullet K$	Z,N,V	1

Whenever it comes to execution times and exact timing, like in the case of the second blinking, this is the relevant information source.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Fast blinking](#) [Second blinking](#)

3.2 Hardware, components and mounting

For blinking the LED we use the same hardware as before in lecture 2. Nothing to be added or changed here.

3.3 Fast blinking

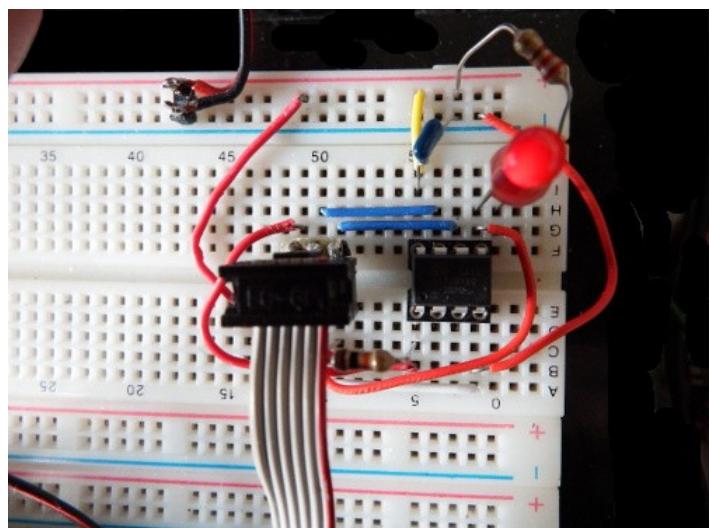
3.2.1 The simple fast blinker

As described at the beginning we can just add "sbi PORTB,PORPB0" to the source code and we are done:

```
sbi DDRB,DDRB0 ; PB0 output driver enable
cbi PORTB,PORPB0 ; LED on
sbi PORTB,PORPB0 ; LED off
```

The result would be disappointing: a nearly dark LED. We would not see much of the LED because it is ON just for a too short period.

The reason for this is that the diode is just for two clock cycles active, followed by 511 inactive clock cycles where the controller reads the empty flash storage, finds 0xFFFF there and does nothing. Until he reaches the end of the flash and starts new at address 0x0000.



Two things can be learned here: first, a controller cannot do nothing, he is executing on and on (the SLEEP instruction where the controller is send to bed is later introduced and used), and second that we need a mechanism to restart the CBI/SBI sequence on and on.

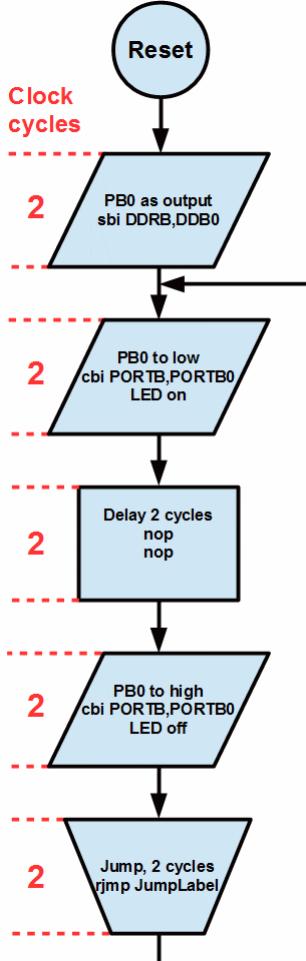
The mechanism to start back with CBI after SBI has been executed is a jump instruction, that redirects the execution address to the CBI instruction. As the address space of the flash is small we use the relative jump instruction RJMP, which can jump forward and backward by more than 2000 instructions. The code would look like this:

```
sbi DDRB,DDB0 ; PB0 as output, driver stage on, 2 clock cycles
Loop:
    cbi PORTB,PORTB0 ; LED on, 2 clock cycles
    sbi PORTB,PORTB0 ; LED off, 2 clock cycles
    rjmp Loop ; Jump relative back to label Loop, 2 clock cycles
```

The label "Loop:" is the jump address. Labels always end with ":". The instruction RJMP calculates a relative displacement between the current address and the label's address and inserts this automatically into the RJMP's binary representation, so we do not have to care about this.

The formulation above has a disadvantage: the LED is on for two cycles and off for four cycles. To correct this we insert two NOP between CBI and SBI, this delays the LED-Off and gives a 50% duty cycle rectangle of four ON and four OFF cycles.

```
sbi DDRB,DDB0 ; PB0 as output, driver stage on, 2 clock cycles
Loop:
    cbi PORTB,PORTB0 ; LED on, 2 clock cycles
    nop ; do nothing, 1 clock cycle
    nop ; do nothing, 1 clock cycle
    sbi PORTB,PORTB0 ; LED off, 2 clock cycles
    rjmp Loop ; Jump relative back to label Loop, 2 clock cycles
```



The instruction NOP in fact is simply delaying execution for one clock cycle and it does nothing else.

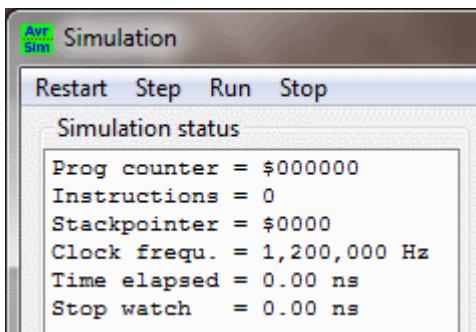
The source code is [here for download](#).

Here we see the so-called flow diagram of the source code. It starts with the reset of the controller and shows I/O operations as trapezoid. The number of clock cycles is added, and they show that the operation of the LED is symmetrical.

Every larger project should have such a flow-diagram, it allows us to systematically analyze the tasks to be performed and the decisions to be made during the different flow stages. We will later in this lecture see how helpful such diagrams can be to plan flows and to document those.

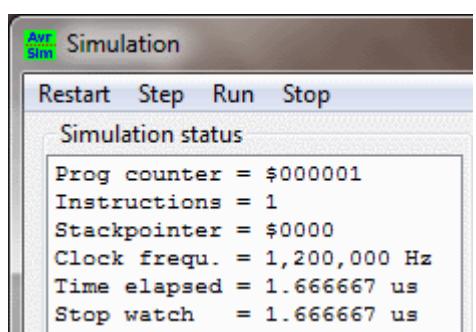
For the purists that learned not to use JUMPING in their academic studies: jumps are an absolutely necessary element in programming. If you are trained to avoid those you are just blind for the fact that you are using them over and over. But they do not look like jumps. In your language those look like "end" or "}", but they are factually jumps (back to something else). In assembler you have the freedom to use jumps whenever they are useful, so forget those useless academic theories: in assembler you and only you and not your compiler decides when to jump to somewhere in your code. If you decide to jump it is up to you to consider if jumping is nasty. The backside of freedom is responsibility.

In the simulator avr_sim (available at the [avr_sim website](#)) the program execution looks like that.



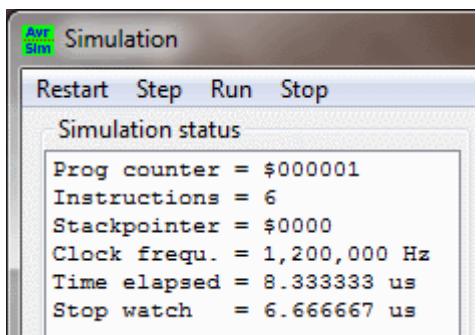
At the start the operating frequency is 1.2 MHz, all port-pins are inactive.

	7	6	5	4	3	2	1	0
(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	0
PINB	0	0	1	0	0	0	0	0
TINnB	0	0					0	
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0



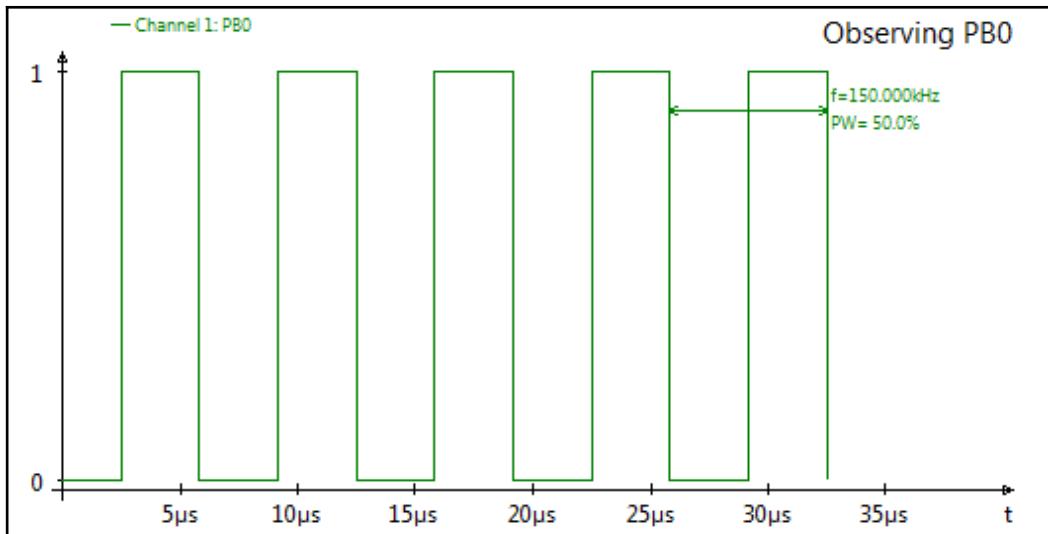
After the first step, executing the instruction SBI DDRB,DDR0, the time elapsed is 1.67 μ s (two clock cycles) and the respective direction bit is one now, driving the output pin low because the respective PORT register bit still is zero.

	7	6	5	4	3	2	1	0
(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	Lc
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0



Here the instructions of a complete cycle were executed. The stop watch which was cleared when entering the cycle, shows 6.67 μ s time, which corresponds to 150 kHz cycle frequency. The built-in scope in [avr_sim](#) demonstrates that (see below).

Unfortunately the blinking LED is going on and off with a frequency of $1,200,000 / 8 = 150,000$ cs/s. In the next section we try to reduce this high speed further.



3.2.2 Delayed fast blinking, 8-Bit

The two NOP instructions that we used to delay execution are not able to delay by more than 500 clock cycles, due to flash limitations. So, more effective solutions are necessary to delay further.

A delay would be to count a counter down until he reaches zero. The following sequence is such a typical delay loop:

```
.equ cCounter = 250 ; define the number of down-counts
    ldi R16, cCounter ; load a register with that constant
Loop:
    dec R16 ; decrease counter by one
    brne Loop ; branch if zero flag was not set in last instruction
```

Here a register (R16) is used to count down. Each AVR has 32 such registers, R0 to R31, each with 8 bits length. So they can store binary values between 0 and decimal 255. LDI means "load immediate". Only the upper half of the registers can be loaded using the LDI instruction.

The constant cCounter defines how often the loop is repeated. DEC decreases the content of the register by one. The Z (zero) flag in the status register of the controller is set if the register reaches zero after DEC, if not it is cleared. The status register is eight bit wide. If and which bit is changed in the status register during execution of an instruction is listed in the [Instruction Set Summary](#).

If the DEC instruction leads to zero in the above example, the execution of the loop ends, if not the loop is repeated. That is reached by the instruction BRNE. That means "Branch if not equal", a so-called conditional jump. Those conditional jumps move up to 63 instructions backward or 64 instructions forward. If this extension is exceeded a different jump mode has to be used. In our case above the branch is only one instruction back, so BRNE is valid.

The [Instruction Set Summary](#) yields the following clock cycles for the loop:

```
.equ cCounter = 250 ; (no clock cycle, assembler internal operation)
    ldi R16, cCounter ; 1 clock cycle
Loop: ; (no clock cycle, assembler internal operation)
    dec R16 ; 1 clock cycle
    brne Loop ; 2 cycles when branching, 1 cycle without branching
```

The [Instruction Set Summary](#) says that BRNE requires one or two clock cycles. As we know the pre-fetch mechanism requires two clock cycles if the jump back is executed. If no jump back occurs the number of clock cycles is one. The total loop execution requires the following clock cycles:

```
.equ cCounter = 250 ; (no code)
    ldi R16, cCounter ; 1 clock once executed
Loop:
    dec R16 ; 1 clock 250 times executed
    brne Loop ; 2 clock 249 times executed, 1 clock once executed
```

The complete execution takes $(1 + 250 + 2 * 249 + 1) = 750$ clock cycles. At a system clock frequency of 1.2 MHz 750 clock cycles are equivalent to 625 μ s, still too short for the human eye.

3.2.3 Delayed fast blinking, 16 Bit

Now try a 16 bit counter. This requires a 16 bit register, of which the AVR has four: the register pairs R25:R24, R27:R26, R29:R28 and R31:R30. Those can be accessed like single registers (e.g. with LDI instructions), but a few instructions work on the whole pair.

The 16 bit counting loop works like this:

```
.equ cCounter16 = 50000 ; 1 to 65535 (does not generate code)
    ldi R25,HIGH(cCounter16) ; 1 clock cycle, executed once
    ldi R24,LOW(cCounter16) ; 1 clock cycle, executed once
Loop16:
```

```
sbiw R24,1 ; count down 16 bit, 2 clock cycles, executed 50000 times
brne Loop16 ; 2 clock cycles when branching 49999 times, 1 clock cycle once
```

The two mathematical formulations "HIGH" and "LOW" split the 16 bit constant in their upper and lower eight bits and place them 16 bit wise to the register pair R25:R24. SBIW register,1 counts the register pair R25:R24 down by one in 16 bit mode. If the pair reaches zero, the Z flag is set. The instruction "BRNE" again branches conditionally as long as Z is clear.

Now the loop requires $(1 + 1 + 2*50000 + 2*49999 + 1) = 200.001$ clock cycles or 0.167 seconds. That is already rather near to a second, but does not match it.

Home	Top	Introduction	Hardware	Fast blinking	Second blinking
----------------------	---------------------	------------------------------	--------------------------	-------------------------------	---------------------------------

3.4 Exact second blinking

The second blinking requires a combined execution of an 8 bit and a 16 bit loop. The [source code is here for download](#).

```
;
; ****
; * Blinking a LED in one second with an ATtiny13 *
; * (C)2017 by http://www.avr-asm-tutorial.net *
; ****

;
.NOLIST ; Output to list file off
.INCLUDE "tn13def.inc" ; Port definitions
.LIST ; Output to list file on
;
; Define Registers
;
.def rCounterA = R16 ; Outer 8 bit counter
.def rCounterIL = R24 ; Inner 16 bit counter, LSB
.def rCounterIH = R25 ; Inner 16 bit counter, MSB
;
; Define constants
;
.equ cInner = 2458 ; Counter inner loop
.equ cOuter = 61 ; Counter outer loop
;
; Program start
;
    sbi DDRB,DDB0 ; Port pin PB0 as output
;
; Program loop
;
Loop:
    cbi PORTB,PORPB0 ; Port pin PB0 low, Led on, 2 clock cycles
    ; Outer delay loop, Led on
    ldi rCounterA,cOuter ; Outer 8 bit counter, 1 clock cycle
Loop1:
    ldi rCounterIH,HIGH(cInner) ; Inner 16 bit counter, 1 clock cycle
    ldi rCounterIL,LOW(cInner) ; 1 clock cycle
Loop1i:
    sbiw rCounterIL,1 ; Inner 16 bit counter downwards, 2 clock cycles
    brne Loop1i ; if not zero: jump to loop1i 2 cycles, if zero 1 cycle
    dec rCounterA ; Outer 8 bit counter downwards, 1 clock cycle
    brne Loop1 ; if not zero: jump to loop1 2 cycles, zero: 1 cycle
    nop ; Delay, 1 clock cycle
    nop ; Delay, 1 clock cycle
;
    sbi PORTB,PORPB0 ; Port pin PB0 high, Led off, 2 clock cycles
    ; Outer delay loop, Led off
    ldi rCounterA,cOuter ; Outer 8 bit counter, 1 clock cycle
Loop2:
    ldi rCounterIH,HIGH(cInner) ; Inner 16 bit counter, 1 clock cycle
    ldi rCounterIL,LOW(cInner) ; 1 clock cycle
Loop2i:
    sbiw rCounterIL,1 ; Inner 16 bit counter downwards, 2 clock cycles
    brne Loop2i ; if not zero: jump to Loop2i 2 cycles, if zero 1 cycle
    dec rCounterA ; Outer 8 bit counter downwards, 1 clock cycle
    brne Loop2 ; if not zero: jump to Loop2 2 cycles, if zero 1 cycle
    ; Cycle end
```

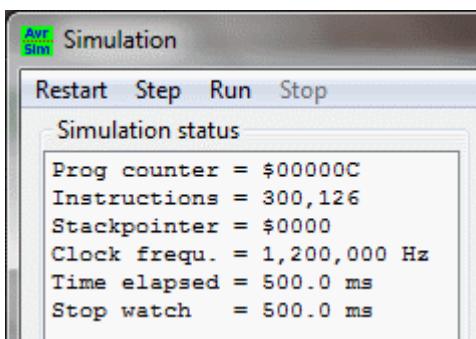
```

rjmp Loop ; start from the beginning, 2 clock cycles
;
; End of source code
;


```

The inner 16 bit and the outer 8 bit counting loops are available two times identically to yield a 0.5 s OFF period of the Led and a 0.5 s ON period.

The resulting formula for the number of cycles is relatively simple. To calculate the optimal constants cOuter and cInner is less simple. The two given constants in the source code are an optimal solution.



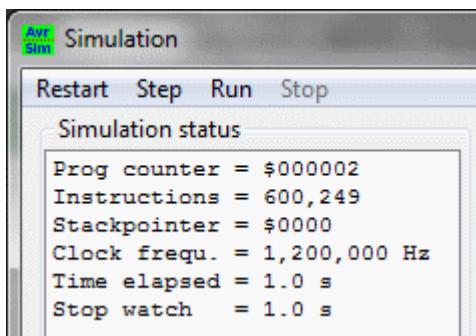
Simulating the source code with [avr_sim](#) yields the following results:

The first half of the loop execution lasts rather exactly 0.5 seconds. The portbit 0 in port B has been set to one, driving the output high and

switching the LED off. Note that more than 300,000 instructions have been executed in that half second.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	Hi
TINnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

Previous B Next



The same after the full second: execution time is exact and the portbit has been cleared, by that switching the LED on.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	Lo
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0

Previous B Next

That example demonstrates the clear advantage of assembler over any other programming language: exact planning and control of execution times via counting loops is a simple and straight-forward task. No uncertainties exist, e.g. on what a compiler thinks how long a second would be and what instructions he inserts to resolve the given delay task. Exact planning instead of guessing around what the compiler does.

On the following page is the flow diagram and the cycle calculation.

Instructions	Clock cycles	Number of executions per loop cycle	Total clock cycles
SBI DDRB,DDB0	2	(outside the loop)	
CBI PORTB,PORTB0	2	1	2
LDI rCountA,cOuter	1	1	1
LDI rCountIH,HIGH(cInner) LDI rCountIL,LOW(cInner)	1 1	cOuter cOuter	2*cOuter
SBIW rCountIL,1	2	cOuter * cInner	2*cOuter*cInner
BRNE Loop1i	2 (Jump) 1 (No jump)	cOuter * (cInner - 1) cOuter	2*cOuter*cInner - cOuter
DEC rCountA	1	cOuter	cOuter
BRNE Loop1	2 (Jump) 1 (No jump)	cOuter - 1 1	2*cOuter - 1
NOP NOP	1 1	1 1	2
SBI PORTB,PORTB0	2	1	2
LDI rCountA,cOuter	1	1	1
LDI rCountIH,HIGH(cInner) LDI rCountIL,LOW(cInner)	1 1	cOuter cOuter	2*cOuter
SBIW rCountIL,1	2	cOuter * cInner	2*cOuter*cInner
BRNE Loop2i	2 (Jump) 1 (No jump)	cOuter * (cInner - 1) cOuter	2*cOuter*cInner - cOuter
DEC rCountA	1	cOuter	cOuter
BRNE Loop2	2 (Jump) 1 (No jump)	cOuter - 1 1	2*cOuter - 1
RJMP Loop	2	1	2

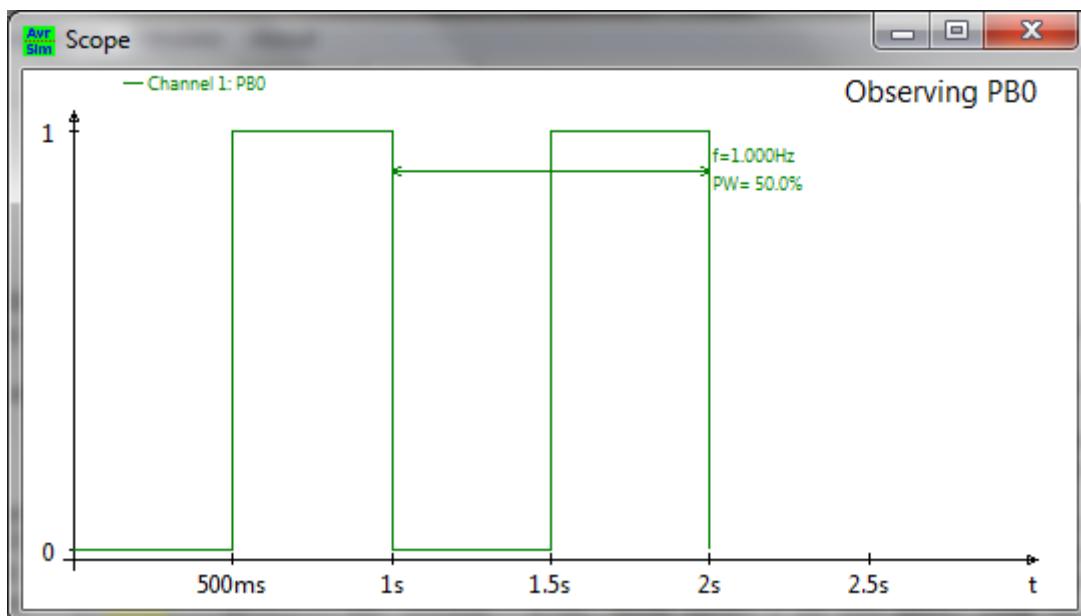
$$\text{Total cycles} = 8 * (\text{cOuter} * \text{cInner} + \text{cOuter} + 1)$$

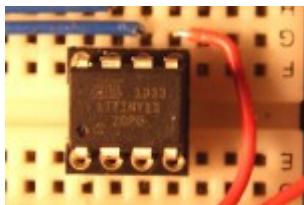
A few things can be learned from that example:

- Performing exact timing loops requires a lot of program planning in advance. Before the first source code can be entered a lot of brain activity is necessary to reach the desired goal.

- Timing loops are not that simple tasks as it seems at first glance. One will run into complicated algebra math very fast, if the task is a little more than just delaying for a few microseconds.
- The task is rather simple as the controller has nothing else to do than counting. It could well get much more complicated if he would have to check for keys pressed (and related actions) or any other tasks. Better avoid that method if you want to stay mentally healthy.
- In later lectures we will learn methods that fit better to this kind of tasks.

Finally, the second blinker on PB0 in [avr_sim's](#) scope.





Lecture 4: Blinking a LED with a timer

In this lecture we will wave Goodbye to lengthy and boring counting loops. We let the internal timer fulfill the task of counting, independent from the execution of the program.

4.0 Overview

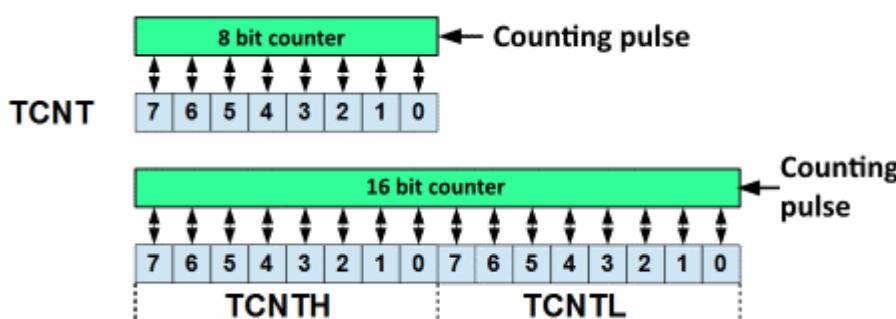
1. [Introduction to timer hardware](#)
2. [Hardware, components, mounting](#)
3. [Timer in standard counting mode](#)
4. [Timer in CTC mode](#)
5. [Timer in 128 kcs/s mode](#)

4.1 Introduction to timer hardware

The build-in timer (exact: timer/counter, TC0) is the most often used internal hardware component. As a versatile element, it has several different modes, depending from our needs. We will use the timer in later lectures, so in the next lectures we will use this timer in different modes to control the LED.

Different AVR devices have a different number of different counters. Those are named TC0, TC1, etc. As timers sometimes can be connected to port pins those port pins names refer to the TC number. The ATTiny13 has only one timer, but the zero in the name is inserted even though unnecessary in that case.

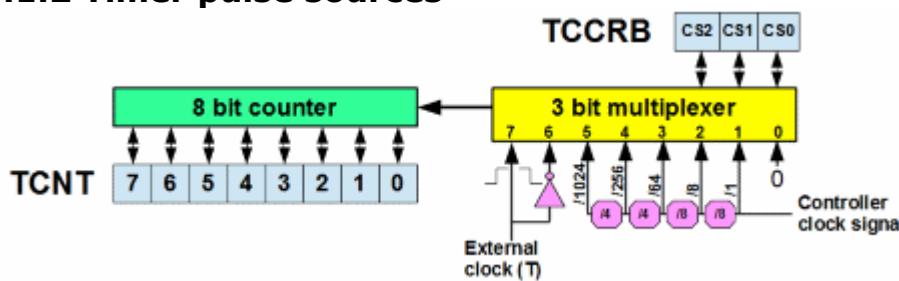
4.1.1 Timer



Timers in the AVR are either 8 or 16 bit wide. They count up or, in certain modes, downwards. With 8 bits they count from decimal 0 to 255, with 16 bits from 0 to 65,535. If they reach and exceed their upper limit, they restart again at zero. Their actual

count state is available by reading the port TCNT (8 bits) or the ports TCNTH and TCNTH (16 bits). Those can be written, too, and the timer counts from this changed state.

4.1.2 Timer pulse sources



There are seven different sources to connect the timer with pulses (and an eight's to switch timing/counting off). Three bits in the timer control port B, TCCRB, control which source is used. Those three bits control a multi-

plexer with eight inputs and control the signal that clocks the timer.

The eight different sources are:

#	Bin	Mode	Clock source
0	000	Off	No signal, counting off
1	001	Timer	Controller clock
2	010	Timer	Controller clock divided by 8
3	011	Timer	Controller clock divided by 64
4	100	Timer	Controller clock divided by 512
5	101	Timer	Controller clock divided by 1,024
6	110	Counter	external pin T, falling edge
7	111	Counter	external pin T, rising edge

With the lines

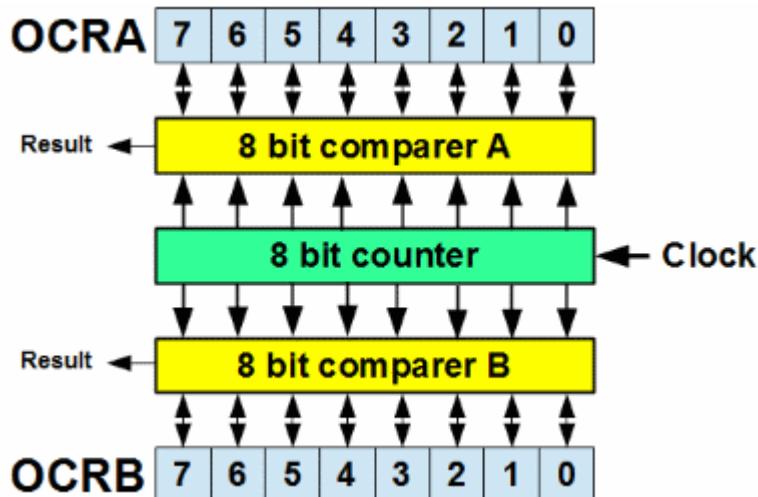
```
ldi R16,1 ; R16 to 1
out TCCR0B,R16 ; to control port B
```

the timer starts with the controller clock as source, with

```
ldi R16,0 ; R16 to zero
out TCCR0B,R16 ; to control port B
```

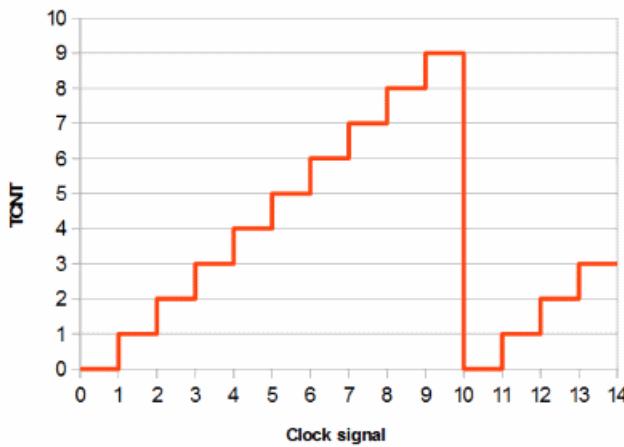
the timer is stopped. The instruction "OUT" writes all eight bits to the port.

4.1.3 Timer and compare match



The whole timing and counting would only have limited sense if we were to read TCNT in a loop and react to certain values. For this task the timer has two comparers. Those compare the current state of TCNT with programmed values in OCRA and OCRB. If the value of TCNT fits with one of those certain hardware functions can be programmed or the controller can be interrupted (see later lectures for this).

4.1.4 Timer in CTC mode



The obvious application of a comparer is to use the compare match to restart the timer/counter. This mode of counting is called "clear timer on compare" or CTC. With that the timer/counter resolution can be set to any desired length (below eight/sixteen bit, of course).

Please note that the compare match resets the counter a) if a match occurs and b) the next counting pulse occurs. This ensures that the duration is exact and does not depend from the speed of the compare match recognition. In the example the compare register is set to nine, by which the CTC resets with a count of ten (pulse 0 to 9 = 10 pulses).

CTC is possible with all timers using the compare match A. Compare match B can then be used for other purposes. Some AVR devices have an additional port that can be used as compare match for CTC, leaving compare match A and B for other purposes.

4.1.5 Timer manipulation of pin outputs

The second frequent use of compare match ports is to set or clear port pins. With that the timer/counter can generate electrical pulses on a port pin of any desired duration and the compare match gets visible on the outside. For the ATTiny13 the port pin PB0 can be programmed to follow compare match A (the port pin's name if programmed in that way: OC0A), PB1 connects to compare match B (OC0B). To enable compare match A PB0 has to be configured as output (e.g. with sbi DDRB,DDB0) and to select the pin mode with two control bits:

COM0A1 COM0B1	COM0AO COM0BO	Function OC0A/OC0B
0	0	No change on port pin
0	1	Port pin changes its polarity at Compare Match (Toggle)
1	0	Port pin clears at Compare Match
1	1	Port pin sets at Compare Match

Please note that this applies for the normal timing/counting mode. In other modes the control bits have different meanings.

4.2 Hardware, components and mounting

In this lecture the same hardware is used as in the previous lectures.

4.3 Timer with standard operation

4.3.1 Functioning

The simplest mode to program a blinker is to let the timer count to 255 with the maximum prescaler available (1,024), to compare with 255 and to toggle the output pin OC0A. With that, a blinking frequency of

$$1,200,000 / 1,024 / 256 / 2 = 2.29 \text{ cs/s}$$

results. That is faster than it should be, but already in a visible range. Not really able to base a watch on, but simple.

4.3.2 Program

Here the whole program ([Source code is here](#)). It first switches PB0 to be output. Then writes 255 to the Compare A port, defines the output pin to the toggle mode and starts the timer with a prescaler value of 1,024.

```
; ****
; * Timer to blink a LED *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ----- Registers -----
.def rmp = R16 ; Multi purpose register
;
; ----- Timing -----
; Internal RC-Oscillator = 9,600,000 Hz
; Clock prescaler CLKPR = 8
; Internal contr. clock = 1,200,000 Hz
; TCO prescaler = 1,024
; TCO tick = 1,171.875 cs/s
; TCO cycle = 256
; TCO single cycle = 4.578 cs/s
; TCO toggle frequency = 2.289 cs/s
;
; ----- Start -----
; PB0 as output
sbi DDRB,DDB0 ; PB0 as output
; Select Compare Match
ldi rmp,0xFF ; Match at 255
out OCR0A,rmp ; to Compare Match A
; toggle PB0 at Compare Match
ldi rmp,1<<COM0A0 ; Toggle Mode
out TCCR0A,rmp ; to control port A
; Start timer
ldi rmp,(1<<CS02)|(1<<CS00) ; prescaler 1024
out TCCR0B,rmp ; to control port B
Loop:
rjmp Loop
```

The instruction OUT writes the eight bits in the register to the given port.

Following the initiation sequence the controller is not needed any more for control. He is sent to an unlimited loop (Remember: the controller cannot do nothing). Note that this is a much more elegant solution than counting loops, which you can see by the fact that it is much shorter. And relieves the controller from control, making it simple to add additional tasks without interfering the timing.

One special feature has to be learned here. We need it from now in each source code, so we should learn this. With the formulation "ldi rmp,1<<COM0A0" the port bit COM0A0 in the port TCCR0A needs to be set to one.

Bit	7	6	5	4	3	2	1	0	
	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00	TCCR0A
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	

COM0A0 is bit 6 in this port. We could formulate this for e.g. as "ldi rmp,0b01000000" (0b is added to signal that the following is a binary number). An hour later, at the latest, we have forgotten what this binary 01000000 stands for, and we will have to consult the port list in the device data-book again. Therefore the formulation "ldi rmp,1<<COM0A0" is much more convenient and transparent. The formulation means:

- Take a binary 1, that is 0b00000001,
- shift this by six times left (COM0A0 is defined as six in the def.inc), << means "shift left", by inserting a zero to the right,
- this shifting finally leads to 0b01000000.

It is essential to understand that the shifting is done during assembling only, not within the controller. The shift instruction of the controller is "LSL" (Logical Shift Left) and has a register as parameter, to be shifted left once. So do not confuse internal assembler calculations with what the controller actually does.

Bit	7	6	5	4	3	2	1	0	
	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Similarly, in the source line "ldi rmp,(1<<CS02)|(1<<CS00)" those shift-left operations are used. In that case the two results of the two shift operations are ORED ("|") and so two bits within the register are set simultaneously. CS02 is 2 (in the def.inc file), while CS00 is zero. So 0b00000100 (1<<CS02) and 0b00000001 (1<<CS00) are bit-wise ORED to yield 0b00000101. This value is finally written to the register rmp. The assembler-internal OR must not be confused with the instruction OR, that ORs two registers and writes the result to the first one. The | is solely done during the assembly process.

The characters << and | are mathematical operators of the assembler, of which there are further ones such as e.g. +, -, *, and /. Those are all commands to the assembler, of which the controller is unaware.

4.3.3 Simulating timer operation in this mode

The following simulates the timer operation with [avr_sim](#).

After initing the timer TC0 is in that condition:

- Timer mode is "normal counting".
- PB0 toggles when the timer compare match A occurs.
- Compare A and TOP count is set to 255.
- The prescaler divides the controller clock by 1,024.

After init the port B direction bit of port-bit 0 is one, that means that the pin follows the state of PORTB0, drives the output low and switches the LED on.

Now the timer is counting. Each clock cycle increases the prescaler value. If the prescaler reaches 1,024, the portregister TCNT is increased and the prescaler restarts.

After $256 * 1,024 = 262,144$ clock cycles the timer reaches the compare match value in the

CompareA port-register. The compare match occurred already 1,024 clock cycles before but the match is recognized by the timer only after the next TCNT increase, when the timer restarts at zero. That is why the divider is (CompareMatchA + 1) or 256.

The compare match execution has toggled PORTB0 now, the output pin is high and the LED is switched off. This is automatically executed by the timer, does not need any instructions or oversight by the controller.

The simulator has counted 131,079 instructions executed (of which 99.9% are RJMP Loop) and 218.46 ms have elapsed since startup. The stopwatch, which was cleared after init, shows 218.452 ms execution time, which corresponds to our calculated 4.58 cycles per second and a blink frequency of 2.29 Hz.

4.3.4 Disadvantage of that solution

Even though an elegant solution, it does not blink in a 1 per second rhythm. Dividing the controllers internal clock of 1.2 Mcs/s by 1.024 in the prescaler, by 256 in the timer and by 2 in the toggle stage of OC0A, a total of 524,288 clock cycles, does not yield one second. The controller clock would have to be reduced by approximately two to come nearer to the frequency of 1 Hz. We will see in the following examples how this can be better achieved.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Standard mode](#) [CTC mode](#) [128kcs mode](#)

4.4 Clock prescaler

4.4.1 Functioning

4.4.1.1 To reduce the controller clock speed

All tiny and mega devices of the AVRs have a prescaler with which the controller clock can be adjusted. As divider factors multiples of two can be selected, before the clock signal is fed into the fetch-and-execute section of the controller. This pre-scaling is done no matter which clock source is selected by setting the fuses of the controller.

Table 6-8. Clock Prescaler Select

CLKPS3	CLKPS2	CLKPS1	CLKPS0	Clock Division Factor
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128
1	0	0	0	256

In the default case, the internal RC oscillator is selected as clock source. By use of a prescaler value of 8 this leads to the controller's default clock of 1.2 Mcs/s. The prescaler value of 8 is set by the fuse CKDIV8, which is by default set. If we clear that fuse, the controller runs at 9.6 Mcs/s clock speed. See [the fuse setting in course 1](#). Please be aware that at very low operating voltages the 9.6 Mcs/s speed does not work in the ATtiny13V type! To set that fuse again, apply a higher operating voltage of e.g. 5 V temporarily.

Bit	7	6	5	4	3	2	1	0	CLKPR
Read/Write	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0	
Initial Value	R/W	R	R	R	R/W	R/W	R/W	R/W	
See Bit Description									

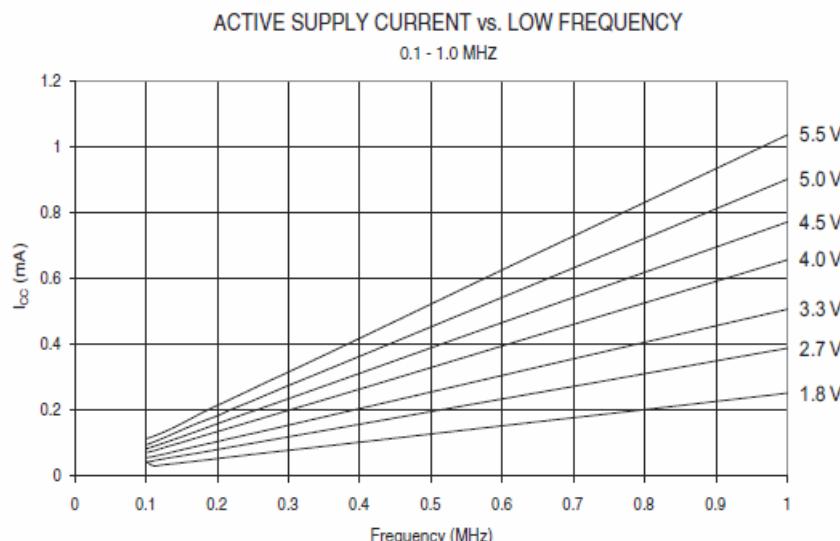
Even with the fuse CKDIV8 set, we can override this setting by programming the CLKPS bits in the port CLKPR. So we can select 4.8 or 2.4 Mcs/s as clock frequency, but even down to $9.6 \text{ Mcs/s} / 256 = 37.5 \text{ kcs/s}$ are possible. The unintended programming of the port CLKPR is protected. First we have to set the enable bit CLKPCE in CLKPR, with all other bits cleared. Immediately after that we can write CLKPR again with the desired CLKPS settings (with CLKPCE cleared).

To set a prescaler value of 32 the following code has to be executed:

```
ldi R16,1<<CLKPCE ; Program Enable Bit to one
out CLKPR,R16 ; write to port CLKPR
ldi R16,(1<<CLKPS2)|(1<<CLKPS0) ; Prescaler = 32
out CLKPR,R16 ; write to port CLKPR
```

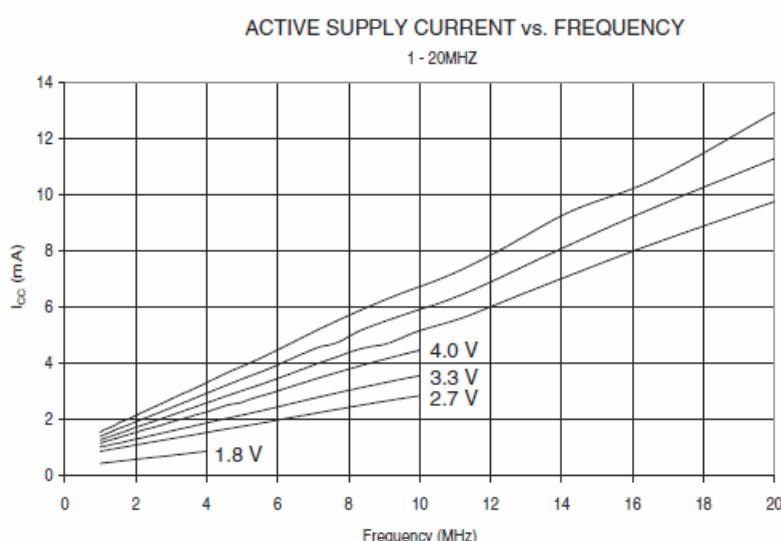
Now the controller runs at a speed of $9.6 \text{ Mcs/s} / 32 = 300 \text{ kcs/s}$ clock frequency. But be aware that the reduction of that speed requires a smaller ISP frequency when accessing the controller via the programming interface (e.g. 75 kcs/s), otherwise error messages occur.

Figure 19-1. Active Supply Current vs. Frequency (0.1 - 1.0 MHz)



By reducing the clock the controller is a lame duck. One does that if you want to reduce current requirements. The operating current is approximately linear with the clock frequency, as the diagram from the data-book shows.

Figure 19-2. Active Supply Current vs. Frequency (1 - 20 MHz)

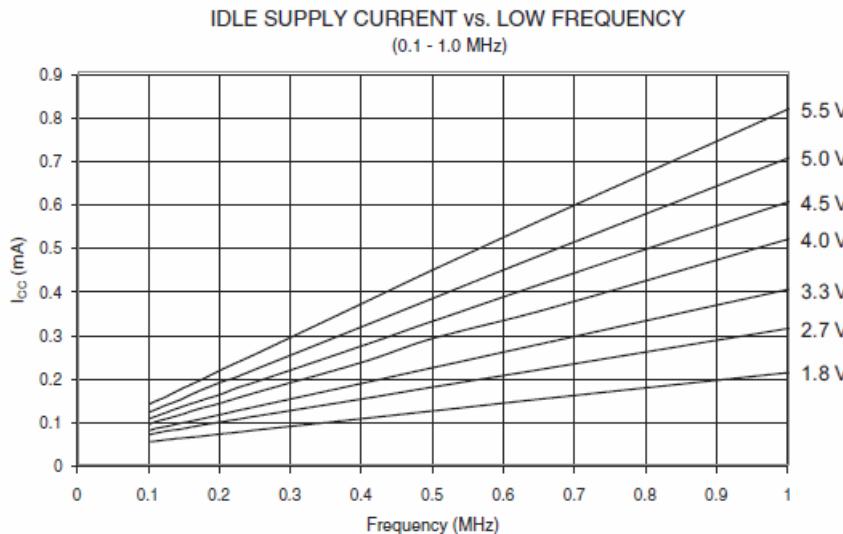


This linearity is even more exact at small clock frequencies. At 300 kcs/s the operating current is reduced to mere 0.3 mA at 4.8 V. The battery or rechargeable battery that you use will be happy and operate very much longer (if you do not drive 10 additional LEDs).

4.4.1.2 Controller sleeping

A further method to reduce current requirements is to not infinitely loop but to send the controller to sleep. In that mode further reading of code from the flash memory, decoding and execution of instructions is suspended completely. This sleep mode is called "idle".

Figure 19-7. Idle Supply Current vs. Frequency (0.1 - 1.0 MHz)



This idle state can be switched on by setting the Sleep-Enable-Bit in the port MCUCR (Microcontroller universal control register) and by executing the SLEEP instruction. Like this:

```
; Set sleep enable to one
ldi R16,1<<SE
; to univers.control port
out MCUCR,R16
; go to sleep
sleep
```

In our case the controller sleeps and no wake-up call other than the RESET will wake him up. Other wake-up mechanisms are shown in later lectures.

4.4.1.3 Timer in CTC mode

The comparer A of the timer/counter can be used for an additional operating mode, the CTC (Clear Timer on Compare match) mode. In our case we can use this to divide the clock frequency by factors that are not multiples of two. With that we come closer to a blinking in a second.

Table 11-8. Waveform Generation Mode Bit Description

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on ⁽¹⁾⁽²⁾
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	TOP	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	TOP	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

This table shows all modes of the timer/counter in an ATtiny13. The CTC mode can be selected by setting the WGM1 bit. The maximum value that the timer has (TOP) is the number that is stored in port OCRA. After that the timer resets to zero with the

next timer clock signal.

Bit	7	6	5	4	3	2	1	0	
ReadWrite	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00	TCCR0A
Initial Value	0	0	0	0	0	0	0	0	
ReadWrite	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Initial Value	0	0	0	0	0	0	0	0	

The three WGM bits WGM02, WGM01 and WGM00 of timer 0 are distributed over the two ports TCCR0A and TCCR0B. As we only need to

set WGM01 to one and COM0A0 to toggle the output pin, we again use $(1 \ll \text{WGM01}) | (1 \ll \text{COM0A0})$.

4.4.2 Program

The mighty program is listed here, the [source code is here](#). First the clock frequency is changed, then port bit PB0 is switched to output, then the timer is configured and finally the controller is send to sleep. In the case he wakes up (which he will not), additional code is added to send him back to sleep.

```

;
; ****
; * LED blinker with 300 kcs/s and CTC *
; * (C)2017 by www.avr-asm-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.NOLIST
;
; ----- Register -----
.def rmp = R16 ; multi purpose register
;
; ----- Timing -----
; Internal RC oscillator = 9.600.000 cs/s
; Prescaler CLKPR      = 32
; Clock controller     = 300.000 cs/s
; Prescaler TCO        = 1.024
; Timer tick TCO       = 292.97 cs/s
; Timer CTC value     = 146
; Clock freq. CTC-OC0A = 2.006 cs/s
; Frequency OC0A toggle = 1.003 cs/s
;
; ----- Constants -----
.equ fRC = 9600000 ; Controller RC oscillator
.equ pClk = 32 ; Clock prescaler
.equ pTC0 = 1024 ; TCO prescaler
.equ pCtc = fRC / pClk / pTC0 / 2 ; CTC divider
.equ cCtc = pCtc - 1 ; CTC value
;
; ----- Program start -----
Start:
    ; To reduce the clock frequency
    ldi rmp,1<<CLKPCE ; CLKPR write enable
    out CLKPR,rmp ; to CLKPR port
    ldi rmp,(1<<CLKPS2) | (1<<CLKPS0) ; Prescaler = 32
    out CLKPR,rmp ; to CLKPR port
    ; PB0 as output
    sbi DDRB,DDB0 ; Data direction output
    ; Programming the timer and timer start
    ldi rmp,cCtc ; The CTC compare match value ...
    out OCR0A,rmp ; ... to the Compare A port
    ldi rmp,(1<<COM0A0) | (1<<WGM01) ; Toggle OC0A, CTC mode
    out TCCR0A,rmp ; to timer control port A
    ldi rmp,(1<<CS02) | (1<<CS00) ; Prescaler 1024
    out TCCR0B,rmp ; to timer control port B
    ; Sleep mode
    ldi rmp,1<<SE ; Sleep mode idle

```

```

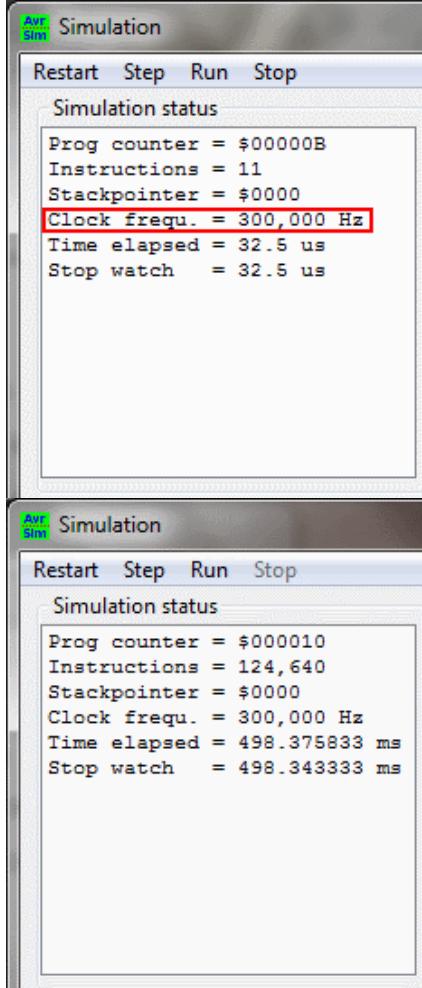
        out MCUCR,rmp ; to universal control port
Wakeup:
    sleep ; send controller to sleep
    nop ; do nothing after wakeup
    rjmp Wakeup ; send controller to sleep again
;
; End of source code
;

```

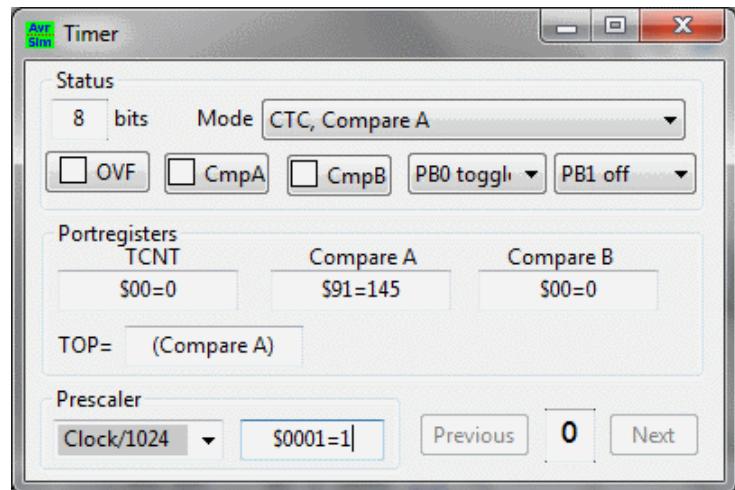
Do not forget: the controller now works at 300 kcs/s clock, the ISP frequency has to be reduced to access the controller via ISP.

4.4.3 Simulating timer operation in CTC mode

Again, we simulate the timer operation in CTC mode with [avr_sim](#).



That is the situation after initiation. The controller's clock has been lowered down to 300 kHz by changing the CLKPR port-register. The timer is in CTC mode, with Compare match port-register A as TOP value, and pin output PB0 toggles on compare match.



This is the state when the first compare match occurred: 498 ms have elapsed and the port-bit PB0 is set, switching the LED off.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	Hi
TINnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

The second compare match occurs after 996.69 ms. This is very close to one second.

4.4.4 Advantages and drawbacks

The current requirements of the controller is strongly reduced, by reduced clock speed and by idle sleep mode. To further reduce the current we can reduce the operating voltage down to e.g. 3.3 V.

A drawback is that we missed the exact second. The deviation is much smaller than the inaccuracy of the internal RC generator from its nominal frequency (+/-10%), but for a watch we would need a better match (and a re-adjustment of the internal RC oscillator) or an external crystal oscillator.

4.5 Timer in 128kcs/s mode

The ATtiny13 has an additional internal RC oscillator that can be selected as clock source by setting fuses. This oscillator has a frequency of 128 kcs/s. This can be divided by the use of the CLKPR port down to 500 cs/s. Do that only in case you own a programmer that can run at less than 125 cs/s ISP frequency. The AVR-ISP-mkII can go down to 50.1 cs/s, so would be suitable for that purpose.

4.5.1 Functioning

With a clock frequency of 128 kcs/s the clock has to be divided by 64,000 to reach a toggle frequency of 2 cs/s. With a TCO prescaler value of 1,024 the frequency would be 62.5 cs/s, an odd number. With 256 as prescaler value the CTC has to be adjusted to 250 pulses to reach 2 cs/s, which is manageable with the 8 bit timer TCO.

4.5.2 Program

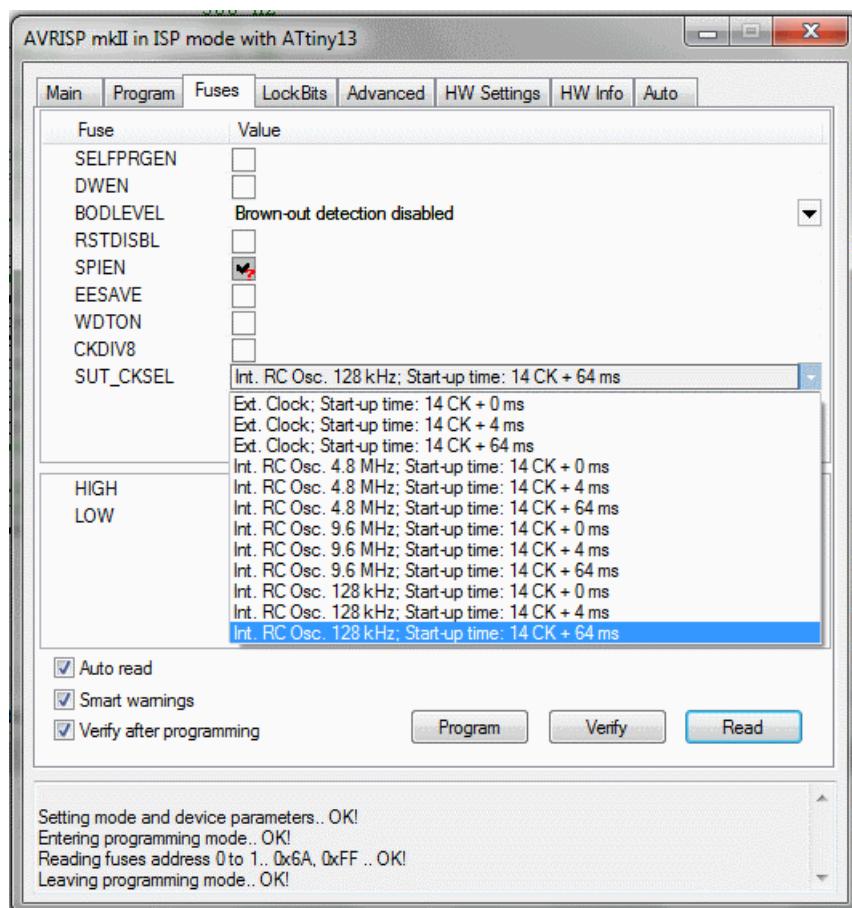
The program is similar to the previous, only the timer prescaler and the CTC value are different. The [source code is here](#).

Before we transfer the hexcode we should set the CKDIV8 fuse.

```
; ****
; * Blink LED with timer at 128 kcs clock *
; * (C)2017 by www.avr-asm-tutorial.net   *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ----- Register -----
.def rmp = R16 ; multi purpose register
;
; ----- Timing -----
; Internal RC oscillator      = 128.000 cs/s
; Clock prescler CLKPR       =      1
; Internal clock frequency    = 128.000 cs/s
; TCO prescaler                = 256
; TCO timer tick              = 500 cs/s
; TCO CTC divider             = 250
; TCO toggle frequency        =      1 cs/s
;
; ----- Constants -----
.equ cCtc = 250-1
;
; ----- Programm -----
; PBO as output
sbi DDRB,DDB0 ; Port bit as output
; Timer Compare Match A to 250
ldi rmp,cCtc ; Match A value
out OCR0A,rmp ; to Match port A
; Timer as CTC and toggle output A
ldi rmp,(1<<COM0A0)|(1<<WGM01) ; Toggle and CTC
out TCCR0A,rmp ; to timer control port A
ldi rmp,1<<CS02 ; Prescaler to 256
out TCCR0B,rmp ; to timer control port B
; Sleep mode
ldi rmp,1<<SE ; Sleep mode idle
out MCUCR,rmp ; to universal control port
; Sleep loop
;
.Wakeups:
sleep ; send to sleep
nop ; after wakeup
rjmp Wakeups
```

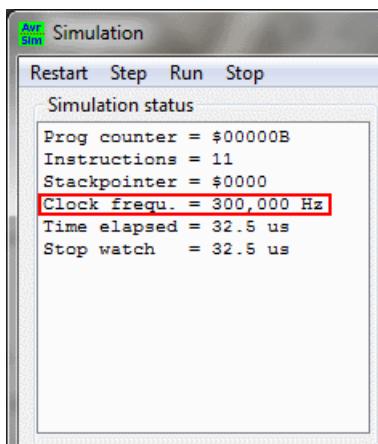
```
; End of source code
```

If the hex-code is transferred to the tiny the blinking of the LED is hectic, because the clock frequency is still 1.2 Mcs/s. Now we have to change clock to 128 kcs/s. To do that we select this oscillator by setting the fuses and clear the CKDIV8 fuse. After transferring the fuses the 1-second-blinking of the LED should occur.

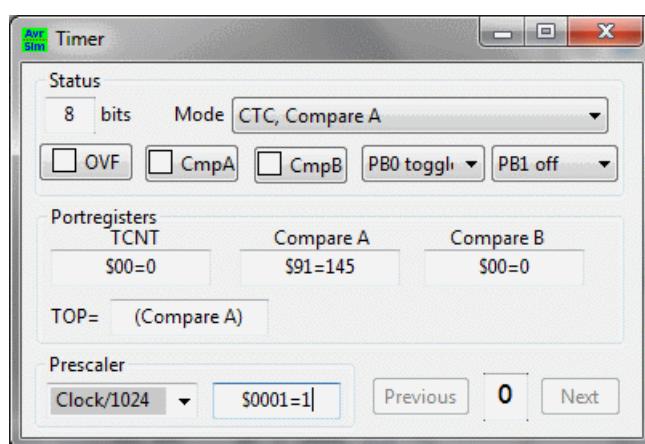


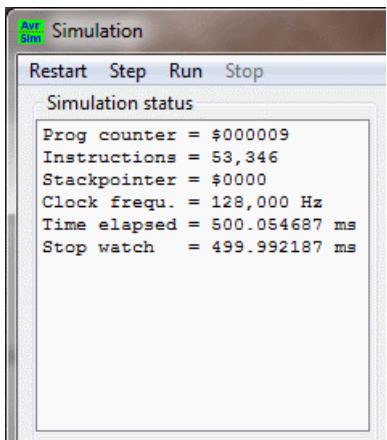
4.5.3 Simulation of the timer operation at 128 kHz

Simulation with [avr_sim](#). yields the following.

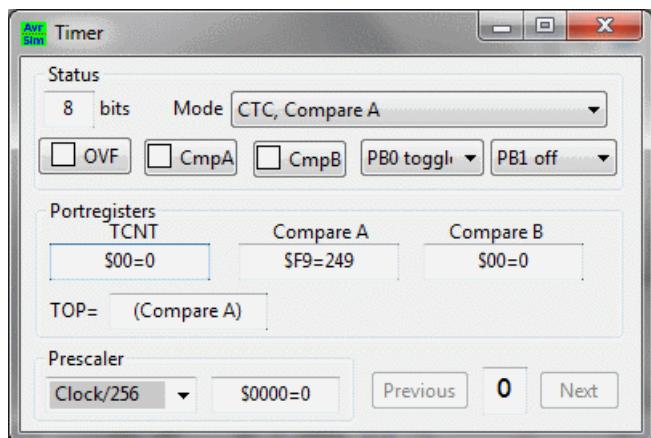


After init at the clock frequency of 128 kHz 62.5 μ s have elapsed, the timer is in CTC mode with Compare Match A port register as TOP value.



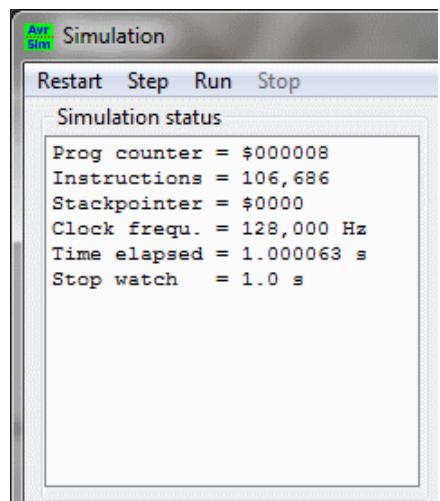
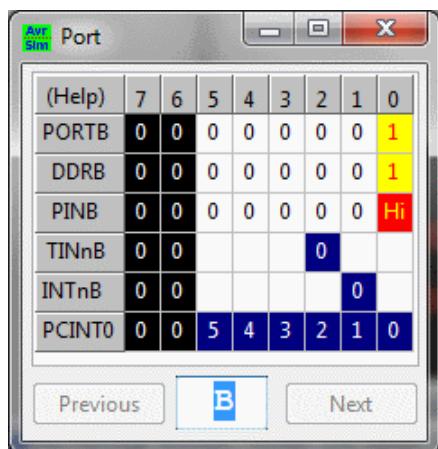


After the first compare match has occurred the simulation looks like this. Times are rather accurate here.



[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Standard mode](#) [CTC mode](#) [128kcs mode](#)

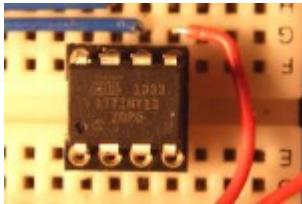
Anything correct also at the second compare match.



4.5.4 Advantages and drawbacks

Now the divider has exactly 1.000000 cs/s. On the accuracy of the internal 128 kcs/s oscillator not much is published by ATTEL. The voltage and temperature dependency of the frequency lets expect an accuracy of -10%. At least the whole controller now is at current requirements of less than one fiftieth of the LED thanks to the sleep mode from which the controller never wakes up.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Standard mode](#) [CTC mode](#) [128kcs mode](#)



Lecture 5: To control the intensity of a LED via PWM

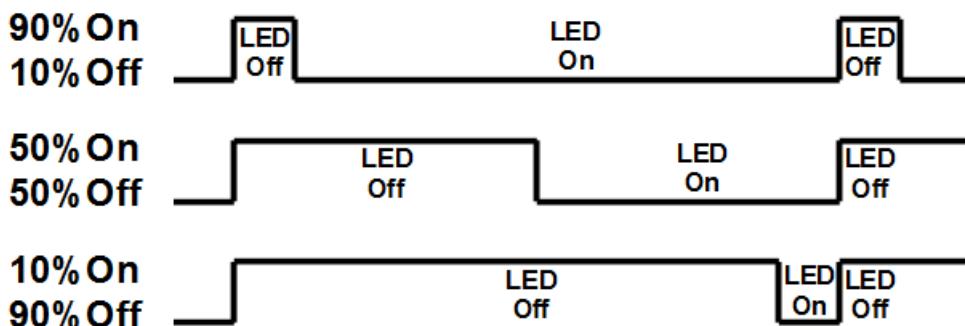
So far we used blinking of a LED. Now we continue blinking, but with a high frequency. With that blinking we change the intensity from weak to high. And this in an exact linear manner and not with a non-linear current.

5.0 Overview

1. [Introduction to the PWM mode of the timer](#)
2. [Hardware, components, mounting](#)
3. [Timer in fast PWM mode](#)
4. [Timer in phase-correct PWM mode](#)

5.1 Introduction to the PWM mode of the timer

5.1.1 8 bit PWM



PWM means pulse width modulation. To do this the timer, on restarting, sets or clears an output (OC0A and/or OC0B). If a compare match occurs (A or B) the polarity of the output changes (clear, set). By reaching the largest value the polarity of the output again changes and the cycle starts again.

The later the compare match occurs within the whole cycle the longer gets the first phase and the shorter gets the second phase. This behaviour allows to control the power of a DC motor as well as the intensity of LEDs. Please note that during the ON period the transistor drives the motor with maximum available power and that only the duration is limited. So the effective motor power is really linear.

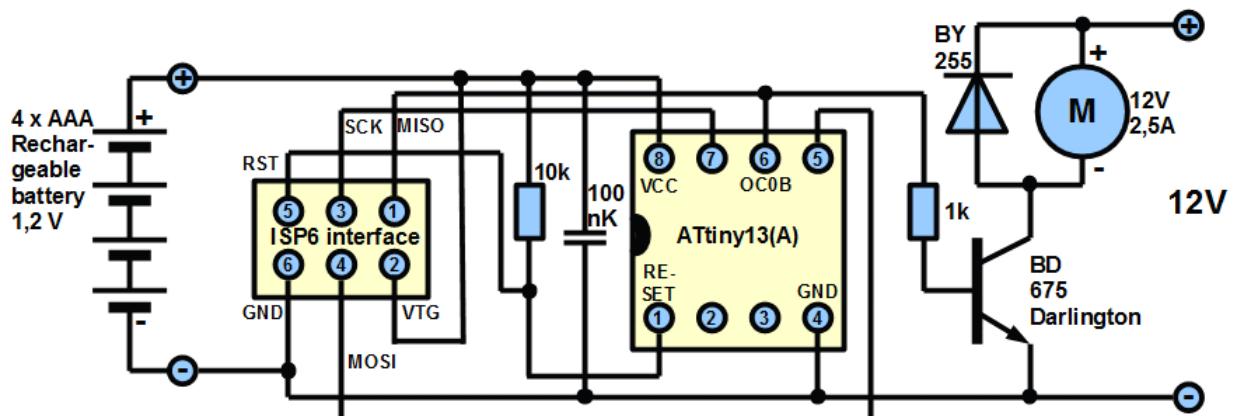


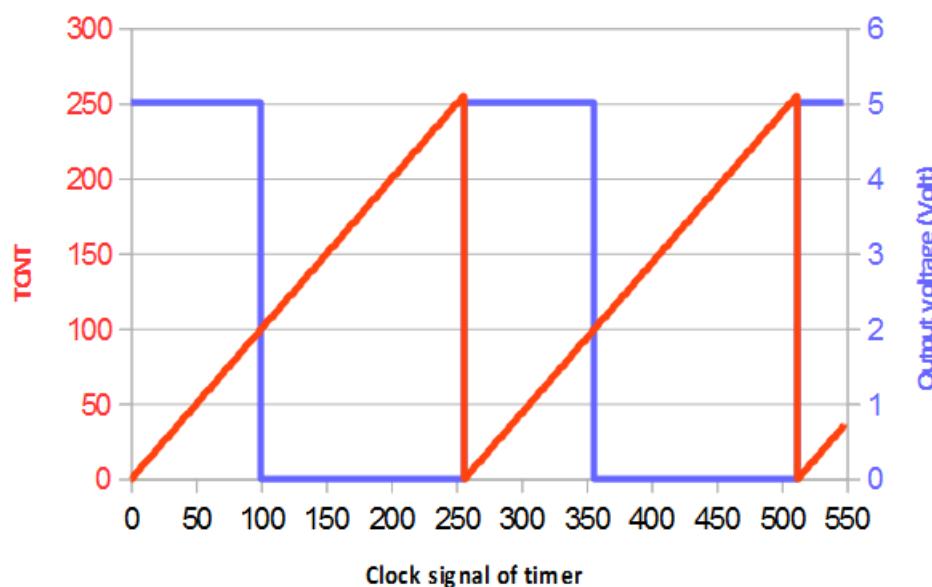
Table 11-3. Compare Output Mode, Fast PWM Mode⁽¹⁾

COM01	COM00	Description
0	0	Normal port operation, OC0A disconnected.
0	1	WGM02 = 0: Normal Port Operation, OC0A Disconnected. WGM02 = 1: Toggle OC0A on Compare Match.
1	0	Clear OC0A on Compare Match, set OC0A at TOP
1	1	Set OC0A on Compare Match, clear OC0A at TOP

Note: 1. A special case occurs when OCR0A equals TOP and COM0A1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Fast PWM Mode" on page 64 for more details.

cleared. Similar to this COM0B1 and COM0B0 in the same port control the output pin OC0B.

Fast PWM, TOP=255, COM=0b10, Compare match = 100



This shows how the voltage on the output OC0A changes in Fast PWM mode with TOP at 255 and a compare match at 100. The resulting pulse width is $100 / 256 = 39\%$ high.

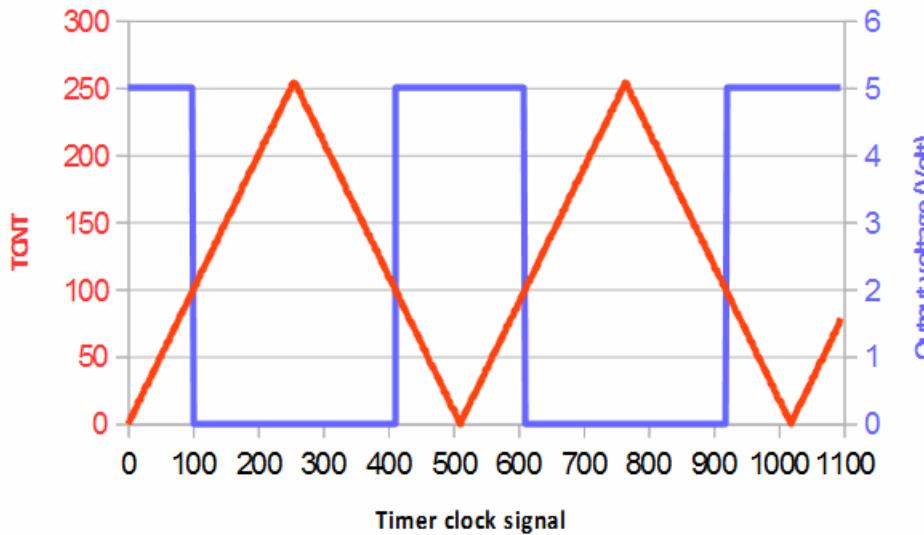
Within the Fast PWM mode the selection of the timer prescaler determines the PWM's frequency. At the different clock frequencies and prescaler values the following frequencies result and offer a wide range.

At 1.2 and 2.4 Mcs/s all PWM frequencies are in an audible range (with motors this can result in uncomfortable humming). A clock frequency of 128 kcs/s is inappropriate for that purpose, at a prescaler value of 1,024 we can visually follow the PWM process.

Clock	P=1	P=8	P=64	P=256	P=1,024
9.6 Mcs/s	37.5 kcs/s	4.69 kcs/s	586 cs/s	146.5 cs/s	36.6 cs/s
4.8 Mcs/s	18.8 kcs/s	2.35 kcs/s	293 cs/s	73.3 cs/s	18.3 cs/s
2.4 Mcs/s	9.4 kcs/s	1.68 kcs/s	147 cs/s	36.7 cs/s	9.15 cs/s
1.2 Mcs/s	4.69 kcs/s	586 cs/s	73.2 cs/s	18.3 cs/s	4.6 cs/s
128 kcs/s	500 cs/s	62.5 cs/s	7.8 cs/s	1.95 cs/s	0.49 cs/s

5.1.2 Phase correct 8 bit PWM

**Phase correct PWM, OCR=100, TOP=255, COM=10
compare match = 100**



This is the signal process in the phase-correct mode. The timer counts up- and downwards, each time the compare match is reached the polarity of the output changes. The PWM frequency is half that in Fast PWM mode.

Phase correct PWM mode is used if the compare value changes very often and with large changes. The change of the compare match value is buffered, the changed value comes only into effect when the TOP value has been reached. So there

are always two periods (up-counting, down-counting) performed for each compare value. With this, the jitter is strongly reduced.

5.1.3 PWM with different resolution

It is not always necessary to use 8 bit resolution, with 256 counting steps (0.39% accuracy). In some cases it is sufficient to have only four (6.25%) or five (3.1%) bits resolution of the PWM signal. For a small motor 8 bits might be overdone. If at a count of 16 (four bit resolution) the cycle restarts, the PWM frequency increases accordingly. This can be reached by combining the PWM mode with CTC: In that case compare match port A is used as CTC and compare match B as PWM signal. Those two modes are possible, as the mode table shows.

Table 11-8. Waveform Generation Mode Bit Description

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on ⁽¹⁾⁽²⁾
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	TOP	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	TOP	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

In case of a 4 bit resolution (Compare Match A = 15), with 1.2 MHz clock and a timer prescaler of 1 the PWM frequency will be at 37.5 kHz, well above any audible regions (excluding bats).

In case of 16 bit timers/counters, of which ATtiny13 has none, the normal phase correct PWM mode would have a

resolution of 16 bits (accuracy: 0.0015%), which is by far higher than any practical requirements. It is therefore nearly always necessary to reduce the resolution. To relieve Compare Match A from this CTC task and allow it to use as additional PWM channel, an additional port (ICR) provides this task.

5.2 Hardware, components and mounting

For the PWM intensity regulation the same hardware is used as for the previous lectures.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Fast PWM](#) [Phase correct PWM mode](#)

5.3 Fast PWM mode

With the previous information on functions, properties and methods a PWM intensity regulation can be programmed in a simple way. As PWM cycle frequency we select at 1.2 Mcs/s clock and a prescaler of 1 4.7 kcs/s. This is fast enough for the human eye and for any digital camera.

By changing the polarity of the PWM output we reverse the inversion by the LED. The [source code is here](#).

```
; ****
; * PWM control of a LED in Fast mode *
; * (C)2017 by www.avr-asm-tutorial.net *
; ****

; .NOLIST
INCLUDE "tn13def.inc"
; .LIST
;

; ----- Register -----
.def rmp = R16 ; multi purpose register
;

; ----- Program -----
    ; Init output pin OC0A
    sbi DDRB,DDB0 ; OC0A as output
    ; PWM compare value to timer 0
    ldi rmp,20 * 256 / 100 ; 20% intensity
    out OCR0A,rmp ; to compare match port A
    ; Timer 0 in Fast PWM mode, output A low at cycle start
    ldi rmp,(1<<COM0A1) | (1<<COM0A0) | (1<<WGM01) | (1<<WGM00)
    out TCCR0A,rmp ; to timer control port A
    ; Start Timer 0 with prescaler = 1
    ldi rmp,1<<CS00 ; Prescaler = 1
    out TCCR0B,rmp ; to timer control port B
    ; Enable sleep mode
    ldi rmp,1<<SE ; Sleep enable, mode idle
    out MCUCR,rmp ; to Universal control port

Loop:
    sleep ; go to sleep
    nop ; in case of wakeup
    rjmp Loop ; again to sleep
;

; End of source code
;
```

One line in this code requires deeper understanding and some background knowledge:

```
ldi rmp,20 * 256 / 100 ; 20% intensity
```

The term $20 * 256 / 100$ usually would result in 51.2. Not so in assembler. All mathematical expressions are integer operations. That means the result is 51, not 51.2. But even more strange: if we write the term " $20 / 100 * 256$ ", which would be - mathematically spoken - equivalent, the result would be 0. We can test that astounding result by adding the line

```
; ----- Constants -----
.equ cIntensity = 20 / 100 * 256
;
```

and reformulate the above line by

```
ldi rmp,cIntensity
```

If we assemble that changed code using gavrasm and the `-s` option (note that other assemblers do not provide such a symbol list) we will see the following in the symbol list:

List of symbols:

Type	nDef	nUsed	Decimalval	Hexvalue	Name
T	1	1	18	12	ATTINY13
L	1	2	9	9	LOOP
R	1	8	16	10	RMP
C	1	1	0	0	CINTENSITY

No macros.

The reason for this is that the assembler ALWAYS a) uses integer math, including interim results, and b) that he always processes expressions from left to right (unless brackets are used), and c) follows priority rules such as "*" and "/" higher than "+" and "-". So the constant calculation term is processed within the assembler as follows:

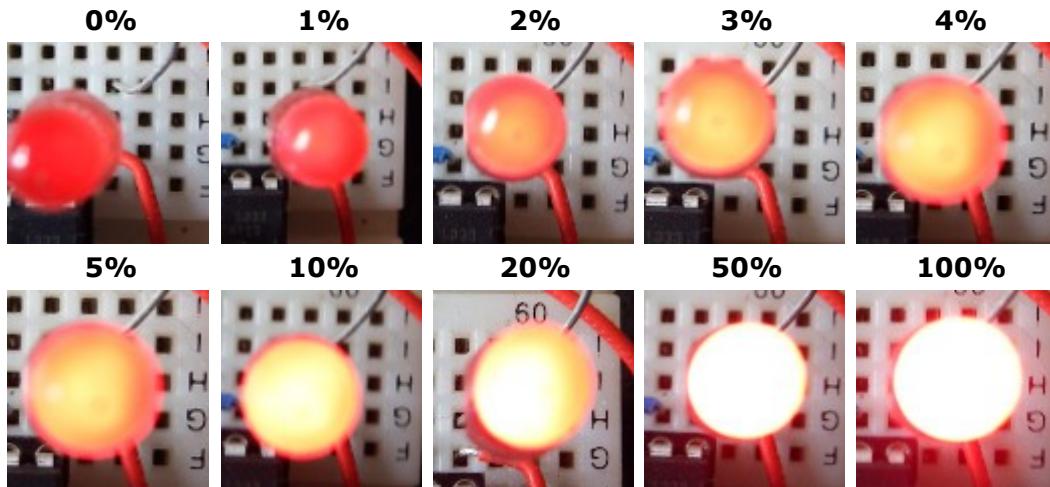
1. Divide 20 by 100 in integer mode, result = 0 (!)
2. Multiply result with 250, result = 0 (!)

If you want to overrule the standard processing scheme you'll have to use brackets. So the formulation

```
.equ cIntensity = (20 * 250) / 100
```

is always more correct as it overrules any priority and processing sequences. See [this page for a complete list of mathematical expressions in assembler](#). This list is sorted by increasing priority.

The PWM intensity control is rather simple code. And here is the result:



The differences in intensities are clearly visible.

Please note that the PWM cannot be set to 0% intensity because the first clock pulse is always executed. If 0% should be really switching the LED off one has to decouple the output pin from the PWM control and to set the port pin to high level.

5.3.1 Simulation of the Fast PWM mode

[avr_sim](#) can be used to test the timing characteristics.

After the 6.67 µs init phase the timer 0 is in Fast PWM mode. The compare value in A is 51,

PB0 is cleared on the start and, on compare match, PB0 will be set.

The screenshot shows two windows from AVR Studio:

- Timer Window:**
 - Status: 8 bits, Mode: Fast PWM, overflow.
 - PortRegisters: TCNT = \$01=1, Compare A = \$33=51, Compare B = \$00=0.
 - TOP = \$FF=255.
 - Prescaler: Clock/1.
- Port Window:**

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	1c
TINnB	0	0					0	
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

That means the LED will be on over the first 51 clock cycles and off for the remaining cycles to 256 (0).

This is the situation after the first compare match comes into effect (the timer exceeds the compare match value). 42.5 μ s are over.

The screenshot shows two windows from AVR Studio:

- Timer Window:**
 - Status: 8 bits, Mode: Fast PWM, overflow.
 - PortRegisters: TCNT = \$34=52, Compare A = \$33=51, Compare B = \$00=0.
 - TOP = \$FF=255.
 - Prescaler: Clock/1.
- Simulation Window:**
 - Restart Step Run Stop.
 - Simulation status:
 - Prog counter = \$000009
 - Instructions = 9
 - Stackpointer = \$0000
 - Clock frequ. = 1,200,000 Hz
 - Time elapsed = 49.166667 us
 - Stop watch = 42.5 us

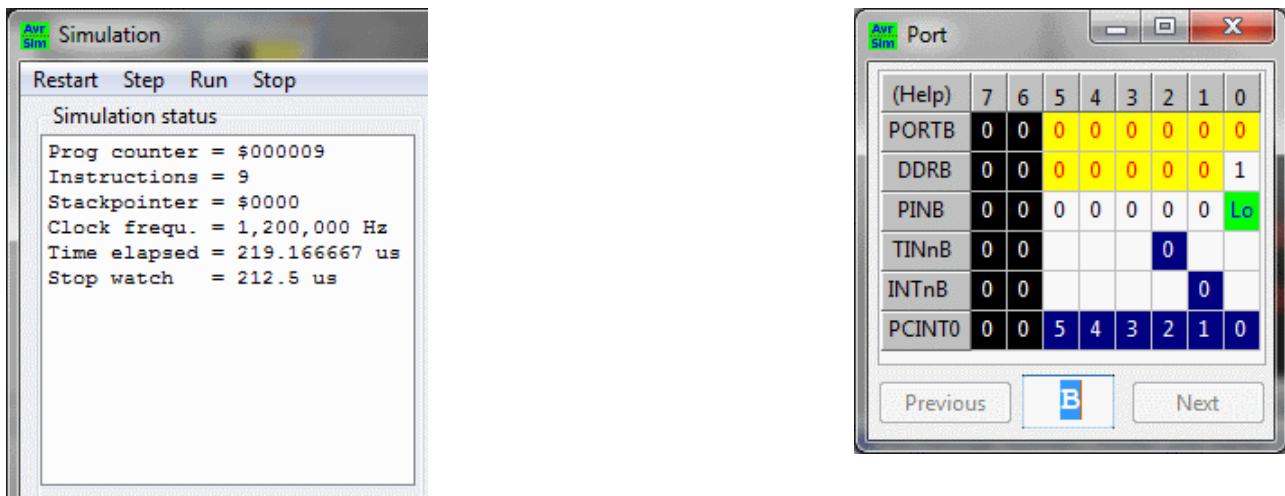
The first match has set the port-bit PORTB0, by that switching the LED off.

The screenshot shows two windows from AVR Studio:

- Port Window:**

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	0	1c
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0
- Timer Window:**
 - Status: 8 bits, Mode: Fast PWM, overflow.
 - PortRegisters: TCNT = \$00=0, Compare A = \$33=51, Compare B = \$00=0.
 - TOP = \$FF=255.
 - Prescaler: Clock/1.

Here, the timer overflowed from 255 to 0 after 212.5 μ s (corresponding to a PWM frequency of $1,200 / 256 = 4.7$ kHz).



The overflow has cleared PORTB0, the LED is on again and the PWM cycle restarts.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Fast PWM](#) [Phase correct PWM mode](#)

5.4 Timer in phase correct PWM mode

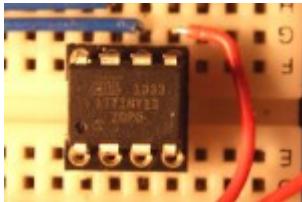
The phase correct PWM mode is achieved by removing the WGM01 bit in the above source code ($WGM01 = 0$). Now the PWM works at 2.34 kcs/s. Still fast enough for this application, but a motor might hum in an audible frequency range.

The difference to Fast PWM mode is that the timer, when it reaches TOP at 255, it does not restart but it counts downwards, and it clears PORTB0 only later when the down-count reaches the compare value again during down-count. Moreover: If we would have changed the compare value in between, which we did not in this simple program, this new compare value would not come into effect immediately but only until TOP has been reached: the compare value is temporarily stored and loaded only at TOP.

That demonstrates how the timer can be used to largely simplify the overhead by the controller: complex up-and-down loops with exact timing and counting are automated, the controller sleeps instead - or can do other things without disturbing counting and timing (as we will see in the next lecture). So whenever you need exact pin switching, use a timer to do the whole work and do not undertake efforts to use loops and counting - this will leave you frustrated, unnecessarily twists your brain and inks your hair in direction to gray. A timer is a clever solution, so you can concentrate on the other tasks that really need your valuable and full attention.

Be aware that timer controlled output pins are generally fixed to certain locations in the output port called OCnA and OCnB. Those pins are sacrosanct and should be banned from other uses, so respect this design constraint with a very high priority.

[Home](#) [Top](#) [Introduction](#) [Hardware](#) [Fast PWM](#) [Phase correct PWM mode](#)



Lecture 6: A LED blinks with the timer interrupt

With this lecture we enter into the large and powerful world of interrupt programming and say good bye to simple linear program execution. As you will see, interrupts enable a whole world of complex operations while linear programming allows to solve only simple projects and requirements. With interrupts the controller will execute several tasks (seemingly) in parallel. If you are familiar with that you will never return to the linear method. But be aware that this requires lots of new techniques that are far outside the linear program execution world (who said that assembler programming will be easy learning as it offers so many new opportunities?).

6.0 Overview

1. [Introduction to interrupt programming](#)
2. [Hardware, components, mounting](#)
3. [Timer with Overflow interrupt](#)
4. [Timer with CTC interrupt](#)

6.1 Introduction to interrupt programming

6.1.1 Interrupts

The main advantage of interrupt programming is that several processes can run in parallel and that all arising conditions that require handling can be handled in due time and correct. This is the end of any delay loops with exact execution times.

Interrupts are automated breaks that occur whenever certain conditions come true. They interrupt normal program execution, perform so-called interrupt service routines and return back to exactly the location where normal execution was interrupted. For example, if the timer overflows (from 0xFF to 0x00), a certain bit in the timer's interrupt mask port is tested. If enabled, the normal program execution is suspended and the controller

1. sets a certain bit in a port to signal that this interrupt occurred,
2. saves the current execution address (PC counter) in a certain storage place,
3. clears the Interrupt Enable bit I in its status register to prevent further interrupts from execution,
4. jumps to a certain fixed location in the flash and continues execution there, and
5. clears the set bit in the port to signal that this interrupt is actually processed (it could well be that an interrupt cannot be processed immediately, so the bit continues to signal that interrupt execution is still to be performed).

The further program execution is a matter of the self-designed interrupt service routine (ISR), e.g. what has to be done if the timer overflows. After having done what has to be done in case of a timer overflow

6. the Interrupt-Enable bit I in the status register has to be set again,
7. the saved program execution address is read from the storage, written back to the PC counter and program execution continues there (where execution was interrupted).

This execution flow has several serious consequences:

- As interrupts can occur at any time during normal execution, the use of registers in ISRs has to be exclusive. If a register's content, used during normal execution, would be changed during ISR execution, this would have unpredictable consequences.

- ISR instructions that change status register flags would have the same consequence. So either avoid instructions that change those flags (very unpractical) or save the whole status register into a normal register and restore its content at the end of the ISR (R15 is a good place for that).
- An ISR is not interrupted by another interrupt, no nested interrupts are possible. This has consequences for the registers used in interrupts: one can use the same registers in all ISRs temporarily, because only one of those can be executed at a time.
- As interrupt requests can occur at any time, the case that two or more can occur at the same time cannot be excluded. For this case a priority rule is necessary to determine which interrupt comes first. This rule is implemented by the AVR hardware and cannot be changed.
- As long as the respective ISR is not finished any further interrupt processing is blocked by the cleared Interrupt Enable bit I. If the ISR is a lengthy, never-ending process no further interrupts will be processed. If during that time another timer overflow occurs it will be detected later (by the set flag), but if another timer overflow occurs this one (and all later one's) will be missed. So the rule is: keep ISRs as short as possible, and in any case shorter than the most time-critical other interrupts. Make sure that overwhelming interrupt execution is avoided otherwise your controller will not have time enough to process its other tasks.
- Any ISR has to end with a set I flag, otherwise further interrupts will not be processed any more.

Bit	7	6	5	4	3	2	1	0	SREG
Read/Write	I	T	H	S	V	N	Z	C	
Initial Value	0	0	0	0	0	0	0	0	

This is the location of the I bit in the status register. It is by default cleared. So to execute interrupts requires to set this bit. Don't forget this. This is done with the SEI instruction. If you need to (temporarily or permanently) disable interrupt execution use the CLI instruction to clear this bit.

This bit is automatically cleared at the start of the ISR, it has to be set at the end of the ISR (see below for a respective instruction).

6.1.2 Stack storage

To enable the execution of interrupts the controller needs a storage where the address of the PC can be temporarily stored after having been interrupted. For this the AVRs use a so-called stack in the static memory. The ATTiny13 has 64 bytes of SRAM on board, by far enough to store those two bytes. Those are not at fixed locations but are dynamically allocated on the stack. The stack is an SRAM area that starts at the last SRAM location (in def.inc defined as RAMEND) and grows downwards, to lower addresses. Its address is in port SPL (stack pointer Low). As the ATTiny13 has less than 256 bytes SRAM, the port SPH (stack pointer High) is not necessary in our case.

The throw of the PC's address at an interrupt is decreasing SPL by two bytes. After re-storing the PC to its original value before interrupting, SPL is having its previous value again.

The stack can further be used to call subroutines and, after execution, to return to the calling location. The respective instruction is RCALL: it pushes the PC (low and high byte) to the stack, jumps to the given address (resp. a relative displacement) and at the instruction RET back to the calling address.

The same is the case at an interrupt. At returning back the I flag in the status register has to

be set to enable pending and further interrupts. The return from the ISR with the instruction RETI additionally sets the I flag.

6.1.3 Interrupt vectors

To which location in the flash storage does an interrupt jump? To the first locations in the flash storage called RESET and INTERRUPT vectors. In an ATtiny13 those are:

#	Address	Name	Description
0	0000	RESET	Jump address after reset, by applying the operating voltage, at brown-out detection and at a watchdog reset
1	0001	INT0	Level change on the INT0 input pin
2	0002	PCINT0	Level change on an input pin
3	0003	TIMO_OVF	Timer 0 overflow
4	0004	EE_RDY	Finished a write cycle to the EEPROM
5	0005	ANA_COMP	Polarity change on the analog comparer
6	0006	TIMO_COMPA	Timer 0 Compare A
7	0007	TIMO_COMPB	Timer 0 Compare B
8	0008	WDT	Watchdog event
9	0009	ADC	AD conversion complete

In later parts of the course we will use all these vectors, excluding the watchdog.

As the vectors are consecutive and have only one word (in larger ATtiny and ATmega two words) a vector has to be either a return from the interrupt or a jump instruction. The first 10 instruction words in an ATtiny13 assembler program therefore always look like this:

```
; Reset- and vector table
;
.CSEG ; Assemble to the flash storage (Code SEGment)
.ORG 0 ; Address to zero (Reset- and interrupt vectors start at zero)
        rjmp Start ; Reset Vector, jump to init
        reti ; INT0-Int, inactive
        reti ; PCINT-Int, inactive
        reti ; TIM0_OVF, inactive
        reti ; EE_RDY-Int, inactive
        reti ; ANA_COMP-Int, inactive
        reti ; TIM0_COMPA-Int, inactive
        reti ; TIM0_COMPB-Int, inactive
        reti ; WDT-Int, inactive
        reti ; ADC-Int, inactive
;
; Program init at Reset
;
Start:
        ; [Here the program starts]
```

The RETI instruction on all inactive vectors ensures that in case of an unplanned activation of an interrupt the return back is executed in a systematic manner.

More chaotic software designers are incautious and set the vector address with an .ORG directive. They forget that a flash memory location cannot be empty but still contains 0xFFFF, an NOP instruction. If they accidentally enable a timer 0 compare match interrupt funny things will happen. In the most probable case a different ISR is executed or the I flag is never set and further interrupts are simply blocked or the init routine restarts all indefinitely. Good luck with debugging and identifying the error.

6.1.4 The timer overflow interrupt

Bit	7	6	5	4	3	2	1	0	
ReadWrite	-	-	-	-	OCIE0B	OCIE0A	TOIE0	-	TIMSK0
Initial Value	0	0	0	0	0	0	0	0	

To use this interrupt the Overflow-Interrupt-Enable-Flag TOIE0 in the Timer Interrupt Mask Register TIMSK0 has to be set, e.g. with

```
ldi R16,1<<TOIE0  
out TIMSK0 R16
```

Of course,

- the timer must be in counting mode to reach an overflow state,
- the I bit in the status register must be set to enable interrupts,
- the stack pointer SPL has to be set to RAMEND.
- And, of course, an ISR has to be written so that the jump in the vector table finds a location.

Such a typical ISR could be:

```
ovflw_isr:  
    in R15,SREG ; save status register in a register  
    dec R17 ; count down a register  
    brne ovflw_isr1 ; jump if not zero  
    sbr R18,0b00000001 ; set bit 0 in register R18  
    ldi R17,10 ; restart count down from 10  
ovflw_isr1:  
    out SREG,R15 ; restore status register  
    reti ; return from interrupt and set I-flag
```

Typical is the saving of the status. Because the DEC and SBR instructions within the ISR change flags, it is absolutely crucial to do that.

Further, the ISR sets a flag in a register to signal that the timer reached overflow for ten times. This flag can be used outside the ISR to perform further operations, e.g. to write a character to an LCD or do other time-consuming operations. The ISR works fine as there is enough time: twenty timer overflows until the flag is overwritten should provide enough time to react.

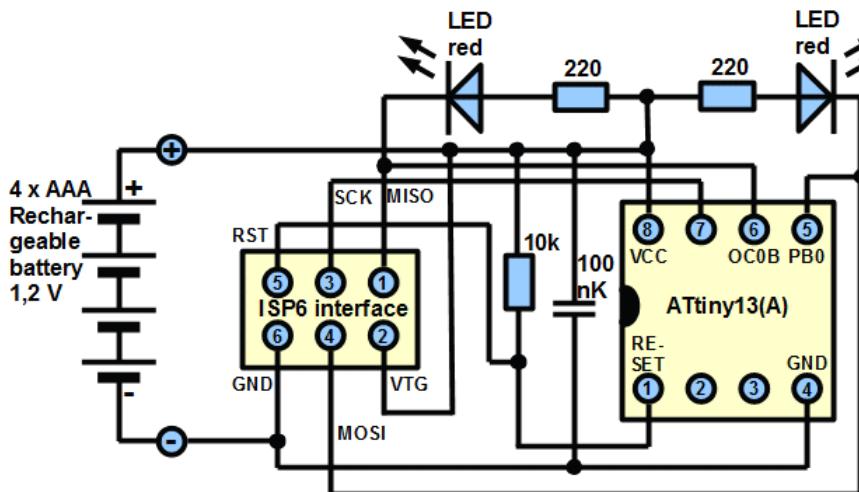
6.1.5 The compare match interrupt

If the compare match interrupt bits OCIE0A and/or OCIE0B are set every match leads to a jump to the respective vector. If the timer is in CTC mode, the compare match A interrupt is initiated on every reset of the timer. The same applies to the PWM mode.

6.1.6 Interrupts and sleep modes

In the sleep mode "idle", that we used already, all interrupts wake up the controller. Execution continues with the ISR of the respective interrupt source, then continues with the instruction following the sleep instruction. In this phase it makes sense to check if any of the ISR routines has set flags that require extended reaction.

6.2 Hardware, components, mounting



For the interrupt experiments the same hardware as already used comes into play. The only change is that an additional red LED and a resistor is attached to OC0B (on pin 6). These components are already described in lecture 2.

6.3 Timer with overflow interrupt

6.3.1 Task description

The following task has to be performed:

- The LED on OC0A blinks in second mode.
- At each fifth second the blinking is suppressed, the LED remains dark for a second.
- A further gimmick: the LED on port pin OC0B shall softly change its intensity.

6.3.2 Steps towards solution

It is clear from the task description that it requires a lot more than simple blinking mechanics. The parallel change in intensity with the second LED also requires some new methods. The solution is an interrupt-controlled counting method.

Such a program looks basically very different than a linear program. As most of the controller software is interrupt-driven a closer look into such structure is recommendable.

We use the timer overflow interrupt. The 1.2 Mcs/s controller clock divided by 256 pulses that are necessary to overflow the timer leads to 4,687.5 interrupts per second, if the prescaler would be at 1. If we divide this by 2 (for 0.5 s ON cycle and 0.5 s OFF cycle), we are at 2,344 ints per second (rounded up). As this is larger than 256 we need a 16 bit wide counter.

Because the masking of the fifth LED pulse requires some more complex processes we place this outside the interrupt service routine. This would not be necessary in any case because the next overflow int is 256 clock cycles later and we have only one single ISR to process, but we do that to learn the principle.

6.3.3 Program

This here is one of the possible solutions, [here is the source code](#). The rigid listing of used registers, ports and port bits, the distinct naming of all constants and the complete interrupt vector list is not only for educational purposes but is a recommendable principle in assembler programming as it eases debugging and finding errors. This and the extended comments in the source code allows to understand the code even weeks after.

```
;  
; *****
```

```

; * Timer with Overflow interrupt      *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ----- Registers -----
; free: R0 .. R14
.def rSreg = R15 ; SREG interim storage
.def rmp = R16 ; multi purpose register
.def rimp = R17 ; multi purpose inside interrupts
.def rFlag = R18 ; Flag register
    .equ bPol = 0 ; Flag polarity change
.def rCnt= R19 ; Blink counter
.def rPwm = R20 ; PWM counter
; free: R20 .. R23
.def rCntL = R24 ; 16 bit counter, LSB
.def rCntH = R25 ; dto., MSB
;free: R26 .. R31
;
; ----- Ports, port bits -----
.equ pLedOut = PORTB ; LED output port
.equ bLedOut = PORTB0 ; LED ON/OFF output pin
.equ pLedDdr = DDRB ; LED direction port
.equ bLedDdr = DDB0 ; LED direction pin
.equ pLedIn = PINB ; LED input port
.equ bLedIn = PINB0
;
.equ bPwmOut = PORTB1 ; LED PWM output pin
.equ bPwmDdr = DDB1 ; Direction pin PWM
;
; ----- Timing -----
.equ cClock = 1200000 ; Controller clock
.equ cPolChange = 2 ; Polarity changes per cycle Led On/Off
.equ cPresc = 1 ; Prescaler timer
.equ cCount = cClock / 256 / cPresc / cPolChange + 1
.equ cBlink = 5 ; Blink counter
;
; ----- Reset- and Interrupt-vectors -----
.CSEG ; Program code segment
.ORG 0 ; Reset- and vector address
    rjmp Start ; Reset vektor, jump to init
    reti ; INT0-Int, inactive
    reti ; PCINT-Int, inactive
    rjmp Tc0Isr0 ; TIM0_OVF, active
    reti ; EE_RDY-Int, inactive
    reti ; ANA_COMP-Int, inactive
    reti ; TIM0_COMPA-Int, inactive
    reti ; TIM0_COMPB-Int, inactive
    reti ; WDT-Int, inactive
    reti ; ADC-Int, inactive
;
; ----- Interrupt-Service-Routines -----
Tc0Isr0: ; Timer 0 Overflow ISR
    in rSreg,SREG ; save status register
    sbiw rCntL,1 ; decrease counter by one
    brne Tc0Isr01 ; jump if not zero
    sbr rFlag,1<<bPol ; set flag polarity change
    ldi rCntH,HIGH(cCount) ; restart counter
    ldi rCntL,LOW(cCount)
Tc0Isr01:
    mov rimp,rCntL ; copy Low byte counter
    andi rimp,0b00011111 ; isolate the lower five bits
    brne Tc0Isr02 ; jump if not zero
    dec rPwm ; decrease PWM value
    out OCR0B,rPwm ; to compare match port B
Tc0Isr02:
    out SREG,rSreg ; restore status register
    reti ; End of ISR, set I bit
;
; ----- Init and Program loop -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; stack pointer to end of SRAM
    out SPL,rmp ; set stack pointer

```

```

; Activate LED output pins
ldi rmp,(1<<bLedDdr) | (1<<bPwmDdr) ; LED output pins on, drivers on
out pLedDdr,rmp ; to direction port
; Init counter
ldi rCnt,cBlink ; init Blink counter
ldi rCntH,HIGH(cCount) ; 16 bit counter to start value
ldi rCntL,LOW(cCount)
; Init compare register for PWM
ldi rmp,0xFF ; Compare match A to 255
out OCR0A,rmp ; to compare match port A
clr rPwm ; Start value for PWM
out OCR0B,rPwm ; to compare match port B
; Start Timer/Counter 0 as timer with overflow int
ldi rmp,(1<<COM0B1) | (1<<WGM01) | (1<<WGM00) ; Fast PWM, COM0B
out TCCR0A,rmp ; to timer control port A
ldi rmp,1<<CS00 ; prescaler 1, start timer
out TCCR0B,rmp ; to timer control port B
ldi rmp,1<<TOIE0 ; overflow interrupt
out TIMSK0,rmp ; to timer interrupt mask
; Sleep mode
ldi rmp,1<<SE ; sleep enable, mode idle
out MCUCR,rmp ; to universal control port
; Enable interrupts
sei ; set I flag in status register
; Program loop
Loop:
    sleep ; send to sleep
    nop ; delay one cycle after wake-up
    sbrc rFlag,bPol ; jump over next instruction if flag not set
    rcall Polarity ; process flag polarity change
    rjmp Loop ; send to sleep again
;
; ----- Change polarity of the LED -----
Polarity:
    cbr rFlag,1<<bPol ; clear flag
    sbis pLedOut,bLedOut ; jump over next instruction if LED is off
    rjmp Polarity3 ; Led is on, switch LED off
    ; Led is Off
    cpi rCnt,0 ; is blink counter zero?
    breq Polarity1 ; yes, restart counter
    ; Led off, counter not zero
    dec rCnt ; decrease blink counter
    brne Polarity2 ; switch Led on
    ret ; no polarity change, return
Polarity1: ; Restart counter
    ldi rCnt,cBlink ; restart counter
    ret ; no polarity change, return
Polarity2: ; switch Led on
    cbi pLedOut,bLedOut ; Led on
    ret ; return, LED on
Polarity3: ; switch Led off
    sbi pLedOut,bLedOut ; Led off
    ret ; return, Led off
;
; End of source code
;

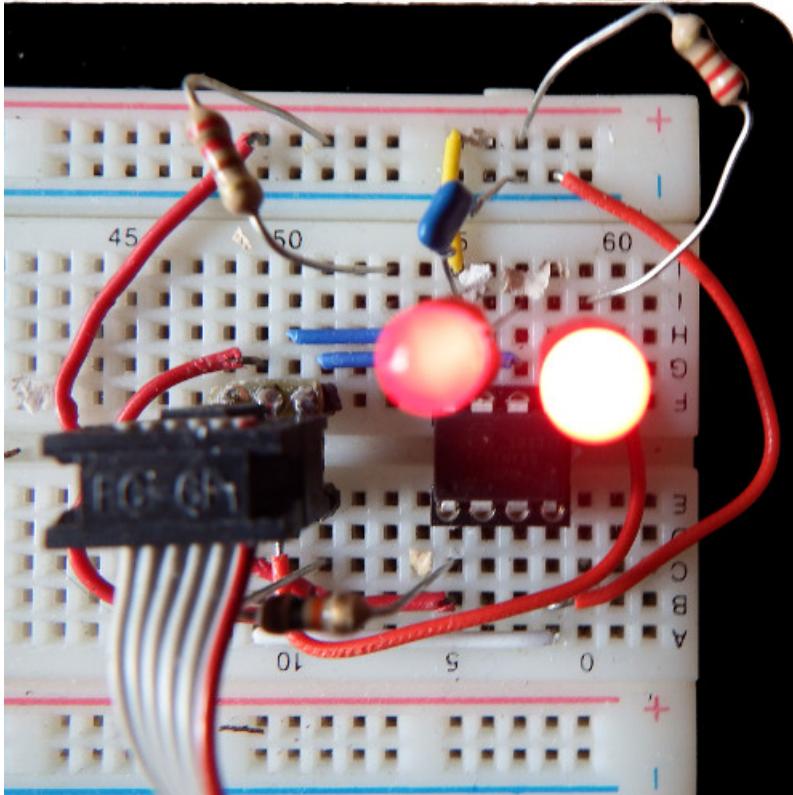
```

New are the following instructions:

- IN Register,Port: reads all eight bits in a port to a register,
- SBR Register,Bitmask: set all bits in the register, that are set in the mask, works only with registers R16 and higher,
- CBR Register,Bitmask: löscht alle Bits in einem Register, die in der Maske Eins sind, works only with registers R16 and higher,
- ANDI Register,Mask: clears all bits in the register, that are clear in the mask, works only with registers R16 and higher,
- CPI Register,Constant: compares the register with the constant (subtracts but does not store the result) and sets/clears the flags in the status register, works only with registers R16 and higher,
- BREQ Label: jumps to the address Label, if the Z flag in the status register is set,
- SBRC Register,Bit: jumps over the next instruction if the bit in the register is zero, works only with registers R16 and higher,
- SBRS Register,Bit: jumps over the next instruction if bit in the register is set, works only

- with registers R16 and higher,
- SBIS Port, Bit: jumps over the next instruction if bit in port is set, works only with ports smaller than 32 decimal,
- SBIC Port, Bit: jumps over the next instruction if bit in port is clear, works only with ports smaller than 32 decimal.

6.3.4 The result



service routines.

This structure makes sense in all interrupt controlled programs, in order to keep the overview. This also helps with potential debugging activities if the program does not do what it should.

6.3.6 Simulation of the processes

Simulation done with `avr_sim` is done with the following framework conditions:

- Controller clock: 1.2 MHz
- TC0 prescaler: 1
- TC0 TOP value: 255
- Overflow Int after $256 / 1,200,000$: 213 μ s
- 16-bit downcounter: 2,344
- Polarity change every $2,344 * 213 \mu\text{s}$: 500 ms
- Change frequency = $2 * 500 \text{ ms}$: 1 second

These are the LEDs in different modes: the one to the right with abrupt on and off cycles, the one to the left with softly changing intensity. Current requirements are minimized by the sleep mode: while the controller is majorly sleeping and is only woken up when really needed.

6.3.5 Program structuring

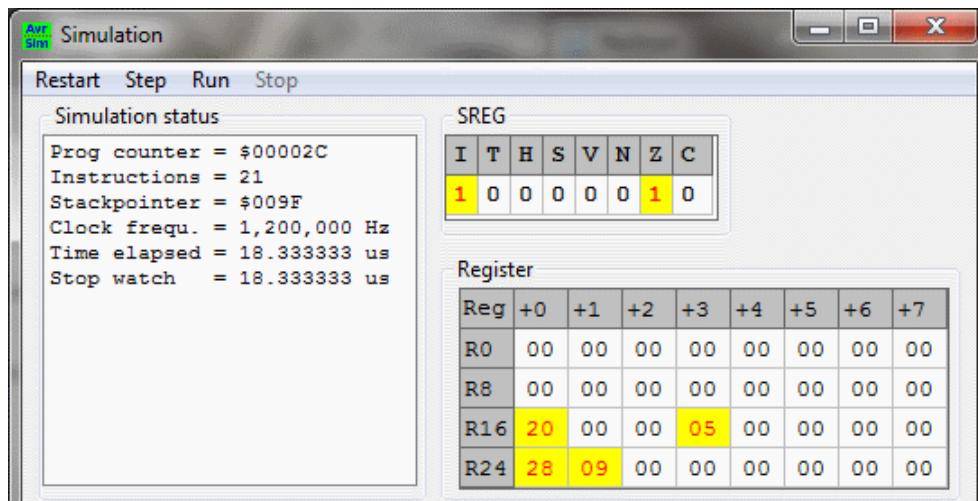
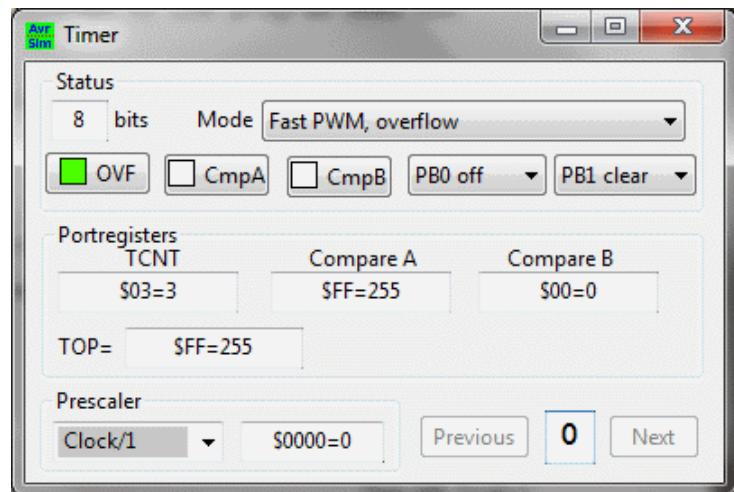
The visible structure of the source code is:

- Register definitions, port definitions and constants at the beginning,
- Reset- and interrupt vectors,
- Interrupt service routines,
- Main program init with hardware configuration and start values,
- Loop processing, and
- routines outside interrupt

	7	6	5	4	3	2	1	0
(Help)	0	0	0	0	0	0	0	0
PORTB	0	0	0	0	0	0	1	1
DDRB	0	0	0	0	0	0	1	1
PINB	0	0	0	0	0	0	Lo	Lo
T1NnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

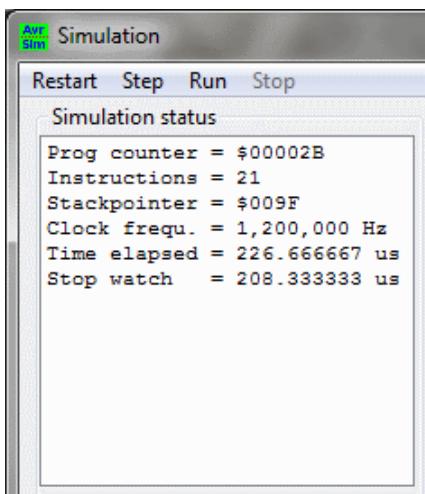
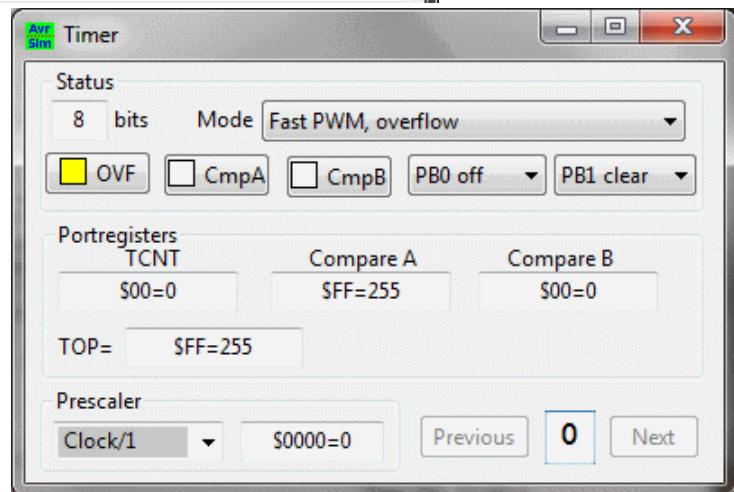
Previous B Next

This is the situation after the init phase. The timer TC0 is in Fast PWM mode, its TOP and its compare match A value is set to 255, the PWM compare match value B is zero, it prescales clock by 1 and its overflow interrupt enable bit is set. In port B the two direction bits for pins PB0 and PB1 are set, while the portbits are zero, so both LEDs are switched on.



Init required 18.33 μ s, set the stack-pointer to the end of SRAM in the device, set the 16-bit counter in R25:R24 to 2,344 (=0928), set the LED down counter in R19 to 5 and enabled the interrupt bit in SREG.

The first overflow occurred, the timer restarts and has its overflow interrupt flag set. With the next instruction, the controller executes the interrupt and jumps to the interrupt service routine.



The overflow occurs after 208.3 μ s, together with the int enable, the I flag enable and the sleep enable instructions 213 μ s.

On overflow the portbit PB1 has been set and the PWM cycle starts.

The Port window shows the state of various registers. The PORTB register is displayed as a 8x8 grid where bit 1 (PB1) is set to 1. Other bits are 0. Buttons at the bottom include Previous, B (highlighted), and Next.

With the next instruction executed, the compare match B is executed, because TCNT was at 0 and Compare Match B was also at zero in the last clock period. The LED was off for one clock cycle and is switched on again. Remember: such an output cannot

The Port window shows the state of various registers. The PORTB register is displayed as a 8x8 grid where bit 1 (PB1) is set to 1. The INTnB register shows bit 0 set to 1. Buttons at the bottom include Previous, B (highlighted), and Next.

be completely on, it is off for at least one timer cycle.

With the next instruction the requested interrupt is executed. The controller dropped the current execution address to the stack and has jumped to the vector address.

The Timer window shows the configuration for a timer. Mode is set to "Fast PWM, overflow". TCNT is \$04=4, Compare A is \$FF=255, and Compare B is \$00=0. Prescaler is set to "Clock/1". Buttons at the bottom include Previous, 0 (highlighted), and Next.

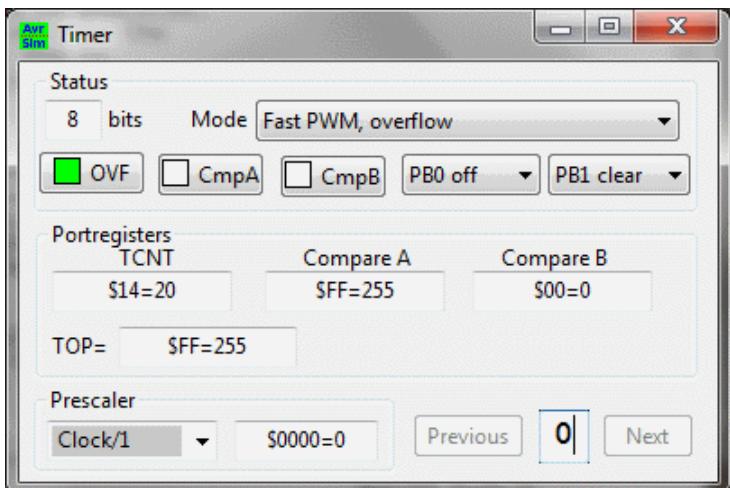
The Simulation window shows the SREG register. The I flag is set to 1, while other flags T, H, S, V, N, Z are 0. The Register section shows the stack pointer at \$009D.

With branching to the interrupt vector the I flag in SREG is cleared, which blocks other interrupts.

The Simulation window shows the SREG register. The I flag is now cleared to 0, while other flags T, H, S, V, N, Z are 0. The Register section shows the stack pointer at \$009D and the R24 register value at 27.

On interrupt execution the stackpointer has decreased by two and the I flag in the status register is cleared.

Within the ISR the 16-bit counter in R25:R24 is decreased. As it is not zero yet (see Z flag in SREG), the polarity change bit is not set during this turn.



Following the RETI instruction the TC0 interrupt is terminated, leaving the timer overflow interrupt enabled, the stackpointer returns to the top of the SRAM and the I flag is set. As the timer interrupt woke up the controller, the code following the sleep instruction will be executed after the interrupt service routine has finished. This code is executed 2,343 times with the polarity change flag cleared. The 2,344th time the flag is set and the LED output pin PB0 changes its polarity.

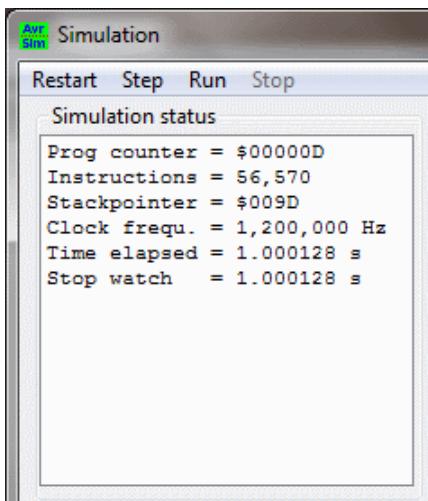
Execution of RETI sets the I flag in SREG and increases the stackpointer.

A special condition occurs if the compare match B value is changed within the execution of the interrupt service routine. As the compare match value in Fast PWM mode is only updated when the TOP of the timer is reached, the new value (255) is written to the compare match B port-register at the end of the current PWM cycle. The yellow background signals that there is a difference between the displayed and the effective value. This mechanism ensures that no repeated compare match occurs in the same cycle and that the same compare match value is applied over the whole cycle.

This is the time when the first counting cycle is over:

- The 16-bit counter in R25:R24 has reached zero.
- The flag bit in R18 has been set, triggering the polarity change routine later on,
- Roughly 500 ms have been elapsed.

The counting time for the blinking LED on PB0 is fairly exact. It would be more exact if we would clear the stopwatch and repeat the cycle all over again, breaking execution always on the same code location.



This is the end of the second cycle. Approximately 1 second has elapsed until the LED on PB0 will be switched on again.

6.3.7 Advantages and disadvantages

The advantage of this solution is high flexibility: it can simply be modified to fit to other needs. The frequency of blinking or changes of the LED port pins can simply be modified by changing a few constants on top.

The disadvantage is that blinking is not exact. This is because dividing by 256 does not lead to an integer value. This disadvantage is addressed by the following opportunity.

6.4 Timer with CTC interrupt

In order to achieve a more exact timing the following is considered.

6.4.1 CTC selection

Different divider factors can be achieved in the CTC mode of the timer. Other division factors than 256 are possible. To achieve more exact timings a combination of the clock frequency, the prescaler, the CTC compare match and the software counter has to be selected in a more intelligent way. The following table provides for all clock frequencies of the ATtiny13 and for 8 and 16 bit counters the potential CTC dividers that result in an integer value of 1 cs/s.

Colors:	16 bit counter	8 bit counter	CTC=256@16 bit counter	CTC=256@8 bit counter
---------	----------------	---------------	------------------------	-----------------------

From that the following consequences result (CTC values larger than 100 only).

Clock (cs/s)	Pre- scaler	Integer CTC divider							
9,600,000	1	256 (37500)	250 (38400)	240 (40000)	200 (48000)	192 (50000)	160 (60000)	150 (64000)	
	8	250 (4800)	240 (5000)	200 (6000)	192 (6250)	160 (7500)	150 (8000)	128 (9375)	
	64	250 (600)	240 (625)	200 (750)	150 (1000)				
	256	250 (150)	150 (250)						
4,800,000	1	256 (18750)	250 (19200)	240 (20000)	200 (24000)	192 (25000)	160 (30000)	150 (32000)	128 (37500)
	8	250 (2400)	240 (2500)	200 (3000)	192 (3125)	160 (3750)	150 (4000)		
	64	250 (300)	200 (375)	150 (500)					
	256	250 (75)	150 (125)						

Clock (cs/s)	Pre- scaler	Integer CTC divider							
2,400,000	1	256 (9375)	250 (9600)	240 (10000)	200 (12000)	192 (12500)	160 (15000)	150 (16000)	128 (18750)
	8	250 (1200)	250 (1200)	240 (1250)	200 (1500)	160 (1875)			
	64	250 (150)	150 (250)						
1,200,000	1	250 (4800)	240 (5000)	200 (6000)	192 (6250)	160 (7500)	150 (8000)	128 (9375)	
	8	250 (600)	240 (625)	200 (750)	150 (1000)				
	64	250 (75)	150 (125)						
128,000	1	256 (500)	250 (512)	200 (640)	160 (800)	128 (1000)			
	8	250 (64)	200 (80)	160 (100)	128 (125)				
	64	250 (8)	200 (10)						
	256	250 (2)							

For 1.2 Mcs/s a CTC divider of 250 (125 for 2 cs/s) and a prescaler value of 64 is appropriate and convenient. Because the second LED has to be controlled by a PWM the timer has to be operated in PWM mode.

6.4.2 Program

Now an 8 bit register is sufficient as CTC counter, [here is the source code..](#)

```

;
; ****
; * Timer with COMP-A-Interrupt      *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;

.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;

; ----- Registers -----
; free: R0 .. R14
.def rSReg = R15 ; SREG interim storage
.def rmp = R16 ; multi purpose register
.def rimp = R17 ; multi purpose inside ISRs
.def rFlag = R18 ; Flag register
    .equ bPol = 0 ; Flag polarity change
.def rCnt= R19 ; Blink counter
.def rCtcInt = R20 ; CTC interrupt counter
.def rPwm = R21 ; PWM counter
; free: R22 .. R31
;
; ----- Ports, Port bits -----
.equ pLedOut = PORTB ; LED output port
.equ bLedOut = PORTB0 ; LED on/off pin
.equ pLedDdr = DDRB ; LED direction port
.equ bLedDdr = DDB0 ; LED direction pin
.equ pLedIn = PINB ; LED input port
.equ bLedIn = PINB0 ; LED input pin
;
.equ bPwmOut = PORTB1 ; second LED Pwm pin
.equ bPwmDdr = DDB1 ; second LED direction pin
;
; ----- Timing -----
.equ cClock = 1200000 ; controller clock
.equ cPolChange = 2 ; frequency Led on/off
.equ cPresc = 64 ; prescaler timer
.equ cCtcInt = 125 - 1 ; CTC int counter
.equ cCtcCnt = cClock / cPresc / cPolChange / (cCtcInt +1)
.equ cBlink = 5 ; fifth pulse off
;
; ----- Reset- and Interrupt-vectors -----
.CSEG ; Program code

```

```

.ORG 0 ; Reset- and vector table address
rjmp Start ; Reset vector, jump to init
reti ; INT0-Int, inactive
reti ; PCINT-Int, inactive
reti ; TIM0_OVF, inactive
reti ; EE_RDY-Int, inactive
reti ; ANA_COMP-Int, inactive
rjmp Tc0IsrA ; TIM0_COMPA-Int, active
reti ; TIM0_COMPB-Int, inactive
reti ; WDT-Int, inactive
reti ; ADC-Int, inactive
;
; ----- Interrupt Service Routines -----
Tc0IsrA: ; Timer 0 Compare A Interrupt
    in rSreg,SREG ; save status register
    dec rPwm ; decrease PWM counter
    brne Tc0IsrA1 ; jump if not zero
    ldi rPwm,cCtcInt ; restart PWM counter
Tc0IsrA1:
    out OCR0B,rPwm ; update PWM compare
    dec rCtcInt ; decrease ctc counter
    brne Tc0IsrA2 ; jump if not zero
    sbr rFlag,1<<bPol ; set flag polarity change
    ldi rCtcInt,cCtcCnt ; restart ctc counter
Tc0IsrA2:
    out SREG,rSreg ; restore status register
    reti ; End of ISR, set I bit
;
; ----- Init and program loop -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; stack pointer to SRAM end
    out SPL,rmp ; set stack pointer
    ; Activate LED output pins
    ldi rmp,(1<<bPwmDdr)|(1<<bLedDdr) ; LED output pins, driver on
    out pLedDdr,rmp ; to direction port
    ; Init counters
    ldi rCnt,cBlink ; init blink counter
    ldi rCtcInt,cCtcCnt ; init ctc counter
    ldi rPwm,cCtcInt ; init PWM start value
    ; Compare match A value for CTC
    ldi rmp,cCtcInt ; CTC divider 125
    out OCR0A,rmp ; to compare match port A
    ; Timer 0 as in PWM mode, start with overflow interrupt
    ldi rmp,(1<<COM0B1)|(1<<WGM01)|(1<<WGM00) ; PWM on port OC0B, Fast PWM
    out TCCR0A,rmp ; to timer control port A
    ldi rmp,(1<<WGM02)|(1<<CS01)|(1<<CS00) ; Fast PWM, prescaler 64, start
    out TCCR0B,rmp ; to timer control port B
    ldi rmp,1<<OCIE0A ; Compare A interrupt
    out TIMSK0,rmp ; to timer interrupt mask
    ; Sleep mode
    ldi rmp,1<<SE ; sleep enable, mode idle
    out MCUCR,rmp ; to universal control register
    ; Enable interrupts
    sei ; set interrupt flag in status register
; Program loop
Loop:
    sleep ; put to sleep
    nop ; delay after wakeup
    sbrc rFlag,bPol ; jump over next instruction if flag clear
    rcall Polarity ; process flag
    rjmp Loop ; go to sleep again
;
; ----- Polarity change of the LED -----
Polarity:
    cbr rFlag,1<<bPol ; clear flag
    sbis pLedOut,bLedOut ; jump over next instruction if LED is off
    rjmp Polarity3 ; Led is on, jump to LED = on
    ; Led is off
    cpi rCnt,0 ; is the blink counter zero?
    breq Polarity1 ; yes, restart counter
    ; Led off, counter not zero
    dec rCnt ; decrease blink counter
    brne Polarity2 ; switch Led on
    ret ; return, no polarity change
Polarity1: ; Restart counter
    ldi rCnt,cBlink ; restart counter

```

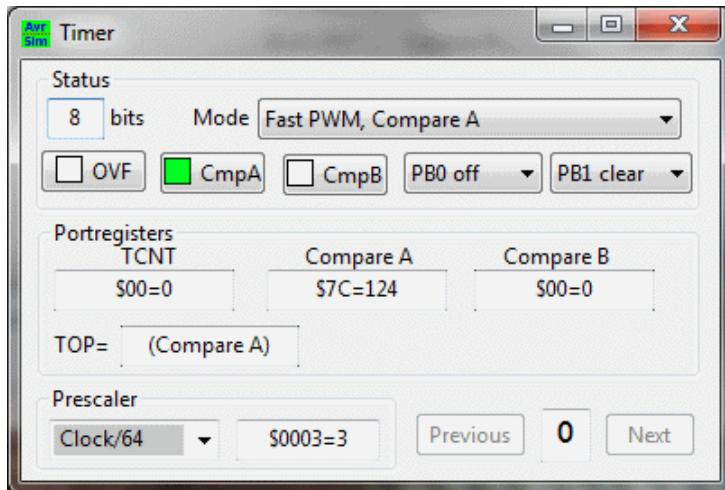
```

    ret ; return, no polarity change
Polarity2: ; switch Led on
    cbi pLedOut,bLedOut ; Led on
    ret
Polarity3: ; switch Led off
    sbi pLedOut,bLedOut ; Led off
    ret
;
; End of source code
;

```

Of course, the change in the LED timing is invisible, the difference to the previous solution is minimal. But we know for sure that this is exact because we programmed that.

6.4.3 Simulation

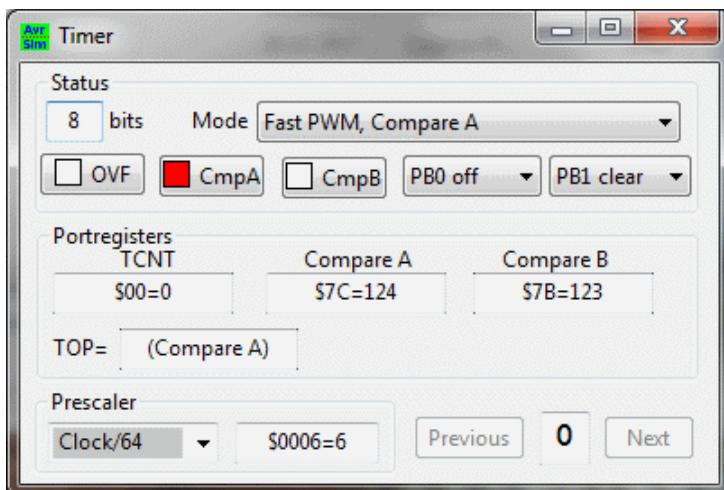


The port drives the two output pins according to the PORTB values, that will be controlled via software (PB0) resp. via timer PWM compare match B (set on PWM cycle start and cleared on compare match B).

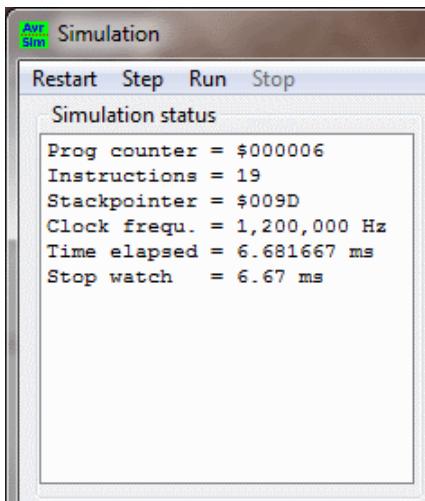
The timer now has reached the CTC value 124 and, with the next counting pulse, has cleared TCNT (CTC on compare Match A) and is executing the Compare Match A interrupt service routine. Compare Match B value has been set to 123, just one timer pulse below the CTC value. The port pin PB0 has been cleared.

Those are the configurations after initiation. The timer is in Fast PWM mode, its TOP value equals the Compare Match A value of 125 (dividing the clock signal by 125), the interrupt on compare match A is enabled and the prescaler is set to divide the controller clock by 64.

(Help)	7	6	5	4	3	2	1	0
PORTE	0	0	0	0	0	0	0	0
DDRE	0	0	0	0	0	0	1	1
PINB	0	0	0	0	0	0	Lo	Lo
TINnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0
Previous B Next								



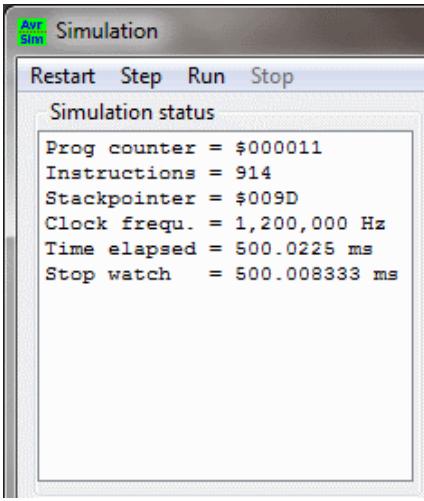
(Help)	7	6	5	4	3	2	1	0
PORTE	0	0	0	0	0	1	0	
DDRE	0	0	0	0	0	0	1	1
PINB	0	0	0	0	0	0	Hi	Lo
TINnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0
Previous B Next								



The screenshot shows the AVR-SIM simulation interface. The menu bar has 'AVR SIM' and 'Simulation'. The toolbar includes 'Restart', 'Step', 'Run', and 'Stop'. The status bar at the bottom says 'Prog counter = \$000006', 'Instructions = 19', 'Stackpointer = \$009D', 'Clock frequ. = 1,200,000 Hz', 'Time elapsed = 6.681667 ms', and 'Stop watch = 6.67 ms'. The main window displays 'Simulation status' with the same information.

Since entering the first sleep instruction 6.67 ms have elapsed, which is rather exact $125 * 64 / 1,200,000$ seconds.

To the right is the end of the first cycle when the polarity flag is set to blink the LED after half a second has been reached.



The screenshot shows the AVR-SIM simulation interface. The menu bar has 'AVR SIM' and 'Simulation'. The toolbar includes 'Restart', 'Step', 'Run', and 'Stop'. The status bar at the bottom says 'Prog counter = \$000011', 'Instructions = 914', 'Stackpointer = \$009D', 'Clock frequ. = 1,200,000 Hz', 'Time elapsed = 500.0225 ms', and 'Stop watch = 500.008333 ms'. The main window displays 'Simulation status' with the same information.

Exact timing with a timer in CTC mode can be achieved at whatever time, no matter at which frequency the controller works and what the controller has to do in between. Here we combined the second blinking with a PWM controlled LED intensity. Many other things might be tight to this timer signals of one single 8-bit timer. Design your own to fit it to your specific needs!

[Home](#)

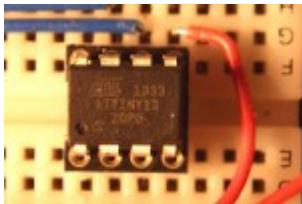
[Top](#)

[Introduction](#)

[Hardware](#)

[Overflow-Int](#)

[CTC-Int](#)



Lecture 7: A LED blinks with a key interrupt

With this lecture we learn how external level changes on a pin can be used to generate an interrupt. We use the interrupt INT0 to detect such level changes without having to poll pins in lengthy loops.

7.0 Overview

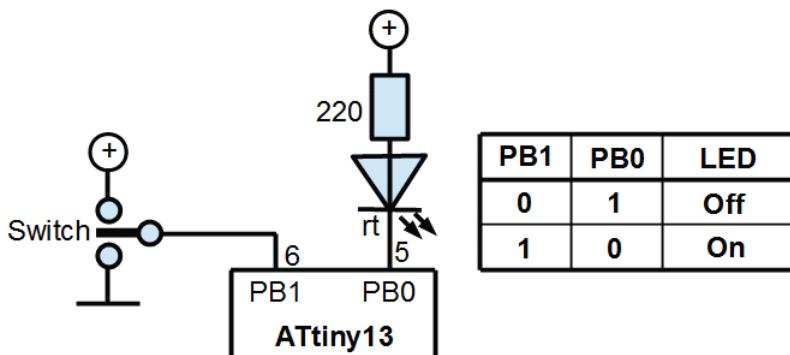
1. [Introduction to key operation and INT0 programming](#)
2. [Tasks to be performed](#)
3. [Hardware, components, mounting](#)
4. [Program](#)

7.1 Introduction to key operation and INT0 programming

7.1.1 Keys on input ports

If a port pins output driver is switched off, by clearing its DDR/port bit, the logical state of the pin can be read in and, depending from this state, decisions can be made such as conditional branching etc.

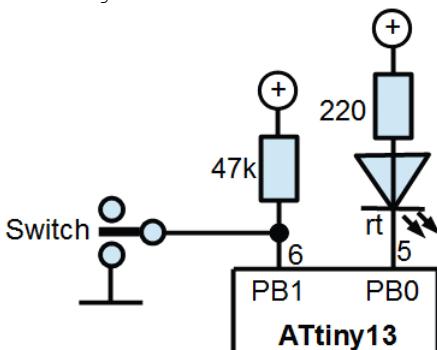
This is such a simple wiring to follow the external switch:



To light a LED on port pin PB0 as long as the switch on PB1 is high (connected to the positive operating voltage), the following program would be necessary:

```
sbi DDRB,PB0 ; enable LED output driver PB0
Loop:
    sbis PINB,PINB1 ; jump over next instruction if input pin is high
    sbi PORTB,PORTB0 ; switch LED off
    sbic PINB,PINB1 ; jump over next if input pin is low
    cbi PORTB,PORTB0 ; switch LED on
    rjmp Loop
```

Now, what happens if the switch is defect or the switch is not attached to the pin? The input is open and, due to the extremely high input resistance reacts on any changes in its proximity, e.g. electrical fields from the 50/60 cs/s electricity supply net, static voltages of fingers in close distance or level changes on neighboring pins. The result is an uncontrolled flickering of the LED.



To end the flickering the input resistance of the port pin has to be reduced and a definitive

level has to be applied. This is done with a resistor of e.g. 47 k. Now the open input has a defined high level, if the resistor is attached to the operating voltage. If the switch is closed the additional current is not very large (0.1 mA), even with a mouse piano attached (eight switches in a row). Note that the default level with such a pull-up resistor is high if no switch or key are attached.

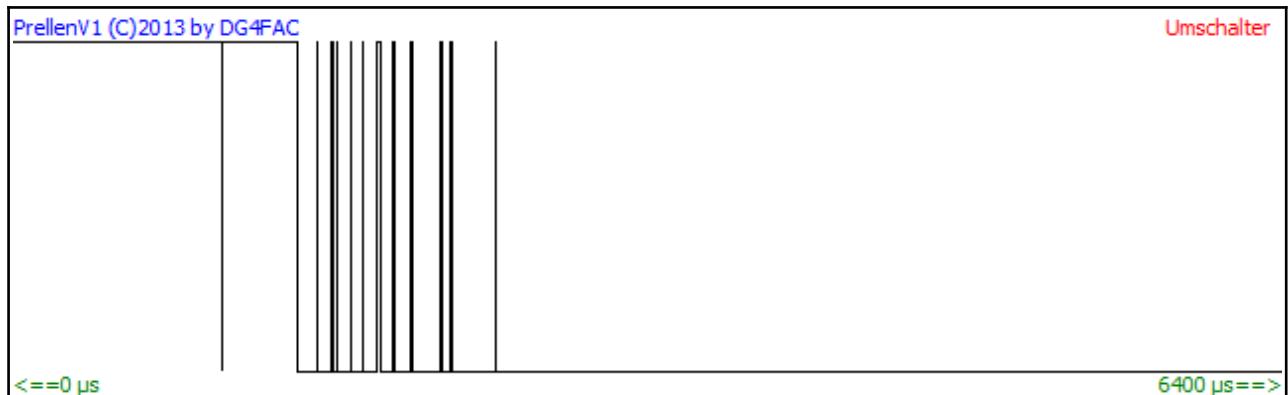
Because pull-up resistors on open input pins are necessary very often, those are built-in in AVR. They are switched on by clearing the data direction bit and setting the output bit. E.g. like this:

```
cbi DDRB,DDB1 ; direction pin PB1 = input
sbi PORTB,PORTB1 ; switch on pull-up resistor
```

Why pull-up and not pull-down? This is due to historic reasons: the input of TTL devices were by default high, so why not have CMOS inputs on the same default level. The same applies if a key or push-button is attached: it is open by default.

7.1.2 Keys and switches are bouncing

This is an example of a switch. If the switch changes polarity, this comes not into effect immediately. On a μ s or ms level the switch is bouncing before reaching a steady state. In the example case 14 level changes occur within approximately 2 ms. Such a signal swarm can cause



immense difficulties if an AVR reacts very much faster than in ms time. We have to consider bouncing in such cases. Bouncing plays a role in the next lecture.

7.1.3 The INT0 interrupt

Bit	7	6	5	4	3	2	1	0	
ReadWrite	-	PUD	SE	SM1	SM0	-	ISC01	ISC00	MCUCR
Initial Value	R	R/W	R/W	R/W	R/W	R	R/W	R/W	

is pin 6 in an ATtiny13. The interrupt branches to the INT0 vector, if the respective interrupt is enabled. The bits ISC01 and ISC00 allow to select which states or changes trigger this interrupt:

The INT0 interrupt detects levels and level changes on the INT0 input, which

ISC01		ISC00		Interrupt triggered
0	0	Low level triggers INT0 interrupt		
0	1	Any change of input level triggers INT0 interrupt		
1	0	Falling level triggers INT0 interrupt		
1	1	Rising level triggers INT0 interrupt		

The default, ISC01 = ISC00 = 0, is rather fatal because a low level on the INT0 pin triggers never ending interrupts, with leaving no time for any other activities. As the INT0 interrupt has the highest priority, no other interrupt comes through as long the low level on the INT0 input pin continues.

The ISC bits are located in the same port, MCUCR, as the SE bits for selecting sleep mode and sleep enable.

Bit	7	6	5	4	3	2	1	0	GIMSK
Read/Write	-	INT0	PCIE	-	-	-	-	-	
Initial Value	R	R/W	R/W	R	R	R	R	R	

The INT0 enable bit is located in the General Interrupt MaSK register GIMSK.

A typical instruction sequence to enable INT0 is as follows:

```
ldi R16, (1<<ISC01) | (1<<ISC00) ; rising level triggers INT0, sleep mode idle
out MCUCR,R16 ; to the universal control port
ldi R16,1<<INT0 ; set INT0 enable bit
out GIMSK,R16 ; to the General Interrupt Mask
sei ; enable interrupt processing
```

In the vector table INT0 is the first interrupt following the reset vector, with the highest priority of all interrupts. In the ATTiny13:

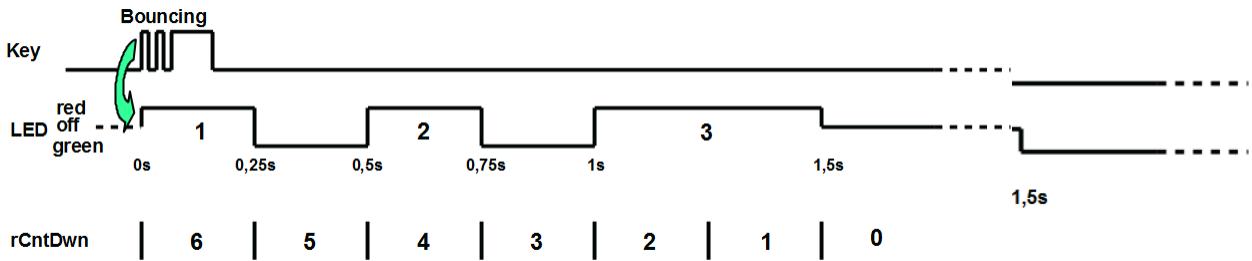
```
.CSEG ; Assemble to program flash (Code Segment)
.ORG 0 ; Address to zero (Reset- and Interrupt vectors start at zero)
        rjmp Start ; Reset vector, jump to init routine
        rjmp Int0_Isr ; INT0 int, active
        reti ; PCINT-Int, inactive
        reti ; TIM0_OVF, inactive
        reti ; EE_RDY-Int, inactive
        reti ; ANA_COMP-Int, inactive
        reti ; TIM0_COMPA-Int, inactive
        reti ; TIM0_COMPB-Int, inactive
        reti ; WDT-Int, inactive
        reti ; ADC-Int, inactive
;
; Interrupt service routine
;
Int0_Isr: ; INT0 ISR
        in R15,Sreg ; save SREG
        [...] ; further actions
        out SREG,R15 ; restore SREG
        reti ; return from int, set I flag
;
; Program init at reset
;
Start:
        ; [Here the program starts]
```

That is it and we can enter practical usage.

[Home](#) [Top](#) [Introduction](#) [Task](#) [Hardware](#) [Program](#)

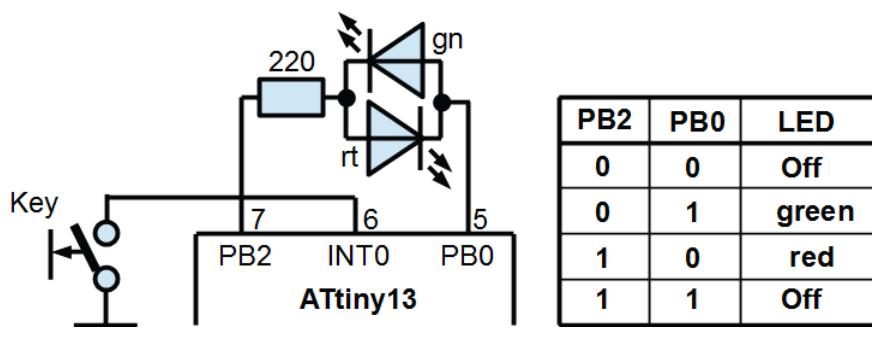
7.2 The task to be programmed

This here is the complex task. A key signal starts the sequence of the duo-led, that consists of two short and a third longer signal with the displayed timing. The program should be insensible against key bouncing.



7.3 Hardware, components and mounting

7.3.1 Hardware



The duo LED is attached to port pins PB2 (anode red) and PB0 (anode green) via a current limiting resistor. The port bit combinations and the resulting colors are listed in the table.

The key is attached to the INT0 input on PB1, the second key pin is on low (minus).

7.3.2 Components

7.3.2.1 The Duo-LED

The duo-LED consists of two LEDs, a red and a green one, connected to two pins. Note that there are other types available with three pins. The longer connecting wire is the anode of the red LED and the cathode of the green LED.



7.3.2.2 Key

This is a key. With this type two of the pins are internally connected, pushing the key connects the four pins.



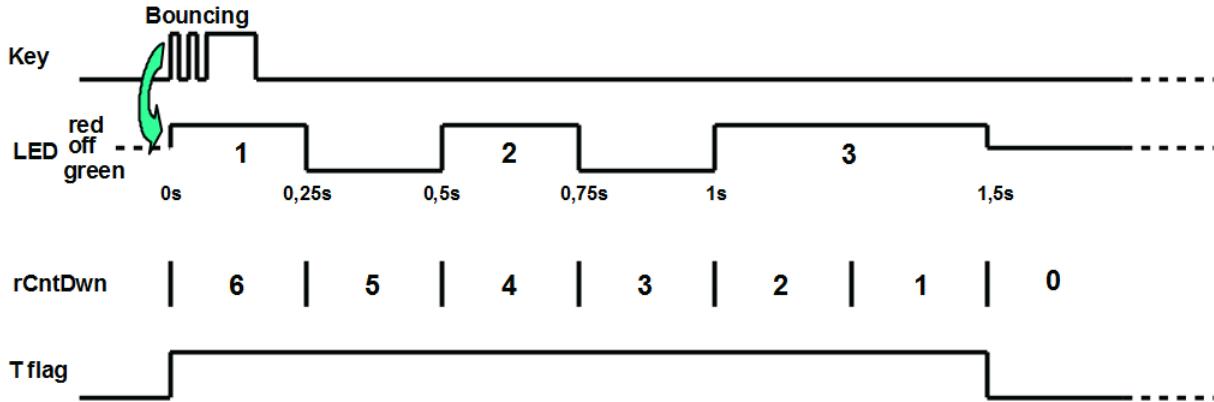
7.4 Program

7.4.1 Processes and flows

To perform the task a clear flow is necessary. In this case this flow is obviously like this:

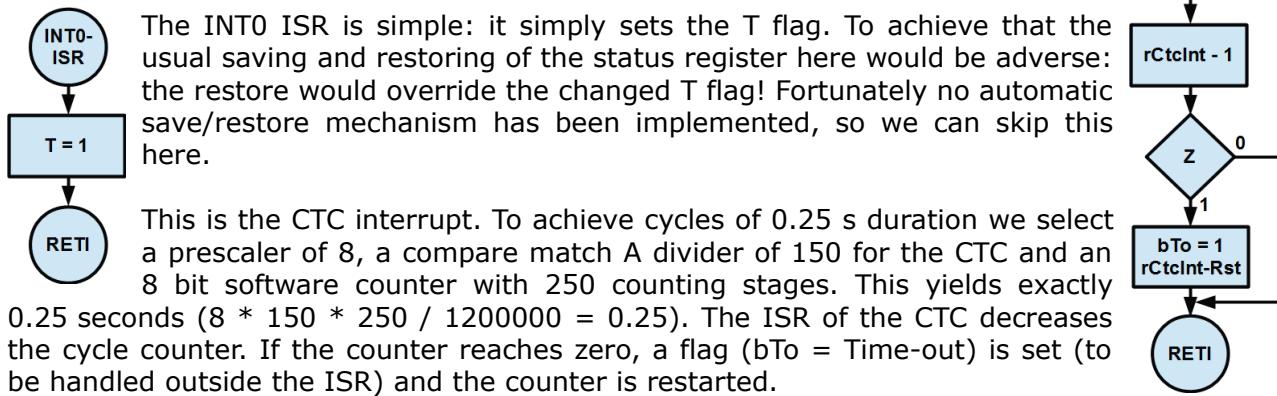
This is a counting scheme. The counter counts down from six to zero, at the different values the on and off periods of the LED and its colors can be derived for the six phases. The count step duration is 0.25 seconds, which has to be derived from a timer overflow or CTC/compare match A interrupt. The counter starts with six and a red LED. Also, in phases 4 and 2 the LED

has to be switched to red, in phases 5 and 3 to green.



If the counter reaches zero, the cycle ends and can be restarted by the key. To ensure that the process is only restarted if the cycle has ended we need a flag that signals an active cycle. We use the T flag for that purpose. This also ensures that bouncing has no effect. The T flag is bit 6 of the status register, it can be used freely as it is not affected by any other instruction. To set this flag the instruction SET can be used, to clear it CLT is available. The instructions BRTC and BRTS can be used to branch if the T bit is clear or set. The instruction BST Register, Bit can be used to copy a bit in the register to the T flag, BLD Register, Bit to copy the T flag to the bit in the register.

7.4.2 Flow diagrams



The main flow diagram shows what has to done outside the two ISRs, after init and after waking up from the sleep phase. The first task is to detect if the counter is at zero. If yes then the T flag is checked. If set, a new cycle starts and the counter is set to six. If the cycle counter is not zero, the bTo flag is checked (which is set by the CTC ISR when a time-out occurs). If this is set, the next stage is handled. In any case the flow returns to the sleep instruction.

The start sequence does the following with the timer:

- the CTC compare match A value of 150 is written,
- the CTC mode is written to control register A,
- with a prescaler of 8 in control register B the timer is started, and
- the compare match A interrupt is enabled.

The LED is switched to red and the cycle counter is started with 6.

7.4.3 The Program

This is the program, the [source code is here](#):

```

;
; ****
; * Key with INT0 interrupt *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;

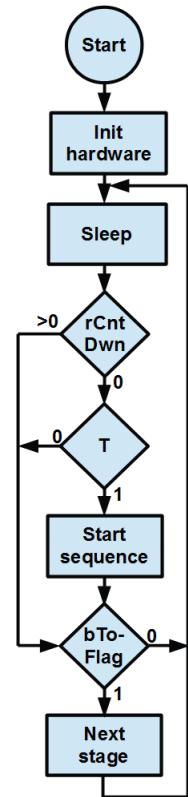
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;

; ----- Registers -----
; free R0 .. R14
.def rSreg = R15 ; save and restore status register
.def rmp = R16 ; multi purpose register
.def rimp = R17 ; multi purpose inside interrupts
.def rFlag = R18 ; Flag register
    .equ bTo = 0 ; Timeout flag timer
.def rCntDwn = R19 ; Cycle count down register
.def rCtcCnt = R20 ; CTC count down
; free R21 .. R31
;

; ----- Ports -----
.equ pOut = PORTB ; Output port
.equ pDir = DDRB ; Direction port
.equ pIn = PINB ; Input port
.equ bA0 = PORTB2 ; Anode LED red output pin
.equ bC0 = PORTB0 ; Cathode LED red output pin
.equ bP0 = PORTB1 ; Pull-Up key output pin
.equ bA1 = DDB2 ; Anode LED red direction pin
.equ bC1 = DDB0 ; Cathode LED red direction
;

; ----- Timing -----
; Clock      = 1200000 cs/s
; Prescaler   = 8
; CTC divider = 150
; Counter     = 250
;
; -----
; Signal duration = 0.250 seconds
;
; ----- Konstanten -----
.equ cCtcCmp = 149 ; CTC divider - 1
.equ cCtcInt = 250 ; CTC Int counter
;
; ----- Reset- and Int vectors -----
.CSEG ; Assemble to program flash (Code Segment)
.ORG 0 ; Address to zero (Reset- and int vectors start at zero)
    rjmp Start ; Reset vector, jump to init
    rjmp Int0_Isr ; INT0-Int, active
    reti ; PCINT-Int, inactive

```



```

reti ; TIM0_OVF, inactive
reti ; EE_RDY-Int, inactive
reti ; ANA_COMP-Int, inactive
rjmp Tc0CmpA ; TIM0_COMPA-Int, active
reti ; TIM0_COMPB-Int, inactive
reti ; WDT-Int, inactive
reti ; ADC-Int, inactive
;
; Interrupt Service Routines
;
Int0_Isr: ; INT0-ISR
    set ; set T flag
    reti ; return, set I flag
;
Tc0CmpA: ; TCO Compare A ISR
    in rSreg,SREG ; save SREG
    dec rCtcCnt ; decrease CTC counter
    brne Tc0CmpA1 ; not yet zero
    sbr rFlag,1<<bTo ; set time out flag
    ldi rCtcCnt,cCtcInt ; restart counter
Tc0CmpA1:
    out SREG,rSreg ; restore SREG
    reti ; return, set I flag
;
; Program start at reset
;
Start:
    ; Stack Init
    ldi rmp,LOW(RAMEND)
    out SPL,rmp;
    ; LED outputs init
    ldi rmp,(1<<bARD)|(1<<bCRD) ; outputs
    out pDir,rmp ; to direction port
    ; Pull-Up resistor on
    sbi pOut,bPuO ; Pull-up on
    ; Start conditions counter
    clr rCntDwn ; Countdown counter off
    clt ; Busy flag off
    ; Sleep enable, external Int0
    ldi rmp,(1<<SE)|(1<<ISCO1) ; Sleep, INT0 falling edge
    out MCUCR,rmp ; to universal control port
    ; INT0 enable
    ldi rmp,1<<INT0 ; INT0 interrupts on
    out GIMSK,rmp ; to General Interrupt Mask
    ; Enable Interrupts
    sei ; set interrupt flag
Loop: ; main program loop
    sleep ; go to sleep
    nop ; after wake-up
    tst rCntDwn ; Countdown count = zero?
    brne Loop1 ; no, do not start
    brtc Loop1 ; T flag=0, do not start
    rcall StartSeq ; start sequence
Loop1:
    sbrc rFlag,bTo ; Jump, if CTC flag off
    rcall Countdown ; count down-ward
    rjmp Loop
;
; Start sequence
;
StartSeq:
    ldi rCntDwn,6 ; start value first cycle
    ldi rmp,(1<<bPuO)|(1<<bARO) ; red LED and pull-up
    out pOut,rmp ; to output port
    ldi rCtcCnt,cCtcInt ; Restart CTC Int counter
    cbr rFlag,1<<bTo ; clear time out flag
    ldi rmp,cCtcCmp ; CTC compare match value
    out OCR0A,rmp ; to compare port A
    ldi rmp,1<<WGM01 ; TCO as CTC with compare A
    out TCCR0A,rmp ; to control port A
    ldi rmp,1<<CS01 ; Prescaler = 8
    out TCCR0B,rmp ; to control port B
    ldi rmp,1<<OCIE0A ; Compare-A-Interrupt
    out TIMSK0,rmp ; to TCO Int Mask
    ret ; Ready, return
;
; 250 ms over, next phase

```

```

;
Countdown:
    cbr rFlag,1<<bTo ; clear bTo flag
    dec rCntDwn ; next phase down
    breq CountDownOff ; reached zero, end of cycle
    cpi rCntDwn,5 ; cycle = 5?
    breq CountDownGreen ; yes, LED to green
    cpi rCntDwn,3 ; cycle = 3?
    breq CountDownGreen ; yes, LED to green
    ; Switch LED to red
    ldi rmp,(1<<bPu0) | (1<<bARO) ; Red and pull-up
    out pOut,rmp ; to output port
    ret
CountDownGreen: ; LED to green
    ldi rmp,(1<<bPu0) | (1<<bCRO) ; Green and pull-up
    out pOut,rmp ; to output port
    ret ; return
CountDownOff: ; End of cycle, switch all off
    clr rmp ; timer off
    out TCCR0B,rmp ; to control register B
    out TIMSK0,rmp ; timer int off
    ldi rmp,1<<bPu0 ; all clear but Pull-up
    out pOut,rmp ; LED off
    clt ; T flag off
    ret ; return
;
; End of source code
;

```

Two instructions are new:

- CLR Register: clears register and sets the Z flag,
- TST Register: tests if register is clear (equivalent to OR Register,Register).

7.4.4 Simulating the processes

(Help)	7	6	5	4	3	2	1	0
PORPB	0	0	0	0	0	0	1	0
DDRB	0	0	0	0	0	1	0	1
PINB	0	0	0	0	0	Lo	Pu	Lo
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

Previous **B** Next

To simulate all these mechanisms we feed the source code to [avr_sim](#) and step through the sequences.

That is the status of port B after init:

- The two pin outputs PB0 and PB2 are defined as outputs, their direction bits are set and their driver is switched on. As they both are low, the Duo-LED between PB0 and PB2 is off.
- On PB1, where the key is attached to, the direction bit is zero and the PORT bit is one, which switches the internal pull-up resistor on. If we would read the PIN1 bit, it would be high.
- On PB1 the INT0 input pin is located. The software has enabled the respective interrupt enable bit on falling edges on this input, which will happen if the key will be closed.

Now the controller goes to sleep mode idle and port B waits for signals on PB1.

By clicking on the INT0 we initiate an INT0 interrupt request. If no other interrupts are executed, the INT0 interrupt will be processed with the next instruction. This would be the case even if another interrupt would be pending because INT0 is the interrupt with the highest priority (highest position in the interrupt vector list).

(Help)	7	6	5	4	3	2	1	0
PORPB	0	0	0	0	1	0	1	0
DDRB	0	0	0	0	0	1	0	1
PINB	0	0	1	1	Pu	Lo	Pu	Lo
TINnB	0	0				0		
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

Previous **B** Next

Port

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	1	0	1	0
DDRB	0	0	0	0	0	1	0	1
PINB	0	0	1	1	Pu	Lo	Pu	Lo
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0

Previous **B** Next

The controller woke up from sleep mode idle, stored the current execution address to the stack and jumped to the INT0 vector address. From there he jumps to the interrupt service routine.

Simulation

Prog counter = \$00000B
Instructions = 15
Stackpointer = \$009D
Clock frequ. = 1,200,000 Hz

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

The only task in the INT0 interrupt service routine is to set the T flag in SREG - to start a new sequence.

Simulation

Prog counter = \$000033
Instructions = 32
Stackpointer = \$009D
Clock frequ. = 1,200,000 Hz
Time elapsed = 0.00 ns
Stop watch = 0.00 ns

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

Register

Reg	+0	+1	+2	+3	+4
R0	00	00	00	00	00
R8	00	00	00	00	00
R16	02	00	00	06	FA
R24	00	00	00	00	00

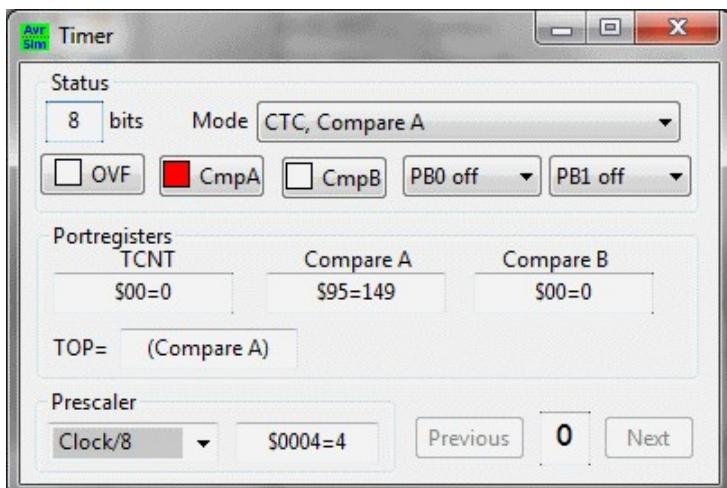
After waking up, the controller realizes that the T flag has been set and branches with an RCALL to the StartSeq: section. By doing that, the stackpointer is decreased by two to store the calling address on the stack.

The StartSeq: has set the down-counter in R19 to 6 to start a down-counting sequence. The CTC counter in R20 has been initiated to 250 to execute 250 timer CTC cycles. Note that the T flag in SREG is still set: it will only be cleared after the whole sequence of six phases has been absolved. Further INTO interrupts therefore do nothing.

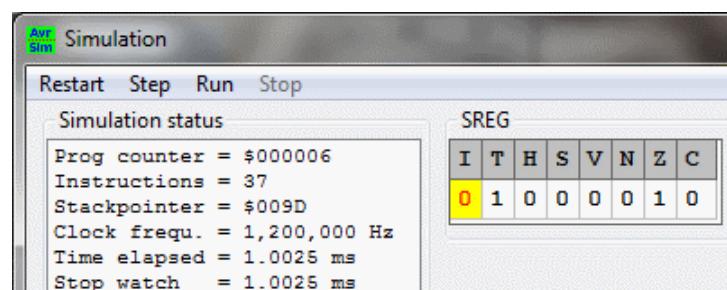
The StartSeq: has brought the timer TC0 to count in CTC mode. The TOP value of the counter is 149, so the counter restarts after 150 pulses. The controller clock is divided by 8 in the prescaler, so that the whole CTC cycle lasts $8 * 150 / 1,200,000 = 1.00 \text{ ms}$.

Timer

Status	8 bits	Mode	CTC, Compare A
<input type="checkbox"/> OFV	<input checked="" type="checkbox"/> CmpA	<input type="checkbox"/> CmpB	PB0 off
Port registers	TCNT	Compare A	Compare B
\$00=0	\$95=149	\$00=0	
TOP=	(Compare A)		
Prescaler	Clock/8	\$0001=1	Previous 0 Next

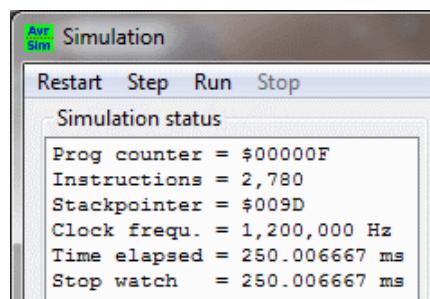


After TC0 reached 150 (and restarted on compare match A) he initiates a compare match interrupt and the controller jumps to the compare match A interrupt vector.

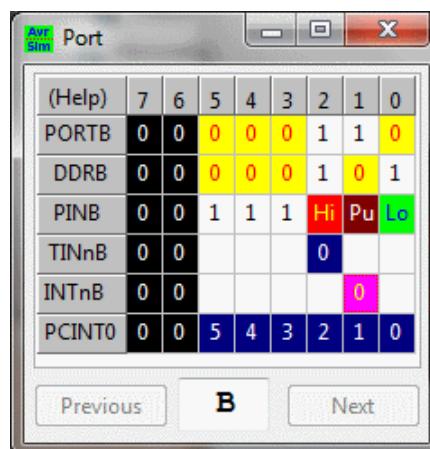


As calculated the first compare match interrupt happens after 1.0025 ms have elapsed. The "overtime" of 0.0025 ms is due to the fact that the compare match had to store the execution address to the stack, to clear the I flag in SREG and to jump to the vector address. If you need more accuracy in your application consider these time delays. In our application

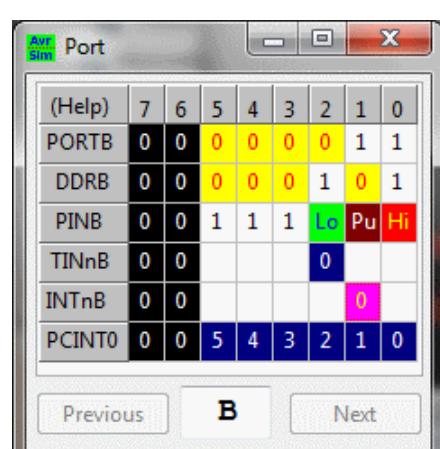
that makes no sense as the delay is always the same and as the human eye is not that fast and accurate.



250 CTC cycles have been absolved here. The bTO flag will be set now to signal the timeout. Execution time since timer start is rather accurate.



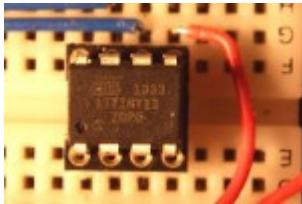
Now that the bTO flag is set, it is time to change the color of the duo LED according to the phase diagram from red to green. The portbits PB0 and PB2 change here.



The further process repeats all this for five times, then clears the T flag and so enables restarting.

This is the end of the cycle after 1.5 seconds.

[Home](#) [Top](#) [Introduction](#) [Task](#) [Hardware](#) [Program](#)



Lecture 8: Regulation of the intensity of a LED

In this lecture we apply the built-in Analogue to Digital Converter (ADC).

8.0 Overview

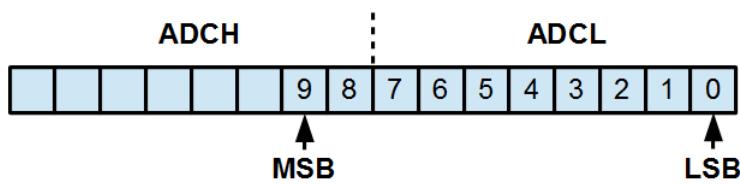
1. [Introduction to AD conversion](#)
2. [Introduction to PCINT interrupts](#)
3. [Hardware, components, mounting](#)
4. [Intensity regulation](#)
5. [Intensity regulation with color change](#)
6. [Intensity regulation dynamically](#)
7. [Red/green](#)

8.1 Introduction to AD conversion

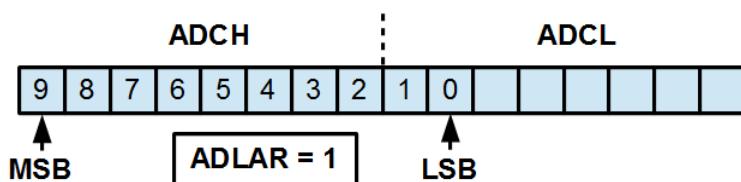
AD converters do the following:

- They store a voltage in a capacitor and then decouple this from the source ("Sample-and-Hold").
- Then this voltage is compared with half the reference voltage. If the stored voltage is smaller, the most significant result bit is low, otherwise high. If high, the comparer voltage remains at half the reference voltage otherwise the compare voltage is zero.
- Then the compare voltage is increased by one quarter of the reference voltage. If the stored voltage is larger than this compare voltage, the next result bit is low, otherwise high. The compare voltage is set depending from this result.
- This goes on with one eights, one sixteenth, one thirty-second, etc. of the reference voltage, until all bits have been collected.

The AD converter in the ATtiny13 has a resolution of 10 bits, so at a reference voltage of 5.0 V voltage differences of $5 / 2^{10} = 4.8 \text{ mV}$ can be measured, approximately one tenth of a percent. The result of a measurement can be converted to a resulting voltage by multiplying the result with 1,024 and dividing the result through the reference voltage. As reference voltage in the ATtiny13 either the operating voltage (bit REFS0 in control port = 0) or an internally generated reference of 1.1 V can be selected (REFS0 = 1). Other AVR offer the opportunity to source the reference voltage from an external source via a port pin.



After conversion has been completed the 10 bit result is in the ports ADCL (the lower 8 bits) and ADCH (the upper two bits, all other bits at zero) and can be read from there to a register with the "IN Register,Port" instruction.



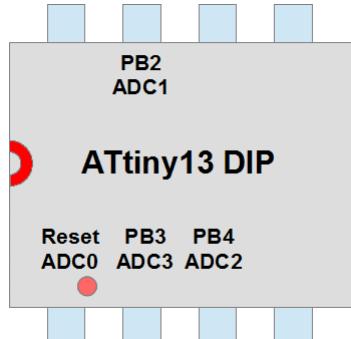
With the bit ADLAR (AD left adjust result) = set this behavior can be changed: in this case the result bits are stored differently. The eight most significant bytes can be read from ADCH. If 8 bit resolution are suffi-

client, ADCL must not be read and processed.

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	REFS0	ADLAR	-	-	-	MUX1	MUX0	ADMUX
Initial Value	0	0	0	0	0	0	0	0	

In the port ADMUX the two bits REFS0 and ADLAR are located. Additionally the bits MUX1 and MUX0 select

the pin from which the input voltage is copied in the sample-and-hold phase is taken.



Those are the ADC pins that can be connected to the internal AD converter. Their numbering has not much in common with the port pin numbering. Note that the RESET pin (ADC0) can only be used as linear input if the RESET function of this pin is disabled by a dedicated fuse. If RESET is disabled, no ISP programming of the ATtiny13 can be made any more. So do that only in case you either own a programmer capable to program in high-voltage mode (e.g. an STK500) or if you are absolutely sure you will never have to re-program the chip because your software is absolutely perfect and the chip will never be used in a different system or for a different purpose.

The bits MUX1 and MUX0 determine which of the sources will feed the sample-and-hold. 00 is ADC0, 01 is ADC1, 11 is ADC3.

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	-	ADC0D	ADC2D	ADC3D	ADC1D	AIN1D	AIN0D	DIDR0
Initial Value	0	0	0	0	0	0	0	0	

The pin or the pins that are used as AD input pins will not be used as digital inputs any more. To

reduce current needs those stages can be disabled. Disabling is performed if the respective bits in DIDR0 are set. Please note the strange numbering of those bits. One more argument to use the 1<<bit construction instead of absolute bit values.

Bit	7	6	5	4	3	2	1	0	
Read/Write	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Initial Value	0	0	0	0	0	0	0	0	

The complete control over the ADC is via the port ADCSRA. In that port the bits ADPS2, ADPS1 and ADPS0

refer to a prescaler for clocking the ADC operation (000 = 2, 111 = 128). Per conversion 13 prescaled clock signals are required (a few more if the ADC was not active before). The bit ADIE enables the generation of an ADC interrupt, if set. ADATE allows to Auto-Start the conversion on certain conditions. Otherwise conversion has to be started via an OUT to ADCSRA with the ADSC bit set (AD start conversion). If set, this bit remains set until the conversion is completed. The bit ADEN switches the ADC on and off.

A typical sequence to start the ADC with ADC3 as voltage source is as follows:

```
ldi rmp, (1<<MUX1) | (1<<MUX0) ; Channel 3
out ADMUX,rmp ; to AD multiplexer port
; switch ADC on, start conversion, interrupt when complete, prescaler 128
ldi rmp, (1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; to ADC control port
```

That were the most relevant things to know about the ADC and its activation.

One important additional hint: if you read ADCL the AD converter is blocked and does not restart until ADCH is read. So a false row of reading blocks the functioning of the AD converter. Always read ADCL first and then do not forget to read ADCH as well.

8.2 Introduction to the PCINT programming

In the previous lecture a key on the INT0 input pin was used to detect external events. If the input pin INT0 is blocked by other needs or if additional keys must be attached and detected, the external interrupt PCINT can be used. This works a little bit different than INT0.

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	-	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK
Initial Value	R	R	R/W	R/W	R/W	R/W	R/W	R/W	

All digital input pins can be monitored by the PCINT. If an input pin's level changes shall lead to a

PCINT just set its bit in the mask register PCMSK. If one or more of these bits are set, software has to be used to determine which of the pins has caused the PCINT. If only one of the PCMSK bits is set, this task is simple. In contrast to INT0 no preselection on rising or falling edges can be made, any changes lead to a PCINT.

Bit	7	6	5	4	3	2	1	0	
Read/Write	-	INT0	PCIE	-	-	-	-	-	GIMSK
Initial Value	R	R/W	R/W	R	R	R	R	R	

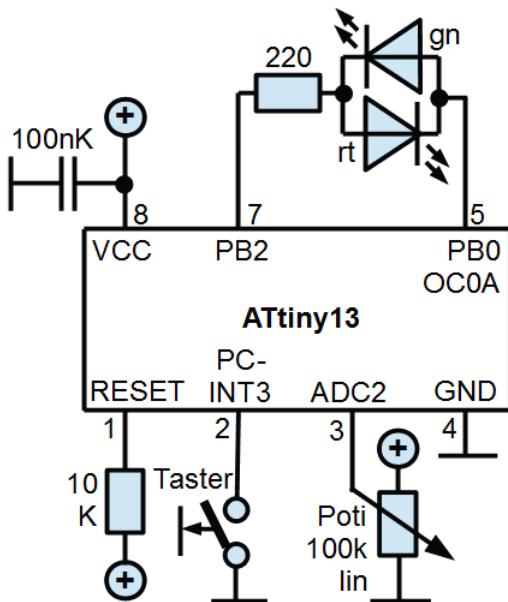
The PCINT has to be enabled in the General Interrupt Mask port GIMSK by setting the bit PCIE.

That is it to use monitoring of any port pin and to react with an interrupt.

[Home](#) [Top](#) [AD conversion](#) [PCINT](#) [Intensity](#) [Color selection](#) [Dynamic](#) [Red/Green](#)

8.3 Hardware, components and mounting

8.3.1 Hardware scheme



The duo-LED is installed like in the previous lecture. The key is on pin 2, which can trigger PCINT3. The potentiometer is connected to ADC2 on pin 3 and divides the operating voltage. The ISP connections remain the same as in all previous lectures. Those connections are not shown here.

8.3.2 Components

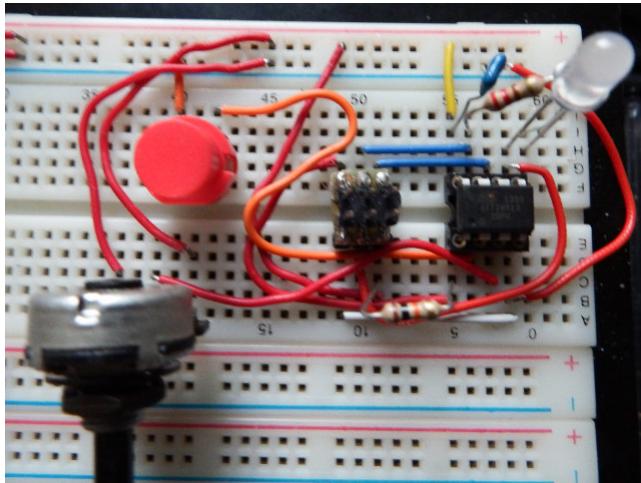
The potentiometer

This is the potentiometer, the middle pin is the slider. All three wire pins are too large to fit into breadboard holes, so short wires



are soldered to these pins.

8.3.3 Mounting



This is an example for mounting. The placement of components is not critical.

[Home](#) [Top](#) [AD conversion](#) [PCINT](#) [Intensity](#) [Color selection](#) [Dynamic](#) [Red/Green](#)

8.4 Intensity regulation

8.4.1 Task 1

In the first task the intensity of the red LED shall be regulated with the potentiometer. The intensity should increase with the increasing angle of the potentiometer (at increased voltages).

8.4.2 Solution 1

It is clear from the task description that

- the timer mode will be Fast PWM,
- the voltage has to be measured continuously,
- the result of the ADC measurement is between 0 and 1023, that has to be converted to the PWM comparer range of 0 to 255,
- the PWM comparer has to be written with this converted value.

To convert 0 to 1023 to the PWM range 0 to 255 it has to be divided by four. This division is one new software skill to be learned here.

We do not necessarily need interrupts here, because all we have to do is

- to start the ADC (by setting the ADEN and the ADSC bit),
- to wait until the AD conversion is completed (the ADSC bit clears),
- to divide the AD result by four and write that to the comparer.

Thanks to the automatic timer operation we do not have to control more. If we program this in linear mode the controller waits for more than 95% of its time for the AD conversion complete condition. This is not very intelligent, it is not very aesthetic and wastes current. As the following tasks require the AD conversion complete interrupt anyway, we use this interrupt controlled technology already here.

8.4.3 Program 1

The program is available [here as source code file](#).

```

;
; **** LED intensity regulator with poti and ADC ****
; * LED intensity regulator with poti and ADC *
; * (C)2017 by www.avr-asm-tutorial.net      *
; ****
;

.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;

; ----- Registers -----
; free: R0 .. R12
.def rAdcL = R13 ; LSB ADC result
.def rAdcH = R14 ; MSB dto.
.def rSreg = R15 ; SREG save/restore register
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside ints
; free: R18 ... R31
;

; ----- Ports -----
.equ pOut = PORTB ; LED output port
.equ pDir = DDRB ; Direction port
.equ bRAO = PORTB2 ; Output pin anode red LED
.equ bRAD = DDB2 ; Direction pin anode red LED
.equ bRKO = PORTB0 ; Output pin cathode red LED
.equ bRKD = DDB0 ; Direction pin cathode red LED
;

; ----- Timing -----
; Controller clock = 1.200.000 cs/s
; ADC prescaler = 128
; ADC cycles = 13
; Measure frequency = 721 cs/s
; TCO prescaler = 64
; PWM resolution = 256
; PWM frequency = 73 cs/s
;

; ----- Reset- and interrupt vectors ---
.CSEG ; Assemble to flash storage (Code Segment)
.ORG 0 ; Adresse to zero (Reset- and interrupt vectors start at zero)
    rjmp Start ; Reset Vector, jump to init
    reti ; INT0-Int, inactive
    reti ; PCINT-Int, inactive
    reti ; TIM0_OVF, inactive
    reti ; EE_RDY-Int, inactive
    reti ; ANA_COMP-Int, inactive
    reti ; TIM0_COMPA-Int, inactive
    reti ; TIM0_COMPB-Int, inactive
    reti ; WDT-Int, inactive
    rjmp AdcIsr ; ADC-Int, active
;

; Interrupt service routines
;

AdcIsr: ; Interrupt Service Routine ADC conversion complete
    in rSreg,SREG ; save SREG
    ; Read ADC result
    in rAdcL,ADCL ; Read ADC result, LSB
    in rAdcH,ADCH ; dto., MSB
    ; Divide result by 4
    lsr rAdcH ; shift MSB right, bit 0 to carry flag
    ror rAdcL ; rotate LSB right, bit 7 from carry flag
    lsr rAdcH ; shift MSB right, bit 0 to carry
    ror rAdcL ; rotate LSB right, bit 7 from carry flag
    ; Set new PWM compare value
    out OCR0A,rAdcL ; to timer Compare port A
    ; Restart ADC (prescaler 128, interrupt enable)
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; to ADC control port
    out SREG,rSreg ; Restore SREG
    reti ; return from interrupt, set I flag
;

; ----- Main program init -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Stack pointer to RAMEND
    out SPL,rmp ; to stack pointer port
    ; Configure LED pins and activate

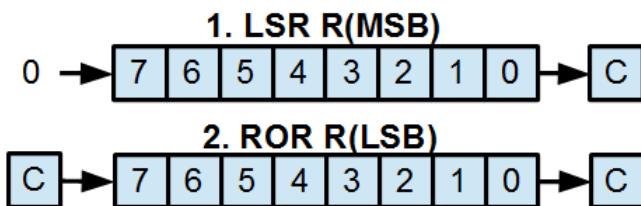
```

```

ldi rmp,(1<<bRAD) | (1<<bRKD) ; Port pins LED to output
out pDir,rmp ; to direction port
ldi rmp,(1<<bRAO) | (1<<bRKO) ; Anode and cathode to high
out pOut,rmp ; to output port
; Timer as 8 bit PWM
ldi rmp,0x80 ; half intensity
out OCR0A,rmp ; to timer compare port A
; Timer Fast PWM, clear PB0 at top, set on compare match
ldi rmp,(1<<COM0A1) | (1<<COM0A0) | (1<<WGM01) | (1<<WGM00)
out TCCR0A,rmp ; to control port A
ldi rmp,(1<<CS01) | (1<<CS00) ; Prescaler 64
out TCCR0B,rmp ; to control port B
; Configure ADC
ldi rmp,1<<MUX1 ; ADC signal input = ADC2
out ADMUX,rmp ; to ADC multiplexer port
ldi rmp,1<<ADC2D ; Disable port driver hardware ADC2
out DIDR0,rmp ; to disable port
; Start ADC, enable interrupt, prescaler = 128
ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; to ADC control port
; Sleep mode idle
ldi rmp,1<<SE ; Enable sleep
out MCUCR,rmp ; to general control port
; Interrupts
sei ; enable interrupts
; Program loop
Loop:
    sleep ; put to sleep
    nop ; after wakeup by interrupt
    rjmp Loop ; back to sleep
;
; End of source code
;

```

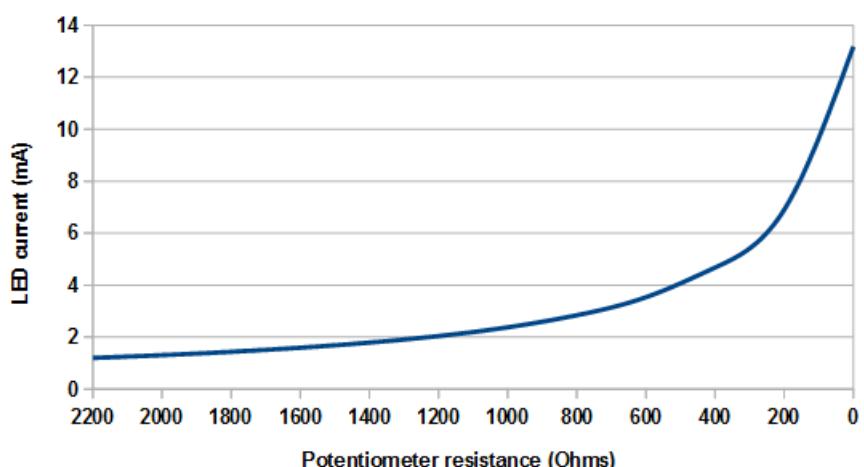
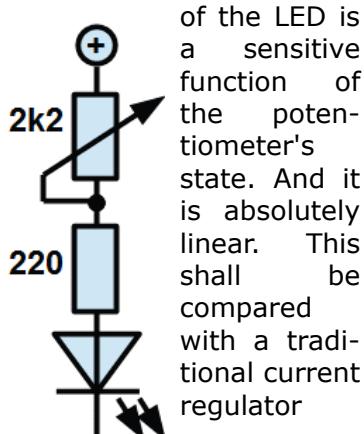
New are the instructions LSR and ROR. We have to look at this closer, because it is used very often when manipulating 16 bit numbers. The instructions LSL and ROL act similar, but shift and rotate to the left.



The task performed here is to shift the two lower bits in the ADCH result to the LSB read from ADCL. This double right shift is a binary division by four. The operation in the picture is performed twice. First, the MSB is shifted to the right. A zero bit is shifted into bit 7, while bit 0 is shifted to the carry flag in SREG. In the second instruction this carry bit is rotated into bit 7 of the LSB, while bit 0 is rotated to the carry flag.

The ATtiny13 does not have a 16 bit shift instruction, so the two instructions LSL and ROR perform this. Repeating LSR and ROR twice the two lowest bit in the LSB are now bit 7 and 6 of the LSB, while the two lowest bits in the LSB are scrapped.

The result of the experiment is that the intensity of the LED is a sensitive function of the potentiometer's state. And it is absolutely linear. This shall be compared with a traditional current regulator



with a potentiometer.

Left is a simple current regulator, to the right the current through the LED in different stages of the potentiometer. It is obvious that the PWM regulator is a much better solution.



8.4.4 Simulating program execution

The following simulates program execution using the simulator [avr_sim](#).

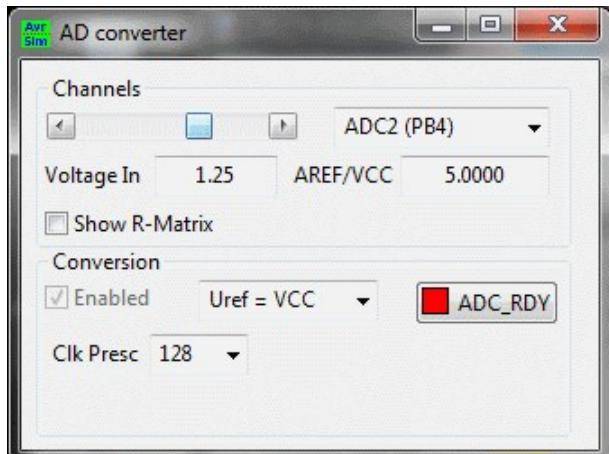
The program init phase has been executed. 18.3 μ s have elapsed.

	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	1	0	1
PINB	0	0	0	0	0	Lo	0	Lo
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

The timer TC0 is initiated in fast PWM mode, with overflow on TOP (255) and sets the I/O pin PB0 on compare match A, which is set to 128 (half intensity).

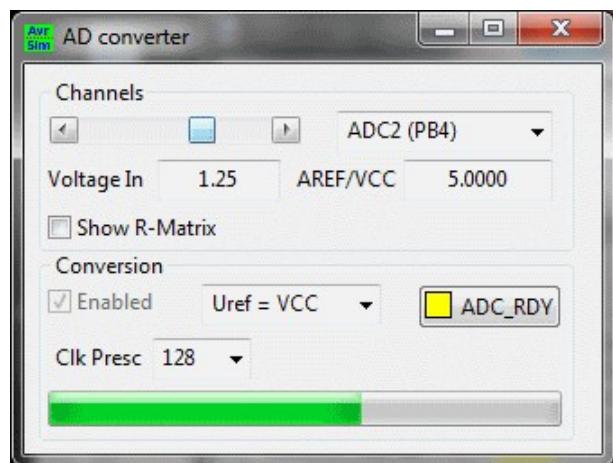
The AD converter channel ADC2 has been set to work with a reference voltage of 5 V (operating voltage), with 1.25 V on the input. Conversion is clocked by the processor clock divided by 128. The yellow field on the interrupt section signals that the conversion complete interrupt is enabled and execution will result in a jump to the ADC vector.

The AD conversion has started, conversion progress is displayed on a progress bar.

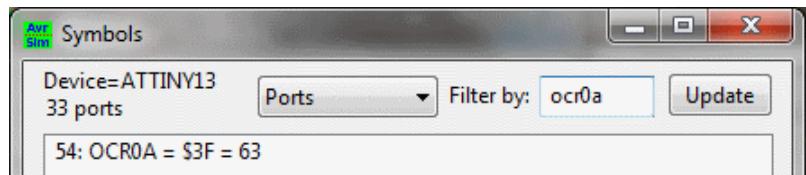
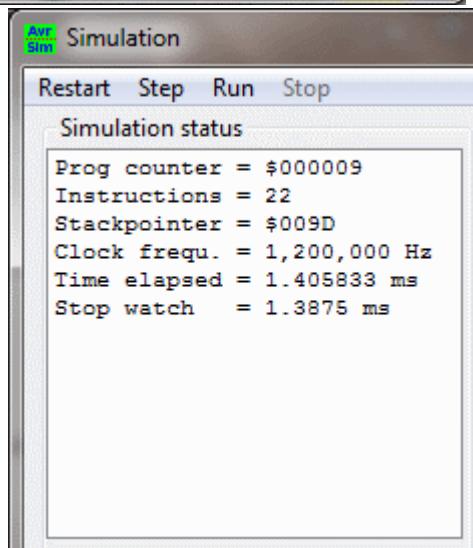


executed if no other higher-leveled int is requested.

On int 1.38 ms have elapsed since the init phase was completed. Conversion is clocked with $1,200,000 / 128 = 9,375$ Hz or 0.107 ms per conversion clock. 13 conversion steps are required, so 1.38 ms is correct.



On conversion complete the respective interrupt flag is set and the interrupt is



The conversion result,

$$1023 * 1.25 / 5.0 = 255$$

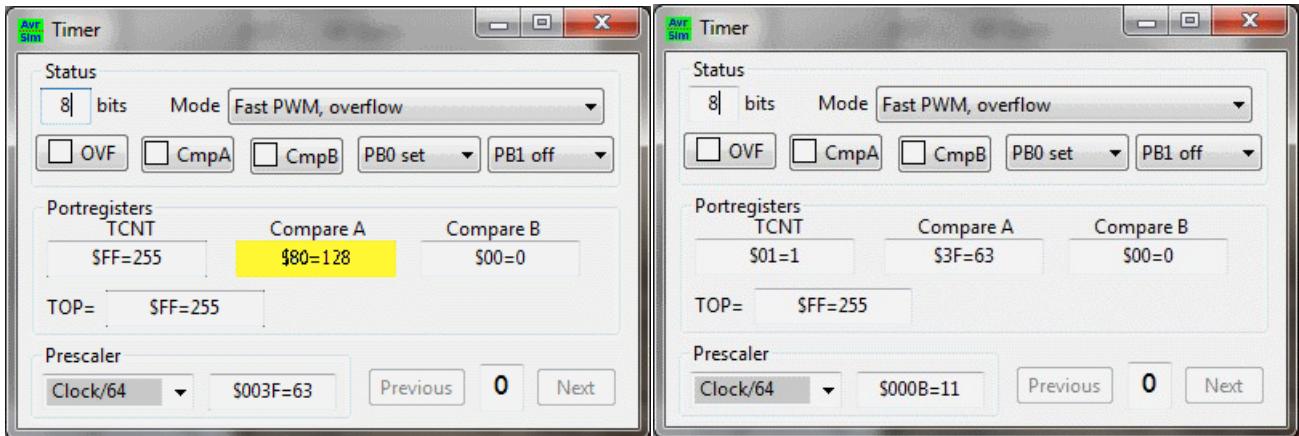
is divided by four (=63) and written to the output compare register OCR0A of the timer TC0, controlling the pulse width of the duo LED.

But this change to the OCR0A port register does not come into effect immediately. If it would so, the compare match would not be correct: if OCR0A would be smaller than the current count of TC0 the compare match would be missed in this cycle and the I/O pin would remain low for the whole cycle.

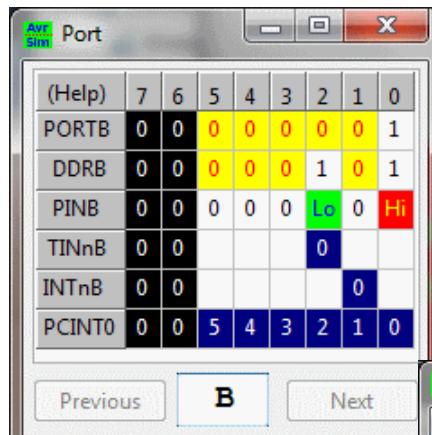
Table 11-8. Waveform Generation Mode Bit Description

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on ⁽¹⁾⁽²⁾
3	0	1	1	Fast PWM	0xFF	TOP	MAX

So it is a good idea to change the effective compare match value only after reaching TOP.



TOP was reached and the compare match value now has been updated.

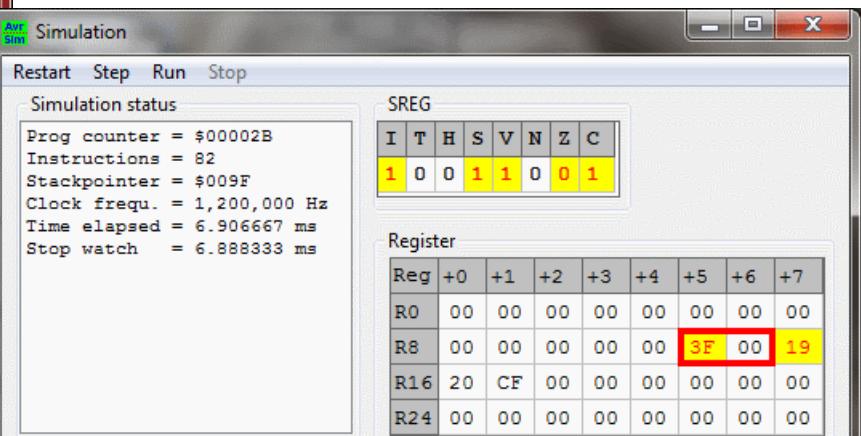


Here the compare match value of 128 has been reached for the first time. I/O pin PB0 is set.

6.89 ms have elapsed since init until the first compare match occurred. This corresponds to

$$64 * (128 + 1) / 1,200,000 \text{ s}$$

and is correct. The whole cycle will last for $64*256/1,200,000$ = 13.65 ms or a PWM frequency of 73.24 Hz.



The registers R14:R13 hold the conversion result, divided by 4.

Simulation offers the opportunity to test and verify source code. Bugs can be detected and corrected. Play around with your source code in this way and see if your software works as designed.



8.5 Intensity regulation with color change

8.5.1 Task 2

Now not only the red but also the green LED shall be regulated. The key shall be used to select the red and green LED. In all cases a higher voltage shall lead to higher intensity.

8.5.2 Debouncing

Switching of the color with the key from red to green and back is simple: if PORTB2 (or PINB2) is one, PORTB2 is switched to zero, and vice versa. This changes the polarity of the LED base reference and its color, while the PWM output on PB0 delivers the pulses.

Fatal in this application is the bouncing of the key. Depending from the number of swarm pulses from the key red or green results. A not very reliable switch, so we need a mechanic that suppresses bounces.

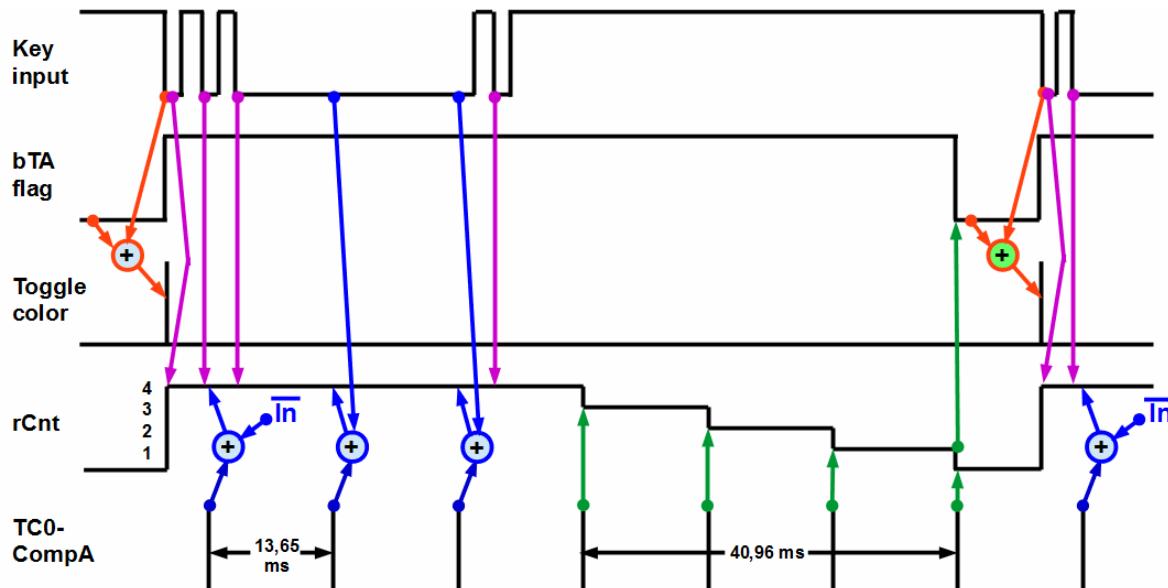
For that we need the following:

- a flag that signals that further key input signals do not lead to actions, and
- this flag is only cleared if the key is not pressed for a certain period of time.

To measure this duration we could again use counting loops, but this would be below our skill level. Two sources for clocking purposes are available:

- the ADC interrupt occurs with 721 cs/s, each 1,39 ms,
- the TC0 produces a PWM signal with 73 cs/s frequency, the compare match would occur each 13,68 ms.

Avoiding the additional task for the AD conversion complete interrupt we use the TC0 compare



match A interrupt. Because 13 ms would be slightly too short for debouncing, we use a down-counter to ensure that at least 40 ms long no further pulse from the key comes in.

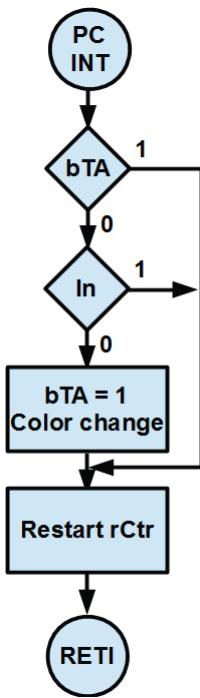
From that we get the following time relations between the key signal, the inactivity flag bTA, the color inversion and the TC0 interrupt.

If the key is pressed (key input goes low), while the flag is zero, the LED changes its color and the downcounter is set to four.

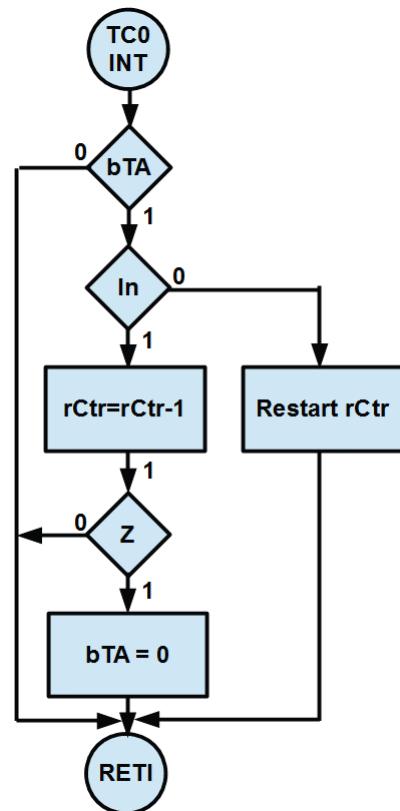
Each following low pulse on the key input resets the counter to four. This ensures that the flag is cleared only after four PWM pulses come in (40 ms delay), following the last key event.

On each PWM interrupt the bTA flag is tested. If not set, nothing further has to be done. If set, the state of the key input is deciding. If this is high, the counter is decreased. If the counter reaches zero, the flag is cleared. If the input is low (key still active) the counter is set to four again.

With that, the flow diagrams for the two interrupt service routines are as follows.



That is it. By combining these mechanics we ensure that no chaos occurs when the key is pressed.



8.5.3 Program 2

Now we use the ADLAR bit of the ADC to avoid shifting and rotating of the 10 bit ADC result. Further we use the switching of the OC0A output to display red and green correctly.

This here is the program, its [source code is here](#).

```

;
; ****
; * LED intensity regulator red/green with ADC and key *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ----- Registers -----
; free: R0 .. R14
.def rSreg = R15 ; Save/Restore SREG
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside interrupts
.def rFlag = R18 ; Flag register
    .equ bTA = 0 ; Key active flag
.def rCnt = R19 ; Counter for debounce suppression
; free: R20 .. R31
;
; ----- Ports -----
.equ pOut = PORTB ; LED output port
.equ pDir = DDRB ; LED direction port
.equ pIn = PINB ; LED read port
.equ bRAO = PORTB2 ; Output pin anode red LED
.equ bRAD = DDB2 ; Direction pin anode red LED
.equ bRAI = PINB2 ; Read pin anode red LED
.equ bRKO = PORTB0 ; Output pin cathode red LED
.equ bRKD = DDB0 ; Direction pin cathode red LED
.equ bTaO = PORTB3 ; Output pin pull-up key
.equ bTaI = PINB3 ; Input pin key
.equ bTaE = PCINT3 ; PCINT mask bit key
.equ bAdD = ADC2D ; Input driver disable ADC pin
;
; ----- Timing -----
; Controller clock = 1,200,000 cs/s
; ADC prescaler = 128

```

```

; ADC cycles      =      13
; ADC complete freq =    721 cs/s
; TCO prescaler   =      64
; PWM resolution   =     256
; PWM frequency    =     73 cs/s
; TCO int duration = 13.65 ms
;
; ----- Constants -----
.equ cClock = 1200000 ; Controller clock
.equ cPresc = 64 ; TCO prescaler
.equ cPwm = 256 ; PWM counter stages
.equ cPrell = 50 ; ms key debouncing time
.equ cFPwm = cClock/cPresc/cPwm ; frequency PWM in cs/s
; Calculation with rounding
.equ cTPwm = (1000+cFPwm/2) / cPwm ; clock duration PWM in ms
.equ cCnt = (cPrell+cTPwm/2) / cTPwm ; count pulses debounce
;
; ----- Reset- and interrupt vectors ---
.CSEG ; Assembler to flash storage (Code Segment)
.ORG 0 ; Address to zero (Reset- and interrupt vectors start at zero)
        rjmp Start ; Reset vector, jump to init
        reti ; INT0-Int, inactive
        rjmp PcIntIsr ; PCINT-Int, active
        rjmp TC0Ovflsr ; TIM0_OVF, active
        reti ; EE_RDY-Int, inactive
        reti ; ANA_COMP-Int, inactive
        reti ; TIM0_COMPA-Int, inactive
        reti ; TIM0_COMPB-Int, inactive
        reti ; WDT-Int, inactive
        rjmp AdcIsr ; ADC-Int, active
;
; Interrupt service routines, with number of clock cycles
;
PcIntIsr: ; Interrupt service routine PCINT
        in rSreg,SREG ; Save SREG, +1 = 1
        sbrc rFlag,bTA ; Jump over if blocking flag active, +1/2 = 2/3
        rjmp PcIntIsrSet ; Restart counter, +2 = 4
        sbic pIn,bTaI ; Jump over if key=low, +1/2 = 4/5
        rjmp PcIntIsrSet ; Restart counter, +2 = 6
        sbr rFlag,1<<bTA ; Set flag, +1 = 6
        sbic pIn,bRAI ; Jump over if anode red is low, +1/2 = 7/8
        rjmp PcIntIsrGreen ; Jump to switch LED green, +2 = 9
        ; Switch LED to red
        sbi pOut,bRAO ; Set LED anode red output pin, +2 = 10
        ldi rimp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|(1<<WGM00) ; Fast PWM/clear on match, +1 = 11
        out TCCROA,rimp ; to timer control port A, +1 = 12
        rjmp PcIntIsrSet ; Restart counter, +2 = 14
PcIntIsrGreen: ; Switch LED to green
        cbi pOut,bRAO ; Clear LED anode pin, + 2 = 11
        ldi rimp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00) ; Fast PWM/set on match, +1=12
        out TCCROA,rimp ; to timer control port A, +1 = 13
PcIntIsrSet: ; Restart counter
        ldi rCnt,cCnt ; Load constant to counter, + 1 = 5/7/15/14
        out SREG,rSreg ; Restore SREG, +1 = 6/8/16/15
        reti ; Return from int/set I flag, + 4 = 10/12/20/19
;
TC0Ovflsr: ; Interrupt service routine TC0-Overflow
        in rSreg,SREG ; Save SREG, +1 = 1
        sbrs rFlag,bTa ; Jump over if bTA flag set, +1/2 = 2/3
        rjmp TC0OvflsrRet ; End ISR, +2 = 4
        sbis pIn,bTaI ; Jump over if key input high, +1/2 = 4/5
        rjmp TC0OvflsrNew ; Restart counter, +2 = 6
        dec rCnt ; Decrease counter, + 1 = 6
        brne TC0OvflsrRet ; Not yet zero, return, +1/2 = 7/8
        cbr rFlag,1<<bTa ; Clear bTa flag, +1 = 8
        rjmp TC0OvflsrRet ; Jump to return, +2 = 10
TC0OvflsrNew:
        ldi rCnt,cCnt ; Restart down-counter, +1 = 7
TC0OvflsrRet:
        out SREG,rSreg ; Restore SREG, +1 = 5/9/8
        reti ; Return from int/set I flag, + 4 = 9/13/12
;
AdcIsr: ; Interrupt service routine ADC conversion complete
        ; Read ADC result
        in rimp,ADCH ; Read MSB ADC result, +1 = 1
        ; Set new PWM compare value
        out OCR0A,rimp ; to compare port A, +1 = 2

```

```

; Restart ADC conversion (Prescaler 128, interrupt enable)
ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0); +1=3
out ADCSRA,rimp ; to ADC control port, +1 = 4
reti ; Return from int/set I flag, +4 = 8
;
; ----- Main program init -----
Start:
; Init stack
ldi rmp,LOW(RAMEND) ; Stack pointer to RAMEND
out SPL,rmp ; to stack port
; Configure LEDs
ldi rmp,(1<<bRAD) | (1<<bRKD) ; Port pins as output
out pDir,rmp ; to direction port
ldi rmp,(1<<bRKO) | (1<<bTaO) ; LED to red, key input pull-up
out pOut,rmp ; to port output
; Timer as 8 bit PWM
ldi rmp,0x80 ; Half intensity
out OCR0A,rmp ; to compare match port A
ldi rmp,(1<<COM0A1) | (1<<WGM01) | (1<<WGM00) ; Fast PWM, clear on compare match
out TCCR0A,rmp ; to timer control port A
ldi rmp,(1<<CS01) | (1<<CS00) ; Timer prescaler 64, start timer
out TCCR0B,rmp ; to timer control port B
ldi rmp,1<<TOIE0 ; Enable overflow int for key processing
out TIMSK0,rmp ; to timer int mask
; Configure ADC: Left adjust result, signal input = ADC2
ldi rmp,(1<<ADLAR) | (1<<MUX1) ; Set ADLAR and MUX1
out ADMUX,rmp ; in ADC multiplexer port
; Disable ADC input driver port
ldi rmp,1<<bAdD ; Disable port driver
out DIDR0,rmp ; in disable port
; Start first ADC conversion, enable ADC interrupt, prescaler = 128
ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; to ADC control port
; Enable PCINT
ldi rmp,1<<bTaE ; Key interrupts
out PCMSK,rmp ; to PCINT mask port
ldi rmp,1<<PCIE ; Enable PCINT
out GIMSK,rmp ; in general interrupt mask port
; Sleep mode idle and interrupts
ldi rmp,1<<SE ; Enable sleep mode idle
out MCUCR,rmp ; in universal control port
sei ; Enable interrupts
;
; Program loop
Loop:
    sleep ; put to sleep
    nop ; After wake-up by int
    rjmp Loop ; back to sleep
;
; End of source code
;

```

The counting of the clock cycles in the interrupt service routines yields a maximum of 20 cycles, which yields $20/1.2 = 16.7 \mu\text{s}$ duration. This is far below of the milliseconds over which key bouncing occurs and far below the PWM cycle of the timer. The routines are short enough so that we do not need to transfer code to the main program loop.

New instructions are not used.

If we imagine that we program this in linear mode with counting loops we can imagine how complicated this all would get: waiting for the ADC, monitoring the key input, counting times, inverting the color, etc. etc. Compared to this real mass interrupt programming and execution are straight forward and simple if one has understood the concept. Interrupts therefore are simple and useful.

A comment on programming this in high-level languages: even the simple PWM timing and down counting to avoid bouncing is a complicated issue in that languages as those are not really comfortable in execution timing. They are unable to really use the components of the controller to their best, as assembler offers. High level languages are too far away from the available hardware.

8.6 Intensity regulation dynamically

8.6.1 Task 3

In this task the intensity of the LED is increasing and decreasing automatically. The potentiometer determines how fast or slow this happens. The key again switches between red and green of the LED, which occurs only in case of switching from increasing to decreasing and vice versa.

8.6.2 Solution

Now the TC0 interrupt at the end of the PWM cycle not only has to control the key event but also has to determine whether the compare value has to be increased and/or decreased. To control the speed of increases/decreases the ADC value has to determine after which number of PWM cycles this increase or decrease has to happen. For that the 8 bit AD result (ADLAR) has to be divided by 16. To avoid a zero value (at which no increase or decrease would happen), we add one. This leads to between one and sixteen stages.

From my experience those cycles are relatively long because of the 256 single steps of the PWM. Therefore we increase the timer clock eight-fold (prescaler = 8 instead of 64). This leads to a visible dynamic of the LED.

Rising and falling intensity of the LED is again be performed by changing the OCR0A value. To determine whether a change in direction is necessary requires some more complicated decisions. Therefore this part is performed outside the ISR. A flag signals that the main loop has to do that part of the work.

The main loop also controls the color of the LED. The color is one bit in the flag register. This bit is inverted by a key event, the color change actually occurs only after enough PWM cycles have been processed.

All elements of the program are clear now and we can start programming source code.

8.6.3 Program 3

This is the source code, the [code for download is here](#).

```
;  
; *****  
; * LED intensity control red/green up/down *  
; * (C)2017 by www.avr-asn-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn13def.inc"  
.LIST  
;  
; ----- Registers -----  
; free: R0 .. R12  
.def rCyc = R13 ; Cycle counter  
.def rAdc = R14 ; ADC result  
.def rSreg = R15 ; Save/Restore SREG  
.def rmp = R16 ; Multi purpose register  
.def rimp = R17 ; Multi purpose inside interrupts  
.def rFlag = R18 ; Flag register  
    .equ bTA = 0 ; Key active flag  
    .equ bCy = 1 ; Task flag cycle end  
    .equ bAb = 2 ; Count down flag  
    .equ bGn = 3 ; LED green flag  
.def rCnt = R19 ; Counter for key debouncing  
; free: R20 .. R31  
;  
; ----- Ports -----  
.equ pOut = PORTB ; Output port  
.equ pDir = DDRB ; Direction port
```

```

.equ pIn = PINB ; Input port
.equ bRAO = PORTB2 ; Output pin anode red LED
.equ bRAD = DDB2 ; Direction pin anode red LED
.equ bRAI = PINB2 ; Read pin anode red LED
.equ brKO = PORTB0 ; Output pin cathode red LED
.equ bRKD = DDB0 ; Direction pin cathode red LED
.equ bTaO = PORTB3 ; Pull-up pin key
.equ bTaI = PINB3 ; Input pin key
.equ bTaE = PCINT3 ; PCINT mask bit key
.equ bAdD = ADC2D ; Input disable ADC pin
;
; ----- Timing -----
; Controller clock = 1.200.000 cs/s
; ADC prescaler = 128
; ADC cycles neces. = 13
; ADC frequency = 721 cs/s
; TCO prescaler = 8
; PWM stages = 256
; PWM frequency = 585 cs/s
; TCO int repeat = 1.7 ms
; Count time min = 1.7 ms
; Count time max = 28.9 ms
; Up/Down cycle min = 0.88 s
; max = 14.8 sec
;
; ----- Constants -----
.equ cClock = 1200000 ; Controller clock
.equ cPresc = 8 ; TCO prescaler
.equ cPwm = 256 ; PWM resolution
.equ cBounce= 50 ; ms bouncing duration key
.equ cFPwm = cClock/cPresc/cPwm ; Frequency PWM in cs/s
; Calculation with rounding
.equ cTPwm = (1000+cFPwm/2)/ cFPwm ; Clock duration PWM in ms
.equ cCnt = (cBounce+cTPwm/2) / cTPwm ; Count pulses debouncing
;
; ----- Reset- and interrupt vectors ---
.CSEG ; Assemble to the flash storage (Code Segment)
.ORG 0 ; Address to zero (Reset- and interrupt vectors start at zero)
    rjmp Start ; Reset vector, jump to init
    reti ; INT0-Int, inactive
    rjmp PcIntIsr ; PCINT-Int, active
    rjmp TC0OvfiSr ; TIM0_OVF, active
    reti ; EE_RDY-Int, inactive
    reti ; ANA_COMP-Int, inactive
    reti ; TIM0_COMPA-Int, inactive
    reti ; TIM0_COMPB-Int, inactive
    reti ; WDT-Int, inactive
    rjmp AdcIsr ; ADC-Int, active
;
; Interrupt service routines, with number of clock cycles
;
PcIntIsr: ; Interrupt service routine PCINT
    in rSreg,SREG ; Save SREG, +1 = 1
    sbrc rFlag,bTA ; Jump over if flag clear, +1/2 = 2/3
    rjmp PcIntIsrSet ; Restart counter, +2 = 4
    sbic pIn,bTaI ; Jump over if key input low, +1/2 = 4/5
    rjmp PcIntIsrSet ; Restart counter, +2 = 6
    sbr rFlag,1<<bTA ; Set bTA flag, +1 = 6
    ldi rimp,1<<bGn ; Invert Green flag, +1 = 7
    eor rFlag,rimp ; from red to green or back, +1 = 8
PcIntIsrSet:
    ldi rCnt,cCnt ; Restart counter, + 1 = 5/7/9
    out SREG,rSreg ; Restore SREG, +1 = 6/8/10
    reti ; Return from int/set I flag, + 4 = 10/12/14
;
TC0OvfiSr: ; Interrupt service routine TC0-Overflow
    in rSreg,SREG ; Save SREG, +1 = 1
    sbrs rFlag,bTa ; Jump over if bTA flag set, +1/2 = 2/3
    rjmp TC0OvfiSrDwn ; Jump to count down, +2 = 4
    sbis pIn,bTaI ; Jump over if key input high, +1/2 = 4/5
    rjmp TC0OvfiSrNew ; Restart counter, +2 = 6
    dec rCnt ; Decrease counter, + 1 = 6
    brne TC0OvfiSrDwn ; Not yet zero, count down, +1/2 = 7/8
    cbr rFlag,1<<bTa ; Clear bTa flag, +1 = 8
    rjmp TC0OvfiSrDwn ; Count down, +2 = 10
TC0OvfiSrNew:
    ldi rCnt,cCnt ; Restart down counter, +1 = 7

```

```

TC0OvfIsrDwn:
    dec rCyc ; Decrease count cycles, +1 = 5/9/11/8
    brne TC0OvfIsrRet ; Not yet zero, +1/2 = 6/7/12/13/9/10
    mov rCyc,rAdc ; Restart cycle counter, +1 = 7/13/10
    sbr rFlag,1<<bCy ; Set handling flag, +1 = 8/14/11
TC0OvfIsrRet:
    out SREG,rSreg ; Restore SREG, +1 = 8/13/11/9/15/12
    reti ; Return from int/set I flag, + 4 = 12/17/15/13/19/16
;
AdcIsr: ; Interrupt service routine ADC
    ; Read ADC result
    in rAdc,ADCH ; Read MSB ADC result, +1 = 1
    lsr rAdc ; divide by 16, +1 = 2
    lsr rAdc ; +1 = 3
    lsr rAdc ; +1 = 4
    lsr rAdc ; +1 = 5
    inc rAdc ; +1 = 6
    ; Restart ADC (Prescaler 128, Enable interrupt)
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0); +1=7
    out ADCSRA,rimp ; to ADC control port, +1 = 8
    reti ; Return from int/set I flag, +4 = 12
;
; ----- Main program init -----
Start:
    ; Stack setup
    ldi rmp,LOW(RAMEND) ; Stack pointer to RAMEND
    out SPL,rmp ; to stack port
    ; Configure LEDs
    ldi rmp,(1<<bRAD) | (1<<bRKD) ; Port pins to output
    out pDir,rmp ; to direction port
    ldi rmp,(1<<bRKO) | (1<<bTaO) ; LED red, key pull-up on
    out pOut,rmp ; to port output
    ; Start conditions
    clr rFlag ; Clear flags
    ldi rmp,0x10 ; Short cycle
    mov rAdc,rmp ; to ADC register
    mov rCyc,rmp ; and to cycle counter
    ; Timer as 8 bit PWM
    ldi rmp,0x80 ; Half intensity
    out OCR0A,rmp ; to timer compare port A
    ldi rmp,(1<<COM0A1) | (1<<WGM01) | (1<<WGM00) ; Fast PWM, clear on TOP
    out TCCR0A,rmp ; to timer control port A
    ldi rmp,(1<<CS01) ; Prescaler 8
    out TCCR0B,rmp ; to control port B
    ldi rmp,1<<TOIE0 ; Overflow Interrupt
    out TIMSK0,rmp ; to timer int mask
    ; Configure ADC: Left adjust result, signal input = ADC2
    ldi rmp,(1<<ADLAR) | (1<<MUX1) ; ADLAR and ADC-Pin
    out ADMUX,rmp ; to ADC multiplexer port
    ; Disable ADC pin driver
    ldi rmp,1<<bAdD ; Disable port driver
    out DIDR0,rmp ; to Disable port
    ; Switch on ADC, enable int and start conversion
    ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rmp ; to ADC control port
    ; Enable PCINT
    ldi rmp,1<<bTaE ; Key interrupts
    out PCMSK,rmp ; to PCINT mask port
    ldi rmp,1<<PCIE ; Enable PCINT
    out GIMSK,rmp ; to General Interrupt mask
    ; Sleep mode and interrupts
    ldi rmp,1<<SE ; Enable sleep mode idle
    out MCUCR,rmp ; to control port
    sei ; Enable interrupts
;
Loop:
    sleep ; put to sleep
    nop ; After wakeup by interrupts
    sbrc rFlag,bCy ; Handling flag cycle end?
    rcall Cycle ; Handle flag
    rjmp Loop ; Go to sleep again
;
Cycle: ; Handle flag, with clock cycles
    cbr rFlag,1<<bCy ; Clear flag, +1 = 1
    sbrc rFlag,bGn ; Green LED?, +1/2 = 2/3
    rjmp CycleGreen ; Yes, LED to green, +2 = 4
    sbi pOut,bRAO ; switch LED red, +2 = 5

```

```

ldi rmp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00); +1 = 6
out TCCR0A,rmp ; to timer control port A, +1 = 7
rjmp CycleDirection ; Change direction, +2 = 9
CycleGreen:
    cbi pOut,bRA0 ; switch LED to green, +2 = 6
    ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|(1<<WGM00); +1 = 7
    out TCCR0A,rmp ; to timer control port A, +1 = 8
CycleDirection:
    in rmp,OCR0A ; Read PWM compare match value, +1 = 10/9
    sbrc rFlag,bAb ; Jump over if downward flag clear, +1/2 = 11/12/10/11
    rjmp CycleDown ; to downward ; +2 = 13/12
    ; Compare value upward
    inc rmp ; PWM one step up, +1 = 13/12
    brne CycleSet ; not zero, write compare, +1/2 = 14/14
    sbr rFlag,1<<bAb ; Set downward flag, +1 = 15
    ldi rmp,0xFF ; to TOP value, +1 = 16
    rjmp CycleSet ; write new value, +2 = 18
CycleDown:
    subi rmp,1 ; Decrease by one, +1 = 14/13
    brcc CycleSet ; write value if carry clear, +1/2 = 15/16/14/15
    cbr rFlag,1<<bAb ; clear downward flag, +1 = 16/15
    clr rmp ; start at zero, +1 = 17/16
CycleSet:
    out OCR0A,rmp ; write new compare value, +1 = 15/15/19/17/16/18/17
    ret ; ready, return, +4 = 19/19/23/21/20/22/21
;
; End of source code
;

```

The following instructions are new:

- MOV Register,Register: copies the content of the second register to the first,
- EOR Register,Register: Exclusive OR of the two registers, result to the first one (inverts all bits in the first register that are set in the second one),
- INC Register: increases the register by one,
- BRCC Label: branches to the label if the carry flag in SREG is clear,
- SUBI Register,Constant: subtracts the constant from the register and writes the result to the register (works only in registers R16 to R31).

Compared to the previous program the timer now runs eight times faster. Only the line with the timer init has been changed. In the calculation of the constant in the header 64 has been exchanged with 8. The constant to determine the inactivity duration of the key has increased accordingly to increase the monitored time. The constant now is cCnt=25. This is the advantage to calculate such constants on top of the source code, any subsequent changes are then simplified.

To find out which value cCnt now has you need again the symbol listing that gavrasm provides if -s has been set as parameter.

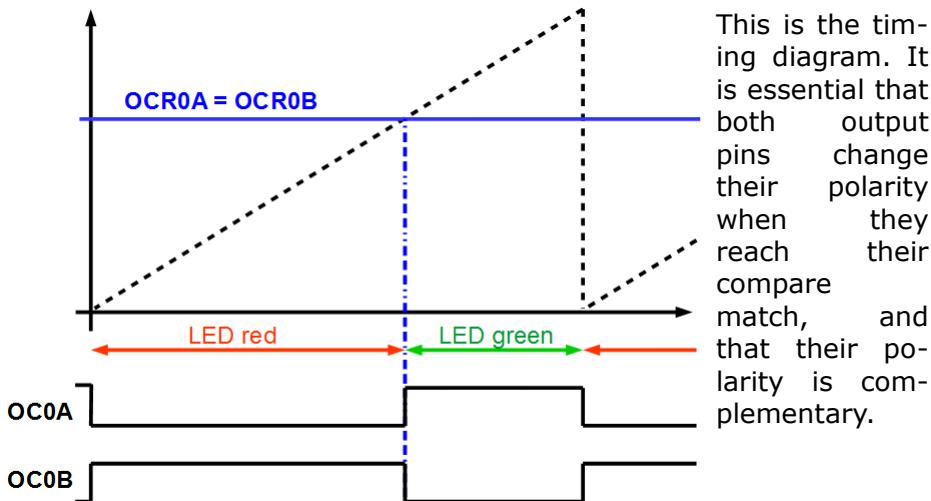
Home	Top	AD conversion	PCINT	Intensity	Color selection	Dynamic	Red/Green
----------------------	---------------------	-------------------------------	-----------------------	---------------------------	---------------------------------	-------------------------	---------------------------

8.7 Fast red/green change

8.7.1 Task 3

This is a bonus task. What happens with the color of the LED if we switch fast between red and green? Higher voltages on the potentiometer shall increase the red duration.

8.7.2 Solution



This is the timing diagram. It is essential that both output pins change polarity when they reach their compare match, and that their polarity is complementary.

The signal outputs OC0A and OC0B, when correctly programmed, deliver this push-pull signal that switches between the two colors.

Because the OC0B pin is needed here, the second pin of the duo LED has to be placed on pin 6 (PB1). The key is not needed here, but can remain part of the hardware.

8.7.3 Program

This is the program, the [source code is here](#). Because the switching of OC0A and OC0B is performed by the timer, only the interrupt service routine for the AD conversion is needed.

```

;
; **** -----
; * Duo-LED in push-pull ATtiny13 *
; * (C)2017 by gsc-elektronic.net *
; **** -----
;

.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;

; ----- Registers -----
; frei: R0 .. R15
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside ints
;

; ----- Ports -----
.equ pDir = DDRB ; Port outputs
.equ bARD = DDB1 ; Red anode
.equ bCRD = DDB0 ; Red cathode
;

; ----- Timing -----
; Clock          = 1200000 Hz
; Prescaler      =    64
; PWM-Stages     =    256
; PWM-Frequency =   73 Hz
;

; -- Reset- and Interrupt vectors -
.CSEG ; Code Segment
.ORG 0 ; Start at address 0
    rjmp Start ; Reset vector, jump to init
    reti ; INT0-Int, inactive
    reti ; PCINT-Int, inactive
    reti ; TIM0_OVF, inactive
    reti ; EE_RDY-Int, inactive
    reti ; ANA_COMP-Int, inactive
    reti ; TIM0_COMPA-Int, inactive
    reti ; TIM0_COMPB-Int, inactive
    reti ; WDT-Int, inactive

```

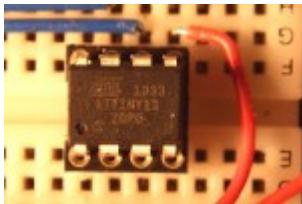
```

rjmp AdcIsr ; ADC-Int, active
;
; ----- Interrupt Service Routines -----
;
AdcIsr:
    in rimp,ADCH ; Read ADC result MSB
    out OCR0A,rimp ; to Compare port A
    out OCR0B,rimp ; to Compare port B
    ; Restart ADC (prescaler 128, Interrupt enable)
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; to ADC control port
    reti
;
; ----- Main program init -----
Start:
    ; Stack init
    ldi rmp,LOW(RAMEND) ; to SRAM end
    out SPL,rmp ; to stack pointer
    ; Init output pins
    ldi rmp,(1<<bARD) | (1<<bCRD) ; Anode and cathode are output
    out pDir,rmp ; to direction port
    ; Init comparer
    ldi rmp,0x80 ; half/half at start
    out OCR0A,rmp ; to compare port A
    out OCR0B,rmp ; to compare port B
    ; Timer as PWM with A- and B-output control
    ldi rmp,(1<<COM0A1) | (1<<COM0A0) | (1<<COM0B1) | (1<<WGM01) | (1<<WGM00)
    out TCCR0A,rmp ; to timer control port A
    ; Start timer with a prescaler of 64
    ldi rmp,(1<<CS01) | (1<<CS00) ; Prescaler 64
    out TCCR0B,rmp ; to timer control port B
    ; Configure ADC: Left adjust, signal input = ADC2
    ldi rmp,(1<<ADLAR) | (1<<MUX1) ; ADLAR and ADC pin
    out ADMUX,rmp ; to ADC multiplexer port
    ; ADC input driver off
    ldi rmp,1<<ADC2D ; Disable port driver
    out DIDR0,rmp ; in disable port
    ; Switch on and start ADC
    ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rmp ; to ADC control port
    ; Sleep enable
    ldi rmp,1<<SE ; Sleep mode idle
    out MCUCR,rmp ; to general control port
    ; Enable interrupts
    sei ; set I flag
Loop:
    sleep ; go to sleep
    nop ; after waking up
    rjmp Loop ; go to sleep again
;
; End of source code
;

```

No new instructions are used here.

The result is not exactly what is to be expected: the red and green light of the LED does not really mix to new colors. The reason for that is that the red and green light are not generated by the same diode. There are two separated diodes in action, that are located in some distance to each other. So unfortunately we still see the origin of the green and red light separately instead of an ideally mixed color.



Lecture 9: An audio generator with ADC, tone table, multiplication

The OC0A output is used here as a variable audio generator with adjusting the tone's frequency with a potentiometer. Further more 8 bit numbers are multiplied and music is played here.

9.0 Overview

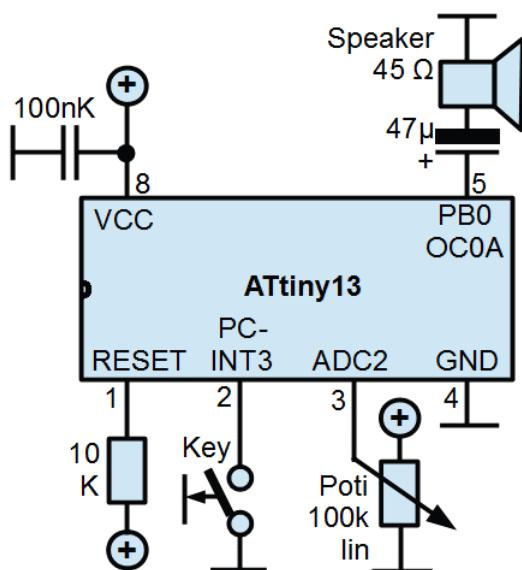
1. [Introduction to audio generation](#)
2. [Hardware, components and mounting](#)
3. [Tone control](#)
4. [Introduction to tables](#)
5. [Introduction to multiplication](#)
6. [Gamut output](#)
7. [Playing pieces of music](#)

9.1 Introduction to audio generation

Generating tones is essentially the same as blinking a LED, with frequencies between 30 cs/s and 20 kcs/s (for bats: up to 40 kcs/s). So we learn here not much on timers, only how to attach a speaker to a port pin instead of a LED.

9.2 Hardware, components and mounting

9.2.1 The hardware scheme



To sound tones a speaker is needed. This is attached to OC0A. A capacitor of $47 \mu\text{F}$ decouples the DC from the port pin and transfers only the AC component.

A key and the potentiometer are attached like in the previous experiments.



9.2.2 The components

9.2.2.1 The speaker

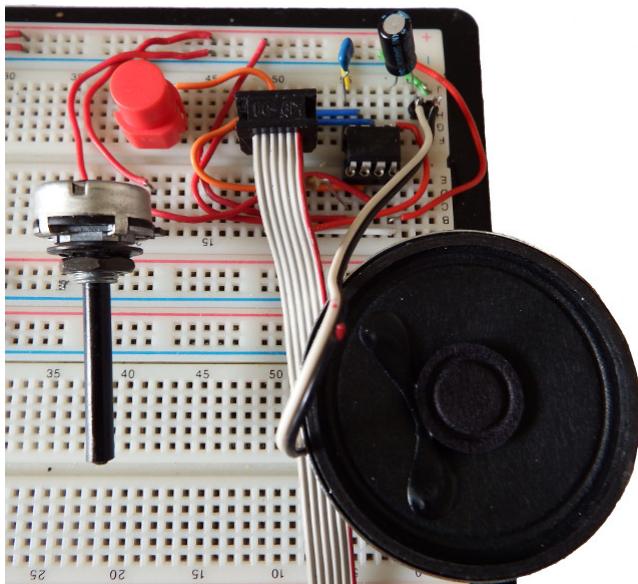
This is the speaker. He has an impedance of 45Ω to yield a strong audio signal. The two pins are soldered to a short cable and short pins that fit into the breadboard. The polarity of the speaker is irrelevant for our application, it has only acoustic consequences if two more of those are operated.

9.2.2.2 The electrolytic capacitor



This is an electrolytic capacitor of $47 \mu\text{F}$. This is a component for which correct polarity is essential. The minus pole is marked, the longer of the two wires is plus.

9.2.3 Mounting



The speaker is tied to pin 5, via the electrolyte.

With that we can start sound-generation.

Home	Top	Tone generation	Hardware	Tone control	Tables	Multiplication	Gamut	Music
----------------------	---------------------	---------------------------------	--------------------------	------------------------------	------------------------	--------------------------------	-----------------------	-----------------------

9.3 Regulating the tone frequency

9.3.1 Simple task 1

Task 1 is to output audio tones via the speaker and to regulate their frequency with the potentiometer. The tones shall range between 300 cs/s and 75 kcs/s. The tone shall only be audible if the key is pressed.

9.3.2 Solution

9.3.2.1 Frequency ranges

It is already clear that the CTC mode of the timer has to be used here. The OC0A output pin has to toggle (from low to high and back). As each swing of the rectangle requires two CTC periods, the resulting frequency is half that of the CTC frequency. The frequency depends from the prescaler and the compare value. Ranges cover the following frequencies:

Clock	Pre-scaler	OCR0A =0	OCR0A =255	Clock	Pre-scaler	OCR0A =0	OCR0A =255
9.6 Mcs/s	1	4.8 Mcs/s	18.75 kcs/s	1.2 Mcs/s	1	600 kcs/s	2.34 kcs/s
	8	600 kcs/s	2.34 kcs/s		8	75 kcs/s	292.5 cs/s
	64	75 kcs/s	292.5 Hz		64	9.38 kcs/s	36.6 cs/s
	256	18.75 kcs/s	73.1 cs/s		256	2.35 kcs/s	9.15 cs/s
	1024	4.69 kcs/s	18.3 cs/s		1024	586 cs/s	2.29 cs/s

At 1.2 Mcs/s clock, the audible range is well covered with a prescaler of 8.

9.3.2.2 AD values and OCR0A values

The higher the measured voltage from the potentiometer the higher the tone frequency should be. The OCR0A value behaves opposite: the larger the lower is the frequency. So either the potentiometer has to be reverted or a reversion of the measured value has to take place. A software solution for this would be to subtract the measured 8 bit value from hex 0xFF. That would go like this:

```
ldi Register1,0xFF
sub Register1,Register2 ; Register2 = measured value
mov Register2,Register1
```

Inverting all bits in a register is a task that the controller's Central Processing Unit can do with a special instruction, see the source code, so that we do not need to take this path.

9.3.3 Program

This is the program, [the source code here](#).

```
;;
; ****
; * Audio generator with key and tone regulator *
; * (C)2017 by http://www.avr-asm-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
----- Registers -----
; free: R0 .. R14
.def rSreg = R15 ; Save/restore status register
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside interrupts
; free: R18 .. R31
;
----- Ports -----
.equ pOut = PORTB ; Output port
.equ pDir = DDRB ; Direction port
```

```

.equ pInp = PINB ; Input port
.equ bLspD = DDB0 ; Speaker output direction pin
.equ bTasO = PORTB3 ; Pull up key output pin
.equ bTasI = PINB3 ; Key input pin
.equ bAdID = ADC2D ; ADC input disable
;
; ----- Timing -----
; Clock          = 1200000 cs/s
; Prescaler     = 8
; CTC TOP range = 0 .. 255
; CTC divider range = 1 .. 256
; Toggle divider = 2
; Frequency range = 75 kcs/s .. 293 cs/s
;
; ----- Reset- und Interrupt vectors -----
.CSEG ; Assemble to Code-Segment
.ORG 0 ; Start at address zero
    rjmp Start ; Reset Vector, jump to init
    reti ; INT0-Int, inactive
    rjmp PcIntIsr ; PCINT-Int, active
    reti ; TIM0_OVF, inactive
    reti ; EE_RDY-Int, inactive
    reti ; ANA_COMP-Int, inactive
    reti ; TIM0_COMPA-Int, inactive
    reti ; TIM0_COMPB-Int, inactive
    reti ; WDT-Int, inactive
    rjmp AdcIsr ; ADC-Int, active
;
; ----- Interrupt Service Routines -----
PcIntIsr: ; PCINT-Interrupt key
    sbic pInp,bTasI ; Skip if key input = 0
    rjmp PcIntIsrOff ; Key is not pressed
    ldi rimp,(1<<COM0A0) | (1<<WGM01) ; Toggle output, CTC-A
    out TCCR0A,rimp ; to timer control port A
    rjmp PcIntIsrRet ; return
PcIntIsrOff:
    ldi rimp,(1<<COM0A1) | (1<<WGM01) ; Clear output, CTC-A
    out TCCR0A,rimp ; to timer control port A
PcIntIsrRet:
    reti ; return from interrupt, set I flag
;
AdcIsr: ; ADC interrupt
    in rSreg,SREG ; save SREG
    in rimp,ADCH ; read MSB result
    com rimp ; invert value
    out OCR0A,rimp ; to CTC TOP port
    ; Restart ADC, int enable
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; to ADC control port A
    out SREG,rSreg ; restore SREG
    reti ; return from interrupt, set I flag
;
; ----- Program start and init -----
Start:
    ; Stack init
    ldi rmp,LOW(RAMEND) ; Set to SRAM end
    out SPL,rmp ; to stack pointer
    ; In- and output ports
    ldi rmp,1<<bLspD ; Speaker direction output
    out pDir,rmp ; to direction port
    ldi rmp,1<<bTasO ; Pull up on key port pin
    out pOut,rmp ; to output port
    ; Configure timer as CTC
    ldi rmp,(1<<COM0A1) | (1<<WGM01) ; Clear output, CTC-A
    out TCCR0A,rimp ; to control port A
    ldi rmp,1<<CS01 ; Prescaler = 8, start timer
    out TCCR0B,rimp ; to timer control port B
    ; Configure and start AD conversion
    ldi rmp,(1<<ADLAR) | (1<<MUX1) ; Left adjust, ADC2
    out ADMUX,rimp ; to ADC MUX port
    ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; to ADC control port A, start
    ; Configure PCINT for key input
    ldi rmp,1<<PCINT3 ; Enable PB3 interrupt
    out PCMSK,rimp ; in PCINT mask port
    ldi rmp,1<<PCIE ; Enable PCINT
    out GIMSK,rimp ; in interrupt mask port

```

```

; Enable sleep mode
ldi rmp,1<<SE ; Sleep mode idle
out MCUCR,rmp ; to MCU control port
; Enable interrupts
sei
; ----- Main program loop -----
Loop:
    sleep ; go to sleep
    nop ; Wake up dummy
    rjmp Loop ; go to sleep again
;
; End of source code
;

```

The following instruction is new:

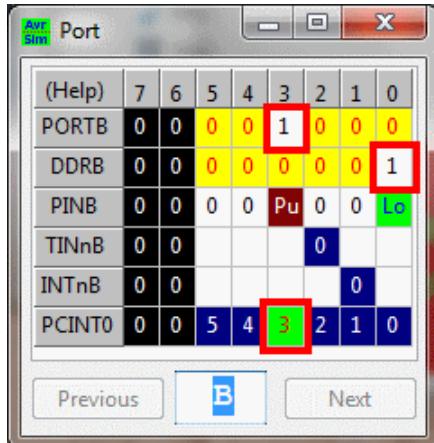
- COM Register: inverts all bits in the register. From 0x00 to 0xFF and from 0xFF to 0x00. Subtract the register content from 0xFF, so called one's complement.

With the machine we can generate Morse code with a regulated tone.



9.3.4 Simulation of the program

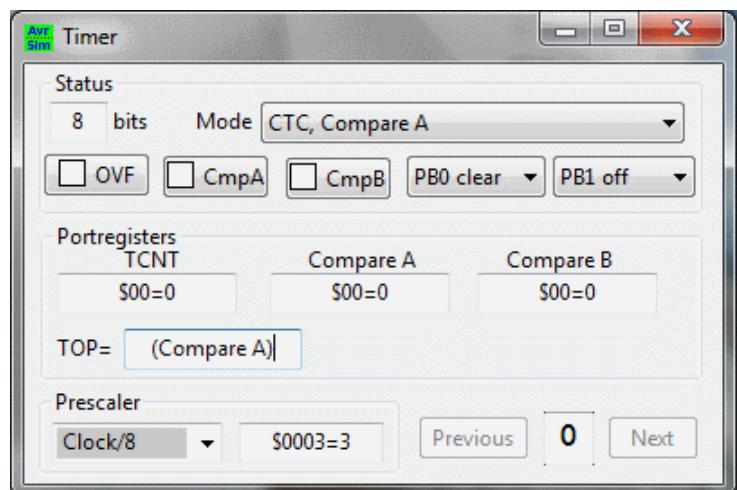
Simulation goes with [avr_sim](#) as follows.



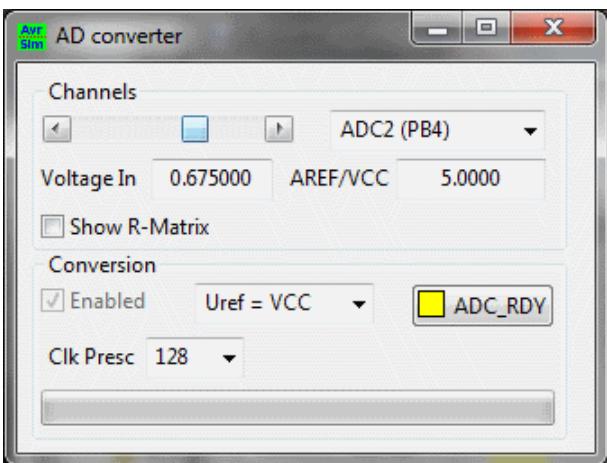
The I/O port PB0 is set to be output, the output pin is low.

The key input portbit PB3 is set to switch on the pull-up resistor. Closing the key pushes the input to low.

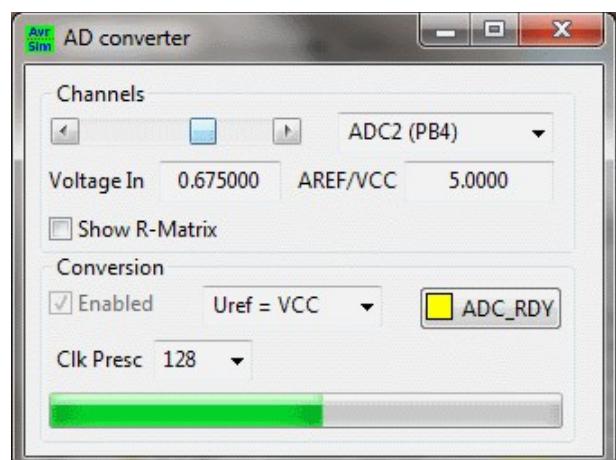
On PB3 the PCINT enable mask is active and the PCINT enable bit is set.



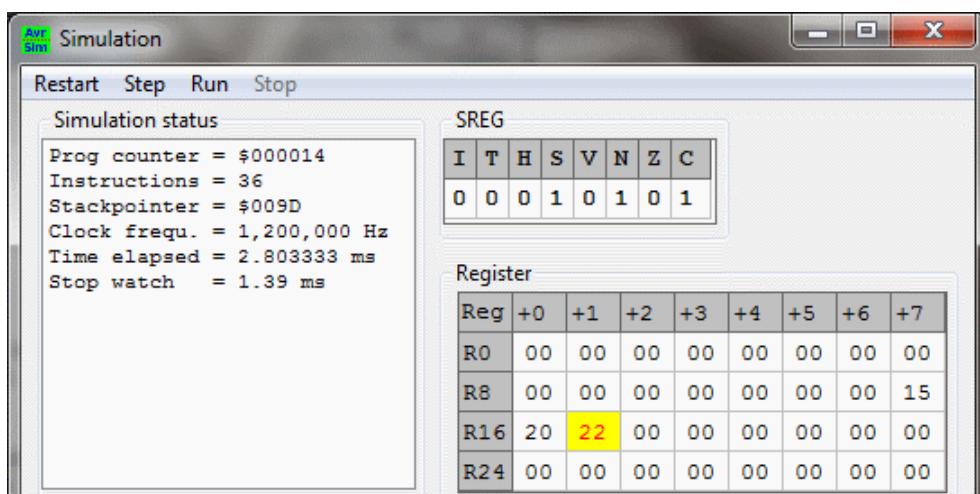
The timer TC0 is set to CTC mode, with clearing the counter after the compare match in A has been reached. The output PB0 is cleared on compare match, leaving PB0 at low voltage. The prescaler is at eight, dividing 1,200,000 Hz to 150,000 Hz counting frequency.



The ADC works with the clock divided by 128: 1,200,000 Hz means 9,375 Hz or, for 13 ADC cycles, requires 1.386 ms per conversion. The reference voltage is 5 V. The voltage value of 0.675 V shall lead to an ADC value of 138. The left adjust should yield a MSB of 34 (0x22). The first conversion has been started (the progress status bar is visible).

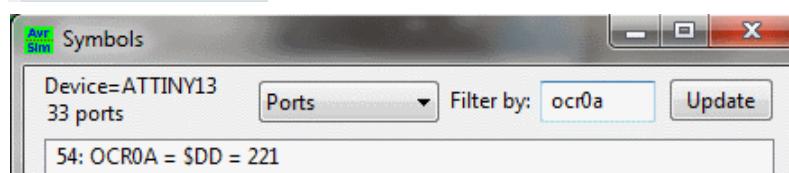
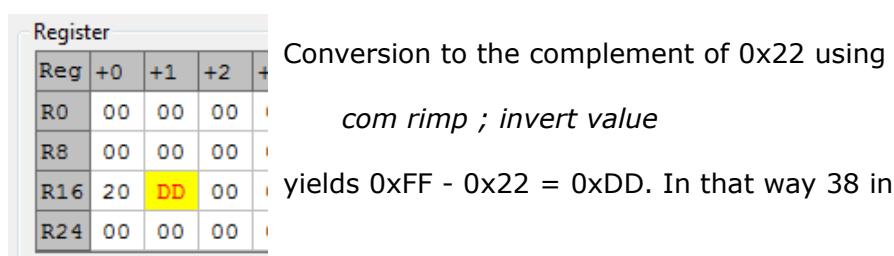


The first AD conversion has made some progress. As the conversion lasts 1.386 ms the controller remains sleeping.



An ADC complete interrupt has occurred and the ADC result is read to R17 using the instruction:

*; read MSB result
in rimp,ADCH*



With "out OCR0A, rimp" this is written to the timer's compare portregister. This does not lead to an audible change as far as the "PB0 clear" condition remains unchanged.

AVR Sim Port

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	1	0	0	1
PINB	0	0	0	0	Lo	0	0	Lo
TINnB	0	0			0			
INTnB	0	0			0			
PCINT0	0	0	5	4	3	2	1	0

Previous B Next

This changes only if a PCINT is executed. Initiated either by changing the input signal in PINB3 manually or by clicking on PCINT3 such a PCINT is initiated and, after four wait cycles and if no other interrupt request is pending, executed.

AVR Sim Timer

Status	8 bits	Mode	CTC, Compare A
<input type="checkbox"/> OVF	<input type="checkbox"/> CmpA	<input type="checkbox"/> CmpB	PB0 toggle ▾
<input type="checkbox"/> PB1 off			
Portregisters	TCNT	Compare A	Compare B
\$02=2	SDD=221	\$00=0	
TOP=	(Compare A)		
Prescaler	Clock/8	\$0000=0	Previous 0 Next

Within the interrupt service routine, the two instructions

```
ldi rimp,(1<<COM0A0)|(1<<WGM01)
out TCCR0A, rimp
```

switch toggling of the PB0 port-pin by the timer on.

AVR Sim Timer

Status	8 bits	Mode	CTC, Compare A
<input type="checkbox"/> OVF	<input type="checkbox"/> CmpA	<input type="checkbox"/> CmpB	PB0 toggle ▾
<input type="checkbox"/> PB1 off			
Portregisters	TCNT	Compare A	Compare B
\$02=2	SDD=221	\$00=0	
TOP=	(Compare A)		
Prescaler	Clock/8	\$0000=0	Previous 0 Next

The pin PB0 now is high.

After reaching the compare value in A and restarting at TCNT = 0 the respective port-bit changes polarity ...

AVR Sim Port

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	1	0	0	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	Pu	0	0	Hi
TINnB	0	0					0	
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

Previous B Next

AVR Sim Simulation

Restart	Step	Run	Stop
Simulation status			
Prog counter = \$00002F			
Instructions = 82			
Stackpointer = \$009F			
Clock frequ. = 1,200,000 Hz			
Time elapsed = 5.8825 ms			
Stop watch = 1.4775 ms			

Since the last CTC 1.4775 ms have elapsed. The time for two such CTC cycles corresponds to an audio frequency of 338 Hz, something between e¹ and f¹.

9.4 Task 2: To play the tones of the gamut

9.4.1 Task

With this part of the lecture the gamut shall be played, the potentiometer shall select the tone to be played.

9.4.2 Introduction to tables

9.4.1.1 The gamut table

Unfortunately german/european/american taste allows only certain tones. Softly changing tone heights are out and not allowed here. We need a table of those allowed tones. Those are collected in the following table.

To teach those tones to the controller's timer, the CTC values and prescalers are associated to those tones. As the timer does not exactly fit the desired frequency, the resulting frequencies and the deviation from the gamut tone are also given.

Tone	Cs/s	Presc	CTC	Real(cs/s)	Delta(%)	#
a	440	8	170	441.18	0.27%	0
h	495	8	152	493.42	-0.32%	1
cis	550	8	136	551.47	0.27%	2
d	586.66	8	128	585.94	-0.12%	3
e	660	8	114	657.89	-0.32%	4
fis	733.33	8	102	735.29	0.27%	5
gis	825	8	91	824.18	-0.10%	6
a'	880	8	85	882.35	0.27%	7
h'	990	8	76	986.84	-0.32%	8
cis'	1100	8	68	1102.94	0.27%	9
d'	1173.32	8	64	1171.88	-0.12%	10
e'	1320	8	57	1315.79	-0.32%	11
fis'	1466.66	8	51	1470.59	0.27%	12
gis'	1650	8	45	1666.67	1.01%	13
A	1760	8	43	1744.19	-0.90%	14
H	1980	8	38	1973.68	-0.32%	15
CIS	2200	8	34	2205.88	0.27%	16
D	2346.64	8	32	2343.75	-0.12%	17
E	2640	1	227	2643.17	0.12%	18
FIS	2933.32	1	205	2926.82	-0.22%	19
GIS	3300	1	182	3296.70	-0.10%	20
A'	3520	1	170	3529.41	0.27%	21
H'	3960	1	152	3947.36	-0.32%	22
CIS'	4400	1	136	4411.76	0.27%	23
D'	4693.28	1	128	4687.5	-0.12%	24
E'	5280	1	114	5263.15	-0.32%	25
FIS'	5866.64	1	102	5882.35	0.27%	26

Tone	Cs/s	Presc	CTC	Real(cs/s)	Delta(%)	#
GIS'	6600	1	91	6593.40	-0.10%	27
A"	7040	1	85	7058.82	0.27%	28

The deviations are, all in all, relatively small. I am not able to realize a difference between 440 and 441 cs/s. So we can accept that, without having an extra xtal oscillator of 1.1968 Mcs/s (which is not available anyway).

To enable the timer to play those frequencies we need a gamut table from which the controller can read the CTC and prescaler values. As the values from a to a', from a' to A, from A to A' and from A' to A" always differ by a constant value of 2, we could calculate those from a base table. But that would be complicated because of the changes in the optimal prescaler value (at high frequencies above D = 1, at smaller ones = 8). So the table should as well hold the prescaler value together with the CTC value. The table length covers four octaves.

9.4.2.1 Tables and their placement

The table needs $2 \times 29 = 58$ bytes length. There are principally three locations in the controller where this table can be placed:

1. the SRAM storage. In an ATtiny13 64 bytes of SRAM are available. The SRAM would be rather full and conflicts with the stack are to be expected., which also uses SRAM.
2. the EEPROM storage. This provides also 64 bytes. That would fit, but would be rather full.
3. the Flash storage. This has 512 words or 1,024 bytes and would provide enough space, even for additional octaves.

9.4.2.2 Table in SRAM

In this first case there is no other opportunity than writing each value of the table, e.g. with the instruction STS Address,Register, to its place. The following would have to be programmed:

```

ldi R16,8 ; Prescaler value
sts 0x60,R16 ; store at address 0x0060 in SRAM
ldi R16,170 ; CTC value
sts 0x60+1,R16 ; store at address 0x0061 in SRAM
[...]
ldi R16,1 ; Prescaler value
sts 0x60+56,R16 ; store at address 0x0098 in SRAM
ldi R16,85 ; CTC value
sts 0x61,R16 ; store at address 0x0099 in SRAM

```

Each value pair would require four instructions, of which two (STS) are double-word instructions (six instruction words per pair). Even if we simplify those instructions a little bit by using the instruction ST Z+,Register, this would be a lengthy affair. ST Z+ goes as follows:

```

ldi ZH,HIGH(0x0060) ; Pointer Z to SRAM start address, MSB
ldi ZL,LOW(0x0060) ; dto., LSB
ldi R16,8 ; Prescaler value
st Z+,R16 ; store R16 in SRAM and increase address in Z
ldi R16,170 ; CTC value
st Z+,R16 ; to next address
[...]
ldi R16,1 ; Prescaler value
st Z+,R16 ; store in SRAM and increase address in Z
ldi R16,85 ; CTC value
st Z,R16 ; store R16 to last address in SRAM

```

With the last value to be written the storing with automatic address increment is changed to ST Z,Register, without increasing Z.

The opposite instruction, to decrease the address in Z, is also available, but works a bit differ-

ent. ST -Z, Register the address is first decreased and then the register content is written to SRAM to the already decreased address. Norwegian language constructors know how to confuse beginners.

Besides the lengthy procedure to code the assembler source, the SRAM is not a convenient place to locate a lengthy table.

9.4.2.3 Table in EEPROM

The second location to place the table is the EEPROM. Which offers slightly better conditions for that. Here the construction would be:

```
.ESEG  
.ORG 0  
.db 8,170  
[...]  
.db 1,85  
.CSEG
```

The assembler directive ".ESEG" places the resulting code not into the storage flash memory but into an extra hex file named ".EEP", which can be written to the EEPROM directly, byte by byte. At the end of the EEPROM content the directive ".CSEG" switches back into the code segment, so that further instructions can be programmed to the flash memory.

The ".ORG 0" directive defines the address place to which the table is written in the EEPROM. The 0 places the EEPROM table to the beginning.

The numerous ".DB" directives with the comma-separated bytes are placed to subsequent addresses in the EEPROM. .DB accepts bytes, words or text (ASCII characters in ", character by character).

How we access to the EEPROM's content we learn in a later lecture.

This is a more comfortable manner, but not as elegant like the third opportunity.

9.4.2.4 Table in program flash memory

Now it is a little bit more complicated, because the program storage memory is organized word-wise, in 16 bit words. With

```
Table:  
.db 8,170  
[...]  
.db 1,85
```

the following will happen:

- The "8" will be converted to the LSB, the "170" as MSB and placed as a 16 bit word into the program memory. The current address at which this word is stored is given by the label "Table:".
- All subsequent ".DB" directives place similar words to the following addresses.

It is clear that per ".DB" ALWAYS whole words are written to program memory. If ".DB 1" is written to the source code, effectively 0x0001 is written there. The automatic addition of 0x00 as MSB, if the number of bytes placed with a single .DB directive is odd, will be signaled by a warning during the assembly process. In the .DB directive the first byte written is always the LSB.

The same warning of the assembler results, if we place text with a .DB directive into flash memory, e.g. with "ABC", and if the text has an odd number of characters. In that case a 0x00 character is added, if we do not formulate this different, e.g. as .DB "ABC ".

A second opportunity to construct the table in the flash is by definitely writing words to the table, e.g.

```
Table:  
.dw 8+170*256  
[...]  
.dw 1+85*256
```

This creates directly the words to be placed to the table, and we can select which part is the LSB and which part is the MSB.

Now we have to access this table to read values from it. To read the first byte, we can use the LPM instruction. Without parameters this instruction reads the byte on address Z in the flash memory to the register R0. Note that the address is in the bits 1 to 15 of Z while bit 0 specifies if the lower (0) or upper (1) byte on that address shall be read. We can formulate as follows:

```
ldi ZH,HIGH(2*Table) ; MSB pointer to Z  
ldi ZL,LOW(2*Table) ; dto., LSB  
lpm ; Load from Program Memory
```

Note that the address "Table:" is multiplied by two, because each address location holds two bytes. "2*Table" accesses the LSB. If access to the the MSB is desired write "2*Table+1".

The LSB read now is stored to register R0. To read it to somewhere else we formulate

```
ldi ZH,HIGH(2*Table) ; MSB pointer to Z  
ldi ZL,LOW(2*Table) ; dto., LSB  
lpm R16,Z ; Load from Program Memory to R16
```

To read both bytes one after the other, the LPM r,Z+ instruction has to be executed: the Z+ increments the content of Z automatically after reading the content at Z. Like this:

```
ldi ZH,HIGH(2*Table) ; MSB pointer to table in Z  
ldi ZL,LOW(2*Table) ; dto., LSB  
lpm XL,Z+ ; Load LSB from Program Memory to register XL  
lpm XH,Z+ ; Load MSB from Program Memory to register XH
```

This increases the address automatically. Backwards the Norwegian programming style is again to decrease the address first and then to read the content, with LPM r,-Z. But beware: this instruction is not implemented in the ATtiny13. By the way, the registers XL and XH as well as YH and YL and ZH and ZL as well as the double register pairs X, Y and Z are defined in the "def.inc" file. If you do not include this in the assembler source header, you will get error messages from the assembler, unless you defined those e.g. with ".def ZH = R31". This has historic reasons because the first AVR devices (e.g. the ancient AT90S1200) had no pointer register pairs.

With that, the gamut table is constructable, like this:

GamutTable: .db 1<<CS01, 169 ; a #0 .db 1<<CS01, 151 ; h #1 .db 1<<CS01, 135 ; cis #2 .db 1<<CS01, 127 ; d #3 .db 1<<CS01, 113 ; e #4 .db 1<<CS01, 101 ; fis #5 .db 1<<CS01, 90 ; gis #6 .db 1<<CS01, 84 ; a' #7 .db 1<<CS01, 75 ; h' #8	.db 1<<CS01, 67 ; cis' #9 .db 1<<CS01, 63 ; d' #10 .db 1<<CS01, 56 ; e' #11 .db 1<<CS01, 50 ; fis' #12 .db 1<<CS01, 44 ; gis' #13 .db 1<<CS01, 42 ; A #14 .db 1<<CS01, 37 ; H #15 .db 1<<CS01, 33 ; CIS #16 .db 1<<CS01, 31 ; D #17 .db 1<<CS01, 226 ; E #18	.db 1<<CS00, 204 ; FIS #19 .db 1<<CS00, 181 ; GIS #20 .db 1<<CS00, 169 ; A' #21 .db 1<<CS00, 151 ; H' #22 .db 1<<CS00, 135 ; CIS' #23 .db 1<<CS00, 127 ; D' #24 .db 1<<CS00, 113 ; E' #25 .db 1<<CS00, 101 ; FIS' #26 .db 1<<CS00, 90 ; GIS' #27 .db 1<<CS00, 84 ; A' #28
---	---	--

This table requires 29 words in the flash storage, which are 5.7% of the memory of the ATtiny13 and is an acceptable size.

If we want to read the tenth note from the table and write it to the timer, the code for that goes like this:

```
ldi R16,10 ; tenth note  
lsl R16 ; Note multiplied by 2 (2 bytes per note)  
ldi ZH,HIGH(2*GamutTable)
```

```

ldi ZL,LOW(2*GamutTable)
add ZL,R16 ; add note to pointer
ldi R16,0 ; CLR would also clear the carry flag!
adc ZH,R16 ; add eventual carry
lpm R0,Z+ ; read prescaler value
out TCCR0B,R0 ; write to timer control port B
lpm R0,Z ; read CTC value
out OCR0A,R0 ; write to CTC compare value

```

With that, the storage, the reading and the use of the gamut table is resolved for our case here.



9.4.3 Introduction to multiplication

One problem solved, but another problem comes up immediately. The ADC provides measurement data between 0 and 1,023 (without ADLAR) or 0 and 255 (with ADLAR). But our gamut table has 29 entries only. We could resolve that by just limiting the values down to 28, and the problem is already solved. That would not be a really nice and intelligent solution, because those 28 different notes would be available within 2.8% of the potentiometer range (10 bit), while note #29 takes all the rest (97.2%). With ADLAR, 11% cover 28 tones while 89% covers only one single note, not very much more convenient. Not a real linear coverage, A'' will be seriously overrated.

Somehow the incoming 1,023 or 255 should be linearly downsized to 28. The C programmer has no problem with that, he divides by 36.5 resp. by 9.1 and rounds the result. Unfortunately dividing with those numbers the C compiler invokes the floating number library. And this library alone fills the available space in an ATtiny13 completely, and even exceeds that. The C programmer now moves to a 96 pin ATxmega to have enough flash memory for his monster library. The assembler programmer instead invests a little bit of intelligence and comes up with a much more clever solution, well fitting to the available memory of an ATtiny13.

The solution is to multiply the ADC result, with ADLAR set, with 29 and to divide it by 256. As a math formula: "Result = 29 * ADC / 256".

Dividing by 256 in binary math is as simple as can be: just skip the last eight bits of the result of the multiplication (or: just ignore the LSB).

The problem therefore is reduced to the question on how to multiply the ADC result with 29. Several methods are possible to do this.

9.4.3.1 Simplest multiplication

The simplest type of multiplication is to add the ADC result 29 times to yield a 16 bit result. E.g. like this (with the number of necessary clock cycles to get the execution times):

```

in R0,ADCH ; Read ADC result, +1 = 1
clr R2 ; Clear R2:R1 at start adding, +1 = 2
clr R1 ; +1 = 3
ldi R16,29 ; Multipliator 29, +1 = 4
AddLoop:
    add R1,R0 ; add ADC to result, LSB, +29*1 = 33
    brcc PostCarry ; No overflow to carry, +29*1 = 62
    inc R2 ; Increase MSB when carry is set, +29*1 = 91
PostCarry:
    dec R16 ; count downwards, + 29*1 = 130
    brne AddLoop ; add once again, +28*2 + 1 = 187

```

The 187 clock cycles that are needed are not very long. At 1.2 Mcs/s those are 156 µs, less than a single audio sine wave.

The method to add can be used in all cases where the multiplicator is not too large and where

the extended execution time is not too large. E.g. in case of 10 it is a preferred method: add the base number once, multiply the result twice by 2 (LSL, ROL), then add the base number again and multiply the result by 2 (again LSL and ROL).

In our case the 187 clock cycles are not too much, but there are other methods to multiply that are very much faster.

9.4.3.2 Faster multiplication

Multiplying by 2 is, in the binary world, the simplest and fastest task (LSL and ROL). We can multiply by 2 on and on until we are near our 29. Then we add or subtract a little bit and we have the 29 fold. The next 2-potency is 32, from which we can subtract the ADC result three times. Like in this example:

```

in R0,ADCH ; Read result from ADC to R0, +1 = 1
clr R2 ; R2:R1 is the result, +1 = 2
mov R1,R0 ; Copy ADC result once, +1 = 3
lsl R1 ; LSB * 2, +1 = 4
rol R2 ; MSB * 2 plus carry, +1 = 5
lsl R1 ; LSB * 4, +1 = 6
rol R2 ; MSB * 4 plus carry, +1 = 7
lsl R1 ; LSB * 8, +1 = 8
rol R2 ; MSB * 8 plus carry, +1 = 9
lsl R1 ; LSB * 16, +1 = 10
rol R2 ; MSB * 16 plus carry, +1 = 11
lsl R1 ; LSB * 32, +1 = 12
rol R2 ; MSB * 32 plus carry, +1 = 13
sub R1,R0 ; Subtract once, +1 = 14
brcc NoCarry1 ; Carry clear, +1/2 = 15/16
dec R2 ; Decrease MSB, +1 = 16
NoCarry1:
    sub R1,R0 ; Subtract twice, +1 = 17
    brcc KeinCarry2 ; Carry clear, +1/2 = 18/19
    dec R2 ; Decrease MSB, +1 = 19
NoCarry2:
    sub R1,R0 ; Subtract three times, +1 = 20
    brcc NoCarry3 ; Carry clear, +1/2 = 21/22
    dec R2 ; Decrease MSB, +1 = 22
NoCarry3:

```

New is the instruction ROL register. This is the left-rolling version, compared to the right-rolling ROR. ROL rolls

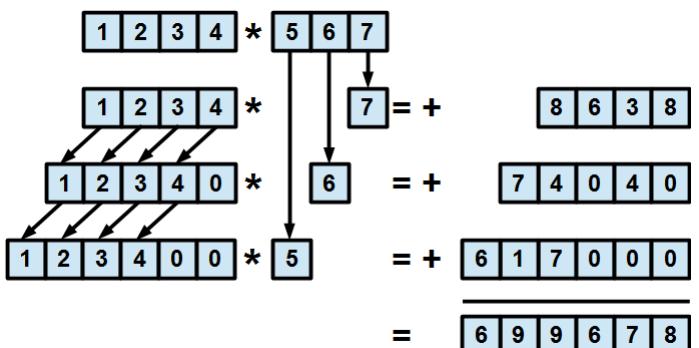
- the bits 0 to 6 to the next left bit (1 to 7),
- the carry bit to bit 0 of the register, and
- bit 7 of the register to the carry flag.

The 22 clock cycles of this mode of multiplication are by a factor of eight faster than the primitive multiplication. But we can do it even faster.

9.4.3.3 Even faster multiplication

The previous methods are tailored closely to the task. The real multiplication, applicable to any combination of two 8 bit binaries, is not so complicated that even C programmer can learn that method (to avoid monster libraries and the associated monster controllers). The binary multiplication is even simpler than decimal multiplication. But let us start with decimal to understand the mechanism.

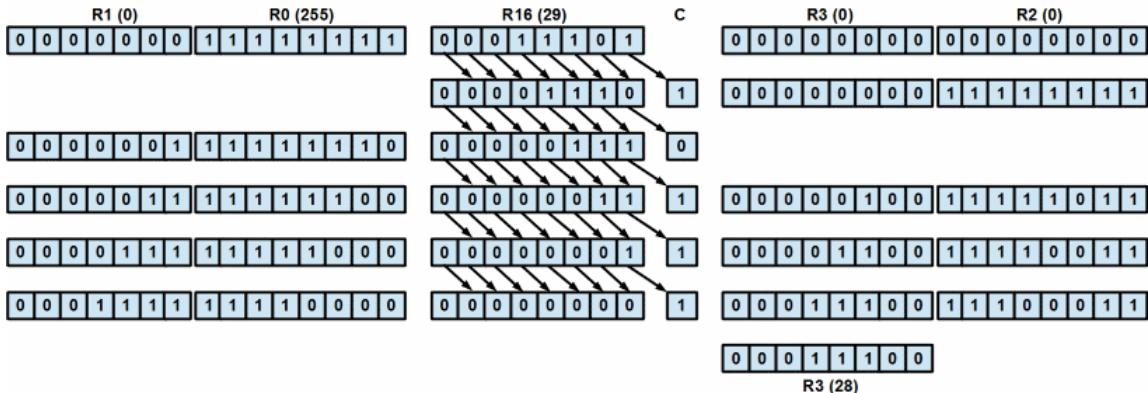
Decimal multiplication goes like this. The first step is to multiply the number with the least significant digit and to add this to the result. Then the number is shifted once left (multiplied by 10), multiplied by the



second significant digit and again added to the result. This is repeated until all digits of the multiplicator have been multiplied and added.

The binary multiplication is even simpler, because only a 0 or a 1 can roll out. Which means no adding to the sum (0) or adding to the sum (1). The mechanism is the same (to roll out one bit to the right, to add the left-shifted number (or not) and to left-shift the number).

The multiplication of 255 by 29 is shown in the picture.



The second number (in R16) is shifted to the right, by which the next 0 or 1 is shifted to the carry flag. If that is 0, the first number (in R1:R0) is not added. If it is a 1 it is added 16 bit wise to the result (in R3:R2). Then the first number (in R1:R0) is shifted left (16 bit left shift). Again a right-shift of the second number, to add or not to add, etc. If all ones are shifted out of the second number, the multiplication is ended.

That is how the code looks like:

```

        in R0,ADCH ; Read MSB from the ADC as LSB, +1 = 1
        clr R1 ; Clear MSB, +1 = 2
        clr R2 ; Clear LSB result, +1 = 3
        clr R3 ; dto., MSB, +1 = 4
        ldi R16,29 ; Set multiplicator, +1 = 5

MultLoop:
        lsr R16 ; Shift lowest bit to carry, +5*1 = 10
        brcc NoAdding ; C = 0, do not add, +1*2+4*1 = 16
        add R2,R0 ; add LSB, +5*1 = 21
        adc R3,R1 ; add MSB with carry, +5*1 = 21

NoAdding:
        lsl R0 ; Left shift first number, LSB, +5*1 = 26
        rol R1 ; Left shift MSB and roll carry to MSB, +5*1 = 31
        tst R16 ; End reached?, +1*5 = 36
        brne MultLoop ; still ones to be processed, +4*2+1*1 = 45

```

The result (28) is now in register R3 (we ignore the LSB R2 = division by 256).

OK, these are 45 clock cycles, and more than method 2. But the routine is processing any 8-by-8 bit multiplication and is not tailored to the 29 in our case. The routine is so simple that even C programmer can learn that, without having to import massive libraries.

So our problem is solved on how to make 28 out of an input of 255 (or whatever value the ADC returns). Without dividing (division by 256 does not really count as a division in the binary world). Many mathematical problems, that require a division, can be solved in that way by avoiding divisions.

9.4.4 Programming the gamut

With that we have the basis to implement the gamut. The program follows, the [source code](#)

[can be downloaded here.](#)

```
;  
; *****  
; * Gamut tones with a potentiometer on an ATTiny13 *  
; * (C)2017 by www.avr-asm-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn13def.inc"  
.LIST  
;  
; ----- Registers -----  
; Used: R0 for LPM and calculations  
; Used: R1 for calculations  
.def rMultL = R2 ; Multipliator, LSB  
.def rMultH = R3 ; dto., MSB  
; free: R4 .. R14  
.def rSreg = R15 ; Save/restore SREG  
.def rmp = R16 ; Multi purpose outside ints  
.def rimp = R17 ; Multi purpose inside ints  
.def rFlag = R18 ; Flag register  
    .equ bAdcR = 0 ; Read in ADC value  
; free: R18 .. R29  
; Used: R31:R30, Z = ZH:ZL for LPM  
;  
; ----- Ports -----  
.equ pOut = PORTB ; Output port  
.equ pDir = DDRB ; Direction port  
.equ pInp = PINB ; Input port  
.equ bSpkD = DDB0 ; Speaker output pin  
.equ bKeyO = PORTB3 ; Pull up Key input pin  
.equ bKeyI = PINB3 ; Key input pin  
.equ bAdID = ADC2D ; ADC Input Disable pin  
;  
; ----- Timing -----  
; Clock          = 1200000 cs/s  
; Prescaler      = 1 or 8  
; CTC TOP range = 0 .. 255  
; CTC divider range = 1 .. 256  
; Toggle divider = 2  
; Frequency range = 600 kcs/s .. 293 cs/s  
;  
; ---- Reset- and Interrupt vectors ---  
.CSEG ; Assemble to code segment  
.ORG 0 ; At start address  
    rjmp Start ; Reset vector, Init  
    reti ; INT0-Int, inactive  
    rjmp PcIntIsr ; PCINT-Int, active  
    reti ; TIM0_OVF, inactive  
    reti ; EE_RDY-Int, inactive  
    reti ; ANA_COMP-Int, inactive  
    reti ; TIM0_COMPA-Int, inactive  
    reti ; TIM0_COMPB-Int, inactive  
    reti ; WDT-Int, inactive  
    rjmp AdcIsr ; ADC-Int, active  
;  
; ---- Interrupt Service Routines ----  
PcIntIsr: ; PCINT-Interrupt key  
    sbic pInp,bKeyI ; Skip next instruction if key = 0  
    rjmp PcIntIsrOff ; Key is not pressed  
    ; Tone output on  
    ldi rimp,(1<<COM0A0)|(1<<WGM01) ; Toggle OC0A, CTC-A  
    out TCCR0A, rimp ; to timer control port A  
    rjmp PcIntIsrRet ; return  
PcIntIsrOff:  
    ; Tone output off  
    ldi rimp,(1<<COM0A1)|(1<<WGM01) ; Clear OC0A, CTC-A  
    out TCCR0A, rimp ; to timer control port A  
PcIntIsrRet:  
    reti  
;  
AdcIsr: ; ADC-Interrupt  
    in rSreg,SREG ; Save SREG  
    in rMultL,ADCH ; Read MSB of ADC (ADLAR)  
    sbr rFlag,1<<bAdcR ; Set flag new ADC value
```

```

; Restart ADC
ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rimp ; to ADC control port A
out SREG,rSreg ; Restore SREG
reti
;
; ----- Program start and Init -----
Start:
; Init stack
ldi rmp,LOW(RAMEND) ; to SRAM end
out SPL,rmp ; to stack pointer
; Init In- and Output ports
ldi rmp,1<<bSpkD ; Speaker output direction
out pDir,rmp ; to direction port
ldi rmp,1<<bKeyO ; Pull up on key pin
out pOut,rmp ; to output port
; Configure timer as CTC
ldi rmp,(1<<COM0A1) | (1<<WGM01) ; Clear OC0A, CTC-A
out TCCR0A,rmp ; to timer control port A
; Prescaler and timer start by ADC int
; Configure ADC and start
ldi rmp,(1<<ADLAR) | (1<<MUX1) ; Left adjust, ADC2
out ADMUX,rmp ; to ADC MUX port
ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; to ADC control port A, start ADC
; PCINT for key input
ldi rmp,1<<PCINT3 ; Enable PB3-Int
out PCMSK,rmp ; to PCINT mask port
ldi rmp,1<<PCIE ; Enable PCINT
out GIMSK,rmp ; to Interrupt Mask port
; Enable sleep
ldi rmp,1<<SE ; Sleep mode idle
out MCUCR,rmp ; to MCU control port
; Enable interrupts
sei
;
----- Main program loop -----
Loop:
sleep ; go to sleep
nop ; After wake up by int
sbrc rFlag,bAdcR ; Skip next if ADC flag zero
rcall AdcCalc ; Convert ADC value to tone
rjmp Loop ; go to sleep again
;
; ----- Convert AD value -----
; ADC value to tone height and timer start
; AD value is in rMultL
AdcCalc:
cbr rFlag,1<<bAdcR ; Clear flag
; Multiply ADC value by 29
clr rMultH ; Clear MSB
clr R0 ; Clear result LSB
clr R1 ; dto., MSB
ldi rmp,29 ; Number of tones plus one
AdcCalcShift:
lsl rmp ; Shift lowest bit to carry
brcc AdcCalcNoAdd
add R0,rMultL ; add LSB to result
adc R1,rMultH ; add MSB with carry
AdcCalcNoAdd:
lsl rMultL ; Multiply by two
rol rMultH ; Roll carry to MSB and multiply by two
tst rmp ; rmp already empty?
brne AdcCalcShift ; no, go on multiplying
; Tone from gamut table
lsl R1 ; Tone number multiply by two
ldi ZH,HIGH(2*GamutTable) ; Z to tone table
ldi ZL,LOW(2*GamutTable)
add ZL,R1 ; Add tone number
ldi rmp,0 ; Add carry
adc ZH,rmp ; add 0 with carry
lpm R0,Z+ ; Read table value LSB to R0
out TCCR0B,R0 ; to timer control port B
lpm ; Read next table value MSB to R0
out OCR0A,R0 ; to compare match A port
ret ; done
;
; ----- Gamut table -----

```

```

GamutTable:
.db 1<<CS01, 169 ; a #0
.db 1<<CS01, 151 ; h #1
.db 1<<CS01, 135 ; cis #2
.db 1<<CS01, 127 ; d #3
.db 1<<CS01, 113 ; e #4
.db 1<<CS01, 101 ; fis #5
.db 1<<CS01, 90 ; gis #6
.db 1<<CS01, 84 ; a' #7
.db 1<<CS01, 75 ; h' #8
.db 1<<CS01, 67 ; cis' #9
.db 1<<CS01, 63 ; d' #10
.db 1<<CS01, 56 ; e' #11
.db 1<<CS01, 50 ; fis' #12
.db 1<<CS01, 44 ; gis' #13
.db 1<<CS01, 42 ; A #14
.db 1<<CS01, 37 ; H #15
.db 1<<CS01, 33 ; CIS #16
.db 1<<CS01, 31 ; D #17
.db 1<<CS00, 226 ; E #18
.db 1<<CS00, 204 ; FIS #19
.db 1<<CS00, 181 ; GIS #20
.db 1<<CS00, 169 ; A' #21
.db 1<<CS00, 151 ; H' #22
.db 1<<CS00, 135 ; CIS' #23
.db 1<<CS00, 127 ; D' #24
.db 1<<CS00, 113 ; E' #25
.db 1<<CS00, 101 ; FIS' #26
.db 1<<CS00, 90 ; GIS' #27
.db 1<<CS00, 84 ; A'' #28
;
; End of source code
;

```

Besides the already applied LPM instructions in all of its sub variants no new instructions are used here.

A serious hint for programming this into the chip: on pin 5 a large electrolytic capacitor and a relatively low resistance of the speaker are attached. Those components conflict with the relatively high serial programming pulses, and error messages from the programmer will result. So please remove the electrolytic capacitor first, before starting to program, and plug it in again after this is finished.



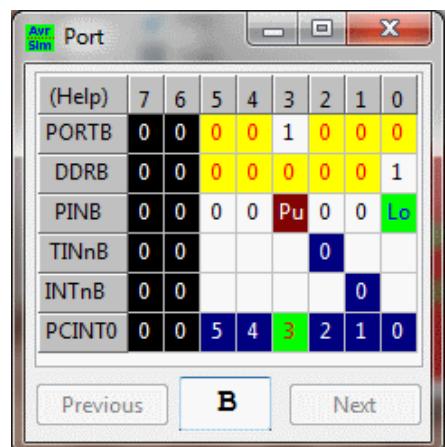
9.4.5 Debugging with avr_sim

The simulation with [avr_sim](#) shows the following results. Simulated are the multiplication routine and the gamut table access with LPM.

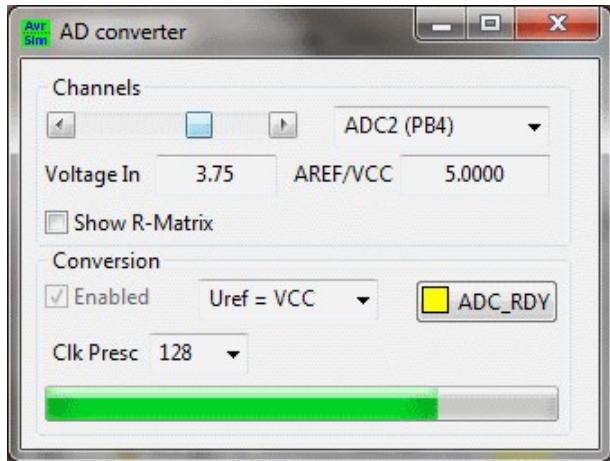
To initiate the hardware we step through the first instructions until the interrupts were enabled by SEI. Then we click on the PCINT in the ports display and initiate starting to play the melody.

Like in the first case the audio output on pin PB0 is set as output and is cleared.

Pin PB3 has its pull-up resistor on and the PCINT interrupt mask enables interrupts on falling edges.



The AD converter is constantly running to measure the input voltage of ADC2, which is on port-pin PB4. The input voltage simulated is 3.75 V, which should yield $3.75 / 5.00 * 1023 = 767$ or, in the left adjusted operating mode, 191 decimal or 0xBF.



After reaching the ADC conversion complete interrupt, the interrupt service routine has written the upper eight bits of the result to register R2 and has set the bAdcR flag. This triggers the routine AdcCalc: after the interrupt woke up the controller from SLEEP. Now we have to calculate which gamut note is selected with the potentiometer.

The instructions

```

124: 000032    2433  clr rMultH ; Clear MSB
125: 000033    2400  clr R0 ; Clear result LSB
126: 000034    2411  clr R1 ; dto., MSB
127: 000035    E10D  ldi rmp,29 ; Number of tones plus one

```

clear rMultH (R3), clear the multiplication result in R0 and R1 and load the number of available gamut tones, decimal 29, to register rmp (R16). Now multiplication can start.

The instruction

```
129: 000036    9506  lsr rmp ; Shift lowest bit to carry
```

shifts R16 to the right and shifts bit 0 to the carry flag in the SREG. In this case the bit is 1.

Register			
Reg	+0	+1	+2
R0	BF	00	BF
R8	00	00	00
R16	OE	CF	00
R24	00	00	00

The instructions

```

130: 000037    F410  brcc AdcCalcNoAdd
131: 000038    0C02  add R0,rMultL ; add LSB to result
132: 000039    1C13  adc R1,rMultH ; add MSB with carry

```

Register			
Reg	+0	+1	+2
R0	00	00	BF
R8	00	00	00
R16	1D	CF	00
R24	00	00	00

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

Register				
Reg	+0	+1	+2	+3
R0	00	00	BF	00
R8	00	00	00	00
R16	OE	CF	00	00
R24	00	00	00	00

first check if the carry flag is cleared, and adding will be skipped. Here this is not the case and the 16 bit number in R3:R2 is added to the result in R1:R0. In the third instruction ADC handles the carry from any overflows to the MSB that might have occurred during the first ADD.

Register				
Reg	+0	+1	+2	+3
R0	BF	00	7E	01
R8	00	00	00	00
R16	OE	CF	00	00
R24	00	00	00	00

The two instructions

```

134: 00003A    0C22  lsl rMultL ; Multiply by two
135: 00003B    1C33  rol rMultH ; Roll carry to MSB, multiply by 2

```

shift the multiplicator in R3:R2 one position to the left and multiply the content by two.

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

Register					
Reg	+0	+1	+2	+3	
R0	BF	00	7E	01	
R8	00	00	00	00	
R16	07	CF	00	00	
R24	00	00	00	00	

As R16 is still not zero, the multiplication is continued by shifting the next bit in R16 to the carry flag. This time a zero is shifted, so adding R3:R2 to R1:R0 is skipped.

Register				
Reg	+0	+1	+2	+3
R0	BF	00	FC	02
R8	00	00	00	00
R16	07	CF	00	00
R24	00	00	00	00

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

Register					
Reg	+0	+1	+2	+3	
R0	B3	09	F0	0B	
R8	00	00	00	00	
R16	00	CF	00	00	
R24	00	00	00	00	

The next step is again multiplying R3:R2 by two, again shift and rotate.

There are still ones in rmp, so we have to repeat shifting. With the third shift, again a one is shifted to the carry.

Register				
Reg	+0	+1	+2	+3
R0	BB	03	FC	02
R8	00	00	00	00
R16	03	CF	00	00
R24	00	00	00	00

Again R3:R2 are added to R1:R0 in 16-bit manner, with ADD/ADC. We do not display the boring multiplication of R3:R2 by 2 that follows next.

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

Register					
Reg	+0	+1	+2	+3	
R0	BF	00	FC	02	
R8	00	00	00	00	
R16	03	CF	00	00	
R24	00	00	00	00	

It is getting boring, but we have to shift rmp again. This time a one is shifted to the carry. We do not display addition of R3:R2 to R1:R0, but this has to be done next.

The next adding of R3:R2 to R1:R0 happens here.

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

Register					
Reg	+0	+1	+2	+3	
R0	BB	03	F8	05	
R8	00	00	00	00	
R16	01	CF	00	00	
R24	00	00	00	00	

Register				
Reg	+0	+1	+2	+3
R0	B3	09	F8	05
R8	00	00	00	00
R16	01	CF	00	00
R24	00	00	00	00

And the next multiplication by 2 for R3:R2 here.

Register				
Reg	+0	+1	+2	+3
R0	B3	09	FO	0B
R8	00	00	00	00
R16	01	CF	00	00
R24	00	00	00	00

Now the last 1 rolls out to carry, and we are nearly done with multiplication.

The last addition of R3:R2 to R1:R0. The following next multiplication by 2 is not necessary any more and makes no sense because R16 is already empty and the multiplication loop will not repeat any more.

Reg	+0	+1	+2	+3
R0	A3	15	F0	0B
R8	00	00	00	00
R16	00	CF	00	00
R24	00	00	00	00

Simulation status								
Prog counter = \$00002C								
Instructions = 86								
Stackpointer = \$009F								
Clock frequ. = 1,200,000 Hz								
Time elapsed = 1.471667 ms								
Stop watch = 55.833333 us								

The stop watch was used to measure the time for multiplication. It shows 55 μ s, which is rather fast considering the many steps of multiplication. No need to upgrade to a ATmega with built-in hardware multiplication though, to avoid the necessary time (that we have plenty of) and replacing the 12 instructions of the multiplication routine by just one single MUL instruction.

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A3	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	92	00

Now Z is set to the gamut table. The instructions

```

138: ; Tone from gamut table
139: 00003E 0C11 lsl R1 ; Tone number multiply by two
140: 00003F E0F0 ldi ZH,HIGH(2*GamutTable) ; Z table
141: 000040 E9E2 ldi ZL,LOW(2*GamutTable)

```

multiply the result of the multiplication by 2 and load the register pair Z with the doubled address of the gamut table.

The gamut table is here:

```

151: ; ----- Gamut table -----
152: GamutTable:
153: .db 1<<CS01, 169 ; a #0
     000049 A902
154: .db 1<<CS01, 151 ; h #1
     00004A 9702
...

```

Its address is 0x000049, which is doubled to 0x0092 in Z to access the content of the table byte-wise.

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A3	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BC	00

Now the doubled value in R1, 0x2A, is added to the table address 0x0092.

```

142: 000041 0DE1 add ZL,R1 ; Add tone number
143: 000042 E000 ldi rmp,0 ; Add carry
144: 000043 1FF0 adc ZH,rmp ; add 0 with carry

```

Here, ZL and R1 are added, the result goes to ZL. Then rmp is set to zero and is added with the carry flag to ZH. This ensures that the MSB is correct. Please note that CLR rmp would have also cleared the carry flag, so LDI is a better choice. Now Z points to 0x00BC.

The note to play at 0x00BC, which corresponds to table address 0x005E, is

```

174: .db 1<<CS00, 169 ; A' #21
     00005E A901

```

So the prescaler will be set to CS00 (prescaling by 1) and the compare match A value to 169. That means a frequency of

$$f = 1,200,000 / 1 / (169 + 1) / 2 = 3,529.4 \text{ Hz}$$

which is roughly the gamut note A' (should be 3,520 Hz).

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	01	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BD	00

Now the instruction

145: 000044 9005 lpm R0,Z+; Read table value
LSB to R0

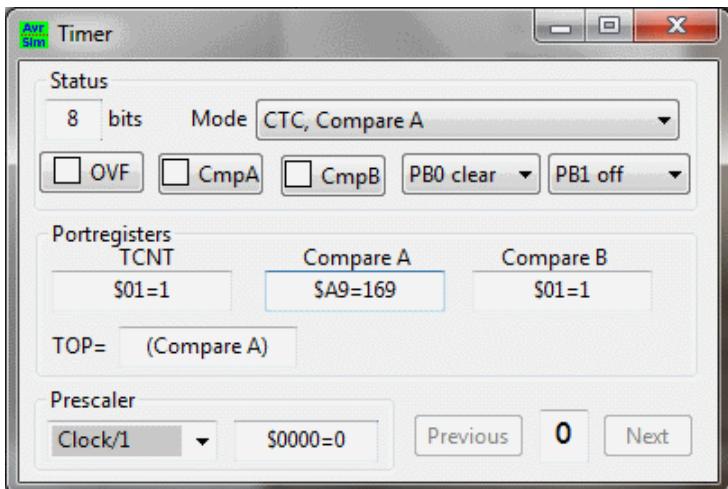
reads the LSB of the table value to R0 and increases the address in Z by one. The result in R0 is written to the timer control register TCCR0B and sets the prescaler to 1.

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A9	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BD	00

Now the instruction

147: 000046 95C8 lpm ; Read next table value MSB to
R0

reads in the second byte in the gamut table, 0xA9 or decimal 169, to R0. LPM always loads to R0 and Z is not increased here. This value is written to the compare match A port-register.



Prescaler and compare match A value of timer TC0 have been set and are ready to play gamut tone A' now, but the port-pin PB0 is still cleared on compare match and the sound is off.



9.4.6 Debugging with the Studio

If we write such routines such as the multiplication or the reading from the gamut table, we would like to know if that part really works fine. At the latest if no tone emerges from the speaker when the key is pressed, we know that we have a bug in our source code software. There are opportunities to follow the controller's work step by step and so to search for such bugs and to identify and correct those. The step-by-step analysis is integrated into the Studio and its name is simulator.

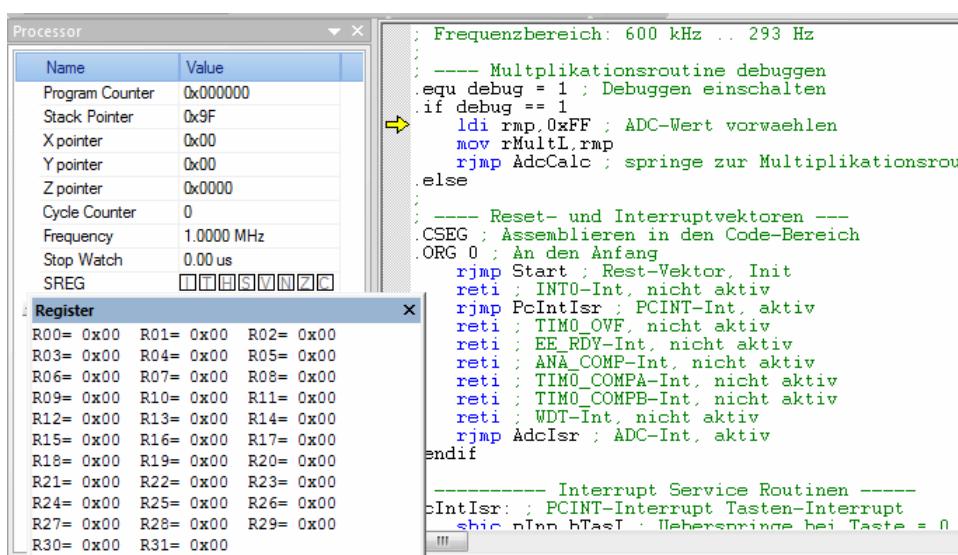
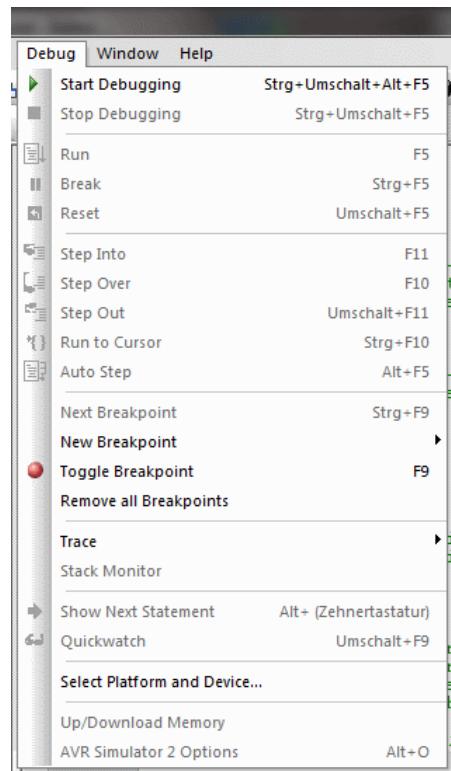
To start the simulator, we just select "Start Debugging" from the menu. Before we do that with our source code, we add the following lines:

```
; ---- To debug the multiplication routine
.equ debug = 1 ; Switch debugging on
;if debug == 1 ; if switch is on do the following:
    ldi rmp,0xFF ; To preselect a simulated ADC value
    mov rMultL,rmp ; copy to the register used for that
    rjmp AdcCalc ; Jump directly to the multiplication
routine
.else
; Skip all the following when the debug switch is on:
;
; ---- Reset- und Interruptvektoren ---
.CSEG ; Assemblieren in den Code-Bereich
.ORG 0 ; An den Anfang
    rjmp Start ; Rest-Vektor, Init
    reti ; INT0-Int, nicht aktiv
    rjmp PcIntIsr ; PCINT-Int, aktiv
    reti ; TIM0_OVF, nicht aktiv
    reti ; EE_RDY-Int, nicht aktiv
    reti ; ANA_COMP-Int, nicht aktiv
    reti ; TIM0_COMPA-Int, nicht aktiv
    reti ; TIM0_COMPB-Int, nicht aktiv
    reti ; WDT-Int, nicht aktiv
    rjmp AdcIsr ; ADC-Int, aktiv

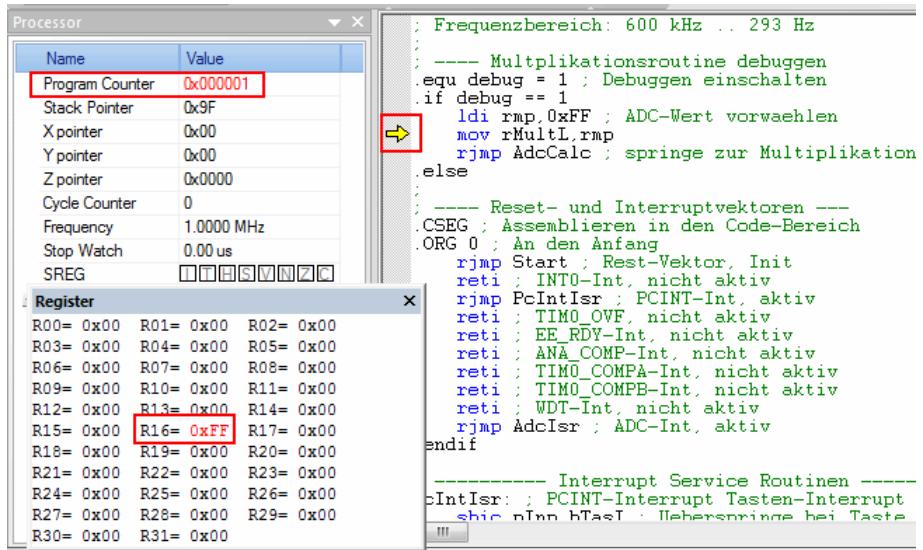
.endif
;
```

Those lines mask the reset and vector table if the debug switch is on. With that the simulator loads a test number to rMultL and jumps directly to the routine, without all the other init procedures. The reason is that the int and vector table does not allow to add code lines prior to address 0. We could also change the init source code section to load our test value and to jump to the routine.

After the simulator has been started, he offers manifold information on the interior of the simulated controller. We can look at the content of the registers (below, left), you can start a stop watch counting cycles and execution times (left, middle). The yellow cursor (above, right) points to the first executable instruction.

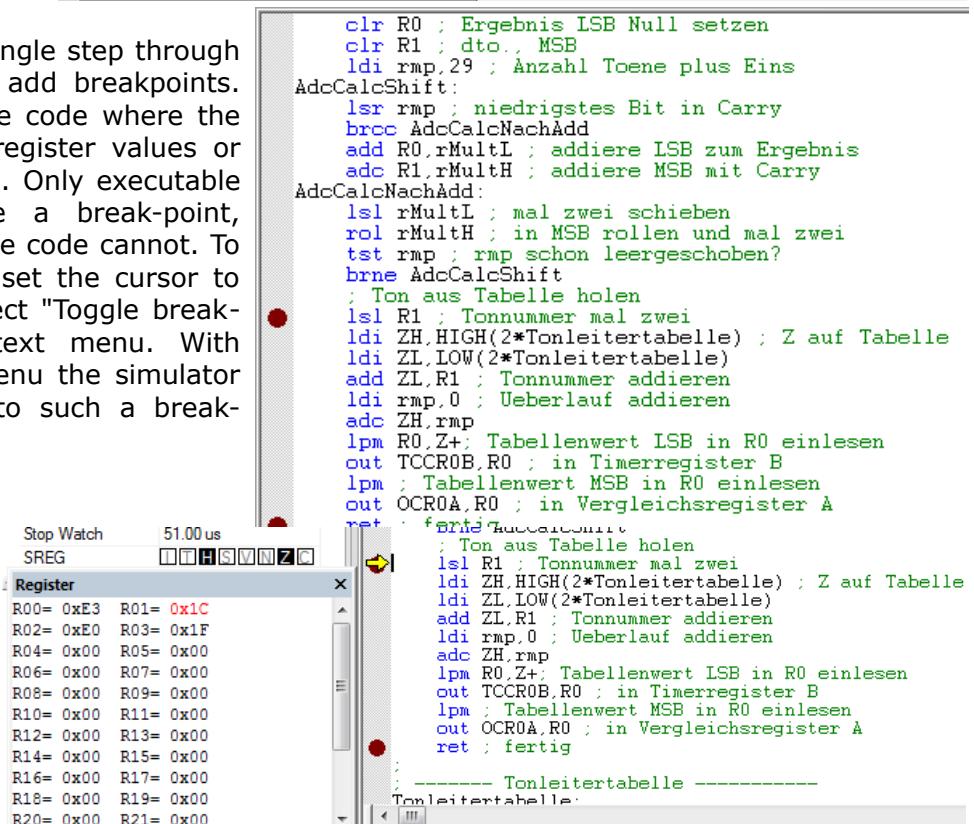


With the menu entry "Step into" (F11) the execution of this instruction is simulated. Executed was the source code "ldi rmp,0xFF", with rmp = R16. In the list of registers the changed value of R16 is marked in red, the program counter is now at 0001 and the cursor points to the next instruction.



If we do not want to single step through lengthy code, we can add breakpoints. Those are points in the code where the simulation stops and register values or ports can be examined. Only executable instructions can have a break-point, other lines in the source code cannot. To add a break-point we set the cursor to that code line and select "Toggle break-point" from the context menu. With "Run" in the debug menu the simulator runs until he comes to such a break-point. He then stops.

In our case of debugging the multiplication routine is of interest and the end of writing the table values to the timer ports. In the first case the register R1 holds 0x1C, which is the expected decimal 28.



This is the state of the timer when the second breakpoint is reached. The two ports that we wrote the gamut table values to, TCCR0B and OCR0A show the correct values from the gamut table for tone #28. Because we skipped the timer init procedure the other timer ports are incorrect, but our multiplication and timer write is correct.

Name	Address	Value	Bits
GTCCR	0x28 (0x48)	0x00	██████████
OCR0A	0x36 (0x56)	0x54	██████████
OCR0B	0x29 (0x49)	0x00	██████████
TCCR0A	0x2F (0x4F)	0x00	██████████
TCCR0B	0x33 (0x53)	0x01	██████████
TCNT0	0x32 (0x52)	0x04	██████████
TIFR0	0x38 (0x58)	0x00	██████████
TIMSK0	0x39 (0x59)	0x00	██████████

With those tools we can search and identify errors in our source code.

9.5 Playing music

9.5.1 Task

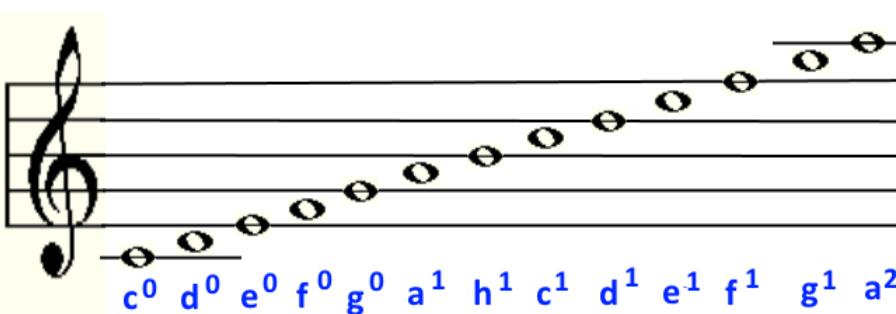
The controller shall play the Internationale when the key is pressed.

9.5.2 The melody to be played



This has to be played.

Völ-ker, hört die Sig-na - le!



Those are the notes to translate the melody to the gamut table. In order to translate it would be more convenient to not handle notes numbers but to have names associated with those numbers. To do this we add ".equ name=number" for each note. Those names start with an n,

then the tones a to g, followed by the octave 0 to 4. E.g. like that:

; Notes symbols	.equ na0 = 0	.equ nh0 = 1	.equ nc0 = 2	.equ nd0 = 3	.equ ne0 = 4	.equ nf0 = 5	.equ ng0 = 6	.equ na1 = 7	.equ nh1 = 8	.equ nc1 = 9	.equ nd1 = 10	.equ ne1 = 11	.equ nf1 = 12	.equ ng1 = 13	.equ na2 = 14	.equ nh2 = 15	.equ nc2 = 16	.equ nd2 = 17	.equ ne2 = 18	.equ nf2 = 19	.equ ng2 = 20	.equ na3 = 21	.equ nh3 = 22
-----------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------

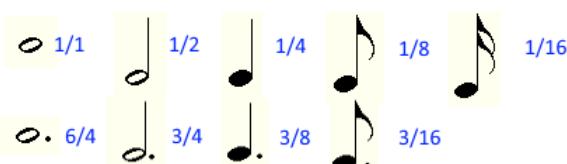
The original notation of notes, that works with small and large letters cannot be used in assembler because assembler does not know the difference between small and capital letters.

With these definitions the table for our melody goes like this:

Melody:
.db ne1,nd1,nc1,ng0,ne0,na1,nf0,0xFF

This can be conveniently handled. The trailing 0xFF signals that the melody ends here (if that would not be here the controller would interpret his whole flash storage as melody and would play that on and on).

9.5.3 Tone duration



The symbols used in music writing encode the duration over which a tone has to be played. We therefore need an additional information on tone duration for our melody. To encode this to our melody table we could add two bits to our notes

table, one means 1/8, two means 1/4, three means 3/8 and four means 1/2. Our melody would look like this:

```
.equ bLow = 1<<5 ; Duration encoding, low bit
.equ bHigh = 1<<6 ; Duration encoding, high bit
.equ d12 = bLow + bHigh ; Duration 1/2, both duration bits high
.equ d14 = bHigh ; Duration 1/4, upper duration bit high
.equ d38 = bLow ; Duration 3/8, lower duration bit high
.equ d18 = 0 ; Duration 1/8, neither upper nor lower bit high
Melody:
.db ne1+d38,nd1+d14,nc1+d12,ng0+d38,ne0+d18,na1+d12,nf0+d14,0xFF
```

In order to decode those notes we would program the following:

```
; Z points to melody table
ldi ZH,HIGH(2*Musik) ; MSB pointer for LPM
ldi ZL,LOW(2*Musik) ; dto., LSB
; Read note byte
lpm R17,Z ; Read first note from table to R17
; Check end of melody
cpi R17,0xFF ; End of melody signature?
breq MelodyOff ; Switch music off
; Decode duration of note
andi R17,0b01100000 ; Isolate bits 5 and 6
ldi R16,1 ; Number of octas
sbrc R17,5 ; Check bit 5
subi R16,-1 ; Increase by one (addi is not available)
sbrc R17,6 ; Check bit 6
subi R17,-2 ; Add two
; Convert note to timer prescaler and timer CTC
lpm R17,Z+ ; Again read note, inc pointer
andi R17,0b10011111 ; Clear duration bits
[Convert and play note for duration in R16]
MelodyOff:
[Switch off music output]
```

We can also encode the duration in an extra byte. This would ease note processing, but double the length of the melody table. With our simple and short melody we can afford this without risking flash memory shortages:

```
Melody: ; LSB: Note or FF, MSB: Duration in octa
.db ne1,3,nd1,2,nc1,4,ng0,3,ne0,1,na1,4,nf0,2,0xFF,0xFF
```

Now an additional 0xFF has to be added to the table to reach an even number of bytes in the table.

9.5.4 Tone pauses



Völ-ker, hört die Sig-na - le!

Note e¹ d¹ c¹ g⁰ e⁰ a¹ f⁰

Duration 3/8 1/4 1/2 3/8 1/8 1/2 1/4

Pause 1/8 1/8 1/8 1/8 1/8

End 1

To play notes means not only tones but also pauses in between. Not between the first and the second note of that melody, but in between any other notes. So we need not only an end signature but also a pause signal. We select 0xFE for that. Our table now looks like that:

```
Music: ; LSB: Note or FF, MSB: Duration in octa
.db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3,0xFE,1,ne0,1,0xFE,1,na1,4,0xFE,1,nf0,2,0xFF,0xFF
```

9.5.5 Duration to play different notes

From the controller point of view another problem arises. If we play different tones with frequencies between 440 and 7,040 cs/s the duration of each half wave is very different. A different number of half waves have to be absolved to come to the same tone duration. At 440 cs/s 880 CTC events have to be absolved to yield one second, but at 7,040 cs/s those have to be 14,080 CTC events long. The number of CTC events to be performed is reversely proportional to the frequency. We can resolve this with two opportunities:

- We divide 600.000 by the prescaler and the CTC value and get the number of CTC cycles, or
- we add this number to our gamut table and read that out with LPM.

As we are lazy and are not willing to divide 24 bit integers by 8 bit numbers in assembler and as we do not want the controller to do that lengthy and boring procedure each time he reads a note, we choose the second option. The C programmer sees no problem here and imports its floating point arithmetic library here, but changes to an Atxmega.

The formulation of the gamut table, now with the duration of the tone in one octa of a second, looks like this:

```
GamutTable_Duration:  
.DB 1<<CS01, 169, 110, 0 ; a #0  
.DB 1<<CS01, 151, 123, 0 ; h #1  
.DB 1<<CS01, 135, 138, 0 ; cis #2  
.DB 1<<CS01, 127, 146, 0 ; d #3  
.DB 1<<CS01, 113, 164, 0 ; e #4  
.DB 1<<CS01, 101, 184, 0 ; fis #5  
.DB 1<<CS01, 90, 206, 0 ; gis #6  
.DB 1<<CS01, 84, 221, 0 ; a' #7  
.DB 1<<CS01, 75, 247, 0 ; h' #8  
.DB 1<<CS01, 67, 20, 1 ; cis' #9  
.DB 1<<CS01, 63, 37, 1 ; d' #10  
.DB 1<<CS01, 56, 73, 1 ; e' #11  
.DB 1<<CS01, 50, 112, 1 ; fis' #12  
.DB 1<<CS01, 44, 161, 1 ; gis' #13  
.DB 1<<CS01, 42, 180, 1 ; A #14  
.DB 1<<CS01, 37, 237, 1 ; H #15  
.DB 1<<CS01, 33, 39, 2 ; CIS #16  
.DB 1<<CS01, 31, 74, 2 ; D #17  
.DB 1<<CS00, 226, 149, 2 ; E #18  
.DB 1<<CS00, 204, 220, 2 ; FIS #19  
.DB 1<<CS00, 181, 56, 3 ; GIS #20  
.DB 1<<CS00, 169, 114, 3 ; A' #21  
.DB 1<<CS00, 151, 219, 3 ; H' #22  
.DB 1<<CS00, 135, 79, 4 ; CIS' #23  
.DB 1<<CS00, 127, 148, 4 ; D' #24  
.DB 1<<CS00, 113, 36, 5 ; E' #25  
.DB 1<<CS00, 101, 191, 5 ; FIS' #26  
.DB 1<<CS00, 90, 112, 6 ; GIS' #27  
.DB 1<<CS00, 84, 229, 6 ; A'' #28  
; Pause for one octa of a second  
; .DB (1<<CS01) | (1<<CS00), 255, 18, 0 ; Pause #254
```

Our table now has four bytes per tone and we can start programming the melody.

9.5.6 Processing structure

To program this we need to

- decide on a key press event, if the melody is still processed. If not, the process has to be started by setting a start flag,
- read the next note from the melody table,
- to read the tone's prescaler and CTC values from the gamut table and to write these to the timer,
- to read the duration over which the note will be played from the gamut table, to read the duration factors (full notes, half notes, etc.) from the melody table and to write this to a 16 bit counter accordingly,
- switch on (in tones) or off (in pauses) the speaker pulse output mode,
- detect the end of the melody and if detected to switch off the speaker output, clear the timer and to re-enable key events.

Within interrupt service routines the following has to done:

- PCINT: To detect the polarity of the key input, if high then set a start flag.
- TC0-Compare-A-Int: Decrease the 16 bit counter, if zero then set an end flag.

Within the main program loop both flags are checked and the associated actions performed (start melody from scratch, output next note, etc.).

9.5.7 Program

This is the final program, the [source code is here](#).

```
;  
; *****  
; * To play a melody with an ATtiny13 *  
; * (C)2017 by www.avr-asm-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn13def.inc"  
.LIST  
;  
; ----- Register -----  
; free: R0 .. R14  
.def rSreg = R15 ; Save SREG  
.def rmp = R16 ; Multi purpose register  
.def rFlag = R17 ; Flag register  
    .equ bStart = 0 ; Start melody play  
    .equ bNote = 1 ; Play next note  
; free: R18 .. R23  
.def rCTrl = R24 ; 16 bit counter CTC events, LSB  
.def rCtrlh = R25 ; dto., MSB  
; used: X, XH:XL for duration calculation and to save Z  
; used: Y, YH:YL for octa duration  
; used: Z, ZH:ZL for reading from program memory, as melody pointer  
;  
; ----- Ports -----  
.equ pOut = PORTB ; Output port  
.equ pDir = DDRB ; Direction port  
.equ pInp = PINB ; Input port  
.equ bSpkD = DDB0 ; Speaker output pin  
.equ bKeyO = PORTB3 ; Pull up key input pin, write  
.equ bTasI = PINB3 ; Key input pin, read  
;  
; ----- Timing -----  
; Clock          = 1200000 cs/s  
; Prescaler      = 1, 8, 64  
; CTC TOP range = 0 .. 255  
; CTC divider range = 1 .. 256  
; Toggle divider = 2  
; Frequency range = 600 kcs/s to 36 cs/s  
;  
; ---- Reset- and Interrupt vectors ---  
.CSEG ; Assemble to the code segment  
.ORG 0 ; To the beginning  
    rjmp Start ; Reset-Vector, Init  
    reti ; INT0-Int, inactive  
    rjmp PcIntIsr ; PCINT-Int, active  
    reti ; TIM0_OVF, inactive  
    reti ; EE_RDY-Int, inactive  
    reti ; ANA_COMP-Int, inactive  
    rjmp TC0CAIsr ; TIM0_COMPA-Int, active  
    reti ; TIM0_COMPB-Int, inactive  
    reti ; WDT-Int, inactive  
    reti ; ADC-Int, inactive  
;  
; ---- Interrupt service routines ---  
PcIntIsr: ; PCINT on key events  
    in rSreg,SREG ; Save SREG  
    sbic pInp,bTasI ; Skip next if input is low  
    rjmp PcIntIsrRet ; Ready  
    brts PcIntIsrRet ; If T flag is set, skip  
    sbr rFlag,1<<bStart ; Set start flag  
PcIntIsrRet:  
    out SREG,rSreg ; Restore SREG  
    reti  
;  
TC0CAIsr: ; Timer CTC A Int  
    in rSreg,SREG ; Save SREG  
    sbiw rCTrl,1 ; Down-count 16 bit counter  
    brne TC0CAIsrRet ; not yet zero  
    sbr rFlag,1<<bNote ; Set flag for next note  
TC0CAIsrRet:  
    out SREG,rSreg ; Restore SREG
```

```

    reti
;

; ---- Program start, Init -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Stack pointer to SRAM end
    out SPL,rmp
    ; Clear T flag
    clt ; Set flag inactive
    ; Init and configure port-pins
    ldi rmp,1<<bSpkD ; Speaker output pin output
    out pDir,rmp ; to direction port
    ldi rmp,1<<bKeyO ; Pull up on key port pin
    out pOut,rmp ; to output port
    ; Start timer as CTC
    ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear OC0A, CTC-A
    out TCCR0A,rmp ; to timer control port A
    ; Prescaler, timer start and Int in start routine
    ; PCINT for key input events
    ldi rmp,1<<PCINT3 ; Anable PB3 Int
    out PCMSK,rmp ; in PCINT mask port
    ldi rmp,1<<PCIE ; Enable PCINT
    out GIMSK,rmp ; in Interrupt mask port
    ; Enable sleep
    ldi rmp,1<<SE ; Sleep mode idle
    out MCUCR,rmp ; in MCU control port
    ; Enable interrupts
    sei
; ---- Main program loop -----
Loop:
    sleep ; Go to sleep
    nop ; Wake up
    sbrc rFlag,bStart ; Skip next if start flag zero
    rcall MelodyStart ; Start melody output
    sbrc rFlag,bNote ; Skip next if no note to be played
    rcall PlayNote ; Play next note
    rjmp Loop ; Go back to sleep again
;
; ----- Flag handling routines -----
MelodyStart: ; Start melody output
    cbr rFlag,1<<bStart ; Clear flag
    set ; Set T flag
    ldi ZH,HIGH(2*Melody) ; Pointer to melody
    ldi ZL,LOW(2*Melody)
    rcall PlayNote ; Output next note
    ldi rmp,1<<OCIE0A ; Enable CTC ints
    out TIMSK0,rmp ; to timer interrupt mask
    ret
;
PlayNote: ; Output next note
    cbr rFlag,1<<bNote ; Clear flag
    rcall PlayNext ; Output next note
    ret
;
PlayNext: ; Play the note to which Z points
    lpm rmp,Z+ ; Read note from melody table
    cpi rmp,0xFF ; Check melody end
    brne PlayNext1 ; Not at end
    ; Melody is over
    ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear OC0A, CTC-A
    out TCCR0A,rmp ; to timer control port A
    clr rmp ; Clear timer interrupts
    out TIMSK0,rmp ; in TCO Interrupt mask port
    pop rmp ; Remove call address from stack
    pop rmp
    clt ; Clear T flag
    ret
PlayNext1: ; Not at end
    cpi rmp,0xFE ; Pause?
    brne PlayNext2 ; No
    ; Pause, output off
    ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear OC0A, CTC-A
    out TCCR0A,rmp ; to timer control port A
    ldi rmp,(1<<CS01)|(1<<CS00) ; Prescaler = 64
    out TCCR0B,rmp ; to timer control port B
    ldi rmp,255 ; CTC to largest value
    ldi rCtrlL,18 ; Counter to one eights

```

```

ldi rCtrH,0
lpm R16,Z+ ; Overread duration byte
reti
PlayNext2: ; Normal note
    mov XH,ZH ; Save melody pointer
    mov XL,ZL
    ldi ZH,HIGH(2*GamutTable_Duration) ; Pointer to gamut table
    ldi ZL,LOW(2*GamutTable_Duration)
    lsl rmp ; Note number * 2
    lsl rmp ; * 4
    add ZL,rmp ; add to pointer
    ldi rmp,0 ; Carry adder
    adc ZH,rmp ; Add with carry
    lpm rmp,Z+ ; Read prescaler
    out TCCR0B,rmp ; to timer control port B
    lpm rmp,Z+ ; Read CTC value
    out OCR0A,rmp ; to compare match A port
    lpm YL,Z+ ; Read octa duration to Y
    lpm YH,Z+
    mov ZH,XH ; Restore pointer to melody
    mov ZL,XL
    lpm rmp,Z+ ; Read duration byte
    mov XH,YH ; Copy single duration
    mov XL,YL
PlayNext3:
    dec rmp ; Decrease duration byte
    breq PlayNext4 ; final
    add XL,YL ; Add octa duration, LSB
    adc XH,YH ; dto. MSB plus carry
    rjmp PlayNext3
PlayNext4:
    mov rCtrL,XL ; copy to counter, LSB
    mov rCtrH,XH ; dto., MSB
    ldi rmp,(1<<COM0A0)|(1<<WGM01) ; Toggle OC0A, CTC-A
    out TCCR0A,rmp ; to timer control port A
    ret
;
; Gamut table with duration
GamutTable_Duration:
.DB 1<<CS01, 169, 110, 0 ; a #0
.DB 1<<CS01, 151, 123, 0 ; h #1
.DB 1<<CS01, 135, 138, 0 ; cis #2
.DB 1<<CS01, 127, 146, 0 ; d #3
.DB 1<<CS01, 113, 164, 0 ; e #4
.DB 1<<CS01, 101, 184, 0 ; fis #5
.DB 1<<CS01, 90, 206, 0 ; gis #6
.DB 1<<CS01, 84, 221, 0 ; a' #7
.DB 1<<CS01, 75, 247, 0 ; h' #8
.DB 1<<CS01, 67, 20, 1 ; cis' #9
.DB 1<<CS01, 63, 37, 1 ; d' #10
.DB 1<<CS01, 56, 73, 1 ; e' #11
.DB 1<<CS01, 50, 112, 1 ; fis' #12
.DB 1<<CS01, 44, 161, 1 ; gis' #13
.DB 1<<CS01, 42, 180, 1 ; A #14
.DB 1<<CS01, 37, 237, 1 ; H #15
.DB 1<<CS01, 33, 39, 2 ; CIS #16
.DB 1<<CS01, 31, 74, 2 ; D #17
.DB 1<<CS00, 226, 149, 2 ; E #18
.DB 1<<CS00, 204, 220, 2 ; FIS #19
.DB 1<<CS00, 181, 56, 3 ; GIS #20
.DB 1<<CS00, 169, 114, 3 ; A' #21
.DB 1<<CS00, 151, 219, 3 ; H' #22
.DB 1<<CS00, 135, 79, 4 ; CIS' #23
.DB 1<<CS00, 127, 148, 4 ; D' #24
.DB 1<<CS00, 113, 36, 5 ; E' #25
.DB 1<<CS00, 101, 191, 5 ; FIS' #26
.DB 1<<CS00, 90, 112, 6 ; GIS' #27
.DB 1<<CS00, 84, 229, 6 ; A'' #28
;
; ---- Notes symbols -----
;
.equ na0 = 0
.equ nh0 = 1
.equ nc0 = 2
.equ nd0 = 3
.equ ne0 = 4
.equ nf0 = 5

```

```

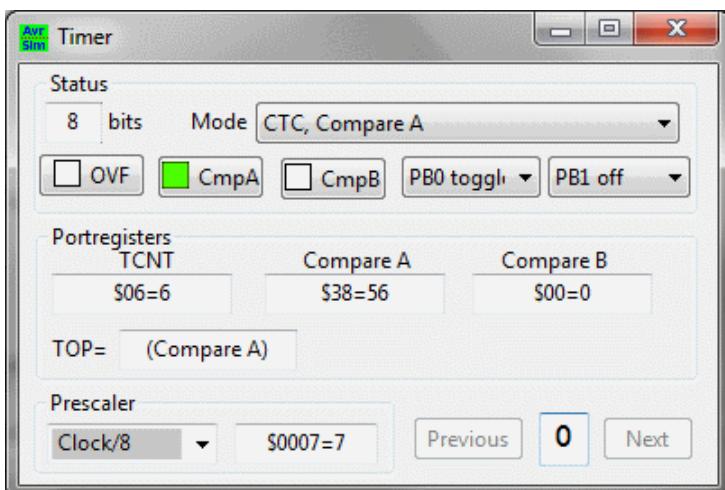
.equ ng0 = 6
.equ na1 = 7
.equ nh1 = 8
.equ nc1 = 9
.equ nd1 = 10
.equ nel = 11
.equ nfl = 12
.equ ngl = 13
.equ nA2 = 14
.equ nH2 = 15
.equ nC2 = 16
.equ nD2 = 17
.equ nE2 = 18
.equ nF2 = 19
.equ nG2 = 20
.equ nA3 = 21
.equ nH3 = 22
.equ nC3 = 23
.equ nD3 = 24
.equ nE3 = 25
.equ nF3 = 26
.equ nG3 = 27
.equ nA4 = 28
;
; ----- Melody -----
Melody: ; LSB: Note or FF, MSB: Duration in octa
;   Völ- ker     hört      die
.db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3,0xFE,1
;   Sig-       na-       le!
.db ne0,1,0xFE,1,na1,4,0xFE,1,nf0,2,0xFF,0xFF
;
;
; End of source code
;

```

The source code uses one new instruction in a somehow strange way. It is the POP instruction. This copies the upmost byte that was pushed onto the stack, e.g. by a call to a subroutine, and increases the stack pointer by one. If one does that two times and ignores the byte content, the calling address is removed from the stack. A return now jumps to the previous calling address and ignores the last call. Make sure in this case that this is in fact the calling address of the previous call and not a pushed data byte or something else.



9.5.8 Simulating execution



Simulation uses [avr_sim](#) to check execution times of the first two notes of the melody played.

The first note of the melody has been loaded. Timer/counter 0 is configured as CTC with Compare A as TOP value, which is at 56. The prescaler is 8 and the execution time of each CTC cycle is

$$t_{CTC} = 8 * (56 + 1) / 1.2 = 380 \mu s$$

The compare match interrupt is enabled and portpin PB0 is toggled each time CTC occurs. This generates a tone with

$$f_1 = 1,000,000 / 380 / 2 = 1,316 \text{ Hz}$$

The first note in the melody table is ne1:

```
252: Melody: ; LSB: Note or FF, MSB: Duration in octa
253: ; Voel- ker hoert die
254: .db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3,0xFE,1
      0000A7 030B 020A 01FE 0409
```

ne1 is defined as

```
232: .equ ne1 = 11
```

being note 11, which is corresponding to the following entry in the gamut table:

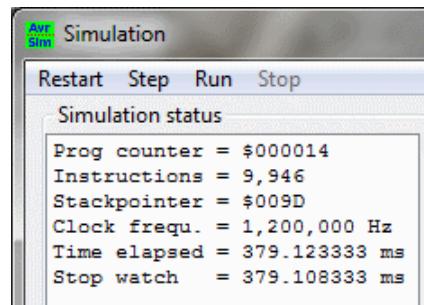
```
200: .DB 1<<CS01, 56, 73, 1 ; e' #11
      000083 3802 0149
```

With CS01 in TCCR0B the prescaler is 8, the 56 defines the compare match A value. 73 and 1 are the durations of that note, which is $1*256 + 73 = 329$. The 3 following the ne1 in the melody table says "increase duration by a factor of 3, which yields 987 CTC cycles (or 0x03DB) and a duration of $987 * 380 \mu\text{s} = 375 \text{ ms}$.

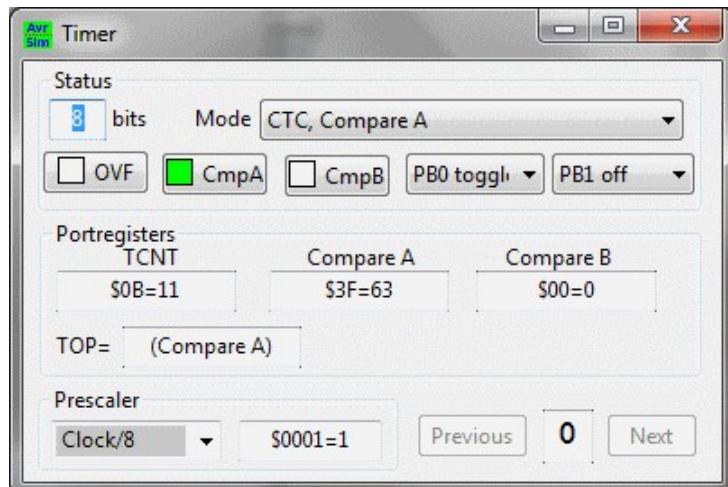
Register							
Reg	+0	+1	+2	+3	+4	+5	+6
R0	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00
R16	04	00	00	00	00	00	00
R24	DB	03	DB	03	49	01	50
							01

The duration 0x03DB is in the double register R25:R24 and will be decreased in the interrupt service routine of the compare match A int.

Z is pointing to the next note in the flash (0x0150), which points to address 0x00A8 and is multiplied by 2. This is correct (see the above listing of the melody).



This is the time when the registers R25:R24 reach zero and the next note will have to be played. The 379 ms are nearly exact (should be 375).



The second note of the melody has been loaded to TC0. The prescaler is again 8, but the compare match A value is now 63. That corresponds to

$$f_2 = 1,200,000 / 8 / (63 + 1) / 2 = 1,171 \text{ Hz}$$

The note to be played is nd1:

```

253: ; Voel- ker      hoert      die
254: .db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3,0xFE,1

```

which is note #10:

```
231: .equ nd1 = 10
```

And note #10 is d':

```
199: .DB 1<<CS01, 63, 37, 1 ; d' #10
000081 3F02 0125
```

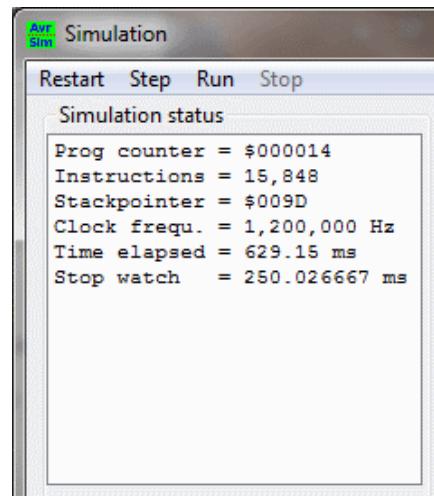
So the prescaler and the compare match A are fine.

The duration of the note is now $(1*256 + 37) * 2 = 586$ or 0x024A CTC cycles or $586 * 380 \mu\text{s} = 223 \text{ ms}$.

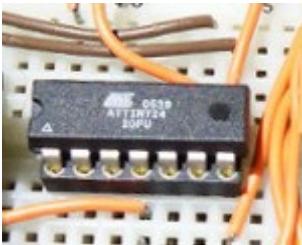
Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	40
R16	42	00	00	00	00	00	00	00
R24	4A	02	4A	02	25	01	52	01

The counter value R25:R24 is fine, the register Z points to the third note.
250 ms have elapsed, just slightly above the 223 ms that were desired and calculated.

The third note is a pause (0xFE). The only difference between a note and a pause is that OCR0A is not toggled but cleared, switching PB0 output to be always low.



Simulation is a powerful tool to debug even complex procedures step-by-step and to measure execution times exactly. Use it to test your own designs and to verify that the controller really does what you thought that he should do and in a timely manner.



Lecture 10: A Lcd display on an ATtiny24

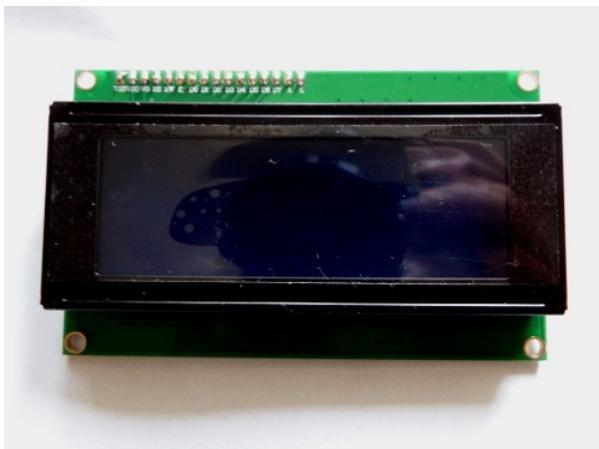
To operate an LCD on a controller we need some more pins than an ATtiny13 has. We therefore switch to the next larger pin package and to an ATtiny24. Two methods can be used to communicate with an LCD: one-way with fixed delay loops and two-way with reading and processing the busy flag.

10.0 Overview

1. [Introduction to LCD displays](#)
2. [Introduction to ATtiny24](#)
3. [Hardware, components and mounting](#)
4. [Access using delays loops](#)
5. [Access using the busy flag](#)
6. [Display of own characters](#)

10.1 Introduction to LCD displays

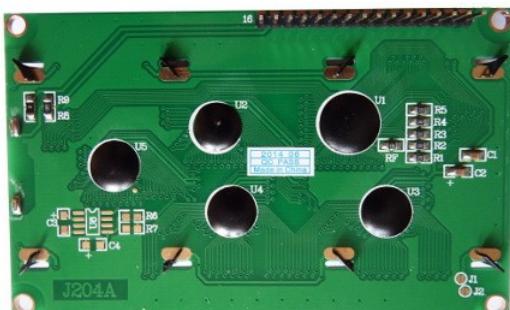
10.1.1 General on LCD displays



There is a large number of colors and types of LCDs available. Green, Yellow, gray and blue. Every taste can be met. There are text and graphics displays. This lecture works with text displays because they have a simple interface and programming mode and they cover nearly all applications.

Text displays work with a pixel matrix of 5 by 8, which is optimal for character recognition even from some distance.

The advantage of LCD displays over e.g. Seven-Segment-Displays is their low current consumption of 1 to 2 mA. For battery operation this is optimal. To have the same contrast 7-segment-displays use at least the 20-fold of current. Additionally, displaying text characters with 7-segment is horrible. And furthermore: to display 8, 16 or 24 characters on a 7-segment basis is a cabling and hardware grave. An LCD has a simple hardware interface, no matter how many characters have to be displayed.

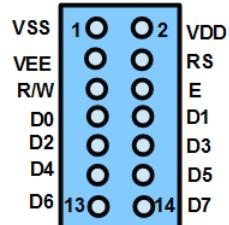
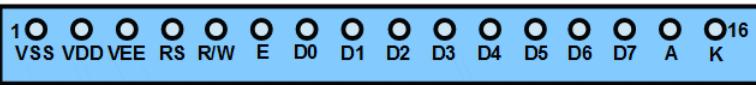
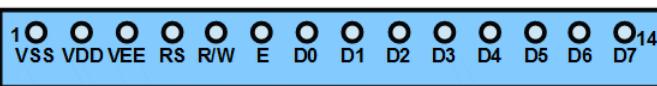


Available are a number of displays covering a different number of lines and characters per line. Single line displays with eight characters are ideal for small measuring devices, more characters would be adverse. Two line LCDs with 8, 16

or 20 characters per line can display a large amount of information. If you need more, you select a four line LCD with 20 or 24 characters. Like we do in this course.

10.1.2 Interfaces of LCD displays

The pins of a LCD are more or less the same for all displays, nearly all use the same row of pins. A few others do not care about this norm, e.g. 18-pin devices.



Those pins mean the following:

Pin	Name	Description	Pin	Name	Description
1	VSS	Operating voltage, minus	9	D2	Data bit 2
2	VDD	Operating voltage, plus, 3 or 5 Volt	10	D3	Data bit 3
3	VEE	Contrast regulation voltage	11	D4	Data bit 4
4	RS	Register Select, 0=Control, 1=Data	12	D5	Data bit 5
5	R/W	Read/Write, 0=Write to LCD, 1=Read from LCD	13	D6	Data bit 6
6	E	Enable, 0=inactive, 1=active/Read/Write	14	D7	Data bit 7
7	D0	Data bit 0	15	A	Backlight Anode
8	D1	Data bit 1	16	K	Backlight Cathode

The connector with two rows is usable for flat cable. The data sheets for the display holds definitive information on the pin configuration.

The regulation voltage for contrast is adjusted with a trim potentiometer that divides the operating voltage. The potentiometer is adjusted to have the optimal display contrast. If this pin is tied to zero Volt or to the operating voltage, no characters can be seen on the display.

In most of the displays the character positions can be identified if the operating voltage is applied without any actions of the controller. In case of multiple lines single lines will not show character positions. With that one can see if the operating voltage is correctly attached, that the contrast regulation works fine and that the display is alive. If not, and if certain chips on the display get hot or even explode: your operating voltage has been applied reversely. Then transfer the display to your local electronics recycler.

10.1.2.1 8 bit interface

When the operating voltage is applied LCDs are going to the default of communication in 8 bit mode. Every read and write transfers eight data bits. All data bit inputs are by default on high level.

10.1.2.2 4 bit interface

With a special command LCDs can change to the 4 bit communication mode. In this mode only the four upper data bits D4 to D7 are used. Each write to the display now has to be made in two portions, each read includes two read cycles. First the upper four bits, then the lower four bits are read or written.

10.1.3 Controlling LCD devices

10.1.3.1 Init phase

The following procedure initializes a LCD, it should be executed after the operating voltage is stable. In all cases the RS pin has to be held low. Transfer of the data occurs by activating the E pin for at least 1 μ s duration.

"x" means that this bit is ignored and that it does not matter if it is high or low.

- Following the switching on of the operating voltage of the display the LCD controller needs some time to start up. Before this has been finished no communication is possible. The time over which this internal initiation takes place varies from display to display. 50 ms is in any case sufficient.
- After waiting for start-up the Function Set is executed. The binary code for that is 0b001L.NFxx. The single bits in that mean:
 - L=1 if an 8 bit interface is attached.
 - N defines the number of lines, N=0 is a single line, N=1 defines multiple lines.
 - F selects the character format, 0 means a 5-by-8 character set.
- If a 4 bit interface is attached and selected, then sending of 0b0011.xxxx for three times in a row switches the LCD safely to the eight bit mode. After that the switching to four bit mode with 0b0010.xxxx is performed in 8 bit mode (no splitting to upper and lower nibble). After having done this, the N and F can be set by sending 0b0010.NFxx, now in 4 bit split mode (0b0010 first, then 0bNFxx).
- Switching modes requires slightly longer than 1 ms, waiting for 5 ms between those commands is sufficient.
- The following commands require approximately 40 μ s. From now on the busy flag can be read from the LCD alternatively. To do that the data port that drive the LCDs data lines has to be configured as input (clear the direction bits), the R/W input of the LCD has to be set high. By activating the E input of the LCD for at least one μ s long this outputs its busy flag on bit 7. In 4 bit mode a second activation of E is necessary. If the busy flag is zero, the previous command has been executed and the LCD is listening for the next command.
- The following command is 0b0000.DCBx.
 - D=0 switches the LCD off, D=1 on.
 - C=0 switches the output cursor off, C=1 on.
 - B=0 switches blinking at the cursor position off, B=1 on.
- The command 0b0000.0001 clears the display.
- The command 0b0000.001S switches the auto-increment of the display address on (I=1) or off (I=0). S=1 shifts the content of the display, S=0 overwrites the content.
- With the command 0b0000.001x the address is set to character 1 on line 1 (home position) and any shifting is cleared (if S=1). This command requires more than 1 ms, 5 ms are sufficient.

With that the initiation of the LCD is complete and we can output characters.

10.1.3.2 Data output

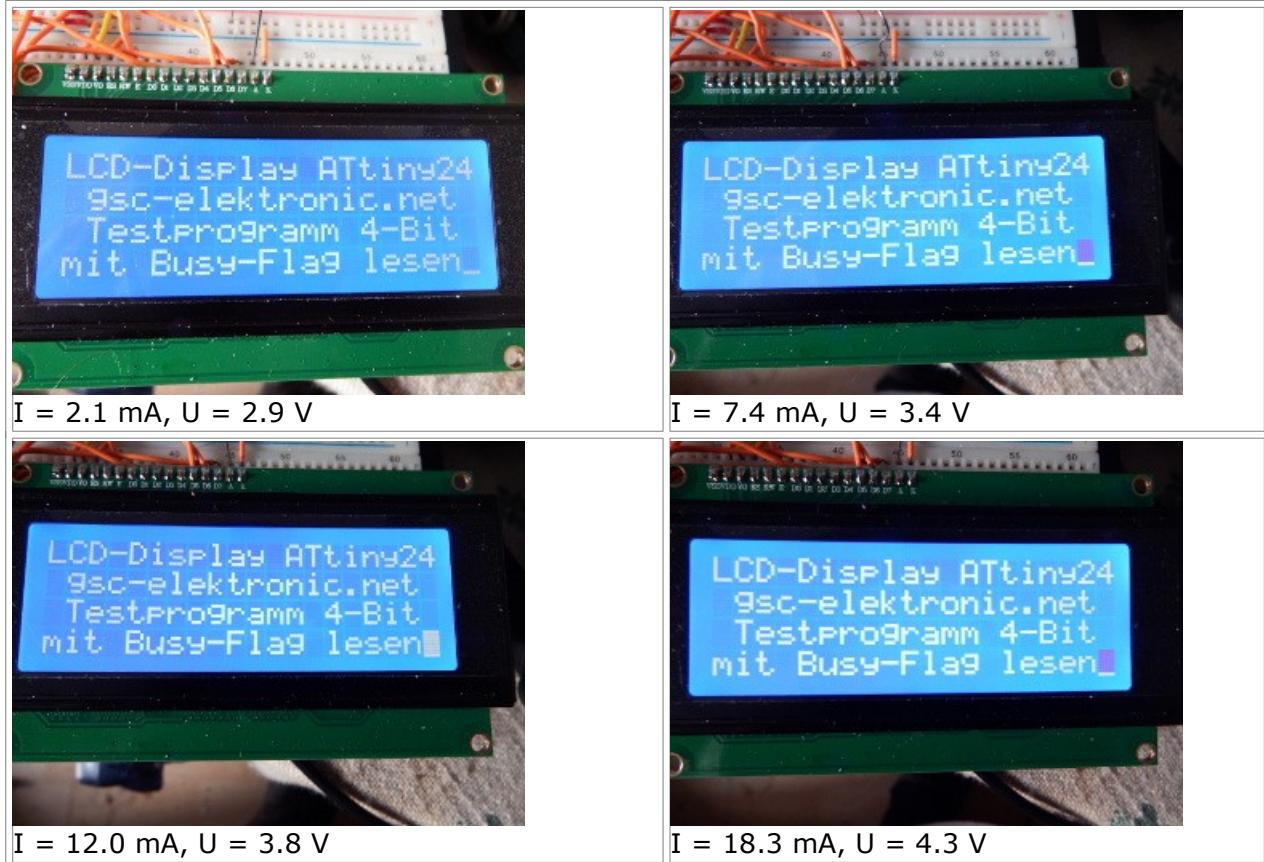
To send data to the LCD the RS input of the LCD must be set to high level and the R/W input to low level. The subsequently written characters occur one after the other on the LCD if Auto-Indent is selected (I=1). To change the address of the display the following numbers have to be written to the display with RS=0 and R/W=0 (if multiple lines N=1 are selected and the display has N characters per line). "Column" is a number between 1 and the number of characters per line (e.g. 1 .. 20).

- Line 1: 0x80 + (Column - 1),
- Line 2: 0xC0 + (Column - 1),
- Line 3: 0x80 + N + (Column - 1),
- Line 4: 0xC0 + N + (Column - 1).

10.1.4 Backlight of LCDs

LCD can be better read (especially in the dark) and are nicer if they have a back-light. To switch this on, the cathode pin is tied to the negative operating voltage and current is fed to the anode input. The current is specified in the data sheet for the LCD. For the 4 line LCD used here 70 mA are specified.

The following pictures demonstrate that above 5 mA no visible effect occurs.

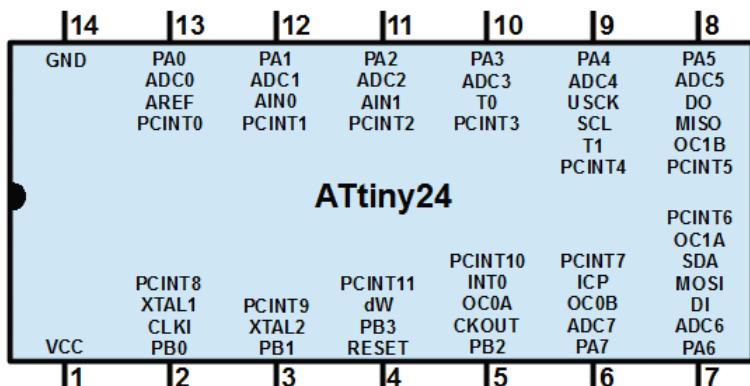


The effect of the current for the battery life is higher than the visibility and the optical effect.

10.2 Introduction to the ATtiny24

The ATtiny24 is a 14 pin controller type. As the multiple pin functions show, he not only has more pins but also more internal hardware than an ATtiny13.

- Port A is available with all eight bits, additionally three bits of Port B are accessible. If ISP is unnecessary the RESET input can be used for other purposes.
- Eight pins can be used as ADC0 to ADC7 inputs.
- Behind OC0A and OC0B an 8 bit timer is working, behind OC1A and OC1B a 16 bit timer. All four pins can be used as clock or PWM outputs.
- An external xtal can be attached to pins 2 and 3 and enabled by setting fuses.
- All pins can be monitored for level changes and can trigger two PCINTs (0 .. 7, 8 .. 10/11).
- Of course, via the pins USCK, MISO, MOSI and RESET ISP programming can be performed.
- The ATtiny24 has an internal 8 Mcs/s RC oscillator. With the internal clock divider this is divided by eight if the DIV8 fuse is set (default), so that a clock of 1 Mcs/s results. By clearing the DIV8 fuse it can be set to higher clock speeds, by writing the CLKPR port lower speeds can be selected, as already shown for the ATtiny13.



If you run out of pins with an ATtiny13, you can change to this type. The multiple functions per pin demonstrate how optimal controller selection should work: one has to have a clear idea about the hardware components that are necessary, what has to be available as an external pin, which port-pins have to be accessible in a row (in our case the four data bits of the LCD). Conflicting multiple functions of pins are then to be avoided. This leads us to optimal type selection. The C programmer knows only one criterion: has the controller enough flash memory to host all his libraries?

If the overview on the necessary hardware is lacking, one tends to over-dimension the controller type. If you decide to use a smaller type, and if you need an additional OC0A or OC1B output pin then, not only a few changes in the interrupt vector table will result. It could well be that the whole hard- and software design has to be changed. That is a main issue when planning such a project: more consequent planning avoids later changes. If you start with an Arduino, your elephant in a porcelain shop prevents you from such hazzle of learning optimal design.

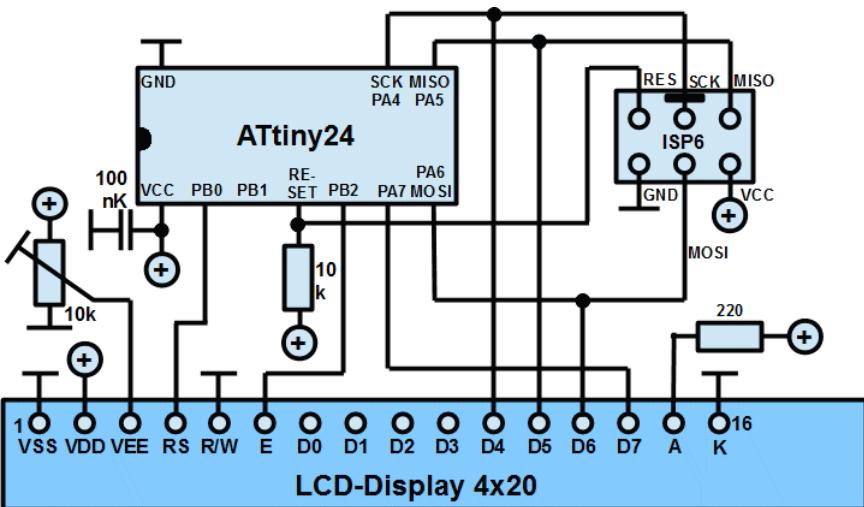
The theme multiple pin functions has several aspects. As we want to use ISP, the pins USCK, MOSU and MISO have to be available for that. As the LCD's E input allows to free the four data bits for other uses (just by holding E low and R/W on write), no conflict results when using those pins for ISP as well as as LCD data pins. The use of the USCK pin for measuring analogue voltages or for driving a motor would rise several questions.

10.3 Hardware, components and mounting

10.3.1 Scheme

The scheme shows how the LCD is controlled by the ATtiny24:

- The pins PB0 (LCD-RS) and PB2 (LCD-E) drive the control pins of the LCD, the data bus (upper nibble, nibble = 4 bits) is controlled by the upper nibble of port A. The R/W input of the LCD is tied to low, the LCD is only used to write to it and not to read from it.
- The ISP is attached to the respective pins, which are also used as LCD data bus.
- The back-light of the LCD is driven via a $220\ \Omega$ resistor with a small but sufficient current.



10.3.2 Components

10.3.2.1 LCD display

First we solder a 16 pin pin header to the LCD so that this fits into the breadboard. If you use a different LCD header you'll have to develop a solution for that.



10.3.2.2 ATtiny24

This is the 14 pin ATtiny24.



10.3.2.3 Socket 14 pin

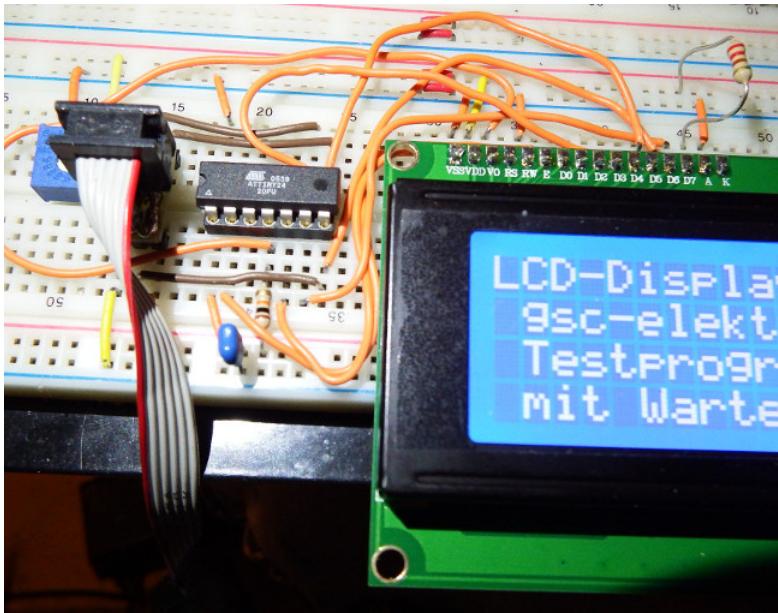
The controller fits into that 14 pin socket.



10.3.2.4 Trimmer

This trim potentiometer is for adjusting the contrast of the LCD. The two pins in the foreground of the right picture are the endpoints of the resistor layer, the middle pin in the background is the slider.

10.3.3 Mounting



This is how the Tiny24 and the LCD is mounted.

A hint on such breadboards: the contacts of the board are subject to aging. Especially if you leave such mighty pin headers in the holes over a longer period of time. If you use a very simple breadboard without a metal surface below those contacts are driven out, contact stability is very small then. Short pins, as that of an IC socket, do not fit well then. If that concerns the operating voltage, it can ruin a controller. If you are suspicious that aging might be a concern, let your measuring device test contact. And finally order your next breadboard.

10.3.4 Alternative Mounting: The ATtiny24 LCD experimental board

In order to ease mounting for this and the following experiments you can also use the ATtiny24 LCD experimental board described [here](#). All the necessary components (ATtiny24, LCD, trim potentiometer, etc.) are mounted on a small single-sided PCB. The board is equipped with a six-pin ISP interface to program the ATtiny24 and another six-pin plug to interface the PCB with the breadboard.

All software provided here can be used without any changes on this board.

The experimental board can be used with any type of LCD in respect to sizes (lines, characters per line). A software tool (as assembler include file) to accommodate source code to those different sizes and two demonstrative appliances of this include code round up this versatile experimental board, so that it can be used for lots of own experiments, too.

[Home](#) [Top](#) [LCD](#) [ATtiny24](#) [Hardware](#) [Wait mode](#) [Busy mode](#) [Characters](#)

10.4 Controlling a LCD with delay loops

10.4.1 Timing requirements and construction of delay loops

The procedure is to write a command or a character and then to delay further execution in a loop.

The ATtiny24 runs with an internal clock of 1 Mcs/s by default, each clock cycle requires one μs .

The following delay times are necessary to control an LCD:

- Initiation: 50 ms,
- Clearing the display: 5 ms,
- Character write, simple control commands: 40 μs .

A 16 bit wait loop fits all those needs. The code formulation:

```

ldi ZH,HIGH(n)
ldi ZL,LOW(n)
LcdWtLoop:
    sbiw ZL,1
    brne LcdWtLoop

```

This delays execution by

- two clock cycles for the two LDI instructions,
- four clock cycles multiplied by (n-1) for the loop,
- three clock cycles for the last loop cycle.

Therefore the formula is $nClock = 2 + 4 * (n-1) + 3 = 4*n + 1$ or $n = (nClock - 1) / 4$. There is some additional overhead when you use the delay loop in practice:

- three clock cycles for an RCALL instruction to jump to the loop,
- four clock cycles for a RET instruction at the end,
- if the register pair Z has to be saved first and restored after using it, two PUSH and two POP with two clock cycles each (8 clock cycles in total) are added to the total.

The formulas then are

$$nClock = 4*n + 16 \text{ or } nClock = 4*(n+4)$$

and

$$n = (nClock-4)/4.$$

If the delay time is 50,000 μ s and the clock frequency is 1 Mcs/s, the number of clock cycles $nClock = 50,000$ and n is = 12,499. For 40 μ s n is 9. So the 16 bit loop covers all LCD delay times.

The whole routine then is:

```

; rcall wait50ms ; + 3 clocks
.set n = 12499 ; N for 50 ms delay
LcdWt50ms:
    push ZH ; + 2 clocks
    push ZL ; + 2 clocks
    ldi ZH,HIGH(n) ; + 1 clock
    ldi ZL,LOW(n) ; + 1 clock
    rjmp LcdWtN ; + 2 clocks
; In total: 11 clock cycles
; And the wait routine:
LcdWtN:
    sbiw ZL,1 ; + 2 clocks
    brne warте ; +1/2 clock(s)
    pop ZL ; + 2 clocks
    pop ZH ; + 2 clocks
    ret ; + 4 clocks
; In total: (n-1)*4 + 3 + 8 = 4*n + 7
; Complete: nClock = 4*n + 18 cycles

```

The complete formula to calculate n for 50 ms at 1 Mcs/s is

$$n = (50000-18+2)/4 = 12496$$

The adder "+2" is for rounding the division result.

The directive ".set" defines a variable, which can be changed later on (e.g. .set N = 1246 for 5 ms and .set N = 6 for 40 μ s). The directive .equ does not allow to change a constant, .set does.

10.4.2 Task

On the LCD a message with four lines (alternative: one, two or three lines) with 20 (alternative: eight, sixteen, twentyfour) characters shall be displayed. Execution times shall be con-

trolled with delay loops.

10.4.3 Program

This is the program, [this is the source code for download](#). It is a linear program because initiation and writing to the LCD does not require interrupts. In fact, the necessary delay loops would not fit to a interrupt service routine. That is why all LCD read and write operations to/from the LCD has to be done in the main program loop and outside of interrupt service routines.

```
;  
; *****  
; * LCD-Display 4*20 on an ATtiny24, 4 bit interface *  
; * (C)2017 by www.avr-asm-tutorial.net  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
;  
; ----- Hardware -----  
;  
;  
; + 5 V o--|VCC      GND|--o 0 V  
; |          |  
; LCD-RS o--|PB0      |--o  
; |          |  
;(LCD-RW) o--|          |--o  
; |          |  
RESET o--|RES      |--o  
; |          |  
LCD-E o--|PB2      |--o  
; |          |  
LCD-D7 o--|PA7      PA4|--o LCD-D4  
; |          |  
LCD-D6 o--|PA6      PA5|--o LCD-D5  
; |          |  
;  
;  
; ----- Ports, Port pins -----  
;  
; LCD control port  
.equ pLcdCO  = PORTB ; LCD control port output  
.equ pLcdCR  = DDRB ; LCD control port direction  
.equ bLcdCOE = PORTB2 ; LCD enable pin output  
.equ bLcdCRE = DDB2 ; LCD enable pin direction  
.equ bLcdCORS = PORTB0 ; LCD RS pin output  
.equ bLcdCRRS = DDB0 ; LCD RS pin direction  
.equ bLcdCORW = PORTB1 ; LCD RW pin output  
.equ bLcdCRRW = DDB1 ; LCD RW pin direction  
;  
; LCD data port  
.equ pLcdDO  = PORTA ; LCD data port output  
.equ pLcdDI  = PINA ; LCD data port input  
.equ pLcdDR  = DDRA ; LCD data port direction  
.equ mLcdDR  = 0xF0 ; LCD data port direction mask  
;  
;  
; ----- Registers -----  
;  
; free: R0 .. R15  
.def rmp = R16 ; Multi purpose register  
.def rmo = R17 ; Second multi purpose register  
.def rLine = R18 ; Line counter LCD  
;  
; free: R19 .. R29  
;  
; used: R31:R30, ZH:ZL, for counting and pointing  
;  
;  
; ----- Constants -----  
.equ Clock = 1000000 ; Default clock frequency  
;  
;  
; ----- Program start, init -----  
.CSEG ; Code Segment  
.ORG 0 ; Start at 0  
    ; Init stack for subroutines  
    ldi rmp,LOW(RAMEND) ; Init stack  
    out SPL,rmp ; to stack pointer
```

```

; Init port outputs
ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW) ; LCD
out pLcdCR,rmp ; control port outputs
clr rmp ; Outputs off
out pLcdCO,rmp ; to control port
ldi rmp,mLcdDR ; Data port output mask
out pLcdDR,rmp ; to output
; Init LCD
rcall LcdInit
; Output text on LCD
ldi ZH,HIGH(2*Texttable)
ldi ZL,LOW(2*Texttable)
rcall LcdText ; Text in flash from ZH:ZL
; Sleep enable
ldi rmp,1<<SE
out MCUCR,rmp

Loop:
    sleep ; to sleep
    rjmp Loop
;

; Output text on LCD
Texttable:
.db "LCD display ATTiny24",0x0D,0xFF ; Line 1
.db "avr-asm-tutorial.net",0x0D,0xFF ; Line 2
.db " Test program 4 bit",0x0D ; Line 3
.db " with wait loops",0x00 ; Line 4
;
; ----- LCD control Init -----
LcdInit:
    ; Wait 50 ms until LCD has initiated
    rcall Wait50ms ; Delay by 50 ms
    ; Set to 8 bit mode (three times)
    ldi rmp,0x30 ; 8 bit mode
    rcall LcdC8Byte ; Write in 8 bit mode
    rcall Wait5ms ; Wait 5 ms
    ldi rmp,0x30 ; Second repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ldi rmp,0x30 ; Third repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ; Switch to 4 bit mode
    ldi rmp,0x20 ; 4 bit mode
    rcall LcdC4Byte
    rcall Wait5ms
    ; Function set LCD
    ldi rmp,0x28 ; 4 bit mode, 4 lines, 5*7
    rcall LcdC4Byte
    rcall Wait5ms
    ldi rmp,0x0F ; Display on, blink
    rcall LcdC4Byte
    rcall Wait5ms
    ldi rmp,0x01 ; Clear display
    rcall LcdC4Byte
    rcall Wait5ms
    ldi rmp,0x06 ; Auto-indent
    rcall LcdC4Byte
    rjmp Wait40us

;
; Output of text on the LCD
; Z points to text in flash memory
LcdText:
    clr rLine ; Line counter
LcdText1:
    lpm rmp,Z+ ; Read character from flash
    cpi rmp,0 ; End of Output?
    breq LcdTextRet ; no
    cpi rmp,0xFF ; Dummy character?
    breq LcdText1 ; yes, next character
    cpi rmp,0x0D ; Line change?
    brne LcdText2 ; No line change
    inc rLine ; Next Zeile
    mov rmp,rLine ; Select line
    rcall LcdLineSet ; Set output line
    rjmp LcdText1 ; Continue with characters
LcdText2: ; Write character
    rcall LcdD4Byte ; Character out

```

```

    rcall Wait40us ; Wait
    rjmp LCDText1 ; Go on
LCDTextRet:
    ret ; Ready
;
; Sets the output cursor to the line start
; of the selected line
; Line: rmp 0 to 3
LCDLineSet:
    cpi rmp,1 ; Line 2?
    brcs LCDLineSet1 ; no, to line 1
    breq LCDLineSet2 ; yes, to line 2
    cpi rmp,2 ; Line 3?
    breq LCDLineSet3 ; yes, to line 3
    rjmp LCDLineSet4 ; no, to line 4
LCDLineSet1:
    ldi rmp,0x80 ; Line 1
    rjmp LCDLineSetRmp
LCDLineSet2:
    ldi rmp,0xC0 ; Line 2
    rjmp LCDLineSetRmp
LCDLineSet3:
    ldi rmp,0x80+20 ; Line 3
    rjmp LCDLineSetRmp
LCDLineSet4:
    ldi rmp,0xC0+20 ; Line 4
    rjmp LCDLineSetRmp
LCDLineSetRmp:
    rcall LCDC4Byte ; Output to LCD control port
    rjmp Wait40us
;
; Data byte write in 4 bit mode
; Data byte in rmp
LCD4Byte:
    sbi pLCDCO,bLCDORS ; Set RS bit
    rjmp LCD4Byte ; Write byte in rmp
;
; Control write in 4 bit mode
; Data in rmp
LCDC4Byte:
    cbi pLCDCO,bLCDORS ; Clear RS bit
; Write byte in 4 bit mode
LCD4Byte:
    push rmp ; rmp to stack
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLCDI ; Read LCD data port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; OR upper and lower nibble
    out pLCDDO,rmp ; Write to data port LCD
    nop ; Wait for one clock cycle
    sbi pLCDCO,bLCDCOE ; Activate LCD-Enable
    nop ; Wait for one clock cycle
    cbi pLCDCO,bLCDCOE ; Clear LCD-Enable
    pop rmp ; rmp back from stack
    andi rmp,0x0F ; Clear upper nibble
    swap rmp ; Exchange upper/lower nibble
    or rmp,rmo ; OR upper and lower nibble
    out pLCDDO,rmp ; to LCD data
    nop ; Wait for one clock cycle
    sbi pLCDCO,bLCDCOE ; Activate LCD-Enable
    nop ; Wait for one clock cycle
    cbi pLCDCO,bLCDCOE ; Clear LCD-Enable
    ret ; Complete
;
; Control byte output in 8 bit mode
; Data byte in rmp
LCDC8Byte:
    cbi pLCDCO,bLCDORS ; Clear LCD-RS
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLCDI ; Read data bus port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; OR upper and lower nibble
    out pLCDDO,rmp ; Write to LCD data port
    nop ; Wait for one clock cycle
    sbi pLCDCO,bLCDCOE ; Activate LCD-Enable
    nop ; Wait for one clock cycle
    cbi pLCDCO,bLCDCOE ; Clear LCD-Enable

```

```

        ret ; Done
;
; ----- Wait routines -----
Wait50ms: ; Wait for 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-18+2)/4
; rcall: + 3
    push ZH ; + 2
    push ZL ; + 2
    ldi ZH,HIGH(n50ms) ; + 1
    ldi ZL,LOW(n50ms) ; + 1
    rjmp LcdWait ; + 2, total = 11
Wait5ms: ; Wait for 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-18+2)/4
    push ZH
    push ZL
    ldi ZH,HIGH(n5ms)
    ldi ZL,LOW(n5ms)
    rjmp LcdWait
Wait100us: ; Wait for 100us
.equ c100us = 100
.equ n100us = (c100us-18+2)/4
    push ZH
    push ZL
    ldi ZH,HIGH(n100us)
    ldi ZL,LOW(n100us)
    rjmp LcdWait
Wait40us: ; Wait for 40us
.equ c40us = 40
.equ n40us = (c40us-18+2)/4
    push ZH
    push ZL
    ldi ZH,HIGH(n40us)
    ldi ZL,LOW(n40us)
    rjmp LcdWait
; Wait routine Z delay loops
LcdWait: ; Wait loop, Clock cycles = 4*(n-1)+11 = 4*n + 7
    sbiw ZL,1 ; + 2
    brne LcdWait ; + 1 / 2
    pop ZL ; + 2
    pop ZH ; +2
    ret ; + 4, Total=4*n+18
;

```

This program utilizes one new instruction. SWAP register exchanges the lower and upper four bits (nibbles) of the register. This is much faster than shifting the register four times to the left or to the right.

Further new is RCALL distance. This jumps to a relative location given by the distance parameter and returns back to the origin if a ret instruction is executed because it has pushed the original program counter to the stack prior to jumping. The distance is calculated by the assembler, if a label to be called specifies the parameter. The distance is limited, in larger devices a wide call "CALL" label can be used instead. This requires two instruction words and longer execution times.

The instructions PUSH register and POP save a register on the stack or restore this from the stack and adjust the stack pointer accordingly. A POP always yields the last pushed byte.

Home	Top	LCD	ATtiny24	Hardware	Wait mode	Busy mode	Characters
----------------------	---------------------	---------------------	--------------------------	--------------------------	---------------------------	---------------------------	----------------------------

10.4.4 Simulating the wait routines

Here it makes some sense to test the wait routines with the simulator [avr_sim](#). For that we place calls to the wait routines behind the stack pointer init,

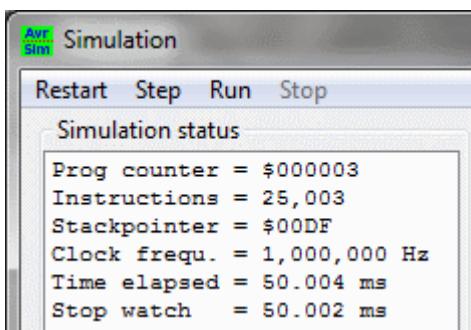
```
rcall Wait50ms
```

```

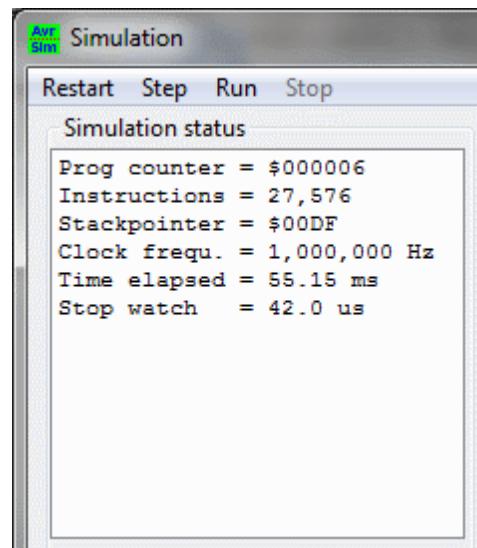
rcall Wait5ms
rcall Wait100us
rcall Wait40us
nop

```

place breakpoints to all of those lines and clear the stopwatch after every line executed.

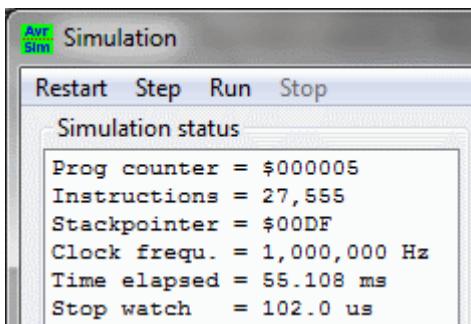


The routine is rather exact, only one clock cycle longer than calculated.



The same with the 5 ms loop.

And also for 100 μ s.



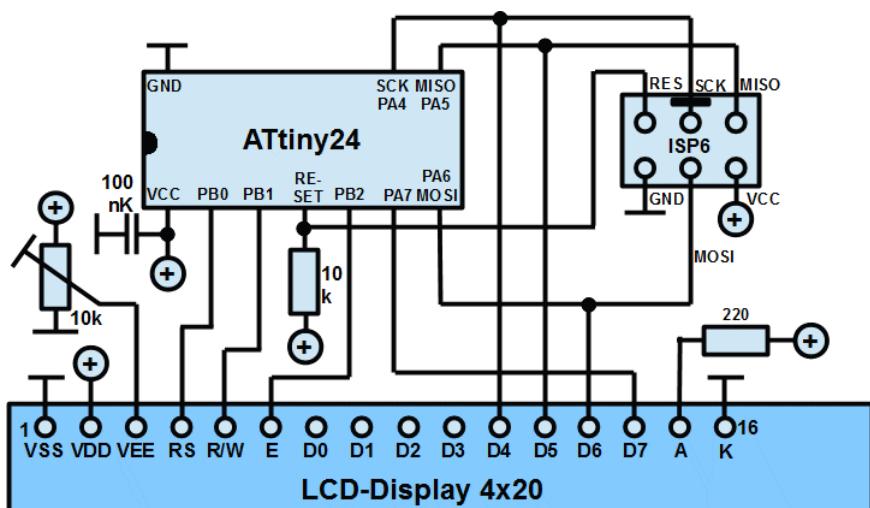
And, it is getting boring now, all the same for the 40 μ s loop. It seems that the formula has to be corrected for one cycle.

[Home](#) [Top](#) [LCD](#) [ATtiny24](#) [Hardware](#) [Wait mode](#) [Busy mode](#) [Characters](#)

10.5 Control of the LCD in busy mode

10.5.1 Reading the busy flag of the LCD

For reading the busy flag the LCD data bus has to be put to Write mode by setting the LCD-R/W input to one. Of course the ATtiny's data bus bits must change direction first. The LCD-R/W input requires an additional port pin of the ATtiny24, for which PB1 has to be connected.



10.5.2 Task

The task is the same as before (text output on the LCD), instead of wait cycles the busy flag of the LCD shall be utilized.

10.5.3 Program

This is the program, the [source code is here](#).

```
;  
; *****  
; * LCD display 4*20 on ATtiny24, 4 bit interface with busy *  
; * (C)2017 by http://www.avr-asm-tutorial.net  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Hardware -----  
;  
;  
;  
+ 5 V o--|VCC      GND|--o 0 V  
|  
LCD-RS o--|PB0      |--o  
|  
LCD-R/W o--|PB1      |--o  
|  
RESET o--|RES      |--o  
|  
LCD-E o--|PB2      |--o  
|  
LCD-D7 o--|PA7      PA4|--o LCD-D4  
|  
LCD-D6 o--|PA6      PA5|--o LCD-D5  
|  
|  
;  
;  
; ----- Ports, port pins -----  
; LCD control port  
.equ pLcdCO    = PORTB ; LCD control port output  
.equ pLcdCR    = DDRB ; LCD control port direction  
.equ bLcdCOE   = PORTB2 ; LCD enable pin output  
.equ bLcdCRE   = DDB2 ; LCD enable pin direction  
.equ bLcdCORS  = PORTB0 ; LCD RS pin output  
.equ bLcdCRRS  = DDB0 ; LCD RS pin direction  
.equ bLcdCORW  = PORTB1 ; LCD R/W pin output  
.equ bLcdCRRW  = DDB1 ; LCD R/W pin direction  
;  
; LCD data port  
.equ pLcdDO    = PORTA ; LCD data port output  
.equ pLcdDI    = PINA ; LCD data port input  
.equ pLcdDR    = DDRA ; LCD data port direction  
.equ mLcdDRW   = 0xF0 ; LCD data port mask write  
.equ mLcdDRR   = 0x00 ; LCD data port mask read  
;  
;  
; ----- Registers -----  
; free: R0 .. R15  
.def rmp = R16 ; Multi purpose register  
.def rmo = R17 ; Additional multi purpose register  
.def rLine = R18 ; Line counter LCD  
.def rLese = R19 ; Read result from LCD data port  
; free: R20 .. R29  
; used: R31:R30, ZH:ZL, for counting and as pointer  
;  
;  
; ----- Constants -----  
.equ Clock = 1000000 ; Default clock frequency  
;  
;  
; ---- Program start, Initiation -----  
.CSEG ; Code Segment  
.ORG 0 ; Start at 0  
    ; Init stack for subroutines  
    ldi rmp,LOW(RAMEND) ; Init stack  
    out SPL,rmp ; to stack pointer  
    ; Init ports and port pins  
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)  
    out pLcdCR,rmp ; to LCD control port direction  
    clr rmp ; Clear outputs  
    out pLcdCO,rmp ; to LCD control port  
    ldi rmp,mLcdDRW ; Data port output mask write  
    out pLcdDR,rmp ; to data port direction port
```

```

; Init LCD
rcall LcdInit
; Text output on LCD
ldi ZH,HIGH(2*Texttable)
ldi ZL,LOW(2*Texttable)
rcall LcdText ; Text from ZH:ZL in flash
; Sleep enable
ldi rmp,1<<SE
out MCUCR,rmp
Loop:
    sleep ; sleep
    rjmp Loop
;
; Text for LCD
Texttable:
.db "LCD display ATTiny24",0x0D,0xFF ; Line 1
.db "avr-asm-tutorial.net",0x0D,0xFF ; Line 2
.db " Testprogram 4 bit",0x0D,0xFF ; Line 3
.db "with busy flag read",0x00 ; Line 4
;
; ----- LCD control init -----
LcdInit:
    ; Wait 50 ms until LCD is available
    rcall Wait50ms ; Wait time 50 ms
    ; Set LCD to 8 bit mode
    ldi rmp,0x30 ; 8 bit mode
    rcall LcdC8Byte ; Write to LCD in 8 bit mode
    rcall Wait5ms ; Wait 5 ms
    ldi rmp,0x30 ; Second repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ldi rmp,0x30 ; Third repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ; Switch to 4 bit mode
    ldi rmp,0x20 ; To 4 bit mode
    rcall LcdC8Byte
    rcall Wait5ms
    ; Function set LCD
    ldi rmp,0x28 ; 4 bit mode, 4 lines, 5*7
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x0F ; Display on, blink
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x01 ; Clear Display
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x06 ; Auto-indent
    rjmp LcdC4Byte ; Byte to LCD control
;
; Output of text on the LCD
; Z points to text in flash
LcdText:
    clr rLine ; Line counter = 0
LcdText1:
    lpm rmp,Z+ ; read character from flash
    cpi rmp,0 ; End of text?
    breq LcdTextRet ; no
    cpi rmp,0xFF ; Dummy character?
    breq LcdText1 ; yes, continue
    cpi rmp,0x0D ; Line change?
    brne LcdText2 ; No line change
    inc rLine ; Next line
    mov rmp,rLine ; Select line
    rcall LcdLineSet ; Set line
    rjmp LcdText1 ; continue with characters
LcdText2: ; Write character
    rcall LcdD4Byte ; Write character to LCD
    rjmp LcdText1 ; Continue
LcdTextRet:
    ret ; Done
;
; Sets the output address to the line start of the
; selected line
; Line: rmp 0 to 3
LcdLineSet:
    cpi rmp,1 ; Line 2?
    brcs LcdLineSet1 ; no, to line 1
    breq LcdLineSet2 ; yes, to line 2

```

```

cpi rmp,2 ; Line 3?
breq LcdLineSet3 ; yes, to line 3
rjmp LcdLineSet4 ; no, to line 4
LcdLineSet1:
    ldi rmp,0x80 ; Line 1
    rjmp LcdLineSetRmp
LcdLineSet2:
    ldi rmp,0xC0 ; Line 2
    rjmp LcdLineSetRmp
LcdLineSet3:
    ldi rmp,0x80+20 ; Line 3
    rjmp LcdLineSetRmp
LcdLineSet4:
    ldi rmp,0xC0+20 ; Line 4
    rjmp LcdLineSetRmp
LcdLineSetRmp:
    rjmp LcdC4Byte ; Control byte to LCD
;
; Data output to LCD in 4 bit mode
;   Data in rmp
LcdD4Byte:
    rcall LcdBusy ; Wait until busy = low
    sbi pLcdCO,bLcdCORS ; Set LCD-RS
    rjmp Lcd4Byte ; Write byte to LCD
;
; Control output to LCD in 4 bit mode
;   Data in rmp
LcdC4Byte:
    rcall LcdBusy ; Wait bis busy = Null
    cbi pLcdCO,bLcdCORS ; Clear LCD-RS
; Output byte in 4 bit mode with busy
Lcd4Byte:
    push rmp ; Save rmp on stack
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLcdDI ; Read data bus port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; Combine lower and upper nibble
    out pLcdDO,rmp ; to data bus of the LCD
    nop ; Wait one cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
    nop ; Wait one cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    pop rmp ; Restore rmp from stack
    andi rmp,0x0F ; Clear upper nibble
    swap rmp ; Exchange nibbles
    or rmp,rmo ; Combine lower and upper nibble
    out pLcdDO,rmp ; Write to data bus LCD
    nop ; Wait one cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
    nop ; Wait one cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    ret ; Done
;
; Wait until busy clear
LcdBusy:
    push rmp ; Save rmp
    ldi rmp,mLcdDRR ; Read mask
    out pLcdDR,rmp ; to direction port
    cbi pLcdCO,bLcdCORS ; Clear LCD-RS
    sbi pLcdCO,bLcdCORW ; Set LCD-R/W
LcdBusy1:
    sbi pLcdCO,bLcdCOE ; Set LCD-Enable
    nop ; Wait one cycle
    in rLese,pLcdDI ; Read upper nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    andi rLese,0xF0 ; Clear lower nibble
    sbi pLcdCO,bLcdCOE ; Set LCD-Enable
    nop ; Wait one cycle
    in rmp,pLcdDI ; Read lower nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    andi rmp,0xF0 ; Clear lower nibble
    swap rmp ; Exchange upper/lower nibble
    or rLese,rmp ; Combine upper and lower nibble
    sbrc rLese,7 ; Skip if Busy=0
    rjmp LcdBusy1 ; Repeat until busy=0
    cbi pLcdCO,bLcdCORW ; Clear LCD-R/W
    ldi rmp,mLcdDRW ; Load write mask

```

```

        out pLcdDR,rmp ; to direction port
        pop rmp ; Restore rmp
        ret ; Return
;
; LCD control output in 8 bit mode
;   Data in rmp
LcdC8Byte:
        cbi pLcdCO,bLcdCORS ; Clear LCD-RS
        andi rmp,0xF0 ; Clear lower nibble
        in rmo,pLcdDI ; Read data port
        andi rmo,0x0F ; Clear upper nibble
        or rmp,rmo ; Combine lower and upper nibble
        out pLcdDO,rmp ; Write to LCD data port
        nop ; Wait one cycle
        sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
        nop ; Wait one cycle
        cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
        ret ; Done
;
; ----- Wait routines -----
Wait50ms: ; Wait routine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-18)/4
;    rcall: + 3
        push ZH ; + 2
        push ZL ; + 2
        ldi ZH,HIGH(n50ms) ; + 1
        ldi ZL,LOW(n50ms) ; + 1
        rjmp LcdWait ; + 2, total = 11
Wait5ms: ; Wait routine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-18)/4
        push ZH
        push ZL
        ldi ZH,HIGH(n5ms)
        ldi ZL,LOW(n5ms)
        rjmp LcdWait
; Wait routine Z loops
LcdWait: ; Wait loop, nClocks = 4*(n-1)+11 = 4*n + 7
        sbiw ZL,1 ; + 2
        brne LcdWait ; + 1 / 2
        pop ZL ; + 2
        pop ZH ; +2
        ret ; + 4, Total=4*n+18
;

```

No new instructions are used here.

[Home](#) [Top](#) [LCD](#) [ATtiny24](#) [Hardware](#) [Wait mode](#) [Busy mode](#) [Characters](#)

10.6 To design new characters

10.6.1 The character generator in LCDs

All LCDs offer the opportunity to define own characters, but only those between 0 and 7. Those are designed by default (remember: memories cannot be empty) but their content can be overwritten.

Overwriting works like this:

1. A byte of the type 0b01NNNZZZ is send to the control port of the LCD (with LCD-RS = 0). Therein NNN is the number of the character to be overwritten (0 bis 7). ZZZ is the line of the character, o is the most upper line, 7 the most lower line.
2. Following that a byte of the type 0b000BBBBB is send to the data port of the LCD, with LCD-RS = 1. The Bs stand for the pixels of that line, with the most significant B as the leftmost pixel.

3. For all eight lines of the characters one pair of address and data are to be written.

10.6.2 Software for character design

To ease the design of such characters one can use a spread sheet.

Comfortable character design goes like this: open either the [OpenOffice spreadsheet](#) or the [M\\$ Office Excel spreadsheet](#). To display a pixel in white, write a one to the cell, in background color write a zero there. The address and data in decimal format are generated from this on the right side. To better remind yourself what character what is fill in the description below those characters.

From that the spreadsheet generator constructs an assembler table. It starts with the address of the character, followed by a null byte (remember: every line must have an even number of bytes). Then the eight data lines of the character follow. If all designed characters are defined (there could well be less than eight), a zero byte follows to signal the end. The table can be selected in total, copied with Ctrl-C and inserted to the source code with Strg-V.

10.6.3 Task

Define the arrow characters listed in the above graphic and display those on line 4 of the LCD.

10.6.4 The program

10.6.4.1 The spreadsheet calculation

The spreadsheet is available [in OpenOffice format](#) and [in Excel-Format](#).

10.6.4.2 Changes to the previous program

The previous program can be used as a basis. The following changes are required:

- Between the LCD's init code and the text output to the LCD a new routine has to be added that writes the new characters to the LCD.
 - The text output routine for the characters has to be changed. To write the character null the previous end character that signalled the table end has to be changed. The character 0xFE is a good selection for this.

10.6.4.3 The code

Here is the program code, also available [as source code in asm format](#).

```

;
; **** LCD display 4*20 on ATtiny24/4 bit/busy/arrows ****
; * (C)2017 by http://www.avr-asm-tutorial.net      *
; ****
;

.NOLIST
.INCLUDE "tn24def.inc"
.LIST

;
; ----- Hardware -----
;

;
; + 5 V o--|VCC      GND|--o 0 V
; |          |
; LCD-RS o--|PB0      |--o
; |          |
; LCD-R/W o--|PB1      |--o
; |          |
; RESET o--|RES      |--o
; |          |
; LCD-E o--|PB2      |--o
; |          |
; LCD-D7 o--|PA7      PA4|--o LCD-D4
; |          |
; LCD-D6 o--|PA6      PA5|--o LCD-D5
; |          |
;

;
; ----- Ports, port pins -----
; LCD control port
.equ pLcdCO = PORTB ; LCD control port output
.equ pLcdCR = DDRB ; LCD control port direction
.equ bLcdCOE = PORTB2 ; LCD enable pin output
.equ bLcdCRE = DDB2 ; LCD enable pin direction
.equ bLcdCORS = PORTB0 ; LCD RS pin output
.equ bLcdCRRS = DDB0 ; LCD RS pin direction
.equ bLcdCORW = PORTB1 ; LCD R/W pin output
.equ bLcdCRRW = DDB1 ; LCD R/W pin direction
; LCD data port
.equ pLcdDO = PORTA ; LCD data port output
.equ pLcdDI = PINA ; LCD data port input
.equ pLcdDR = DDRA ; LCD data port direction
.equ mLcdDRW = 0xF0 ; LCD data port mask write
.equ mLcdDRR = 0x00 ; LCD data port mask read
;
; ----- Registers -----
; free: R0 .. R15
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Additional multi purpose register
.def rLine = R18 ; Line counter LCD
.def rLese = R19 ; Read result from LCD data port
.def rAddr = R20 ; Line address LCD character
; free: R21 .. R29
; used: R31:R30, ZH:ZL, for counting and as pointer
;
; ----- Constants -----
.equ Clock = 1000000 ; Default clock frequency
;
; ---- Program start, Initiation -----
.CSEG ; Code Segment
.ORG 0 ; Start at 0
    ; Init stack for subroutines
    ldi rmp,LOW(RAMEND) ; Init stack
    out SPL,rmp ; to stack pointer
    ; Init ports and port pins
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port direction
    clr rmp ; Clear outputs
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; Data port output mask write
    out pLcdDR,rmp ; to data port direction port
    ; Init LCD
    rcall LcdInit
    ; Define characters

```

```

rcall LcdChars ; New characters
; Text output on LCD
ldi ZH,HIGH(2*Texttable)
ldi ZL,LOW(2*Texttable)
rcall LcdText ; Text from ZH:ZL in flash
; Sleep enable
ldi rmp,1<<SE
out MCUCR,rmp
Loop:
    sleep ; sleep
    rjmp Loop
;
; Output text on LCD
Texttable:
.db "LCD display ATTiny24",0x0D,0xFF ; Line 1
.db "avr-asm-tutorial.net",0x0D,0xFF ; Line 2
.db "Own characters here:",0x0D,0xFF ; Line 3
.db "",0x00," ",0x01," ",0x02," ",0x03 ; Line 4 left
.db "",0x04," ",0x05," ",0x06," ",0x07 ; Line 4 right
.db 0xFE,0xFE ; End of text
;
; ----- LCD control init -----
LcdInit:
    ; Wait 50 ms until LCD is available
    rcall Wait50ms ; Wait time 50 ms
    ; Set LCD to 8 bit mode
    ldi rmp,0x30 ; 8 bit mode
    rcall LcdC8Byte ; Write to LCD in 8 bit mode
    rcall Wait5ms ; Wait 5 ms
    ldi rmp,0x30 ; Second repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ldi rmp,0x30 ; Third repeat
    rcall LcdC8Byte
    rcall Wait5ms
    ; Switch to 4 bit mode
    ldi rmp,0x20 ; To 4 bit mode
    rcall LcdC8Byte
    rcall Wait5ms
    ; Function set LCD
    ldi rmp,0x28 ; 4 bit mode, 4 lines, 5*7
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x0F ; Display on, blink
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x01 ; Clear Display
    rcall LcdC4Byte ; Byte to LCD control
    ldi rmp,0x06 ; Auto-indent
    rjmp LcdC4Byte ; Byte to LCD control
;
; Define own characters
LcdChars:
    ldi ZH,HIGH(2*Codechars) ; Z to character code table
    ldi ZL,LOW(2*Codechars)
LcdChars1:
    lpm rAddr,Z ; Read character address
    tst rAddr ; Check zero (end of table)
    breq LcdChars3 ; Done
    adiw ZL,2 ; to first data byte
    ldi rLine,8
LcdChars2:
    mov rmp,rAddr ; Write Address
    rcall LcdC4Byte ; to LCD
    lpm rmp,Z+ ; Read line pixels
    rcall LcdD4Byte ; Write to LCD
    inc rAddr ; Increase address
    dec rLine ; Count downwards
    brne LcdChars2 ; Further pixel bytes
    rjmp LcdChars1 ; Next character
LcdChars3:
    ret ; Done

; Table of code characters
Codechars:
.db 64,0,0,12,6,31,6,12,0,0 ; Z = 0, Arrow right
.db 72,0,0,6,12,31,12,6,0,0 ; Z = 1, Arrow left
.db 80,0,4,14,31,21,4,4,4,0 ; Z = 2, Arrow up
.db 88,0,4,4,4,21,31,14,4,0 ; Z = 3, Arrow down

```

```

.db 96,0,0,15,3,5,9,16,0,0 ; Z = 4, Arrow right up
.db 104,0,0,16,9,5,3,15,0,0 ; Z = 5, Arrow right down
.db 112,0,0,1,18,20,24,30,0,0 ; Z = 6, Arrow left down
.db 120,0,0,30,24,20,18,1,0,0 ; Z = 7, Arrow left up
.db 0,0 ; End of table
;
; Output of text on the LCD
; Z points to text in flash
LcdText:
    clr rLine ; Line counter = 0
LcdText1:
    lpm rmp,Z+ ; read character from flash
    cpi rmp,0xFE ; End of text?
    breq LcdTextRet ; no
    cpi rmp,0xFF ; Dummy character?
    breq LcdText1 ; yes, continue
    cpi rmp,0x0D ; Line change?
    brne LcdText2 ; No line change
    inc rLine ; Next line
    mov rmp,rLine ; Select line
    rcall LcdLineSet ; Set line
    rjmp LcdText1 ; continue with characters
LcdText2: ; Write character
    rcall LcdD4Byte ; Write character to LCD
    rjmp LcdText1 ; Continue
LcdTextRet:
    ret ; Done
;
; Sets the output address to the line start of the
; selected line
; Line: rmp 0 to 3
LcdLineSet:
    cpi rmp,1 ; Line 2?
    brcc LcdLineSet1 ; no, to line 1
    breq LcdLineSet2 ; yes, to line 2
    cpi rmp,2 ; Line 3?
    brcc LcdLineSet3 ; yes, to line 3
    rjmp LcdLineSet4 ; no, to line 4
LcdLineSet1:
    ldi rmp,0x80 ; Line 1
    rjmp LcdLineSetRmp
LcdLineSet2:
    ldi rmp,0xC0 ; Line 2
    rjmp LcdLineSetRmp
LcdLineSet3:
    ldi rmp,0x80+20 ; Line 3
    rjmp LcdLineSetRmp
LcdLineSet4:
    ldi rmp,0xC0+20 ; Line 4
    rjmp LcdLineSetRmp
LcdLineSetRmp:
    rjmp LcdC4Byte ; Control byte to LCD
;
; Data output to LCD in 4 bit mode
; Data in rmp
LcdD4Byte:
    rcall LcdBusy ; Wait until busy = low
    sbi pLcdCO,bLcdCORS ; Set LCD-RS
    rjmp Lcd4Byte ; Write byte to LCD
;
; Control output to LCD in 4 bit mode
; Data in rmp
LcdC4Byte:
    rcall LcdBusy ; Wait bis busy = Null
    cbi pLcdCO,bLcdCORS ; Clear LCD-RS
; Output byte in 4 bit mode with busy
Lcd4Byte:
    push rmp ; Save rmp on stack
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLcdDI ; Read data bus port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmr ; Combine lower and upper nibble
    out pLcdDO,rmp ; to data bus of the LCD
    nop ; Wait one cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
    nop ; Wait one cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable

```

```

pop rmp ; Restore rmp from stack
andi rmp,0x0F ; Clear upper nibble
swap rmp ; Exchange nibbles
or rmp,rmo ; Combine lower and upper nibble
out pLcdDO,rmp ; Write to data bus LCD
nop ; Wait one cycle
sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
nop ; Wait one cycle
cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
ret ; Done
;
; Wait until busy clear
LcdBusy:
    push rmp ; Save rmp
    ldi rmp,mLcdDRR ; Read mask
    out pLcdDR,rmp ; to direction port
    cbi pLcdCO,bLcdCORS ; Clear LCD-RS
    sbi pLcdCO,bLcdCORW ; Set LCD-R/W
LcdBusy1:
    sbi pLcdCO,bLcdCOE ; Set LCD-Enable
    nop ; Wait one cycle
    in rLese,pLcdDI ; Read upper nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    andi rLese,0xF0 ; Clear lower nibble
    sbi pLcdCO,bLcdCOE ; Set LCD-Enable
    nop ; Wait one cycle
    in rmp,pLcdDI ; Read lower nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    andi rmp,0xF0 ; Clear lower nibble
    swap rmp ; Exchange upper/lower nibble
    or rLese,rmp ; Combine upper and lower nibble
    sbrc rLese,7 ; Skip if Busy=0
    rjmp LcdBusy1 ; Repeat until busy=0
    cbi pLcdCO,bLcdCORW ; Clear LCD-R/W
    ldi rmp,mLcdDRW ; Load write mask
    out pLcdDR,rmp ; to direction port
    pop rmp ; Restore rmp
    ret ; Return
;
; LCD control output in 8 bit mode
; Data in rmp
LcdC8Byte:
    cbi pLcdCO,bLcdCORS ; Clear LCD-RS
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLcdDI ; Read data port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; Combine lower and upper nibble
    out pLcdDO,rmp ; Write to LCD data port
    nop ; Wait one cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD-Enable
    nop ; Wait one cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD-Enable
    ret ; Done
;
; ----- Wait routines -----
Wait50ms: ; Wait routine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-18)/4
; rcall: + 3
    push ZH ; + 2
    push ZL ; + 2
    ldi ZH,HIGH(n50ms) ; + 1
    ldi ZL,LOW(n50ms) ; + 1
    rjmp LcdWait ; + 2, total = 11
Wait5ms: ; Wait routine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-18)/4
    push ZH
    push ZL
    ldi ZH,HIGH(n5ms)
    ldi ZL,LOW(n5ms)
    rjmp LcdWait
; Wait routine Z loops
LcdWait: ; Wait loop, nClocks = 4*(n-1)+11 = 4*n + 7
    sbiw ZL,1 ; + 2
    brne LcdWait ; + 1 / 2
    pop ZL ; + 2

```

```
pop ZH ; +2  
ret ; + 4, Total=4*n+18  
;
```

This is the result of that all (the German version): all nice arrows in line 4.

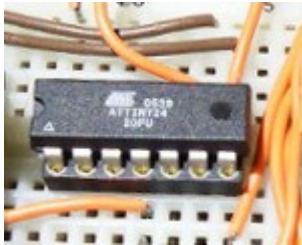
Here instead of CPI register,0 the instruction TST register was used, if the register is zero the Z flag is set, otherwise cleared.

New is ADIW register,N. This adds N to the register pair word-wise, and is only available for the LSB of the double registers (R24, R26, R28 and R30).

An additional hint that might be useful: There is a useful versatile LCD include file available that masters all variations (4-/8-bit interface, wait/busy mode, special characters) in one include file. That can be configured from within the main source code by setting constants. The routine is described [here](#), the include file can be downloaded [here](#) as assembler include file.



[Home](#) [Top](#) [LCD](#) [ATtiny24](#) [Hardware](#) [Wait mode](#) [Busy mode](#) [Characters](#)



Lecture 11: EEPROM with LCD on an ATtiny24

New is in this lecture to read from and to write to the internal EEPROM and to convert binary numbers to decimal and to display those.

11.0 Overview

1. [Introduction to EEPROM](#)
2. [Introduction to decimal conversion](#)
3. [Hardware, components, Mounting](#)
4. [An 8 bit counter counts power-ups](#)
5. [A 16 bit counter](#)

11.1 Introduction to EEPROM

11.1.1 An EEPROM as a permanent memory

The internal SRAM is rather short-lived, its content is deleted whenever the operating voltage gets lost. When the controller starts new, he has to initiate the SRAM's content to their default values. If you have had a lengthy procedure to set the SRAM's content to your very special setting, e.g. ten favoured wake-up times in a digital clock, you start all over again whenever the controller restarts. Not very convenient ...

The built-in EEPROM is not that short-lived. Its content remains unchanged even if the operating voltage is missing for longer times. Typical content where EEPROM to use for are such wake-up times, preselected time-outs of a stop watch, a preselected and adjustable temperature for your refrigerator or - like in our case - hidden power-up counters for printers or for other equipment.

EEPROMs are not designed to withstand millions of read and write cycles. ATMEL guarantees 100,000 erase and write cycles. EEPROM therefore should not be used like SRAM, with several write operations per second. With 3 ms duration per write cycle this is nevertheless the most ineffective way to store something temporarily.

The ATtiny24 has 128 byte EEPROM, which is sufficient for far more than ten favored wake-up times. It has its own address space, starts at 0x0000 and ends at 0x007F.

11.1.2 To write to the EEPROM

Three different write procedures are possible with the EEPROM:

1. to clear its content in total. This requires a programming tool like the built-in tools in the Studio. No, the content after clearing is not 0x00, it is 0xFF. A post-write of a binary one to a location that is already zero is not possible, unless you write the whole byte by controller action or you clear the whole EEPROM to 0xFF first.
2. to write its content. This also requires a programming tool, and a hex file that holds addresses and content in a special encoding format. The write cycle has to be preceded by an erase.
3. to erase a byte (to 0xFF) and to program its content in two subsequent steps by controller action.

The first write process (clearing) is automatically performed if you program the flash memory. So whenever you transfer a new version of your software, the EEPROM content gets lost and has to be programmed new (to program the EEPROM hex file). Most of the more recent AVR's allow to set a fuse named "EESAVE", which prevents clearing the EEPROM content during flash memory write procedures. If this fuse is set, the EEPROM content has to be separately cleared and rewritten. Of course, the controller can write content at any time even with this fuse set.

To create and later write an EEPROM hex file, insert the following to your assembler source code (at whatever place in the code):

```
.ESEG ; Switch to EEPROM segment and EEPROM address counter
.ORG 0 ; Start at address 0 (or wherever you want)
.db 0,1,2,3,4 ; Write bytes
.db "01234" ; Write text in ASCII codes
.CSEG ; Switches back to the code segment
```

The directive ".ESEG" advises the assembler to assemble the following content to the EEPROM hex file. This file is named "SourceCodeName.eep", where SourceCodeName is the name of your assembler source code file in .asm format. The generated .eep file is in Intel hex format and can be read with any simple editor.

As the EEPROM is organized byte-wise, any number of bytes can be written to the ESEG until it is full (if you specified your device type in the .include directive). With the .ORG directive it is possible to write the specified content to any location of the EEPROM, even if other EEPROM space already holds other content (remember: it is not possible to overwrite zeros!).

Within the ESEG only ".DB"- and ".DW"-directives are accepted by the assembler.

In order to write the content of such an .eep file one uses its favoured programming tool (e.g. within the Studio), selects the EEPROM write section and selects the generated .eep file (in the Studio by clicking on the small square behind the file name with the "..." on it). By clicking on "WRITE" the content is written to the EEPROM.

The same section of the programming tools can be used to read the content of the EEPROM to a file. Which then is also in Intel hex format, so better have a look at those kind of files in case you ever need that information or if you want to verify the EEPROM's content.

11.1.3 To read EEPROM content in assembler

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	-	-	EEPM1	EEPMD	EERIE	EEMPE	EEPE	EERE	EECR
ReadWrite	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	X	X	0	0	X	0	

Of course reading EEPROM content in assembler is done via ports. The following steps have to be absolved:

1. Make sure that the EEPROM is not occupied by any write operation. This can be done by reading the EEPE bit in the EEPROM control port EECR. If that bit is zero the EEPROM is ready to be read.
2. The address, from which content shall be read is to be written to the ports EEARH (MSB) and EEARL (LSB). Even if the device has less or equal 256 EEPROM bytes, the MSB shall be cleared.
3. The Read Enable bit EERE in the control port is set. This blocks the controller for four clock cycles and transfers the data from the pre-selected address in the EEPROM to the data port "EEDR", from which the content can be read.

Subsequent read accesses need only to adjust the LSB of the read address (as long as the MSB is not different), there is no auto-increment implemented.

12.1.4 To write data to the EEPROM in assembler

Writing EEPROM content goes as follows:

1. First it has to be checked that previous write cycles are finished (EEPE in port EECR is zero).
2. By writing the EECR to all zero it has to be assured that EERE as well as EEPM1 and EEPM0 are clear. EEPM1 and EEPM0 specify the write mode, 00 is the sequence "Erase" and subsequent "Write".
3. MSB and LSB of the write address are written to the address ports EEARH and EEARL.
4. The byte that is to be written to the EEPROM is written to the data port EEDR.
5. The EEMPE bit in the EEPROM control port EECR is set. This bit deletes itself after four clock cycles (so nothing else shall be executed in between this and the next instruction, no interrupt shall occur and be handled).
6. Within these four clock cycles set the EEPE bit in EECR. The Erase&Write cycle now lasts for 3.2 ms, at the end the EEPE bit clears itself.

After starting the programming sequence (not before!) setting the EERIE bit in EECR enables to trigger the EE_RDY interrupt. Following the last write operation the EERIE bit has to be cleared, otherwise the ready-to-program interrupt would cause a permanent interrupt and block the controller from doing something else.

[Home](#) [Top](#) [EEPROM](#) [Decimal conversion](#) [Hardware](#) [Byte counter](#) [Word counter](#)

11.2 Introduction to decimal conversion of binaries

Because we have a LCD now, we would like to display human-readable numbers, which are of course decimal. So we have to learn how to convert binaries.

11.2.1 The most primitive (and most lengthy) version

8 bit numbers in binary are between 000 and 255 in decimal. So we can start with "CPI rNmbr,200" if the first digit is a 2 (if carry is not set after CPI). If yes we place a "2" to the first decimal digit. If not, we compare with decimal 100 to find out, if it is a "1". Otherwise it is a "0", or with suppressing leading zeros a blank character. We can send the 2, 1 or the blank in ASCII to the LCD.

After we subtracted 200 or 100 from the binary number (if those were the case) we are between 0 and 99. Now we can in the same manner look downwards if the number is larger than 90, 80, 70, etc. to identify the next digit. A very lengthy process, but controllers are dumb and will execute whatever is told, no matter how intelligent the source code is.

Beware to write a blank to the LCD if this second digit is zero: it would be confusing to see for example "2 8".

If we have identified the second digit now and reduced the number down to the last digit 0 to 9, we can just add an ASCII-0 (decimal 48) and we are complete. But do not try "ADDI rNmbr,48", it won't assemble. Only SUBI is correct, so we can use an old trick by formulating "SUBI rNmbr,-48". As we learned in school "minus minus is plus". So who cares if the AVRs do not know an ADDI instruction?

11.2.2 The improved version

Instead of comparing we can subtract 100 (or 10 for the second digit) on and on until a carry occurs and to count the number of times this did not lead to a carry bit set. That goes like this:

; The binary number is in R0

```

ldi R16,100 ; R16 is the decimal digit
clr R1 ; R1 is counter
Count1:
    cp R0,R16 ; Compare with 100
    brcs Digit1 ; Overflow, digit complete
    sub R0,R16 ; Subtract 100
    inc R1 ; Increase counter
    rjmp Count1 ; Go on comparing/subtracting
Digit1:
    ldi R16,'0' ; ASCII-Zero
    add R16,R1 ; Add to counter
    ; Here: output digit 1
    ldi R16,10 ; Now the 10s
    clr R1 ; Again start at zero
Count2:
    cp R0,R16 ; Compare with 10
    brcs Digit2 ; Overflow, digit complete
    sub R0,R16 ; Subtract 10
    inc R1 ; Increase counter
    rjmp Count2 ; Go on comparing/subtracting
Digit2:
    ldi R16,'0' ; ASCII-Zero
    add R16,R1 ; Add counter
    ; Here: output digit 2
    ldi R16,'0' ; ASCII-Zero
    add R16,R0 ; Add remaining rest
    ; Here: output digit 3

```

New is the instruction CP register, register. This subtracts the second register from the first one temporarily, sets the flags in SREG and leaves the first register as it was.

The two parts marked red are identical. If you are short in flash memory you can formulate it as a subroutine and call it with RCALL.

11.2.3 The 16 bit version

If we have a binary with 16 bits length, the following changes:

- Comparers and subtractors are now two bytes long.
- The digits to be compared with now are 10,000, 1,000, 100 and 10 (between 0 and 65535, five decimal digits).
- Comparing and subtracting now have to be in 16 bit mode.

This goes like this (now comparison, subtracting and conversion to ASCII in a subroutine):

```

; The binary to be converted is in R1:R0
    ldi ZH,HIGH(10000) ; Ten thousands
    ldi ZL,LOW(10000)
    rcall Count
    ; Digit 1 output
    ldi ZH,HIGH(1000) ; Thousands
    ldi ZL,LOW(1000)
    rcall Count
    ; Digit 2 output
    ldi ZH,HIGH(100) ; Hundreds
    ldi ZL,LOW(100)
    rcall Count
    ; Digit 3 output
    ldi ZL,LOW(10) ; Tens
    rcall Count
    ; Digit 4 output
    ldi R16,'0'
    add R16,R0
    ; Digit 5 output
    ; done
; Subroutine
Count:
    clr R16 ; R16 is counter
Count1:
    sub R0,ZL ; Subtract LSB
    sbc R1,ZH ; Subtract MSB with carry
    brcs Count2 ; Overflow during subtract
    inc R16 ; no overflow

```

```

rjmp Count1 ; continue subtracting
Count2:
    add R0,ZL ; Add LSB to revert last subtract
    adc R1,ZH ; Add MSB and carry
    subi R16,-'0' ; Add ASCII-Zero
    ret ; Return with result in R16

```

New instructions here are ADC register,register and SBC register,register. This adds resp. subtracts the second register from the first and adds/subtracts one if the carry flag is set. Overflow from the previous LSB operation to the MSB of the 16 bit value can be handled with that.

One further branching instruction used here for the first time is BRCS distance. It branches in case that the carry flag was set by preceding instructions and adds the given distance to the program counter. Distances can be forward (distance positive) or backwards (distance negative). The assembler calculates those distances from the given label that is used to mark the branching target, but can also be specified manually with a constant value (e.g. brcs +3 or brcs -15). All SREG bits can be used to branch if set (BRxS) or cleared (BRxC). Note that the Z bit in SREG equals E in the branch instruction, using its original name Z for that leads to an error message of the assembler.

The problem is solved, but leading zeros occur as 0 and not as blank. This problem is resolved in the two software examples of this lecture.

Now it is easy to even convert 24 or 32 bit numbers (or even 40 bits, like in one of the upcoming lectures) to decimal.

[Home](#) [Top](#) [EEPROM](#) [Decimal conversion](#) [Hardware](#) [Byte counter](#) [Word counter](#)

11.3 Task, hardware, components and mounting

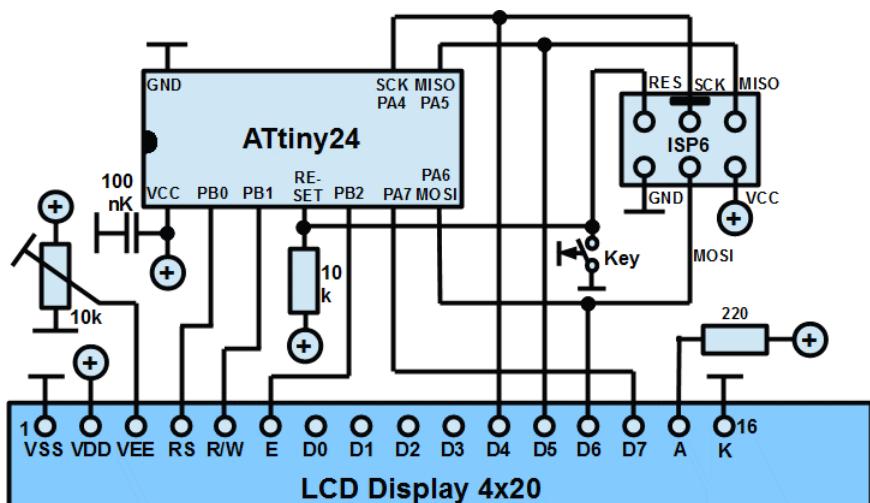
11.3.1 Task

The following task has to be solved: on every reset and if the operating voltage is applied a counter has to be increased and the previous and current state has to written to the LCD.

11.3.2 Scheme

To not having to switch off the operating voltage during the debugging phase we install a reset key.

All components are well known, mounting is also simple. The experimental board described [here](#) can also be used instead.



11.4 Byte counter

11.4.1 The LCD routines as include file

The routines to init and to write to the LCD can be used in general whenever a LCD is part of the program. In that case we can formulate those routines in a way that they can be used in different applications. We can write those routines to a different file, add a header that documents the properties and the routines provided and save this with the extension *.inc in the

same folder like where the .asm is located. In that case we can add a directive in the source code that includes these source code routines, such as .include "filename.inc". If the file to be included is stored in a different, the filename must include the path information. Assembling then is continued in this include file and finally returns back to the main code source. Note that the included routines appear at the place where the .include directive has been called.

This is the perfect 4 bit busy LCD include file, [here in source code format](#). Please note that files with the extension .inc in some cases are blocked by anti-virus software from downloading. In our case the .inc file is not dangerous.

```

;
; ****
; * Include Code for a 4 bit busy access to a LCD *
; * with base routines
; * (C)2017 by www.avr-asm-tutorial.net
; ****
;

; ----- Used registers -----
; rmp, rmo, rLine, rRead, R0
;

; ----- Routinen -----
; Routine Provides Call parameters Registers
; ----- -----
; LcdInit Init LCD Non rmp,rmo
; ; rRead
; LcdChars Generate own ZH:ZL Character rmp,rmo
; ; characters table rRead,R0
; LcdText Output the ZH:ZL Text table rmp,rmo
; ; text in the 0D=Next line rLine
; ; flash table FF=Ignore (Dummy) rRead
; ; in line 1 FE=End of table
; ; aus
; LcdTextC Output the (see LcdText) (see LcdText)
; ; text in the
; ; flash table
; ; on the cur-
; ; rent position
; LcdSRam Output the XH:XL: points to rmp,rmo
; ; text in SRAM position in SRAM rRead
; ; ZH:ZL: Lcd position
; ; rmp: Number of
; ; characters
; LcdPos Set output ZH: Line 0..3 rmp,rmo
; ; position ZL: Column 0..19 rRead
; LcdLine Set line, rmp: Line 0..3 rmp,rmo
; ; Column = 0 rRead
; LcdLineN Set line N: Line 1..4 rmp,rmo
; ; Column = 0 rRead
;

; -----Ports and portpins of the LCD -----
; LCD control port
.equ pLcdCO = PORTB ; LCD control port output
.equ pLcdCR = DDRB ; LCD control port direction
.equ bLcdCOE = PORTB2 ; LCD enable pin output
.equ bLcdCRE = DDB2 ; LCD enable pin direction
.equ bLcdCORS = PORTB0 ; LCD RS pin output
.equ bLcdCRRS = DDB0 ; LCD RS pin direction
.equ bLcdCORW = PORTB1 ; LCD R/W pin output
.equ bLcdCRRW = DDB1 ; LCD R/W pin direction
; LCD data port
.equ pLcdDO = PORTA ; LCD data port output
.equ pLcdDI = PINA ; LCD data port input
.equ pLcdDR = DDRA ; LCD data port direction
.equ mLcdDRW = 0xFO ; LCD data port write mask
.equ mLcdDRR = 0x0F ; LCD data port read mask
;

; ----- LCD access init -----
LcdInit:
    ; Wait for 50 ms until LCD is ready
    rcall Wait50ms ; Inactive delay 50 ms
    ; Switch to 8 bit mode (three times)
    ldi rmp,0x30 ; 8 bit mode
    rcall Lcd8Byte ; Write byte in 8 bit mode
    rcall Wait5ms ; Warte 5 ms

```

```

ldi rmp,0x30
rcall LcdC8Byte
rcall Wait5ms
ldi rmp,0x30
rcall LcdC8Byte
rcall Wait5ms
; Switch to 4 bit mode
ldi rmp,0x20 ; Switch to 4 bit mode
rcall LcdC8Byte ; Write byte in 8 bit mode
rcall Wait5ms
; Function set LCD
ldi rmp,0x28 ; 4 bit mode, 4 lines, 5*7
rcall LcdC4Byte ; Byte in 4 bit mode to LCD
ldi rmp,0x0F ; Display on, blink
rcall LcdC4Byte ; Byte in 4 bit mode to LCD
ldi rmp,0x01 ; Clear display
rcall LcdC4Byte ; Byte in 4 bit mode to LCD
ldi rmp,0x06 ; Auto-indent
rjmp LcdC4Byte ; Byte in 4 bit mode to LCD
;
; Output text in flash to LCD
; Z points to text in flash
LcdText:
    rcall LcdLine1 ; Set LCD to line 1
LcdTextC: ; Continue text output on current position
    clr rLine ; Clear line counter
LcdText1:
    lpm rmp,Z+ ; Read character from flash
    cpi rmp,0xFE ; End of text?
    breq LcdTextRet ; yes
    cpi rmp,0xFF ; Dummy character?
    breq LcdText1 ; yes, next character
    cpi rmp,0x0D ; Line change?
    brne LcdText2 ; No line change
    inc rLine ; Next line
    mov rmp,rLine ; Set line
    rcall LcdLine ; Change line
    rjmp LcdText1 ; Continue with characters
LcdText2: ; Output character
    rcall LcdD4Byte ; Output character
    rjmp LcdText1 ; and continue
LcdTextRet:
    ret ; Ready
;
; Output of text in SRAM to the LCD
; ZH = Line 0..3; ZL = Column 0..19,
; XH:XL = Address in SRAM, rmp: Number of chars
LcdSram:
    mov R0,rmp ; Copy number of characters
    rcall LcdPos ; Set LCD position to ZH:ZL
LcdSram1:
    ld rmp,X+ ; Read byte from SRAM
    rcall LcdD4Byte ; Output on LCD
    dec R0 ; Counter downwards
    brne LcdSram1 ; Continue with characters
    ret ; Done
;
; Set LCD output cursor to the line start
; of the selected line
; Line: rmp 0 .. 3
LcdLine:
    cpi rmp,1 ; Line 2?
    brcc LcdLine1 ; to line 1
    breq LcdLine2 ; to line 2
    cpi rmp,2 ; Line 3?
    breq LcdLine3 ; to line 3
    rjmp LcdLine4 ; to line 4
LcdLine1:
    ldi rmp,0x80 ; Line 1
    rjmp LcdC4Byte ; Output control byte
LcdLine2:
    ldi rmp,0xC0 ; Line 2
    rjmp LcdC4Byte ; Output control byte
LcdLine3:
    ldi rmp,0x80+20 ; Line 3
    rjmp LcdC4Byte ; Output control byte
LcdLine4:

```

```

ldi rmp,0xC0+20 ; Line 4
rjmp LcdC4Byte ; Output control byte
;
; Set the LCD output cursor to the position
; in ZH:ZL, ZH is line (0..3), ZL is column
LcdPos:
    ldi rmp,0x80 ; Set line 1
    cpi ZH,1 ; Line 2?
    brcc LcdPos1 ; Line = 1
    ldi rmp,0xC0 ; Set line 2
    breq LcdPos1 ; Line = 2
    ldi rmp,0x80+20 ; Set line 3
    cpi ZH,2 ; Line 3?
    breq LcdPos1 ; Line = 3
    ldi rmp,0xC0+20 ; Line = 4
LcdPos1:
    add rmp,ZL ; Add column
    rjmp LcdC4Byte ; Output control byte
;
; Define own character
LcdChars:
    lpm ; Read character address
    tst R0 ; Zero?
    breq LcdChars2 ; Done
    adiw ZL,2 ; Overread dummy byte
    ldi rLine,8
LcdChars1:
    mov rmp,R0 ; Address set
    rcall LcdC4Byte ; to LCD
    lpm rmp,Z+ ; Read data
    rcall LcdD4Byte ; Output to LCD
    inc R0 ; Increase address
    dec rLine ; Count down
    brne LcdChars1 ; Further bytes
    rjmp LcdChars ; Next character
LcdChars2:
    ret ; Done
;
; Data byte output in 4 bit mode
;   Data in rmp
LcdD4Byte:
    rcall LcdBusy ; Wait until busy = 0
    sbi pLcdCO,bLcdCORS ; Set RS bit
    rjmp Lcd4Byte ; Output byte in rmp
;
; Control byte output in 4 bit mode
;   Data in rmp
LcdC4Byte:
    rcall LcdBusy ; Wait until busy = 0
    cbi pLcdCO,bLcdCORS ; Clear RS bit
; Output byte in 4 bit mode with busy
Lcd4Byte:
    push rmp ; Save rmp on stack
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLcdDO ; Read data input port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; Combine lower and upper nibble
    out pLcdDO,rmp ; to data port LCD
    nop ; Wait one clock cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD enable
    nop ; Wait one clock cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD enable
    pop rmp ; Restore rmp
    andi rmp,0x0F ; Clear upper nibble
    swap rmp ; Exchange nibble
    or rmp,rmo ; Combine upper and lower nibble
    out pLcdDO,rmp ; to data port LCD
    nop ; Wait for one clock cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD enable
    nop ; Wait for one clock cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD enable
    ret ; Done
;
; Wait until busy zero
LcdBusy:
    push rmp ; Save rmp
    in rmp,pLcdDR ; Read LCD data bus

```

```

andi rmp,mLcdDRR ; Apply read mask
out pLcdDR,rmp ; To direction port
cbi pLcdCO,bLcdCORS ; Clear RS
sbi pLcdCO,bLcdCORW ; Set R/W
LcdBusyl:
    sbi pLcdCO,bLcdCOE ; Set LCD enable
    nop ; Wait for one clock cycle
    in rRead,pLcdDI ; Read upper nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD enable
    andi rRead,0xF0 ; Clear lower nibble
    sbi pLcdCO,bLcdCOE ; Set LCD enable
    nop ; Wait for one clock cycle
    in rmp,pLcdDI ; Read lower nibble
    cbi pLcdCO,bLcdCOE ; Clear LCD enable
    andi rmp,0xF0 ; Clear lower nibble
    swap rmp ; Exchange nibbles
    or rRead,rmp ; Combine upper and lower nibble
    sbrc rRead,7 ; Skip if busy=0
    rjmp LcdBusyl ; Repeat until busy=0
    cbi pLcdCO,bLcdCORW ; Clear R/W
    in rmp,pLcdDR ; Read direction
    ori rmp,mLcdDRW ; Set write bit mask
    out pLcdDR,rmp ; to direction port
    pop rmp ; Restore rmp
    ret ; Done
;
; Control byte output in 8 bit mode
; Data in rmp
LcdC8Byte:
    cbi pLcdCO,bLcdCORS ; Clear RS bit
    andi rmp,0xF0 ; Clear lower nibble
    in rmo,pLcdDO ; Read data input port
    andi rmo,0x0F ; Clear upper nibble
    or rmp,rmo ; Combine lower and upper nibble
    out pLcdDO,rmp ; to data port LCD
    nop ; Wait for one clock cycle
    sbi pLcdCO,bLcdCOE ; Activate LCD enable
    nop ; Wait for one clock cycle
    cbi pLcdCO,bLcdCOE ; Clear LCD enable
    ret ; Done
;
; ----- Wait routines -----
Wait50ms: ; Wait routine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-18)/4
; rcall: + 3
    push ZH ; + 2
    push ZL ; + 2
    ldi ZH,HIGH(n50ms) ; + 1
    ldi ZL,LOW(n50ms) ; + 1
    rjmp LcdWait ; + 2, total = 11
Wait5ms: ; Wait routine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-18)/4
    push ZH
    push ZL
    ldi ZH,HIGH(n5ms)
    ldi ZL,LOW(n5ms)
    rjmp LcdWait
; Wait routine Z cycles
LcdWait: ; Wait loop, Clock = 4*(n-1)+11 = 4*n + 7
    sbiw ZL,1 ; + 2
    brne LcdWait ; + 1 / 2
    pop ZL ; + 2
    pop ZH ; +2
    ret ; + 4, Total=4*n+18
;
; End of include
;

```

The include source uses the instruction OR register,register to combine the current lower nibble that was read, and masked with the ANDI instruction, with the desired upper nibble. The OR sets all bits that are either set in the first or in the second register and stores the result in the first register.

With this include file the following program will be much shorter.

11.4.2 The program

The program for solving the task is as follows, [the source code is here](#). Remember that the [include routine is required](#).

```
;  
; *****  
; * Read and write EEPROM of an ATtiny24 and a 4 line LCD *  
; * (C)2017 by http://www.avr-asm-tutorial.net  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Ports, port bits -----  
; (All LCD-Ports are defined in the include routine)  
;  
; ----- Registers -----  
; Use: R0 locally by the character definition routine  
; free: R1 .. R15  
.def rmp = R16 ; Multi purpose register  
.def rmo = R17 ; Additional multi purpose register  
.def rLine = R18 ; Line counter LCD  
.def rRead = R19 ; Read result from LCD data port  
.def rEep = R20 ; Data byte for EEPROM counter  
; free R21 .. R25  
; Used: XH:XL R27:R26 pointer  
; free: YH:YL R29:R28  
; Used: R31:R30, ZH:ZL, for counting and LCD  
;  
; ----- Constants -----  
.equ Clock = 1000000 ; Default clock frequency  
.equ EepAddress = 0 ; Read and write address EEPROM  
;  
; ----- SRAM -----  
.DSEG ; Assemble to data segment  
.ORG 0x0060 ; To SRAM start address  
Number: ; Convert byte to ASCII text  
.BYTE 3 ; Requires three characters  
;  
; ----- EEPROM segment -----  
.ESEG ; Assemble to EEPROM segment  
.ORG EepAddress  
.db 0 ; Init the EEPROM counter  
;  
; ----- Program start, init -----  
.CSEG ; Code Segment  
.ORG 0 ; Start at 0  
    ; Stack init for subroutine calls  
    ldi rmp,LOW(RAMEND) ; Init stack  
    out SPL,rmp ; To stack pointer  
    ; Init port outputs for LCD operation  
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)  
    out pLcdCR,rmp ; Control port outputs  
    clr rmp ; Clear outputs  
    out pLcdCO,rmp ; To control port  
    ldi rmp,mLcdDRW ; Data port output mask write  
    out pLcdDR,rmp ; To direction port  
    ; Init LCD  
    rcall LcdInit  
    ; Output text on LCD  
    ldi ZH,HIGH(2*TextTable)  
    ldi ZL,LOW(2*TextTable)  
    rcall LcdText ; Output text at ZH:ZL  
    ; Read byte from EEPROM and output in line 3  
    rcall EepRead  
    inc rEep ; Increase EEPROM byte  
    rcall EepWrite ; Write to EEPROM  
    ; Sleep enable  
    ldi rmp,1<<SE ; Sleep mode idle  
    out MCUCR,rmp ; to master control port
```

```

Loop:
    sleep ; go to sleep
    rjmp Loop
;
; Output text to LCD
TextTable:
.db "LCD display ATtiny24",0x0D,0xFF ; Line 1
.db "avr-asm-tutorial.net",0x0D,0xFF ; Line 2
.db "Previous count =      ",0x0D,0xFF ; Line 3
.db "New count      =      ",0xFE,0xFF ; Line 4
;
; Read EEPROM
EepRead:
; Wait until EEPROM is ready
    sbic EECR, EEPE ; Check EEPROM Program Enable bit
    rjmp EepRead ; not yet done
; Set EEPROM read address
    ldi rmp,HIGH(EepAddress) ; MSB to address port
    out EEARH,rmp ; ATtiny24/44 has less than 257 byte EEPROM
    ldi rmp,LOW(EepAddress) ; LSB address EEPROM
    out EEARL,rmp ; to LSB address port
; Set EERE bit in EEPROM control port
    sbi EECR,EERE ; Read enable
; Read byte from data port
    in rEep,EEDR
; Convert number read to decimal ASCII
    mov R0,rEep ; Copy number to R0
    ldi XH,HIGH(Number) ; SRAM buffer pointer for result, MSB
    ldi XL,LOW(Number) ; dto., LSB
    rcall Decimal ; Convert to decimal ASCII
    ldi XH,HIGH(Number) ; SRAM buffer pointer for output, MSB
    ldi XL,LOW(Number) ; dto., LSB
    ldi ZH,2 ; To line 3
    ldi ZL,17 ; in column 17
    ldi rmp,3 ; Three characters
    rcall LcdSram ; Call include routine SRAM out
    ret
;
; Write to EEPROM
EepWrite:
; Wait until EEPROM ready
    sbic EECR, EEPE ; Check Program Enable bit in control port
    rjmp EepWrite ; not yet ready
; Set EEPROM write mode
    ldi rmp,(0<<EEPM1)|(0<<EEPM0) ; Erase and write
    out EECR,rmp ; To EEPROM control port
; Address to Address port
    ldi rmp,HIGH(EepAddress) ; MSB
    out EEARH,rmp ; For ATtiny24/44 not necessary
    ldi rmp,LOW(EepAddress) ; LSB
    out EEARL,rmp ; to address port
; Byte to be written to data port
    out EEDR,rEep ; To the data port
; Set Memory Program enable EEMPE
    sbi EECR,EEMPE ; Enable write
; Write to EEPROM with EEPE = One
    sbi EECR,EEPE ; Start write, duration 3.4 ms
; Write new counter to LCD
    mov R0,rEep ; Copy number to R0
    ldi XH,HIGH(Number) ; SRAM buffer pointer for result, MSB
    ldi XL,LOW(Number) ; dto. LSB
    rcall Decimal ; Convert to decimal ASCII
    ldi XH,HIGH(Number) ; SRAM buffer pointer for output, MSB
    ldi XL,LOW(Number) ; dto., LSB
    ldi ZH,3 ; To line 4
    ldi ZL,17 ; To column 17
    ldi rmp,3 ; Three characters
    rcall LcdSram ; Call included routine
    ret
;
; Convert R0 to a decimal number in ASCII code
; XH:XL = Position in SRAM
Decimal:
    set ; Set T-Flag for leading zeroes
    ldi rmp,100 ; Decimal 100
    clr R1 ; R1 is result counter
Decimal100:

```

```

cp R0,rmp ; Smaller than 100?
brccs Decimal2 ; yes
sub R0,rmp ; Subtract 100
inc R1 ; Increase result counter
rjmp Decimal100 ; Continue 100s
Decimal2: ; Blank leading zeroes
tst R1 ; Result = 0?
brne Decimal3 ; No
ldi rmp,' ' ; Blank
rjmp Decimal4 ; To SRAM buffer
Decimal3: ; No leading zero
clt ; Clear flag
ldi rmp,0x30 ; ASCII-0
add rmp,R1 ; Add 100s result
Decimal4: ; Store result in SRAM
st X+,rmp ; Write to SRAM
; Check 10s
ldi rmp,10 ; Decimal 10
clr R1 ; Clear result
Decimal10: ; 10s loop
cp R0,rmp ; Compare with 10
brccs Decimal5 ; Already complete
sub R0,rmp ; Subtract 10
inc R1 ; next 10
rjmp Decimal10 ; Continue 10s
Decimal5: ; 10s are complete
brtc Decimal6 ; Leading zero flag is off
tst R1 ; Leading zero?
brne Decimal6 ; No
ldi rmp,' ' ; Blank
rjmp Decimal7 ; To buffer write
Decimal6:
ldi rmp,'0' ; ASCII-0
add rmp,R1 ; Add result
Decimal7:
st X+,rmp ; Store 10s in SRAM
; 1s
ldi rmp,'0' ; ASCII-0
add rmp,R0 ; Add rest
st X+,rmp ; Store in SRAM
ret
;
; Include Lcd4Busy routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

Do not forget to burn the EEPROM content in .eep after programming the flash code.

This is the result of the program (the German version). Looks perfect.



[Home](#) [Top](#) [EEPROM](#) [Decimal conversion](#) [Hardware](#) [Byte counter](#) [Word counter](#)

11.4.3 Simulation

The useful simulation of the software can verify their functions. The following uses [avr_sim](#) to simulate source code.

First we mask all LCD calls (LcdInit, LcdText, LcdSram) by ";" and re-assemble.

EEPROM view

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

This is the EEPROM content when we start simulation. The counter at address 0x0000 is cleared, the rest of the EEPROM has been cleared. The content of the EEPROM at address 0x0000 has been read. Here the 0x00 that was read is converted to decimal and written in ASCII to the SRAM. The first two characters are blanks. Those three bytes in SRAM are written to the LCD (not simulated here).

SRAM

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

Simulation

Restart	Step	Run	Stop																																													
Simulation status																																																
Prog counter = \$00000C Instructions = 53 Stackpointer = \$00DFE Clock frequ. = 1,000,000 Hz Time elapsed = 75.0 us Stop watch = 75.0 us																																																
SREG <table border="1"> <tr> <td>I</td> <td>T</td> <td>H</td> <td>S</td> <td>V</td> <td>N</td> <td>Z</td> <td>C</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </table> Update after every 1000 instructions				I	T	H	S	V	N	Z	C	0	1	0	0	0	0	0	0																													
I	T	H	S	V	N	Z	C																																									
0	1	0	0	0	0	0	0																																									
Register <table border="1"> <tr> <th>Reg</th> <th>+0</th> <th>+1</th> <th>+2</th> <th>+3</th> <th>+4</th> <th>+5</th> <th>+6</th> <th>+7</th> </tr> <tr> <td>R0</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> </tr> <tr> <td>R8</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> <td>00</td> </tr> <tr> <td>R16</td> <td>03</td> <td>00</td> <td>00</td> <td>00</td> <td>01</td> <td>00</td> <td>00</td> <td>00</td> </tr> <tr> <td>R24</td> <td>00</td> <td>00</td> <td>60</td> <td>00</td> <td>00</td> <td>00</td> <td>11</td> <td>02</td> </tr> </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	03	00	00	00	01	00	00	00	R24	00	00	60	00	00	00	11	02
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	03	00	00	00	01	00	00	00																																								
R24	00	00	60	00	00	00	11	02																																								

The counter rEep in R20 has been increased to one.

Now the increased counter has to be written back to the EEPROM. We do this by setting the EEPROM port-registers for address and data (marked here).

EEPROM view

D=\$01 A=\$0000 MWE WE EE_RDY

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

First the Master Write Enable bit of the EEPROM has to be set.

EEPROM view

D=\$01 A=\$0000 MWE WE

	+00	+01	+02	+03	+04	+05	+06
\$0000	00	FF	FF	FF	FF	FF	FF
\$0010	FF						
\$0020	FF						
\$0030	FF						

Within four clock cycles the Write Enable bit has to be set. Programming starts.

EEPROM view

D=\$01 A=\$0000 MWE WE

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0000	00	FF														

The first stage of an atomic write process is clearing the cell to all ones. Roughly at half of the time the content is cleared (the content has all ones).

EEPROM view

D=\$01 A=\$0000 MWE WE

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0000	FF															

The content of the EEPROM cell at address 0x0000 has changed after the write process is completed.

EEPROM view

D=\$01 A=\$0000 MWE WE

	+00	+01	+02	+03	+04	+05	+06
\$0000	01	FF	FF	FF	FF	FF	FF

The time required for erase-and-write is roughly 3.5 ms.

Simulation

Restart Step Run Stop

Simulation status

Prog counter = \$0000FC
 Instructions = 2,591
 Stackpointer = \$00D6
 Clock frequ. = 1,000,000 Hz
 Time elapsed = 3.534 ms
 Stop watch = 3.534 ms

After restarting the code, without overwriting the EEPROM, the counter is at one and converted to ASCII in the SRAM.

The screenshot shows two windows from AVR Studio. The top window is titled 'SRAM' and displays memory starting at address \$0060. The first byte at \$0060 is \$20 (32 in decimal), the second is \$20 (32 in decimal), and the third is \$31 (49 in decimal, representing the ASCII character '1'). The bottom window is titled 'Simulation' and shows the state of the SREG register and the Register window. The SREG register has bit I set (value 1). The Register window shows R0 with value 01, R16 with value 03, and R24 with value 60. The value 02 is highlighted in yellow in the R16 register's +4 column.

The second increase, of course, leads to 2 in rEep in R20. And so on, and so on, ...

[Home](#) [Top](#) [EEPROM](#) [Decimal conversion](#) [Hardware](#) [Byte counter](#) [Word counter](#)

11.5 Word counter

11.5.1 Task

The previous program shall be changed, so that after reaching a maximum counter value the chip shall discontinue to work (planned obsolescence). To being able to define higher limits a 16 bit counter shall be used. On reaching the limit, line 4 of the LCD shall display a message.

11.5.2 The program for that task

This is the program, the [source code is here](#). It requires the [LCD routines](#).

```
;
; ****
; * To read and write an EEPROM word and a message on LCD *
; * (C)2017 by http://www.avr-asm-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Ports, port bits -----
; (All LCD ports are defined in the include routine)
;
; ----- Registers -----
; Used: R0 to R3 locally by character definition and decimal conversion
; free: R4 .. R15
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Additional multi purpose register
.def rLine = R18 ; Line counter LCD
.def rRead = R19 ; Read result from LCD data port
.def rEepH = R20 ; MSB data for EEPROM counter
```

```

.def rEepL = R21 ; dto., LSB
; free: R22 .. R25
; Used: XH:XL R27:R26 Pointer
; free: YH:YL R29:R28
; Used: R31:R30, ZH:ZL, for counting and LCD
;
; ----- Constants -----
.equ Clock = 1000000 ; Default clock frequency
.equ EepAddress = 0 ; Read and write address EEPROM
.equ Obsolescence = 3 ; Obsolescence criterion 1..65535
;
; ----- SRAM -----
.DSEG ; Assemble to data segment
.ORG 0x0060 ; to SRAM start
Number: ; Conversion word to decimal ASCII
.BYTE 5 ; needs five characters
;
; ----- EEPROM init value -----
.ESEG ; Assemble EEPROM segment
.ORG EepAddress
.db LOW(32767),HIGH(32767) ; Init EEPROM counter
;
; ----- Program start, init -----
.CSEG ; Assemble to code Segment
.ORG 0 ; Start at 0
    ; Stack init for subroutine calls
    ldi rmp,LOW(RAMEND) ; Init Stack
    out SPL,rmp ; to stack pointer
    ; Init port outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to control port directions
    clr rmp ; Clear outputs
    out pLcdCO,rmp ; to control port output
    ldi rmp,mLcdDRW ; Data port output mask write
    out pLcdDR,rmp ; to direction port
    ; Init LCD
    rcall LcdInit ; Call included init routine
    ; Output text on LCD
    ldi ZH,HIGH(2*TextTable)
    ldi ZL,LOW(2*TextTable)
    rcall LcdText ; Output flash text at ZH:ZL
    ; Read word from EEPROM und display in line 3
    rcall EepReadW
    inc rEepL ; Increase EEPROM LSB
    brne MsbOk ; No overflow
    inc rEepH ; Overflow to MSB
MsbOk:
    ; Check obsolescence criterion
    rcall ChkObsolescence; End, output message
    brcc Obsolet ; System is obsolete
    rcall EepWriteW ; Write to EEPROM and to line 4
Obsolet:
    ; Sleep enable
    ldi rmp,1<<SE ; Sleep mode idle
    out MCUCR,rmp ; to master control port
Loop:
    sleep ; go to sleep
    rjmp Loop
;
; Output text for LCD
TextTable:
.db "LCD display ATtiny24",0x0D,0xFF ; Line 1
.db "avr-asm-tutorial.net",0x0D,0xFF ; Line 2
.db "Previous      =",0x0D,0xFF ; Line 3
.db "New count     =",0xFE,0xFF ; Line 4
;
; Check Obsolescence
ChkObsolescence:
    cpi rEepL,LOW(Obsolescence) ; Compare with constant
    ldi rmp,HIGH(Obsolescence)
    cpc rEepH,rmp ; With overflow from LSB compare
    brcs ChkObsolescenceRet
    ; End of service reached
    rcall LcdLine4 ; To line 4
    ldi ZH,HIGH(2*ObsoletText) ; Text message
    ldi ZL,LOW(2*ObsoletText)
    rcall LcdTextC ; Output text on LCD

```

```

    clc ; Clear carry flag
ChkObsolescenceRet:
    ret
; Text of the message
ObsoleteText:
.db "* Maximum reached! *",0xFE,0xFE
;
; Read EEPROM word
EepReadW:
; Wait until EEPROM is ready
    sbic EECR, EEPE ; Check EEPROM Program Enable bit
    rjmp EepReadW ; not yet ready
; EEPROM read address
    ldi rmp,HIGH(EepAddress) ; MSB to address port
    out EEARH,rmp ; ATTiny24/44 has less than 257 Byte EEPROM
    ldi rmp,LOW(EepAddress) ; LSB
    out EEARL,rmp ; to LSB address port
; Set EERE bit in EEPROM control port
    sbi EECR,EERE
; Read byte from data port
    in rEepL,EEDR ; read result to LSB
    cbi EECR,EERE ; Read bit off
    ldi rmp,LOW(EepAddress+1) ; Address next byte
    out EEARL,rmp ; to LSB address port
    sbi EECR,EERE ; Set Read Enable bit
    in rEepH,EEDR ; Read MSB from data port
; Convert binary read to decimal ASCII
    mov R0,rEepL ; Copy counter to R1:R0
    mov R1,rEepH
    ldi XH,HIGH(Number) ; SRAM buffer for result, MSB
    ldi XL,LOW(Number) ; dto. LSB
    rcall DecimalW ; Convert word to decimal ASCII
    ldi XH,HIGH(Number) ; SRAM buffer pointer for output, MSB
    ldi XL,LOW(Number) ; dto., LSB
    ldi ZH,2 ; In lin 3
    ldi ZL,15 ; Column 16
    ldi rmp,5 ; Five characters
    rcall LcdSram ; Call SRAM output to LCD include routine
    ret
;
; Write to EEPROM
EepWriteW:
; Wait until EEPROM ready
    sbic EECR,EEPE ; Check Program Enable bit in control port
    rjmp EepWriteW ; not yet ready
; Set EEPROM write mode
    ldi rmp,(0<<EEP1)|(0<<EEP0) ; Erase and write
    out EECR,rmp ; to EEPROM control port
; Address to address ports
    ldi rmp,HIGH(EepAddress) ; MSB address
    out EEARH,rmp ; ATTiny24/44 has less than 257 bytes
    ldi rmp,LOW(EepAddress) ; LSB address
    out EEARL,rmp
; Byte to be written to data port, LSB
    out EEDR,rEepL ; into data port
; Set Memory Program enable bit EEMPE
    sbi EECR,EEMPE ; Enable writing
; Write EEPROM with EEPE = one
    sbi EECR,EEPE ; Start writing, duration 3.4 ms
EepWrite1:
    sbic EECR,EEPE ; Wait until ready written
    rjmp EepWrite1 ; not yet ready
    ldi rmp,LOW(EepAddress+1) ; Write MSB counter address
    out EEARL,rmp
; MSB to write into data port
    out EEDR,rEepH ; Write MSB to data port
; Set Memory Program enable bit EEMPE
    sbi EECR,EEMPE ; Enable write
; Write to EEPROM with EEPE set
    sbi EECR,EEPE ; Start write
; Display new content on LCD
    mov R0,rEepL ; Copy count to R1:R0
    mov R1,rEepH
    ldi XH,HIGH(Number) ; SRAM buffer pointer result, MSB
    ldi XL,LOW(Number) ; dto. LSB
    rcall DecimalW ; Convert to decimal ASCII
    ldi XH,HIGH(Number) ; SRAM buffer pointer for output, MSB

```

```

ldi XL,LOW(Number) ; dto., LSB
ldi ZH,3 ; To line 4
ldi ZL,15 ; In column 17
ldi rmp,5 ; Five characters
rcall LcdSram ; Call include routine output SRAM to LCD
ret

;
; Convert R1:R0 to decimal ASCII
;   XH:XL = Position in SRAM
;   uses R3:R2, ZH:ZL
DecimalW:
    set ; Set T flag for leading zero suppression
    ldi ZH,HIGH(2*DecTab) ; Z is pointer to decimal table
    ldi ZL,LOW(2*DecTab)

DecimalW1:
    lpm R2,Z+ ; Read LSB from decimal table
    tst R2 ; Is LSB zero?
    breq DecimalWOnes ; yes, finished
    lpm R3,Z+ ; Read MSB from decimal table
    clr rmp ; rmp is counter

DecimalW2:
    sub R0,R2 ; Subtract LSB
    sbc R1,R3 ; Subtract MSB and carry from LSB
    brcc DecimalW3 ; Too far
    inc rmp ; Increase digit counter
    rjmp DecimalW2 ; Go on subtracting

DecimalW3:
    add R0,R2 ; Get back one subtraction
    adc R1,R3
    brtc DecimalW6 ; No suppression of leading zeroes
    tst rmp ; Another leading zero?
    brne DecimalW5 ; no
    ldi rmp,' ' ; Output a blank
    rjmp DecimalW7 ; Direct output

DecimalW5:
    clt ; Not zero, clear T flag

DecimalW6:
    subi rmp,-'0' ; Add ASCII-0

DecimalW7:
    st X+,rmp ; Store in SRAM buffer
    rjmp DecimalW1 ; Next decimal digit

DecimalWOnes:
    ldi rmp,'0' ; ASCII-0
    add rmp,R0 ; Add ones
    st X+,rmp ; Store in SRAM
    ret

;
; Decimal table
DecTab:
.dw 10000 ; Ten thousands
.dw 1000 ; Thousands
.dw 100 ; Hundreds
.dw 10 ; Tens
.dw 0 ; Table end
;
; Include Lcd4Busy routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

New is the instruction CPC register, register. It compares the first register with the second one, as increased by the carry flag. Flags are resulting similarly as with the instruction SBC.

Home	Top	EEPROM	Decimal conversion	Hardware	Byte counter	Word counter
----------------------	---------------------	------------------------	------------------------------------	--------------------------	------------------------------	------------------------------

11.5.3 Simulation

Simulation is again done with [avr_sim](#). Please comment the lcd routine calls out before simulation.

EEPROM view							
D=\$00	A=\$0000	<input type="checkbox"/> MWE	<input type="checkbox"/> WE				
	+00	+01	+02	+03	+04	+05	+06
\$0000	FF	00	FF	FF	FF	FF	FF
\$0010	FF	FF	FF	FF	FF	FF	FF
\$0020	FF	FF	FF	FF	FF	FF	FF
\$0030	FF	FF	FF	FF	FF	FF	FF
\$0040	FF	FF	FF	FF	FF	FF	FF
\$0050	FF	FF	FF	FF	FF	FF	FF

The EEPROM here holds the 16 bit counter in 0x0000 and 0x0001. By .ESEG it has been set to 0x00FF at the beginning.

The two bytes have been read from the EEPROM and have been placed to R20:R21, with the MSB in R20.

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	05	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	05	00	00	00	00	00	FF	00
R24	00	00	60	00	00	00	0F	02

The decimal in SRAM now needs five bytes, because numbers of up to 65,535 can occur.

SRAM																ASCII text	
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	20	20	32	35	35	FF	255									
\$0070	FF															
\$0080	FF															
\$0090	FF															
\$00A0	FF															
\$00B0	FF															
\$00C0	FF															
\$00D0	FFh..															

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	05	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	FF	00	00	00	01	00	00	00
R24	00	00	60	00	00	00	0F	02

The value of the counter has been increased and written to SRAM (and to the LCD, which is not simulated here).

First the LSB of the increased counter is written to address 0x0000 of the EEPROM.

EEPROM view																EE_RDY	
D=\$01	A=\$0001	<input type="checkbox"/> MWE	<input type="checkbox"/> WE													<input type="checkbox"/> EE_RDY	
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0000	00	00	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

EEPROM view

D=\$01	A=\$0001	<input type="checkbox"/> MWE	<input type="checkbox"/> WE
	+00 +01 +02 +03 +04 +05 +06 +07		
\$0000	00 01 FF FF FF FF FF FF		
\$0010	FF FF FF FF FF FF FF FF		
\$0020	FF FF FF FF FF FF FF FF		
\$0030	FF FF FF FF FF FF FF FF		
\$0040	FF FF FF FF FF FF FF FF		
\$0050	FF FF FF FF FF FF FF FF		

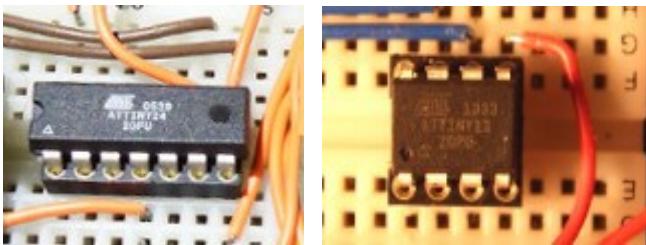
After completion of the write process the MSB is written to the EEPROM.

Here both bytes have been written to EEPROM.

Roughly 7 ms have elapsed until both bytes are written to the EEPROM.

Simulation

Restart	Step	Run	Stop
Simulation status			
Prog counter = \$000013			
Instructions = 2,562			
Stackpointer = \$00DF			
Clock frequ. = 1,000,000 Hz			
Time elapsed = 7.099 ms			
Stop watch = 7.099 ms			



Lecture 12: IR receiver and transmitter

Getting practical: to receive, monitor and analyze infrared remote control signals and to generate our own infrared signals with a transmitter. To reduce the number of source code variations conditioned assembly is used here to some extend.

12.0 Overview

1. [Introduction to infrared signals](#)
2. [Introduction to conditioned assembly](#)
3. [Hardware, components, mounting](#)
4. [To measure infrared signals](#)
5. [An IR transmitter](#)
6. [An IR data transmit/receive system](#)
7. [A self-learning IR three-channel switch](#)

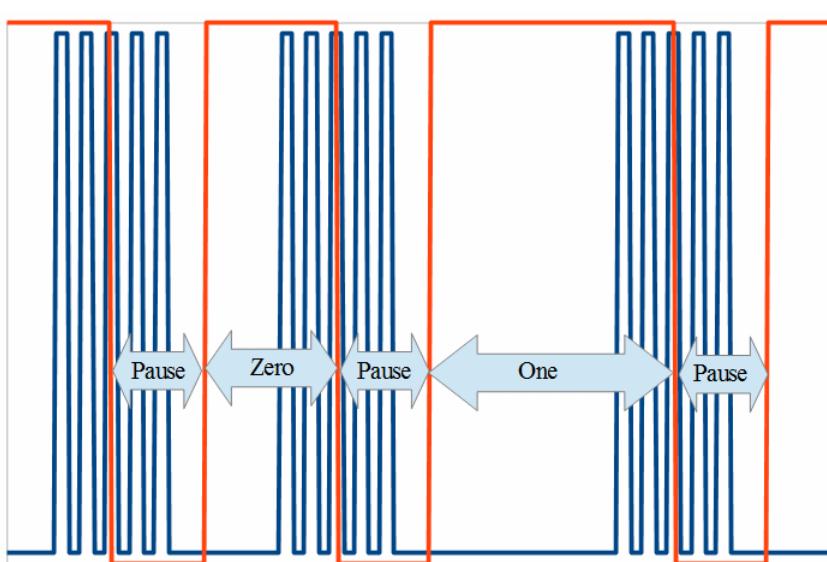
12.1 Introduction to infrared signals

The small well-designed boxes that cover our living room table and that wait until a key is pressed, are majorly unknown boxes. Not many people know how those work and how they control a TV or a CD player via invisible rays. In the earlier years, without those boxes, we had to stand up from the sofa and push the buttons on those devices manually. The remote control boxes allow to stay on the sofa and to push secondary buttons.

Infrared means that those work with heat rays and that we cannot see those signals. As those small boxes transmit with only 250 mW, we cannot sense that with our skin (which is normally our built-in infrared sensor). But even if we could feel and see those rays we would not see a lot because those boxes work only for a few milliseconds, to switch the TV to channel 15.

In order to discriminate those rays from boxes from other infrared generators (such as those from a cup of hot tea), those infrared rays are switched on-and-off in a very fast rhythm. This on-and-off is on bat frequencies of 35 to 56 kcs/s, so even if we would see infrared we would see nothing but a short burst.

In the receiver those on-and-off signals are filtered out and selectively amplified so that the TV even on a hot summer-day with lots of infrared rays understands the command to switch to channel 15. Such remote control signals look like this (only parts of the signal are displayed here).



While the IR LED is off, the receiver (in red) does not identify a signal, his output is inactive (high!). The LED then is switched on and off e.g. with 40 kcs/s (12.5 µs on, 12.5 µs

off). With the delay of the filter the receiver detects the signal and the output follows by going low, it goes active. The data transmitted is encoded in the duration of the inactive phase. Between two active phases represent between 15 and 30 a binary zero and 50 to 130 a binary one.

The design decision for those signals, to encode the ones and zeroes in the inactive period duration and to leave the active LED signal for the pauses is positive for the battery life-time: zeroes and ones consume the same LED current, zero.

Why we always use the term "e.g." is caused by the fact that every producer of IR remote controls has its own special design for the signals. The design goal is rather to have a unique signal design to avoid that different boxes on the living-room table have a conflict and that switching the TV to channel 15 increases the loudness of the tuner/amplifier of the CD player, too. That is why any device brings its own box with it, and why nobody has invented a "one-box-for-all" solution.

We will have to care about this absence of industrial norms later on.

[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.2 Introduction to conditioned assembly programming

Version 2 of ATMEL's assembler offered the opportunity to define conditions under which the source code is assembled or not. Directives were introduced that allow to control the assembly flow. This makes sense if the basic structure of the source code remains the same and only selected and limited portions of the code need to be changed to fit to other needs.

12.2.1 To define conditions

By using the following formulation the following source code is only assembled if the condition is "1":

```
.equ Switch = 1 ; Define a switch
; [...]
.if Switch == 1
    ; [Assemble this part of the source code]
    .endif
; [...]
```

The double "==" in the .if condition means that this is a logic decision (that is either true or false), while our Switch can as well be zero, two or three. Again: this decision is made during assembling, the controller will not see and has nothing to do with that decision and with the .if and .endif directive (this decision would involve the instruction BREQ, e.g.). Any .if directive must be followed by an .endif directive, otherwise none of the following code is assembled and the assembler ends with an error message (open condition ...).

If you want to switch on the opposite condition, two methods come into question:

```
.if Switch != 1
    ; [Assemble this part if Switch is not one]
    .endif
; [...]
```

where "!=" means "not equal". The same effect would have

```
.if Switch == 0
    ; [Do not assemble this part in case Switch is 1]
    .endif
; [...]
```

12.2.2 If-then-else alternatives

If the switch shall select between two alternative paths to assemble the .else directive can be used. The following formulation assembles for different polarities of an input pin:

```
.equ Switch = 1
; [...]
.if Switch == 1
    sbic PINB,PIN0 ; Skip next instruction if input pin is low
    .else
        sbis PINB,PIN0 ; Skip next instruction if input pin is high
    .endif
    rjmp PinConditionNotFulfilled
```

The code behind the .else directive is assembled if the .if condition is not met. This formulation can get rather intransparent because it mixes assembler conditions and conditioned jump instructions. In such cases it can help to consult the listing that the assembler produces, in order to reduce complexities by one level.

In the .else case it can be sometimes appropriate to decide with an additional condition if assembling should be performed. This can be done by using the .elif directive instead of .else. By doing this requires some attention because it can be that neither source code is assembled. An example to fit a source code as well for an ATtiny13 and an ATtiny24 is:

```
.equ cType == 13 ; can be 13 or 24
; [...]
.if cType == 13
    ; Place Reset- and Int-vector table for ATtiny13 here
.elif cType == 24
    ; Place Reset- and Int-vector table for ATtiny24 here
.endif
```

If the user now neither sets cType to 13 nor to 24, no reset and int vector table will be in the assembled code and that could have funny effects.

To avoid this the assembler gavrasm has an additional directive .ifdevice Type. Here you can explicitly formulate .IFDEVICE "ATtiny13" or .IFDEVICE ATTINY13.

12.2.3 Other helpful directives

The following outputs an error message during assembling if neither 13 nor 24 is defined as device:

```
.if (cType != 13) && (cType != 24)
    .error "Wrong type!"
    .endif
```

This directive can further be used to provoke an error message if a constant exceeds certain value ranges (e.g. if the constant is larger than 255).

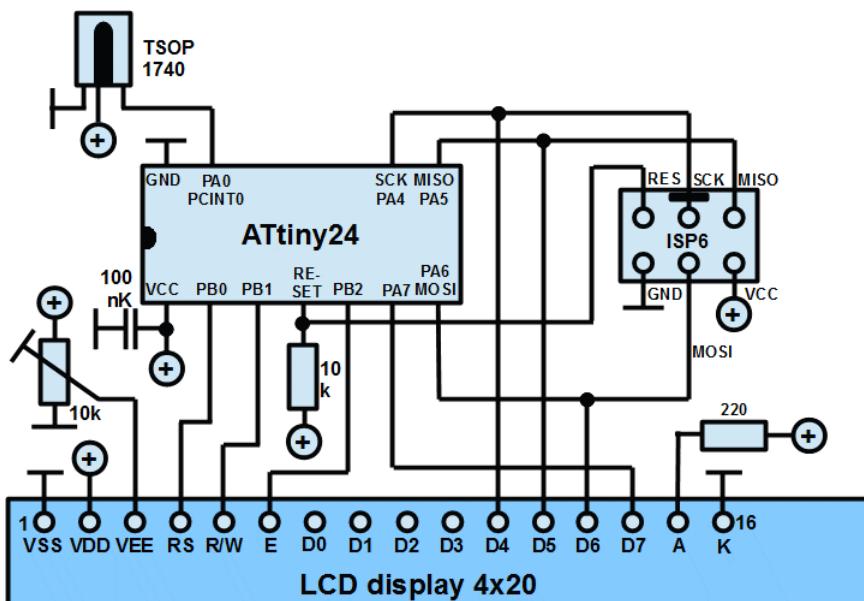
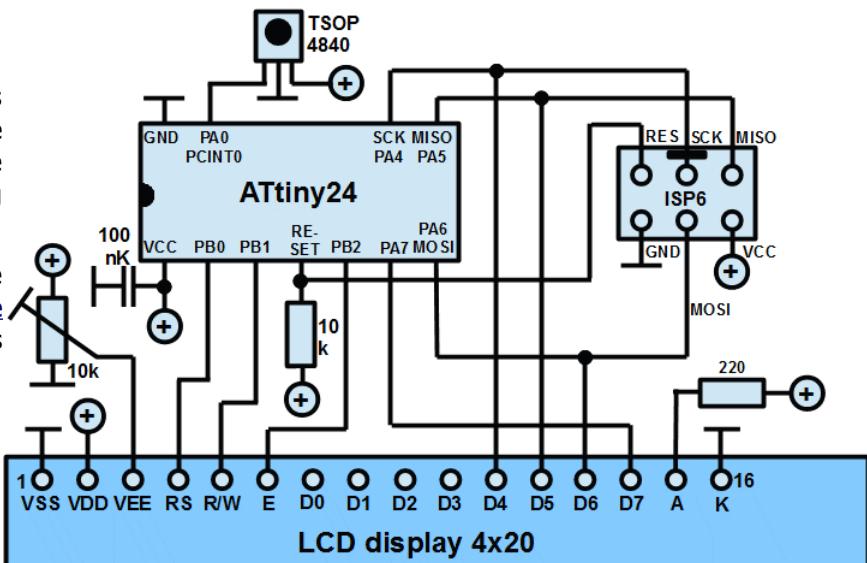
If only a message shall be outputted, but assembling shall continue, the directive .MESSAGE "Message text" provides that.

12.3 Hardware, components and mounting

12.3.1 Schemes

The schemes for the two types of IR receiver modules are slightly different. Beyond the IR receiver module nothing else changes in the scheme.

Of course, you can also use the experimental board [here](#) instead, without any changes to the source code.



12.3.2 Components: the IR receiver modules

Both receiver modules are necessary for different remote control types. TSOP4840 is best for older, TSOP1740 for newer remote control devices. In practise this difference is irrelevant, as I realized with my boxes of different ages.

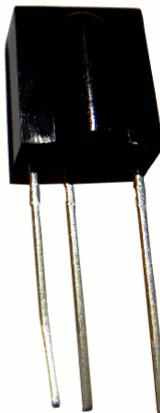
The pinning of both types is different. The signal output is to be connected to PA0 of the ATtiny24. That is it.

Both devices are active low, which means that an actively detected IR signal brings the output to low level.

TSOP 4840



TSOP 1740



12.3.3 Mounting

Mounting is trivial.

[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.4 Measuring IR signals

12.4.1 Task

Besides the feeling that it would be nice to know how those small boxes work it could be helpful to analyze IR signals in case one likes to build his own. In that case we can switch the TV to our own channel 15 without being depending from others in our living room, who might prefer other channels. This requires some packages of research software which we generate here. This also will help us to analyze our own IR transmitter later on in this course.

12.4.2 Start signals

12.4.2.1 Task

The first task is to analyze start signals of remote control boxes.

12.4.2.2 Examples for IR signal structures

IR-Analyse ATtiny24
gsc-elektronik.net
Messungen IR-Signal
in Hex mal 8 µs -

That is how the software looks like at the start (the German version). All following software displays values in hexadecimal format. You will see what that is good for later on. Who wants to have those in decimal format can use a hex calculator or uses a spread sheet application like OpenOffice to convert hex to decimal.

All values have to be multiplied with 8 because the timer used to measure times runs with a prescaler of 8 at 1 Mcs/s.

The software now waits for signals from the IR receiver module. After 16 level changes have been found, the measured times are displayed.

■1696,0232 „0229,0049
„00C9,0049 „00C9,0049
„00C8,0048 „003F,0049
„003F,0048 „003F,0048

The signal starts with a long, inactive pause of $0x1696 * 8 \mu s$ duration. Then an active LED period of $0x0232 * 8 \mu s$ follows (input signal is low, L). This is followed by pause periods (input signal is high, H) of $0xC8 * 8 \mu s$ or $0x3F * 8 \mu s$, in between active signals of $0x49 * 8 \mu s$ duration. Looks pretty simple.

■2404,01B0 „00D2,003E
„002C,003A „009C,0041
„0029,003E „002D,003D
„002C,003C „002C,003D

This looks a little more complex, it stems from an Panasonic Harddrive Recorder. The signal start has a long High period, a shorter Low period, a half-as-long High period and a slightly shorter Low period. The signal pauses (H) are all between $0x0029$ and $0x002D$, one High signal is at $0x009C$. The active signal periods are between $0x3C$ and $0x41$.

```

13B0,045B,021D,004F
,00C3,004B,00C5,004B
,003C,004F,00C2,004B
,003C,004C,003B,004C

```

Those are the signals from a camera control. The header structure is similar to the two devices above, but the durations are different.

Signal#	H/L	TV	μs	N	L+H	HDR	μs	N	L+H	Camera	μs	N	L+H
1	H	1696	46.256	3700		2404	73.760	5.901		13B0	40.320	3.226	
	L	0232	4.496	360		01B0	3.456	276		045B	8.920	714	
2	H	0229	4.424	354		00D2	1.680	134		021D	4.328	346	
	L	0049	584	47	176	003E	496	40	68	004F	632	51	176
3	H	00C9	1.608	129		002C	352	28		00C3	1.560	125	
	L	0049	584	47	176	003A	464	37	137	004B	600	48	174
4	H	00C9	1.608	129		009C	1.248	100		00C5	1.576	126	
	L	0049	584	47	175	0041	520	42	68	004B	600	48	86
5	H	00C8	1.600	128		0029	328	26		003C	480	38	
	L	0048	576	46	86	003E	496	40	69	004F	632	51	175
6	H	003F	504	40		002D	360	29		00C2	1.552	124	
	L	0049	584	47	87	003D	488	39	67	004B	600	48	86
7	H	003F	504	40		002C	352	28		003C	480	38	
	L	0048	576	46	86	003C	480	38	66	004C	608	49	87
8	H	003F	504	40		002C	352	28		003B	472	38	
	L	0048	576	46		003D	488	39		004C	608	49	
Average				64				40				64	
Average L				47				39				49	
Average H				129				100				125	

These are the example data in a spreadsheet. The header signals are with 50, 76 resp. 50 ms duration the longest. data bits require 0.7 to 1.0 ms.

An additional value N is listed. Those would be the number of timer periods to perform the on-and-off of a LED with 40 kcs/s in an active period using the CTC mode of the timer.

12.4.2.3 The program

In principal the program measures the duration between rising and falling as well as between falling and rising levels. For that the 16 bit timer TC1 is used, that is read and then cleared again.

12.4.2.4 Program structure

The program works in the following stages:

- First the stack is initiated, the ports are initiated, the LCD is initiated in 4 bit mode, the special characters for the LCD are initiated and the start text written to the LCD. The pointer Y (R29:R28) is set to the buffer start for received signals in the SRAM. Timer 1 runs in normal mode with a prescaler of 8. The input pin, where the IR module is attached, has Pin Change Interrupts enabled.
- If a level change interrupt occurs on the IR input pin, it is first checked if the input pin is high or low. If the input is low, it was high before and the IR receiver module was inactive. To always store complete active/inactive and high/low pairs, the signal durations are stored with the STD Y+N,register instruction. The storage row is: MSB-inactive, LSB inactive, MSB active, LSB active. This mode is necessary because in a 16 bit timer the LSB has to be read first because this read simultaneously transfers the timer's MSB to an internal buffer, from which it can be read then. This mechanism assures that the LSB/MSB pair read is correct and reflects exactly the state of the counter when the LSB was read first. Otherwise the timer would have counted further and between the first and the second read operation there might have been an overflow from the LSB to the MSB. The pointer Y to the SRAM buffer is only increased (by four) after a complete set has been written. If the SRAM buffer is full, the display flag is set, to be processed outside the interrupt service routine within the main loop. The timer count is finally cleared in the service routine.
- The output to the LCD display is performed in the main loop if the respective flag is activated by the PCINT service routine.

12.4.2.5 Program

This is the program, [the source code file is here](#). To assemble this requires [the LCD routines](#) in the same folder. Please note that the switch "FromBehind" also allows to display the last received data pairs when set to one (conditional assembly).

```
;  
; *****  
; * IR receiver with ATTiny24 and LCD to measure header durations *  
; * (C)2017 by http://www.avr-asm-tutorial.net  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Switch -----  
.equ FromBehind = 0 ; 0: Header signals  
; 1: The last signals  
;  
; ----- Ports -----  
.equ pIrIn = PINA ; IR mudule port  
.equ bIrIn = 0 ; IR module pin  
;  
; ----- Timing -----  
; Processor clock 1.000.000 cs/s  
; Time per clock cycle 1 us  
; Prescaler T1 8  
; Time per timer tick 8 us  
; TC1 count after 1 ms 125  
; TC1 overflow after 524 ms  
;  
; ----- Registers -----  
; Used: R0 by LCD and for decimal conversion  
; Used: R1 for decimal conversion  
; free: R2..R14  
.def rSreg = R15 ; Save/Restore SREG  
.def rmp = R16 ; Multi purpose register  
.def rmo = R17 ; Second multi purpose register  
.def rLine = R18 ; LCD line counter  
.def rRead = R19 ; LCD read register  
.def rimp = R20 ; Interrupt multi purpose  
.def rFlag = R21 ; Flag register  
    .equ bUpd = 0 ; Update display flag  
    .equ bShf = 1 ; Shift backwards flag  
; free: R22 .. R25  
; Used: XH:XL R27:R26 for shifting backwards  
; Used: YH:YL R29:R28 Pointer to SRAM  
; Used: ZH:ZL R31:R30 in LCD routines  
;  
; ----- SRAM -----  
.DSEG ; Data segment  
.ORG 0x0060 ; at beginning of SRAM  
Buffer:  
.byte 32 ; Buffer for receiver data  
BufferEnd:  
Last:  
.byte 4 ; Last measured value  
LastEnd:  
;  
; ----- Reset- and interrupt vectors -----  
.CSEG ; Code segment  
.ORG 0 ; to the start  
    rjmp Start ; Reset vector, init  
    reti ; INT0 External Interrupt Request 0  
    rjmp PcioIsr ; PCINT0 Pin Change Interrupt Request 0  
    reti ; PCINT1 Pin Change Interrupt Request 1  
    reti ; WDT Watchdog Time-out  
    reti ; TIM1_CAPT Timer/Counter1 Capture Event  
    reti ; TIM1_COMPA Timer/Counter1 Compare Match A  
    reti ; TIM1_COMPB Timer/Counter1 Compare Match B  
    reti ; TIM1_OVF Timer/Counter1 Overflow  
    reti ; TIM0_COMPA Timer/Counter0 Compare Match A  
    reti ; TIM0_COMPB Timer/Counter0 Compare Match B  
.if FromBehind == 1 ; Last signals only
```

```

rjmp Tc0Isr ; TIM0_OVF TCO Overflow, update display
.else
; Header signals only
reti ; TIM0_OVF Timer/Counter0 Overflow
.endif
reti ; ANA_COMP Analog Comparator
reti ; ADC_ADC Conversion Complete
reti ; EE_RDY EEPROM Ready
reti ; USI_STR USI START
reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr: ; PCINT0 interrupt
    in rSreg,SREG ; Save SREG
    sbic pIrIn,bIrIn ; Skip next instruction when input low
    rjmp Pci0Isr1 ; Input is high
    ; Input is low, was high before
    in rimp,TCNT1L ; Low byte first
    std Y+1,rimp ; LSB to SRAM byte 1
    in rimp,TCNT1H ; High byte next
    st Y,rimp ; MSB to SRAM byte 0
.if FromBehind == 0 ; Assemble only if direction is forward
    cpi rimp,0x05 ; Check for signal pause
    brcs Pci0Isr2 ; Not a signal pause
    ldi YH,HIGH(Buffer) ; Pause, Y to SRAM buffer start
    ldi YL,LOW(Buffer)
    in rimp,TCNT1L ; Read again, low Byte first
    std Y+1,rimp ; LSB to SRAM buffer byte 1
    in rimp,TCNT1H ; High byte next
    st Y,rimp ; MSB to SRAM Byte 0
.endif
    rjmp Pci0Isr2 ; Done, clear TC1 count
Pci0Isr1:
    ; Input is high, was low before
    in rimp,TCNT1L ; Low byte first
    std Y+3,rimp ; LSB to SRAM byte 3
    in rimp,TCNT1H ; High byte next
    std Y+2,rimp ; MSB to SRAM byte 2
.if FromBehind == 1 ; Assembled only when backwards
    sbr rFlag,1<<bShf ; Set shift flag
.else
    ; Assemble when forward
    adiw YL,4 ; Buffer pointer four bytes further
    cpi YL,LOW(BufferEnd) ; 32 bytes = 8 Low/High pairs
    brcs Pci0Isr2 ; Not yet reached
    sbr rFlag,1<<bUpd ; Set update flag
    ldi YH,HIGH(BufferEnd) ; Y to end of buffer
    ldi YL,LOW(BufferEnd)
.endif
Pci0Isr2:
    ldi rimp,0 ; Restart TC1 counter
    out TCNT1H,rimp ; First the MSB
    out TCNT1L,rimp ; Next the LSB
    out SREG,rSreg ; Restore SREG
    reti ; fertig
;
; TCO overflow Interrupt Service Routine
Tc0Isr:
    in rSreg,SREG ; Save SREG
    sbr rFlag,1<<bUpd ; Set update flag
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Start, Init -----
Start:
    ldi rmp,LOW(RAMEND) ; Stack to RAMEND
    out SPL,rmp ; to stack pointer
    ; Init I/O port pins
    ; Init LCD port outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; LCD Control port outputs
    clr rmp ; Outputs off
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; Data port output for write
    out pLcdDR,rmp ; to direction port
    ; Init LCD
    rcall LcdInit ; Init of the LCD, call include

```

```

ldi ZH,HIGH(2*CodeChars) ; Special characters
ldi ZL,LOW(2*CodeChars)
rcall LcdChars ; Write those to LCD
ldi ZH,HIGH(2*LcdStart) ; Output start text
ldi ZL,LOW(2*LcdStart)
rcall LcdText ; call include routine
; Start value for buffer pointer
.if FromBehind == 1 ; Backwards
    ldi YH,HIGH(Last) ; Behind buffer end
    ldi YL,LOW(Last)
.else
; Forward
    ldi YH,HIGH(Buffer) ; Y to buffer start
    ldi YL,LOW(Buffer)
.endif
; Start value for flags
    clr rFlag
.if FromBehind == 1 ; For backwards only
; Init timer TCO
    ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler = 1024
    out TCCR0B,rmp ; Start timer
.endif
; Init timer TC1, free running with prescaler = 8
    ldi rmp,1<<CS11 ; Prescaler 8
    out TCCR1B,rmp ; to timer control port B
; PCINT0 for IR Rx module
    ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
    out PCMSK0,rmp ; to PCINT0 mask
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
    out GIMSK,rmp ; in General Interrupt mask port
; Sleep Enable
    ldi rmp,1<<SE ; Sleep mode Idle
    out MCUCR,rmp ; in MCU control port
; Enable interrupts
    sei ; Set I flag in SREG

Loop:
    sleep ; go to sleep
    nop ; Wake up
    sbrc rFlag,bShf ; Skip next if shift flag is clear
    rcall Shift ; Process shift flag
    sbrc rFlag,bUpd ; Skip next if update flag is clear
    rcall Update ; Process update flag
    rjmp Loop ; go to sleep again
;
; Shift buffer content backwaerts
Shift:
    cbr rFlag,1<<bShf ; Clear shift flag
    clr rmp ; Clear TCO count
    out TCNT0,rmp
    ldi rmp,1<<TOIE0 ; Disable TCO interrupt
    out TIMSK0,rmp ; in interrupt mask
    ldi XH,HIGH(Buffer+4) ; Buffer pointer to second set
    ldi XL,LOW(Buffer+4)
    ldi ZH,HIGH(Buffer) ; Buffer pointer to shift target
    ldi ZL,LOW(Buffer)
Shift1: ; Shift buffer backwards
    ld rmp,X+ ; Read byte from origin
    st Z+,rmp ; Write byte to target
    cpi ZL,LOW(BufferEnd) ; All shifted?
    brcc Shift1 ; no, go on
    ret

; Process update flag
Update:
    cbr rFlag,1<<bUpd ; Clear update flag
.if FromBehind == 1 ; Only in case of backward
    clr rmp ; Disable TCO interrupt
    out TIMSK0,rmp ; in interrupt mask port
.endif
    ldi rmp,1 ; Clear LCD
    rcall LcdC4Byte ; Write to LCD control port
    ldi ZH,HIGH(Buffer) ; Point Z to buffer start
    ldi ZL,LOW(Buffer)
    clr rLine ; Clear line counter

Update1:
.if FromBehind == 1 ; Only if backwards
    ldi rmp,'e' ; Last value pair

```

```

cpi ZL,LOW(BufferEnd-4)
brcc Update2 ; 'e' marks last value
.else
ldi rmp,0x02 ; Pause character when forward
cpi ZL,LOW(Buffer) ; First word?
breq Update2 ; Output pause character
.endif
ldi rmp,0x01 ; Output low character
sbrc ZL,1 ; Bit 2 buffer address zero?
ldi rmp,0x00 ; no, Output high character
Update2:
rcall LcdD4Byte ; Output special char
ld rmp,Z+ ; Read MSB from buffer
rcall Hex2Lcd ; Output MSB in hex
ld rmp,Z+ ; Read LSB from buffer
rcall Hex2Lcd ; Output LSB in hex
mov rmp,ZL ; Check line end
andi rmp,0x07 ; 8 bytes written?
brne Update1 ; no, go on
inc rLine ; Next line
cpi rLine,4 ; End of display lines?
brcc Update3 ; Yes, done
mov rmp,rLine ; Next line
rcall LcdLine ; To LCD
rjmp Update1 ; And repeat
Update3:
ldi rmp,0x0C ; Cursor and blink off
rjmp LcdC4Byte ; To LCD
;
; Byte in rmp in hex to LCD
Hex2Lcd:
push rmp ; Save byte on stack
swap rmp ; Upper nibble first
rcall HexNibble2Lcd ; Write Nibble
pop rmp ; Restore rmp
; Output nibble in hex on LCD
HexNibble2Lcd:
andi rmp,0x0F ; Mask lower nibble
subi rmp,-'0' ; Add ASCII-0
cpi rmp,'9'+1 ; A to F?
brcs HexNibble2Lcd1 ; No
subi rmp,-7 ; Add 7 to yield A to F
HexNibble2Lcd1:
rjmp LcdD4Byte ; Write rmp to LCD
;
; Start text LCD
LcdStart:
.db "IR analysis ATtiny24",0x0D,0xFF
.db "avr-asm-tutorial.net",0x0D,0xFF
.db " Signal duration ",0x0D,0xFF
.db " in hex Hi/Lo, 8 us",0xFE
;
; Special chars
CodeChars:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High signal
.db 80,0,0,0,0,0,0,0,7,0 ; Z = 2, Pause
.db 0,0 ; End of table
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

A new instruction that is used here is LD register,pointer. This loads the register from the location that the pointer points to, e.g. to an SRAM location. Pointers can be X, Y or Z register pairs. Variations are auto-incrementing the pointer following loading, e.g. ld R16,Z+, or auto-decrementing before loading, e.g. ld R16,-Z.

12.4.3 End signals

To find out if the IR signals have a different format on the end of the transmit cycle we need to evaluate the last signals that were sent. This requires a significant change to the program

source. Here a different way was chosen: a switch in the program head decides whether to measure and display header info or the last bytes.

```
; ----- Switch -----
.equ FromBehind = 0 ; 0: Header signals
;                                1: The last signals
```

```
h0034,h0038,h0031,h0034
h009F,h0039,h009C,h0039
h0030,h0035,h009E,h0034
h0035,h0038e009De0038
```

With `FromBehind = 1` the last arrived signals are displayed. Assemble the so changed source code and burn it into the flash, this will implement this massive change.

The last received signals are marked with a small "e". This shows that the last signals have no changed structure.

12.4.4 Number of signals

IR-Analyse ATtiny24
asc-elektronic.net
Messungen IR-Signal
Signalanzahl ■

A wide variety has the number of Hi/Lo pairs transmitted.

The next piece of software (German version shown) counts the number of signals, sorts them by high and low polarity and counts those that are longer than $256 \times 8 = 2,048 \mu\text{s}$.

12.4.4.1 Program

The software has a similar structure and is listed in the following. The [source code is here](#), the [LCD routines to be included are here](#).

```
;;
; ****
; * IR receiver with ATTiny24 and LCD to measure signal numbers *
; * (C)2017 by http://www.avr-asm-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Constant -----
.equ cSet      = 0x10 ; MSB long pause between signal bursts
;
; ----- Ports -----
.equ pIrIn = PINA ; IR-Detektor-Port
.equ bIrIn = 0     ; IR-Detektor-Pin
;
; ----- Timing -----
; Processor clock      1.000.000 cs/s
; Time per clock cycle    1 us
; Prescaler T1          8
; Time per timer tick    8 us
; T1 count after 1 ms    125
; T1 overflow after      524 ms
;
; ----- Registers -----
; Used: R0 by LCD, decimal conversion
; Used: R1 Decimal conversion
.def rHigh = R2 ; Number of High signals
.def rLow = R3 ; Number of Low signals
.def rHead = R4 ; Number of header signals
.def rSet = R5 ; Number of signal sets
; free: R6..R14
.def rSreg = R15 ; Save/Restore SREG
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Second multi purpose register
.def rLine = R18 ; LCD line counter
```

```

.def rRead = R19 ; LCD read register
.def rImp = R20 ; Interrupt multi purpose
.def rFlag = R21 ; Flag register
    .equ bUpd = 0 ; Update display flag
; free: R22 .. R29
; Used: ZH:ZL R31:R30 in LCD routines
;
; ----- Reset- and interrupts -----
.CSEG ; Assemble to code segment
.ORG 0 ; to the beginning
    rjmp Start ; Reset vector, Init
    reti ; INT0 External Interrupt Request 0
    rjmp Pci0Isr ; PCINT0 Pin Change Interrupt Request 0
    reti ; PCINT1 Pin Change Interrupt Request 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT Timer/Counter1 Capture Event
    reti ; TIM1_COMPA Timer/Counter1 Compare Match A
    reti ; TIM1_COMPB Timer/Counter1 Compare Match B
    reti ; TIM1_OVF Timer/Counter1 Overflow
    reti ; TIM0_COMPA Timer/Counter0 Compare Match A
    reti ; TIM0_COMPB Timer/Counter0 Compare Match B
    rjmp Tc0Isr ; TIM0_OVF TCO Overflow, update display
    reti ; ANA_COMP Analog Comparator
    reti ; ADC_ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr:
    in rSreg,SREG ; Save SREG
    sbic pIrIn,bIrIn ; Skip next if IR input low
    rjmp Pci0Isrl ; Input is high
    ; Input is low, was high before
    inc rHigh ; Count high signals
    in rImp,TCNT1L ; Read TC1 count LSB
    in rImp,TCNT1H ; dto., MSB
    tst rImp ; MSB larger than 0?
    breq Pci0Isr2 ; no
    inc rHead ; Increase number of head signals
    cpi rImp,cSet ; Test if larger or equal cSet
    brccs Pci0Isr2 ; no
    inc rSet ; Increase number of data sets
    rjmp Pci0Isr2 ; To clear TC1
Pci0Isrl:
    inc rLow ; Count low signals
Pci0Isr2:
    ldi rImp,0 ; Restart counter TC1
    out TCNT1H,rImp ; Write MSB first
    out TCNT1L,rImp ; Then LSB
    out TCNT0,rImp ; Restart counter TCO
    ldi rImp,1<<TOIE0 ; Enable time out interrupts
    out TIMSK0,rImp ; in TCO int mask
    out SREG,rSreg ; Restore SREG
    reti ; Done
;
; TCO-Overflow Interrupt Service Routine
Tc0Isr:
    in rSreg,SREG ; Save SREG
    sbr rFlag,1<<bUpd ; Set update flag
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Start, Init -----
Start:
    ldi rmp,LOW(RAMEND) ; Stack to RAMEND
    out SPL,rmp ; to stack pointer
    ; Init I/O
    ; Init LCD port outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Output pins off
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; Data port mask write
    out pLcdDR,rmp ; to data port direction
    ; Init LCD
    rcall LcdInit ; Call included LCD routine

```

```

ldi ZH,HIGH(2*CodeChars) ; Special characters
ldi ZL,LOW(2*CodeChars)
rcall LcdChars ; Write special chars to LCD
ldi ZH,HIGH(2*LcdStart) ; Display start text
ldi ZL,LOW(2*LcdStart)
rcall LcdText ; Call to included routine
; Start value for flags
clr rFlag
; Clear all counters and content
rcall Restart ; Restart values
; Init timer TCO
ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler 1024
out TCCR0B,rmp ; Start timer
; Init timer TC1, free running by clock div 8
ldi rmp,1<<CS11 ; Prescaler = 8
out TCCR1B,rmp ; to timer control port B
; PCINT0 for IR-Rx
ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
out PCMSK0,rmo ; to mask port 0
ldi rmp,1<<PCIE0 ; Enable interrupt PCINT0
out GIMSK,rmp ; to General interrupt mask
; Sleep Enable
ldi rmp,1<<SE ; Sleep Mode Idle
out MCUCR,rmp ; in MCU control register
; Enable interrupts
sei ; Set I flag in SREG

Loop:
sleep ; go to sleep
nop ; Wake up
sbrc rFlag,bUpd ; Check update flag
rcall Update ; Process flag
rjmp Loop ; go to sleep again

; Process update flag
Update:
cbr rFlag,1<<bUpd ; Clear update flag
clr rmp ; Clear TCO
out TCNT0,rmp ; TCO = 0
out TIMSK0,rmp ; Disable TCO interrupts
; Display clear
ldi rmp,0x01 ; Delete display content
rcall LcdC4Byte ; to display
; Display highs
ldi ZH,High(2*LcdForm) ; Form template
ldi ZL,Low(2*LcdForm)
rcall LcdText ; Write on LCD
ldi rmp,0x01 ; Cursor home
rcall LcdD4Byte
mov rmp,rHigh ; Display highs on LCD
rcall Hex2Lcd
ldi rmp,' ' ; Blank on LCD
rcall LcdD4Byte
ldi rmp,0x00 ; Special character 0
rcall LcdD4Byte
mov rmp,rLow ; Display lows on LCD
rcall Hex2Lcd
ldi rmp,' ' ; Blank
rcall LcdD4Byte
ldi rmp,'h' ; Display header symbol
rcall LcdD4Byte
mov rmp,rHead ; Display header signals
rcall Hex2Lcd
ldi rmp,' ' ; Blank
rcall LcdD4Byte
ldi rmp,'S' ; Display data sets symbol
rcall LcdD4Byte
mov rmp,rSet ; Display data sets
rcall Hex2Lcd
ldi rmp,0x0C ; Cursor and blink off
rcall LcdC4Byte ; to LCD

; Restart
Restart:
clr rHead ; Clear number of headers
clr rHigh ; Clear number of highs
clr rLow ; Clear number of lows
ret

```

```

;
; Byte in rmp in hex to LCD
Hex2Lcd:
    push rmp ; Save byte
    swap rmp ; Upper nibble first
    rcall HexNibble2Lcd
    pop rmp ; Restore rmp
; Display nibble in hex on LCD
HexNibble2Lcd:
    andi rmp,0x0F ; Mask lower nibble
    subi rmp,-'0' ; Add ASCII-0
    cpi rmp,'9'+1 ; A to F?
    brccs HexNibble2Lcd1 ; no
    subi rmp,-7 ; Adjust to A to F
HexNibble2Lcd1:
    rjmp LcdD4Byte ; Display rmp on LCD
;
; Start text LCD
LcdStart:
.db "IR analysis ATtiny24",0x0D,0xFF
.db "avr-asm-tutorial.net",0x0D,0xFF
.db " Measuring IR signal",0x0D,0xFF
.db "    Signal numbers ",0xFE
;
; Template
LcdForm:
.db "Number H/L Sig-",0x0D,0xFF
.db "nals, headers, ",0x0D,0xFF
.db "long signals ",0x0D,0xFE
;
; Special characters
CodeChars:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High signal
.db 0,0 ; End of table
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

12.4.4.2 Example results

Anzahl High/Low-Signale, Kopfbytes, Langwartesignale
„43 „43 k03 S47

Developers of remote control equipment do obviously not apply rational rules.

Anzahl High/Low-Signale, Kopfbytes, Langwartesignale
„92 „93 k02 S3A

Anzahl High/Low-Signale, Kopfbytes, Langwartesignale
„23 „24 k03 S3F

With this piece of software we learn that the remote control of the TV consists of 71 Hi/Lo data pairs, that three header signals are longer than 2.048 ms and that 67 data pairs are part of the burst. This is obviously more than 64 bits, and allows encoding of enormous 1.84E+18 combinations, a lot more than buttons on the box. Developers of remote control equipment do obviously not apply rational rules.

This here is a real miracle: 146 Hi- and 147 Lo values provide 8.92E+42 combinations, each atom in our galaxy can be encoded individually, leaving rationality far behind.

The nine buttons of the remote control box for a camera can well be encoded in four bits, but this has 35 Hi and 36 Lo signals. From which 429 million combinations can be encoded. Obviously remote control designers have lost all technical rationality.

12.4.5 Signal durations of data bits

In principle we already see from the diagnostic software so far which duration Hi and Lo signals have. With the following we expand the display of data bits. If we skip the MSB, which is always zero in data bits, we see 24 bytes on the LCD. If we only display either Hi or Lo signals and if we enable the selection of the direction (from the beginning, from the end), we see 48 data bits.

IR-Analyse ATTiny24
gsc-elektronic.net
Kurzsignale von vorn
Hex Hi./Lo. 8 us_

That is the start text on display (the German version).

12.4.5.1 Program

The program is listed as follows, [the source code is here](#) and has to be combined with [the LCD routines](#).

```
;  
; *****  
; * IR receiver with ATTiny24 and LCD to examine bits *  
; * (C)2017 by http://www.avr-asm-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Switches -----  
.equ FromBehind = 0 ; 0: first bits  
; ; 1: last bits  
.equ cHigh = 1 ; 1: Examine high bytes  
.equ cLow = 1 ; 1: Examine low bytes  
;  
; ----- Byte output -----  
; .bb.bb.bb.bb.bb ; 6 bytes  
; .bb.bb.bb.bb.bb ; 12 bytes  
; .bb.bb.bb.bb.bb ; 18 bytes  
; .bb.bb.bb.bb.bb ; 24 bytes  
;  
; ----- Ports -----  
.equ pIrIn = PINA ; IR receiver module port  
.equ bIrIn = 0 ; IR receiver module pin  
;  
; ----- Timing -----  
; Controller clock 1.000.000 cs/s  
; Time per clock cycle 1 us  
; Prescaler TC1 8  
; Time per timer tick 8 us  
; Long signal duration 30.000 us  
; Long signal timer count, MSB 14  
.equ cLong = 14 ; MSB of a long signal  
;  
; ----- Registers -----  
; Used: R0 by LCD routine, decimal conversion  
; Used: R1 Decimal conversion  
; free: R2..R14  
.def rSreg = R15 ; Save/Restore SREG  
.def rmp = R16 ; Multi purpose register  
.def rmo = R17 ; Additional multi purpose  
.def rLine = R18 ; LCD line counter  
.def rRead = R19 ; LCD read register  
.def rimp = R20 ; Interrupt multi purpose  
.def rFlag = R21 ; Flag register  
    .equ bUpd = 0 ; Update flag  
    .equ bShf = 1 ; Shift backwards flag  
    .equ bLng = 2 ; Long high signal flag  
; free: R22 .. R25  
; Used: XH:XL R27:R26 for Shifting  
; Used: YH:YL R29:R28 Pointer to SRAM  
; Used: ZH:ZL R31:R30 in LCD routines  
;
```

```

; ----- SRAM -----
.DSEG ; Assemble to data segment
.ORG 0x0060 ; Start address of SRAM
Buffer:
.byte 24 ; Buffer for receiver data
BufferEnd:
Last:
.byte 1 ; Last measured value
LastEnd:
;
; ----- Reset and Interrupts -----
.CSEG ; Assemble to code segment
.ORG 0 ; At the beginning of flash memory
    rjmp Start ; Reset vector, Init
    reti ; INT0 External Interrupt Request 0
    rjmp Pci0Isr ; PCINT0 Pin Change Interrupt Request 0
    reti ; PCINT1 Pin Change Interrupt Request 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT Timer/Counter1 Capture Event
    reti ; TIM1_COMPA Timer/Counter1 Compare Match A
    reti ; TIM1_COMPB Timer/Counter1 Compare Match B
    reti ; TIM1_OVF Timer/Counter1 Overflow
    reti ; TIM0_COMPA Timer/Counter0 Compare Match A
    reti ; TIM0_COMPB Timer/Counter0 Compare Match B
    rjmp Tc0Isr ; TIM0_OVF TCO Overflow, Update display
    reti ; ANA_COMP Analog Comparator
    reti ; ADC ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr: ; PCINT0 ISR
    in rSreg,SREG ; Save SREG
    sbic pIrIn,bIrIn ; Skip next if IR input low
    rjmp Pci0Isr1 ; IR input is high
    ; IR Input is low, was high before
    in rimp,TCNT1L ; Read TC1 low byte first
    st Y,rimp ; Store LSB in SRAM
    in rimp,TCNT1H ; Read high byte second
    cpi rimp,cLong ; Long signal?
    brcc Pci0IsrNoLong ; Not a long signal
    sbr rFlag,1<<bLng ; Set long flag
.if FromBehind == 0 ; Backwards
    ldi YH,HIGH(Buffer) ; Y to buffer start
    ldi YL,LOW(Buffer)
    .endif
    rjmp Pci0Isr2 ; Done, clear TC1
Pci0IsrNoLong:
.if cHigh == 1 ; Only if high signal to be examined
    tst rimp ; test MSB
    brne Pci0IsrMsb ; MSB not zero
.if FromBehind == 1 ; Only if last signals to be examined
    ; High signal
    sbr rFlag,1<<bShf ; Set shift flag
    .else
    ; Only if first signals to be examined
    cpi YL,LOW(BufferEnd) ; End of buffer reached?
    brcc Pci0IsrNoInc ; Behind buffer end
    adiw YL,1 ; Before buffer end, inc pointer
Pci0IsrNoInc:
    .endif
Pci0IsrMsb: ; MSB not zero
    .endif
    rjmp Pci0Isr2 ; Done
Pci0Isr1:
    ; IR input is high, was low before
.if cLow == 1 ; Only if low signals to be examined
    in rimp,TCNT1L ; Read TC1 low byte first
    st Y,rimp ; LSB to SRAM
    in rimp,TCNT1H ; Read TC1 high byte second
    tst rimp ; If MSB not zero, ignore
    brne Pci0IsrMsb1 ; ignore
.if FromBehind == 1 ; Only if last signals to be examined
    sbr rFlag,1<<bShf ; Set shift flag
    .else
    clr rimp ; Restart TCO

```

```

        out TCNT0, rimp
        cpi YL, LOW(BufferEnd) ; Beyond buffer?
        brcc Pci0IsrNoIncl ; Yes, do not increase
        adiw YL, 1 ; Increase pointer
Pci0IsrNoIncl:
        .endif
Pci0IsrMsbl:
        .endif
Pci0Isr2:
        ldi rimp, 0 ; Restart TC1
        out TCNT1H, rimp ; First write MSB
        out TCNT1L, rimp ; Then LSB
        out SREG, rSreg ; Restore SREG
        reti ; Done
;
; TC0-Overflow Interrupt Service Routine
Tc0Isr:
        in rSreg, SREG ; Save SREG
        sbr rFlag, 1<<bUpd ; Set update flag
        out SREG, rSreg ; Restore SREG
        reti
;
; ----- Start, Init -----
Start:
        ldi rmp, LOW(RAMEND) ; Stack to RAMEND
        out SPL, rmp ; to stack pointer
; Init I/O
; Init LCD port outputs
        ldi rmp, (1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
        out pLcdCR, rmp ; to LCD control port
        clr rmp ; Outputs off
        out pLcdCO, rmp ; to control port outputs
        ldi rmp, mLcdDRW ; LCD data port output mask write
        out pLcdDR, rmp ; to direction port
; Init LCD
        rcall LcdInit ; Call included LCD routine
        ldi ZH, HIGH(2*CodeChars) ; Special characters
        ldi ZL, LOW(2*CodeChars)
        rcall LcdChars ; Write to LCD
        ldi ZH, HIGH(2*LcdStart) ; Output start text
        ldi ZL, LOW(2*LcdStart)
        rcall LcdText ; Call included LCD routine
; Start value for Buffer pointer
.if FromBehind == 1 ; Examine last signals
        ldi YH, HIGH>Last) ; To behind buffer end
        ldi YL, LOW>Last)
.else
        ldi YH, HIGH(Buffer) ; Y to buffer start
        ldi YL, LOW(Buffer)
.endif
; Start value for flags
        clr rFlag
; Init TC0 for time out
        ldi rmp, (1<<CS02) | (1<<CS00) ; Prescaler 1024
        out TCCR0B, rmp ; Start timer
; Init TC1 initieren, free running
        ldi rmp, 1<<CS11 ; Prescaler = 8
        out TCCR1B, rmp ; to timer control port B
; PCINT0 for IR Rx
        ldi rmp, 1<<PCINT0 ; Pin 0 Level change Ints
        out PCMSK0, rmo ; to mask 0
        ldi rmp, 1<<PCIE0 ; Enable interrupt PCINT0
        out GIMSK, rmp ; in General interrupt mask
; Sleep Enable
        ldi rmp, 1<<SE ; Sleep mode idle
        out MCUCR, rmp ; to MCU control port
; Enable interrupts
        sei ; Set I flag in SREG
Loop:
        sleep ; go to sleep
        nop ; Wake up
        sbrc rFlag, bShf ; Skip next if shift flag inactive
        rcall Shift ; Process shift flag
        sbrc rFlag, bLng ; Skip next if long flag inactive
        rcall Long ; Process long flag
        sbrc rFlag, bUpd ; Skip next if update flag inactive
        rcall Update ; Process update flag

```

```

        rjmp Loop ; go to sleep again
;
; Long wait signal detected
Long:
    cbr rFlag,1<<bLng ; Clear long flag
    clr rmp ; Clear TCO
    out TCNT0,rmp
    ldi rmp,1<<TOIE0 ; Enable overflow int
    out TIMSK0,rmp ; in TCO interrupt mask
    ret
;
; Shift last value to buffer
Shift:
    cbr rFlag,1<<bShf ; Clear shift flag
    clr rmp ; Clear TCO
    out TCNT0,rmp
    ldi rmp,1<<TOIE0 ; Enable overflow interrupt
    out TIMSK0,rmp ; in TCO interrupt mask
    ldi XH,HIGH(Buffer) ; X buffer pointer to start
    ldi XL,LOW(Buffer)
    ldi ZH,HIGH(Buffer+1) ; Z buffer pointer one byte higher
    ldi ZL,LOW(Buffer+1)
Shift1: ; Shift bytes backward
    ld rmp,Z+ ; Read byte one position forward
    st X+,rmp ; Write byte one position backward
    cpi ZL,LOW>LastEnd) ; End of buffer reached?
    brcc Shift1 ; no, go on
    ret
;
; Process update flag
Update:
    cbr rFlag,1<<bUpd ; Clear update flag
    clr rmp ; Disable TCO overflow interrupt
    out TIMSK0,rmp ; in TCO interrupt mask
    ldi rmp,1 ; Clear LCD
    rcall LcdC4Byte ; to LCD control port
    ldi ZH,HIGH(Buffer) ; Z to buffer start
    ldi ZL,LOW(Buffer)
Update1:
    cpi ZL,LOW(BufferEnd) ; Complete buffer displayed?
    brcc Update5 ; Yes, done
.if FromBehind == 1 ; Only if last bytes to be displayed
    ldi rmp,'e' ; Mark last value pair
    cpi ZL,LOW(BufferEnd-1) ; Last value pair?
    brcc Update2 ; Yes, 'e' marks last
    .endif
.if cHigh == 1
    ; Examine cHigh is on
    .if cLow == 1 ; Both signals
        ; cHigh=1, cLow=1
        ldi rmp,' ' ; Blank
    .else
        ; cHigh=1, cLow=0
        ldi rmp,0x01 ; Set high mark
    .endif
    .else
        ; cHigh=0
        ldi rmp,0x00 ; Set low mark
    .endif
Update2:
    rcall LcdD4Byte ; Display special character
    ld rmp,Z+ ; Read byte from buffer
    rcall Hex2Lcd ; Write in hex
    cpi ZL,LOW(Buffer+6) ; Line 1 full?
    brne UpDate3 ; Z<>6
    rcall LcdLine2 ; Set line 2
    rjmp Update1 ; Continue
Update3:
    cpi ZL,LOW(Buffer+12) ; Line 2 full?
    brne UpDate4
    rcall LcdLine3 ; Set line 3
    rjmp Update1 ; Continue
Update4:
    cpi ZL,LOW(Buffer+18) ; Line 3 full?
    brne UpDate1 ; no, continue
    rcall LcdLine4 ; Set line 4
    rjmp UpDate1

```

```

Update5:
    ldi rmp,0x0C ; Cursor and blink off
    rcall LcdC4Byte ; to LCD
    clr rmp ; Clear buffer
.if FromBehind == 1
    ; Clear buffer backwards
    ldi ZH,HIGH(LastEnd) ; to buffer end
    ldi ZL,LOW(LastEnd)
Lang1:
    st -Z,rmp ; Clear buffer
    cpi ZL,LOW(Buffer+1) ; Already complete?
    brcc Lang1 ; no
.else
    ; Clear buffer forward
    ldi ZH,HIGH(Buffer) ; Z to buffer start
    ldi ZL,LOW(Buffer)
Lang1:
    st Z+,rmp ; Clear next byte
    cpi ZL,LOW(LastEnd) ; Already at buffer end?
    brccs Lang1 ; no, go on
.endif
    ret
;
; Byte in rmp in hex to LCD
Hex2Lcd:
    push rmp ; Save byte
    swap rmp ; Upper nibble first
    rcall HexNibble2Lcd
    pop rmp ; Restore rmp
; Display nibble in hex on LCD
HexNibble2Lcd:
    andi rmp,0x0F ; Mask lower nibble
    subi rmp,-'0' ; Add ASCII-0
    cpi rmp,'9'+1 ; A to F?
    brccs HexNibble2Lcd1 ; no
    subi rmp,-7 ; Adjust A to F by adding 7
HexNibble2Lcd1:
    rjmp LcdD4Byte ; Display char in rmp on LCD
;
; Start text on LCD
LcdStart:
.db "IR analysis ATTiny24",0x0D,0xFF
.db "avr-asm-tutorial.net",0x0D,0xFF
.if FromBehind == 1
    .db "Short signals at end",0x0D,0xFF
.else
    .db "Short signals start ",0x0D,0xFF
.endif
.if cHigh == 1
    .if cLow ==1
        .db "Hex Hi",0x01,"/Lo",0x00," 8 us",0xFE,0xFE
    .else
        .db "Hex High",0x01,", 8 us",0xFE
    .endif
    .else
        .db "Hex Low",0x00,", 8 us",0xFE,0xFE
    .endif
;
; Special characters
CodeChars:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High signal
.db 80,0,0,0,0,0,0,0,7,0 ; Z = 2, Pause separator
.db 0,0 ; End of table
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

A new instruction used here is ST -Z register, which decreases Z before writing the register content to the SRAM cell that Z points to.

12.4.5.2 Examples

E0	31	9F	34	36	2E
30	33	31	34	37	37
34	3A	A3	3A	37	31
34	34	37	2E	9D	37

The code generates the following output on LCD. With both switches set the high as well as the low durations are measured and displayed. A blank between the signals marks that condition.

48	40	47	40	48	40
47	40	48	CA	48	3F
47	CA	48	C9	48	C9
48	C9	48	C9	49	e1C

The same display with the last bytes received, the e signals the last byte.

wD6	w3C	wA4	w35	w3B	w38
w3F	w38	w39	w39	w33	w35
w32	w35	wA4	w36	w39	w3C
w3C	w3B	w3C	w35	wA1	w39

That is displayed if only the high signals are to be examined.

w3C	w38	wA4	wA4	wA1	w3F
w3F	w3C	wA8	w3C	w3B	w39
wAA	w38	w35	w3C	wA7	w38
wA4	wAB	w3E	wA7	w39	eAB

And this is the same for the last signals. With this, any byte combinations can be examined.

12.4.6 Encoding of IR commands

With the previous methods the encoded keys of a remote control box can only be analyzed with extraordinary high efforts, and only for up to 48 bits. Therefore here is the more convenient solution for even more bits.

12.4.6.1 Program

The program is here, [the source code file here](#). Of course [the LCD routines are needed to assemble this](#).

```
; ****
; * IR receiver with ATTiny24 and LCD for burst analysis *
; * (C)2017 by http://www.avr-asm-tutorial.net          *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Switches -----
.equ cActiveHi = 0 ; 1: High signals encode bits
;                 0: Low signals encode bits
;
; ----- Constants -----
.equ cIRStart = 0x10 ; Pause before signal (MSB)
.equ cIRHigh = 0x50 ; Zero/one discrimination level
;
; ----- Ports -----
.equ pIrIn = PINA ; IR receiver port
.equ bIrIn = 0      ; IR receiver pin
```

```

;
; ----- Timing -----
; Controller clock      1.000.000 cs/s
; Time per clock cycle   1 us
; Prescaler TC1          8
; Time per TC1 tick       8 us
;
; ----- Registers -----
; Used: R0 by LCD, Decimal conversion
; Used: R1 Decimal conversion
.def rShift0 = R2 ; Shift register, byte 0
.def rShift1 = R3 ; dto., byte 1
.def rShift2 = R4 ; dto., byte 2
.def rShift3 = R5 ; dto., byte 3
.def rShift4 = R6 ; dto., byte 4
.def rShift5 = R7 ; dto., byte 5
.def rShift6 = R8 ; dto., byte 6
.def rShift7 = R9 ; dto., byte 7
; free: R10..R14
.def rSreg = R15 ; Save/Restore SREG
.def rmp   = R16 ; Multi purpose register
.def rmo   = R17 ; Another multi purpose register
.def rLine = R18 ; LCD line counter
.def rRead = R19 ; LCD read register
.def rimp  = R20 ; Interrupt multi purpose
.def rFlag = R21 ; Flag register
    .equ bInp = 0 ; Input value available
    .equ bPol = 1 ; Polarity (high/low)
    .equ bUpd = 2 ; Update after pause
    .equ bHOf = 3 ; Header data overflow
    .equ bDOF = 4 ; Data bits overflow
.def rHead = R22 ; Header signal counter
.def rData = R23 ; Bit counter register
; free: R24 .. R25
; Used: XH:XL R27:R26 for shifting
; Used: YH:YL R29:R28 Timer count
; Used: ZH:ZL R31:R30 in LCD routines
;
; ----- SRAM -----
.DSEG ; Data segment
.ORG 0x0060 ; Start of SRAM memory
Head:
.byte 6 ; Buffer for header data, MSB, LSB
HeadEnd:
StatisticLow:
.byte 3 ; Sum duration low signal, MSB/LSB/Number
StatisticHigh:
.byte 3 ; Sum duration high signal, MSB/LSB/Number
StatisticPause:
.byte 3 ; Sum duration pause, MSB/LSB/Number
StatisticEnd:
;
; ----- Reset and Interrupts -----
.CSEG ; Assemble to code segment
.ORG 0 ; to the start
    rjmp Start ; Reset vector, Init
    reti ; INTO External Interrupt Request 0
    rjmp Pci0Isr ; PCINT0 Pin Change Int Req 0
    reti ; PCINT1 Pin Change Interrupt Req 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT Timer/Counter1 Capture Event
    reti ; TIM1_COMPA Timer/Counter1 Compare Match A
    reti ; TIM1_COMPB Timer/Counter1 Compare Match B
    reti ; TIM1_OVF Timer/Counter1 Overflow
    reti ; TIM0_COMPA Timer/Counter0 Compare Match A
    reti ; TIM0_COMPB Timer/Counter0 Compare Match B
    rjmp Tc0Isr ; TIM0_OVF TCO Overflow, update display
    reti ; ANA_COMP Analog Comparator
    reti ; ADC ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr: ; PCINT0
    in rSreg,SREG ; Save SREG
    cbr rFlag,1<<bPol ; Clear polarity flag

```

```

sbic pIrIn,bIrIn ; Skip next if input low
; IR input is low, was high before
sbr rFlag,1<<bPol ; Set polarity flag
in YL,TCNT1L ; Read TC1, low byte first
in YH,TCNT1H ; Then high byte
Pci0Isr2:
ldi rimp,0 ; Restart TC1
out TCNT1H,rimp ; First MSB
out TCNT1L,rimp ; Then LSB
sbr rFlag,1<<bInp ; Flag received signal
out SREG,rSreg ; Restore SREG
reti ; Done
;
; TC0-Overflow Interrupt Service Routine
Tc0Isr:
in rSreg,SREG ; Save SREG
sbr rFlag,1<<bUpd ; Set update flag
out SREG,rSreg ; Restore SREG
reti
;
; ----- Start, Init -----
Start:
ldi rmp,LOW(RAMEND) ; Stack to RAMEND
out SPL,rmp ; to stack pointer
; Init I/O
; Init LCD port outputs
ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
out pLcdCR,rmp ; to LCD control port outputs
clr rmp ; Outputs off
out pLcdCO,rmp ; to LCD control port
ldi rmp,mLcdDRW ; LCD data port mask write
out pLcdDR,rmp ; to LCD data direction port
; Init LCD
rcall LcdInit ; Call to included LCD routines
ldi ZH,HIGH(2*CodeChars) ; Special chars
ldi ZL,LOW(2*CodeChars)
rcall LcdChars ; to LCD
ldi ZH,HIGH(2*LcdStart) ; Display start text
ldi ZL,LOW(2*LcdStart)
rcall LcdText
; Start value for flags
clr rFlag
; Delete all content
rcall Restart ; Restart values
; Init timer TCO
ldi rmp,(1<<CS02) | (1<<CS00) ; Prescaler 1024
out TCCR0B,rmp ; Start timer
; Init timer TC1, free running
ldi rmp,1<<CS11 ; Prescaler to 8
out TCCR1B,rmp ; to timer control port B
; PCINT0 for IR Rx
ldi rmp,1<<PCINT0 ; Pin 0 Level change Ints
out PCMSK0,rmo ; to mask port 0
ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
out GIMSK,rmp
; Sleep Enable
ldi rmp,1<<SE ; Sleep Mode Idle
out MCUCR,rmp ; in MCU port
; Enable interrupts
sei ; Set I flag in SREG
Loop:
sleep ; go to sleep
nop ; Wake up
sbrc rFlag,bInp ; Check input flag
rcall Input ; Process input flag
sbrc rFlag,bUpd ; Check update flag
rcall Update ; Process update flag
rjmp Loop ; sleep again
;
; Process input flag
Input:
cbr rFlag,bInp ; Clear input flag
mov XH,YH ; Copy counter
mov XL,YL
; Clear timer TCO
ldi rmp,0 ; Clear timer TCO
out TCNT0,rmp

```

```

ldi rmp,1<<TOIE0 ; Enable timer overflow ints
out TIMSK0,rmp
tst XH ; MSB not zero?
breq Input3 ; MSB = zero
; MSB unequal zero
cpi XH,cIRStart ; MSB larger than start condition?
brccs Input1 ; no
rcall Restart ; Clear all
Input1:
inc rHead ; Next header count
cpi rHead,4 ; More than three headers?
brccs Input2 ; No
sbr rFlag,1<<bHOf ; Set header overflow flag
ret
Input2:
ldi ZH,HIGH(Head) ; Point Z to header info
ldi ZL,LOW(Head)
lsl rHead ; Multiply by two
add ZL,rHead ; Add to pointer
ldi rmp,0 ; MSB overflow?
adc ZH,rmp ; Add carry to MSB
lsr rHead ; Multiply by four
st Z+,XH ; To header storage
st Z,XL
ret
Input3:
.if cActiveHi == 1 ; If input is active high
    sbrc rFlag,bPol ; Skip if polarity is normal
.else
    sbrs rFlag,bPol ; Skip if polarity is reverse
.endif
rjmp Input5 ; Low signal/high signal
; Input was active, collect data bits
ldi rmp,XL ; Load low-high-level
cp rmp,XL ; Smaller: C clear, larger: C set
rol rShift0 ; Roll carry into shift register
rol rShift1
rol rShift2
rol rShift3
rol rShift4
rol rShift5
rol rShift6
rol rShift7
inc rData ; Count data bits
cpi rData,66 ; 66 bits are fine
brccs Input4 ; Fine
sbr rFlag,1<<bDOf ; Set data bits overflow flag
Input4:
; Add to sum
ldi ZH,HIGH(StatisticLow) ; Point Z to low
ldi ZL,LOW(StatisticLow)
cp rmp,XL ; Again result to carry flag
brcc Input6 ; Carry = zero
ldi ZH,HIGH(StatisticHigh) ; Point Z to high
ldi ZL,LOW(StatisticHigh)
rjmp Input6 ; Continue
Input5:
; Input is reverse polarity
ldi ZH,HIGH(Statisticpause) ; Point Z to pause
ldi ZL,LOW(Statisticpause)
Input6:
ldd rmp,Z+1 ; Load LSB
add rmp,XL ; Add LSB
std Z+,rmp ; and store
ld rmp,Z ; Load MSB
adc rmp,XH ; Add MSB
st Z,rmp ; and store
adiw ZL,2 ; Point to number counter
ld rmp,Z ; Load number counter
inc rmp ; Increase
st Z,rmp ; and store
ret
;
; Process update flag
Update:
cbr rFlag,1<<bUpd ; Clear update flag
clr rmp ; Disablöe TCO ints

```

```

out TIMSK0,rmp ; in TCO interrupt mask
ldi rmp,0x01 ; Clear display
rcall LcdC4Byte
ldi ZH,HIGH(2*LcdMask) ; Show mask on display
ldi ZL,LOW(2*LcdMask)
rcall LcdText

Update1:
; Display head data
ldi ZH,0 ; LCD position head
ldi ZL,5
rcall LcdPos ; Set LCD position
ldi ZH,HIGH(Head) ; Point to head data
ldi ZL,LOW(Head)
clr R0 ; Polarity character low

Update2:
ldi rmp,0x01 ; Revert polarity character
eor R0,rmp ; Exor bit 0
mov rmp,R0 ; Copy to rmp
rcall LcdD4Byte ; Display on LCD
ld rmp,Z+ ; Read MSB
rcall Hex2Lcd ; Display hex on LCD
ld rmp,Z+ ; Read LSB
rcall Hex2Lcd ; Display hex on LCD
cpi ZL,LOW(HeadEnd) ; All header info?
brcs Update2 ; No, go on
sbrs rFlag,bHOf ; Skip next if header overflow
rjmp Update3 ; Display header
ldi ZH,0 ; Display overflow message on line 1
ldi ZL,17
rcall LcdPos
ldi rmp,' ' ; Blank
rcall LcdD4Byte
ldi rmp,'0' ; Large 0
rcall LcdD4Byte
ldi rmp,'!' ; Error
rcall LcdD4Byte

Update3: ; Output bit sums and numbers
ldi ZH,1 ; Line 2
ldi ZL,2
rcall LcdPos ; Set Lcd position
ldi ZH,HIGH(StatisticLow) ; Point Z to statistic
ldi ZL,LOW(StatisticLow)
ld rmp,Z+ ; Read MSB sum
rcall Hex2Lcd ; Display hex on Lcd
ld rmp,Z+ ; Read LSB sum
rcall Hex2Lcd ; Display hex on Lcd
ldi rmp,' ' ; Write blank
rcall LcdD4Byte ; to Lcd
ld rmp,Z+ ; Read number of signals
rcall Hex2Lcd ; Write hex on Lcd
ldi rmp,' ' ; Add blank
rcall LcdD4Byte
ldi rmp,'1' ; Write 1: on Lcd
rcall LcdD4Byte
ldi rmp,:'
rcall LcdD4Byte
ld rmp,Z+ ; Read MSB 1
rcall Hex2Lcd ; Write hex on Lcd
ld rmp,Z+ ; Read LSB 1
rcall Hex2Lcd ; Write hex on Lcd
ldi rmp,' ' ; Add blank
rcall LcdD4Byte
ld rmp,Z+ ; Read number 1
rcall Hex2Lcd ; Write hex on Lcd
ldi ZH,2 ; Lcd to line 3
ldi ZL,2
rcall LcdPos
ldi ZH,HIGH(Statisticpause) ; Point to pause
ldi ZL,LOW(Statisticpause)
ld rmp,Z+ ; Read MSB pause
rcall Hex2Lcd ; Write hex on Lcd
ld rmp,Z+ ; Read LSB pause
rcall Hex2Lcd ; Write hex on Lcd
ldi rmp,' ' ; Add blank
rcall LcdD4Byte
ld rmp,Z+ ; Read number of pauses
rcall Hex2Lcd ; Write hex on Lcd

```

```

; Shift registers on line 4
ldi ZH,3 ; LCD to line 4
ldi ZL,2
rcall LcdPos
ldi ZH,0 ; Point Z to register R10
ldi ZL,10
Update4:
ld rmp,-Z ; Decrease Z and read byte
rcall Hex2Lcd ; Display on Lcd in hex
cpi ZL,3 ; All displayed?
brcc Update4 ; No, go on
sbrs rFlag,bDOF ; Bit overflow?
rjmp Update5 ; No, continue below
ldi rmp,'0' ; Signal an overflow with 0!
rcall LcdD4Byte
ldi rmp,'!'
rcall LcdD4Byte
Update5:
ldi rmp,0x0C ; Cursor and blink off
rcall LcdC4Byte ; to LCD
;
; Restart
Restart:
ser rHead ; Number of heads to 0xFF
clr rData ; Clear data bit counter
cbr rFlag,(1<<bHOf) | (1<<bDOF) ; Clear flags
ldi ZH,0 ; Clear shift registers, point Z to R2
ldi ZL,2
Restart1:
st Z+,rData ; Clear register
cpi ZL,10 ; All cleared?
brcs Restart1 ; No, continue
ldi ZH,HIGH(Head) ; Clear SRAM content
ldi ZL,LOW(Head)
Restart2:
st Z+,rData ; Write zero
cpi ZL,LOW(StatisticEnd) ; All cleared?
brcs Restart2 ; No, go on
ret
;
; Byte in rmp in hex on LCD
Hex2Lcd:
push rmp ; Save byte
swap rmp ; Upper nibble first
rcall HexNibble2Lcd
pop rmp ; Restore byte
; Display nibble in hex on LCD
HexNibble2Lcd:
andi rmp,0x0F ; Mask lower nibble
subi rmp,-'0' ; Add ASCII-0
cpi rmp,'9'+1 ; A to F?
brcs HexNibble2Lcd1 ; no
subi rmp,-7 ; Adjust to A to F
HexNibble2Lcd1:
rjmp LcdD4Byte ; Display rmp on LCD
;
; Start text LCD
LcdStart:
.db "IR analysis ATtiny24",0x0D,0xFF
.db "avr-asm-tutorial.net",0x0D,0xFF
.db "Measuring IR signals",0x0D,0xFF
.db " in hex * 8 us ",0xFE
;
; Output mask for measured data
LcdMask:
.db "Head:",0x0D
;      5
.db "0:xxxx nn 1:xxxx nn ",0x0D,0xFF
;      2
.db "P:xxxx nn",0x0D
;      2
.db "D:xxxxxxxxxxxxxx ",0xFE
;      2
;
; Special chars
CodeChars:
.db 64,0,0,0,0,4,4,6,0 ; Z = 0, Low signal

```

```

.db 72,0,0,0,0,0,5,7,5,0 ; z = 1, High signal
.db 0,0 ; End of table
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
; End of source code
;
```

Two new instructions here are

- SER register. This sets all bits in a register and so the register content to 0xFF. This is only implemented for registers from R16 up.
- EOR register,register. This inverts all bits in the first register that are one in the second register.

Another thing that is new here is the application of pointer registers to access registers. If the pointer X, Y or Z is smaller than 32, it accesses the registers R0 to R31. The instructions LD and ST work with this. If the pointer points to locations 0x20 to 0x5F, the LD and ST instructions work on port numbers (add 0x20 to the original port number to get the pointer value). That is why SRAM starts at 0x0060, in larger devices with more ports even beyond. Use the constant RAMSTART to find out where SRAM in the device starts (it is listed in the def.inc file for the device) or consult the data-book.

12.4.6.2 Examples

```

Kopf: .W225E.01AC.0 U!
0:0768 24 1:0861 0D
P:0A55 32
D:0001400405380835
```

This is the result from my TV remote control box (German version). This generates more than three header signals ("U!" in the first line signals an overflow). But for analyzing header signals we already have a better tool. From the sums of low and high bits and combined with their number we can calculate averages.

Even more helpful is the data-bit examination behind D:. Here, the data bits encoding the different keys can be read in hex format.

TV device	
Hex	Key
4004.0538.0835	1
4004.0538.88B5	2
4004.0538.4875	3
4004.0538.C8F5	4
4004.0538.2815	5
4004.0538.A895	6
4004.0538.6855	7
4004.0538.E8D5	8
4004.0538.1825	9
4004.0538.98A5	0
0000.E0E0.E01F	Increase volume
0000.E0E0.D02F	Decrease volume
4004.0520.2C09	Increase program
4004.0520.AC89	Decrease programm
4004.0544.6322	OK

This is such a table with the keys encoded by my TV remote control box. It is obvious that the developer of that encoding follows its own rules.

I wish you more luck when exploring your own black boxes, the necessary tools are available here.

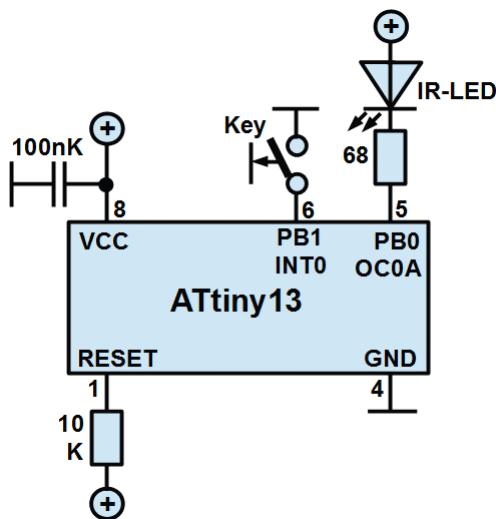
[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.5 An IR transmitter

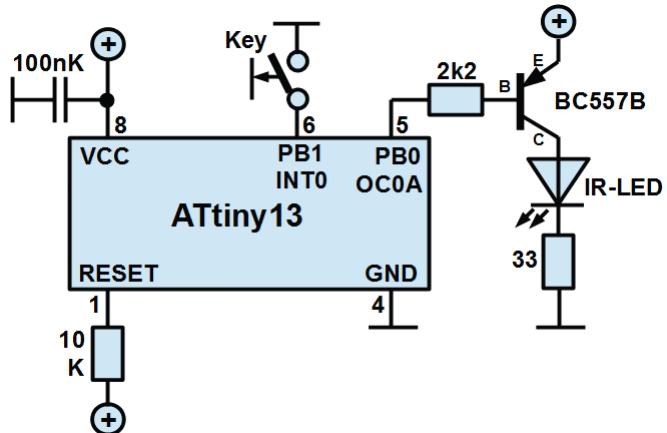
To not only being able to use existing remote control boxes, we built our own IR transmitter, design our own encoding protocol and our own norm. Because we have an ATtiny13 from the last experiments, we use this for our transmitter.

12.5.1 The hardware for IR transmitting

12.5.1.1 Scheme



That is all we need: an ATtiny13, a key to start the signal and an infrared LED. We avoid driving our IR LED with full power (at 100 mA) because in that case the current that can be driven by an output pin of an ATtiny13 would be exceeded. And the ISP interface would be overloaded with current and we would have to unmount the LED during ISP programming. This reduces our IR signal strength, but works fine.



Who wants to drive the IR LED with its full power and to use the ISP interface, use this driver with a PNP transistor. An NPN would reverse the polarity of the signal and a few changes to the program would be necessary (to clear the output pin to shut the LED off).

12.5.1.2 The IR LED



This is an infrared LED. At nominal 100 mA current it has a voltage drop of 1.35 V. The current limiting resistor at 5 V for a current of 73 mA that the ATtiny13 output pin can drive at maximum is

$$R [\Omega] = 1000 * (5 - 1,35 - 2 [V]) / 73 [\text{mA}] = 23 [\Omega]$$

If a transistor driver is used and 100 mA are desired the resistor is:

$$R [\Omega] = 1000 * (5 - 1,35 - 0,2 [V]) / 100 [\text{mA}] = 35 [\Omega]$$

12.5.1.3 68 Ohm resistor

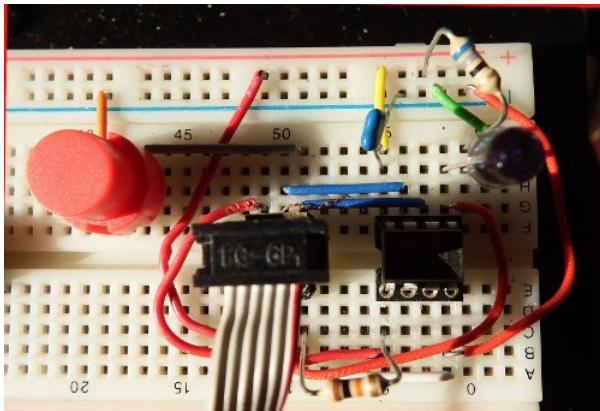


This is the 68Ω resistor. With this the driver current is

$$I [\text{mA}] = 1000 * (5 - 1,35 - 1,0 [V]) / 68 = 39 [\text{mA}]$$

which is compatible with the ISP programming hardware.

12.5.1.4 Mounting



This here is the simple mounting of the hardware on the breadboard.

12.5.2 Transmitter algorithms

12.5.2.1 IR transmit signal

For transmitting we use the timer TC0 in the ATtiny13 in CTC mode. When reaching its top level the OC0A output is either set (LED is off) or toggled (LED switched on and off).

The whole timing of the software centers around the generation of the 40 kcs/s signal of the LED. By counting the number of TOP events in the TIM0-COMPA interrupts of the timer the duration of the LED signal is controlled. As there are very long durations are to be generated (header signals) a 16 bit counter is necessary.

At a clock frequency of 1.2 Mcs/s and a LED frequency of 40 kcs/s the two on-and-off signal interrupts occur in $1,200,000 / 40,000 / 2 = 15$ clock cycles. The triggering of the interrupt (pushing the current address to the stack, jumping to the vector address) requires four clock cycles, the jump from the vector address to the interrupt service routine requires two clock cycles. This leaves $15 - 4 - 2 = 9$ clock cycles for the interrupt service routine. The further execution in the interrupt service routine would be:

```
TC0CAIsr:  
    in R15,SREG ; Save SREG  
    sbiw R24,1 ; Count down  
    brne TC0CAIsrRet ; Not yet zero  
    ; [... further code ...]  
TC0CAIsrRet:  
    out SREG,R15 ; Restore SREG  
    reti ; Done, return
```

The timing of that routine is as follows:

```
TC0CAIsr: ; 4 for Int, 2 for vector jump = 6  
    in R15,SREG ; +1 = 7  
    sbiw R24,1 ; +2 = 9  
    brne TC0CAIsrRet ; +2 = 11 for branching  
    ; [... further code ...]  
TC0CAIsrRet: ; 11 clock cycles  
    out SREG,R15 ; +1 = 12  
    reti ; +4 = 16
```

The interrupt service routine in the normal run (counter is not zero) is therefore 16 clock cycles long and the next COMPA interrupt request is initiated before the execution of the routine ends. For this, and for further operations, not enough time is available. At the default clock frequency this does not work.

A clock frequency of 2.4 Mcs/s, that can be selected by writing to the CLKPR port, is sufficient. Even if, within the service routine, further operations such as switching from toggle to set OC0A or setting flags have to be performed, the available 30 clock cycles offers enough time.

13.5.2.2 Control of the signal duration

The control of signal duration works with a 16 bit counter. It makes sense to use the register pair R25:R24 for that purpose: this pair can be down-counted with SBIW R24,1 but it cannot be used as a pointer.

Each down-count of this counter stands, at 40 kcs/s, for 12.5 µs. For a wait signal in the header $65536 * 12.5 = 819,200 \mu\text{s}$ are possible which provides enough range.

If we specify signal durations in µs and we want to know the counter value we have to multiply those by 2 and divide those by 25. We should do this conversion in the assembler source instead of doing that with the controller.

Our transmitter will be able to transmit up to eight header signals (four active, four inactive). To be able to define the duration for each of those signals we program those durations in short routines, place a table with the addresses of those routines to the flash, read those addresses using LPM and jump with the instruction IJMP to that routine. This indirect jump jumps to the address that the Z register pair points to. This enables us to calculate jump addresses and to jump very fast to the target address.

12.5.3 Program for transmitting

The following program can serve as a flexible base for own formulations and formats. It can be adjusted to fit to any need with very small changes. Each key stroke starts two transmitter sequences (bursts), separated by a long inactive phase in between.

During ISP transfer of the assembled code the IR LED shall be disconnected to avoid error messages due to the high current.

The program is listed below, the [source code is here](#).

```
;  
; *****  
; * IR transmitter 40 kcs/s with an ATtiny13 *  
; * (C)2017 by www.avr-asn-tutorial.net      *  
; *****  
;  
.NOLIST  
.INCLUDE "tn13def.inc"  
.LIST  
;  
; ----- Registers -----  
; free: R0 .. R5  
.def rData0 = R6 ; Transmit bits, highest, 1 to 8  
.def rData1 = R7 ; dto., next lower, 9 to 16  
.def rData2 = R8 ; dto., next lower, 17 to 23  
.def rData3 = R9 ; dto., next lower, 24 to 31  
.def rData4 = R10 ; dto., next lower, 32 to 39  
.def rData5 = R11 ; dto., next lower, 40 to 47  
.def rData6 = R12 ; dto., next lower, 48 to 55  
.def rData7 = R13 ; dto., next lower, 56 to 63  
.def rEor = R14 ; for polarity reversion Clear/Toggle  
.def rSreg = R15 ; Save SREG  
.def rmp = R16 ; Multi purpose register  
.def rimp = R17 ; Multi purpose inside int  
.def rFlag = R18 ; Flag register  
    .equ bRun = 0 ; Transmit sequence is active  
    .equ bSta = 1 ; Start transmit  
    .equ btTo = 2 ; Time out during transmit  
    .equ bRst = 3 ; Second burst  
.def rTOC = R19 ; Time out counter, counts signals  
; free R20 .. R23  
.def rCntL = R24 ; Counts CTC compares  
.def rCntH = R25  
; Used: XH:XL R27:R26  
; free: YH:YL R29:R28  
; Used: ZH:ZL R31:R30 Read from flash, IJMP  
;  
; ----- Constants and Timing -----  
.equ cClock = 2400000 ; Controller clock
```

```

.equ cClockNs = 1000000000 / cClock ; Clock in ns
.equ cIRF = 40000 ; IR transmit frequency cs/s
.equ cIRNs = 1000000000 / cIRF / 2 ; IR TX in ns
.equ cCtc = (cIRNs+cClockNs/2) / cClockNs - 1 ; CTC cycles
; @9.6 Mcs/s: 120; @4.8 Mcs/s: 60; @2.4 Mcs/s: 30
; @1.2 MHz: 15(!!), faster than ISR execution!!
;
; ----- Structure IR signal -----
; All times in us
; Number of header pairs, 1 .. 4
.equ nHead = 4 ; Number of header pairs H/L, 1..4
.equ cHead = 2*nHead-1 ; Number of header signals
; Duration of header signals (duration in 2*us/25)
.equ cTH1 = 2*50000/25 ; Duration High 1
.equ cTH2 = 2*5000/25 ; Duration Low 1
.equ cTH3 = 2*2500/25 ; Duration High 2
.equ cTH4 = 2*500/25 ; Duration Low 2
.equ cTH5 = 2*1250/25 ; Duration High 3
.equ cTH6 = 2*500/25 ; Duration Low 3
.equ cTH7 = 2*1250/25 ; Duration High 3
.equ cTH8 = 2*500/25 ; Duration Low 3
.equ cTH9 = 2*1250/25 ; Duration High 4
.equ cTH10 = 2*500/25 ; Duration Low 4
; Number of data bits to be transmitted
.equ cBits = 64 ; Number of bits, 1 .. 64
.equ cSign = cHead + 2*cBits ; Total number of signals
; Duration of data bits and active period
.equ cShort = 2*250/25 ; Duration binary zero
.equ cLong = 2*750/25 ; Duration binary one
.equ cPaus = 2*250/25 ; Duration active period
;
; ----- Transmit data -----
; Test data to transmit
.equ cWord1 = 0xAAAA ; Highest
.equ cWord2 = 0x5555
.equ cWord3 = 0xAAAA
.equ cWord4 = 0x5555 ; Lowest
;
; ----- Reset and interrupt vectors ---
.CSEG
.ORG 0
    rjmp Start ; Reset vector, program init
    rjmp Int0Isr ; INT0 Key interrupt
    reti ; PCINT0 Pin Change Int Request 0
    reti ; TIM0_OVF Timer/Counter Overflow
    reti ; EE_RDY EEPROM Ready
    reti ; ANA_COMP Analog Comparator
    rjmp TC0CAIsr ; TIM0_COMPA TC0 Compare A
    reti ; TIM0_COMPB TC0 Compare Match B
    reti ; WDT Watchdog Time-out
    reti ; ADC Conversion Complete
;
; ----- Interrupt Service Routines -----
Int0Isr: ; Key interrupt, starts transmit sequence
    sbrc rFlag,bRun ; Skip if not yet running
    reti ; Already running, ignore
    in rSreg,SREG ; Save SREG
    sbr rFlag,(1<<bRun) | (1<<bSta) ; Set flags
    out SREG,rSreg ; Restore SREG
    reti ; Return
; CTC-Timeout Timer, switch IR LED
TC0CAIsr: ; 6 cycles for int and vector rjmp
    in rSreg,SREG ; Save SREG, +1 = 7
    sbiw rCtlL,1 ; CTC count down, +2 = 9
    brne Tc0CAIsrRet ; Not zero, +1/2 = 10/11
    sbr rFlag,1<<bTTO ; Set timeout flag, +1 = 11
    in rimp,TCCR0A ; Read toggle bit, +1 = 12
    eor rimp,rEor ; Reverse, +1 = 13
    out TCCR0A,rimp ; Write new toggle bit, +1 = 14
    TC0CAIsrRet: ; 11/14
        out SREG,rSreg ; Restore SREG, +1 = 12/15
        reti ; Done, +4 = 16/19
;
; ----- Program start, Init -----
Start:
    ; Stack init
    ldi rmp,LOW(RAMEND) ; To RAMEND

```

```

out SPL,rmp ; to stack pointer
; Clock to 2.4 Mcs/s
ldi rmp,1<<CLKPCE ; CLK-Enable
out CLKPR,rmp ; to clock prescaler port
ldi rmp,1<<CLKPS1 ; Prescaler = 4
out CLKPR,rmp ; to clock prescaler port
; Init LED port
sbi PORTB,PORTB0 ; LED off
sbi DDRB,DDB0 ; Pin is output
; Key port pin with pull-up
sbi PORTB,PORTB1 ; Pull-up on
; Set start value
clr rFlag ; Clear flags
ldi rmp,1<<COM0A1 ; Toggle reverse bit
mov rEor,rmp ; to xor register
; Start TC0 as CTC
ldi rmp,cCtc ; Compare A value 12.5 us
out OCR0A,rmp ; to compare A port
; Set OC0A, CTC mode
ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)
out TCCR0A,rmp ; to timer 0 control port A
ldi rmp,1<<CS00 ; Prescaler = 1
out TCCR0B,rmp ; to timer 0 control port B
; INT0 and sleep mode
ldi rmp,(1<<SE)|(1<<ISC01) ; Sleep idle, INT0
out MCUCR,rmp ; to master control port
ldi rmp,1<<INT0 ; Enable INT0 interrupt
out GIMSK,rmp ; in general interrupt mask
; Enable interrupts
sei ; Set I flag in SREG
; for simulation
; cbi PORTB,PORTB1 ; INT0 request
; sbi PORTB,PORTB1

;
; ----- Program loop -----
Loop:
    sleep ; go to sleep
    nop ; After wake up
    sbrc rFlag,bTTo ; Time out?
    rcall TimeOut ; Process time out
    sbrc rFlag,bSta ; Start flag?
    rcall StartOut ; Process transmit start
    rjmp Loop ; go to sleep again
;
; ----- CTC time out -----
TimeOut: ; Flag was set
    cbr rFlag,1<<bTTo ; Clear time out flag, 1
    inc rTOC ; Next transmit phase, 2
    cpi rTOC,cHead ; Send header?, 3
    brcc TimeOut1 ; No, send data bits, 4/5
    ldi ZH,HIGH(TOH2) ; MSB address head, 6
    ldi ZL,LOW(TOH2) ; dto, LSB, 7
    mov rmp,rTOC ; Copy 8
    lsl rmp ; Multiply by two, 9
    add rmp,rTOC ; Plus one, 10
    add ZL,rmp ; Add to address, 11
    ldi rmp,0 ; Overflow adder, 12
    adc ZH,rmp ; Add overflow, 13
    ijmp ; Jump to address in Z, 15
TimeOut1: ; 5
    cpi rTOC,cSign ; All sent?, 6
    brcc TimeOutEnd ; Yes, finalize, 7/8
    in rmp,TCCR0A ; Read toggle/set bit, 8
    sbrc rmp,COM0A1 ; Check toggle on, 9/10
    rjmp TimeOut2 ; Toggle is off, 11
    ; Toggle is on
    ldi rCntH,High(cPaus) ; Load pause duration, 11
    ldi rCntL,Low(cPaus) ; 12
    ret ; Done, 16
TimeOut2: ; Bit transmit, 11
    ldi rCntH,High(cShort) ; Load short bit, 12
    ldi rCntL,Low(cShort) ; 13
    lsl rData7 ; Shift bits, 14
    rol rData6 ; 15
    rol rData5 ; 16
    rol rData4 ; 17
    rol rData3 ; 18

```

```

rol rData2 ; 19
rol rData1 ; 20
rol rData0 ; 21
brcc TimeOut3 ; Bit is zero, 22/23
ldi rCntH,High(cLong) ; Load long bit, 23
ldi rCntL,Low(cLong) ; 24
TimeOut3: ; 23/24
    ret ; done, 27/28
TimeOutEnd: ; 8
    sbrs rFlag,bRst ; Skip next if restart flag set, 9/10
    rjmp Restart ; Restart again, 11
    clr rmp ; Disable timer int, 11
    out TIMSK0,rmp ; 12
    ; Output OC0A to high
    ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01) ; 13
    out TCCR0A,rmp ; 14
    cbr rFlag,(1<<bRun)|(1<<bRst) ; Clear flags, 15
    ret ; Done, 19
;
; Timeout header, set duration
; (All routines must be three instructions long because the
; target address is calculated!)
TOH2: ; 15
    ldi rCntH,High(cTH2) ; 16
    ldi rCntL,Low(cTH2) ; 17
    ret ; 21
;
    ldi rCntH,High(cTH3)
    ldi rCntL,Low(cTH3)
    ret
;
    ldi rCntH,High(cTH4)
    ldi rCntL,Low(cTH4)
    ret
;
    ldi rCntH,High(cTH5)
    ldi rCntL,Low(cTH5)
    ret
;
    ldi rCntH,High(cTH6)
    ldi rCntL,Low(cTH6)
    ret
;
    ldi rCntH,High(cTH7)
    ldi rCntL,Low(cTH7)
    ret
;
    ldi rCntH,High(cTH8)
    ldi rCntL,Low(cTH8)
    ret
;
; Restart: Start a burst
Restart: ; 11
    clr rmp ; Disable timer int, 12
    out TIMSK0,rmp ; 13
    sbr rFlag,1<<bRst ; Set restart flag, 14
    rjmp StartOut1 ; 16
;
StartOut:
    cbr rFlag,(1<<bSta)|(1<<bRst) ; Clear flags
StartOut1:
    ldi ZH,High(2*DataTable)
    ldi ZL,Low(2*DataTable)
    ldi XH,0 ; Point X to register 14
    ldi XL,14
StartOut2:
    lpm rmp,Z+ ; Read table value
    st -X,rmp ; Decrement and store in register
    cpi XL,7 ; Already at register R6?
    brcc StartOut2 ; No, go on
    ser rToC ; Counter to 0xFF
    ldi rCntH,High(cTH1) ; First header duration
    ldi rCntL,Low(cTH1)
    ldi rmp,1<<OCIE0A ; Enable timer int
    out TIMSK0,rmp
    ret
;

```

```

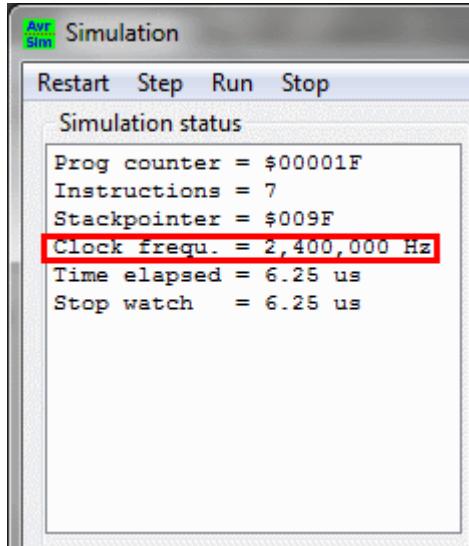
; DataTable
DataTable:
.dw cWord4,cWord3,cWord2,cWord1
;
; End of source code
;

```

The only new instruction is IJMP, a jump to the address in the register pair Z.

12.5.4 Simulating the transmission

The following simulates the transmission using [avr_sim](#).



On startup, following stack pointer init, the following instructions increase the clock speed of the device:

```

118:           ; Clock to 2.4 Mcs/s
119: 00001B    E800 ldi rmp,1<<CLKPCE ; CLKPR-Enable
120: 00001C    BD06 out CLKPR,rmp ; to CLKPR
121: 00001D    E002 ldi rmp,1<<CLKPS1 ; Pr = 4
122: 00001E    BD06 out CLKPR,rmp ; to CLKPR

```

Simulation reflects this change. The controller clock is now at 2.4 MHz.

The instructions

```

123:           ; Init LED port
124: 00001F    9AC0 sbi PORTB,PORTB0 ; off
125: 000020    9AB8 sbi DDRB,DDB0 ; Output

```

have switched the IR LED output PB0 to one and configured it as output. Then instruction

```

126:           ; Key port pin with pull-up
127: 000021    9AC1 sbi PORTB,PORTB1 ; Pull-up

```

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	1	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	Pu	Hi
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

activates the pull-up resistor on the key input. The instructions

```

140:           ; INT0 and sleep mode
141: 00002B    E202 ldi rmp,(1<<SE)|(1<<ISC01) ; Sleep idle, INT0
142: 00002C    BF05 out MCUCR,rmp ; to master control port
143: 00002D    E400 ldi rmp,1<<INT0 ; Enable INT0 interrupt
144: 00002E    BF0B out GIMSK,rmp ; in general interrupt mask

```

enable the INT0 interrupt on pin PB1 on falling edges (key pressed) and sleep mode idle.

The timer/counter 0 has been initiated as CTC with the 12.5µs repetition time, with the prescaler at 1. The PB0 output pin will be set on compare match, setting the IR LED off.

Status		Mode	PortRegisters		Compare A	Compare B	
8	bits	CTC, Compare A	OVF	CmpA	CmpB	PB0 set	PB1 off
TCNT	\$06=6	\$1D=29	\$00=0				
TOP=	(Compare A)						
Prescaler	Clock/1	\$0000=0	Previous	0	Next		

```

132: ; Start TCO as CTC
133: 000025 E10D ldi rmp,cCtc ; Compare A value 12.5 us
134: 000026 BF06 out OCR0A,rmp ; to compare A port
135: ; Set OC0A, CTC mode
136: 000027 EC02 ldi rmp,(1<<COM0A1) | (1<<COM0A0) | (1<<WGM01)
137: 000028 BD0F out TCCROA,rmp ; to timer 0 control port A
138: 000029 E001 ldi rmp,1<<CS00 ; Precaler = 1
139: 00002A BF03 out TCCR0B,rmp ; to timer 0 control port B

```

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	1	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	Pu	Hi
TINnB	0	0			0			
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

Previous **B** Next

By clicking on the INT0 the key interrupt is initiated.

With the next instruction the INT0 service routine is executed.

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	1	1
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	Pu	Hi
TINnB	0	0			0			
INTnB	0	0				0		
PCINT0	0	0	5	4	3	2	1	0

Previous **B** Next

With the instruction

```
97: 00000D 6023 sbr rFlag, (1<<bRun) | (1<<bSta) ; Set flags
```

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	80	02
R16	40	00	03	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

the two flags in the flag register R18 are set. Further handling of the flags are performed outside the service routine, with sleep ended by the interrupt.

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	AA	AA
R8	55	55	AA	AA	55	55	80	02
R16	AA	00	01	00	00	00	00	00
R24	00	00	06	00	00	00	1C	01

The instructions

```

254: StartOut1:
255: 00007C E0F1 ldi ZH,High(2*DataTable)
256: 00007D E1E4 ldi ZL,Low(2*DataTable)
257: 00007E E0B0 ldi XH,0 ; Point X to R14
258: 00007F E0AE ldi XL,14
259: StartOut2:
260: 000080 9105 lpm rmp,Z+ ; Read table
261: 000081 930E st -X,rmp ; Dec and store
262: 000082 30A7 cpi XL,7 ; Already at R6?
263: 000083 F7E0 brcc StartOut2 ; No, go on

```

copy the data table in the flash memory,

```

271: ; DataTable
272: DataTable:
273: .dw cWord4,cWord3,cWord2,cWord1
      00008A 5555 AAAA 5555 AAAA

```

backwards to the registers R13 to R6.

```

266: 000086 EA80 ldi rCntL,Low(cTH1)
267: 000087 E004 ldi rmp,1<<OCIE0A ; Enable timer int

```

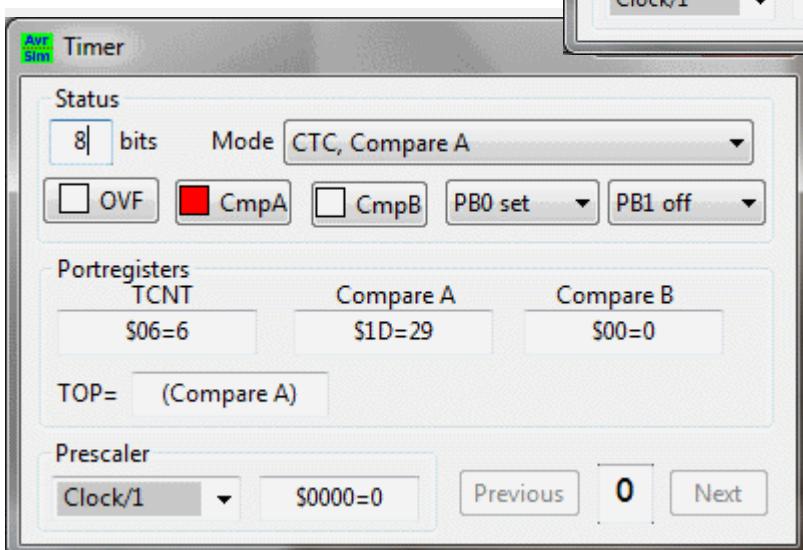
has enabled the compare match A interrupt of the timer. The output control for pin PB0 still is inactive. The interrupt counter has been set to the header bit length:

```

265: 000085 E09F ldi rCntH,High(cTH1) ; First header duration
266: 000086 EA80 ldi rCntL,Low(cTH1)
List of symbols:
Type nDef nUsed Decimalval Hexvalue Name
C 1 2 4000 FA0 CTH1

```

The first interrupt is executed. Note that the counter has already counted forward because of the delay times that the interrupt execution requires (pushing execution address, vector jump).



The instructions

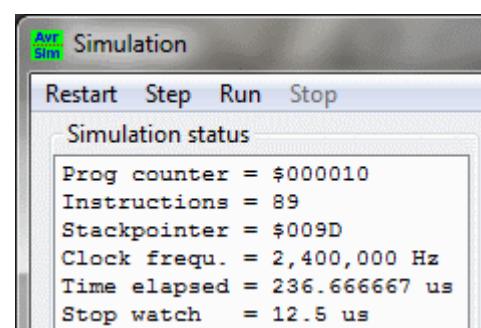
```

103: 000011 9701 sbiw rCntL,1
; count
104: 000012 F421 brne
Tc0CAIsrRet ; Not zero

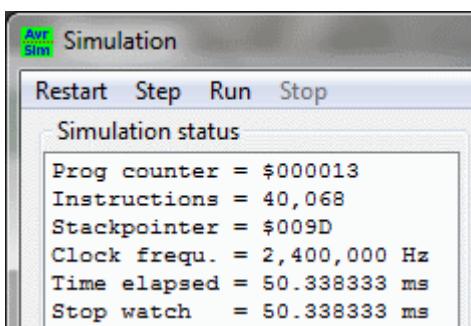
```

decrease the 16 bit counter in R25:R24 and check if zero has been reached by branching on the Z flag in SREG (which is not set yet here).

Interrupt repetition time is at 12.5 µs as designed.



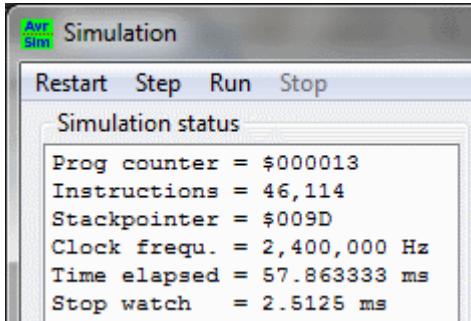
On compare match, PB0 has been toggled. The IR LED is on now for 12.5 µs.



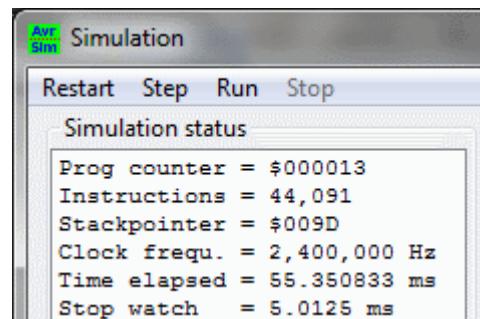
(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	1	0
DDRB	0	0	0	0	0	0	0	1
PINB	0	0	0	0	0	0	Pu	Lo
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

This is the time that the first time out occurs, the counter reached zero. Roughly 50 ms have elapsed.

This is the second phase: the IR LED has been pulsed active for 5 ms.

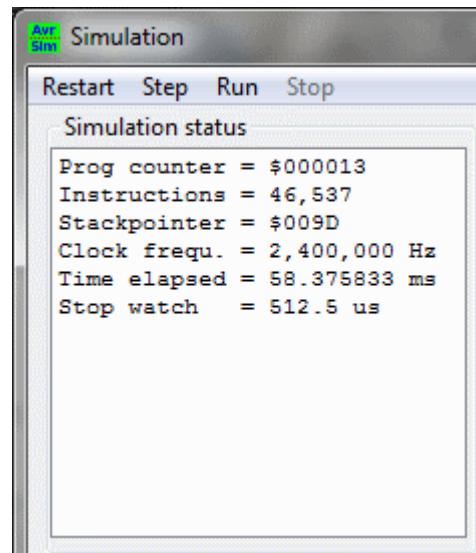


In the third phase, the IR LED is inactive for 2.5 ms.



In the next phase the IR LED is actively pulsed again.

And so on, and so on, until all 40 bits have been transmitted.



[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.5.5 Analysis of the transmitted signals

```
Kopf: .1901.0288.0 U!
0:036A 20 1:0CC3 22
P:0ABB 44
D:AAAA5555AAAA5555U!
```

That is what the ATtiny24 has understood from those transmitted signals (German version). The first two header signals with 0x1901 and 0x0288 (51,208 resp. 5,184 µs) are slightly too long. The number of header signals is four, so the header overflow warning is correct.

The data signals at zero show a sum of 0x036A (=874) for 32 signals, for 32 ones a sum of 0x0CC3. This corresponds to 219 µs resp. 769 µs. The shorter zeroes are slightly too short (should be 250), the longer ones are slightly too long (should be 750). Both can be caused by the filter delays of the receiver modules and is tolerable. The total number of signal pairs is 68 and this is correct.

All eight data bytes were correctly identified.

[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.6 An IR data transfer system

The system consists of two parts:

1. An IR transmitter with an ATtiny13, on which a potentiometer is attached. If the ad-

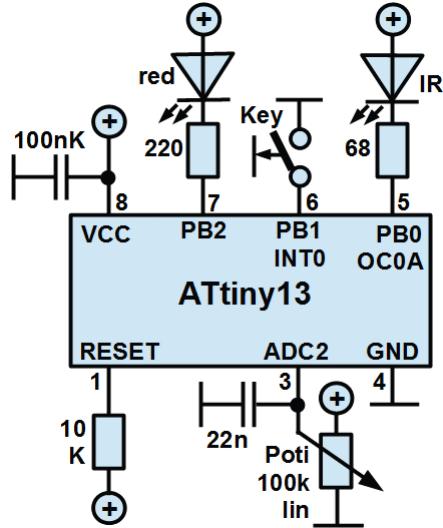
justed value changes or if a predefined time is over, the ATtiny13 transmits the 10 bit result in two 16 bit bursts.

2. An IR receiver with an ATtiny24 and an attached LCD. Received IR burst signals are analyzed and, if correct, the 10 bit result is converted to a percentage and displayed. If errors occur respective messages are displayed.

12.6.1 The IR analog data transmitter

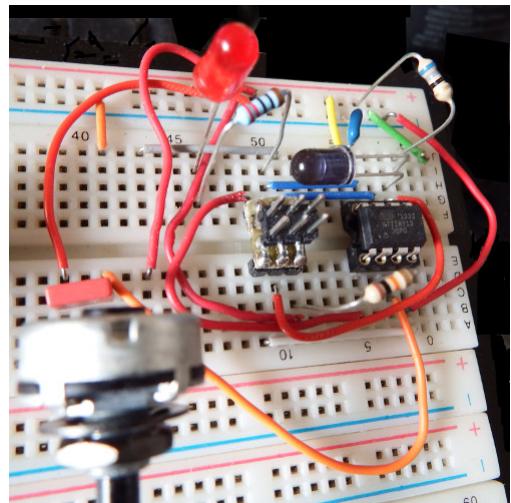
12.6.1.1 Scheme

This here generates the transmit signals. The IR LED on OC0A produces the transmit signal. An additional red LED is on during transmit. The key starts a transmit sequence manually. The potentiometer signal is equipped with a capacitor of 22 nF that suppresses small changes on the ADC input, to reduce the number of transmit bursts.



12.6.1.2 Components

This is the 22 nF capacitor.



12.6.1.3 Mounting

Mounting is not very complicated.

12.6.2 Software

The software is based on the previous template, unnecessary parts were removed. The following lists the software, the [source code is here](#). The only parameter that can be adjusted is the time delay of the default transmit sequence (cAdc10min). This constant is set to 10 seconds and can be increased to up to 3 hours.

```
;
; ****
; * IR transmit an analog value with 40 kcs/s on an ATtiny13 *
; * (C)2017 by www.avr-asm-tutorial.net
; ****
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ----- Registers -----
; free: R0 .. R8
.def r10mL = R9 ; 10 minute counter
.def r10mM = R10
.def r10mH = R11
.def rData0 = R12 ; Transmit register, upper bits
```

```

.def rData1 = R13 ; Transmit register, lower bits
.def rEor = R14 ; For Polarity reversion Toggle/Set OC0A
.def rSreg = R15 ; Save/Restore SREG
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside ints
.def rFlag = R18 ; Flag register
    .equ bRun = 0 ; Transmit sequence running
    .equ bSta = 1 ; Start transmit
    .equ bTTo = 2 ; Time out during transmit
    .equ bRst = 3 ; Restart for second run
    .equ bAdc = 4 ; Analog value complete
.def rTOC = R19 ; Time out counter, counts signals
.def rAnaL = R20 ; Current analog value
.def rAnaH = R21
.def rSntL = R22 ; Last transmitted analog value
.def rSntH = R23
.def rCntL = R24 ; Counts CTC compares
.def rCntH = R25
; Used: XH:XL R27:R26
; free: YH:YL R29:R28
; Used: ZH:ZL R31:R30
;
; ----- Ports and pins -----
.equ pOut = PORTB ; LED/key output port
.equ pDir = DDRB ; LED/key direction port
.equ pIn = PINB ; Key input port
.equ bIrO = PORTB0 ; IR LED output pin
.equ bIrD = DDB0 ; IR LED direction pin
.equ bIrI = PINB0 ; IR LED input pin
.equ bLdO = PORTB2 ; Red LED output pin
.equ bLdD = DDB2 ; Red LED direction pin
.equ bKyO = PORTB1 ; Key output pin for INT0
;
; ----- Constants and timing -----
.equ cClock = 2400000 ; Controller clock
.equ cClockNs = 1000000000 / cClock ; Clock in ns
.equ cIRF = 40000 ; IR-Sendefrequenz Hz
.equ cIRNs = 1000000000 / cIRF / 2 ; IR-TX ns
.equ cCtc = (cIRNs+cClockNs/2) / cClockNs - 1
; @2.4 Mcs/s: 30
;
; ----- Auto transmit and ADC -----
; Clock          2400000 cs/s
; ADC prescaler      128
; ADC measure cycles      13
; ADC measure frequency 1442 cs/s
; Seconds per 10 minutes 600
.equ cAdc10min = 1442*600 ; Three bytes
.equ cAdc10sec = 1442*10
;
; ----- Structure IR signal -----
; All times in us
; Number of header signals, 1 .. 4
.equ nHead = 2 ; Number of header pairs H/L
.equ cHead = 2*nHead-1 ; Number of header signals
; Duration of header signals (Duration in 2*us/25)
.equ cTH1 = 2*5000/25 ; Duration High 1
.equ cTH2 = 2*5000/25 ; Duration Low 1
.equ cTH3 = 2*2500/25 ; Duration High 2
.equ cTH4 = 2*500/25 ; Duration Low 2
; Number of data bits to send
.equ cBits = 16 ; Number of bits
.equ cSign = cHead + 2*cBits ; Total number of signals
; Duration of data bits and active periods
.equ cShort = 2*250/25 ; Duration binary zero
.equ cLong = 2*750/25 ; Duration of binary one
.equ cPaus = 2*250/25 ; Duration of active periods
;
; --- --- --- --- --- ... --- --- ... ---
; |TH1|TH2|TH3|TH4|B16|P16|B15|P15|...|B00|P00|TH1|...|P00|
; --- --- --- --- ... --- --- ... ---
;IR Off On Off On Off On ... Off On Off ... On
; Milliseconds, all data bits zero:
; 50.0   57.5   58.25   58.75   ... 73.25   123.5
;      55.0   58.0   58.5    59.0     73.5     ... 147
; Milliseconds, all data bits one:
; 50.0   57.5   58.75   59.75   ... 88.75   139

```

```

;      55.0    58.0    59.0    60.0    89.0    ... 178
; Duty cycle = 2*(5 + 0,5 + 16*0.25)/2 = 9.5 ms, 6 resp. 5%
; Current cons. LED: I = 39 mA, E = 39*9.5/1000/3600 = 0.11uAh
;                      I = 100mA, E = 100*9.5/1000/3600= 0.26uAh
;
; ----- Reset and interrupt vectors ---
.CSEG
.ORG 0
        rjmp Start ; Reset vector, program init
        rjmp Int0Isr ; INT0 Key interrupt
        reti ; PCINT0 Pin Change Int Request 0
        reti ; TIM0_OVF Timer/Counter Overflow
        reti ; EE_RDY EEPROM Ready
        reti ; ANA_COMP Analog Comparator
        rjmp TC0CAIsr ; TIM0_COMPA TC0 Compare A
        reti ; TIM0_COMPB TC0 Compare Match B
        reti ; WDT Watchdog Time-out
        rjmp AdcIsr ; ADC Conversion Complete
;
; ----- Interrupt Service Routines -----
Int0Isr: ; Key interrupt, starts transmitting sequence
        sbrc rFlag,bRun ; Skip if not yet transmitting
        reti ; Already running, ignore
        in rSreg,SREG ; Save SREG
        sbr rFlag,(1<<bRun) | (1<<bSta) ; Set flags
        out SREG,rSreg ; Restore SREG
        reti ; Return
;
; CTC compare A interrupt, switch IR LED
TC0CAIsr: ; 6 clock cycles for int and vector rjmp
        in rSreg,SREG ; Save SREG, 7
        sbiw rCntL,1 ; Count CTC down, 9
        brne Tc0CAIsrRet ; Not zero, 10/11
        sbr rFlag,1<<bTTO ; Set flag, 11
        in rimp,TCCROA ; Read toggle state, 12
        eor rimp,rEor ; Invert toggle bit, 13
        out TCCROA,rimp ; Write new toggle bit, 14
TC0CAIsrRet: ; 11/14
        out SREG,rSreg ; Restore SREG, 12/15
        reti ; Done, 16/19
;
; ADC conversion complete
AdcIsr:
        in rSreg,SREG ; Save SREG
        in rAnaL,ADCL ; Read result
        in rAnaH,ADCH
        sbr rFlag,1<<bAdc ; Set flag
        ; Restart ADC
        ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
        sbrs rFlag,bRun ; Skip restart if transmit is already running
        out ADCSRA,rimp ; Restart ADC
        out SREG,rSreg ; Restore SREG
        reti
;
; ----- Program start, init -----
Start:
        ; Init stack
        ldi rmp,LOW(RAMEND) ; Point to RAMEND
        out SPL,rmp ; to stack pointer
        ; Controller clock to 2.4 Mcs/s
        ldi rmp,1<<CLKPCE ; CLK-Enable
        out CLKPR,rmp ; to clock prescaler port
        ldi rmp,1<<CLKPS1 ; Prescaler = 4
        out CLKPR,rmp ; to clock prescaler port
        ; Init IR port
        sbi pOut,bIrO ; IR LED off
        sbi pDir,bIrD ; IR pin is output
        ; Init red LED
        sbi pOut,bLdO ; LED off
        sbi pDir,bLdD ; Pin is output
        ; Key port with pull-up
        sbi pOut,bKyO ; Pull-up on
        ; Set start values
        clr rFlag ; Clear flags
        ldi rmp,1<<COM0A1 ; Set toggle/set reversion bit
        mov rEor,rmp ; to exor register
        ; Start TC0 as CTC

```

```

ldi rmp,cCtc ; Compare A value for 12.5 us
out OCR0A,rmp ; to compare A port
; TCO as CTC, set OC0A
ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)
out TCCR0A,rmp ; to TCO control port A
ldi rmp,1<<CS00 ; Prescaler = 1
out TCCR0B,rmp ; to TCO control port B
; Start ADC
ldi rmp,1<<MUX1 ; Set ADC2 as ADC input pin
out ADMUX,rmp ; in AD mux port
; Start first conversion with interrupts enabled
ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
out ADCSRA,rmp ; to ADC control port A
; INTO and sleep mode
ldi rmp,(1<<SE)|(1<<ISC01) ; Sleep mode idle, falling key edges
out MCUCR,rmp ; to master control register
ldi rmp,1<<INT0 ; Enable INT0 interrupts
out GIMSK,rmp ; in general interrupt mask
; Enable interrupts
sei ; Set I flag in SREG
;
; ----- Program loop -----
Loop:
    sleep ; go to sleep
    nop ; After wake up, 20
    sbrc rFlag,bTTo ; Skip if time out flag clear, 21/22
    rcall Timeout ; Process time out, 24
    sbrc rFlag,bSta ; Skip if start flag clear
    rcall StartOut ; Start transmit
    sbrc rFlag,bAdc ; Skip if ADC flag clear
    rcall AdcRdy ; Process ADC
    rjmp Loop ; go to sleep again
;
; ----- Process ADC value -----
AdcRdy:
    cbr rFlag,1<<bAdc ; Clear ADC flag
    mov ZH,rAnaH ; Copy current value to Z
    mov ZL,rAnaL
    adiw ZL,2 ; Tolerance +/- 2 digits
    sub ZL,rSntL ; Subtract last value sent
    sbc ZH,rSntH ; with carry
    brcc AdcRdyUneq ; Start transmission
    cpi ZL,5 ; Above tolerance?
    brcc AdcRdyUneq ; Start transmission
    ; Increase 10 minute counter
    inc r10mL
    brne AdcRdy10m ; No overflow
    inc r10mM ; Inc next higher byte
    brne AdcRdy10m ; No overflow
    inc r10mH ; Increase third counter byte
AdcRdy10m:
    mov rmp,r10mL ; Time limit reached?
    cpi rmp,BYTE1(cAdc10min) ; Compare byte 1
    brne AdcRdyRet ; No
    mov rmp,r10mM
    cpi rmp,BYTE2(cAdc10min) ; Compare byte 2
    brne AdcRdyRet ; No
    mov rmp,r10mH
    cpi rmp,BYTE3(cAdc10min) ; Compare byte 3
    brne AdcRdyRet ; No
    ; 10 minutes are over
    clr r10mL ; Clear timer registers
    clr r10mM
    clr r10mH
AdcRdyUneq: ; Switch ADC off
    ldi rmp,(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
    out ADCSRA,rmp ; to ADC control port A
    nop ; Wait one cycle
    cbr rFlag,1<<bAdc ; Clear ADC flag
    sbr rFlag,(1<<bRun) ; Set start flag
    rjmp StartOut
AdcRdyRet:
    ret
;
; ----- CTC-Timeout -----
TimeOut: ; Time out flag was set, 24
    cbr rFlag,1<<bTTo ; Clear time out flag, 25

```

```

inc rTOC ; Next signal, 26
cpi rTOC,cHead ; Headers to send?, 27
brcc TimeOut1 ; No, data bits, 28/29
ldi ZH,HIGH(TOH2) ; MSB address header table, 29
ldi ZL,LOW(TOH2) ; dto, LSB, 30
mov rmp,rTOC ; Copy counter, 31
lsl rmp ; Multiply by 2, 32
add rmp,rTOC ; Add once, 33
add ZL,rmp ; Add to LSB address, 34
ldi rmp,0 ; Overflow adder, 35
adc ZH,rmp ; Add overflow, 36
ijmp ; Jump to Z, 38
TimeOut1: ; 29
    cpi rTOC,cSign ; All sent?, 30
    brcc TimeOutEnd ; Yes, finalize, 31/32
    in rmp,TCCR0A ; Read toggle/set bit, 32
    sbrc rmp,COM0A1 ; Skip if toggle bit zero, 33/34
    rjmp TimeOut2 ; Toggle is off, 35
    ; Toggle is on
    ldi rCntH,High(cPaus) ; Load pause duration, 35
    ldi rCntL,Low(cPaus) ; 36
    ret ; Done, 40
TimeOut2: ; 35
    ldi rCntH,High(cShort) ; Load short bit duration, 36
    ldi rCntL,Low(cShort) ; 37
    lsl rDataL ; Shift highest bit to carry, 38
    rol rData0 ; 39
    brcc TimeOut3 ; Bit is zero, 40/41
    ldi rCntH,High(cLong) ; Load long bit duration, 41
    ldi rCntL,Low(cLong) ; 42
TimeOut3: ; 41
    ret ; Done, 45/46
TimeOutEnd: ; 32
    sbrs rFlag,bRst ; Skip if restart flag set, 33/34
    rjmp Restart ; No, start new, 35
    clr rmp ; Disable timer ints, 35
    out TIMSK0,rmp ; in TCO interrupt mask, 36
    ; Set OC0A output
    ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01) ; 37
    out TCCR0A,rmp ; in TCO control port A, 38
    ; Update last value sent
    mov rSntH,rAnaH ; To sent register, 39
    mov rSntL,rAnaL ; 40
    ; Clear run flag
    cbr rFlag,(1<<bRun)|(1<<bRst) ; Clear flags, 41
    ; Restart ADC
    ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
    out ADCSRA,rmp ; in ADC control port A, 43
    sbi pOut,bLd0 ; Clear red LED, 45
    ret ; Done, 49
;
; Time out header, adjust duration
TOH2: ; 38
    ldi rCntH,High(cTH2) ; 39
    ldi rCntL,Low(cTH2) ; 40
    ret ; 44
;
    ldi rCntH,High(cTH3)
    ldi rCntL,Low(cTH3)
    ret
;
    ldi rCntH,High(cTH4)
    ldi rCntL,Low(cTH4)
    ret
;
; Restart again, second burst
Restart:
    clr rmp ; Disable timer int, 12
    out TIMSK0,rmp ; 13
    sbr rFlag,1<<bRst ; Set restart flag, 14
    rjmp StartOut1 ; 16
;
; Start transmit
StartOut:
    cbr rFlag,(1<<bSta)|(1<<bRst) ; Clear start and restart flag, 15
    mov rSntH,rAnaH ; Copy measured value to sent registers
    mov rSntL,rAnaL

```

```

Startout1:
    mov rData1,rAnal ; LSB Analog value, 16/17
    mov rData0,rAnaH ; dto, MSB, 17/18
    ser rToC ; Zaehler auf 0xFF, 18/19
    ldi rCntH,High(ctH1) ; Load first signal duration, 21/22
    ldi rCntL,Low(ctH1) ; 22/23
    ldi rmp,1<<OCIE0A ; Enable timer int, 23/24
    out TIMSK0,rmp ; 24/25
    cbi pOut,bLdO ; Set red LED on, 25/26
    ret ; 29/30
;
; End of source code
;

```

12.6.3 Simulation of the software with the Studio

The software builds extensively on exact timings. This can be tested with the Studio's simulation software. The following will show the results (source code in the German version). To sent the (inactive) very long first header phase required the per-calculated number of clock cycles, differences are caused by rounding.

The screenshot shows the AVR Studio 6 interface. On the left, the 'Processor' window displays the state of various registers and counters. The Program Counter is at 0x00006A. The SREG register shows the flags I, T, H, S, V, N, Z, C set to 0. The Stop Watch register shows a value of 49982.50 us. The Cycle Counter is at 119958. The Frequency is listed as 2.4000 MHz. On the right, the assembly code for the `AdcRdyRet:` routine is shown. A yellow arrow points to the instruction `ldi ZH,HIGH(TOK2)`, which is highlighted in green. The code handles CTC-timeouts and data transmission.

```

AdcRdyRet:
    ret
;
; ----- CTC-Timeout -----
TimeOut: ; Flagge war gesetzt
    cbr rFlag,1<<bTTo ; Flagge loeschen
    inc rTOC ; naechstes Zeichen
    cpi rTOC,cKopf ; noch Kopf senden?
    brcc TimeOut1 ; nein, Datenbits
    ldi ZH,HIGH(TOK2) ; MSB Adresse Kopf
    ldi ZL,LOW(TOK2) ; dto, LSB
    mov rmp,rTOC
    lsl rmp ; mal zwei
    add rmp,rTOC ; mal drei
    add ZL,rmp ; zu Adresse
    ldi rmp,0 ; Ueberlauf
    adc ZH,rmp ; addieren

```

This screenshot shows the Processor window with a different set of register values. The Program Counter is now at 0x00006A. The Stop Watch register shows a value of 5012.50 us. The SREG register shows the flags I, T, H, S, V, N, Z, C set to 0. The Cycle Counter is at 12030. The Frequency is listed as 2.4000 MHz.

Name	Value
Program Counter	0x00006A
Stack Pointer	0x9D
X pointer	0x00
Y pointer	0x00
Z pointer	0x0094
Cycle Counter	12030
Frequency	2.4000 MHz
Stop Watch	5012.50 us
SREG	I T H S V N Z C
Registers	

The active phase of the first header signal is a slight bit too long, one additional CTC phase has been performed. This is a tolerable difference.

The inactive phase of the second header signal also is by exactly one CTC phase of 12.5 μ s too long.

This screenshot shows the Processor window with yet another set of register values. The Program Counter is at 0x00006A. The Stop Watch register shows a value of 2512.50 us. The SREG register shows the flags I, T, H, S, V, N, Z, C set to 0. The Cycle Counter is at 6030. The Frequency is listed as 2.4000 MHz.

Name	Value
Program Counter	0x00006A
Stack Pointer	0x9D
X pointer	0x00
Y pointer	0x00
Z pointer	0x0097
Cycle Counter	6030
Frequency	2.4000 MHz
Stop Watch	2512.50 us
SREG	I T H S V N Z C
Registers	

The analysis shows that it is by one CTC phase longer than expected. Those who want it more exact reduce the header time constants by one phase.

Again the inactive phase of a long 1 bit is by one phase longer.

Name	Value
Program Counter	0x00006A
Stack Pointer	0x9D
X pointer	0x00
Y pointer	0x00
Z pointer	0x009A
Cycle Counter	630
Frequency	2.4000 MHz
Stop Watch	262.50 us
SREG	I <input checked="" type="checkbox"/> H <input checked="" type="checkbox"/> S <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> Z <input checked="" type="checkbox"/>
+ Registers	

Also, the inactive phase of a short 0 bit is by one phase longer.

Name	Value
Program Counter	0x00006A
Stack Pointer	0x9D
X pointer	0x00
Y pointer	0x00
Z pointer	0x009A
Cycle Counter	1830
Frequency	2.4000 MHz
Stop Watch	762.50 us
SREG	I <input checked="" type="checkbox"/> H <input checked="" type="checkbox"/> S <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> Z <input checked="" type="checkbox"/>
+ Registers	

The duration of an active signal phase is also slightly longer than expected.

All routines in the main loop need a slightly longer time than the 30 clock cycles available for one CTC round, therefore one CTC cycle is added to every active and inactive cycle. As this additional time is insignificant we do not have to correct this. If you want to do this change all respective constants by subtracting 25/2 from each constant calculation.

Name	Value
Program Counter	0x00006A
Stack Pointer	0x9D
X pointer	0x00
Y pointer	0x00
Z pointer	0x009A
Cycle Counter	1230
Frequency	2.4000 MHz
Stop Watch	512.50 us
SREG	I <input checked="" type="checkbox"/> H <input checked="" type="checkbox"/> S <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> Z <input checked="" type="checkbox"/>
+ Registers	

12.6.3 The IR analog data receiver

12.6.3.1 Hardware

The receiving of the data transmitted by the analog transmitter uses the same hardware (ATtiny24, LCD, IR receiver module) that was already used before, no changes are necessary.

12.6.3.2 Software

The software for receiving, analyzing and displaying the analog transmitter signals is listed as follows, the [source code is here](#), optimized for the process. It requires [the LCD routines](#). In parts the receiver module delivers very different signals than those that were transmitted. Included therefore are program parts that can be used for diagnosis. If you like you can set the switch "Diagnose" to "1" or "2" to use that diagnosis parts.

```
;
; ****
; * IR receiver with ATTiny24 and LCD for analog values *
; * (C)2017 by http://www.avr-asm-tutorial.net           *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
----- Switch -----
. equ Diagnose = 0 ; 0: No diagnose
;                 1: Display data bytes in hex
;                 2: Display all received words in hex
;
```

```

; ----- Durations remote control signals
.equ cLong = 20000 ; Lange Pause
.equ cHead = 2700 ; Dauer High-Signal Kopf us
.equ cZero = 512 ; Dauer Null-Byte us
.equ cOne = 1064 ; Dauer Eins-Byte us
;
; ----- Timing -----
; Controller clock      1.000.000 cs/s
; Time per clock cycle   1 us
; Prescaler TC1          8
; Time per TC1 timer tick 8 us
; Overflow TC1 after ticks 65,536
; Time until TC1 overflows 524.288 ms
.equ cPresc = 8 ; TC1 prescaler
;
; ----- Signal duration and tolerances -----
.equ nLong = cLong/cPresc ; Nominal long signal pause
.equ nHead = cHead/cPresc ; Nominal head signal
.equ nZero = cZero/cPresc ; Nominal zero signal
.equ nOne = cOne/cPresc ; Nominal one signal
.equ cTolerance = 20 ; Tolerance in +/- %
.equ nHeadMin = nHead-(nHead*cTolerance+50)/100
.equ nHeadMax = (nHead*2*cTolerance+50)/100+1
.equ nZeroMin = nZero-(nZero*cTolerance+50)/100
.equ nZeroMax = nZero+(nZero*cTolerance+50)/100+1
.equ nOneMin = nOne-(nOne*cTolerance+50)/100
.equ nOneMax = nOne+(nOne*cTolerance+50)/100+1
;
; Check all data constans
.if nZeroMin > 255
    .error "nZeroMin too large!"
    .endif
.if nZeroMax > 255
    .error "nZeroMax too large!"
    .endif
.if nOneMin > 255
    .error "nOneMin too large!"
    .endif
.if nOneMax > 255
    .error "nOneMax too large!"
    .endif
;
; ----- Ports -----
.equ pIrIn = PINA ; IR receiver port
.equ bIrIn = 0 ; IR receiver port pin
;
; ----- Registers -----
; Used: R0 by LCD, decimal conversion
; Used: R1 Decimal conversion
; free: R2..R5
.def rErrL = R6
.def rErrH = R7
.def rMull = R8 ; Multiplication
.def rMul2 = R9
.def rMul3 = R10
.def rMulH = R11
.def rFlags = R12 ; Flag storage
.def rDataL = R13 ; Receiver data
.def rDataH = R14
.def rSreg = R15 ; Save/restore SREG
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Additional multi purpose
.def rLine = R18 ; LCD line counter
.def rRead = R19 ; LCD read register
.def rimp = R20 ; Multi purpose inside interrupts
.def rFlag = R21 ; Flag register
    .equ bSta = 0 ; Start flag
    .equ bHead = 1 ; Head correct
    .equ bHdSh = 2 ; Head too short
    .equ bHdLg = 3 ; Head too long
    .equ bDSh = 4 ; Data bit too short
    .equ bDMi = 5 ; Data bit middle long
    .equ bDLg = 6 ; Data bit too long
    .equ bDOv = 7 ; Wrong number of data bits
.def rDCtr = R22 ; Data bit counter
; free: R23 .. R25
; Used: XH:XL R27:R26 diverse uses

```

```

; Used: YH:YL R29:R28 Counter signal duration
; Used: ZH:ZL R31:R30 diverse uses
;
; ----- Diagnose SRAM buffer -----
.DSEG
.ORG 0x0060
.if Diagnose == 1
    Buffer:
        .byte 16
    BufferEnd:
        .endif
.if Diagnose == 2
    Buffer:
        .byte 36
    BufferEnd:
        .endif
;
; ----- Reset and interrupts -----
.CSEG ; Code segment
.ORG 0 ; to the beginning
    rjmp Start ; Reset vector, init
    reti ; INT0 External Interrupt Request 0
    rjmp Pci0Isr ; PCINT0 Pin Change Int Req 0
    reti ; PCINT1 Pin Change Interrupt Req 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT Timer/Counter1 Capture Event
    reti ; TIM1_COMPA Timer/Counter1 Compare Match A
    reti ; TIM1_COMPB Timer/Counter1 Compare Match B
    rjmp Tc1Isr ; TIM1_OVF Timer/Counter1 Overflow
    reti ; TIM0_COMPA Timer/Counter0 Compare Match A
    reti ; TIM0_COMPB Timer/Counter0 Compare Match B
    reti ; TIM0_OVF TCO Overflow
    reti ; ANA_COMP Analog Comparator
    reti ; ADC ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr: ; PCINT0 ISR
    sbis pIRIn,bIRIn ; Skip next if IR input high, 1/2
    reti ; Ignore signal, 5
    ; IR input is low, was high before
    in YL,TCNT1L ; Read counter TC1, 3
    in YH,TCNT1H ; 4
    ldi rimp,0 ; Clear counter TC1, 5
    out TCNT1H,rimp ; 6
    out TCNT1L,rimp ; 7
    ldi rimp,1<<TOIE1 ; Enable timer int
    out TIMSK1,rimp ; in timer int mask
    in rSreg,SREG ; Save SREG
    cpi YH,High(nLong) ; Long pause?
    brcc Pci0Isr1 ; Signal was shorter
    ldi rFlag,1<<bSta ; Set start flag
.if diagnose != 0
    ldi XH,High(Buffer) ; Restart buffer
    ldi XL,Low(Buffer)
    .if diagnose == 2
        st X+,YH ; Counter to SRAM
        st X+,YL
        .endif
    .endif
    rjmp Pci0IsrRet
Pci0Isr1: ; Signal is not a long header
    sbrc rFlag,bSta ; Skip next if start flag set?
    rjmp Pci0IsrRet ; Wait until start
    sbrc rFlag,bHead ; Skip next if head flag clear?
    rjmp Pci0Isr2 ; Already correct
.if diagnose == 2
    st X+,YH ; Counter to SRAM
    st X+,YL
    .endif
    subi YL,LOW(nHeadMin) ; Counter minus min head
    sbci YH,HIGH(nHeadMin)
    brcc Pci0IsrErrHS ; Head too short
    subi YL,LOW(nHeadMax) ; Counter minus max head
    sbci YH,HIGH(nHeadMax)

```

```

brcc Pci0IsrErrHL ; Head too long
sbr rFlag,1<<bHead ; Set flag head
clr rDCtr ; Start data bit counter
clr rDataL ; Clear data register
clr rDataH
rjmp Pci0IsrRet
Pci0Isr2: ; Head was correct, check bits
.if Diagnose == 1
    tst YH ; Is MSB zero?
    breq Pci0Isr2d ; Yes
    ldi YL,0xFF ; Mark overflow
Pci0Isr2d:
    st X+,YL ; To SRAM
    rjmp Pci0IsrRet
.endif
.if Diagnose == 2
    st X+,YH ; MSB and
    st X+,YL ; LSB to SRAM
    rjmp Pci0IsrRet
.endif
cpi YL,LOW(nZeroMin) ; Check min zero bit
brcc Pci0IsrErrDSh ; Shorter than min zero bit
cpi YL,LOW(nZeroMax) ; Check shorter than max zero bit
brcc Pci0Isr3 ; Beyond zero bit, check one
clc ; Carry clear
rjmp Pci0Isr4 ; Shift a zero into result registers
Pci0Isr3:
    cpi YL,Low(nOneMin) ; Shorter than min one bit?
    brccs Pci0IsrErrDMi ; Middle long between zero and one
    cpi YL,Low(nOneMax) ; Shorter than max one bit?
    brcc Pci0IsrErrDLg ; Too long for a one
Pci0Isr4:
    rol rDataL ; Roll bit in carry into result registers
    rol rDataH
    inc rDCtr ; Increase number of bits
    cpi rDCtr,17 ; Number of data bits ok?
    brccs Pci0IsrRet ; Yes
    sbr rFlag,1<<bDOv ; Too many bits
    rjmp Pci0IsrErrData ; Error
Pci0IsrErrHS:
    sbr rFlag,(1<<bHead) | (1<<bHdSh) ; Head too short
    ldi rmp,Low(nHeadMin) ; Restore original count
    add YL,rmp
    ldi rmp,High(nHeadMin)
    adc YH,rmp
    rjmp Pci0IsrErrData ; Error
Pci0IsrErrHL:
    sbr rFlag,(1<<bHead) | (1<<bHdLg) ; Head too long
    ldi rmp,Low(nHeadMin+nHeadMax) ; Restore original count
    add YL,rmp
    ldi rmp,High(nHeadMin+nHeadMax)
    adc YH,rmp
    rjmp Pci0IsrErrData ; Error
Pci0IsrErrDSh:
    sbr rFlag,1<<bDSh ; Data bit too short
    rjmp Pci0IsrErrData ; Error
Pci0IsrErrDMi:
    sbr rFlag,1<<bDMi ; Data bit medium long
    rjmp Pci0IsrErrData ; Error
Pci0IsrErrDLg:
    sbr rFlag,1<<bDLg ; Data bit too long
Pci0IsrErrData:
    lsl YL ; N multiplied by 8 (micro seconds)
    rol YH
    lsl YL
    rol YH
    lsl YL
    rol YH
    mov rErrL,YL ; to error register
    mov rErrH,YH
Pci0IsrRet:
    out SREG,rSreg ; Restore SREG
    reti ; Done
;
; TC1 overflow Interrupt Service Routine
Tc1Isr:
    sbrs rFlag,bSta ; Skip next if start bit set

```

```

rjmp TclIsr1 ; Ignore, return
sbrc rFlag,bHead ; Skip next if head flag clear
set ; Set output flag
TclIsr1:
    reti
;
; ----- Start, init -----
Start:
    ldi rmp,LOW(RAMEND) ; Stack to RAMEND
    out SPL,rmp ; to stack pointer
; Init I/O
; Init LCD port output pins
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; To LCD control port
    clr rmp ; LCD control Outputs off
    out pLcdCO,rmp ; to control port output
    ldi rmp,mLcdDRW ; LCD data port mask write
    out pLcdDR,rmp ; to LCD direction port
; Init LCD
    rcall LcdInit ; Call to included LCD routine
    ldi ZH,HIGH(2*LcdStart) ; Display start text
    ldi ZL,LOW(2*LcdStart)
    rcall LcdText
; Start value for flags
    clr rFlag
    clt ; T is display flag
; Init Timer 1, free running
    ldi rmp,1<<CS11 ; Prescaler by 8
    out TCCR1B,rmp ; to TC1 control port B
; PCINT0 for IR Rx
    ldi rmp,1<<PCINT0 ; Pin 0 level change ints
    out PCMSK0,rmo ; to mask port 0
    ldi rmp,1<<PCIE0 ; Enable interrupt PCINT0
    out GIMSK,rmp ; in general interrupt mask
; Sleep enable
    ldi rmp,1<<SE ; Sleep mode idle
    out MCUCR,rmp ; to MCU control port
; Enable interrupts
    sei ; Setting I flag in SREG
Loop:
    sleep ; go to sleep
    nop ; Wake up
    brtc Loop ; Back to sleep if display flag clear
    rcall Update ; Process display flag
    rjmp Loop ; go to sleep again
;
; Process display flag
Update:
    clt ; Clear display flag
    clr rmp ; Disable interrupts TC1
    out TIMSK1,rmp ; to interrupt mask
    mov rmp,rFlag ; Copy error flags
    andi rmp,0xFC ; isolate error flags
    mov rFlags,rmp ; Interim store
    ldi rFlag,0 ; Clear flags for restart
.if Diagnose == 1
    rjmp Diag ; Display diagnose
.endif
.if Diagnose == 2
    rjmp DiagW ; Display diagnose
.endif
    brne UpdateErr ; Display error flags
    cpi rDCtr,16 ; 16 bits received?
    breq UpdateCorrect ; Yes, display
    rjmp UpdateTooFewBits ; Display error
UpdateCorrect:
    ldi ZH,2 ; Set position of LCD to line 3
    clr ZL
    rcall LcdPos
    ldi rmp,'R' ; Display R:
    rcall LcdD4Byte
    ldi rmp,':'
    rcall LcdD4Byte
    ldi ZH,2 ; Set position LCD line 3 col 3
    ldi ZL,2
    rcall LcdPos
    rcall Multi ; Multiply

```

```

    rjmp Percent ; Display as percentage
UpdateErr: ; Display error messages
    rcall LcdLine3 ; Clear from line 3
    ldi ZH,HIGH(2*LcdOutput)
    ldi ZL,LOW(2*LcdOutput)
    rcall LcdTextC
    rcall LcdLine3 ; Set line 3
    ldi rmp,'E' ; Display E:
    rcall LcdD4Byte
    ldi rmp,':'
    rcall LcdD4Byte
    ldi ZH,HIGH(2*ErrHSh) ; Error header too short
    ldi ZL,LOW(2*ErrHSh)
    mov rmp,rFlags ; Copy error flags
    sbrc rmp,bHdSh ; Skip next if header too short clear
    rjmp Update1
    ldi ZH,HIGH(2*ErrHLg) ; Error header too long
    ldi ZL,LOW(2*ErrHLg)
    sbrc rmp,bHdLg ; Skip next if header too long clear
    rjmp Update1
    ldi ZH,HIGH(2*ErrDSh) ; Error data too short
    ldi ZL,LOW(2*ErrDSh)
    sbrc rmp,bDSh ; Skip next if data too short clear
    rjmp Update1
    ldi ZH,HIGH(2*ErrDMi) ; Error data in the middle
    ldi ZL,LOW(2*ErrDMi)
    sbrc rmp,bDMi ; Skip next if data in the middle clear
    rjmp Update1
    ldi ZH,HIGH(2*ErrDLg) ; Error data too long
    ldi ZL,LOW(2*ErrDLg)
    sbrc rmp,bDLg ; Skip next if data too long clear
    rjmp Update1
    ldi ZH,HIGH(2*ErrDBi) ; Error too many data bits
    ldi ZL,LOW(2*ErrDBi)

Update1:
    rcall LcdTextC ; Display error message
    ldi rmp,'='
    rcall LcdD4Byte
    rjmp DecimalY ; Display Y in decimal
;
UpdateTooFewBits: ; Display number of received bits
    rcall LcdLine4
    ldi rmp,'B' ; Display B:
    rcall LcdD4Byte
    ldi rmp,':'
    rcall LcdD4Byte
    rcall DecimalOut ; Display number of bits decimal
    ldi rmp,' '
    rcall LcdD4Byte
    ldi rmp,'b' ; Display bit
    rcall LcdD4Byte
    ldi rmp,'i'
    rcall LcdD4Byte
    ldi rmp,'t'
    rjmp LcdD4Byte
;
; Convert number in rDataH:rDataL to %
.equ cMulti = 1000*256/1023 ; Multiply by 1000/1023 = 250
Multi:
    mov rmp,rDataH ; Copy received MSB
    andi rmp,0x03 ; Clear unused bits
    mov rDataH,rmp ; and copy back
    clr rMul1 ; Clear result registers
    clr rMul2
    clr rMul3
    clr rMulH ; Clear help register
    ldi rmp,cMulti ; Multipliator to rmp
Multil:
    lsr rmp ; Shift multiplicator right
    brcc Multi2 ; No carry, do not add to result
    add rMul1,rDataL ; Add to result
    adc rMul2,rDataH
    adc rMul3,rMulH
Multi2:
    lsl rDataL ; Shift data left
    rol rDataH
    rol rMulH

```

```

tst rmp ; End of multiplication?
brne Multil ; Go on multiplying
ret
;
; Display number in rMul3:rMul2 in decimal on LCD as percentage
Percent:
    ldi ZH,HIGH(2*Decimal) ; Point Z to decimal table
    ldi ZL,LOW(2*Decimal)
    clt ; Suppress leading zeroes
Percent1:
    lpm XL,Z+ ; Read decimal from table to X
    lpm XH,Z+
    tst XL ; End of table?
    breq Percent5 ; Yes, to last digit
    clr rmp ; rmp is counter
Percent2:
    sub rMul2,XL ; Subtract decimal LSB
    sbc rMul3,XH ; MSB with carry
    brcs Percent3 ; Carry occurred
    inc rmp ; count up
    rjmp Percent2 ; Repeat subtraction
Percent3:
    add rMul2,XL ; Retract one subtraction, LSB
    adc rMul3,XH ; dto., MSB with carry
    tst rmp ; Zero?
    brne Percent4 ; No, display
    brts Percent4 ; Leading zero suppression is off
    cpi XL,10 ; Decimal separator?
    brne Percent1 ; No, go on
    set ; Set leading suppression off
Percent4:
    subi rmp,-'0' ; Convert to decimal ASCII
    rcall LcdD4Byte ; Display
    cpi XL,10 ; Decimal separator?
    brne Percent1 ; No
    ldi rmp,'.' ; Display decimal separator
    rcall LcdD4Byte
    rjmp Percent1 ; Continue
Percent5:
    clt ; Clear T flag, used for interrupt signal
    mov rmp,rMul2 ; Last decimal digit
    subi rmp,-'0' ; Convert to ASCII
    rcall LcdD4Byte
    ldi rmp,'%' ; Add percentage char
    rcall LcdD4Byte
    ldi rmp,' ' ; and blank
    rjmp Lcd4Byte
;
; Display number of digits rDCnt as byte in decimal
DecimalOut:
    ldi ZH,HIGH(2*Decimal100) ; Point Z to hundreds
    ldi ZL,LOW(2*Decimal100)
    clt ; Suppress leading zeros
DecimalOut1:
    lpm XL,Z+ ; Read decimal number from table
    lpm XH,Z+
    tst XL ; End of table?
    breq DecimalOut5 ; Yes, last digit
    clr rmp ; rmp is counter
DecimalOut2:
    sub rDCtr,XL ; Subtract decimal number
    brcs DecimalOut3 ; Carry, complete
    inc rmp ; Count up
    rjmp DecimalOut2 ; and subtract again
DecimalOut3:
    add rDCtr,XL ; Retract last subtraction
    tst rmp ; Leading zero?
    brne DecimalOut4 ; No, display
    brts DecimalOut4 ; Leading zero suppression is off
    rjmp DecimalOut1 ; Go on converting
DecimalOut4:
    subi rmp,-'0' ; Convert to decimal ASCII
    rcall LcdD4Byte ; Display character
    set ; Do not suppress leading zeros any more
    rjmp DecimalOut1 ; and repeat
DecimalOut5:
    clt ; Clear T flag

```

```

        mov rmp,rDCtr ; Last digit
        subi rmp,'0' ; Convert to decimal ASCII
        rjmp LcdD4Byte ; and display
;
; Display rErrH:rErrL in decimal format
DecimalY:
    ldi ZH,HIGH(2*Decimal) ; Point Z to decimal table
    ldi ZL,LOW(2*Decimal)
    clt ; Suppress leading zeros
DecimalY2:
    lpm XL,Z+ ; Read decimal from table
    lpm XH,Z+
    tst XL ; End of table?
    breq DecimalY6 ; Yes, to last digit
    clr rmp ; rmp is counter
DecimalY3:
    sub rErrL,XL ; Subtract decimal number from rErr
    sbc rErrH,XH
    brcc DecimalY4 ; Carry, done
    inc rmp ; Increase counter
    rjmp DecimalY3 ; Go on subtracting
DecimalY4:
    add rErrL,XL ; Retract last subtraction
    adc rErrH,XH
    tst rmp ; Result zero?
    brne DecimalY5 ; No
    brts DecimalY5 ; Leading zero suppression off
    rjmp DecimalY2 ; Continue
DecimalY5:
    set ; Do not suppress leading zeros any more
    subi rmp,'0' ; Convert to ASCII
    rcall LcdD4Byte
    rjmp DecimalY2 ; Continue conversion
DecimalY6:
    clt ; Clear T flag
    mov rmp,rErrL ; Last decimal digit
    subi rmp,'0' ; Convert to ASCII
    rcall LcdD4Byte
    ldi rmp,' ' ; Add blank
    rjmp LcdD4Byte
;
; Decimaltable
Decimal: ; 16 bit binary from here
.dw 10000
.dw 1000
Decimal100: ; 8 bit binary from here
.dw 100
.dw 10
.dw 0
;
; Start text on LCD
LcdStart:
.db "IR data receiver 24 ",0x0D,0xFF
.db "avr-asm-tutorial.net",0x0D,0xFF
LcdOutput:
.db " " ,0x0D,0xFF
.db " " ,0xFE,0xFF
;
; Error text
ErrHSh:
.db "Head too short!",0xFE
ErrHLg:
.db "Head too long!",0xFE,0xFE
ErrDSh:
.db "DBit too short!",0xFE
ErrDMi:
.db "DBit middle long!",0xFE
ErrDLg:
.db "DBit too long!",0xFE,0xFE
ErrDBi:
.db "DBits > 16",0xFE,0xFE
;
.if Diagnose == 1 ; Data byte output
Diag:
    ldi rmp,0x01 ; Clear display
    rcall LcdC4Byte
    ldi XH,High(Buffer) ; X to buffer start

```

```

    ldi XL,Low(Buffer)
Diag1:
    ld rmp,X+ ; Buffer byte
    rcall HexOut ; Display in hex
    ldi rmp,' '
    rcall LCD4Byte
    cpi XL,Low(Buffer+6) ; Line 2?
    brne Diag1a
    rcall LCDLine2 ; Line 2
    rjmp Diag2
Diag1a:
    cpi XL,Low(Buffer+12) ; Line 3?
    brne Diag2
    rcall LCDLine3 ; Line 3
Diag2:
    cpi XL,Low(BufferEnd) ; Done?
    brne Diag1 ; No, go on
    ret
    .endif
;
.if Diagnose != 0 ; Display rmp in hex
HexOut:
    push rmp ; Save rmp
    swap rmp ; Upper nibble first
    rcall HexOutNibble ; Display nibble
    pop rmp ; Restore rmp
HexOutNibble:
    andi rmp,0x0F ; Isolate lower nibble
    subi rmp,-'0' ; Convert to ASCII
    cpi rmp,'9'+1 ; A to F?
    brcs HexOutNibble1 ; No
    subi rmp,-7 ; Convert to A to F
HexOutNibble1:
    rjmp LCD4Byte ; Display character
    .endif
;
.if Diagnose == 2 ; Display words
DiagW:
    ldi rmp,0x01 ; Clear LCD
    rcall LCDC4Byte
    ldi XH,HIGH(Buffer) ; X to buffer
    ldi XL,LOW(Buffer)
    ldi ZH,0 ; Line counter
    ldi ZL,0 ; Column counter
DiagW1:
    ld rmp,X+ ; Read MSB
    tst rmp ; MSB zero?
    brne DiagWW ; No, as word
    cpi ZL,19 ; Enough space in that line?
    brcs DiagW2 ; Yes, display byte
    rcall NextLine ; Next line on LCD
DiagW2:
    ld rmp,X+ ; Read LSB
    rcall HexOut ; Display in hex
    subi ZL,-2 ; Add two positions to column count
    rjmp DiagW3 ; Continue
DiagWW:
    cpi ZL,16 ; Enough space in the line for a word?
    brcs DiagWW1 ; Yes
    rcall NextLine ; Next line
DiagWW1:
    rcall HexOut ; Display MSB
    ld rmp,X+ ; Read LSB
    rcall HexOut ; Display LSB
    subi ZL,-4 ; Add four positions to column count
DiagW3:
    cpi ZL,20 ; End of line?
    brcc DiagW4 ; Yes
    ldi rmp,' ' ; Display blank
    rcall LCD4Byte
    subi ZL,-1 ; Add one position to column count
DiagW4:
    cpi XL,Low(BufferEnd) ; End of buffer?
    brcs DiagW1 ; No, go on
    ret
;
NextLine: ; Set LCD position to next line start

```

```

push rmp ; Save rmp
inc ZH ; Next line
clr ZL ; Column = 1
rcall LcdPos ; Set position on LCD
pop rmp ; Restore rmp
ret
.endif
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

One new instruction here is SBCI register,constant. This subtracts the constant from the register and the carry bit as well.

A further new instruction here is ADD register,register. It adds the second register to the first one and sets flags.

12.6.4 Application

IR-Datenempfang tn24
gsc-elektronic.net
R: 73,9%■

That is how it looks like (the German version). And all this without the mighty floating point math library, just with some intelligence added.

[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)

12.7 A self-learning IR receiver with three channel relay switches

12.7.1 Task

Who is not willing to have a small box that can switch electric devices on and off with an ancient remote controller? Now, here it is. And the best is: you do not have to care about your old remote controller and its special encoding of keys, the small box does that for you. This remote control receiver learns the codes that you want to switch your devices with by training it once. It stores the recognized codes in its internal EEPROM and recalls those at power-up. Until the next training exercise has been completed.

12.7.2 Hardware, components and mounting

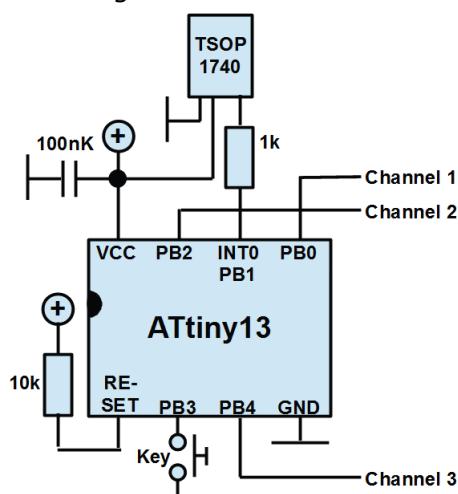
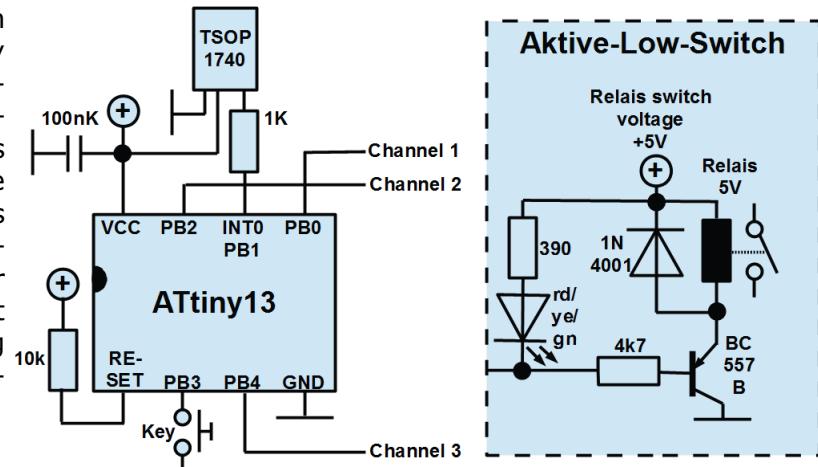
12.7.2.1 Hardware

The hardware comes in two versions:

- Simple, user-friendly and power saving with an ATtiny13, and
- in the comfort version with LCD for the researcher and developer who wants to find out why his special remote controller device does not work on the ATtiny13 version.

This is the ATtiny13 version with 5 V switching stages. If your relay has less than 50 mA current consumption you can skip the transistor driver, but the ISP interface is incompatible with large inductive loads. The 1 k resistor decouples the TSOP from the ISP programming interface. The key or jumper on PB3 serves as an initiation at start-up to go to the self-learning process, it is not used during operation.

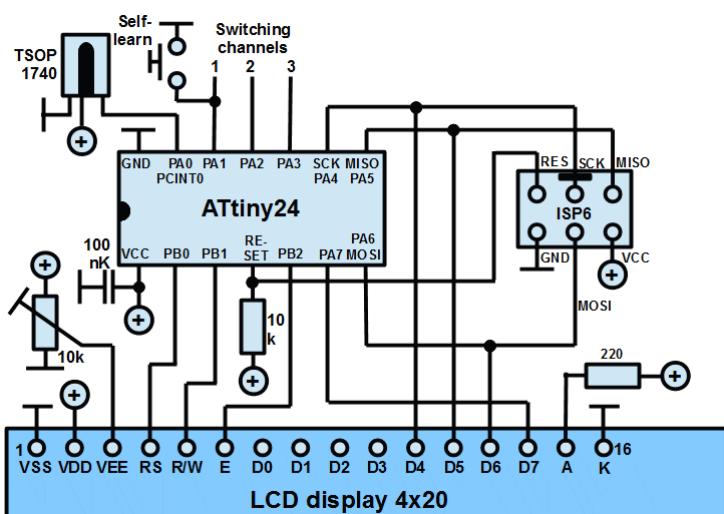
The same scheme, but now with active-high switch drivers.



Now the version for the ATtiny24.

The placing of the output channels can be seen here. PA1 serves to sense the state of the key or jumper at startup. Remove this jumper after startup to enable the self-learning process.

In the design of the switching stage on channel 1 it has to be considered that the internal pull-up on this pin is only about 50 kΩ and that the driver stage must not pull down this to below the state when self-learning is initiated, because this would result in "always self learning". Selecting an active high output signal is therefore not a good idea for that ATtiny24 switch.

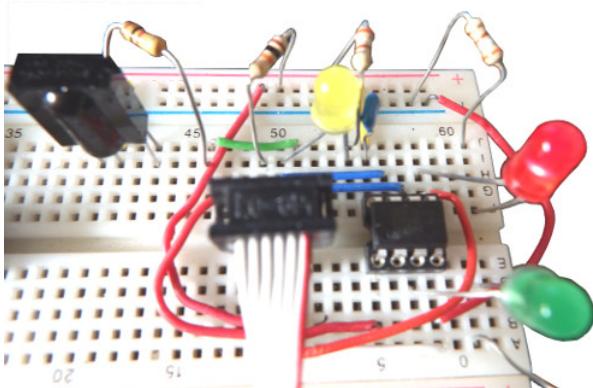


12.7.2.2 Components

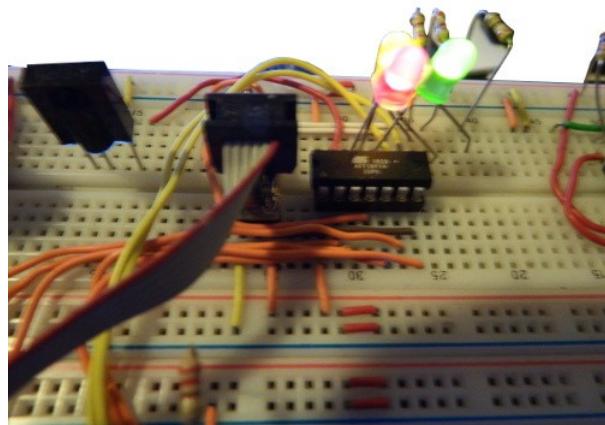
The TSOP receiver module has already been described. The 1 k resistor has the rings brown-black-red-gold (5%) or brown-black-black-brown-brown (1%).

The components for the driver stages are not listed in the hardware table because those are too close to the application and what has to be switched on and off.

12.7.2.3 Mounting



This is the ATTiny13 version. Attached are LEDs only. Those are necessary to follow the self learning process and to signal switching.



The same with the ATTiny24 version.

12.7.3 Program

12.7.3.1 Structure and specialties

The program works similar no matter if it runs on an ATTiny13 or an ATTiny24 with LCD attached. All device specific code is enclosed in .if directives. Of course this applies to the reset and interrupt vector table, too. Because the ATTiny13 does not have a 16 bit timer, time counting is done with the 8 bit timer and an additional MSB register.

As a difference to the previous experiments the timer/counter works with a prescaler of 64. So each timer tick in the Tiny13 is 53, in the Tiny24 64 μ s long. This lower resolution has been proven to be absolutely sufficient.

To increase the precision of recognizing zeros and ones data bits, their duration is averaged over 16 signals. Those durations are stored in a buffer in the SRAM, are summed up and averaged after any burst and the average is stored together with the recognized key codes in the EEPROM.

Recognizing the keys uses only the last 16 bits of a burst. This has been proven as sufficient.

Measuring and recognition of the IR signal is completely performed in the INT0 (ATTiny13) resp. the PCINT0 (ATTiny24) interrupt service routine. Except for two instructions at the beginning this routine is identical for both devices. The algorithm of the ISR is as follows:

- If a long signal occurs (header signal) a restart is initiated (clearing the number of bits received and both shift registers).
- If the signal is short, its duration is compared with the reference value (zero/one discrimination level) and a zero or a one is shifted into the two shift registers.
- In all cases the timer is restarted.

The recognition that all data bits have been received is after a programmable time during which the timer is not cleared by received data bits and the MSB of the timer exceeds the previously chosen value. A flag is then set, the processing of the information is performed in the main loop.

12.7.3.2 Self learning mode

The following conditions initiate the self-learning mode at startup:

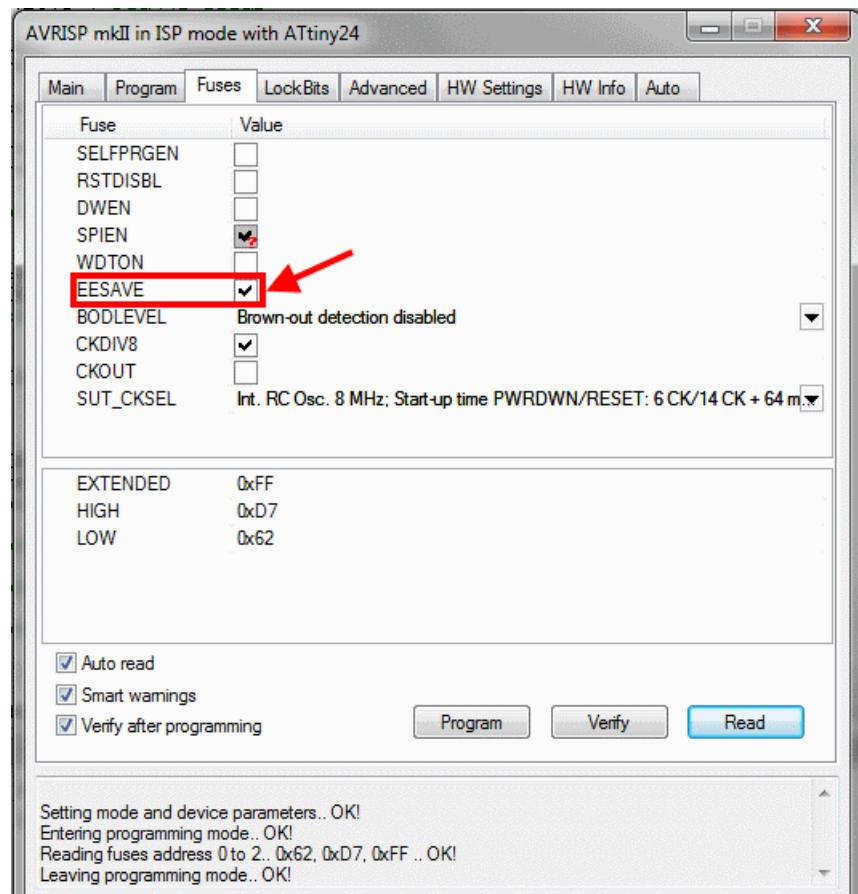
- The key or jumper is activated at startup, or
- The assembler switch cInput is set to 1 prior to assembly, or
- during initially reading the EEPROM the zero/one recognition level is either 0x00 or 0xFF (native or erased EEPROM).

If the self-learning mode is active then

- the LED on the output pin that is to be programmed next blinks in a seconds rhythm,
- the last two bit combinations received are compared,
- if equal it is checked if this bit combination has already been associated to any other channel's on or off combination; if yes, it is not stored and the same channel waits for a different combination,
- if the bit combination is not used by other channels, the combination is associated to the respective channel's on or off recognition, the LED remains on for five seconds and the next channel blinks.
- First the three channel's on recognition is performed, then the three channel's off recognition follow.
- If all six remote control keys have been recognized, the bit combinations are stored in EEPROM and the self-learn mode is switched off.

It should be noted that programming the flash memory with a new version erases the EEPROM content (sets it to 0xFF). So after programming the self-learning mode is entered automatically.

This can be changed by setting the EESAVE fuse. In that case the EEPROM content is not erased.



12.7.3.3 Normal mode

During normal operation the incoming bit combinations are compared with all stored bit combinations. If those are identical to any one combination, the respective channel is switched on or off.

12.7.4 Measured IR remote control codes

The content of the EEPROM can be read with programming tools such as those integrated in ATMEL's Studio. This generates files with the extension .hex that look like this:

```
:100000001108F5C8B5881528754895A80DFF7FFF1C
:10001000FFFFFFFFFFFFFFFFFFFFFFFFF0
:10002000FFFFFFFFFFFFFFFFFFF0
:10003000FFFFFFFFFFFFFFFFFFF0
:10004000FFFFFFFFFFFFFFFFFFFC0
:10005000FFFFFFFFFFFFFFFFFFFB0
```

```
:10006000FFFFFFFFFFFFFFFFFFFA0
:10007000FFFFFFFFFFFFFFFFFFF90
:00000001FF
```

The bytes stored in the EEPROM start with "1108...", the last byte ("1C") is a check sum.

The following lists the content in hex if programmed with different RC equipment (all numbers in hex).

RC device	Red		Yellow		Green		Zero/ One
	On	Off	On	Off	On	Off	
HDR	0805	C8C5	8885	2825	4528	A8A5	11
TV	20DF	10EF	A05F	906F	609F	50AF	12
Camera	41BE	619E	817E	E11E	C13E	A15E	19

While the HDR and the TV RC are faster (Zero/One = 11..12), the camera RC device is less quick (Zero/one = 19).

12.7.5 Diagnosis with the ATTiny24 version with LCD

12.7.5.1 Default display

Displayed are the following information in hex (German version):

- the number of received bits,
- the number of programmed and recognized keys (even: switch on, uneven: switch off),
- Flags (A: Learn mode, D: Double, E: Error),
- 0: Last received bit combination, 1: Previously received bit combination,
- Zero/one discrimination level in timer ticks,
- Comparison result (learn mode only).

```
IR-Rx-Schalter tn24
Bits=20 Sel=5 F1g=
0 = A15E 1 = A15E
Eins = 19 Vergl = U
```

12.7.5.2 Further possibilities

With the switches on top of the source code two further diagnoses can be switched on.

```
22 23 12 22
11 12 22 22
11 11 11 11
23 22 22 11 19■
```

Similarly the switch cDataW lists the results word-wise.

With the switch cDataB set to one the last received 16 data bytes and the calculated zero/one discrimination level are displayed.

```
■011 0012 0011 0011
0022 0023 0011 0012
0022 0011 0011 0011
0012 0011 0011 0023
```

12.7.6 Program

The program for this learning remote control receiver is listed in the following, the [source code in asm format is here](#). If the ATTiny24 with an attached LCD is to be used, the [include file with LCD routines](#) is required.

Prior to assembling the switches on top have to be adjusted to what is needed.

;

```

; ****
; * IR receiver and 3-channel switch with tn13/24 *
; * (C)2017 by www.avr-asm-tutorial.net *
; ****
;
; ----- Switches -----
.equ cAvrType = 13 ; Target device type, ATtiny13 or 24
.equ cActiveH = 1 ; 1 = Channel outputs active high
.equ cLcd = 0 ; 1 = LCD attached
.equ cInput = 0 ; 1 = Self-learning input at startup
.equ cDataB = 0 ; 1 = Display raw data bytes
.equ cDataW = 0 ; 1 = Display raw data words
;
.if (cAvrType != 13) && (cAvrType != 24)
    .error "Wrong device type"
    .endif
;if (cLcd == 1) && (cAvrType == 13)
    .error "LCD does not fit to device type"
    .endif
;
.NOLIST
.if cAvrType == 13
    .INCLUDE "tn13def.inc"
    .else
    .INCLUDE "tn24def.inc"
    .endif
.LIST
;
.if cAvrType == 13
; Hardware: ATtiny13 version
;
;      /_____
; +5V/10k o--|Reset   VCC|--o +5V
; |          |
; Key/Jumpero--|PB3     PB2|--o Ch2 out ye
; |          |
; Ch3 out gn o--|PB4     PB1|--o TSOP1740
; |          |
; 0V o--|GND     PB0|--o Ch1 out rd
; |          |
;
.else
; Hardware: ATtiny24
;
;      /_____
; +5V o--|VCC     GND|--o 0V
; |          |
; LCD-RS o--|PB0     PA0|--o TSOP1740
; |          |
; LCD-R/W o--|PB1     PA1|--o Ch 1 out rd
; |          |
; RESET o--|RES     PA2|--o Ch 2 out ye
; |          |
; LCD-E o--|PB2     PA3|--o Ch 3 out gn
; |          |
; LCD-D7 o--|PA7     PA4|--o LCD-D4/SCK
; |          |
; MOSI/LCD-D6o--|PA6     PA5|--o LCD-D5/MISO
; |          |
;
.endif
;
; ----- Ports, portpins -----
;if cAvrType == 13
; Ports
.equ pOut = PORTB ; Output port
.equ pDir = DDRB ; Direction port
.equ pIn = PINB ; Input port
; Port-bits
.equ bIrO = PORTB1 ; IR receiver output pin
.equ bIrD = DDB1 ; IR receiver direction pin
.equ bIrI = PINB1 ; IR receiver input pin
.equ bRdO = PORTB0 ; Channel output 1 pin red
.equ bRdD = DDB0 ; Channel direction 1 pin red
.equ bYeO = PORTB2 ; Channel output 1 pin yellow
.equ bYeD = DDB2 ; Channel direction 2 pin yellow
.equ bGnO = PORTB4 ; Channel output 3 pin green

```

```

.equ bGnD = DDB4 ; Channel direction 3 pin green
.equ bKyO = PORTB3 ; Key output pin for pull-up
.equ bKyI = PINB3 ; Key input pin
.else
; Ports ATtiny24
.equ pOut = PORTA ; Output port
.equ pDir = DDRA ; Direction port
.equ pIn = PINA ; Input port
; Portpins
.equ bIrO = PORTA0 ; IR receiver output pin
.equ bIrD = DDA0 ; IR receiver direction pin
.equ bIrI = PINA0 ; IR receiver input pin
.equ bRdO = PORTA1 ; Channel 1 output pin red
.equ bRdD = DDA1 ; Channel 1 direction pin red
.equ bYeO = PORTA2 ; Channel 2 output pin yellow
.equ bYeD = DDA2 ; Channel 2 direction pin yellow
.equ bGnO = PORTA3 ; Channel 3 output pin green
.equ bGnD = DDA3 ; Channel 3 direction pin green
.equ bKyO = PORTA1 ; Key output pin for pull-up
.equ bKyD = DDA1 ; Key direction pin
.equ bKyI = PINA1 ; Key input pin
.endif
;
; ----- Timing, IR signals -----
;if cAvrType == 13 ; ATTiny13
; Clock 1200000 cs/s
; TCO prescaler 64
; TCO tick 18.750 cs/s
; 53.33 us
; TCO overflow 13,563 us
.equ cTcTick = 53
;
.else ; ATTiny24
; Clock 1000000 cs/s
; TCO prescaler 64
; TCO tick 15.625 cs/s
; 64 us
; TCO overflow 16.384 us
; Pause until eval 30.000 us
; Medium zeros 448 us
; Medium ones 1,288 us
; Zero/one discrim. 868 us
.equ cTcTick = 64
.endif
; Zero/one discrimination level:
.equ cOne = 868/cTcTick ; tn13:16; tn24:13
; Header/data bit discrimination
.equ cHead = 4*cOne ; tn13:48; tn24:39
; Evaluation pause after last bit received
.equ cEval = 1
; Half second, MSB
.equ cSecH = 500000/cTcTick / 256 ; tn13: 36, tn24: 30
;
; ----- Registers -----
; Used: R0 LCD-Routine (only if LCD attached)
; free: R1 .. R4
.def rBits= R5 ; Received bits
.def rBitsO=R6 ; Received bits display
.def rOne= R7 ; Zero/One detection level
.def rI1L = R8 ; Pre last bit burst, LSB
.def rI1H = R9 ; dto., MSB
.def rI0L = R10 ; Last bit burst, LSB
.def rI0H = R11 ; dto., MSB
.def rIRL = R12 ; Current bit burst, LSB
.def rIRH = R13 ; dto., MSB
.def rCnTH= R14 ; MSB TCO counter
.def rSreg= R15 ; Save SREG
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside interrupts
.def rFlag= R18 ; Flags
    .equ bOvf = 0 ; Overflow, evaluate signal
    .equ bAdj = 1 ; Learn mode active
    .equ bSec = 2 ; Half second reached
    .equ bLng = 3 ; Long confirmation phase
    .equ bTwo = 4 ; Second burst
    .equ bEqu = 5 ; Equal data burst 1 and 2
    .equ bDop = 6 ; Double received

```

```

.equ bErr = 7 ; Burst 1 and 2 not equal
.def rSel = R19
; 0: Red is on, 1: Red is off
; 2: Yellow is on, 3: Yellow is off
; 4: Green is on, 5: Green is off
.def rCtr = R20 ; Counter EEPROM/Input
.def rmo = R21 ; For LCD and LED control
;if cLcd == 1
    .def rLine = R22 ; Line counter LCD
    .def rRead = R23 ; Read result LCD
    .endif
; free: R24 .. R25
; Used: X for pointer operations
; Used: Y for storing in SRAM
; Used: Z for diverse purposes
;
; ----- SRAM -----
.DSEG
.ORG 0x0060
sBuffer: ; Data storage for average calculation
.Byte 16 ; of zeros and ones
sBufferEnd:
;
sCodes:
(Byte 4 ; Recognition red, On/Off
.Byte 4 ; dto., yellow, On/Off
.Byte 4 ; dto., green, On/Off
sCodesKeyEnd:
sCodesOne:
(Byte 1 ; Zero/one discrimination level
sCodesEnd:
;
; ----- Reset and int vectors ---
.CSEG
.ORG 0x0000
;if cAvrType == 13
    rjmp Start ; RESET Vector, init
    rjmp Int0Isr ; INT0 Ext. Int Request 0
    reti ; PCINT0 Pin Change Int Request 0
    rjmp TC0OIsr ; TIM0_OVF TCO Overflow
    reti ; EE_RDY EEPROM Ready
    reti ; ANA_COMP Analog Comparator
    reti ; TIM0_COMPA TCO Compare Match A
    reti ; TIM0_COMPB TCO Compare Match B
    reti ; WDT Watchdog Time-out
    reti ; ADC ADC Conversion Complete
    .else ; ATtiny24
        rjmp Start ; Reset Vector, init
        reti ; INT0 External Int Request 0
        rjmp Pci0Isr ; PCINT0 Pin Change Int 0
        reti ; PCINT1 Pin Change Int Request 1
        reti ; WDT Watchdog Time-out
        reti ; TIM1_CAPT TC1 Capture
        reti ; TIM1_COMPA TC1 Comp Match A
        reti ; TIM1_COMPB TC1 Compare Match B
        reti ; TIM1_OVF Timer/Counter1 Overflow
        reti ; TIM0_COMPA TCO Compare Match A
        reti ; TIM0_COMPB TCO Compare Match B
        rjmp Tc0OIsr ; TC0_OVF, MSB Timer
        reti ; ANA_COMP Analog Comparator
        reti ; ADC ADC Conversion Complete
        reti ; EE_RDY EEPROM Ready
        reti ; USI_STR USI START
        reti ; USI_OVF USI Overflow
    .endif
;
; ----- Interrupt Service Routines -----
; Check pulse from IR receiver
;if cAvrType == 24
Pci0Isr: ; ATtiny24: PCINT0 interrupt
    sbic pIn,bIRI ; Skip next if IR input low
    reti ; Ignore low phase
    .else
Int0Isr: ; ATtiny13: INT0 interrupt
    .endif
    in rSreg,SREG ; Save SREG
;if cDataW == 1 ; Display data words

```

```

    st Y+,rCntH ; Store MSB of counter
    in rimp,TCNT0 ; Read LSB of counter
    st Y+,rimp ; and store
    clr rimp ; Clear TCO
    out TCNT0,rimp
    mov rCntH,rimp
    cpi YL,Low(sBufferEnd) ; Y beyond buffer end
    brcs Int0IsrWRet ; No, continue
    sbr rFlag,1<<bOvf ; Set overflow flag
    ldi YH,High(sBuffer) ; Point to buffer start
    ldi YL,Low(sBuffer)

Int0IsrWRet:
    out SREG,rSreg ; Restore SREG
    reti
    .endif ; End of word storage
    tst rCntH ; Test long signal MSB larger than 0
    brne Int0IsrLong ; Process long signal
    in rimp,TCNT0 ; Read LSB counter
    cpi rimp,cHead ; Compare longer than header
    brcs Int0IsrShort ; No, short signal
    ; Header signal received, restart

Int0IsrLong:
    clr rimp ; Restart counter
    out TCNT0,rimp
    clr rCntH ; Clear MSB counter
    mov rBits0,rBits ; Copy bit count
    clr rBits ; Clear bit count
    clr rIRL ; Clear bit sampler
    clr rIRH
    out SREG,rSreg ; Restore SREG
    reti

Int0IsrShort:
    ; Short data signal
    in rimp,TCNT0 ; Read LSB counter
    st Y+,rimp ; Store in SRAM
    cp rOne,rimp ; Compare zero/one discrimination level
    rol rIRL ; Shift bit into sampler
    rol rIRH
    inc rBits ; Increase bit counter
    cpi YL,Low(sBufferEnd) ; Buffer end?
    brne Int0IsrRet ; Not beyond
    ldi YH,High(sBuffer) ; Restart buffer
    ldi YL,Low(sBuffer)

Int0IsrRet:
    ; Return from short signal
    clr rimp ; Clear TCO
    out TCNT0,rimp
    out SREG,rSreg ; Restore SREG
    reti

;
; Check counter overflows

TC00Isr:
    in rSreg,SREG ; Save SREG
    inc rCntH ; Count overflows
    mov rimp,rCntH ; Compare with longer pause?
    cpi rimp,cSecH ; Is a half second over?
    brne TC00Isr1 ; No
    sbr rFlag,1<<bSec ; Set second flag
    clr rCntH ; Clear MSB counter
    rjmp TC00IsrRet ; Return

TC00Isr1:
    cpi rimp,cEval ; Compare with evaluation period?
    brne TC00IsrRet ; Not yet reached
    tst rBits ; Check if bits received
    breq TC00IsrRet ; No
    sbr rFlag,1<<bOvf ; Set overflow flag
    mov rIOH,rIRH ; Copy received bits
    mov rIOL,rIRL
    mov rBits0,rBits ; Copy number of bits
    clr rBits

TC00IsrRet:
    out SREG,rSreg ; Restore SREG
    reti

;
; ----- Start, init -----
Start:
    ; Init Stack

```

```

ldi rmp,LOW(RAMEND) ; Point to RAMEND
out SPL,rmp ; to stack pointer
; Init port output directions
sbi pDir,bRdD ; Channels, direction
sbi pDir,bYeD
sbi pDir,bGnD
; Port outputs inactive
.if cActiveH == 1
    cbi pOut,bRdO ; Port pins low
    cbi pOut,bYeO
    cbi pOut,bGnO
.else
    sbi pOut,bRdO ; Port pins high
    sbi pOut,bYeO
    sbi pOut,bGnO
.endif
; Init receiver input
sbi pOut,bIrO ; Pull-up IR receiver on
; Clear flags
clr rFlag
; Preset zero/one discrimination level
ldi rmp,cOne ; Default constant
mov rOne,rmp ; to register
; Buffer pointer for receiver values
ldi YH,High(sBuffer) ; To buffer start
ldi YL,Low(sBuffer)
; Read recognition values from EEPROM
rcall ReadCodes
lds rOne,sCodesOne ; Copy zero/one level
; If key input active then enter adjust
.if cAvrType == 24
    cbi pDir,bKyD ; Interim input
    sbi pOut,bKyO ; Switch pull-up on
.else
    sbi pOut,bKyO ; Switch pull-up on
.endif
nop ; Wait a bit
nop
sbis pIn,bKyI ; Skip next if key input high
sbr rFlag,1<<bAdj ; Set flag learn mode
.if cAvrType == 24
    sbi pDir,bKyD ; Channel 1 output again
.endif
; If force input clear all
.if cInput == 1
    rcall StartInput ; Prepare input phase
.else
    sbrc rFlag,bAdj ; Skip if learn mode flag clear
    rcall StartInput ; Prepare input phase
.endif
; If an LCD is attached, init LCD
.if cLcd == 1
    ; Init LCD control ports
    cbi pLcdCO,bLcdCOE ; Clear output E
    cbi pLcdCO,bLcdCORS ; Clear output RS
    cbi pLcdCO,bLcdCORW ; Clear output RW
    sbi pLcdCR,bLcdCRE ; Direction E output
    sbi pLcdCR,bLcdCRRS ; Direction RS output
    sbi pLcdCR,bLcdCRRW ; Direction RW output
    ; Data port LCD
    rcall LcdStart ; Init and text display
.endif
; Start timer TCO
ldi rmp,(1<<CS01)|(1<<CS00) ; Prescaler to 64
out TCCR0B,rmp ; to TCO control port B
ldi rmp,1<<TOIE0 ; Enable TCO overflow int
out TIMSK0,rmp ; in TCO interrupt mask
; Sleep mode idle, INT0 resp. PCINT0 IR receiver
.if cAvrType == 13
    ldi rmp,(1<<SE)|(1<<ISC01) ; Falling edges
    out MCUCR,rmp ; in master control port
    ldi rmp,1<<INT0 ; Enable INT0 interrupts
    out GIMSK,rmp ; in general interrupt mask
.else
    ldi rmp,1<<SE ; Sleep-Mode Idle
    out MCUCR,rmp ; to master control port
    ldi rmp,1<<PCINT0 ; Pin A0 level change ints

```

```

        out PCMSK0,rmp ; to PCINT mask 0
        ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupts
        out GIMSK,rmp ; in general interrupt mask
        .endif
        ; Enable interrupts
        sei ; by setting I flag in SREG

Loop:
        sleep ; go to sleep
        nop ; Wake up
        sbrc rFlag,bOvf ; Skip next if overflow flag clear
        rcall Overflow ; Process overflow
        sbrc rFlag,bSec ; Skip next if second flag clear
        rcall Second ; Process second
        rjmp Loop ; go back to sleep
;

Second:
        ; A half second is over
        cbr rFlag,1<<bSec ; Clear second flag
        ; Only if in adjust mode
        sbrs rFlag,bAdj ; Skip next if adjust flag is set
        ret ; Not in adjust mode
        ; Long confirmation phase?
        sbrs rFlag,bLng ; Skip next if long flag is set
        rjmp LedBlink ; No, go to blink
        ; Long confirmation phase at end?
        dec rCtr ; Decrease counter
        brne SecondRet ; Not ending
        ; Longe confirmation phase is over
        cbr rFlag,1<<bLng ; Clear long flag
        ; Next Input position
        rcall LedOff ; Current LED/channel off
        subi rSel,-2 ; Add two
        cpi rSel,6 ; Already at rSel = 6
        brcs LedBlink ; Not yet, blink
        brne SecondEnd ; All adjusted
        ldi rSel,1 ; Go on with off codes
        rjmp LedBlink ; Blink
SecondEnd:
        ; All channels adjusted
        rcall WriteCodes ; EEPROM write
        clr rSel ; Restart with channel 1 on
        cbr rFlag,1<<bAdj ; Clear adjust flag
        ; All outputs off
.if cActiveH == 1
        cbi pOut,bRdO ; Output low
        cbi pOut,bYeO
        cbi pOut,bGnO
.else
        sbi pOut,bRdO ; Output high
        sbi pOut,bYeO
        sbi pOut,bGnO
.endif
SecondRet:
        ret
;
; Blink LED rSel
LedBlink:
        rcall GetOutPin ; Active pin to rmp
        in rmo,pOut ; Read pin
        eor rmp,rmo ; Inverse polarity
        out pOut,rmp ; Write pins
        ret
;
; LED rSel on
LedOn:
        rcall GetOutPin ; Active pin to rmp
        in rmo,pOut ; Read pins
.if cActiveH == 1
        or rmp,rmo ; Activate pin
.else
        com rmp ; Reverse
        and rmp,rmo ; Clear pin
.endif
        out pOut,rmp ; Write pins
        ret
;
; LED rSel off

```

```

LedOff:
    rcall GetOutPin ; Active pin to rmp
    in rmp,pOut ; Read pins
.if cActiveH == 1
    com rmp ; Reverse
    and rmp,rmo ; Clear active pin
.else
    or rmp,rmo ; Set active pin
.endif
    out pOut,rmp ; Write pins
    ret
;
; Get current output pin to rmp
GetOutPin:
    mov rmp,rSel ; Copy current position
    lsr rmp ; Shift on/off bit to carry
    cpi rmp,1 ; Yellow LED active?
    brcc GetOutPinRd ; No, red
    brne GetOutPinGn ; No, green
    ldi rmp,1<<bYeO ; Yellow active
    ret
GetOutPinRd:
    ldi rmp,1<<bRdO ; Red active
    ret
GetOutPinGn:
    ldi rmp,1<<bGnO ; Green active
    ret
;
; Overflow of the timer, evaluate result
Overflow:
    cbr rFlag,1<<bOvf ; Clear flag
.if cLcd == 1
    .if cDataW == 1
        ldi rmp,0x01 ; Clear LCD
        rcall LcdC4Byte
        ldi XH,High(sBuffer) ; X points to buffer
        ldi XL,Low(sBuffer)
        clr rLine ; Line 0
        RawDataW1:
        ld rmp,X+
        rcall LcdHex
        ld rmp,X+
        rcall LcdHex
        ldi rmp,' '
        rcall LcdD4Byte
        mov rmp,XL
        andi rmp,0x07
        brne RawDataW1
        inc rLine
        cpi rline,4
        brcc RawDataWRet
        mov ZH,rLine
        clr ZL
        rcall LcdPos
        rjmp RawDataW1
        ldi YH,High(sBuffer) ; Y points to buffer
        ldi YL,Low(sBuffer)
        RawDataWRet:
        ret
    .endif
.endif
    ; Check adjust mode
    sbrs rFlag,bAdj ; Skip next if adjust flag set
    rjmp OverflowCheck ; Adjust is off
    ; Learn mode
OverflowCalc:
    clr rmp ; Disable ints
    out GIMSK,rmp
    out TIMSK0,rmp
    ; Calculate average duration of the last 16 bits
    ldi ZH,High(sBuffer) ; Z to buffer
    ldi ZL,Low(sBuffer)
    clr XH ; Clear X as sum
    clr XL
OverflowSum:
    ld rmp,Z+ ; Byte from buffer
    add XL,rmp ; Add to sum

```

```

ldi rmp,0 ; Overflow adder
adc XH,rmp ; Add overflow
cpi ZL,LOW(sBufferEnd) ; End of buffer?
brne OverflowSum ; No, continue
lsr XH ; Divide by 2
ror XL
lsr XH ; by 4
ror XL
lsr XH ; by 8
ror XL
lsr XH ; by 16
ror XL
mov rOne,XL ; Copy to zero/one recognition register
sts sCodesOne,XL ; and to SRAM
; Compare with the previous value
sbrc rFlag,bTwo ; Skip next if second burst flag clear
rjmp OverflowTwo ; Compare two values
sbr rFlag,1<<bTwo ; Set second burst flag
mov rI1L,rI0L ; Copy value
mov rI1H,rI0H
rjmp OverflowNormal
OverflowTwo:
; Compare current and previous value
cbr rFlag,(1<<bTwo) | (1<<bEqu) ; Clear flags
cp rI1L,rI0L ; Compare LSB
brne OverflowNormal ; Unequal
cp rI1H,rI0H ; Compare MSB
brne OverflowNormal ; Unequal
; Equal, check code
sbr rFlag,1<<bEqu ; Set equal flag
ldi XH,High(sCodes) ; X to codes
ldi XL,Low(sCodes) ; in SRAM
OverflowExisting:
cpi XL,Low(sCodesKeyEnd)
breq OverflowKeyOk ; End of codes reached
ld rmp,X+ ; Read LSB
cp rmp,rI1L ; compare LSB
ld rmp,X+ ; Read MSB
brne OverflowExisting ; LSB unequal, next code
cp rmp,rI1H ; Compare MSB
brne OverflowExisting ; MSB unequal, next code
sbr rFlag,1<<bDop ; Set double flag
rjmp OverflowNormal ;
; Store code, start long confirmation
OverflowKeyOk:
ldi XH,High(sCodes) ; Code Start
ldi XL,Low(sCodes)
mov rmp,rSel ; Copy active channel
lsl rmp ; Multiply by two
add XL,rmp ; Add to pointer
ldi rmp,0 ; Overflow adder
adc XH,rmp ; Add carry
st X+,rI1L ; Store value
st X,rI1H
rcall LedOn ; Switch LED on
ldi rCtr,5 ; Long confirmation phase
sbr rFlag,(1<<bEqu) | (1<<bLng) ; Set flags
rjmp OverflowNormal
OverflowCheck:
; Check received data
ldi XH,High(sCodes) ; Pointer to codes
ldi XL,Low(sCodes)
ser rSel ; Start with 0xFF
OverflowCheck1:
inc rSel ; Next value
cpi rSel,6 ; End reached?
brcc OverflowNf ; Yes, not found in list
ld rmp,X+ ; Read value from SRAM
cp rmp,rI0L ; Compare LSB
ld rmp,X+ ; Read MSB from SRAM
brne OverflowCheck1 ; LSB unequal
cp rmp,rI0H ; Compare MSB
brne OverflowCheck1 ; MSB unequal
; Correct code recognized
cbr rFlag,(1<<bErr) | (1<<bDop) ; Clear flags
sbrs rSel,0 ; Skip next if lowest bit set
rcall LedOn ; Switch pin active

```

```

sbrc rSel,0 ; Skip next if lowest bit clear
rcall LedOff ; Switch pin inactive
mov rI1H,rIOH ; Copy counter value
mov rI1L,rI0L
rjmp OverflowNormal
OverflowNf:
; Not equal
clr rSel ; Clear selected channel
sbr rFlag,1<<bErr ; Set error flag
OverflowNormal:
; If LCD is attached: display
.if cLcd == 1
; If raw data bytes
.if cDataB == 1
    ldi rmp,0x01 ; Clear LCD
    rcall LcdC4Byte
    ldi XH,High(sBuffer) ; X pointer to buffer
    ldi XL,Low(sBuffer)
    clr rLine ; To line 0
RawData1:
    ld rmp,X+ ; Read byte from buffer
    rcall LcdHex ; Display in hex
    ldi rmp,' ' ; Blank
    rcall LcdD4Byte
    cpi XL,Low(sBufferEnd) ; Complete?
    brcc RawDataEnd ; Yes
    mov rmp,XL ; Four bytes per line
    andi rmp,0x03 ; Next line?
    brne RawData1 ; No
    inc rLine ; Next line
    mov ZH,rLine
    clr ZL
    rcall LcdPos
    rjmp RawData1 ; Continue
RawDataEnd:
    mov rmp,rOne ; Zero/one level
    rcall LcdHex ; in hex
    .else
; LCD attached, diverse data
    ldi ZH,1 ; Number of bits to line 2
    ldi ZL,5
    rcall LcdPos
    mov rmp,rBits0
    rcall LcdHex
    ldi ZH,1 ; rSel in line 2
    ldi ZL,12
    rcall LcdPos
    mov rmp,rSel
    andi rmp,0x07
    subi rmp,-'0'
    rcall LcdD4Byte
    ldi ZH,1 ; Flags in line 2
    ldi ZL,18
    rcall LcdPos
    ldi rmp,' '
    sbrc rFlag,bAdj
    ldi rmp,'A'
    sbrc rFlag,bErr
    ldi rmp,'E'
    rcall LcdD4Byte
    ldi ZH,2 ; Last two values
    ldi ZL,4 ; in line 3
    rcall LcdPos
    mov rmp,rIOH
    rcall LcdHex
    mov rmp,rI0L
    rcall LcdHex
    ldi ZH,2
    ldi ZL,13
    rcall LcdPos
    mov rmp,rI1H
    rcall LcdHex
    mov rmp,rI1L
    rcall LcdHex
    ldi ZH,3 ; Zero/one level in line 3
    ldi ZL,7
    rcall LcdPos

```

```

        mov rmp,rOne
        rcall LcdHex
        ldi ZH,3 ; Equal/unequal double flag in line 4
        ldi ZL,18
        rcall LcdPos
        ldi rmp,'U' ; Unequal
        sbrc rFlag,bEqu ; Skip if equal
        ldi rmp,'E' ; Equal
        sbrc rFlag,bDop ; Double?
        ldi rmp,'D'
        sbrc rFlag,bErr ; Skip if error flag clear
        ldi rmp,'E'
        rcall LcdD4Byte
        .endif
    .endif
    ; Clear MSB, enable interrupts
    clr rCntH
    ldi rmp,1<<TOIE0 ; Timer Int
    out TIMSK0,rmp
.if cAvrType == 13
    ldi rmp,1<<INT0 ; Enable INT0 interrupts
    out GIMSK,rmp
.else
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupts
    out GIMSK,rmp ; in interrupt mask
.endif
    ret
;
; Read codes from the EEPROM
ReadCodes:
    ldi ZH,0 ; Pointer to EEPROM address
    ldi ZL,0
    ldi XH,High(sCodes) ; Pointer to SRAM
    ldi XL,Low(sCodes)
    ldi rmp,sCodesEnd-sCodes ; Number of bytes
    mov rCtr,rmp
ReadCodes1:
    sbic EECR,EEPE ; Wait until EEPROM ready
    rjmp ReadCodes1
    out EEARL,ZL ; Address port
    sbi EECR,EERE ; Read enable
    in rmp,EEDR ; Read byte
    st X+,rmp ; Store in SRAM
    adiw ZL,1 ; Next address
    dec rCtr ; Count down
    brne ReadCodes1 ; Read further
    cpi rmp,0x00 ; Zero/one level zero?
    breq ReadCodes2 ; Yes, data corrupt
    cpi rmp,0xFF ; Zero/one at FF?
    brne ReadCodes3 ; Yes, data corrupt
ReadCodes2:
    rjmp StartInput ; Data corrupt, restart learning mode
ReadCodes3: ; Data ok
    ret
;
; Write codes to EEPROM
WriteCodes:
    ldi ZH,0 ; Pointer to EEPROM address
    ldi ZL,0
    ldi XH,High(sCodes) ; Pointer to SRAM
    ldi XL,Low(sCodes)
    ldi rmp,sCodesEnd-sCodes
    mov rCtr,rmp
WriteCodes1:
    sbic EECR,EEPE ; Wait until EEPROM ready
    rjmp WriteCodes1 ; Wait on
    clr rmp ; Erase and write mode
    out EECR,rmp ; to control port
    out EEARL,XL ; Address write
    ld rmp,X+ ; Byte from SRAM
    out EEDR,rmp ; to data port
    cli ; Disable interrupts
    sbi EECR, EEMPE ; Set Master Program
    sbi EECR, EEPE ; Program enable
    sei ; Enable interrupts
    dec rCtr ; Byte counter
    brne WriteCodes1 ; Continue

```

```

        ret
;
; Restart: clear all codes, set default zero/one level,
; Channel zero, set adjust flag
StartInput:
    ldi ZH,High(sCodes) ; Pointer to codes
    ldi ZL,Low(sCodes)
    clr rmp ; Clear codes
StartInput1:
    st Z+,rmp ; Write code
    cpi ZL,Low(sCodesEnd) ; Code table end?
    brne StartInput1 ; Go on
    ldi rmp,cOne ; Define default zero/one level
    sts sCodesOne,rmp ; to SRAM
    mov rOne,rmp ; and to register
    clr rSel ; Start with channel 1 and on
    sbr rFlag,1<<bAdj ; Set learn mode
    ret
;
.if cLcd == 1 ; Code for LCD
    ; Load LCD include
    .include "Lcd4Busy.inc"
    ; Start LCD
    LcdStart:
        rcall LcdInit ; Init the LCD
        ldi ZH,High(2*TextOutput)
        ldi ZL,Low(2*TextOutput)
        rjmp LcdText ; Display text
    ; LCD text to be displayed
    TextOutput:
    .db " IR Rx Switch tn24 ",0x0D
    .db "Bits=xx Sel=x Flg=x",0x0D
    ;      5      12     18
    .db "0 = xxxx 1 = xxxx ",0x0D
    ;      4      13
    .db " One = xx Comp = x",0xFE
    ;      7      18
    ;
    ; Byte in rmp in hex on LCD
    LcdHex:
        push rmp ; Save rmp
        swap rmp ; Upper nibble first
        rcall LcdHexN ; Display nibble
        pop rmp ; Restore rmp
        ; Display nibble in hex on LCD
    LcdHexN:
        andi rmp,0x0F ; Mask lower nibble
        subi rmp,-'0' ; Add ASCII-0
        cpi rmp,'9'+1 ; A to F?
        brccs LcdHexN1 ; No
        subi rmp,-7 ; Adjust to A to F
    LcdHexN1:
        rjmp LcdD4Byte ; Send rmp to LCD
.endif
;
; Pre-definition of keys
; (Example TV remote controller)
.ESEG
.ORG 0x00
EeStart:
    ; Red, key 1 on, key 4 off
    .DW 0x0835,0xC8F5
    ; Yellow, key 2 on, key 5 off
    .DW 0x88B5,0x2815
    ; Green, key 3 on, key 6 off
    .DW 0x4875,0xA895
    ; Zero/one discrimination level
    .DB cOne
EeEnd:
;
; End of source code
;

```

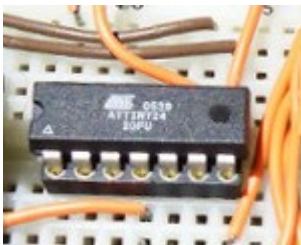
One new instruction used here is ROR register. It rolls the content of the register one position to the right, rolls the carry flag into bit 7 of the register and rolls bit 0 of the register to the carry flag, all in one single step.

The code requires 57% of the flash memory of an ATtiny13 and 45% of the SRAM (without stack operation) and so fits perfectly into this small device:

```
ATtiny13 memory use summary [bytes]:  
Segment Begin End Code Data Used Size Use%  
-----  
[.cseg] 0x000000 0x000248 584 0 584 1024 57.0%  
[.dseg] 0x000060 0x00007d 0 29 29 64 45.3%  
[.eseg] 0x000000 0x00000d 0 13 13 64 20.3%
```

There is enough space left for additional needs.

[Home](#) [Top](#) [IR](#) [Conditioned](#) [Hardware](#) [Measuring](#) [Transmit](#) [Receive](#) [Switch](#)



Lecture 13: Frequency counter and inductance meter

And again something really practical: a frequency counter with conversion of 24 and 40 bit binaries to decimal ASCII. And further a practical device to measure inductance, with squaring and division in assembler.

13.0 Overview

1. [Introduction to frequency measuring](#)
2. [Introduction to decimal conversion](#)
3. [Digital signal measuring with PCINT](#)
4. [Analog signal measuring with analog comparer](#)
5. [Induction measuring with PCINT](#)

13.1 Introduction to frequency measuring

13.1.1 Digital signals and counting limits

Measuring frequencies is rather trivial: one detects polarity changes, counts those during one second and displays those on a LCD. Because one second is a bit long, how about counting for 0.5 or 0.25 seconds?

If the signal is already a digital one we can use the INT0 or any PCINT to detect polarity changes. If INT0 is selected as input, we can select the type of level change that triggers the interrupt (rising, falling or both edges). If both edges are selected or if PCINT is used, two signals trigger the interrupt for each wave.

The complicated task in assembler is then the multiplication of the counted signals by two or four. The C programmer now includes his large math library (and changes to a larger AVR type because the flash is too small). People familiar with assembler just do a double shift or rotate left on the counter registers, such as:

```
; Count result during 0.25 seconds gate time in R3:R2:R1
lsl R1 ; Multiply by two
rol R2
rol R3
lsl R1 ; and by four
rol R2
rol R3
```

and are done.

But why three registers? With those one can count to $256 \times 256 \times 256 - 1 = 16,777,215$ in a quarter second, so up to 66.8 Mcs/s. This is far above what the usual audio generator can generate, and far beyond what an AVR with 1 Mcs/s clock frequency can count. The following interrupt service routine for counting has the listed execution times:

```
; Count interrupt: 4 clock cycles for interrupt trigger plus 2 clock cycles for vector jump
CntIsr:
    in R15,SREG ; Save SREG, +1 = 7
    inc R1 ; Count LSB up, +1 = 8
    brne CntIsrRet ; No overflow, +1/2 = 9/10
    inc R2 ; MSB up, +1 = 10
```

```

brne CntIsrRet ; No overflow, +1/2 = 11/12
inc R3 ; HSB up, +1 = 12
CntIsrRet: ; 10/12/13 clock cycles
out SREG,R15 ; Restore SREG, +1 = 11/13/14
reti ; +4 = 15/17/18

```

With 15 clock cycles at 1 Mcs/s system clock 15 μ s are elapsed and at $1,000,000 / 15 = 66,667$ cs/s counting is at its limit. If we would increase the controller's clock rate to 8 Mcs/s, we would be at 533 kcs/s counting rate.

Another opportunity to count digital signals automatically is the timer/counter. Instead of the controller clock prescaler that we previously used to clock the timer we can select either the input pin T0 to clock the timer/counter TC0 or the input pin T1 to clock the timer/counter TC1, either at rising or falling edges. If we use this we can count up to 250 kcs/s (at 1 Mcs/s clock) or 2 Mcs/s (at 8 Mcs/s clock).

As T1 is on pin 9, together with PA4, and as we use this pin already as data port for the LCD, we run into a conflict in use. With a different type of AVR we can possibly avoid this conflict.

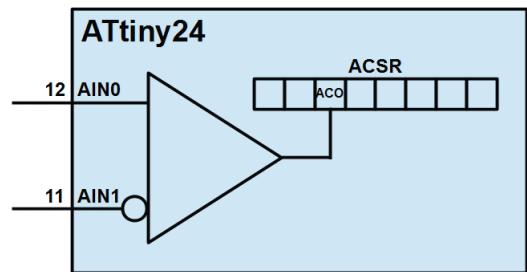
Another method to increase the counting limit would be to increase the controller clock to the maximum possible would be to clock the controller with an external clock source of up to 20 Mcs/s (for which we would need a port B port-pin) or to prescale the digital signal with an external divider by two, four, eight, ten, sixteen or 256 (by that reducing the resolution).

At the other end of the spectrum, at very low frequencies, we can measure the time between two signals, e.g. in μ s, and to calculate the frequency from that. How division with large binaries works is shown later on in that lecture. The good news is that we do not need the large C library for that.

13.1.2 Detecting analog signals

Very often frequency signals are not digital and 5 V level but sine waves with small amplitudes. Signals from a dynamic microphone (5 mV) or a speaker (2 V at 1 Watt power on 4Ω) do not fit to a digital input. The electronic hobbyist now amplifies those signals so that a steep rectangle wave results and uses a Schmitt-trigger to increase its steepness further. Adding that external electronics is disparate: a small 14 pin controller surrounded by three of four times larger external electronics.

And it is completely unnecessary because the AVR has an analog comparator on board. This comparator is an integrated op-amp. Its positive input is attached to pin AIN0 (PA1), the negative input is AIN1 (PA2). The result of the op-amp can be read from the bit ACO in the analog status and control port ACSR. If the ACIE bit in this port is set, any change in this result leads to a analog comparator interrupt and calls the Analog Comparator Interrupt vector (in ATtiny24: vector 0x000C). This mechanism can be used to measure analog frequencies with very small amplitudes by simply counting those interrupts. As interrupts occur every time the comparison result changes each sine wave triggers two interrupts.



13.1.3 Measuring inductivities

Inductors are coils. The more turns the coil has, the higher its inductance. It is measured in Henry (H). Measuring can be based on the impedance Z_L that the coil shows when a sine wave of a certain frequency F is applied. The larger the coil the larger is its resistance Z_L against AC. The formula shows the relation of the impedance Z_L and the inductance L.

$$Z_L \text{ (Ohms)} = 2 * \pi * F \text{ (cs/s)} * L \text{ (H)}$$

The part "2 * π * F" is called angular frequency and abbreviated as ω .

To measure the resistance is a little bit complicated as it is AC. An AD converter would have to measure the voltage many times and would have to determine the maximum of the sine wave. For a small inductance at elevated frequencies this method is not very reliable. More intelligent is it to add a capacity to the inductance, to oscillate this combination and to measure the frequency with which it oscillates. At this oscillation frequency the impedance of the inductance equals the impedance of the capacitor, which is

$$Z_C = 1 / (\omega * C \text{ (Farad)})$$

and therefore decreases with higher frequencies. If $Z_L = Z_C$ are combined the oscillation frequency is:

$$\begin{aligned} Z_L &= Z_C \quad \text{or} \\ \omega * L &= 1 / (\omega * C) \quad \text{or} \\ \omega^2 &= 1 / (L * C) \quad \text{or} \\ \omega &= \sqrt{1/(L * C)} \quad \text{or} \\ F &= 1 / (2 * \pi) * \sqrt{1/(L * C)} \end{aligned}$$

Measuring the frequency F of the resonant circuit delivers the inductance L of the coil as

$$L(H) = 1 / (4 * \pi^2 * C(F) * F(\text{cs/s})^2)$$

The assembler programmer hates such formulas (the C programmer doesn't, invokes the powerful math lib and changes device type to xmega). Not so in assembler, applying some intelligence. As 4 as well as π^2 as well as the capacity C do not change, we have to calculate $1 / (4 * \pi^2 * C)$ only once and divide this constant by F^2 . If we multiply this constant by 1.000.000 the result is in μH directly. For a capacity of 50 nF the constant is 506,605,918,079 or hexadecimal 0x75.F4.10.D7.7F, a 40 bit binary.

To divide this constant by F^2 we first multiply F with itself. The frequency F is a 24 bit long number, which yields a 48 bit long multiplication result. We can skip the upper eight bits of that because that high frequencies cannot be measured. The following table shows the maximum and minimum limits of the division (thousands separator ".", decimal point ",").

Max frequency:	Hex	Decimal	Dimension
$F^2 =$	7FFFFFFFFF	549.755.813.887	(cs/s) ²
$F =$	0B504F	741.455	cs/s
Min frequency:	Hex	Decimal	Dimension
$L =$	003B9AC9FF	999.999.999	μH
$F^2 =$	00000001FB	507	(cs/s) ²
$F =$	000017	23	cs/s

If we limit F^2 to 40 bits, F can be at maximum 741.455 kcs/s. This is beyond what can be measured at 1 Mcs/s clock frequency.

The lower limit results from the fact that an inductance above 999 H is unrealistic. Only frequencies equal or above 23 cs/s come into question, which is low enough and fits all our needs.

13.1.3.1 Division 8 bit by 8 bit

For the inductance display we will need a binary division. The simplest division is 8 by 8 bit. And this goes as follows.

For division by 8 bits we need a further 8 bit register. In the first step the highest bit of the dividend is shifted into bit 0 of this register, by shifting that into the carry and rolling it to the additional register. Now the divider is subtracted from this extra register. If this subtraction yields the carry flag set, the subtraction is taken back by adding the divider again. The reversed carry bit from the subtraction is then shifted left into the result register.

These steps are repeated seven times. The division is complete. In assembler this goes like that:

```

;
; Division 8 bit by 8 Bit
;
    ldi R16,0xEE ; dividend
    ldi R17,10 ; Divider
    ldi R18,8 ; Number of bits
    clr R19 ; Clear dividend extra register
    clr R20 ; Clear result

Shift:
    lsl R16 ; Shift most significant dividend bit to carry
    rol R20 ; Roll into the MSB of the dividend
    sub R20,R17 ; Subtract divider
    brcc One ; Carry is clear, shift a one into the result
    add R20,R17 ; Restore previous MSB dividend
    clc ; Clear result bit in carry
    rjmp Result ; Shift into result

One:
    sec ; Set carry bit

Result:
    rol R19 ; Roll carry bit into result
    dec R18 ; Count down
    brne Shift ; Go on dividing
    ; done

```

The simulation in the Studio says that the division lasts 92 µs.

That is rather simple and quick. And it is even easier than decimal dividing.

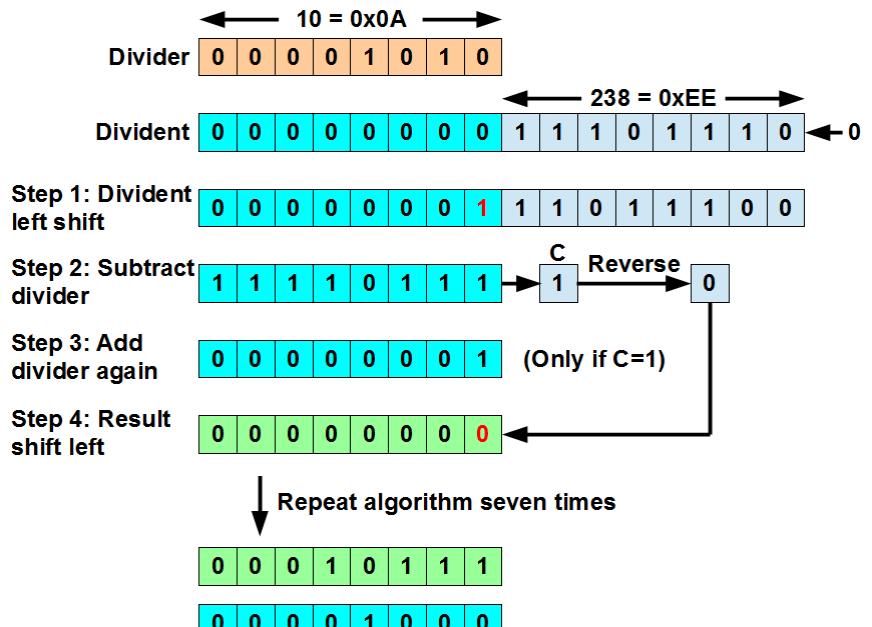
13.1.3.2 Division 16 bit by 8 bit

If a 16 bit long binary has to be divided two registers come into play for the dividend, the extra dividend shifter and the result.

```

;
; Division 16 bit by 8 bit
;
    ldi R31,High(50000) ; dividend
    ldi R30,Low(50000)
    ldi R16,75 ; Divider, LSB
    clr R17 ; MSB dividend
    clr R27 ; Result, MSB
    clr R26 ; dto., LSB
    clr R18 ; Extra dividend, LSB
    clr R19 ; dto., MSB
    ldi R20,16 ; 16 bits, counter

```



```

Shift:
    lsl R30 ; Shift LSB dividend left
    rol R31 ; Roll into MSB, shift bit 7 MSB to carry
    rol R18 ; Roll carry into extra dividend, LSB
    rol R19 ; Roll bit 7 LSB into MSB
    sub R18,R16 ; Subtract Divider, LSB
    sbc R19,R17 ; Subtract MSB with carry
    brcc One ; No carry, shift a one to the result
    add R18,R16 ; Carry set, restore original before subtract
    adc R19,R17 ; Add MSB and carry
    clc ; Shift a zero to the result
    rjmp Result ; Roll into the result

One:
    sec ; Shift a one to result

Result:
    rol R26 ; Roll carry into result, LSB
    rol R27 ; Roll carry into result, MSB
    dec R20 ; Count down
    brne Shift ; Go on dividing
    ; Done

```

Again this is not very complicated. It needs 265 µs.

13.1.3.3 Division 40 bit by 40 bit

After understanding the principle of division it seems easy to extend the division to larger binaries. But dividing the 40 bit dividend by a 40 bit divider using an additional 40 bit long number and resulting in a 40 bit number requires 160 bits or twenty 8 bit registers. Using registers would not leave much registers for other purposes. The solution for this shortage is the SRAM: we place the 40 bit there and by repeating

```

ld R16,Z ; Z points to number in SRAM
rol R16 ; roll into carry
st Z+,R16 ; and store rolled result

```

for five times we have the highest bit of the remaining dividend in the carry flag and can roll that into the extra division register. This needs a bit longer execution time but works fine.

The following table demonstrates the stages of the division for a measured frequency of 1000 cs/s (=0x0003E8, $F^2 = 1.000.000 = 0x0F4240$).

Dec.	Hex	Result hex	Post subtr.	Subtr	Post rolling	Roll	SRAM, Hex	SRAM, Dec
40	28	0000000000	0000000000	0	0000000000	0	75F410D77F	506,605,918,079
39	27	0000000000	0000000001	0	0000000001	1	E8E821AEFE	1,013,211,836,158
38	26	0000000000	0000000003	0	0000000003	1	D7D0435DFC	926,912,044,540
37	25	0000000000	0000000007	0	0000000007	1	AFA086BBF8	754,312,461,304
36	24	0000000000	000000000E	0	000000000E	0	5F410D77F0	409,113,294,832
35	23	0000000000	000000001D	0	000000001D	1	BE821AEFE0	818,226,589,664
34	22	0000000000	000000003A	0	000000003A	0	7D0435DFC0	536,941,551,552
33	21	0000000000	0000000075	0	0000000075	1	FA086BBF80	1,073,883,103,104
32	20	0000000000	00000000EB	0	00000000EB	1	F410D77F00	1,048,254,578,432
31	1F	0000000000	00000001D7	0	00000001D7	1	E821AEFE00	996,997,529,088
30	1E	0000000000	00000003AF	0	00000003AF	1	D0435DFC00	894,483,430,400
29	1D	0000000000	000000075F	0	000000075F	1	A086BBF800	689,455,233,024
28	1C	0000000000	0000000EBE	0	0000000EBE	0	410D77F000	279,398,838,272
27	1B	0000000000	0000001D7D	0	0000001D7D	1	821AEFE000	558,797,676,544
26	1A	0000000000	0000003AFA	0	0000003AFA	0	0435DFC000	18,083,725,312
25	19	0000000000	00000075F4	0	00000075F4	0	086BBF8000	36,167,450,624
24	18	0000000000	000000EBE8	0	000000EBE8	0	10D77F0000	72,334,901,248

Dec.	Hex	Result hex	Post subtr.	Subtr	Post rolling	Roll	SRAM, Hex	SRAM, Dec
23	17	00000000000	000001D7D0	0	000001D7D0	0	21AEFE0000	144,669,802,496
22	16	00000000000	000003AFA0	0	000003AFA0	0	435DFC0000	289,339,604,992
21	15	00000000000	0000075F41	0	0000075F41	1	86BBF80000	578,679,209,984
20	14	00000000000	00000EBE82	0	00000EBE82	0	0D77F00000	57,846,792,192
19	13	00000000001	00000E3AC4	1	00001D7D04	0	1AEFE00000	115,693,584,384
18	12	00000000003	00000D3348	1	00001C7588	0	35DFC00000	231,387,168,768
17	11	00000000007	00000B2450	1	00001A6690	0	6BBF800000	462,774,337,536
16	10	0000000000F	0000070661	1	00001648A1	1	D77F000000	925,548,675,072
15	0F	0000000001E	00000E0CC3	0	00000E0CC3	1	AEFE000000	751,585,722,368
14	0E	0000000003D	00000CD746	1	00001C1986	0	5DFC000000	403,659,816,960
13	0D	0000000007B	00000A6C4D	1	000019AE8D	1	BBF8000000	807,319,633,920
12	0C	000000000F7	000005965A	1	000014D89A	0	77F0000000	515,127,640,064
11	0B	000000001EE	00000B2CB5	0	00000B2CB5	1	EFE0000000	1,030,255,280,128
10	0A	000000003DD	000007172B	1	000016596B	1	DFC0000000	960,998,932,480
9	09	000000007BA	00000E2E57	0	00000E2E57	1	BF80000000	822,486,237,184
8	08	0000000F75	00000D1A6E	1	00001C5CAE	0	7F00000000	545,460,846,592
7	07	0000001EEB	00000AF29D	1	00001A34DD	1	FE00000000	1,090,921,693,184
6	06	0000003DD7	000006A2FB	1	000015E53B	1	FC00000000	1,082,331,758,592
5	05	0000007BAE	00000D45F7	0	00000D45F7	1	F800000000	1,065,151,889,408
4	04	000000F75D	00000B49AF	1	00001A8BEF	1	F000000000	1,030,792,151,040
3	03	000001EEBB	000007511F	1	000016935F	1	E000000000	962,072,674,304
2	02	000003DD76	00000EA23F	0	00000EA23F	1	C000000000	824,633,720,832
1	01	000007BAED	00000E023F	1	00001D447F	1	8000000000	549,755,813,888
0	Rdg	000007BAEE	00000CC23E	1	00001C047E	0	0000000000	0

The result of the division before rounding, 506,606 μH , equals our expected result. The calculation requires 277 μs for the multiplication of F with itself, 2,919 μs for the division and 487 μs for the decimal conversion, in total 3.232 μs for all. Not too long for such a lengthy operation.

We can use this in the software for the inductance meter, without a giant math lib and all within a small and tiny 24.

[Home](#) [Top](#) [Frequencies](#) [Decimal conversion](#) [Digital](#) [Analog](#) [induction](#)

13.2 Introduction to decimal conversion (24 and 32 bit)

Already back in lecture 11 we converted 8 and 16 bit binaries to ASCII and suppressed leading zeros. During the infrared experiments in lecture 12 we made it easy and displayed hexadecimal numbers. Now we have monster numbers with 24 or 32 bits to display. I am not sure what the C programmer now does, in any case he is stuck to his mighty floating point math lib and changes to an xmega type. What makes the rumor that assembler is too complicated out of people that usually are intelligent enough to understand rather complex issues.

The extension of the 16 bit decimal conversion to 24 or 32 long binaries is rather simple if the algorithm is well understood: repeatedly subtracting the binary representation of the decimal digits. Starting with the largest and down to the 10s. In 8 bit our largest was 100, in 16 bit 10,000. In 24 bit the largest is 256 power to three minus 1 = 16.777.215, so we start with

10 millions, at 32 bit (256 powered to 4 minus $1 = 4.294.967.295$) simply with one billion. At 40 bit we would come to a limitation of the assembler that can handle integers only as signed 32 bit binaries. But this can be avoided by splitting the 40 bit binaries into two 24 bit binaries and is demonstrated later on in the course.

[Home](#) [Top](#) [Frequencies](#) [Decimal conversion](#) [Digital](#) [Analog](#) [induction](#)

13.3 Measuring digital signals with the PCINT

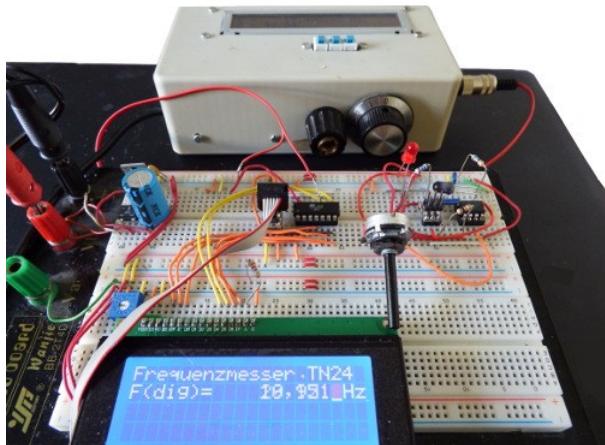
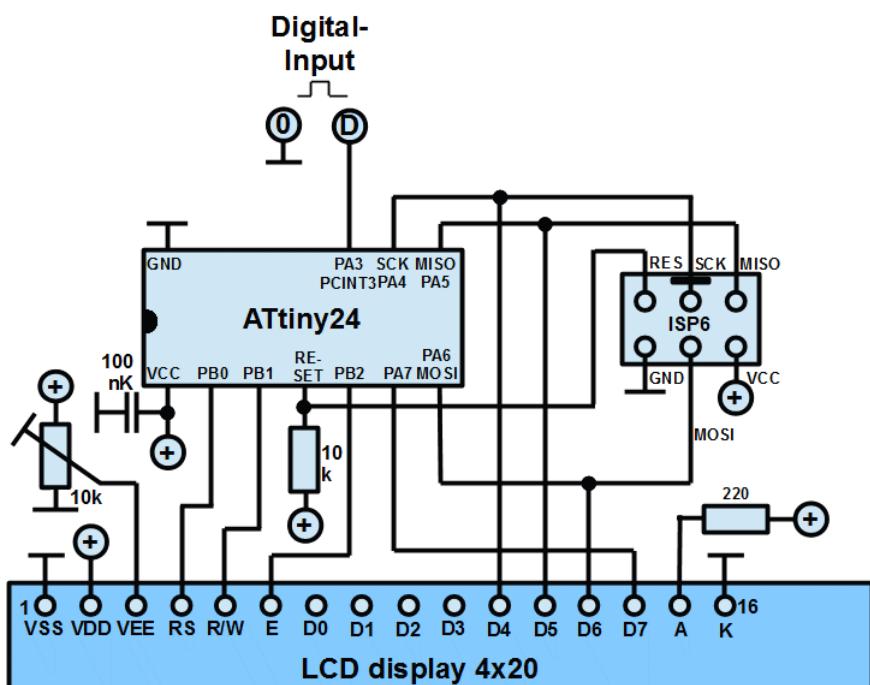
13.3.1 Task

As first task we measure the frequency of digital signals.

13.3.2 Hardware, mounting

The hardware needed for this task is simply nil: the signal source is simply attached to input pin PA3.

Of course, the ATtiny24_LCD experimental board [here](#) can also be used instead.



The source in my case is a home-brewed digital signal generator (see [here](#) for this ATmega8 tool).

13.3.3 Program

The program is listed in the following, the [source code is here](#).

```
;;
; ****
; * Frequency counting of digital signals with ATtiny24/LCD *
; * (C)2017 by www.avr-asm-tutorial.net *
; ****
```

```

;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Switches -----
.equ RawData = 0 ; 1: Display raw data
;           0: Average the last four
.equ debugAverages = 0 ; 1: Jump to average calc
;           0: normal
    .equ New = 0x10203 ; Newest measurement
    .equ Last = New ; Last measurement
    .equ PreLast = New/2 ; PreLast measurement/2
    .equ PrePreLast = New/4 ; PrePreLast measurement/4
    .equ PrePrePreLast=New/4 ; PrePrePreLast measurement/4
;
; ----- Hardware -----
; Digital rrequency counter input on PCINT3/PA3
;
; ----- Timing -----
; Gate time          250 ms
; Controller clock   8,000,000 cs/s
; TC1-Prescaler      64
; TC1 clock          125,000 cs/s
; TC1 clocks in 250 ms 31,250
.equ cTc1CmpA = 31249
;
; ----- Value calculation -----
; Current value      / 2 plus
; Last value         / 4 plus
; Pre last value     / 8 plus
; Pre pre last value / 8 =
;   Current display value
;
; ----- Ports, portpins -----
.equ pOut = PORTA ; Output port for pull-up
.equ pDir = DDRA ; Direction port for pull-up
.equ bIO = PORTB3 ; Pin digital input
.equ bID = DDA3 ; Pin direction digital input
;
; ----- Registers -----
; Used: R0, R1 for LCD
.def rM0L = R2 ; Current value, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
; free: R5 .. R14
.def rSreg = R15 ; Save/Restore SREG
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Additional multi purpose
.def rLine = R18 ; LCD line counter
.def rRead = R19 ; LCD-Register
.def rimp = R20 ; Multi purpose inside interrupts
.def rFlag = R21 ; Flags
    .equ bTO = 0 ; Timeout from timer
.def rHelp = R22 ; Reserve register decimal
; free: R22 .. R25
; Used: R27:R26 X ; for diverse purposes
; free: R29:R28 Y
; Used: R31:R30 Z ; for LCD
;
; ----- SRAM -----
.DSEG
.ORG 0x0060
sM: ; Measure value storage, four values:
; current/2, last/4, pre-last/8,
; pre-pre-last/8
; each as: L(+0), M(+1), H(+2)
.Byte 12
sMEnd:
;
; ---- Reset and interrupt vectors -----
.CSEG
.ORG 0x0000
    rjmp Start ; Reset-Vektor, init
    reti ; INT0 External Int 0
    rjmp Pci0Isr ; PCI Request 0
    reti ; PCINT1 PCI Request 1

```

```

reti ; WDT Watchdog Time-out
reti ; TIM1_CAPT TC1 Capture Event
rjmp Tc1Isr ; TIM1_COMPA TC1 Compare Match A
reti ; TIM1_COMPB TC1 Compare Match B
reti ; TIM1_OVF TC1 Overflow
reti ; TIM0_COMPA TC0 Compare Match A
reti ; TIM0_COMPB TC0 Compare Match B
reti ; TIM0_OVF TC0 Overflow
reti ; ANA_COMP Analog Comparator
reti ; ADC_ADC Conversion Complete
reti ; EE_RDY EEPROM Ready
reti ; USI_STR USI START
reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
Pci0Isr: ; PCINT0 ISR, count pulses on the digital input
    in rSreg,SREG ; Save SREG
    inc rM0L ; Count LSB
    brne Pci0IsrRet ; No overflow
    inc rM0M ; Increase MSB
    brne Pci0IsrRet ; No overflow
    inc rM0H ; Increase HSB
Pci0IsrRet:
    out SREG,rSreg ; Restore SREG
    reti
; TC1 Compare match A interrupt, end of gate time
Tc1Isr: ; Time out counter
    in rSreg,SREG ; Save SREG
    ldi rimp,0 ; Disable PCINT0 interrupt
    out GIMSK,rimp ; in general interrupt mask
    sbr rFlag,1<<bTO ; Set time out flag
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Main program init -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; to stack pointer
    ; Change to 8 Mcs/s clock
    ldi rmp,1<<CLKPCE ; Set change enable bit
    out CLKPR,rmp ; in clock prescaler port
    ldi rmp,0 ; Precaler / 1
    out CLKPR,rmp ; in clock prescaler port
    ; Init LCD port control outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Outputs off
    out pLcdCO,rmp ; to Lcd control port
    ldi rmp,mLcdDRW ; LCD data port write
    out pLcdDR,rmp ; to LCD data port
    ; Init input port
    sbi pOut,bIO ; Input pin pull-up
    cbi pDir,bID ; Input pin as input
.if debugAverages == 1 ; Debug-Code: calc average
    ldi ZH,High(sM) ; Point to values in SRAM
    ldi ZL,Low(sM)
    ldi rmp,Byte3(Last) ; Last value/2
    st Z+,rmp
    ldi rmp,Byte2(Last)
    st Z+,rmp
    ldi rmp,Byte1(Last)
    st Z+,rmp
    ldi rmp,Byte3(PreLast) ; Pre-last value /4
    st Z+,rmp
    ldi rmp,Byte2(PreLast)
    st Z+,rmp
    ldi rmp,Byte1(PreLast)
    st Z+,rmp
    ldi rmp,Byte3(PrePreLast) ; Pre-pre-last value /8
    st Z+,rmp ;
    ldi rmp,Byte2(PrePreLast)
    st Z+,rmp
    ldi rmp,Byte1(PrePreLast)
    st Z+,rmp
    ldi rmp,Byte3(PrePrePreLast) ; Pre-pre-pre last value /8
    st Z+,rmp

```

```

ldi rmp,Byte2(PrePrePreLast)
st Z+,rmp
ldi rmp,Byte1(PrePrePreLast)
st Z+,rmp
Repeat:
ldi rmp,Byte3(New) ; Latest value
mov rM0H,rmp
ldi rmp,Byte2(New)
mov rM0M,rmp
ldi rmp,Byte1(New)
mov rM0L,rmp
rcall Evaluate
rjmp Repeat
.endif
; Init LCD
rcall LcdInit ; Start LCD
ldi ZH,High(2*LcdTextOut) ; Point Z to text
ldi ZL,Low(2*LcdTextOut)
rcall LcdText ; Display text
; Init timer
ldi rmp,High(cTc1CmpA) ; Compare A value
out OCR1AH,rmp ; MSB to compare A port
ldi rmp,Low(cTc1CmpA)
out OCR1AL,rmp ; LSB to compare A port
clr rmp ; TC1 Normal operation
out TCCR1A,rmp
ldi rmp,(1<<CS11)|(1<<CS10) ; Presc 64
out TCCR1B,rmp
ldi rmp,1<<OCIE1A ; Compare Match Int
out TIMSK1,rmp ; Enable
; Activate PCINT3
ldi rmp,1<<PCINT3 ; Pin change PA3
out PCMSK0,rmp ; to mask
ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
out GIMSK,rmp ; to int mask
; Sleep Mode
ldi rmp,1<<SE ; Sleep enable idle
out MCUCR,rmp ; to control port
; Enable interrupts
sei ; Set I bit in SREG
Loop:
sleep ; schlafen legen
nop ; nach Aufwachen
sbrc rFlag,bTO ; Skip next if time out flag clear
rcall Evaluate ; Process evaluate
rjmp Loop
;
; Evaluate counting results
Evaluate:
cbr rFlag,1<<bTO ; Clear time out flag
clr rmp ; Compare Match Int off
out TIMSK1,rmp
.if RawData == 1 ; Debug switch, display raw data
lsl rM0L ; Multiply value by 2
rol rM0M
rol rM0H
.else ; Calculate average
; Shift and divide values in SRAM
ldi ZH,High(sMEnd) ; Z is pointer to target
ldi ZL,Low(sMEnd)
ldi XH,High(sMEnd-3) ; X is pointer to source
ldi XL,Low(sMEnd-3)
ld rmp,-X ; Pre-last to pre-pre-last
st -Z,rmp ; Copy
ld rmp,-X
st -Z,rmp
ld rmp,-X
st -Z,rmp
ld rmp,-X ; Last to pre-last with division
lsr rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp

```

```

ld rmp,-X ; Newest to last with division
lsl rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp
st -Z,rM0H ; Store current value
st -Z,rM0M
st -Z,rM0L
adiw ZL,3 ; Point to last value
ldi rmp,4
mov R0,rmp ; R0 is counter
Evaluate1:
    ld rmp,Z+ ; Read LSB
    add rM0L,rmp ; Add to current
    ld rmp,Z+ ; Read MSB
    adc rM0M,rmp ; Add to current with overflow
    ld rmp,Z+ ; Read HSB
    adc rM0H,rmp ; Add to current with overflow
    dec R0 ; Count down
    brne Evaluate1 ; Add further values
    .endif
.if debugAverages == 1 ; Debug switch
    ret ; Skip LCD display, for debug breakpoint
.endif
    ldi ZH,1 ; LCD display position
    ldi ZL,8
    rcall LcdPos
    rcall DecimalDisplay ; Display decimal
Newstart:
    clr rM0L ; Clear count
    clr rM0M
    clr rM0H
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
    out GIMSK,rmp ; to int mask
    clr rmp ; Clear TC1
    out TCNT1H,rmp ; MSB
    out TCNT1L,rmp ; LSB
    ldi rmp,1<<OCIE1A ; Enable compare match A int
    out TIMSK1,rmp
    ret
;
; 3 byte binary in rM0H:rM0M:rM0L to decimal
; on LCD
DecimalDisplay:
    ldi ZH,High(2*DecimalTab)
    ldi ZL,Low(2*DecimalTab)
    clt ; Supprss leading zeros
DecimalDisplay1:
    lpm XL,Z+ ; Read decimal, LSB
    lpm XH,Z+ ; MSB
    lpm rHelp,Z+ ; HSB
    clr rmp ; Clear divider counter
    cp XL,rmp ; Check table end
    brne DecimalDisplay2 ; No
    cp XH,rmp ; Check MSB
    brne DecimalDisplay2 ; No
    cp rHelp,rmp ; Check HSB
    breq DecimalDisplayEnd ; Calculation end
DecimalDisplay2:
    sub rM0L,XL ; Subtract decimal, LSB
    sbc rM0M,XH ; dto., MSB
    sbc rM0H,rHelp ; dto., HSB
    brccs DecimalDisplay3 ; Overflow
    inc rmp ; Next higher digit
    rjmp DecimalDisplay2 ; Subtract further
DecimalDisplay3:
    add rM0L,XL ; Take back last subtraction, LSB
    adc rM0M,XH ; MSB
    adc rM0H,rHelp ; HSB
    tst rmp ; Zero?
    brne DecimalDisplay4 ; Not zero
    brts DecimalDisplay5 ; Do not suppress zero
    ldi rmp,' ' ; Blank

```

```

rcall LcdD4Byte ; to LCD
ldi rmp,' ' ; Check decimal thousand
rjmp DecimalDisplayKomma
DecimalDisplay4:
    set ; Do not suppress zeros any more
DecimalDisplay5:
    subi rmp,-'0' ; Convert to ASCII
    rcall LcdD4Byte ; To LCD
    ldi rmp,', ' ; Display a thousands separator
DecimalDisplayKomma:
    cpi XL,Byte1(1000000) ; A million?
    breq DecimalDisplayKomma1
    cpi XL,Byte1(1000) ; One thousand?
    breq DecimalDisplayKomma1
    rjmp DecimalDisplay1 ; Continue
DecimalDisplayKomma1:
    rcall LcdD4Byte ; Output rmp
    rjmp DecimalDisplay1 ; Continue
DecimalDisplayEnd:
    ldi rmp,'0' ; Last digit
    add rmp,rM0L ; Add rest
    rjmp LcdD4Byte ; Display and return
;
DecimalTab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(1000000),Byte3(100000)
.db Byte1(100000),Byte2(10000)
.db Byte3(100000),Byte1(1000)
.db Byte2(10000),Byte3(1000)
.db Byte1(1000),Byte2(100)
.db Byte3(100),Byte1(10)
.db Byte2(10),Byte3(10)
.db 0,0,0
;
; LCD Start text
LcdTextOut:
.db "Frequency meter tn24",0x0D,0xFF
.db "F(dig)= x,xxx,xxx Hz",0x0D,0xFF
;           8
.db "                 ",0x0D,0xFF
.db "                 ",0xFE,0xFE
;
; LCD include routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

13.3.4 Examples



With a xtal controlled signal generator the result is as follows:

Frequenzmesser TN24
F(dig)= 19.962 Hz

The two results are not equal, the ATtiny24 result is too low by roughly 0.19%. This might be tolerable for most of the applications. It seems that the ATtiny24 runs with a slightly higher RC frequency. That is due to the rather rough adjustment of the RC, which is adjusted at lower operating voltages and runs faster at 4.8 V. To achieve a higher accuracy we can change the oscillator calibrator byte, as described in the device's data-book. Also we can adjust with a multiplication, but 0.19% is not a large difference and we have to use 17 bit multiplication (result = raw result * 65661 / 65536).

13.4 Frequency measurement with the analog comparator

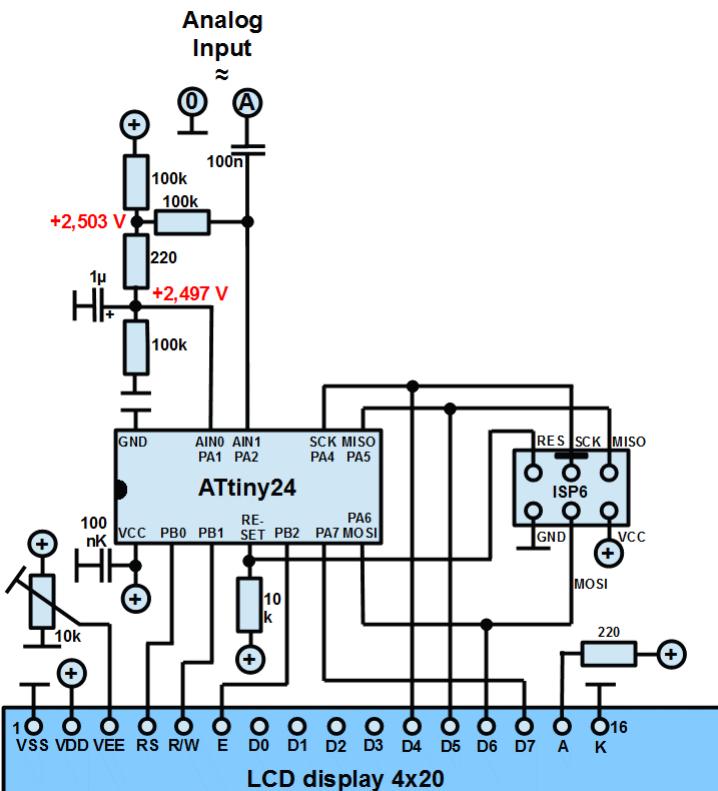
13.4.1 Task

The frequency of sine wave AC with 5 mV (eff.) amplitude and above shall be measured and displayed.

13.4.2 Hardware and components

13.4.2.1 Scheme

This is the necessary hardware to measure. The main components are a voltage divider. This divider provides a voltage that is roughly half of the operating voltage, to which the negative op-amp input is tied to and AC is blocked with the $1\ \mu F$ capacitor. The small $220\ \Omega$ resistor provides a roughly 5 mV higher voltage to which the other input is connected via a $100\ k$ resistor. This design blocks signal changes on the input line that have a very small amplitude (open input). The AC to be measured comes via a $100\ nF$ capacitor and modulates the voltage divider's DC.



13.4.2.2 Components



This is the $1\ \mu F$ electrolytic capacitor. The longer of the two wires is the plus pole, be sure to have it polarized in the right direction.



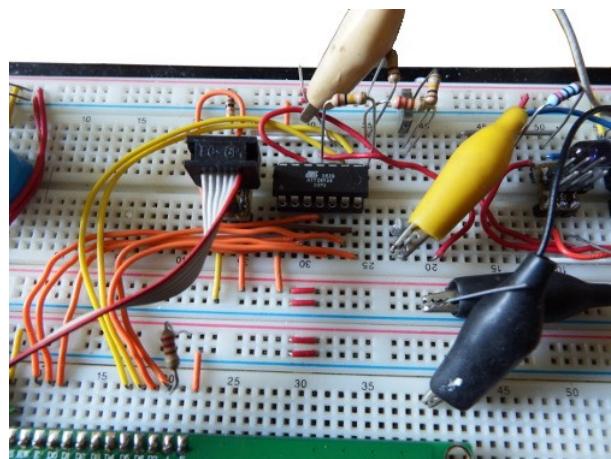
This is a possible form of the film capacitor of $100\ nF$.

These are the three $100\ k\Omega$ resistors, from which the voltage divider is made of and that feed the DC to the sensing input. The $220\ \Omega$ resistor was shown in an earlier lecture.



13.4.2.3 Mounting

The mounting looks like this. If you shorten the resistor wires and if you mount the components in a compact manner you get less noise on the input and the measurement is more stable.



13.4.3 Program

The program is listed in the following, the [source code is here](#). A very special condition occurs here: the operation of the analog comparator and the sleep mode are conflicting. Sometimes, and unpredictable, the controller does not wake up on interrupts. Neither comparator nor timer interrupts wake up the CPU, the controller goes to deep sleep and is dead. Only a reset wakes up the controller again. ATMEL confirms this error in the device handbook. This software works without sleep mode, therefore.

Counting events from the PCINT on the digital input as well as those from the analog comparator are measured in two time phases with only one of the two interrupts enabled, one after the other. And the software uses the same interrupt service routine, the software part to calculate averages uses two different value buffers in the SRAM.

```
; ****
; * Frequency meter analog and digital with ATtiny24/LCD *
; * (C)2017 by www.avr-asm-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Hardware -----
; Analog frequency counter on analog-
;   comparator AIN0/PA1 and AIN1/PA2
; Digital frequency counter on PCINT3/PA3
;
; ----- Timing -----
; Gate time          250 ms
; Controller clock  8.000.000 cs/s
; TC1 prescaler     64
; TC1 clock          125.000 cs/s
; TC1 clock in 250 ms 31.250
.equ cTc1CmpA = 31249
;
; ----- Value averaging -----
; Current value      / 2 plus
; Last value         / 4 plus
; Pre last value    / 8 plus
; Pre pre last value / 8 =
;   Displayed averaged value
;
; ----- Ports, port pins -----
.equ pOut = PORTA ; Output port
.equ pDir = DDRA ; Direction port
.equ bIO = PORTB3 ; Port pin digital pull-up
.equ bID = DDA3 ; Port pin direction pull-up
;
; ----- Registers -----
; Used: R0, R1 for LCD
.def rM0L = R2 ; Current value, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
; free: R5 .. R14
.def rSreg = R15 ; Save/restore SREG
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Another multi purpose register
```

```

.def rLine = R18 ; LCD line counter
.def rRead = R19 ; LCD-Register
.def rImp = R20 ; Multi purpose inside interrupts
.def rFlag = R21 ; Flags
    .equ bTO = 0 ; Time out timer
    .equ bAn = 1 ; Analog comparer active
.def rHelp = R22 ; Additional register decimal
; free: R23 .. R25
; Used: R27:R26 X ; for diverse purposes
; free: R29:R28 Y
; benutzt: R31:R30 Z ; for LCD
;
; ----- SRAM -----
.DSEG
.ORG 0x0060
sMD: ; Digital measured values
.Byte 12
sMDEnd:
sMA: ; Analog measured values
.Byte 12
sMAEnd:
;
; ---- Reset and interrupt vectors -----
.CSEG
.ORG 0x0000
    rjmp Start ; Reset vector, init
    reti ; INT0 External Int 0
    rjmp CntIsr ; PCI Request 0
    reti ; PCINT1 PCI Request 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT TC1 Capture Event
    rjmp Tc1Isr ; TIM1_COMPA TC1 Compare Match A
    reti ; TIM1_COMPB TC1 Compare Match B
    reti ; TIM1_OVF TC1 Overflow
    reti ; TIM0_COMPA TC0 Compare Match A
    reti ; TIM0_COMPB TC0 Compare Match B
    reti ; TIM0_OVF TC0 Overflow
    rjmp CntIsr ; ANA_COMP Analog Comparator
    reti ; ADC ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
CntIsr: ; PCINT0/ANA_COMP, count pulses
    in rSreg,SREG ; Save SREG
    inc rM0L ; Count up
    brne CntIsrRet ; No overflow
    inc rM0M ; Increase MSB
    brne CntIsrRet ; No overflow
    inc rM0H ; Increase HSB
CntIsrRet:
    out SREG,rSreg ; Restore SREG
    reti
;
Tc1Isr: ; TC1 time out interrupt
    ldi rImp,0
    out ACSR,rImp ; Disable Int Comparator
    out GIMSK,rImp ; Disable PCInt
    in rSreg,SREG ; Save SREG
    sbr rFlag,1<<bTO ; Set time out flag
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Main program init -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; to stack pointer
    ; Set 8 Mcs/s controller clock source
    ldi rmp,1<<CLKPCE ; Clock change enable
    out CLKPR,rmp ; in clock prescaler port
    ldi rmp,0 ; Precaler = 1
    out CLKPR,rmp ; to clock prescaler port
    ; Init LCD control port outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; control port outputs

```

```

clr rmp ; Outputs clear
out pLcdCO,rmp ; to Lcd control port
ldi rmp,mLcdDRW ; LCD data port mask write
out pLcdDR,rmp ; to LCD data direction port
; Init digital input port
sbi pOut,bIO ; Input port pull-up
cbi pDir,bID ; Input port is input
; Init LCD
rcall LcdInit ; Call included init routine
ldi ZH,High(2*LcdTextOut) ; Point Z to text
ldi ZL,Low(2*LcdTextOut)
rcall LcdText ; Display text
ldi rmp,0x0C ; Cursor and blink off
rcall LcdC4Byte
; Init timer
ldi rmp,High(cTc1CmpA) ; Compare value MSB
out OCR1AH,rmp ; to MSB timer compare port
ldi rmp,Low(cTc1CmpA) ; dto., LSB
out OCR1AL,rmp ; to timer compare port
clr rmp ; TC1 normal operation
out TCCR1A,rmp
ldi rmp,(1<<CS11)|(1<<CS10) ; Prescaler 64
out TCCRIB,rmp
ldi rmp,1<<OCIE1A ; Enable compare match int
out TIMSK1,rmp
; Deactivate analog comparer
ldi rmp,0 ; Disable interrupt comparator
out ACSR,rmp ; to analog comparer control port
ldi rmp,(1<<ADC2D)|(1<<ADC1D) ; Input pin driver disable
out DIDR0,rmp ; to disable pin port
; Activate PCINT3 digital input
ldi rmp,1<<PCINT3 ; Pin change PA3
out PCMSK0,rmp ; to PCINT0 mask port
ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
out GIMSK,rmp ; to general int mask
; No sleep mode due to analog comparator error!
ldi rmp,0 ; Sleep disable
out MCUCR,rmp ; to master control port
; Enable interrupts
sei ; Set I flag in SREG

Loop:
sbrc rFlag,bTO ; Skip next if time out flag clear
rcall Evaluate ; Flag set, evaluate measured value
rjmp Loop
;
; Evaluate and display measured counting results
Evaluate:
cbr rFlag,1<<bTO ; Clear time out flag
clr rmp ; Disable compare match int
out TIMSK1,rmp ; of TC1
; Shift measured value to SRAM and divide
sbrc rFlag,bAn ; Skip next if digital value
rjmp EvaluateAnalog ; Jump to analog value
ldi ZH,High(sMDEnd) ; Target address
ldi ZL,Low(sMDEnd)
ldi XH,High(sMDEnd-3) ; Source address
ldi XL,Low(sMDEnd-3)
rjmp EvaluateShift ; Shift values in and divide
EvaluateAnalog:
ldi ZH,High(sMAEnd) ; Target address
ldi ZL,Low(sMAEnd)
ldi XH,High(sMAEnd-3) ; Source address
ldi XL,Low(sMAEnd-3)
EvaluateShift:
; Shift measured value to SRAM and divide
ld rmp,-X ; Shift pre-last to pre-pre-last
st -Z,rmp ; Write to target
ld rmp,-X ; Copy from source
st -Z,rmp ; Write to target
ld rmp,-X ; Copy from source
st -Z,rmp ; Write to target
ld rmp,-X ; Shift last to pre-last and divide
lsr rmp ; divide by 2
st -Z,rmp ; Write to target
ld rmp,-X ; Copy from source
ror rmp ; Divide by 2 with carry
st -Z,rmp ; Write to target

```

```

ld rmp,-X ; Copy from source
ror rmp ; Divide by 2 with carry
st -Z,rmp ; Write to target
ld rmp,-X ; Shift latest to last and divide
lsl rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp
ld rmp,-X
ror rmp
st -Z,rmp
st -Z,rM0H ; Store current measurement
st -Z,rM0M
st -Z,rM0L
adiw ZL,3 ; Point to new last
ldi rmp,4 ; Four rounds of adding sum
mov R0,rmp ; R0 is counter
Evaluate1:
    ld rmp,Z+ ; Read LSB
    add rM0L,rmp ; Add to sum
    ld rmp,Z+ ; Read MSB
    adc rM0M,rmp ; Add to sum with carry
    ld rmp,Z+ ; Read HSB
    adc rM0H,rmp ; Add to sum with carry
    dec R0 ; Next round
    brne Evaluate1 ; Go on adding
    ldi ZH,1 ; Point to line 2 in LCD
    sbrc rFlag,bAn ; Skip next if analog flag clear
    ldi ZH,2 ; Point to line 3 in LCD
    ldi ZL,8 ; Point to column 9
    rcall LcdPos ; Set LCD position
    rcall DecimalOut ; Write decimal to LCD
Restart:
    clr rM0L ; Clear last decimal digit
    ldi rmp,1<<bAn ; Invert analog flag
    eor rFlag,rmp
    sbrc rFlag,bAn ; Skip next if analog flag clear
    rjmp RestartAnalog ; Restart an analog comparer cycle
    ; Measure digital, activate PCINT3
    ldi rmp,1<<PCINT3 ; Enable pin change int on PA3
    out PCMSK0,rmp ; to mask port
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
    out GIMSK,rmp ; to int mask
    rjmp Restart1 ; Start timer TC1
RestartAnalog:
    ldi rmp,1<<ACIE ; Enable interrupt analog comparator
    out ACSR,rmp ; in analog comparer control port
Restart1:
    clr rmp ; Clear timer TC1
    out TCNT1H,rmp ; Compare match A port MSB
    out TCNT1L,rmp ; dto., LSB
    ldi rmp,1<<OCIE1A ; Enable compare match interrupt
    out TIMSK1,rmp ; in TC1 mask
    ret
;
; Display 3 byte binary in rM0H:rM0M:rM0L in decimal on LCD
DecimalOut:
    ldi ZH,High(2*DecimalTab) ; Point Z to decimal table
    ldi ZL,Low(2*DecimalTab)
    clt ; Suppress leading zeros
DecimalOut1:
    lpm XL,Z+ ; Read decimal number to X and rHelp
    lpm XH,Z+
    lpm rHelp,Z+
    clr rmp ; Subtraction counter
    cp XL,rmp ; LSB = 0?
    brne DecimalOut2 ; No, go on
    cp XH,rmp ; MSB = 0?
    brne DecimalOut2 ; No, go on
    cp rHelp,rmp ; HSB = 0?
    breq DecimalOutEnd ; Yes, finalize
DecimalOut2:
    sub rM0L,XL ; Subtract decimal from value, LSB
    sbc rM0M,XH ; dto., MSB and carry
    sbc rM0H,rHelp ; dto., HSB and carry
    brcs DecimalOut3 ; Underflow occurred

```

```

inc rmp ; Count subtractions
rjmp DecimalOut2 ; No carry, go on subtracting
DecimalOut3:
    add rM0L,XL ; Take back last subtraction
    adc rM0M,XH
    adc rM0H,rHelp
    tst rmp ; Is digit zero?
    brne DecimalOut4 ; No, not zero
    brts DecimalOut5 ; Zero suppression is off
    ldi rmp,' ' ; Display a blank
    rcall LcdD4Byte
    ldi rmp,' ' ; Thousands separator is a blank
    rjmp DecimalOutKomma ; Check thousands separator
DecimalOut4:
    set ; Do not suppress leading zeros any more
DecimalOut5:
    subi rmp,-'0' ; Convert to ASCII
    rcall LcdD4Byte ; and display on LCD
    ldi rmp,',,' ; Thousands separator
DecimalOutKomma:
    cpi XL,Byte1(1000000) ; Millions?
    breq DecimalOutKommal ; Yes, display comma
    cpi XL,Byte1(1000) ; Thousands?
    breq DecimalOutKommal ; Yes, display comma
    rjmp DecimalOut1 ; No, go on converting
DecimalOutKommal:
    rcall LcdD4Byte ; Comma or blank as thousands separator
    rjmp DecimalOut1 ; Go on converting
DecimalOutEnd:
    ldi rmp,'0' ; Last digit
    add rmp,rM0L ; Add ASCII-0
    rjmp LcdD4Byte ; and display on LCD
;
DecimalTab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(1000000),Byte3(100000)
.db Byte1(100000),Byte2(10000)
.db Byte3(100000),Byte1(1000)
.db Byte2(10000),Byte3(1000)
.db Byte1(1000),Byte2(100)
.db Byte3(1000),Byte1(10)
.db Byte2(100),Byte3(10)
.db 0,0,0
;
; LCD Start text
LcdTextOut:
.db "Frequency meter tn24",0x0D,0xFF
.db "F(dig)= x.xxx.xxx Hz",0x0D,0xFF
;           8
.db "F(ana)= x.xxx.xxx Hz",0x0D,0xFF
;           8
.db " ",0xFE,0xFE
;
; LCD include routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

13.4.4 Experiences and example

The analog input is very sensible, at higher frequencies 2 mV amplitude are enough for a stable count result. At low frequencies the high impedance of the capacitor of 100 nF plays a role so that higher AC voltages are required. If the analog input is open, fast signals on the digital input can stray in.

The same signal, once fed in as well on the analog input (with small amplitude) and on the digital input (with 5 V amplitude) does not always show the same result. The reason for that might be the high noise on the analog input caused by harmonics on the digital input.

Frequenzmesser tn24
 F(dig)= 6.042 Hz
 F(ana)= 6.042 Hz

Frequenzmesser tn24
 F(dig)= 1.973 Hz
 F(ana)= 1.992 Hz

[Home](#) [Top](#) [Frequencies](#) [Decimal conversion](#) [Digital](#) [Analog](#) [induction](#)

13.5 Measuring inductance with PCINT

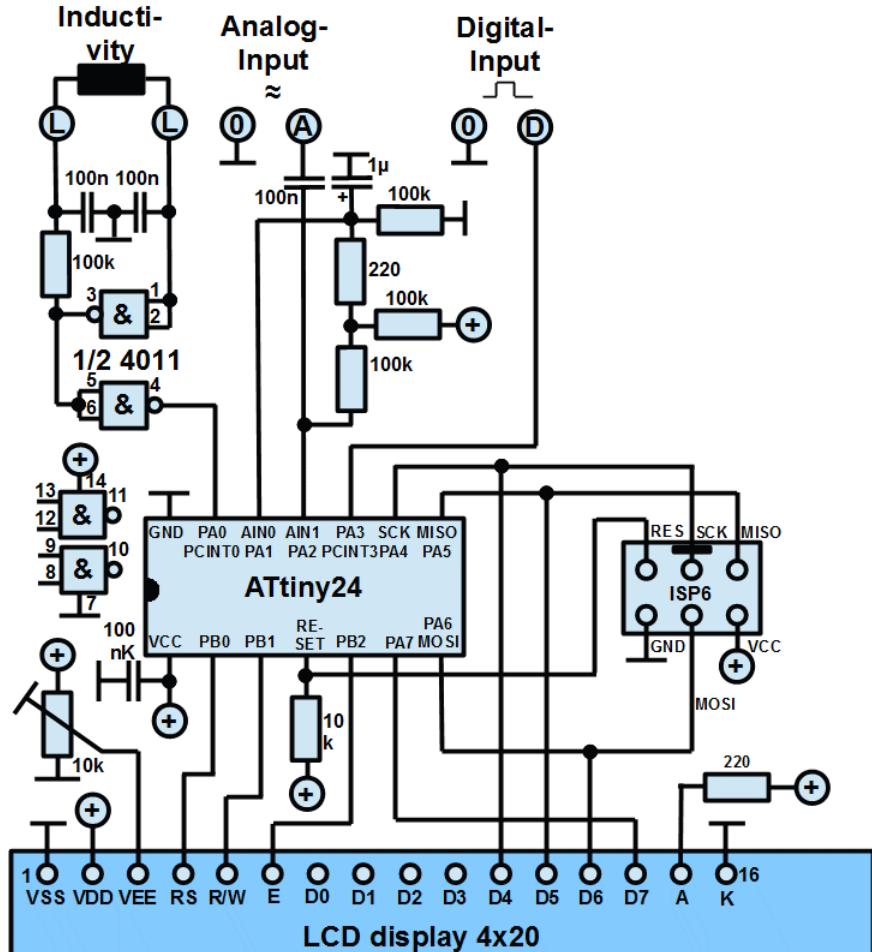
13.5.1 Task

The inductance of coils is to be determined. The measurement shall cover a wide range from 1 mH up to 10 H.

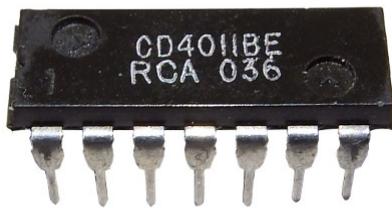
13.5.2 Hardware and components

13.5.2.1 Scheme

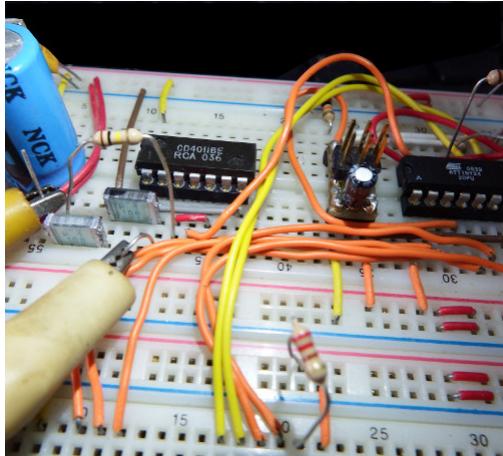
This is the complete schematic to measure digital and analog frequencies as well as the inductance of coils. The inductance measuring works with an oscillator build with a CMOS NAND gate in a 4011, a second gate is used to increase edge slopes. Two gates are not used. The oscillator works stable over a wide range of inductance, much better than a transistorized one or an FET stage. The two capacitors of 100 nF are in series, so that the effective capacity is 50 nF. The feedback is reduced by a 100 k Ω resistor, but this is enough to keep the oscillator swinging over the whole range.



13.5.2.2 Components



This is the quad NAND gate 4011. Any other inverting gates will also work.



13.5.2.3 Mounting

This is the mounting. The coils are attached with crocodile clips.

13.5.3 Program

The following lists the program, the [source code is here](#). Similar to the previous version, the three inputs are measured one after the other. The sleep mode was again not selected because of the incompatibility of the analog comparer with this mode.

```
; ****
; * Digital/analog frequency counter and inductance measurement *
; * (C)2017 by www.avr-asn-tutorial.net *
; ****
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----- Hardware -----
; Digital frequency input on PA3
; and
; Analog frequency input on the analog-
; comparator AIN0/PA1 and AIN1/PA2
; and
; Inductance measurement via frequency
; on input PA0 (4011 LC oscillator)
;
; ----- Timing -----
; Gate time          250 ms
; Controller clock  1,000,000 cs/s (!)
; TC1 prescaler     64
; TC1 clock          15,625 cs/s
; TC1 ticks in 250 ms 3.906
.equ cTc1CmpA = 3905
;
; ----- Value averaging -----
; Current value      / 2 plus
; Last value         / 4 plus
; Pre-last value    / 8 plus
; Pre-pre-last value / 8 =
; displayed value
;
; ----- Ports, port-pins -----
.equ pOut = PORTA ; Output port
.equ pDir = DDRA ; Direction port
.equ bIO = PORTB3 ; Port-pin digital pull-up
.equ bID = DDA3 ; Port-pin digital direction
;
; ----- Registers -----
; Used: R0, R1 for LCD
```

```

.def rM0L = R2 ; Current measuring value, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
.def rMH0 = R5 ; 40 bit help register
.def rMH1 = R6 ; for multiplication
.def rMH2 = R7 ; and division
.def rMH3 = R8 ;
.def rMH4 = R9 ;
.def rMR0 = R10 ; 40 bit result register
.def rMR1 = R11 ; for multiplication
.def rMR2 = R12 ; and division
.def rMR3 = R13 ;
.def rMR4 = R14 ;
.def rSreg = R15 ; Save/restore SREG
.def rmp = R16 ; Multi purpose register
.def rmo = R17 ; Another multi purpose register
.def rLine = R18 ; LCD line counter
.def rRead = R19 ; LCD read register
.def rimp = R20 ; Multi purpose inside interrupts
.def rFlag = R21 ; Flags
    .equ bAn = 0 ; Analog comparer active
    .equ bIp = 1 ; Inductance meter active
    .equ bTO = 2 ; Time out of timer
.def rHelp = R22 ; Help register decimal
; free: R23 .. R25
; Used: R27:R26 X ; for diverse purposes
; free: R29:R28 Y
; Used: R31:R30 Z ; for LCD etc.
;
; ----- SRAM -----
.DSEG
.ORG 0x0060
sMD: ; Digital values, 4*(HSB:MSB:LSB)
.Byte 12
SMDEnd:
sMA: ; Analog values
.Byte 12
sMAEnd:
sMI: ; Inductance values
.Byte 12
sMIEnd:
sdividend: ; for division
(Byte 5
sdividendEnd:
;
; ---- Reset and interrupt vectors -----
.CSEG
.ORG 0x0000
    rjmp Start ; Reset vector, init
    reti ; INTO External Int 0
    rjmp CntIsr ; PCI Request 0
    reti ; PCINT1 PCI Request 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT TC1 Capture Event
    rjmp Tc1Isr ; TIM1_COMPA TC1 Compare Match A
    reti ; TIM1_COMPB TC1 Compare Match B
    reti ; TIM1_OVF TC1 Overflow
    reti ; TIM0_COMPA TC0 Compare Match A
    reti ; TIM0_COMPB TC0 Compare Match B
    reti ; TIM0_OVF TC0 Overflow
    rjmp CntIsr ; ANA_COMP Analog Comparator
    reti ; ADC ADC Conversion Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines -----
CntIsr: ; Count pulses from PCINT0, PCINT3 and ANA_COMP
    in rSreg,SREG ; Save SREG
    inc rM0L ; Count LSB up
    brne CntIsrRet ; No overflow
    inc rM0M ; Increase MSB
    brne CntIsrRet ; No overflow
    inc rM0H ; Increase HSB
CntIsrRet:
    out SREG,rSreg ; Restore SREG
    reti

```

```

;

Tc1Isr: ; Time out timer TC1
    ldi rimp,0
    out ACSR,rimp ; Disable analog comparator int
    out GIMSK,rimp ; Disable PCINT0
    out TIMSK1,rimp ; Disable timer int
    in rSreg,SREG ; Save SREG
    sbr rFlag,1<<bTO ; Set time out flag
    out SREG,rSreg ; Restore SREG
    reti

;
; ----- Main program init -----
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; to stack pointer
    ; Init LCD control port outputs
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Clear LCD control outputs
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; LCD data port output mask write
    out pLcdDR,rmp ; to LCD data port direction
    ; Init digital port
    sbi pOut,bIO ; Digital input pin pull-up
    cbi pDir,bID ; Digital input pin is input
    ; Init LCD
    rcall LcdInit ; Call to included routine
    ldi ZH,High(2*LcdTextOut) ; Point Z to text
    ldi ZL,Low(2*LcdTextOut)
    rcall LcdText ; Display text
    ldi rmp,0x0C ; Cursor and blink off
    rcall LcdC4Byte
    ; Init timer
    ldi rmp,High(cTc1CmpA) ; Compare match A
    out OCR1AH,rmp ; to MSB
    ldi rmp,Low(cTc1CmpA)
    out OCR1AL,rmp ; and to LSB
    clr rmp ; TC1 normal operation
    out TCCR1A,rmp
    ldi rmp,(1<<CS11)|(1<<CS10) ; Prescaler 64
    out TCCR1B,rmp
    ldi rmp,1<<OCIE1A ; Enable compare match int
    out TIMSK1,rmp
    ; Deactivate analog comparator
    ldi rmp,0 ; Disable comparator interrupt
    out ACSR,rmp
    ldi rmp,(1<<ADC2D)|(1<<ADC1D) ; Input pin disable comparator
    out DIDR0,rmp ; to pin disable port
    ; Activate PCINT3
    ldi rmp,1<<PCINT3 ; Pin change on PA3
    out PCMSK0,rmp ; to mask
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
    out GIMSK,rmp ; in int mask
    ; Sleep Mode, no sleep due to analog comparator error
    clr rmp ; Sleep mode disable
    out MCUCR,rmp ; in master control port
    ; Enable interrupts
    sei ; by setting the I flag in SREG

Loop:
    ; Dont sleep
    sbrc rFlag,bTO ; Skip next if time out flag clear
    rcall Evaluate ; Time out flag set, evaluate
    rjmp Loop

;
; Evaluate the measuring results
Evaluate:
    cbr rFlag,1<<bTO ; Clear flag
    clr rmp ; Disable compare match int
    out TIMSK1,rmp ; in TC1
    ; Copy values to SRAM
    cpi rFlag,0x01 ; Inductance values?
    breq EvaluateAnalog ; Go to analog
    brcs EvaluateDigital ; Go to digital
    ; Induktivity value
    ldi ZH,High(sMIEnd) ; Target
    ldi ZL,Low(sMIEnd)

```

```

ldi XH,High(sMIEnd-3) ; Source
ldi XL,Low(sMIEnd-3)
rjmp EvaluateShift
EvaluateDigital:
    ldi ZH,High(sMDEnd) ; Target
    ldi ZL,Low(sMDEnd)
    ldi XH,High(sMDEnd-3) ; Source
    ldi XL,Low(sMDEnd-3)
    rjmp EvaluateShift
EvaluateAnalog:
    ldi ZH,High(sMAEnd) ; Target
    ldi ZL,Low(sMAEnd)
    ldi XH,High(sMAEnd-3) ; Source
    ldi XL,Low(sMAEnd-3)
EvaluateShift:
; Shift measured value to SRAM and divide
    ld rmp,-X ; Copy pre-last to pre-pre-last
    st -Z,rmp
    ld rmp,-X
    st -Z,rmp
    ld rmp,-X
    st -Z,rmp
    ld rmp,-X ; Copy last to pre-last with division
    lsr rmp
    st -Z,rmp
    ld rmp,-X
    ror rmp
    st -Z,rmp
    ld rmp,-X
    ror rmp
    st -Z,rmp
    ld rmp,-X ; Copy latest to pre-last with division
    lsr rmp
    st -Z,rmp
    ld rmp,-X
    ror rmp
    st -Z,rmp
    ld rmp,-X
    ror rmp
    st -Z,rmp
    st -Z,rM0H ; Copy current value
    st -Z,rM0M
    st -Z,rM0L
    adiw ZL,3 ; Point to last value
    ldi rmp,4
    mov R0,rmp ; R0 is counter
Evaluate1:
    ld rmp,Z+ ; Read LSB
    add rM0L,rmp ; Add to current
    ld rmp,Z+ ; Read MSB
    adc rM0M,rmp ; Add to current with carry
    ld rmp,Z+ ; Read HSB
    adc rM0H,rmp ; Add to current with carry
    dec R0 ; Next adder
    brne Evaluate1 ; Go on adding
    cpi rFlag,0x02 ; Inductance?
    brcs Evaluate2 ; No
    rcall Induct ; Calculate and display inductance
    rjmp Restart ; Restart measuring
Evaluate2: ; Display digital or analog measure result
    mov ZH,rFlag ; Position of the digital/analog display
    inc ZH
    ldi ZL,8
    rcall LcdPos
; 24 bit binary in rM0H:rM0M:rM0L to decimal on LCD
Decimal3Out:
    ldi ZH,High(2*Decimal3Tab) ; Point Z to decimal table
    ldi ZL,Low(2*Decimal3Tab)
    clt ; Suppress leading zeros
Decimal3Out1:
    lpm XL,Z+ ; Read decimal number
    lpm XH,Z+
    lpm rHelp,Z+
    clr rmp ; Subtraction counter
    cp XL,rmp ; LSB clear?
    brne Decimal3Out2 ; no
    cp XH,rmp ; MSB clear?

```

```

        brne Decimal3Out2 ; No
        cp rHelp,rmp ; HSB clear
        breq Decimal3OutEnd ; Yes, to end of conversion
Decimal3Out2:
        sub rM0L,XL ; Subtract decimal
        sbc rM0M,XH
        sbc rM0H,rHelp
        brccs Decimal3Out3 ; Overflow, end sub
        inc rmp ; Increase counter
        rjmp Decimal3Out2 ; Go on subtracting
Decimal3Out3:
        add rM0L,XL ; Take back last subtraction
        adc rM0M,XH
        adc rM0H,rHelp
        tst rmp ; Is digit zero?
        brne Decimal3Out4 ; No, not zero
        brts Decimal3Out5 ; Do not suppress zeros
        ldi rmp,' ' ; Suppress leading zero
        rcall LcdD4Byte
        ldi rmp,' ' ; Blank instead of comma
        rjmp Decimal3OutKomma ; Check thousands separator
Decimal3Out4:
        set ; Do not suppress zeros any more
Decimal3Out5:
        subi rmp,-'0' ; Convert to ASCII
        rcall LcdD4Byte ; Write to LCD
        ldi rmp,', ' ; Thousands separator
Decimal3OutKomma:
        cpi XL,Byte1(1000000) ; Millions?
        breq Decimal3OutKommal ; Yes
        cpi XL,Byte1(1000) ; Thousands?
        breq Decimal3OutKommal ; Yes
        rjmp Decimal3Out1 ; No thousands separator
Decimal3OutKommal:
        rcall LcdD4Byte ; Write thousands separator
        rjmp Decimal3Out1
Decimal3OutEnd:
        ldi rmp,'0' ; Last digit
        add rmp,rM0L ; Add ASCII-0
        rcall LcdD4Byte ; To LCD
        rjmp Restart
;
Decimal3Tab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(1000000),Byte3(100000)
.db Byte1(100000),Byte2(10000)
.db Byte3(100000),Byte1(1000)
.db Byte2(10000),Byte3(1000)
.db Byte1(1000),Byte2(100)
.db Byte3(100),Byte1(10)
.db Byte2(10),Byte3(10)
.db 0,0,0,0
;
; Calculate and display inductance
Induct:
        ldi ZH,3 ; Set LCD position
        ldi ZL,6
        rcall LcdPos
        tst rM0M ; MSB zero?
        brne InductN2 ; No
        tst rM0H ; HSB zero?
        brne InductN2 ; No
        mov rmp,rM0L ; LSB frequency to rmp
        cpi rmp,2 ; Frequency 0 or 1?
        brcc InductN1 ; No
        ; F = 0 or 1, clear line and display 0
        ldi XL,10 ; 10 characters
InductN0:
        ldi rmp,' ' ; Clear line
        rcall LcdD4Byte
        dec XL
        brne InductN0 ; Characters left
        ldi rmp,'0' ; Display 0
        rjmp LcdD4Byte
;
InductN1: ; F larger than one

```

```

cpi rmp,23 ; F between 2 and 22?
brcc InductN2 ; No
ldi ZH,High(2*Underflow22) ; Display message
ldi ZL,Low(2*Underflow22)
rjmp LcdTextC
Underflow22:
.db "(F < 23 Hz) ",0xFE,0xFF
InductN2:
    ldi rmp,0x50 ; Result larger than 0x0B5050?
    cp rM0L,rmp
    cpc rM0M,rmp
    ldi rmp,0x0B
    cpc rM0H,rmp
    brcs InductN3 ; No
    ldi ZH,High(2*Overflow) ; Display error message
    ldi ZL,Low(2*Overflow)
    rjmp LcdTextC
Overflow:
.db "(F > Max) ",0xFE,0xFF
InductN3: ; Value within correct range
    ; Multiply rM0H:rM0M:rM0L by itself (F*F)
    mov rM0H,rM0L ; Copy value
    mov rM1,rM0M
    mov rM2,rM0H
    clr rM3 ; Clear upper bytes of value
    clr rM4
    clr rM0 ; Clear result registers
    clr rM1
    clr rM2
    clr rM3
    clr rM4
Induct1: ; Multiply
    lsr rM0H ; Shift lowest bit to carry
    ror rM0M
    ror rM0L
    brcc Induct2 ; Carry is clear, do not add
    add rM0,rM0H ; Add to result
    adc rM1,rM1
    adc rM2,rM2
    adc rM3,rM3
    adc rM4,rM4
Induct2: ; All three bytes clear
    tst rM0L
    brne Induct3 ; No
    tst rM0M
    brne Induct3 ; No
    tst rM0H
    breq Induct4 ; Yes, done
Induct3:
    lsl rM0H ; Multiply by 2
    rol rM1
    rol rM2
    rol rM3
    rol rM4
    rjmp Induct1 ; Continue multiplication
Induct4:
    ; Division, load dividend to SRAM
    ldi ZH,High(2*dividendtable) ; Z points to dividend table
    ldi ZL,Low(2*dividendtable)
    ldi XH,High(sdividend) ; X points to SRAM dividend
    ldi XL,Low(sdividend)
Induct5:
    lpm rmp,Z+ ; Read dividend from table
    st X+,rmp ; Store in SRAM
    cpi XL,Low(sdividendEnd) ; All read?
    brcs Induct5 ; No, go on
    ; Clear dividend help registers
    clr rM0H
    clr rM1
    clr rM2
    clr rM3
    clr rM4
    ; Clear result registers
    clr rM0L
    clr rM0M
    clr rM0H
    clr ZL

```

```

clr ZH
; Shift dividend in SRAM right
ldi rmp,8*(sdividendEnd-sdividend)+1
mov R0,rmp ; R0 is outer counter
Induct6:
    ; Divide
    ldi XH,High(sdividend) ; Point X to SRAM
    ldi XL,Low(sdividend)
    ldi rmp,sdividendEnd-sdividend ; Number of bytes
    mov R1,rmp ; R1 is inner counter
    clc ; Clear carry
Induct7:
    ld rmp,X ; Read byte from SRAM
    rol rmp ; Roll highest bit to carry
    st X+,rmp ; Store multiplied byte in SRAM
    dec R1 ; Count down inner loop
    brne Induct7 ; Go on shifting
    ; Shift carry into help register
    rol rMH0
    rol rMH1
    rol rMH2
    rol rMH3
    rol rMH4
    sub rMH0,rMR0 ; Subtract divider
    sbc rMH1,rMR1
    sbc rMH2,rMR2
    sbc rMH3,rMR3
    sbc rMH4,rMR4
    brcc Induct8 ; Subtraction no carry
    add rMH0,rMR0 ; Carry, take back subtraction
    adc rMH1,rMR1
    adc rMH2,rMR2
    adc rMH3,rMR3
    adc rMH4,rMR4
    clc ; Clear carry
    rjmp Induct9 ; Shift carry into result
Induct8:
    sec ; Set carry
Induct9:
    dec R0 ; Decrease outer counter
    breq Induct10 ; Division done
    rol rM0L ; Roll carry into result
    rol rM0M
    rol rM0H
    rol ZL
    rol ZH
    rjmp Induct6 ; Go on dividing
    ; Round result
Induct10:
    ldi rmp,0 ; Add zero
    adc rM0L,rmp ; with carry
    adc rM0M,rmp ; with carry
    adc rM0H,rmp ; with carry
    adc ZL,rmp ; with carry
    adc ZH,rmp ; with carry
    mov rMH0,ZL ; Copy highest two byte to help register
    mov rMH1,ZH

;
; 32 bit binary in rM0H:rM0H:rM0M:rM0L to decimal on LCD
Decimal4Out:
    ldi ZH,High(2*Decimal4Tab) ; Point Z to decimal tab
    ldi ZL,Low(2*Decimal4Tab)
    clt ; Suppress leading zeros
Decimal4Out1:
    lpm rMR0,Z+ ; Read decimal number
    lpm rMR1,Z+
    lpm rMR2,Z+
    lpm rMR3,Z+
    clr rmp
    or rmp,rMR0 ; End of decimal table?
    or rmp,rMR1
    or rmp,rMR2
    or rmp,rMR3
    breq Decimal4OutEnd ; Yes, finish
    clr rmp ; Subtraction counter
Decimal4Out2:
    sub rM0L,rMR0 ; Subtract decimal

```

```

    sbc rM0M,rMR1
    sbc rM0H,rMR2
    sbc rMH0,rMR3
    brcs Decimal4Out3 ; Carry set, end of subtract
    inc rmp ; Increment result
    rjmp Decimal4Out2 ; Go on subtraction
Decimal4Out3:
    add rM0L,rMR0 ; Take back subtraction
    adc rM0M,rMR1
    adc rM0H,rMR2
    adc rMH0,rMR3
    tst rmp ; Digit is zero?
    brne Decimal4Out4 ; No
    brts Decimal4Out5 ; Do not suppress zeros
    ldi rmp,' ' ; Suppress leading zero
    rcall LCD4Byte
    ldi rmp,' ' ; Blank instead of thousands separator
    rjmp Decimal4Out6 ; Check thousands separator
Decimal4Out4:
    set ; Do not suppress leading zeros any more
Decimal4Out5:
    subi rmp,-'0' ; Add ASCII-0
    rcall LCD4Byte ; and display
    ldi rmp,',,' ; Thousands separator
Decimal4Out6:
    cpi ZL,Low(2*Decimal4Tab1Mio) ; One million?
    breq Decimal4Out7 ; Yes
    cpi ZL,Low(2*Decimal4Tab1000) ; One thousand?
    brne Decimal4Out1 ; No
Decimal4Out7:
    rcall LCD4Byte ; Display thousands separator
    rjmp Decimal4Out1 ; Go on converting
Decimal4OutEnd:
    ldi rmp,'0' ; Last digit
    add rmp,rM0L ; Add ASCII-0
    rcall LCD4Byte ; to LCD
    rjmp Restart ; Restart counting
;
dividendtable:
.db 0x7F,0xD7,0x10,0xF4,0x75,0x00
;
Decimal4Tab:
.dw LWRD(100000000),HWRD(100000000)
.dw LWRD(10000000),HWRD(10000000)
.dw LWRD(1000000),HWRD(1000000)
Decimal4Tab1Mio:
.dw LWRD(100000),HWRD(100000)
.dw LWRD(10000),HWRD(10000)
.dw LWRD(1000),HWRD(1000)
Decimal4Tab1000:
.dw LWRD(100),HWRD(100)
.dw LWRD(10),HWRD(10)
.dw 0,0
;
Restart:
    clr rM0L ; Clear last result
    clr rM0M
    clr rM0H
    inc rFlag ; Next measure mode
    cpi rFlag,3 ; Mode > 2?
    brcs Restart1 ; No, go on
    clr rFlag ; Start first mode
Restart1:
    ; rFlag=0:digital, =1:analog, =2:L
    cpi rFlag,0x01 ; Mode 1?
    brcs RestartDigital ; Mode 0, digital
    breq RestartAnalog ; Mode 1, analog
    ; Restart inductance
    ldi rmp,1<<PCINT0 ; Pin Change PA0
    rjmp RestartPcInt ; Start PCINT
    ; Digital, activate PCINT3
RestartDigital:
    ldi rmp,1<<PCINT3 ; Pin change PA3
RestartPcInt:
    out PCMSK0,rmp ; To PCINT0 mask
    ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupt
    out GIMSK,rmp ; in int mask

```

```

    rjmp Restart2
RestartAnalog:
    clr rmp ; Disable PCINT int
    out GIMSK,rmp
    ldi rmp,1<<ACIE ; Enable analog comp int
    out ACSR,rmp ; in analog comparator control port
Restart2:
    ldi rmp,1<<TSM ; Prescaler Sync Mode
    out GTCCR,rmp ; to control port
    ldi rmp,(1<<TSM)|(1<<PSR10) ; Reset Prescaler 1
    out GTCCR,rmp ; in control port
    clr rmp ; Clear reset prescaler
    out GTCCR,rmp ; in prescaler count mode port
    out TCNT1H,rmp ; Clear 16 bit counter TC1
    out TCNT1L,rmp
    ldi rmp,1<<OCIE1A ; Enable TC1 compare A int
    out TIMSK1,rmp ; in timer int mask
    ret
;
; LCD Start text
LcdTextOut:
.db "Frequency meter tn24",0x0D,0xFF
.db "F(dig)= x.xxx.xxx Hz",0x0D,0xFF
;
     8
.db "F(ana)= x.xxx.xxx Hz",0x0D,0xFF
;
     8
.db "L =   xxx.xxx.xxx ",0xE4,"H",0xFE,0xFE
;
     6
; LCD include routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

13.5.4 Simulating program execution

Program simulation can be made with [avr_sim](#). All calls to the LCD should be replaced by write operations to the SRAM to view the results. Register pair Y can be used for those write operations.

Useful simulation can be made for the 24-bit-decimal conversion and for the inductance calculation.

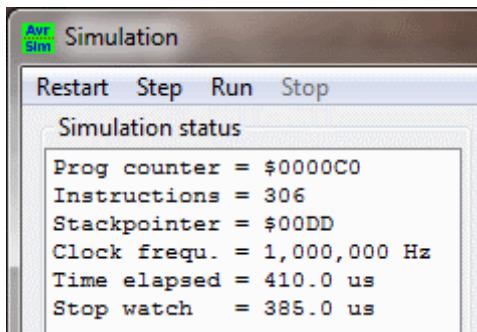
13.5.4.1 Simulation of 24 bit binary to decimal conversion

Register					
Reg	+0	+1	+2	+3	+4
R0	00	00	87	D6	12
R8	00	00	00	00	00
R16	87	00	00	00	00
R24	00	00	00	00	60

For simulating the decimal conversion, the binary number 1.234.567 is written to rM0H:rM0M:rM0L. Then the conversion routine is called.

SRAM																	
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	31	2C	32	33	34	2C	35	36	37	FF	1.234,567..						
\$0070	FF															

The conversion result, as it goes to the LCD, is correct.



The time required for this conversion is well below 1 ms.

13.5.4.2 Simulation of the induction calculation

Register								
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	E8	03	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	E8	00	00	00	00	00	00	00
R24	00	00	00	00	90	00	00	00

First we simulate a measured frequency of 1,000 Hz. The value of 1,000 is loaded to the registers R4:R3:R2.

SRAM																	ASCII text
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	FF															
\$0070	FF															
\$0080	FF	FF	FF	FF	00	00	00	00	FF							
\$0090	20	20	20	20	35	30	36	2C	36	30	36	FF	FF	FF	FF	FF	506,606.....

The result of 506.606 mH is correct, as has been shown in the above example case.

3.6 ms have been elapsed. Division of large numbers is a little bit more time consuming than conversion.

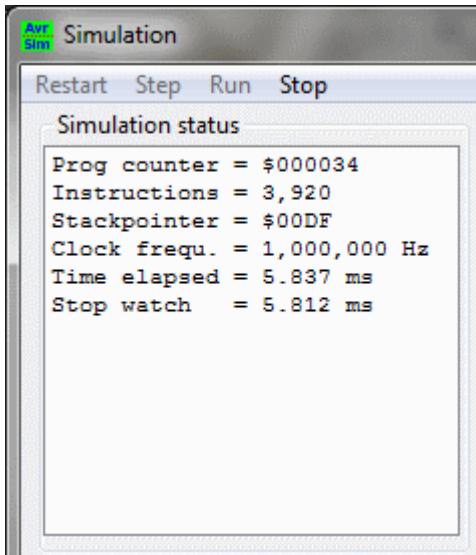
SRAM																	ASCII text
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	FF															
\$0070	FF															
\$0080	FF	FF	FF	FF	00	00	00	00	FF							
\$0090	20	20	20	20	35	30	36	2C	36	30	36	FF	FF	FF	FF	FF	5,066.....

This is the result when 10,000 Hz would be generated by the LC oscillator, 5.06 mH.

SRAM																	ASCII text
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	FF															
\$0070	FF															
\$0080	FF	FF	FF	FF	00	00	00	00	FF							
\$0090	20	20	20	20	20	20	35	2C	30	36	36	FF	FF	FF	FF	FF	5,066.....

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	FF															
\$0070	FF															
\$0080	FF	FF	FF	FF	00	00	00	00	FF							
\$0090	32	30	32	2C	36	34	32	2C	33	36	37	FF	FF	FF	FF	202,642,367

Simulation of 50 Hz oscillator frequency results in an inductance of 202 H.



Calculation time at 50 Hz lasts a little bit longer, but not too long.

Simulation helps in developing and debugging by letting you check even complex programs. It helps to clearly define subroutines for which entry values (in registers, in SRAM) and resulting changes can be clearly tracked.

13.5.4 Examples

During the following examples the digital and the analog input were open, the displayed values in line 2 and 3 of the LCD are capacitive stray signals.

13.5.4.1 Large coil

This is the measurement of a relative large coil.



```

Frequenzmesser tn24
F(dig)=      281 Hz
F(ana)=      748 Hz
L =      3.660.871 µH

```

The measurement is near the inductance that was determined with a different method. Again the displayed inductance is slightly too large, the measured frequency is a little bit too low, caused by the limited accuracy of the internal RC oscillator.

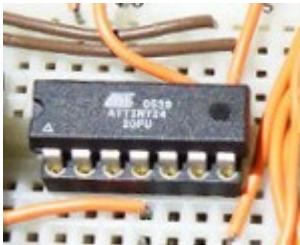
13.5.4.2 Speaker coil



The coil in the speaker that we used in one of the previous lectures has an inductance, too.

Frequenzmesser tn24
F(dig)= 15.473 Hz
F(ana)= 43.591 Hz
L = 783 μ H

The inductance of the speaker's coil is rather small and around 800 μ H.



Lecture 14: Voltage, current and temperature meter

14.0 Overview

1. [Measuring, calculate and display of voltages](#)
2. [Measuring, calculation and display of currents](#)
3. [Measuring, calculation and display of temperatures](#)

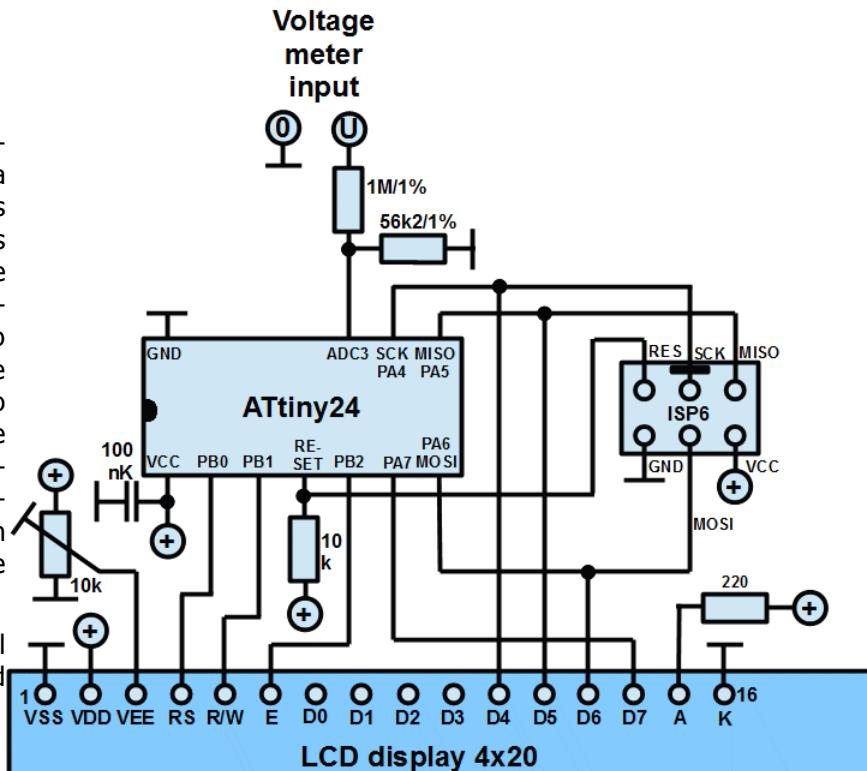
14.1 Measuring, calculate and display of voltages

14.1.1 Hardware

14.1.1.1 Scheme

The components for measuring voltages are minimal: a voltage divider that reduces the input voltage and feeds the divided voltage to the ADC3 input pin. With this configuration voltages of up to 20.5 V can be measured. The resistor 56k2 is selected to ensure a short sampling time of the ADC: if the feed resistance is too large, the sampling time is prolonged from 1.5 to two clock cycles of the ADC.

Of course, the experimental board [here](#) can also be used instead.



14.1.1.2 Components



To the left is the resistor of $56.2 \text{ k}\Omega$, to the right the $1 \text{ M}\Omega$. If you use other values, you will have to change the constants on top of the source code. With that change you can use any resistor combination.



14.1.2 Measuring range, measure/calculation/display issues

14.1.2.1 Measuring range

The ADC is programmed to use the internally generated reference voltage of 1.1 V, to be independent from the operating voltage of the ATTiny13. The voltage on the ADC3 pin is

$$U_{ADC3} = U_{Input} * 56.2 / (1000 + 56.2) = 1.1 [V].$$

The maximum measurable voltage therefore is

$$U_{Input} = 1.1 * (1000 + 56.2) / 56.2 = 20.673 [V].$$

Per ADC bit the resolution is 0.02 V.

14.1.2.2 Measuring conditions

The software is built on the following conditions:

- The source pin for the ADC is ADC3, which is written to the MUX port of the ATTiny24. As no other channels have to be measured in this version, this does not change.
- As reference for the ADC the internal 1.1 V is used, the results are not depending from the operating voltage.
- The measurements are averaged over 64 single measurements. This results in a measurement/display frequency of 119 cs/s.

14.1.2.3 Calculations

In the conversion of the ADC results to the displayed voltage the following parameters play a role:

- the relation of the two resistors, with $Rel = (1M+56k2)/56k2$,
- at a reference voltage of 1.1 V the 10 bit ADC delivers 1,023, so the measurement result is $N_{ADC} = U_{pin} * 1024 / 1.1$,
- by summing up 64 measurements the ADC yields $64 * N_{ADC}$ as sum value,
- to display the voltage a resolution of 0.01 V makes sense because this is the ADC resolution of the 10 bit ADC.

From that the following formula for the display applies:

$$U(0.01V) = (1M+56k2)/56k2 * 1.1 / 1024 * 100 / 64 * N_{Sum-ADC}$$

The first parameters, by which the measured sum has to be multiplied is

$$U(0.01V) = 0.03154442 * N_{Sum-ADC}$$

To be independent from a floating point math lib, we multiply 0.0315442 with 65,536 and skip the last two bytes of the result (dividing by 65,536). So we come to

$$U(0.01V) = 2067 * N_{Sum-ADC} / 65536$$

The calculation therefore is a 16-by-16 bit multiplication, with a 32 bit result.

By changing this constant we can accommodate to any other divider relation.

14.1.2.4 Display

The result of the multiplication and the dividing by 65,536 yields values at maximum slightly above 2,000. For converting the binary to ASCII it is sufficient to start with thousands. Suppressing leading zeros shall only apply to 1,000s, to make sure that at least one digit before

the decimal point is displayed. The decimal point is to be displayed after the hundreds.

14.1.3 Program

The program is rather straight forward, the [source code is here](#). To assemble requires [the LCD include file](#).

```
;  
; *****  
; * Measuring voltages with an ATtiny24 *  
; * (C)2017 by www.avr-asm-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Switches -----  
.equ debug = 0 ; Debug switch for Studio simulation  
.equ debugN = 1000 ; Parameter for Studio simulation  
;  
; ----- Hardware -----  
; ATtiny24  
;      1 / |14  
;      VCC o--|VCC  GND|--o GND  
;      2 |    |13  
;      LCD-RS o--|PB0    |--o NC  
;      3 |    |12  
;      LCD-RW o--|PB1    |--o NC  
;      4 |    |11  
;      VCC-10k o--|RES    |--o NC  
;      5 |    |10  1 M to Input  
;      LCD-E o--|PB2 ADC3|--o 56k2 to GND  
;      6 |    |9  
;      LCD-D7 o--|PA7 PA4|--o LCD-D4  
;      7 |    |8  
;      LCD-D6 o--|PA6 PA5|--o LCD-D5  
;      | _____ |  
;  
; ----- Timing -----  
; Controller clock 1.000.000 cs/s  
; ADC prescaler      128  
; ADC clock frequency 7.812.5 cs/s  
; ADC clocks per convers. 14.5  
; Measure frequency      538.8 cs/s  
; Measure cycle (64)      8.42 cs/s  
;  
; ----- Constants -----  
.equ cRIn = 1000000 ; Ohms  
.equ cRGnd = 56200 ; Ohms  
.equ cMult = ((cRIn+cRGnd)*110+cRGnd/2)/cRGnd  
;  
; ----- Registers -----  
; Used: R0 by LCD  
; free: R1 .. R4  
.def rAL= R5 ; ADC result adder, LSB  
.def rAH = R6 ; dto., MSB  
.def rM0 = R7 ; 32 bit multiplicator  
.def rM1 = R8  
.def rM2 = R9  
.def rM3 = R10  
.def rR0 = R11 ; 32 bit result  
.def rR1 = R12  
.def rR2 = R13  
.def rR3 = R14  
.def rSreg = R15 ; Save/restore SREG  
.def rmp = R16 ; Multi purpose register  
.def rimp = R17 ; Multi purpose inside ints  
.def rFlag = R18 ; Flag register  
    .equ bRdy = 0 ; Ready flag  
.def rCtr = R19 ; Counter measurements  
.def rRead = R20 ; LCD read register  
.def rLine = R21 ; LCD line counter  
.def rmo = R22 ; LCD multi purpose
```

```

; free: R23 .. R29
; Used: Z = R31:R30 for LCD
;
; ----- Reset and interrupt vectors
rjmp Start ; RESET vector
reti ; INT0 External Int Request 0
reti ; PCINT0 Pin Change Int Request 0
reti ; PCINT1 Pin Change Int Request 1
reti ; WDT Watchdog Time-out
reti ; TIM1_CAPT TC1 Capture Event
reti ; TIM1_COMPA TC1 Compare Match A
reti ; TIM1_COMPB TC1 Compare Match B
reti ; TIM1_OVF TC1 Overflow
reti ; TIM0_COMPA TC0 Compare Match A
reti ; TIM0_COMPB TC0 Compare Match B
reti ; TIM0_OVF TC0 Overflow
reti ; ANA_COMP Analog Comparator
rjmp AdcIsr ; ADC ADC Conv Complete
reti ; EE_RDY EEPROM Ready
reti ; USI_STR USI START
reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines --
AdcIsr: ; ADC conversion complete
    in rSreg,SREG ; Save SREG
    in rimp,ADCL ; Read LSB
    add rAL,rimp ; Add to sum
    in rimp,ADCH ; Read MSB
    adc rAH,rimp ; Add to result plus carry
    dec rCtC ; Counter cycles
    brne AdcIsrRet ; Not zero
    sbr rFlag,1<<bRdy ; Set flag
    ldi rCtr,64 ; Restart counter
    mov rM0,rAL ; Copy sum
    mov rM1,rAH
    clr rAL ; Clear sum
    clr rAH
AdcIsrRet:
    ; Start next conversion
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; to ADC control port
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Reset vector, program start
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; Write to stack pointer
.if debug == 1 ; Debugging session
    ldi rmp,Low(debugN*64) ; Simulate N
    mov rM0,rmp
    ldi rmp,High(debugN*64)
    mov rM1,rmp
    rjmp Display
.else
    ; Init LCD
    ; Init LCD control port
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Outputs clear
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; LCD data port mask write
    out pLcdDR,rmp ; to LCD direction port
    ; Init LCD
    rcall LcdInit ; Start LCD
    ldi ZH,High(2*LcdTextOut) ; Z to text
    ldi ZL,Low(2*LcdTextOut)
    rcall LcdText ; Display text
    ldi rmp,0x0C ; Cursor and blink off
    rcall LcdC4Byte
    ; Start ADC
    ldi rmp,(1<<REFS1) | (1<<MUX1) | (1<<MUX0) ; Int.Ref, channel 3
    out ADMUX,rmp ; to ADC mux port
    clr rAL ; Clear result sum
    clr rAH
    ldi rCtr,64 ; 64 measure cycles

```

```

ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; Start ADC
.endif
; Sleep mode
ldi rmp,1<<SE ; Sleep Mode Idle
out MCUCR,rmp
; Enable interrupts
sei ; Set int flag
;
; Main program loop
Loop:
    sleep ; Go to sleep
    nop ; Wake up
    sbrc rFlag,bRdy ; Skip next if ready flag clear
    rcall Display ; Process ready flag
    rjmp Loop
;
; Calculate result and display
Display:
    cbr rFlag,1<<bRdy ; Clear ready flag
    ; Multiply with constant cMult
    clr rM2 ; Clear bytes 3 and 4
    clr rM3
    clr rR0 ; Clear result
    clr rR1
    clr rR2
    clr rR3
    ldi ZH,High(cMult) ; Load constant
    ldi ZL,Low(cMult)
Display1:
    lsr ZH ; Shift bit from MSB to LSB
    ror ZL ; Shift bit from LSB to carry
    brcc Display2 ; Bit was clear
    add rR0,rM0 ; Add multiplicator to result
    adc rR1,rM1
    adc rR2,rM2
    adc rR3,rM3
Display2:
    mov rmp,ZL ; Copy LSB
    or rmp,ZH ; Or MSB
    breq Display3 ; No more bits to multiply
    lsl rM0 ; Multiplicator * 2
    rol rM1
    rol rM2
    rol rM3
    rjmp Display1 ; Multiply further
Display3:
    ; Round result in rR3:rR2
    ldi rmp,0x7F ; Add 0x7F
    add rR0,rmp ; to lowest byte
    adc rR1,rmp ; and to second byte with carry
    ldi rmp,0 ; Carry adder
    adc rR2,rmp ; Add carry to third byte
    adc rR3,rmp ; Add carry to fourth byte
    ; Display on LCD
    ldi ZH,1 ; LCD position
    ldi ZL,4
    rcall LcdPos
    clt ; Suppress leading zeros
    ldi ZH,High(2*DecimalTab)
    ldi ZL,Low(2*DecimalTab)
Display4:
    lpm rR0,Z+ ; Read decimal value
    lpm rR1,Z+
    tst rR0 ; LSB zero?
    breq Display8 ; Finished
    clr rmp ; rmp is digit counter
Display5:
    ; Subtract decimal
    sub rR2,rR0
    sbc rR3,rR1
    brcc Display6 ; Carry, end of subtraction
    inc rmp ; Increase digit count
    rjmp Display5 ; Go on subtracting
Display6:
    add rR2,rR0 ; Take back last subtraction
    adc rR3,rR1

```

```

tst rmp ; Digit=0?
brne Display7 ; No, display
brts Display7 ; No suppression of leading zeros
ldi rmp,' ' ; Replace leading zero by blank
rcall LcdD4Byte
set ; Do not suppress leading zeros any more
rjmp Display4 ; Go on displaying digits
Display7:
set ; Do not suppress leading zeros any more
subi rmp,-'0' ; Convert to ASCII
rcall LcdD4Byte
cpi ZL,LOW(2*DecimalTab10) ; Decimal point?
brne Display4 ; No
ldi rmp,'.' ; Display decimal point
rcall LcdD4Byte
rjmp Display4
Display8:
ldi rmp,'0' ; Last digit
add rmp,rR3 ; Add ASCII-0
rjmp LcdD4Byte
;
; Decimal table thousands and below
DecimalTab:
.dw 1000
.dw 100
DecimalTab10:
.dw 10
.dw 0
;
; ----- LCD routines -----
LcdTextOut:
.db " Voltage measuring ",0x0D,0xFF
.db "U = xx.xx V",0xFE
;
; Include LCD routines
.include "Lcd4Busy.inc"
;
;
; End of source code
;

```

New in this code is SUB register,register. This subtracts the content of the second register from the first one and stores the result in the first register.

14.1.4 Example

Spannungsmessung
U = 5,10 V

The internal 1.1 V reference makes it possible:
the ATtiny24 measures its own operating voltage
(German version).

[Home](#) [Top](#) [Voltage](#) [Current](#) [Temperature](#)

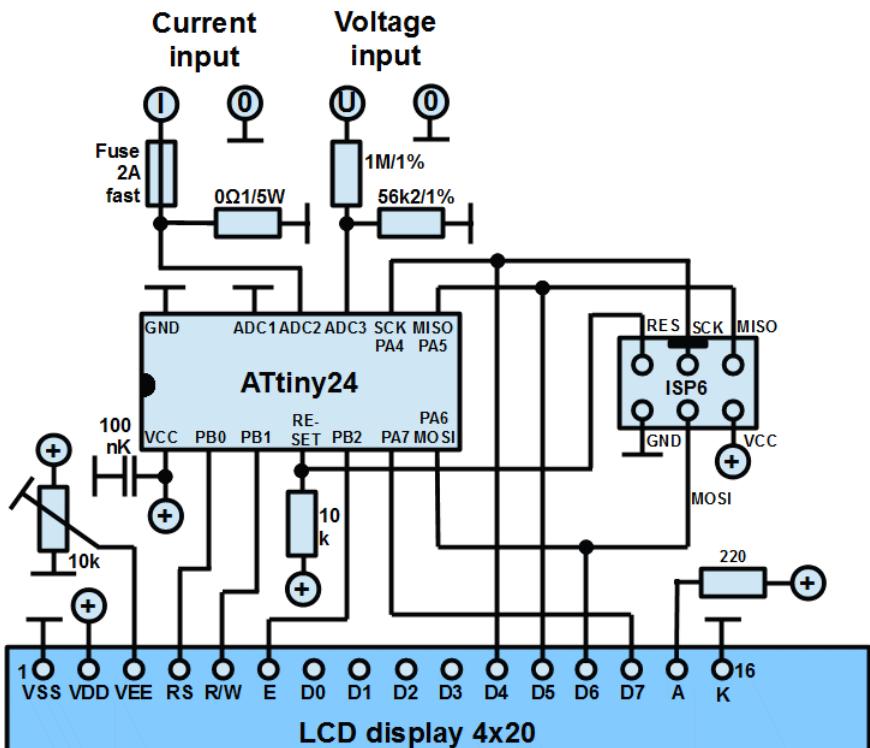
14.2 Measuring, calculation and display of currents

When measuring currents it is crucial to have a as-low-as-possible input resistance to avoid distortions by the measurement. We here use a very special feature of the ADC in newer tiny devices that helps us with that.

14.2.1 Hardware

14.2.1.1 Scheme

The current is measured via the voltage drop over the resistor of 0.1Ω . The fuse that is in series with that resistor limits it to 2 A and protects the resistor and the controller input pin ADC2 against short circuiting. The input ADC1 serves as differential input and is grounded. Measured is the differential voltage between ADC2 and ADC1, and this difference is gained by a factor of 20.

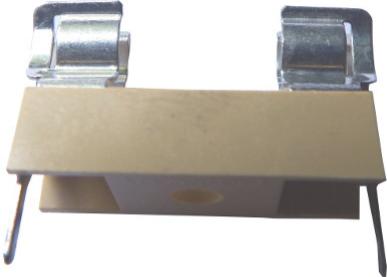


14.2.1.2 Components



This here is a usual 0.1Ω resistor. This wire resistor is specified for 5 Watts thermal power, while we would only need $P = U * I = (I^2 * R) * I = 4 * 0.1 = 0.4 \text{ W}$ or the next higher rated (0.5 W). Unfortunately those resistors are not sold, so we take what we get.

That is how the 2 A fuse looks like.



And this is a respective fuse holder. To fit to our breadboard we need to solder two short wires to it.

To a fuse holder belongs a fuse holder cap. This is not necessary here because we only measure low voltage DC. So see it rather as dust preventer.

14.2.2 Measuring range, measure/calculate/display issues

14.2.2.1 Measuring range

At full swing the ADC input reaches the reference voltage of 1.1 V, divided by the differential gain of 20, which is 0.055 V. With a resistor of 0.1Ω this corresponds to a current of $I = U / R = 0.055 / 0.1 = 0.55 \text{ A}$ or 550 mA. Per ADC bit this is a resolution of 0.53 mA. A display reso-

lution of 0.1 mA would be sufficient.

If we would not select a gain of 20 but of 1 (normal ADC input), our full range covers currents of up to $I = U / R = 1.1 / 0.1 = 11$ A. To cover that whole range our resistor should then have a power of $P = I^2 * R = 11 * 11 * 0.1 = 12.1$ W. Per ADC digit a resolution of around 11 mA would apply. The display should then have a resolution of 0.01 A. And, of course, we need a higher-rated fuse.

14.2.2.2 Measuring

Measuring differential voltages involves just sending a different bit combination to the ADMUX port. The device data-book for the ATtiny24 says which MUX bits can be used and lists all those combinations.

When measuring currents, we sum up 64 single values, just like in the previous chapter.

14.2.2.3 Calculation

The measured voltage on the ADC2 input is

$$U_{\text{meas}} [\text{V}] = R [\Omega] * I [\text{A}].$$

From that

$$I [\text{A}] = U_{\text{meas}} [\text{V}] / R [\Omega].$$

With a differential gain of 20

$$I [\text{A}] = 20 * U_{\text{meas}} [\text{V}] / R [\Omega].$$

With a reference voltage of 1.1 V we see

$$N_{\text{meas}} = 20 * U_{\text{meas}} * 1024 / 1.1.$$

Also is

$$20 * U_{\text{meas}} = N_{\text{meas}} * 1.1 / 1024$$

and therefore

$$I [\text{A}] = N_{\text{mess}} * 1.1 / 1024 / R.$$

For 64 summed up measurements

$$I [\text{A}] = N_{\text{Sum-meas}} * 1.1 / 1024 / R / 64 / 20.$$

For a resolution of 0.0001 A the integer value would be

$$I [0.0001 \text{ A}] = 10000 * N_{\text{Sum-meas}} * 1.1 / 1024 / R / 64 / 20$$

or $0.0083923 * N_{\text{Sum-meas}} / R$. Multiplied by 65,536 a multiplication factor of 550 results. With 0.1Ω a multiplication factor of 5,500 results. The whole formula then is

$$I [0.0001 \text{ A}] = 5,500 * N_{\text{Sum-meas}} / 65536.$$

Again we see: a little algebra and brain effort replaces mighty floating point math libraries that do not fit into the ATtiny24's flash memory.

14.2.2.4 Display

The measured sum is a 16 bit binary, which is to be multiplied with the 16 bit binary 5,500. The lower two bytes of the result can be used to round the result in the upper two bytes (division by 65,536). For conversion into the display format of 123.4 mA first the thousands, then the hundreds and the tens are calculated. Following the hundreds no suppression of leading

zeros is needed any more, prior to the ones a decimal point is to be displayed.

An alternative display format would be 1.234(5) A. That would require some changes to the source code, but is possible and simple.

14.2.3 Program

The program is listed here, the [source code is here](#). For assembling the [LCD include routines are required](#).

```
;  
; *****  
; * To measure and display voltage and current *  
; * (C)2017 by www.avr-asm-tutorial.net *  
; *****  
;  
.NOLIST  
.INCLUDE "tn24def.inc"  
.LIST  
;  
; ----- Hardware -----  
; ATtiny24  
;      1 /_____|14  
;      VCC o--|VCC  GND|--o GND  
;      2 |       |13  
; LCD-RS o--|PB0    |--o NC  
;      3 |       |12  
; LCD-RW o--|PB1 ADC1|--o GND  
;      4 |       |11  
; VCC-10k o--|RES ADC2|--o 0.1 Ohms  
;      5 |       |10  1 M to input  
; LCD-E o--|PB2 ADC3|--o 56k2 to GND  
;      6 |       |9  
; LCD-D7 o--|PA7   PA4|--o LCD-D4  
;      7 |       |8  
; LCD-D6 o--|PA6   PA5|--o LCD-D5  
;      |       |  
;  
; ----- Timing -----  
; Processor clock  1,000,000 cs/s  
; ADC prescaler      128  
; ADC clock          7,812.5 cs/s  
; ADC clocks per conv. 14.5  
; Measuring frequency 538.8 cs/s  
; Measure cycle, 64 m. 8.42 cs/s  
; Two measure cycles 4.21 cs/s  
;  
; ----- Constants -----  
; Voltage measurement  
.equ cRInp = 1000000 ; Ohm  
.equ cRGnd = 56200 ; Ohm  
.equ cMultU = ((cRInp+cRGnd)*110+cRGnd/2)/cRGnd  
; Current measurement  
.equ cRI = 100 ; Resistor in milli ohm  
.equ cMulti = (550000+cRI/2) / cRI  
; ADC-MUX-Constants  
.equ cMuxU = (1<<REFS1)|(1<<MUX1)|(1<<MUX0) ; Voltage input  
.equ cMuxI = (1<<REFS1)|0b101101 ; Current input, diff.gain=20  
;  
; ----- Registers -----  
; Used: R0 by LCD  
; free: R1 .. R4  
.def rAL= R5 ; ADC sum LSB  
.def rAH = R6 ; ADC sum MSB  
.def rM0 = R7 ; 32 bit multiplier  
.def rM1 = R8  
.def rM2 = R9  
.def rM3 = R10  
.def rR0 = R11 ; 32 bit result  
.def rR1 = R12  
.def rR2 = R13  
.def rR3 = R14  
.def rSreg = R15 ; Save/restore SREG  
.def rmp = R16 ; Multi purpose register
```

```

.def rimp = R17 ; Multi purpose inside interrupts
.def rFlag = R18 ; Flag register
    .equ bRdyV = 0 ; Ready flag voltage
    .equ bRdyC = 1 ; Ready flag current
.def rCtr = R19 ; Counter measurements
.def rRead = R20 ; LCD read register
.def rLine = R21 ; LCD line counter
.def rmo = R22 ; LCD multi purpose
; free: R23 .. R29
; Used: Z = R31:R30 for LCD etc.
;
; ----- Reset and interrupt vectors
    rjmp Start ; RESET vector
    reti ; INT0 External Int Request 0
    reti ; PCINT0 Pin Change Int Request 0
    reti ; PCINT1 Pin Change Int Request 1
    reti ; WDT Watchdog Time-out
    reti ; TIM1_CAPT TC1 Capture Event
    reti ; TIM1_COMPA TC1 Compare Match A
    reti ; TIM1_COMPB TC1 Compare Match B
    reti ; TIM1_OVF TC1 Overflow
    reti ; TIM0_COMPA TC0 Compare Match A
    reti ; TIM0_COMPB TC0 Compare Match B
    reti ; TIM0_OVF TC0 Overflow
    reti ; ANA_COMP Analog Comparator
    rjmp AdcIsr ; ADC ADC Conv Complete
    reti ; EE_RDY EEPROM Ready
    reti ; USI_STR USI START
    reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines --
AdcIsr: ; ADC conversion complete
    in rSreg,SREG ; Save SREG
    in rimp,ADCL ; Read LSB ADC
    add rAL,rimp ; Add to sum
    in rimp,ADCH ; Read MSB ADC
    adc rAH,rimp ; Add to sum with carry
    dec rCtr ; Counter measure
    brne AdcIsrRet ; Not zero
    in rimp,ADMUX ; Read Mux port
    cpi rimp,cMuxU ; Was voltage measured?
    breq AdcIsrI ; Yes, start I measuring
    sbr rFlag,1<<bRdyC ; Set ready flag current
    ldi rimp,cMuxU ; Start voltage measuring
    out ADMUX,rimp ; in ADMUX port
    rjmp AdcIsrCont ; End of routine
AdcIsrI:
    sbr rFlag,1<<bRdyV ; Set voltage measurement
    ldi rimp,cMuxI ; Start current measurement
    out ADMUX,rimp ; in ADMUX port
AdcIsrCont:
    ldi rCtr,64 ; Restart counter
    mov rM0,rAL ; Transfer measured sum
    mov rM1,rAH
    clr rAL ; Clear sum
    clr rAH
AdcIsrRet:
    ; Restart ADC conversion
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; in ADC control port
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Reset vector, program start
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; to stack pointer
    ; Init LCD control port
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRSS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Control outputs off
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; LCD data port write
    out pLcdDR,rmp ; to LCD data direction port
    ; Init LCD
    rcall LcdInit ; Call init in include

```

```

ldi ZH,High(2*LcdTextOut) ; Z to text
ldi ZL,Low(2*LcdTextOut)
rcall LcdText ; Display text
ldi rmp,0x0C ; Cursor and blink off
rcall Lcd4Byte
; Start ADC
ldi rmp,cMuxU ; Start voltage measuring
out ADMUX,rmp ; in ADMUX
clr rAL ; Clear sum
clr rAH
ldi rCtr,64 ; 64 measures
ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; Start first ADC conversion
; Sleep mode
ldi rmp,1<<SE ; Sleep Mode Idle
out MCUCR,rmp
; Enable interrupts
sei ; Set I flag
;
; Main program loop
Loop:
    sleep ; Go to sleep
    nop ; Wake up
    sbrc rFlag,bRdyV ; Skip next if voltage ready flag not set
    rcall DisplayV ; Display voltage
    sbrc rFlag,bRdyC ; Skip next if current ready flag not set
    rcall DisplayC ; Display current
    rjmp Loop ; Go back to sleep
;
; Calculate result voltage
DisplayV:
    cbr rFlag,1<<bRdyV ; Clear flag
    ; Multiply with constant cMult
    ldi ZH,High(cMultU) ; Constant to Z
    ldi ZL,Low(cMultU)
    rcall Multiplication ; rM1:rM2 * ZH:ZL
    ; Round result in rR3:rR2
    ldi rmp,0x7F ; Adder
    add rR0,rmp ; Round up
    adc rR1,rmp ; Round up with carry
    ldi rmp,0 ; Adder
    adc rR2,rmp ; Round up with carry
    adc rR3,rmp ; Round up with carry
    ; Display on LCD
    ldi ZH,1 ; Set position on LCD
    ldi ZL,4
    rcall LcdPos
    clt ; Suppress leading zeros
    ldi ZH,High(2*DecimalTab)
    ldi ZL,Low(2*DecimalTab)
DisplayV1:
    lpm rR0,Z+ ; Read decimal number
    lpm rR1,Z+
    tst rR0 ; LSB zero?
    breq DisplayV5 ; Yes, done
    clr rmp ; Digit count
DisplayV2:
    sub rR2,rR0 ; Subtract decimal
    sbc rR3,rR1
    brccs DisplayV3 ; Digit complete
    inc rmp ; Increase digit count
    rjmp DisplayV2 ; Go on subtracting
DisplayV3:
    add rR2,rR0 ; Restore last subtraction
    adc rR3,rR1
    tst rmp ; Digit zero?
    brne DisplayV4 ; No
    brts DisplayV4 ; No suppression of leading zeros
    ldi rmp,' ' ; Suppress leading zero
    rcall LcdD4Byte
    set ; No suppression of leading zeros any more
    rjmp DisplayV1 ; Go on
DisplayV4:
    set ; No suppression of leading zeros any more
    subi rmp,-'0' ; Add ASCII-0
    rcall LcdD4Byte
    cpi ZL,LOW(2*DecimalTab10) ; Decimal point?

```

```

        brne DisplayV1 ; No, go on
        ldi rmp,'.' ; Decimal point
        rcall LCD4Byte
        rjmp DisplayV1 ; Go on
DisplayV5:
        ldi rmp,'0' ; Last digit
        add rmp,rR3 ; Add ASCII-0
        rjmp LCD4Byte
;
; Decimal table thousands
DecimalTab:
.dw 1000
.dw 100
DecimalTab10:
.dw 10
.dw 0
;
; Calculate and display current
DisplayC:
        cbr rFlag,1<<bRdyC ; Clear current flag
        ldi ZH,2 ; Set LCD position
        ldi ZL,4
        rcall LCDPos
        ; Multiply with constant cMulti
        ldi ZH,High(cMulti) ; Constant to Z
        ldi ZL,Low(cMulti)
        rcall Multiplication ; rM1:rM2 * ZH:ZL
        ; Round result in rR3:rR2:rR1:rR0
        ldi rmp,0x7F ; Adder
        add rR0,rmp ; Add last byte
        adc rR1,rmp ; Add pre last byte
        ldi rmp,0 ; Adder
        adc rR2,rmp ; Add carry
        adc rR3,rmp ; Add carry
        ; Display on LCD
        clt ; Suppress leading zeros
        ldi ZH,High(2*DecimalTab) ; Point to decimal table
        ldi ZL,Low(2*DecimalTab)
DisplayC1:
        lpm rM2,Z+ ; Read decimal
        lpm rM3,Z+
        tst rM2 ; LSB zero?
        breq DisplayC7 ; Yes, done
        clr rmp ; Digit counter
DisplayC2:
        sub rR2,rM2 ; Subtract decimal
        sbc rR3,rM3
        brcs DisplayC3 ; Carry: end of subtraction
        inc rmp ; Increase digit
        rjmp DisplayC2 ; Go on subtracting
DisplayC3:
        add rR2,rM2 ; Add again
        adc rR3,rM3
        tst rmp ; Digit zero?
        brne DisplayC4 ; No
        brts DisplayC5 ; Suppression of leading zeros off
        cpi ZL,Low(2*DecimalTab10) ; Decimal point?
        breq DisplayC4 ; No
        ldi rmp,' ' ; Replace 0 by blank
        rcall LCD4Byte
        rjmp DisplayC1 ; Go on
DisplayC4:
        set ; Set flag no zero suppression any more
DisplayC5:
        subi rmp,-'0' ; Convert to ASCII
        rcall LCD4Byte
        rjmp DisplayC1
DisplayC7:
        ldi rmp,'.' ; Display decimal point
        rcall LCD4Byte
        ldi rmp,'0' ; Display last digit
        add rmp,rR2
        rjmp LCD4Byte
;
; 16-by-16 bit multiplication
Multiplication: ; rM1:rM2 * ZH:ZL
; Result in rR3:rR2:rR1:rR0

```

```

    clr rM2 ; Clear bytes 3 and 4
    clr rM3
    clr rR0 ; Clear result
    clr rR1
    clr rR2
    clr rR3
Multiplication1:
    lsr ZH ; Shift bit from MSB to LSB
    ror ZL ; Shift lowest bit to carry
    brcc Multiplication2 ; Do not add
    add rR0,rM0 ; Add multiplicator to result
    adc rR1,rM1
    adc rR2,rM2
    adc rR3,rM3
Multiplication2:
    mov rmp,ZL ; All ones shifted out?
    or rmp,ZH
    breq Multiplication3 ; Yes, done
    lsl rM0 ; Multiply multiplicator by two
    rol rM1
    rol rM2
    rol rM3
    rjmp Multiplication1 ; Continue multiplication
Multiplication3:
    ret
;
; ----- LCD routines -----
LcdTextOut:
.db "Voltage and current",0x0D
.db "U = xx,xx V",0x0D
.db "I = xxx,x mA",0xFE,0xFF
;        4
;
; LCD include routines
.include "Lcd4Busy.inc"
;
; End of source code
;

```

14.2.4 Example

Spannung+Strom tn24
U = 4,90 V
I = 540,9 mA

That is such an example (the German version).

[Home](#) [Top](#) [Voltage](#) [Current](#) [Temperature](#)

14.3 Measuring, calculation and display of temperatures

The ATtiny24 has an internal temperature sensor. This is utilized here.

14.3.1 Hardware

For measuring the temperature no external components are necessary.

14.3.2 Measuring range, measuring, calculation and display

14.3.2.1 Measuring range

ATMEL provides the following typical measurement results for temperatures:

t [°C]	ADC
-40	230
25	300
85	370

Temperatures below -40 and above +85 °C are unpractical and beyond the operating range of the controller.

14.3.2.2 Measuring temperature

The temperature measurement is initiated by setting the ADC multiplexer to 8. This is documented in the device data-book and used in the source code.

14.3.2.3 Calculation

From the above listed values the ADC result is

$$N_{\text{meas}} = 1.1194 * t [^{\circ}\text{C}] + 273.88.$$

From that the temperature calculation is $t [^{\circ}\text{C}] = 0.89286 * N_{\text{meas}} - 244.52$. For 64 measurements and multiplied by 65,536 the following results:

$$\begin{aligned} t [^{\circ}\text{C}] &= 65536 / 64 * 0.89286 * N_{\text{Sum-meas}} / 65536 - 245 \\ &= 914 * N_{\text{Sum-meas}} / 65536 - 245. \end{aligned}$$

Factually the parameter 245 is inaccurate and has to be adjusted. To determine this parameter practically, the display of the measured temperature is written to the LCD in hex. In my case this resulted at 21°C in 0x013A, hence 35.8. The resulting temperature was by 15°C too high, the parameter 245 had to be increased by 15.

Who wants it even more accurate has to determine the slope, too, by determining the ADC result at two different temperatures and change the parameter cMultT in the source code accordingly.

The current source code displays the temperature with a resolution of 1°C. For displaying a resolution of 0.1°C one has to change the math, e.g. the multiplicator should be 9,143 and the subtractor 2.445 to arrive at tenth of degrees. As the physical accuracy of the sensor is only 0.89°C, the displayed tenth of degrees pretends a higher accuracy than really has been measured.

14.3.2.4 Display

The displayed temperature is an integer value. It can be positive or negative.

14.3.3 Program

The program is listed as follows, the [source code is here](#) and requires [the LCD include routines](#) to assemble.

The degree character has to be programmed actively because it is not available with the standard character set of the LCD.

```
; ****
; * Voltage, current and temperature measurement *
; * (C) 2017 by www.avr-asn-tutorial.net *
; ****
;
.NOLIST
```

```

.INCLUDE "tn24def.inc"
.LIST
;
; ----- Switch -----
.equ debugDisplay = 0 ; Displays temperature in hex
;
; ----- Hardware -----
; ATtiny24
;      1 / |14
;      VCC o--|VCC  GND|--o GND
;      2 |    |13
; LCD-RS o--|PB0      |--o NC
;      3 |    |12
; LCD-RW o--|PB1 ADC1|--o GND
;      4 |    |11
; VCC-10k o--|RES ADC2|--o 0.1 Ohms
;      5 |    |10  1 M to input
; LCD-E o--|PB2 ADC3|--o 56k2 to GND
;      6 |    |9
; LCD-D7 o--|PA7  PA4|--o LCD-D4
;      7 |    |8
; LCD-D6 o--|PA6  PA5|--o LCD-D5
;      |_____|

; ----- Timing -----
; Clock frequency   1,000,000 cs/s
; ADC prescaler     128
; ADC clock frequency 7,812.5 Hz
; ADC clocks per measure 14.5
; Measuring frequency 538.8 cs/s
; Measuring cycle (64) 8.42 cs/s
; Three measuring cycles 2.81 cs/s
;

; ----- Constants -----
; Voltage
.equ cRInp = 1000000 ; Ohm
.equ cRGnd = 56200 ; Ohm
.equ cMultU = ((cRInp+cRGnd)*110+cRGnd/2)/cRGnd
; Current
.equ cRI = 100 ; Milliohm
.equ cMulti = (550000+cRI/2) / cRI
; Temperature
.equ cMultT = 914
.equ cMinusT = 245+15 ; Temperature plus correction
; ADMUX constants
.equ cMuxV = (1<<REFS1) | (1<<MUX1) | (1<<MUX0)
.equ cMuxC = (1<<REFS1) | 0b101101
.equ cMuxT = (1<<REFS1) | 0b100010
;

; ----- Register -----
; Used: R0 by LCD
; free: R1 .. R4
.def rAL= R5 ; LSB ADC sum
.def rAH = R6 ; dto., MSB
.def rM0 = R7 ; 32 bit multiplicator
.def rM1 = R8
.def rM2 = R9
.def rM3 = R10
.def rR0 = R11 ; 32 bit result
.def rR1 = R12
.def rR2 = R13
.def rR3 = R14
.def rSreg = R15 ; Save/restore SREG
.def rmp = R16 ; Multi purpose register
.def rimp = R17 ; Multi purpose inside interrupts
.def rFlag = R18 ; Flag register
    .equ bRdyV = 0 ; Ready flag voltage
    .equ bRdyC = 1 ; Ready flag current
    .equ bRdyT = 2 ; Ready flag temperature
.def rCtr = R19 ; Counter for measurements
.def rRead = R20 ; LCD read register
.def rLine = R21 ; LCD line counter
.def rmo = R22 ; LCD additional multi purpose
; free: R23 .. R29
; Used: Z = R31:R30 for LCD etc.
;

; ----- Reset and interrupt vectors

```

```

rjmp Start ; RESET vector
reti ; INT0 External Int Request 0
reti ; PCINT0 Pin Change Int Request 0
reti ; PCINT1 Pin Change Int Request 1
reti ; WDT Watchdog Time-out
reti ; TIM1_CAPT TC1 Capture Event
reti ; TIM1_COMPA TC1 Compare Match A
reti ; TIM1_COMPB TC1 Compare Match B
reti ; TIM1_OVF TC1 Overflow
reti ; TIM0_COMPA TC0 Compare Match A
reti ; TIM0_COMPB TC0 Compare Match B
reti ; TIM0_OVF TC0 Overflow
reti ; ANA_COMP Analog Comparator
rjmp AdcIsr ; ADC ADC Conv Complete
reti ; EE_RDY EEPROM Ready
reti ; USI_STR USI START
reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routines --
AdcIsr:
    in rSreg,SREG ; Save SREG
    in rimp,ADCL ; Read ADC LSB
    add rAL,rimp ; Add to sum
    in rimp,ADCH ; Read ADC MSB
    adc rAH,rimp ; Add to sum plus carry
    dec rCtr ; Count measurements
    brne AdcIsrRet ; Not yet zero
    in rimp,ADMUX ; Read ADMUX
    cpi rimp,cMuxV ; Voltage measured?
    breq AdcIsrC ; Yes, start current measurement
    cpi rimp,cMuxC ; Current measured?
    breq AdcIsrT ; Yes, start temperature measurement
    sbr rFlag,1<<bRdyT ; Set T flag, start voltage
    ldi rimp,cMuxV ; Measure voltage
    out ADMUX,rimp ; in ADMUX
    rjmp AdcIsrCont
AdcIsrC:
    sbr rFlag,1<<bRdyV ; Set voltage flag
    ldi rimp,cMuxC ; Start current measurement
    out ADMUX,rimp ; in ADMUX
    rjmp AdcIsrCont
AdcIsrT:
    sbr rFlag,1<<bRdyC ; Set current flag
    ldi rimp,cMuxT ; Start temperature measurement
    out ADMUX,rimp ; in ADMUX
AdcIsrCont:
    ldi rCtr,64 ; Restart counter
    mov rM0,rAL ; Transfer sum
    mov rM1,rAH
    clr rAL ; Clear sum
    clr rAH
AdcIsrRet:
    ldi rimp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out ADCSRA,rimp ; Restart ADC measuring
    out SREG,rSreg ; Restore SREG
    reti
;
; ----- Reset vector, program start
Start:
    ; Init stack
    ldi rmp,LOW(RAMEND) ; Point to RAMEND
    out SPL,rmp ; to stack pointer
    ; Init LCD control port
    ldi rmp,(1<<bLcdCRE) | (1<<bLcdCRRS) | (1<<bLcdCRRW)
    out pLcdCR,rmp ; to LCD control port
    clr rmp ; Control outputs off
    out pLcdCO,rmp ; to LCD control port
    ldi rmp,mLcdDRW ; LCD data port write
    out pLcdDR,rmp ; to LCD data direction port
    ; Init LCD
    rcall LcdInit ; Call init in include
    ldi ZH,High(2*LcdDefChar) ; Define degree char on LCD
    ldi ZL,Low(2*LcdDefChar)
    rcall LcdChars ; Define chars
    ldi ZH,High(2*LcdTextOut) ; Z to text
    ldi ZL,Low(2*LcdTextOut)
    rcall LcdText ; Display text

```

```

ldi rmp,0x0C ; Cursor and blink off
rcall LcdC4Byte
; Start ADC
ldi rmp,cMuxV ; Start voltage measuring
out ADMUX,rmp ; in ADMUX
clr rAL ; Clear sum
clr rAH
ldi rCtr,64 ; 64 measurements
ldi rmp,(1<<ADEN) | (1<<ADSC) | (1<<ADIE) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
out ADCSRA,rmp ; Start first ADC conversion
; Sleep mode
ldi rmp,1<<SE ; Sleep Mode Idle
out MCUCR,rmp
; Enable interrupts
sei ; Set I flag
;
; Main program loop
Loop:
    sleep ; Go to sleep
    nop ; Wake up
    sbrc rFlag,bRdyV ; Skip next if voltage ready flag not set
    rcall DisplayV ; Display voltage
    sbrc rFlag,bRdyC ; Skip next if current ready flag not set
    rcall DisplayC ; Display current
    sbrc rFlag,bRdyT ; Skip next if temperature ready flag not set
    rcall DisplayT
    rjmp Loop ; Go back to sleep
;
; Calculate result voltage
DisplayV:
    cbr rFlag,1<<bRdyV ; Clear flag
    ; Multiply with constant cMult
    ldi ZH,High(cMultU) ; Constant to Z
    ldi ZL,Low(cMultU)
    rcall Multiplication ; rM1:rM2 * ZH:ZL
    ; Round result in rR3:rR2
    ldi rmp,0x7F ; Adder
    add rR0,rmp ; Round up
    adc rR1,rmp ; Round up with carry
    ldi rmp,0 ; Adder
    adc rR2,rmp ; Round up with carry
    adc rR3,rmp ; Round up with carry
    ; Display on LCD
    ldi ZH,1 ; Set position on LCD
    ldi ZL,4
    rcall LcdPos
    clt ; Suppress leading zeros
    ldi ZH,High(2*DecimalTab)
    ldi ZL,Low(2*DecimalTab)
DisplayV1:
    lpm rR0,Z+ ; Read decimal number
    lpm rR1,Z+
    tst rR0 ; LSB zero?
    breq DisplayV5 ; Yes, done
    clr rmp ; Digit count
DisplayV2:
    sub rR2,rR0 ; Subtract decimal
    sbc rR3,rR1
    brccs DisplayV3 ; Digit complete
    inc rmp ; Increase digit count
    rjmp DisplayV2 ; Go on subtracting
DisplayV3:
    add rR2,rR0 ; Restore last subtraction
    adc rR3,rR1
    tst rmp ; Digit zero?
    brne DisplayV4 ; No
    brts DisplayV4 ; No suppression of leading zeros
    ldi rmp,' ' ; Suppress leading zero
    rcall LcdD4Byte
    set ; No suppression of leading zeros any more
    rjmp DisplayV1 ; Go on
DisplayV4:
    set ; No suppression of leading zeros any more
    subi rmp,-'0' ; Add ASCII-0
    rcall LcdD4Byte
    cpi ZL,LOW(2*DecimalTab10) ; Decimal point?
    brne DisplayV1 ; No, go on

```

```

ldi rmp,'.' ; Decimal point
rcall LcdD4Byte
rjmp DisplayV1 ; Go on
DisplayV5:
    ldi rmp,'0' ; Last digit
    add rmp,rR3 ; Add ASCII-0
    rjmp LcdD4Byte
;
; Decimal table thousands
DecimalTab:
.dw 1000
.dw 100
DecimalTab10:
.dw 10
.dw 0
;
; Calculate and display current
DisplayC:
    cbr rFlag,1<<bRdyC ; Clear current flag
    ldi ZH,2 ; Set LCD position
    ldi ZL,4
    rcall LcdPos
    ; Multiply with constant cMulti
    ldi ZH,High(cMulti) ; Constant to Z
    ldi ZL,Low(cMulti)
    rcall Multiplication ; rM1:rM2 * ZH:ZL
    ; Round result in rR3:rR2:rR1:rR0
    ldi rmp,0x7F ; Adder
    add rR0,rmp ; Add last byte
    adc rR1,rmp ; Add pre last byte
    ldi rmp,0 ; Adder
    adc rR2,rmp ; Add carry
    adc rR3,rmp ; Add carry
    ; Display on LCD
    clt ; Suppress leading zeros
    ldi ZH,High(2*DecimalTab) ; Point to decimal table
    ldi ZL,Low(2*DecimalTab)
DisplayC1:
    lpm rM2,Z+ ; Read decimal
    lpm rM3,Z+
    tst rM2 ; LSB zero?
    breq DisplayC7 ; Yes, done
    clr rmp ; Digit counter
DisplayC2:
    sub rR2,rM2 ; Subtract decimal
    sbc rR3,rM3
    brccs DisplayC3 ; Carry: end of subtraction
    inc rmp ; Increase digit
    rjmp DisplayC2 ; Go on subtracting
DisplayC3:
    add rR2,rM2 ; Add again
    adc rR3,rM3
    tst rmp ; Digit zero?
    brne DisplayC4 ; No
    brts DisplayC5 ; Suppression of leading zeros off
    cpi ZL,Low(2*DecimalTab10) ; Decimal point?
    breq DisplayC4 ; No
    ldi rmp,' ' ; Replace 0 by blank
    rcall LcdD4Byte
    rjmp DisplayC1 ; Go on
DisplayC4:
    set ; Set flag no zero suppression any more
DisplayC5:
    subi rmp,-'0' ; Convert to ASCII
    rcall LcdD4Byte
    rjmp DisplayC1
DisplayC7:
    ldi rmp,'.' ; Display decimal point
    rcall LcdD4Byte
    ldi rmp,'0' ; Display last digit
    add rmp,rR2
    rjmp LcdD4Byte
;
; Calculate temperature
DisplayT:
    cbr rFlag,1<<bRdyT ; Clear T flag
.if debugDisplay == 1 ; Debug, display in hex

```

```

        lsr rM1 ; / 2, divide sum by 64
        ror rM0
        lsr rM1 ; / 4
        ror rM0
        lsr rM1 ; / 8
        ror rM0
        lsr rM1 ; / 16
        ror rM0
        lsr rM1 ; / 32
        ror rM0
        lsr rM1 ; / 64
        ror rM0
        ldi ZH,3 ; Set LCD position
        ldi ZL,11
        rcall LcdPos
        mov rmp,rM1 ; MSB to rmp
        rcall LcdHex ; Display in hex
        mov rmp,rM0 ; LSB to rmp
        rjmp LcdHex ; Display in hex
LcdHex: ; Display rmp in hex
        push rmp ; Save rmp
        swap rmp ; Upper nibble first
        rcall LcdHexN ; Display nibble
        pop rmp ; Restore rmp
LcdHexN: ; Display nibble
        andi rmp,0x0F ; Mask lower nibble
        subi rmp,-'0' ; Add ASCII-0
        cpi rmp,'9'+1 ; A to F?
        brccs LcdHexN1 ; No
        subi rmp,-7 ; Add 7 to adjust to A to F
LcdHexN1:
        rjmp LcdD4Byte ; Display nibble on LCD
        .endif
        ; Multiply with constant cMultT
        ldi ZH,High(cMultT) ; Load constant to Z
        ldi ZL,Low(cMultT)
        rcall Multiplication ; rM1:rM0 * ZH:ZL
        ; Round result in rR3:rR2
        ldi rmp,0x7F ; Adder
        add rR0,rmp ; Add to LSB result
        adc rR1,rmp ; Add to MSB result
        ldi rmp,0 ; Adder = 0
        adc rR2,rmp ; Add carry flag to byte 3
        adc rR3,rmp ; Add carry flag to byte 4
        ; Subtract constant cMinusT
        ldi rmp,Low(cMinusT)
        sub rR2,rmp ; Subtract LSB
        ldi rmp,High(cMinusT)
        sbc rR3,rmp ; Subtract MSB and carry
        ; Display LSB on LCD
        ldi ZH,3 ; Position on LCD
        ldi ZL,4
        rcall LcdPos
        ldi rmp,'+' ; Positive
        tst rR2 ; Test if negative
        brpl DisplayT1 ; No, continue
        ldi rmp,'-' ; Set negative
        neg rR2 ; Subtract from 0xFF
DisplayT1:
        rcall LcdD4Byte ; Display sign
        ldi ZL,10 ; Decimal 10
        clr rmp ; Digit counter
DisplayT2:
        sub rR2,ZL ; Subtract decimal
        brccs DisplayT3 ; Carry, to end
        inc rmp ; Increase digit counter
        rjmp DisplayT2 ; Go on subtracting
DisplayT3:
        add rR2,ZL ; Take back last subtraction
        tst rmp ; Digit = zero?
        brne DisplayT4 ; no
        ldi rmp,' ' ; Blank instead leading zero
        rjmp DisplayT5 ; Display
DisplayT4:
        subi rmp,-'0' ; To ASCII
DisplayT5:
        rcall LcdD4Byte ; Display digit

```

```

ldi rmp,'0' ; ASCII-0
add rmp,rR2 ; Convert digit to ASCII
rjmp LcdD4Byte ; Write last digit
;
;
; 16-by-16 bit multiplication
Multiplication: ; rM1:rM2 * ZH:ZL
; Result in rR3:rR2:rR1:rR0
    clr rM2 ; Clear bytes 3 and 4
    clr rM3
    clr rR0 ; Clear result
    clr rR1
    clr rR2
    clr rR3
Multiplication1:
    lsr ZH ; Shift bit from MSB to LSB
    ror ZL ; Shift lowest bit to carry
    brcc Multiplication2 ; Do not add
    add rR0,rM0 ; Add multiplicator to result
    adc rR1,rM1
    adc rR2,rM2
    adc rR3,rM3
Multiplication2:
    mov rmp,ZL ; All ones shifted out?
    or rmp,ZH
    breq Multiplication3 ; Yes, done
    lsl rM0 ; Multiply multiplicator by two
    rol rM1
    rol rM2
    rol rM3
    rjmp Multiplication1 ; Continue multiplication
Multiplication3:
    ret
;
; ----- LCD routines -----
LcdTextOut:
.db "Voltage/Current/Temp",0x0D,0xFF
.db "U = xx,xx V",0x0D
.db "I = xxx,x mA",0x0D,0xFF
.db "T = +xx ",0x00,"C",0xFE,0xFF
;
        4
; Dregree character
LcdDefChar:
.db 64,0,14,10,14,0,0,0,0,0 ; Z = 0, Grad
.db 0,0 ; End of character table
;
; LCD include routines
.include "Lcd4Busy.inc"
;
; End of source code
;
```

Two new instructions are used here. NEG register subtracts the content of the register from 256 and stores the result in the register. This inverts negative values (bit 7 is set) to their positive value (bit 7 is clear).

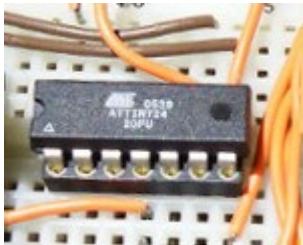
The instruction BRPL label branches to the label if the sign during the last operation is cleared (the value was positive).

14.3.4 Example

Spannung/Strom/Temp.
U = 5,00 V
I = 540,9 mA
T = +21 °C

This is an example for temperature measurement.

[Home](#) [Top](#) [Voltage](#) [Current](#) [Temperature](#)



Conclusions from those lectures

Controller internal hardware

The AVR's offer a wide variety of built-in hardware for use. With a few configuration bits this internal hardware can be brought to function as desired. Switching the internal hardware on and configuring it with a few SBI, CBI, IN or OUT instructions is simple and straight forward. Assumed that you understand the timer, its modes and which bit in the configuration port has which consequences if set or cleared. The whole world of the timer is at your command, not only this small part that your BASIC compiler offers to you. You can do a lot of things with those timers as has been demonstrated in the lectures, if you understand the basics of those timers. And those few instructions that are necessary for that are not high-flying programming art and in-transparent like witches but rather easy to understand.

Only a few hardware features are missing in those 14 lectures. Such as asynchronous (UART) and synchronous communication (e.g. I2C). Who wants to bring 50 controllers together to control an automobile needs this feature, the home-brewer does not. Only if he wants to construct a controller farm for his railway model. Or is a computer nerd that is already satisfied if the controller transmits back the content of a certain register to the laptop, only to see that it works, but nothing really practicable.

The same type of people love to use another feature of the controller that is completely missing here: the boot mode. For those who need to change their software in more than 500 controllers on a nearly daily basis, this feature is highly welcome. The concept of distributing software with serious bugs has become an industry standard now-a-days, so the boot mode offers this extensively. Those who want to solve problems by increasing the number of their problems each time they repair one problem welcome that. Of course you need the boot mode if you have to repair millions of controllers that were programmed to swindle about automobile NO_x emissions back to a less-swindling version. Or, in 10 years, to a non-swindling „nothing but the truth“ version. Normal people that are used to write bug-free software do not need this.

External hardware

This is an issue that needs, and should need, the most brain work. A design error in the hardware can blast the whole software concept and, in the worst case, causes compromises in functionality. Those who forgot the bouncing of switches and keys, just because the key they had tested the software with showed no bouncing, will have a heck of work if the software does not work correct with a different switch. I know what I am talking about, I have to click the de-alarm key of my wake-up clock at least three times until the controller accepts that I do not like to be woken up again. This bouncing only started after two years, corrosion is a very slow process sometimes.

The design of external hardware nearly always has consequences for selecting the AVR type, and for the software as well. I encountered several cases when a small change of the hardware caused a completely new design of the software. So putting brain into the design process is a time-and-life-saving approach.

The other side of that, to choose an elephant design to be safe, is pure theory. Why should one use a 96 pin AVR in an airplane model, of which only 6 pins are useful and needed (one as pulse input, one as PWM output for motor power, and two each for elevator and side rudders)? Many people now-a-days using AVR's were socialized by their Arduino. Not because it is rather

practical but just because it is low cost. And those cannot imagine to design and program controllers for solving real problems. Just using one LED to debug their software is out of their scope because they are used to use far more mighty debugging tools. Those are not usable in a small ATtiny10 for a rudder machine, because their powerful but extensive libraries do not fit in there.

Program design

Especially for starters this aspect is the most dissuasive. The lengthy code listings that started in lecture 6 can give the impression that this is all too complicated to understand. Therefore here the following hints:

1. Interrupt driven programs have all the same basic structure. First find out what the interrupts and their associated routines do and which flags they set. This is already half the way to fully understanding what is going on.
2. The controller hardware is first initialized in the RESET vector. Which internal hardware is involved there? In which modes run timers, ADCs and other internal hardware?
3. Try to conclude on standard processes using the documentation of the software. A multiplication routine always looks the same, 18 to 20 instructions for that belong together, you do not need to understand each of those if the sense of this code package is clear.
4. Try to identify and paint algorithms and to clear the conditions that cause branching. Identify callings and their input parameters and their output provided.

The basic decision if existing software is usable to solve a task or if a new own design might be more effective is not simple for a beginner. Very often the decision to use existing software turns out to be the more complicated way because adjustments to own needs are too extensive. With increasing experience in design and programming the decision is clearer: other's horses are not the first choice if you need reliability.

Design your own from scratch and you can be sure that all your needs are optimally reached.

Good luck and short debug sessions.

Attachment 1: Preferred register uses

#	8 Bit, preferred use	Name	16 bit register pair	Names
0	Reading from flash memory (LPM)		Multiplication result (hardware multiplication)	R1:R0
1				
2				
3				
4				
5				
6				
7	Calculation			
8	Storage			
9	SRAM access			
10				
11				
12				
13				
14				
15	SREG save/restore	rSreg		
16	Multi purpose register (Temp)	rmp		
17	Multi purpose register in ISRs (iTemp)	rimp		
18	Flag register	rFlag		
19				
20				
21	Counters with LDI/ORI/ANDI/SUBI/SBR			
22				
23				
24			16 bit counter	rCntH:rCntL
25				
26			Pointer	X, XH:XL
27				
28			Pointer with displacement	Y, YH:YL
29				
30			Reading from flash memory (LPM)	Z, ZH:ZL
31			Pointer with displacement	

Attachment 2: Instructions in AVR-Assembler

(Marked mnemonics lead to additional descriptions and examples when clicked)

Mnem.	P1	P2	Description	Action	Flags	Clk	Limitations	Words
Arithmetical and logical operations								
ADD	Rx	Ry	Add registers	$Rx \leftarrow Rx + Ry + C$	Z,C,N,V,S,H	1		1
ADC	Rx	Ry	Add registers and carry	$Rx \leftarrow Rx + Ry + C$	Z,C,N,V,S,H	1		1
ADIW	RdL	K	Add constant to word	$RdH:RdL \leftarrow RdH:RdL + K$	Z,C,N,V,S	2	RdL=24/26/28/30 K: 0 to 63	1
SUB	Rx	Ry	Subtract registers	$Rx \leftarrow Rx - Ry$	Z,C,N,V,S,H	1		1
SUBI	Rh	K	Subtract constant	$Rh \leftarrow Rh - K$	Z,C,N,V,S,H	1	R: 16 to 31	1
SBC	Rx	Ry	Subtract registers and carry	$Rx \leftarrow Rx - Ry - C$	Z,C,N,V,S,H	1		1
SBCI	Rh	K	Subtract constant and carry	$Rh \leftarrow Rh - K - C$	Z,C,N,V,S,H	1	R: 16 to 31	1
CP	Rx	Ry	Compare registers	$Rx - Ry$	Z,C,N,V,S,H	1		1
CPC	Rx	Ry	Compare registers and carry	$Rx - Ry - C$	Z,C,N,V,S,H	1		1
CPI	Rh	K	Compare with constant	$Rx - K$	Z,C,N,V,S,H	1	R: 16 to 31, K: 0 to 255	1
SBIW	RdL	K	Subtract constant from word	$RdH:RdL \leftarrow RdH:RdL - K$	Z,C,N,V,S	2	RdL=24/26/28/30 K: 0 to 63	1
AND	Rx	Ry	Binary AND	$Rx \leftarrow Rx \text{ AND } Ry$	Z,N,V,S	1		1
ANDI	Rh	K	Binary AND with constant	$Rh \leftarrow Rh \text{ AND } K$	Z,N,V,S	1	R: 16 to 31, K: 0 to 255	1
OR	Rx	Ry	Binary OR of registers	$Rx \leftarrow Rx \text{ OR } Ry$	Z,N,V,S	1		1
ORI	Rh	K	Binary OR with constant	$Rh \leftarrow Rh \text{ OR } K$	Z,N,V,S	1	R: 16 to 31, K: 0 to 255	1
EOR	Rx	Ry	Exclusive OR of registers	$Rx \leftarrow Rx \text{ EXOR } Ry$	Z,N,V,S	1		1
COM	Rx		One's complement	$Rx \leftarrow 255 - Rx$	Z,C,N,V,S	1		1
NEG	Rx		Two's complement	$Rx \leftarrow 256 - Rx$	Z,C,N,V,S,H	1		1
SBR	Rh	K	Set bits in constant K	$Rh \leftarrow Rh \text{ OR } K$	Z,N,V,S	1	R: 16 to 31, K: 0 to 255	1
CBR	Rh	K	Clear bits that are set in constant K	$Rh \leftarrow Rh \text{ AND } (\text{NEG } K)$	Z,N,V,S	1	R: 16 to 31, K: 0 to 255	1
INC	Rx		Increase register content by one	$Rx \leftarrow Rx + 1$	Z,N,V,S	1		1
DEC	Rx		Decrease register content by one	$Rx \leftarrow Rx - 1$	Z,N,V,S	1		1
TST	Rx		Test register	$Rx \leftarrow Rx \text{ ODER } Rx$	Z,N,V,S	1		1
CLR	Rx		Clear register content	$Rx \leftarrow 0$	Z,N,V,S	1		1
SER	Rh		Set all bits in register	$Rh \leftarrow 255$	-	1	R: 16 to 31	1
MUL	Rx	Ry	Multiply 8-by-8 bits	$R1:R0 \leftarrow Rx * Ry$	Z,C	2		1
MULS	Rx	Ry	Multiply signed	$R1:R0 \leftarrow Rx * Ry$	Z,C	2	Rx,Ry: 16 to 31	1
MULSU	Rx	Ry	Multipliziere unsigned and signed	$R1:R0 \leftarrow Rx * Ry$	Z,C	2	Rx,Ry: 16 to 31	1
FMUL	Rx	Ry	Floating point multiplication	$R1:R0 \leftarrow Rx * Ry$	Z,C	2	Rx,Ry: 16 to 23	1
FMULS	Rx	Ry	Floating point multiplication signed	$R1:R0 \leftarrow Rx * Ry$	Z,C	2	Rx,Ry: 16 to 23	1
FMULSU	Rx	Ry	Floating point multiplication unsigned and signed	$R1:R0 \leftarrow Rx * Ry$	Z,C	2	Rx,Ry: 16 to 23	1
DES	K		Data encryption/decryption	(R7:R0, R15:R8)	-	1/2	(MEGA/XMEGA only), K<16	1
Jump instructions								
RJMP	K		Relative jump	$(PC) \leftarrow (PC) +/- K$	-	2	K: -2048 to 2047	1
IJMP			Indirect jump	$(PC) \leftarrow Z$	-	2		1
EIJMP			Extended indirect jump	$(PC) \leftarrow EIND + Z$	-	2	(XMEGA only)	1
JMP	K		Wide jump	$(PC) \leftarrow K$	-	3	K: 0 to 65535	2
RCALL	K		Relative call	$(Stack) \leftarrow (PC), (PC) \leftarrow (PC) +/- K$	-	2/3/4	K: -2048 to 2047	1
ICALL			Indirect call	$(Stack) \leftarrow (PC), (PC) \leftarrow Z$	-	2/3/4		1
EICALL			Extended indirect call	$(Stack) \leftarrow (PC), (PC) \leftarrow EIND+Z$	-	3/4		1
CALL	K		Wide call	$(Stack) \leftarrow (PC), (PC) \leftarrow K$	-	3/4/5		2
RET			Return from call	$(PC) \leftarrow (Stack)$	-	4		1

Mnem.	P1	P2	Description	Action	Flags	Clk	Limitations	Words
RETI			Return from interrupt service	(PC) \leftarrow (Stack), I \leftarrow 1	-	4		1
CPSE	Rx	Ry	Jump over next if equal	Rx=Ry: (PC) \leftarrow (PC + 1)	-	2/3		1
SBRC	Rx	B	Skip next if bit in register is clear	(Bit)=0: (PC) \leftarrow (PC+1)	-	2/3	B: 0 to 7	1
SBRS	Rx	B	Skip next if bit in register is set	(Bit)=1: (PC) \leftarrow (PC+1)	-	2/3	B: 0 to 7	1
SBIC	PL	B	Skip next if port-bit in port is clear	(Bit)=0: (PC) \leftarrow (PC+1)	-	2/3	PL: 0 to 31 B: 0 to 7	1
SBIS	PL	B	Skip next if port-bit in port is clear	(Bit)=1: (PC) \leftarrow (PC+1)	-	2/3	PL: 0 to 31 B: 0 to 7	1
BRBS	K	B	Branch relative if bit in SREG is set	(SREG-Bit=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRBC	K	B	Branch relative if bit in SREG is clear	(SREG-Bit=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BREQ	K		Branch relative if Z bit in SREG is set	(SREG-Z=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRNE	K		Branch relative if Z bit in SREG is clear	(SREG-Z=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRCS	K		Branch relative if C bit in SREG is set	(SREG-C=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRCC	K		Branch relative if C bit in SREG is clear	(SREG-C=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRSH	K		Branch relative if C bit in SREG is clear	(SREG-C=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRLO	K		Branch relative if C bit in SREG is set	(SREG-C=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRMI	K		Branch relative if N bit in SREG is set	(SREG-N=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRPL	K		Branch relative if N bit in SREG is clear	(SREG-N=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRGE	K		Branch relative if S bit in SREG is clear	(SREG-S=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRLT	K		Branch relative if S bit in SREG is set	(SREG-S=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRHS	K		Branch relative if H bit in SREG is set	(SREG-H=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRHC	K		Branch relative if H bit in SREG is clear	(SREG-H=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRTS	K		Branch relative if T bit in SREG is set	(SREG-T=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRTC	K		Branch relative if T bit in SREG is clear	(SREG-T=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRVS	K		Branch relative if V bit in SREG is set	(SREG-V=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRVC	K		Branch relative if V bit in SREG is clear	(SREG-V=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRIE	K		Branch relative if I bit in SREG is set	(SREG-I=1): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1
BRID	K		Branch relative if I bit in SREG is clear	(SREG-I=0): (PC) \leftarrow (PC) +/- K	-	1/2	K: -63 to +64	1

Data copy instructions

MOV	Rx	Ry	Copy register	Rx \leftarrow Ry	-	1		1
MOVW	Rx	Ry	Copy register pair	Rx+1:Rx \leftarrow Ry+1:Ry	-	1	Rx, Ry: Even	1
LDI	Rh	K	Copy constant to register	Rh \leftarrow K	-	1	R: 16 to 31, K: 0 to 255	1
LDS	Rh	A	Copy SRAM to register	Rx \leftarrow (SRAM-A)	-	2/3/4	R: 16 to 31	2
LD	Rx	X	Copy SRAM at address X to register	Rx \leftarrow (X)	-	2/3/4		1
LD	Rx	X+	Copy SRAM at address X to register and increment X	Rx \leftarrow (X), X = X + 1	-	2/3		1
LD	Rx	-X	Decrement X and copy SRAM at address X to register	X = X - 1, Rx \leftarrow (X)	-	2/3/4		1
LD	Rx	Y	Copy SRAM at address Y to register	Rx \leftarrow (Y)	-	2/3/4		1
LD	Rx	Y+	Copy SRAM at address Y to	Rx \leftarrow (Y), Y = Y + 1	-	2/3		1

Mnem.	P1	P2	Description	Action	Flags	Clk	Limitations	Words
			register and increment Y					
LD	Rx	-Y	Decrement Y and copy SRAM at address Y to register	$Y = Y - 1, Rx \leftarrow (Y)$	-	2/3/4		1
LDD	Rx	Y+K	Copy SRAM at address (Y+K) to register	$Rx \leftarrow (Y+K)$	-	2/3	K: 0 to 63	1
LD	Rx	Z	Copy SRAM at address Z to register	$Rx \leftarrow (Z)$	-	2/3/4		1
LD	Rx	Z+	Copy SRAM at address Z to register and increment Z	$Rx \leftarrow (Z), Z = Z + 1$	-	2/3		1
LD	Rx	-Z	Decrement Z and copy SRAM at address Z to register	$Z = Z - 1, Rx \leftarrow (Z)$	-	2/3/4		1
LDD	Rx	Z+K	Copy SRAM at address (Z+K) to register	$Rx \leftarrow (Z+K)$	-	2/3	K: 0 to 63	1
STS	A	Rh	Copy register to address A in SRAM	$(SRAM-A) \leftarrow Rx$	-	2/3/4	R: 16 to 31	2
ST	X	Rx	Copy register to SRAM at address X	$(X) \leftarrow Rx$	-	2/3/4		1
ST	X+	Rx	Copy register to SRAM address X and increment X	$(X) \leftarrow Rx, X = X + 1$	-	2/3		1
ST	-X	Rx	Decrement X and copy register to SRAM at address X	$X = X - 1, (X) \leftarrow Rx$	-	2/3/4		1
ST	Y	Rx	Copy register to SRAM at address Y	$(Y) \leftarrow Rx$	-	2/3/4		1
ST	Rx	Y+	Copy register to SRAM at address Y and increment Y	$(Y) \leftarrow Rx, Y = Y + 1$	-	2/3		1
ST	Rx	-Y	Decrement Y and copy register to SRAM at address Y	$Y = Y - 1, (Y) \leftarrow Rx$	-	2/3/4		1
STD	Y+K	Rx	Copy register to SRAM at address (Y+K)	$(Y+K) \leftarrow Rx$	-	2/3	K: 0 to 63	1
ST	Z	Rx	Copy register to SRAM at address Z	$(Z) \leftarrow Rx$	-	2/3/4		1
ST	Z+	Rx	Copy register to SRAM at address Z and increment Z	$(Z) \leftarrow Rx, Z = Z + 1$	-	2/3		1
ST	-Z	Rx	Decrement Z and copy register to SRAM at address Z	$Z = Z - 1, (Z) \leftarrow Rx$	-	2/3/4		1
STD	Z+K	Rx	Copy register to SRAM at address (Z+K)	$(Z+K) \leftarrow Rx$	-	2/3	K: 0 to 63	1
LPM			Copy byte in flash at address (Z) to register R0	$R0 \leftarrow (\text{Flash } Z)$	-	3		1
LPM	Rx	Z	Copy byte in flash at address (Z) to register	$Rx \leftarrow (\text{Flash } Z)$	-	3		1
LPM	Rx	Z+	Copy byte in flash at address (Z) to register and increment	$Rx \leftarrow (\text{Flash } Z), Z = Z + 1$	-	3		1
ELPM			Copy byte in flash at extended address (Z) to register R0	$R0 \leftarrow (\text{Flash } Z)$	-	3		1
ELPM	Rx	Z	Copy byte in flash at extended address (Z) to register	$Rx \leftarrow (\text{Flash } Z)$	-	3		1
ELPM	Rx	Z+	Copy byte in flash at extended address (Z) to register and increment Z	$Rx \leftarrow (\text{Flash } Z), Z = Z + 1$	-	3		1
SPM			Write word in R1:R0 to flash at (Z)	$(\text{Flash } Z) \leftarrow R1:R0$	-	N		1
SPM	Z+		Write word in R1:R0 to flash at (Z) and increment	$(\text{Flash } Z) \leftarrow R1:R0, Z = Z + 1$	-	N		1
IN	Rx	P	Copy port to register	$Rx \leftarrow P$	-	1	P: 0 to 63	1
OUT	P	Rx	Copy register to port	$P \leftarrow Rx$	-	1	P: 0 to 63	1
PUSH	Rx		Throw register on stack	$(\text{Stack}) \leftarrow Rx, SP = SP - 1$	-	2		1
POP	Rx		Pop register from stack	$Rx \leftarrow (\text{Stack}), SP = SP + 1$	-	2		1
XCH	Z	Rx	Exchange register and SRAM at address (Z)	$Rx \leftrightarrow (Z)$	-	1		1
LAS	Z	Rx	OR register with SRAM at address (Z) and exchange	$Rx \leftarrow Rx \text{ ODER } (Z), (Z) \leftrightarrow Rx$	-	1		1
LAC	Z	Rx	AND complemented register with SRAM at (Z) and copy to SRAM at (Z)	$Rx \leftarrow (255-Rx) \text{ UND } (Z), (Z) \leftrightarrow Rx$	-	1		1

Mnem.	P1	P2	Description	Action	Flags	Clk	Limitations	Words
LAT	Z,	Rd	XOR register with SRAM at (Z) and exchange	Rx EXOR (Z), Rx \leftrightarrow (Z)	-	1		1
Bit operations								
LSL	Rx		Logical left shift	Rx \leftarrow Rx * 2	Z,C,N,V,H	1		1
LSR	Rx		Logical right shift	Rx \leftarrow Rx / 2	Z,C,N,V	1		1
ROL	Rx		Binary rotate left via carry	Rx \leftarrow Rx * 2 with bit 0 = C/C = Bit 7	Z,C,N,V,H	1		1
ROR	Rx		Binary rotate right via carry	Rx \leftarrow Rx / 2 with bit 7 = C/C = Bit 0	Z,C,N,V	1		1
ASR	Rx		Arithmetical right shift	Rx \leftarrow Rx(6:0) / 2, Bit 6 = 0	Z,C,N,V	1		1
SWAP	Rx		Exchange upper and lower nibble	Rx \leftarrow (7:4) \leftrightarrow (3:0)	-	1		1
BSET	B		Set bit in SREG	SREG \leftarrow SREG OR (1<<B)	-	1	B: 0 to 7	1
BCLR	B		Clear bit in SREG	SREG \leftarrow SREG AND (255-(1<<B))	-	1	B: 0 to 7	1
SBI	PL	B	Set bit in port	PL \leftarrow PL OR (1<<B)	-	2	PL: 0 to 31, B: 0 to 7	1
CBI	PL	B	Clear bit in port	PL \leftarrow PL AND (255-(1<<B))	-	2	PL: 0 to 31, B: 0 to 7	1
BST	Rx	B	Copy register bit to T	SREG-T \leftarrow Rx-Bit B	-	1	B: 0 to 7	1
BLD	Rx	B	Copy T to register bit	Rx bit B \leftarrow T	-	1	B: 0 to 7	1
SEC			Set SREG C	SREG-Bit C \leftarrow 1	-	1		1
CLC			Clear SREG C	SREG-Bit C \leftarrow 0	-	1		1
SEN			Set SREG N	SREG-Bit N \leftarrow 1	-	1		1
CLN			Clear SREG N	SREG-Bit N \leftarrow 0	-	1		1
SEZ			Set SREG Z	SREG-Bit Z \leftarrow 1	-	1		1
CLZ			Clear SREG Z	SREG-Bit Z \leftarrow 0	-	1		1
SEI			Set SREG I	SREG-Bit I \leftarrow 1	-	1		1
CLI			Clear SREG I	SREG-Bit I \leftarrow 0	-	1		1
SES			Set SREG S	SREG-Bit S \leftarrow 1	-	1		1
CLS			Clear SREG S	SREG-Bit S \leftarrow 0	-	1		1
SEV			Set SREG V	SREG-Bit V \leftarrow 1	-	1		1
CLV			Clear SREG V	SREG-Bit V \leftarrow 0	-	1		1
SET			Set SREG T	SREG-Bit T \leftarrow 1	-	1		1
CLT			Clear SREG T	SREG-Bit T \leftarrow 0	-	1		1
SEH			Set SREG H	SREG-Bit H \leftarrow 1	-	1		1
CLH			Clear SREG H	SREG-Bit H \leftarrow 0	-	1		1
Controller instructions								
BREAK			Stop execution, control to debugger		-	1		1
NOP			Do nothing		-	1		1
SLEEP			Sleep		-	1		1
WDR			Clear watchdog counter	WDR counter \leftarrow 0	-	1		1

Attachment 3: Directives in AVR assembler

Directive	Parameter	Description
Listing		
.LIST		Switches list output on
.NOLIST		Switches list output off
Code source		
.INCLUDE	,"(File name)"	Assembles the code in the target file
Target segments		
.CSEG		Assembles to the code segment Address counter Program Counter PC
.ESEG		Assembles to the EEPROM segment only labels and .ORG/.DB/.DW directives are possible, specific EEPROM address counter
.DSEG		Assembles to the SRAM segment only labels and .ORG/.BYTE/.WORD directives are possible, specific SRAM address counter
Address manipulation		
.ORG	Address	Sets the address counter of the segment forward to the given address
.BYTE	N	Reserves N bytes and increases the address counter by N (DSEG only)
.WORD	N	Reserves N words and increases the address counter by 2*N (DSEG only)
Create tables		
.DB	b1,b2,..bn "(Text)"	Inserts bytes b1 to bn resp. the ASCII codes of the text (Code and EEPROM segment only)
.DW	w1,w2,..wn	Inserts the 2 byte values w1 to wn (Code and EEPROM segment only)
Symbols, names		
.DEF	Name = Rn	Assigns the name with the register
.EQU	Name = Value	Assigns a constant with the value to the name , no subsequent change possible
.SET	Name = Value	Assigns a variable name to a value, subsequent redefinition possible
.UNDEF	Name	De-assigns the constant or variable name
Macros		
.MACRO	Name, Parameter	Starts a macro of the name and uses the given parameters
.ENDMACRO		Closes the macro
Type definition		
.DEVICE	"Typename"	Switches the instruction check for the given AVR device type on (included in def.inc)
Messages, errors		
.MESSAGE	"TEXT"	Outputs the given text
.ERROR	"TEXT"	Provokes an error and outputs the given text as error message
Conditioned assembling		
.EXIT		Ends assembly (subsequent content is ignored)
.IF	Condition	Assembles if the condition is met (TRUE)
.IFDEF	Symbol	Assembles if the symbol is defined
.IFNDEF	Symbol	Assembles if the symbol is not defined
.ELSE		Assembles if the condition in .IF is not met or if the symbol in .IFDEF is not defined
.ELIF	Condition	Assembles if the condition in .IF is not met and the condition in .ELIF is met
.ENDIF		Ends .IF, .IFDEF, .ELSE and .ELIF and switches assembling on
gavrasm specialties		
.DB	%YEAR% %MONTH% %DAY% %HOUR% %MINUTE% %SOURCE%	Inserts current date and time information resp. the filename of the source file as table
.IFDEVICE	"Devicename"	Conditional assembly if the device name of the current device is equal to the parameter

Please note that directives must be separated from parameters by at least one blank or tab character to be correctly recognized.

Attachment 4: Component list by lectures

Total without general components =		34,96 €	Total all components =		109,31 €
#	Component	Lecture	Pieces	Reichelt name	Price(€)
General					
1	Digital Multimeter	any	1	PEAKTECH 1075	16,40
2	Breadboard	any	1	STECKBOARD 2K4V	11,95
3	Cables for Breadboard	any	1	STECKBOARD DBS	3,95
4	Programmer DIAMEX ALL AVR	any	1	DIAMEX ALL AVR	31,50
5	Rechargeable battery pack 4*AAA	any	1	NH TC 4XAAA-1Z	7,70
6	Stranded wire, 10 m, b&w	any	1	ZL214SWW-10	2,80
7	Pin header 2.54 mm, 1X02, straight	any	1	MPE 087-1-002	0,05
Lecture 1: ISP-Interface					
8	Pin header 2.54 mm, 1X03, straight	any	1	MPE 087-1-003	0,14
9	Pin header 2.54 mm, 2X03, straight	any	1	MPE 087-2-006	0,14
10	Breadboard, laminated paper, 50x100 mm	any	1	H25PR050	0,87
11	IC socket, 8-pin, super flat	any	1	GS 8P	0,19
12	ATtiny13A	1 .. 9, 12	1	ATTINY 13A-PU	1,15
13	Carbon film resistor 1/4W, 5%, 10 kilo-ohms	any	1	1/4W 10K	0,10
14	Multi-layer ceramic capacitor 100N, 20%	any	1	Z5U-5 100N	0,04
Lecture 2 (up to lecture 5: LED-PWM): Switch LED on					
15	Carbon film resistor 1/4 W, 5%, 220 ohm	2 .. 9, 12	1	1/4W 220	0,10
16	LED, 5 mm, low-cost, red	2 .. 9, 12	1	LED 5MM RT	0,06
Lecture 6: LED Interrupt					
17	Carbon film resistor 1/4 W, 5%, 220 ohm	6	1	1/4W 220	0,10
18	LED, 5 mm, low-cost, red	6	1	LED 5MM RT	0,06
Lecture 7: Button Interrupt					
19	Duo-LED, 5 mm, 2-pin, red/green	7	1	LED 5 RG	0,18
20	Push button, round, red	7	1	DT 6 RT	0,79
Lecture 8: LED linear intensity control by PWM					
21	Rotary potentiometer, lin., 4 mm, mono, 100 kOhm	8	1	PO4M-LIN 100K	1,90
Lecture 9: Audio generator					
22	Electrolytic capacitor, 10 µF/35 V, 5 x 11 mm	9	1	RAD 10/35	0,04
23	Miniature loudspeaker/0.2 W/45 Ohm	9	1	BL 50A	1,20
Lecture 10: LCD display					
24	ATtiny24-20PU	10 .. 14	1	ATTINY 24-20 PU	2,40
25	IC socket, 14-pin, super flat, turned, gold-plated	10 .. 14	1	GS 14P	0,23
26	LCD dot matrix module, 4 x 20 characters, blue	10 .. 14	1	LCD 204B BL	19,95
27	Pin headers 2.54 mm, 1X16, straight	10 .. 14	1	MPE 087-1-016	0,25
Lecture 11: EEPROM					
28	Push button, round, white	11	1	DT 6 WS	0,79
Lecture 12: IR transmitter and receiver					
29	IR LED, black, 40°, 5 mm	12	1	SFH 4546	0,44
30	IR receiver module TSOP4840 40 kHz	12	1	TSOP 4840	0,66

31	IR receiver modules TSOP31240 40 kHz	12	1	TSOP 31240	0,74
32	Carbon film resistor 1/4W, 5%, 10 kilo-ohms	12	1	1/4W 10K	0,10
33	Multi-layer ceramic capacitor 100N, 20%	12	1	Z5U-5 100N	0,04
32	WIMA film capacitor, Rm 5 mm, 22 nF	12	1	MKS2-63 22N	0,14

Lecture 13: Frequency and induction meter

33	Carbon film resistor 1/4W, 5%, 100 kilo-ohms	13	4	1/4W 100K	0,41
34	Carbon film resistor 1/4 W, 5%, 220 ohm	13	1	1/4W 220	0,10
35	Film capacitor, 100nF, 100V, RM5	13	3	MKS2-100 100N	0,11
36	Sub-min. electrolytic capacitor, radial, 1.0 µF/63 V	13	1	SM 1,0/63RAD	0,04
37	MOS 4011, 4 x NAND, 2 x INPUT	13	1	MOS 4011	0,25
38	IC socket, 14-pin, super flat, turned, gold-plated	13	1	GS 14P	0,23

Lecture 14: Volt and ampere meter

39	Metal film resistor 1.00 MΩ	14	1	METALL 1,00M	0,08
40	Metal film resistor 56.2 kΩ	14	1	METALL 56,2K	0,08
41	5 W wirewound resistor, series 2088, 0.1 Ohms	14	1	5W AXIAL 0,1	0,36
42	Microfuse 5x20mm, quick-acting 2.0 A	14	1	FLINK 2,0A	0,30
43	Fuse holder, 5 x 20 mm, max. 6,3A/250V	14	1	PL 112000	0,18

Attachment 5: Links to the source code files

File name	Description of task	Device
2_Led_On.asm	Switch a LED on with an ATtiny13	ATtiny13
2_LED_On_notepad.asm	switch a LED on with Notepad edit	ATtiny13
3_Led_Blink.asm	Blinking a LED in one second with an ATtiny13	ATtiny13
3_Led_Blink_Fast.asm	To blink a LED fast with an ATtiny13	ATtiny13
4_Blink_128kcs.asm	Blink LED with timer at 128 kcs clock	ATtiny13
4_Led_Blink_timer.asm	LED blinker with 300 kcs/s and CTC	ATtiny13
4_Timer_Blink.asm	Timer to blink a LED	ATtiny13
5_fast_pwm.asm	PWM control of a LED in Fast mode	ATtiny13
6_tc0_int_compA.asm	Timer with COMP-A-Interrupt	ATtiny13
6_tc0_o_int.asm	Timer with Overflow interrupt	ATtiny13
7_Key_Int.asm	Key with INT0 interrupt	ATtiny13
8_IntensityRegulator_1.asm	LED intensity regulator with poti and ADC	ATtiny13
8_IntensityRegulator_2.asm	LED intensity regulator red/green with ADC and key	ATtiny13
8_IntensityRegulator_3.asm	LED intensity control red/green up/down	ATtiny13
8_IntensityRegulator_4.asm	Duo-LED in push-pull ATtiny13	ATtiny13
9_audiogenerator_1.asm	Audio generator with key and tone regulator	ATtiny13
9_audiogenerator_2.asm	Gamut tones with a potentiometer on an ATtiny13	ATtiny13
9_audiogenerator_3.asm	To play a melody with an ATtiny13	ATtiny13
10_Lcd-Display_1.asm	LCD-Display 4*20 on an ATtiny24, 4 bit interface	ATtiny24
10_Lcd-Display_2.asm	LCD display 4*20 on ATtiny24, 4 bit interface with busy	ATtiny24
10_Lcd-Display_3.asm	LCD display 4*20 on ATtiny24/4 bit/busy/arrows	ATtiny24
11_Eeprom_1.asm	Read and write EEPROM of an ATtiny24 and a 4 line LCD	ATtiny24
11_Eeprom_2.asm	To read and write an EEPROM word and a message on LCD	ATtiny24
12_IR-Rx_1.asm	IR receiver with ATtiny24 and LCD to measure header duration	ATtiny24
12_IR-Rx_1a.asm	IR receiver with ATtiny24 and LCD to measure signal numbers	ATtiny24
12_IR-Rx_1b.asm	IR receiver with ATtiny24 and LCD to examine bits	ATtiny24
12_IR-Rx_1c.asm	IR receiver with ATtiny24 and LCD for burst analysis	ATtiny24
12_IR-Rx_Analog.asm	IR receiver with ATtiny24 and LCD for analog values	ATtiny24
12_IR-Rx_Switch.asm	IR receiver and 3-channel switch with tn13/24	ATtiny24
12_IR-Tx.asm	IR transmitter 40 kcs/s with an ATtiny13	ATtiny13
12_IR-Tx_Analog.asm	IR transmit an analog value with 40 kcs/s on an ATtiny13	ATtiny13
13_F-L-Meter.asm	Digital/analog frequency counter and induction measurement	ATtiny24
13_F-Meter_1.asm	Frequency counting of digital signals with ATtiny24/LCD	ATtiny24
13_F-Meter_2.asm	Frequency meter analog and digital with ATtiny24/LCD	ATtiny24
14_U-I-Meter.asm	To measure and display voltage and current	ATtiny24
14_U-I-T-Meter.asm	Voltage, current and temperature measurement	ATtiny24
14_U-Meter.asm	Measuring voltages with an ATtiny24	ATtiny24

Attachment 6: Themes index

Internal hardware

128kHz oscillator.....	45
AD-converter.....	82
ADCSRA port.....	83
ADIE bit.....	83
ADLAR bit.....	82, 83
DIDR port.....	83
REFS0 bit.....	82
Analog comparator.....	245
Interrupt blocking.....	257
ATtiny24.....	137
Clock prescaler.....	40
Current consumption.....	41, 42
EEPROM.....	156
Flash memory.....	15
Fuses.....	11
128kHz-Oszillat....	46
CKDIV8.....	40, 46
EESAVE.....	157, 230
Interrupts	
ADC.....	83
INT0.....	72
GIMSK port.....	84
PCINT.....	84
Priority.....	73
Timer CTC interrupt.....	66
Timer overflow interrupt.....	58
Ports	
Output port pins.....	17
Pull-up resistor.....	13, 72
Direction.....	13, 16
Sink.....	14
Source.....	13
Register.....	30
Stack.....	56
Status register SREG.....	30, 55, 56
Temperature sensor.....	287
Timer.....	35
Blinking.....	38
CTC mode.....	37
CTC mode, 16 bit with ICR.....	50
Fast PWM.....	49, 51
Fast PWM with CTC.....	50
Phase correct PWM.....	50, 50
Prescaler.....	35
PWM mode.....	48
PWM mode outputs.....	99
Switch output signals.....	37

External hardware

Components, all.....	303
ATtiny13.....	5
ATtiny24.....	138
Battery pack, rechargeable.....	4
Breadboard.....	5

Ceramic capacitor 100 nF.....	4
CMOS NAND Gate 4011.....	263
Duo-LED.....	74
Electrolytic capacitor 47µF.....	102
Electrolytic capacitor 1µF.....	256
Film capacitor 22 nF.....	212
Film capacitor 100 nF.....	256
Fuse 2A, holder and cap.....	281
IC socket, 8 pin.....	5
IC socket, 14 pin.....	138
IR LED.....	202
IR receiver modules.....	179
ISP plug.....	4
Key.....	74
LCD display 4*20.....	138
LED red 5mm.....	15
Potentiometer 100k lin.....	84
Resistor 0,1Ω/5W.....	281
Resistor 220Ω.....	15
Resistor 10k.....	4
Resistor 56k2 1%.....	275
Resistor 68Ω.....	202
Resistor 100k.....	256
Resistor 1M 1%.....	275
Speaker 45Ω.....	102
Trim resistor 10k.....	138
INT0 hardware.....	71
LCD.....	133
4 bit interface.....	134
8 bit interface.....	134
Back-light.....	136
Busy flag.....	135
Data transfer.....	135
Init.....	135
Interface.....	135
Character generation.....	149
LED.....	12
Voltage drop.....	12
Resistor.....	13
In-System-Programming.....	4
ISP6 plug.....	1
Programming devices.....	2
Infrared signals.....	176
Data bits.....	195
End signals.....	185
Number of signals.....	195
Start signals.....	176
Transmitting.....	201
Key bouncing.....	72, 73, 91
Schemes	
Current meter.....	281
Duo LED.....	84
Duo LED key.....	74
Duo-LED fast color change.....	99
Frequency analog.....	256
Frequency digital.....	250
Induction measurement.....	262
IR receiver.....	179
IR transmitter.....	202
IR transmitter analog voltage.....	212
IR 3 channel switch.....	227
ISP interface.....	2
Key reset.....	160

LCD with ATtiny24 and wait.....	138
LCD with ATtiny24 and busy.....	145
LED port output, single.....	14
LED port output, double.....	59
PWM DC motor controller.....	48
Speaker.....	101
Voltage meter.....	275

Programming

Assemble.....	16
with gavrasm.....	18
with Studio assembler.....	20
Assembler bit shift 1<<bit name.....	38
Conditioned assembly.....	177
Decimal conversion	
8 bit, 16 bit.....	158
24 bit, 32 bit.....	249
Delay loops	
8 bit.....	30
16 bit.....	30, 139
Inner and outer loop.....	31
Directives.....	302
Error.....	178
Message.....	178
Division	
8 bit by 4.....	87
8 bit by 8 bit.....	246
16 bit by 8 bit.....	247
32 bit by 65,366.....	114
40 bit by 40 bit.....	248
EEPROM	
Read access.....	157
Table with .ESEG.....	110
Write access.....	156, 158
Execution time of instructions.....	27
Flag.....	91
Busy from LCD.....	145
in Interrupt Service Routines.....	58
OCIE0A/B.....	58
SREG-C.....	87, 98, 160
SREG-I.....	55
SREG-T.....	75
SREG-Z.....	30, 31, 61
TOIE0.....	58
Flow diagram.....	28, 32, 75
Frequency measurement.....	256
Induction measurement.....	262
Instructions.....	16, 298
Interrupt.....	55
Execution time.....	203
Program structure.....	59, 62
Sleep mode.....	58
Service routine.....	55
Status register SREG.....	56
Timer compare match.....	58
Vector.....	57

Label (named address).....	28	Square 24 bit.....	246	Simulation with
Listing.....	19	Pre-Fetch.....	26	ATMEL Studio.....
Hex code.....	21	Programming tools ATMEL Studio		avr_sim 24, 39, 44, 46, 52, 54, 62,
Mnemonics.....	16	Fuse read.....	11	69, 78, 88, 117, 130, 144, 167,
List of mnemonics.....	298	Fuse setting.....	46	173, 208, 271
Multiplication		Lock-bits.....	11	Subroutines (CALL/RCALL).....
16 bit by 4.....	87	Signature.....	10	Tables.....
8 bit by 8 bit.....	112, 113	Registers.....	30	109 in SRAM memory.....
		Preferred uses.....	297	in EEPROM memory.....
				110 in flash memory.....
				110