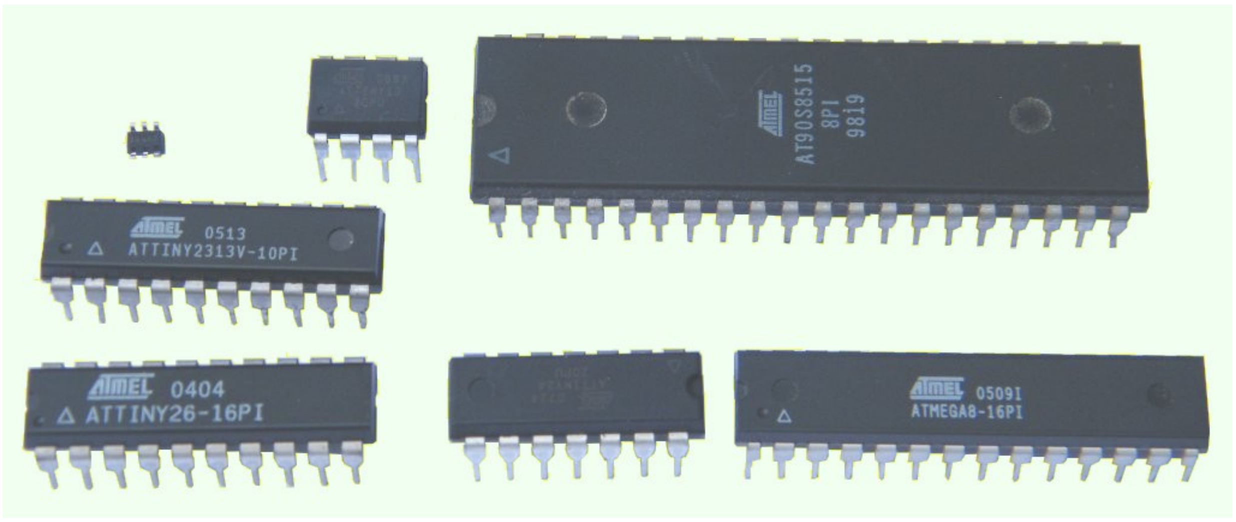
Learning AVR assembler - the very first steps



There are lots of short or long introductions to AVR assembler. The following was written for absolute beginners who want to know why they should learn assembler and what the first steps of learning this are.

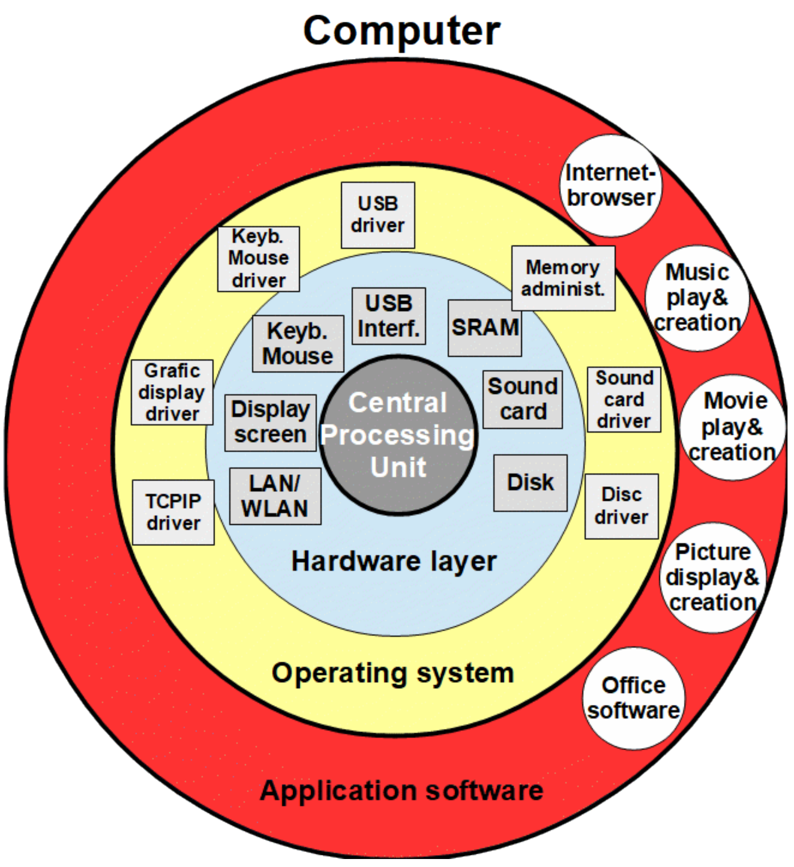
1st step: Forget all you know about programming that you learned so far

If you already know programming languages such as C, Pascal, diverse Basic dialects, Java or whatever: kick that all aside, it makes no sense in assembler and blocks your first steps in learning that. One basic rule in learning theory is: learning something that is really new starts with destroying previous or old knowledge. And even if it hurts: without that you'll make no progress in learning, instead you'll always will stay at: "Why wasn't that made like I always used to do things?" and "That's too complicated for me, I'll never learn that." Previous or old knowledge stands in your way and blocks you from learning new things.

So, let us first see what our standard view on computers is like. And then learn why micro-controllers are a different world.

1.1 Our standard approach to programming languages

Why that? Now, assembler is very, very different from other languages and even though programs that run on a micro-controller still are bits and bytes, but the generation process is very different.



These are the four levels of a usual computer. We usually see only the red level outside of the onion: we start an office software (that produces a picture of an empty sheet of paper on the screen), we click on a key on the keyboard and the letter written on the key appears in the picture. We deal with the outer level, what happens below that is in-transparent and we need not to know what goes on there as long as the application software works as desired.

Each of the above actions is associated with thousands of steps of the inner level, the Central Processing Unit or CPU. Our click onto the keyboard involves those thousands of steps, until the keyboard driver has found out which key has been pressed and what letter is written on the key. The keyboard driver hands out the letter to the application software, that stores it and sends 1,000s of commands to the display screen driver to change the picture of the empty sheet, that now shows the letter of the key pressed.

Fortunately the millions of CPU actions associated with that need by far less than a second so that we do not have to wait for long to see if it works.

1.2 Programming in high-level languages

So what changes if we do programming in C, Pascal, Basic or Java? We type in commands in that language in an editor, that stores a source code file on the disk. Then we call a so-called compiler, that reads the source code and translates it to a series of commands, packs those into an executable file, makes an application program from it and stores it on the disk. If we start that by clicking on it, the compiled commands are executed.

More than 90% of the executable are calls to lower levels such as to the operating system and very seldom to drivers. Some manipulate memory spaces, via the memory administration driver. Nearly none of those directly address the CPU. How the source code compiles is completely in-transparent for the programmer: he needs only to know if the produced executable works as desired, not how the executable does this. The compiler decides over this and does not ask the programmer if he should do it in this or another way.

Therefore usual programming in high-level languages does neither need to nor does it factually increase our knowledge and understanding of how the computer finally works. We are still dump users even though we know how to program (and with that solve our application needs).

Learning to program here means: to write well understood lines of source code to an editor file and leave it to the compiler to compose an executable with calls to the operating system. The millions of single steps that are necessary to do what the user expects remain in-transparent and need not to be understood.

1.3 The special world of the CPU

When you work with a computer, you see only the red level, built out of application programs. Sometimes you'll have to care on the operating system: if you set-up the computer for the first time, if changes are made by the system designer of the operating system. But most of the time you spend on a computer you are working with applications. Your knowledge on its operating system is very limited.

Contacts with the driver level are even less seldom. They happen if you change your hardware, In the most cases your contact with this level is installing a new driver for the new hardware. Nothing that urges you to really go deep into that level. So you learn near to nothing about that level. Those remain black boxes.

Even more a black box is the CPU in your computer. Even though it does all the hard work, you'll never use it on a direct way: Only all the other levels (the hardware drivers, the operating system and sometimes application programs access the CPU directly. You'll never see when those do it, because the CPO never asks you as user if she shall perform this or that step. It she would do that, you would see what she does. So she is not even a black box, she is simply nothing that you get aware of.

Micro-controllers are different: they consist of a CPU and some internal attached hardware. No drivers, no operating system, no application software but only what is written in its flash memory. They have nothing around themselves and are completely naked. That is why you have to forget all the things you learned about programming so far: The CPU is an own universe, has its own rules and functions in a very special manner. So all you know so far about programming makes no sense in this universe. All those programming commands you learned, all your knowledge on meaningful algorithms makes no sense in this special world.

2 The world of the CPU: how it functions

The center of all computers is the CPU: it does all the work. So we'll have to learn how this works first, before we will be able to program it. First we learn how all CPUs in the world operate, then we'll go into specifics of the AVR CPUs.

2.1 General functioning of all CPUs

All CPUs in whatever computer type function like this:

1. The CPU has a counter on board, which counts up and points to the current execution location. This is called program counter or PC. In the first step the CPU fetches command bits from this location, the source. Those can be portions of 8 bits (such as in an ancient 8080 CPU), 12 bits (such as in a PIC controller), 16 bits (such as in an AVR, an ancient 186 or Z80-CPU), 32 bits or even 64 bits. The source, where these bits are stored, is either a location in a RAM, but can be from other sources, too, such as an ancient Read-Only-Memory (ROM or EPROM).

2. The bit combination read is then decoded: certain bit combinations do add two numbers, another one subtracts, another one ANDs or ORs or EXORs them, etc. Some CPUs know also more complex operations, such as writing bit content to a location using a pointer register, and auto-incrementing that pointer register afterwards. What the CPU can do is called its **instruction set**. So decoding the input bits has to translate the instruction bits to CPU operations.
3. The decoded actions are then performed, e. g. two numbers are added and the result is written to a certain location. If during adding an overflow occurs, a flag called **CARRY** in the CPU's flag register is set to one, otherwise this flag is cleared.
4. The CPU then increases its PC and restarts with step 1 above.

Even though very different from their design, speed and capabilities all CPUs work in this way.

2.2 Specifics of the AVR CPUs

Now to the more specific properties of the AVR's CPU. The largest difference to all other CPU types is that the AVRs have a lot of registers: 32. Registers are bit storage locations with a size of 8 bits that the AVR CPU can directly use. So the CPU can add any one of the 32 registers with any other one (even with itself) and can store the result in any one of these 32 registers. So the bit combination of the instruction word (16 bits) for adding looks like this:

16-bit Opcode:

0000	11rd	dddd	rrrr

So, whenever the AVR CPU fetches 16 bits, of which the highest four are zeros and the next two following are ones, she knows what to do:

1. Convert the **r-rrrr** bits to a number that points to register 0 to 31 and place its content to its adder input.
2. Convert the **ddddd** bits to a number that points to another register between 0 and 31, place its content to the second adder input, and add these two bytes.
3. Write the result, the sum of both, to the register that **r-rrrr** points to by replacing its previous content.
4. If an overflow occurs (the result is larger than what can be fitted into an 8 bit register), the CPU's so-called status register SREG gets a one into its bit zero, called **CARRY** or "C" flag. Otherwise this bit is cleared. If the result is zero, it sets bit 1 of SREG. which is the **ZERO** or "Z" flag. Whatever follows can use those flags, as long as no other instruction alters those flags.

Don't be afraid, you don't have to learn all 16 bit opcodes. There are ways to simplify those: with mnemonics. The mnemonic for **0000,11xx,xxxx,xxxx** is simply **ADD**, the two registers **r-rrrr** and **ddddd** follow as parameters. The complete instruction would therefore be: **ADD r,d** with r and d representing numbers between 0 and 31. A simple rule: the first parameter is always the register where the result is written to!

AVR CPUs know more than 100 different instructions, each of those having a specific opcode (that the CPU can decode) and a mnemonic (that the human programmer can use). If you forgot a mnemonic you can search it in every data sheet of the respective AVR type in the chapter **Instruction Set Summary**, if you really need to know the binary opcode search for the document **Instruction Set Manual** on ATMEL/Microchip's website.

More specific to the AVRs is that the CPU can only fetch opcodes that are stored in the flash memory of the chip. No external or opcodes stored in SRAM or in EEPROM can be executed. So if you want the CPU to execute instructions, write those to the flash memory.

2.3 Assembler is the language of the CPU?

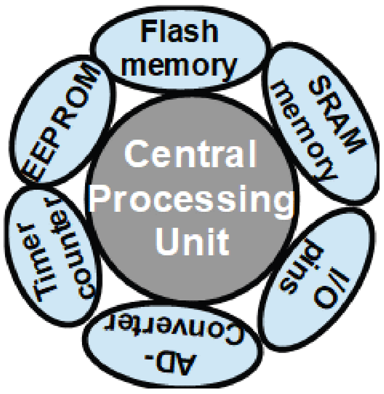
Well, yes and no. Exactly is that the 16-bit opcodes are the language of the CPU. The mnemonics, such as ADD, are assembly language. A program also named **assembler** converts the mnemonics to opcodes: it "assembles" the bits **0000,11** and the two numbers of the two registers in binary form and produces the binary output, to be programmed to the flash memory for execution by the CPU.

As the language assembler only consists of mnemonics for the CPU's binary opcodes, it is as near as possible to the CPU. So assembler is called a "hardware-near" language, in contrast to languages that are far away from the hardware, in their own world.

2.4 A controller instead of a computer

Other than the computer onion as shown above very different hardware is already integrated in a controller's CPU core:

- memory: flash memory for executable instructions, static RAM for storage of data and as stack, EEPROM for storing durable content,
- handling and controlling of in- and output pins,
- starting, stopping and controlling of timers and counters,
- enabling a comparison between two analog voltages and reading the comparison result as a bit,
- connecting input pins to an AD converter that converts an analog voltage to a number, using either the operating voltage, an externally supplied voltage or a constant internal voltage as reference,
- serial interfaces, such as synchronous or asynchronous communication devices, with the necessary baud rate generators and dividers,
- means to manipulate clock rates, via fuses or via programmable dividers,
- and much, much more.



Therefore our controller looks rather like this. It combines a CPU with the hardware layer, but lacks the driver, the operating system and the application software layer.

And, with all that hardware integrated, another feature of computers is already build into these controllers: an interrupt controller. This allows to enable interrupts for all integrated hardware, to prioritize those interrupts and to interrupt program execution whenever hardware needs the attention of the CPU. And this already very fast: within micro-seconds a reaction on a hardware event is possible. Try to do that with a PC or notebook: even though fast enough, the operating system is not designed to react fast on such events. You'll be lucky if it reacts in the milliseconds range, but not that fast like a controller. You have to go deep into the layers and to program on the driver level, if you need that.

What is unusual in computer programming is the standard in controller programming: you are programming the drivers for the hardware needed. But that is not very complicated, as you will see later on.

2.5 High-level language versus assembler

You can skip this section if you never learned a high-level language. The following is only for people who are socialized by such a high-level language, to name the major differences.

Nobody programs a PC or notebook in assembler. If you do that you are a pervert nerd or, at least, very special. Your operating system provides all you need, and you only have to call the appropriate routines. Not so in an AVR: there are neither drivers (for the hardware) nor an operating system nor available routines of whatsoever type. It is just a bare-naked controller doing nothing if not put alive by your software. So you'll have to do it by yourself, no system does it for you.

But doing it by yourself has immense advantages: not only can you exactly tailor the software to your needs but you are the complete master of the complete processes. Nothing wastes time or blocks your flash memory with unneeded trash. Fast and effective instead of nerdy and in-transparent.

And, the best is: With each program you write your knowledge on how the controller works increases and, byte-by-byte, a complete new world is at your command. Anything is allowed, no rules and "Don't do that"s limit your creativity.

Programming on the CPU's level is very different than programming on the basis of an operating system: by calling operating system routines more than 99% of that is unneeded trash or, even worse, simply forbidden.

Let us take an example. If we type in a source code in a high-level language, such as

```
C := A + B; or
C = A + B or
LET C = A + B
```

or under whatever language rule, the (C-, Pascal-, Basic- or whatever) compiler does the following:

- he checks of what type the variables A, B and C are (signed or unsigned integers with 8, 16, 24, 32 or 64 bits, fractional numbers with 16, 32 or 64 bit length), strings, characters, or whatever types the language knows and allows,
- he checks if the + operator is valid for A and B, otherwise throws error messages to the programmer,
- with the size of the variables he checks where A, B and C are located in his allocated memory,
- he checks if previous code has written something useful to A and B, otherwise throws error messages or warnings,
- writes a call to an appropriate addition function, that is able to add A and B of the identified type, and

- adds the step that stores the result in the memory storage place for the variable C.

The programmer does not need to know

- where A, B and C will be stored,
- what function exactly will be used for adding, and
- at which memory address the execution of adding happens.

All this is solely up to the compiler.

Not so in assembler. Your program looks like this:

```
; Add two 8-bit numbers in R1 and R2, result in R3
mov R3,R1 ; Copy #1 to R3
add R3,R2 ; Add #2 to R3 and store the result in R3
```

This is very different from C=A+B:

- You'll have to decide exactly where the variables A, B and C are stored, either
    1. in registers like here, of which AVRs have 32 8-bit called R0 to R31 (PICs have only one), and in which of those,
    2. in SRAM memory, and at which addresses: in that case the two numbers have first to be transferred to registers, because the CPU can only add registers.

  Wherever you store these, it is up to the programmer to decide. And it is his responsibility to protect this space, too. If not be needed any more, he can overwrite this. Nobody and no compiler warns you if you overwrite the content of this storage space. The freedom of placement is associated with the responsibility to protect (as long as needed) and the risk of erroneous actions.
- While in the high-level language the "+" initiates adding, you'll have to write "ADD", which is a CPU instruction. While behind "+" a large number of CPU instructions can follow, depending from the size and type of A, B and C, the "ADD" here is exactly and only one instruction or CPU command, nothing else than that.
- If you'll add two 8 bit numbers, your result can be larger than 255. That would require an additional register for the result because it would not fit into only one. So either reserve R4 for that or do something else if the result is larger than 255. If you reserved R4 for that, just insert the instruction **clr R4** after the **mov R3,R1** and add **adc R4,R4** at the end and your source code above looks like that:

  ```
  ; Add two 8-bit numbers in R1 and R2, result in R4:R3
  mov R3,R1 ; Copy #1 to R3
  clr R4 ; Clear the upper byte of the result
  add R3,R2 ; Add #2 to R3 and store the result in R3
  adc R4,R4 ; Add the carry flag to the empty upper byte
  ```

  Please do not ask me why the mnemonic for copying a register is not **COPY** but **MOV**, even though nothing is "moved" here (the source will not be empty, other than the term "move" implies).

- In a high-level-language the try to store a result larger than 255 into a variable defined as an 8-bit byte throws an exception (overflow). If so enabled, the exception stops the program from further execution and displays an error message. It is up to the programmer, in a high-level language as well as in assembler, to care for such overflows. But in assembler nothing happens if you ignore a carry flag being set by an ADD instruction: nothing happens if you do not program that. If you want to stop further execution you'll have to program that by yourself, because there is no overlying operating system that can display an error message.

So the basic differences between high-level languages and assembler are:

1. Assembler programmer can only use instructions that the CPU knows and performs. More complex functions, such as adding two 64 bit numbers, have to be written by the assembler programmer.
2. Locations of variables have to be explicitly defined in assembler. Only the content of registers can be treated by the CPU, the register location has to be explicitly defined in the instruction. No unknown, undefined or in-transparent positioning of variables in assembler.
3. Any available storage space can be used by the assembler programmer, the responsibility to protect content from overwriting remains with the programmer, no protection is automatically working.
4. Storage spaces can be combined in any possible manner in assembler: if you need eight bytes for a 64-bit number, just place them either in eight registers or in eight SRAM storage cells. Row and content are up to you, no compiler disturbs your own design considerations. In high-level languages the programmer has no choice, he has to take what he gets and has no chance to decide something.
5. A central category in high-level languages are types of variables and constants. Those type definitions limit automatically what can be done with a variable. Assembler does not know any types of variables or constants, all are bits and bytes. No limitations are given on what can be done with those bits and bytes. The freedom to add three to an ASCII character stored in a register, by that making a 'D' from an 'A', is a simple one-line instruction in assembler. Try to formulate this in a high-level language, you'll end up in an orgy (Pascal: c:=SUCC(SUCC(SUCC(c))) or c:=CHAR(BYTE(c)+3))!. In assembler this is simply **SUBI R16,-3** if the ASCII character is in register R16. In AVR assembler there is no ADDI, so you'll have to
6. All protectional features, such as detecting overflows, under-flows, division by zero, etc., have to be explicitly programmed by the assembler programmer. This is not a large difference as high-level languages provide tools to cope with that, but most programming languages just stop their execution, should this occur.

These here are the No-No's in assembler - forget these high-level constructions for now:

- High-level languages such as Java, C or Pascal allow to define functions, allow to call those with parameters in brackets and those return a result in one of the types of the language. Simply forget this: a CALL to a function in assembler can use any storage location, and the result of the CALL can also use any location.
- We already learned that type definitions make no sense in assembler because anything is in bits and bytes in the different locations. All structuring of data has to be made by the assembler programmer, and if he cares not enough for his own rules, he alone suffers from bugs. So forget type definitions.

- Also forget all private declarations: anything is always accessible in assembler. Privacy have to be made by the assembler programmer. If he does not care, it is his own fault. No compiler or operating system warns him if he accesses those areas. This is the price of freedom.
- Also forget any collections of constants, variables, routines, virtuals and so on as records or classes. All this makes no sense in assembler, so better forget it now. If you think you need something like that: you'll have to design that on your own.
- In assembler only unsigned or signed binary numbers are standard. On the other hand: any size of integers can be handled, as long as you write the math for them (even 64-bit). Floating point math is also possible, but you'll have to write all basic routines by yourself. Some of the AVRs provide hardware multiplication for floats, but for mantissas only.

With that, 90% of the language construction of high-level languages is obsolete and makes no sense in a micro-controller. So better forget these constructions. Any methods to avoid Spaghetti code in assembler are always highly welcome, but high-level languages are inappropriate means, do more confusion than they provide assistance in that.

In the contrary: in high-level languages the normal starting sequence of new projects is to start with parts of the code and after that adding additional code and increasing complexity. It is exactly that work procedure that leads to Spaghetti code in assembler. Better start with planning in a depth that is appropriate for the complexity of what you plan to have at the end. So better also forget this work procedure, too.

Another example to demonstrate the different approach in assembler. High-Level languages allow the use of **FOR** and **NEXT** to construct counting loops. To use those you'll have to define

- a variable for the counter,
- a counting rule (plus one, plus two, minus one or minus two),
- an end condition (a constant or a variable) for ending the loop.

The same in assembler:

- the counter can be an 8-bit register or a 16-bit double register,
- increasing the counter can be done by **INC** (increase by one), decreasing by **DEC** (decrease by one), but you can use any other method to advance the counter: adding or subtracting another register with **ADD** or **SUB**, **ADIW/SBIW** for adding or subtracting a constant between 0 and 63 to a 16-bit double register, or more exotic methods like shifting the counter left with **LSL** (multiply by two) or right with **LSR** (dividing by two), anything goes,
- to test if the endpoint of the loop has been reached, use either the SREG flags (e. g. the Z flag when down-counting) or compare instructions like **CP** (compare two registers and clear/set the Z and C flags in SREG) or **CPI** to compare a registers with a constant,
- jumping back to the loop's beginning is done with the conditional jump instructions **BRNE** (jump back if the Z flag is clear), **BREQ** (jump back if the Z flag is set), or with the C flag **BRCC/BRCS**.

As you can simply see, anything, and even more than that, can be done in assembler, too. More than that means: you can change the step width in between, you can manipulate the counter in between the loop (which is not allowed in several high-level languages), you can test if additional conditions are met (e. g. if a port-pin has changed polarity in between) that should stop the loop, you can even jump to any location outside the loop should a certain condition require that. Even if your loop is nested (another loop inside an outer loop) you can jump to where you want in assembler. All features are allowed in assembler, no limitations. But: increased freedom means increased responsibility. If your counter counts with zero steps up or down, the AVR will accept that and your loop will never end. It is up to you to find that bug and replace it with something that functions well.

As you can see: better forget the FOR/NEXT construction and use whatever is the appropriate method for your needs.

3 Simulating the hardware of micro-controllers

As AVRs do not have hardware that can output a screen picture we can hardly see what happens if we program our first assembler routines. So we have to have another way to see what is going on.

3.1 The hardware to be simulated

As we saw a micro-controller (here some older types) is a little bit different from a computer: it is a naked CPU with some additional hardware packed around it. Small storages, such as a flash memory with a few kilo-bytes or a static RAM with also a few bytes or kilo-bytes, and an even smaller EEPROM with only a few bytes. But: all that needs less than one per-mill of the current or power of a computer, and can be operated for hours, days and months from a small battery.

The available hardware consists of

1. a flash memory storage for the running program, and
2. a SRAM memory that looses its content when the operating voltage falls below a minimum value (in that case all bits will fall back to ones), and
3. an EEPROM storage, that keeps its content for some years even in absence of an operating voltage, and
4. lots of additional small machines such as timers/counters, AD converters, etc., and first of all
5. a rather large number of pins that can be switched on and off by the program in the AVR's flash memory.

By means of the last named hardware we could place information (bits and bytes) on these pins and by adding some LEDs and resistors we could see if a bit is zero or one. But that is not a very transparent mode, e. g. because we do not see very fast changes and we cannot reduce the speed of the controller.

3.2 A simulator as a do-as-if-controller

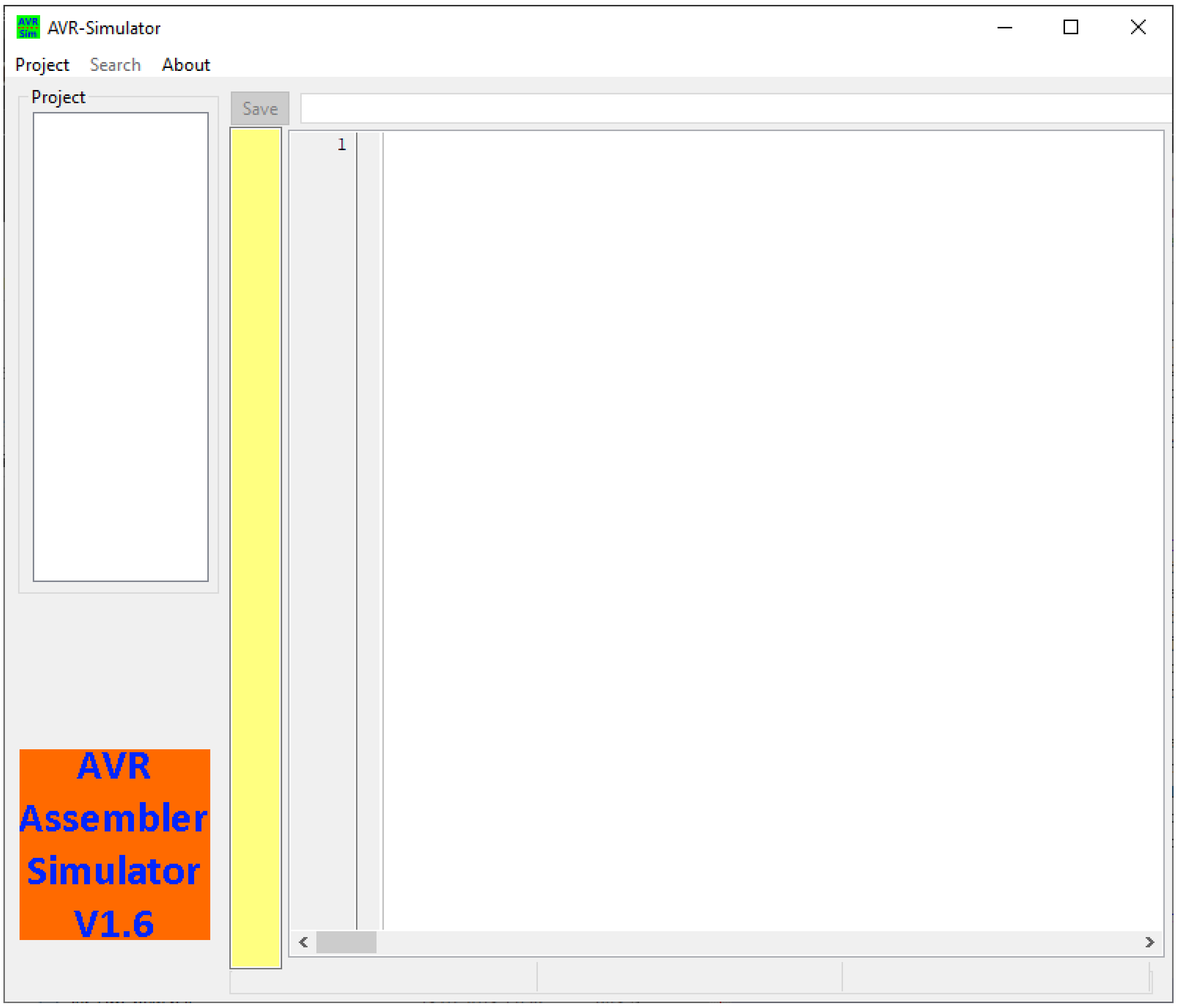
A simulator is an application program for a computer that behaves like a micro-controller. Other than the original, it allows to single-step through the assembler program: it executes an instruction, sets CPU flags, displays register content and displays SRAM or EEPROM content, and then stops. Before doing the next step we have all the time we need to read and understand what the instruction has done.

The simulator avr_sim is available for free, can be downloaded as executable for 64-bit Linux and Windows versions or can be compiled using Lazarus from the Pascal source code that is also provided for many other operating systems. Each version that can be downloaded (just use the latest one) comes with a handbook that describes all features of the simulator. So if you need other information than described here, consult the handbook.

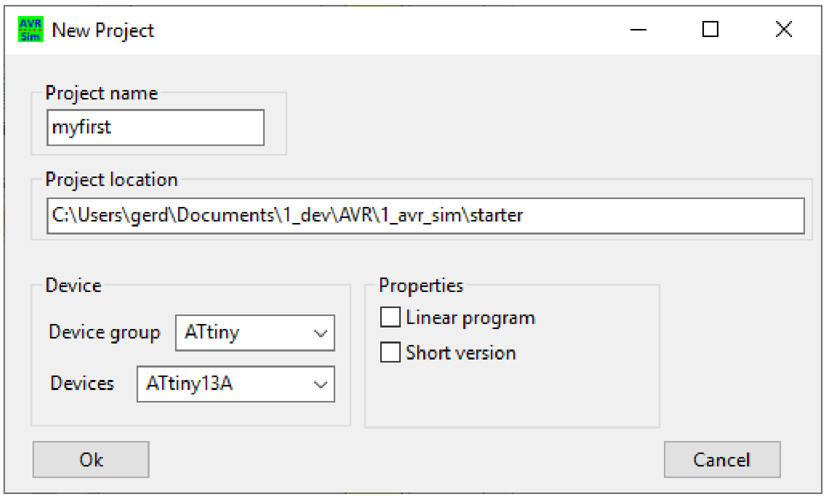
One special thing to learn here is that information on registers and other controller internals is displayed in hexadecimal format. A hexadecimal digit takes four single bits and displays those as 0 to 9 (0000 to 1001) or as A to F (1010 to 1111). So 16-bit numbers are represented by four hexadecimal digits. Use a programmer calculator to convert hexadecimal to decimal if you need it.

3.3 Our first assembler program

Start avr_sim, it should come up with an empty editor field to the right, with an empty project field to the left and a menu on the top.



Select the **New** entry below the **Project** menu entry and a window pops up.



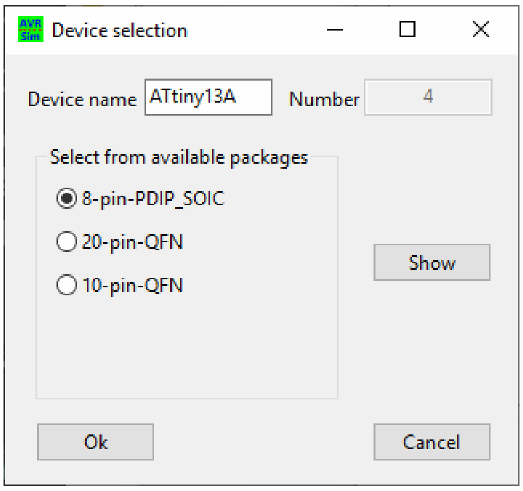
Project name: myfirst

Project location: C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\starter

Device — Device group: ATtiny; Devices: ATtiny13A

Properties: ☐ Linear program; ☐ Short version

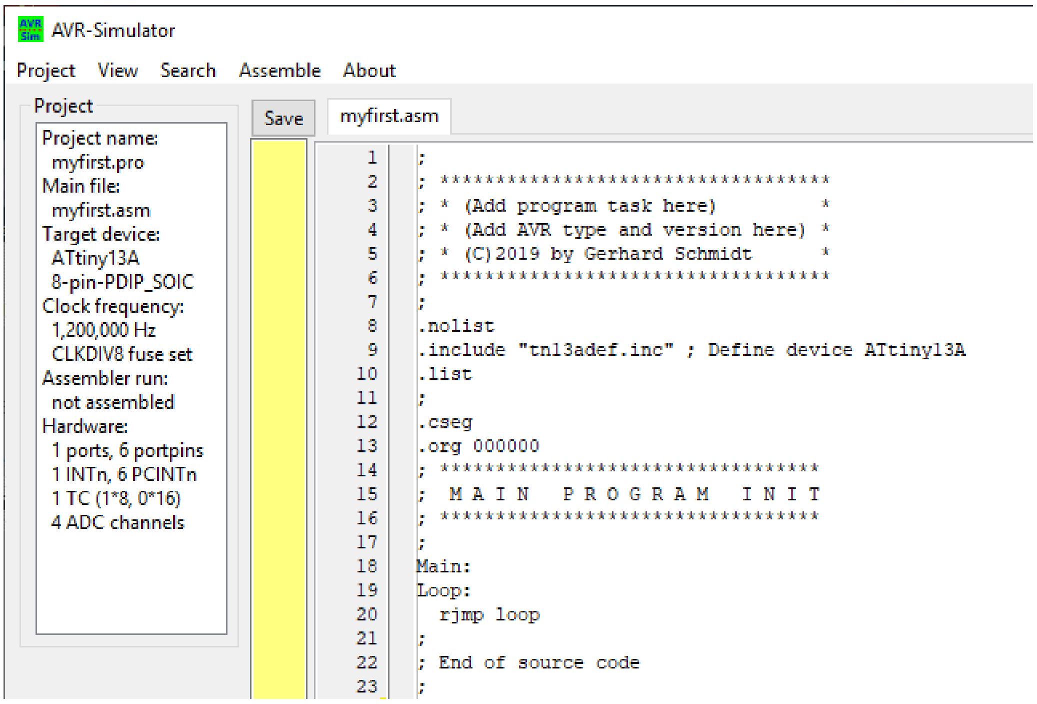
In this window, input a project name, navigate to the folder where the project shall be stored (click into the edit field and navigate through the folders, make a new one or select an existing folder), in the Properties section disable Interrupts and Comprehensive version, from the drop-down field select ATtiny and ATtin13A. Now close the window with Ok. After that another window opens, asking you for the device package. Select one of the three, it doesn't matter which.



Device name: ATtiny13A; Number: 4

Select from available packages: [✓ 8-pin-PDIP_SOIC] 20-pin-QFN 10-pin-QFN

After that the project window shows some properties of the project and a standard source file is displayed in the editor part of the window.



Project

Project name:
 myfirst.pro
Main file:
 myfirst.asm
Target device:
 ATtiny13A
 8-pin-PDIP_SOIC
Clock frequency:
 1,200,000 Hz
 CLKDIV8 fuse set
Assembler run:
 not assembled
Hardware:
 1 ports, 6 portpins
 1 INTn, 6 PCINTn
 1 TC (1*8, 0*16)
 4 ADC channels

```
;
; ***********************************
; * (Add program task here)         *
; * (Add AVR type and version here) *
; * (C)2019 by Gerhard Schmidt      *
; ***********************************
;
.nolist
.include "tn13adef.inc" ; Define device ATtiny13A
.list
;
.cseg
.org 000000
; **********************************
;  M A I N   P R O G R A M   I N I T
; **********************************
;
Main:
Loop:
  rjmp loop
;
; End of source code
;
```

This shows some lines that start with a ";": those lines are pure comments and are ignored by the assembler. But not only at line start: whenever the assembler sees a ";", the rest of the line is ignored.

The first lines that are not comments are the following:

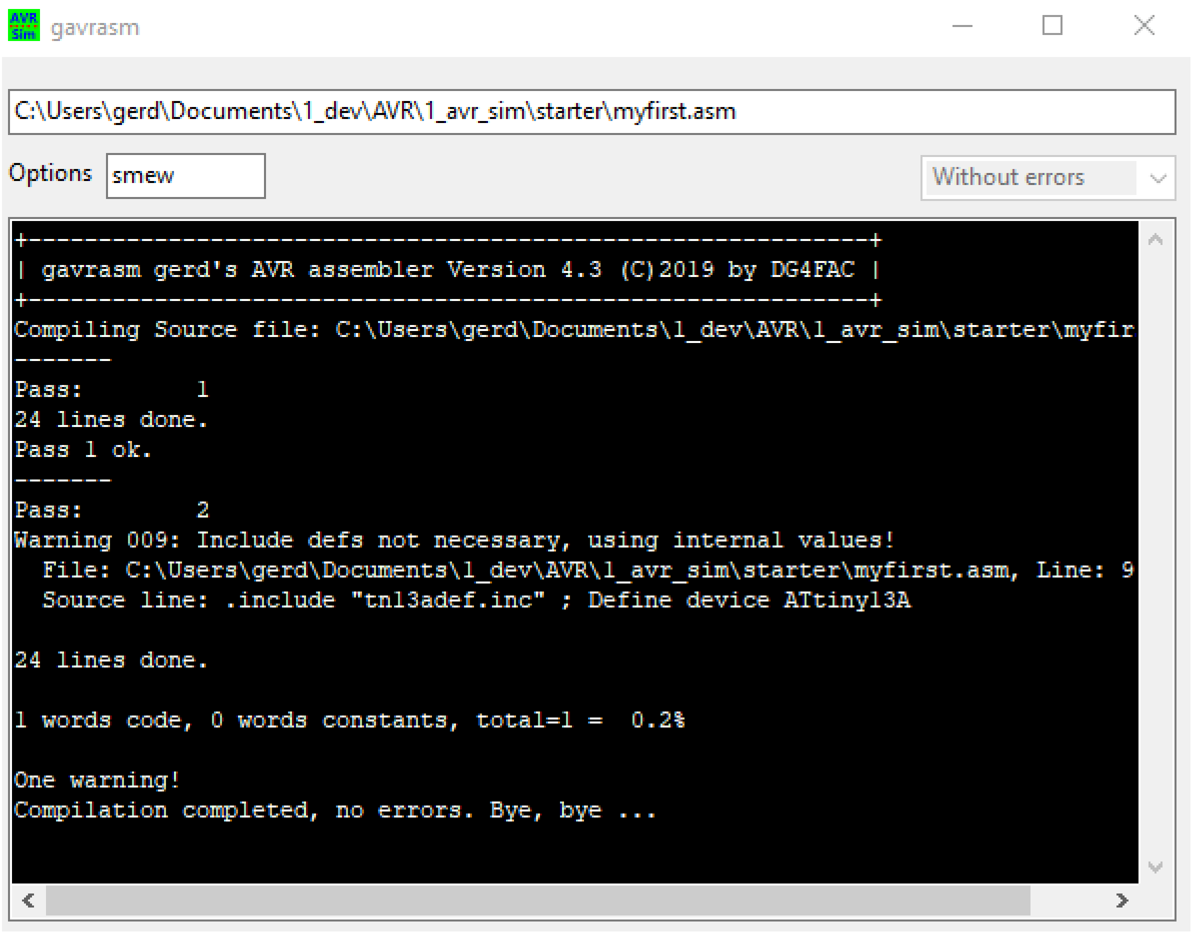
```
.nolist
.include "tn13adef.inc" ; Define device ATtiny13A
.list
```

These are three directives, starting with a ".": those are commands for the assembler. The first switches the output in the list file off, the third one on again. The second directive includes a file named "tn13adef.inc". This file holds addresses and constants for the device and switches a check on if the instructions are valid in that device type. But the file is not really read in avr_sim: all those files are implemented in avr_sim, so no extra file is necessary.

After that two additional directives follow: **.cseg** instructs the assembler to produce code (code sgment), **.org 000000** set the address for which the code will be assembled. As the device starts at address 0 when the operating voltage is applied, this ensures that our code is executed.

In the **Main program** section, the first entry is a label: **Main:**. Such labels all end with a ":" and tell the assembler to associate the label's name with the current address (here: 000000).

The label is followed by another label called **Loop:** and a line saying **rjmp loop**. This construction is an indefinite loop: the **rjmp** means "Relative Jump" and causes the controller to jump back to the label **Loop**. The **rjmp** is a jump instruction. We assemble this source code by clicking on the **Assemble** entry on the menu line.



C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\starter\myfirst.asm

Options smew

Without errors

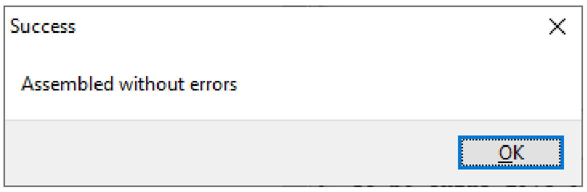
```
+------------------------------------------------------------+
| gavrasm gerd's AVR assembler Version 4.3 (C)2019 by DG4FAC |
+------------------------------------------------------------+
Compiling Source file: C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\starter\myfir
-------
Pass:        1
24 lines done.
Pass 1 ok.
-------
Pass:        2
Warning 009: Include defs not necessary, using internal values!
  File: C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\starter\myfirst.asm, Line: 9
  Source line: .include "tn13adef.inc" ; Define device ATtiny13A

24 lines done.

1 words code, 0 words constants, total=1 =  0.2%

One warning!
Compilation completed, no errors. Bye, bye ...
```

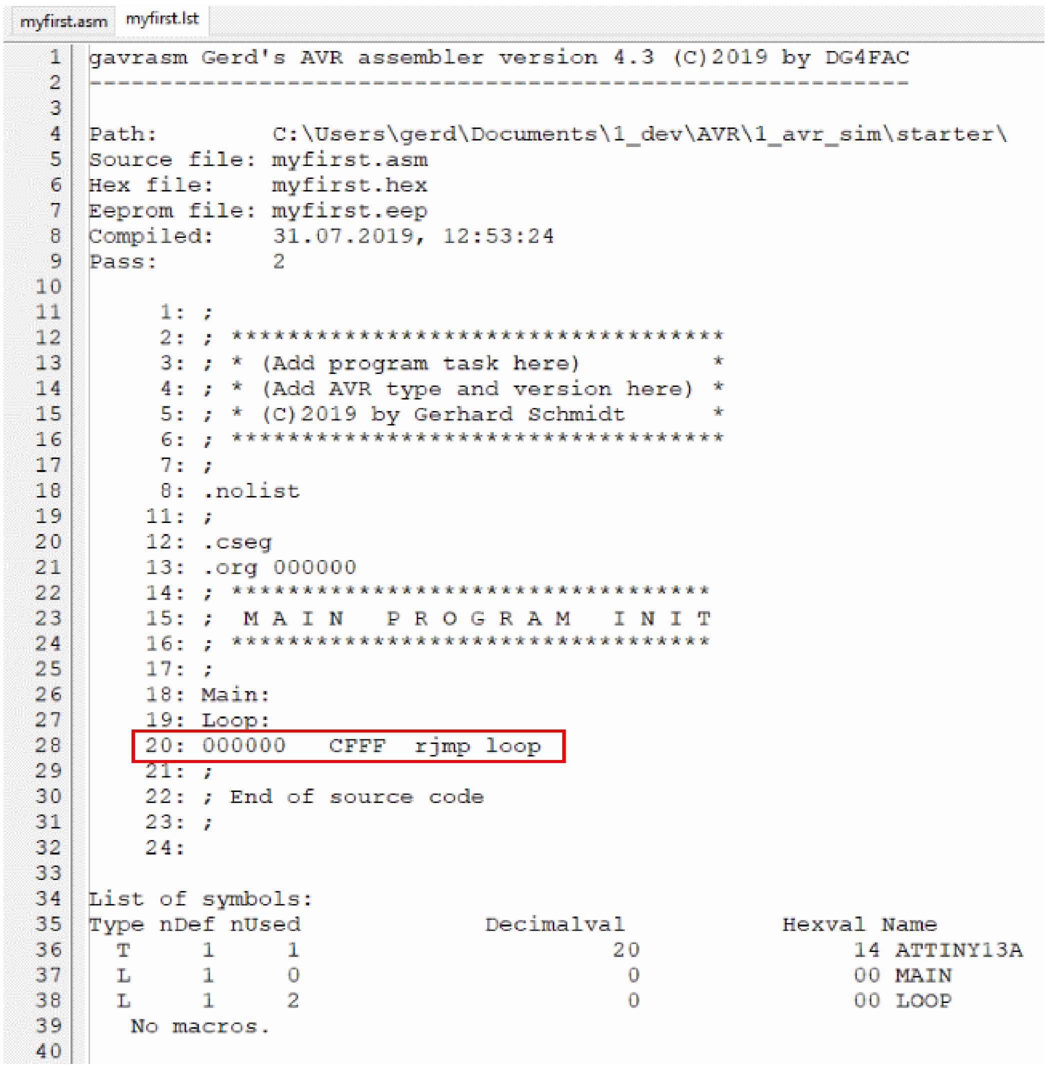
Assembling opens a window, that displays the progress during assembling. Here, the gavrasm assembler is at work. You can ignore the displayed messages, but not the following window that reports success.



```
Success

Assembled without errors
```

The editor window now displays the listing that the assembler produced. It holds all relevant information. The most relevant here is the line that is in the red box: the **rjmp loop** has been translated by the assembler to an instruction at address 000000. The instruction is hexadecimal CFFF.



```
gavrasm Gerd's AVR assembler version 4.3 (C)2019 by DG4FAC
----------------------------------------------------------

Path:         C:\Users\gerd\Documents\1_dev\AVR\1_avr_sim\starter\
Source file:  myfirst.asm
Hex file:     myfirst.hex
Eeprom file:  myfirst.eep
Compiled:     31.07.2019, 12:53:24
Pass:         2

     1: ;
     2: ; **********************************
     3: ; * (Add program task here)        *
     4: ; * (Add AVR type and version here) *
     5: ; * (C)2019 by Gerhard Schmidt     *
     6: ; **********************************
     7: ;
     8: .nolist
    11: ;
    12: .cseg
    13: .org 000000
    14: ; **********************************
    15: ;  M A I N   P R O G R A M   I N I T
    16: ; **********************************
    17: ;
    18: Main:
    19: Loop:
    20: 000000   CFFF  rjmp loop
    21: ;
    22: ; End of source code
    23: ;
    24:

List of symbols:
Type nDef nUsed             Decimalval           Hexval Name
  T     1     1                     20               14 ATTINY13A
  L     1     0                      0               00 MAIN
  L     1     2                      0               00 LOOP
   No macros.
```

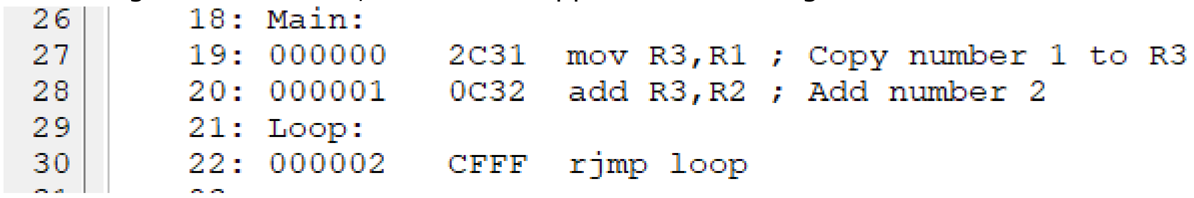
If we would program the hex file myfirst.hex to the flash memory of a device, the controller forever jumps back to address 000000, and does nothing else.

In case you need it: the symbol table at the end of the listing shows all relevant information on constants, labels and all other symbols. The labels Main and Loop, for example, point to address 000000.

To add another instruction before the controller is send into the indefinite loop, we add the two lines before the loop label:

```
mov R3,R1 ; Copy number 1 to register R3
add R3,R2 ; Add number 2 to register R3
```

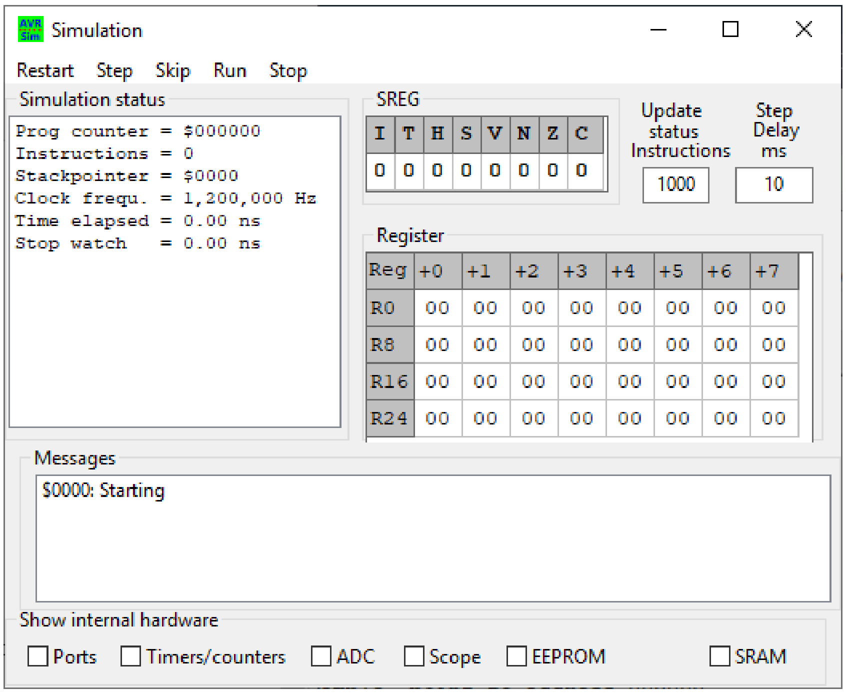
This copies the content of register R1 to R3 and adds the content of R2. After re-assembling the changed source code, these lines appear in the listing:



```
18: Main:
19: 000000   2C31  mov R3,R1 ; Copy number 1 to R3
20: 000001   0C32  add R3,R2 ; Add number 2
21: Loop:
22: 000002   CFFF  rjmp loop
```

The two instructions added increase the number of instructions to three. We can now simulate the execution of those instructions. To do that we click in the menu line. A new window opens.

That window shows the current status of the simulation, the flags of the CPU in the status register and the content of the 32 registers.



```
Prog counter = $000000
Instructions = 0
Stackpointer = $0000
Clock frequ. = 1,200,000 Hz
Time elapsed = 0.00 ns
Stop watch   = 0.00 ns
```

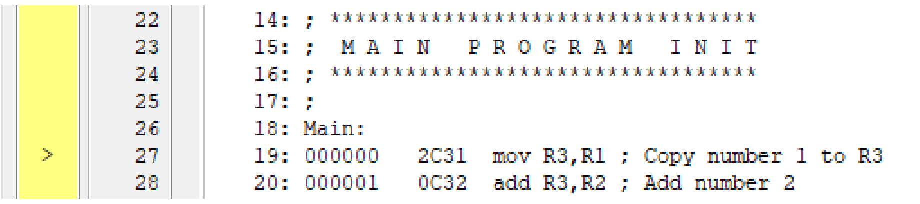
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

Update status Instructions: 1000 — Step Delay ms: 10

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

```
$0000: Starting
```

In the meantime, in the list view of the editor window, a ">" points to the first executable instruction in the line **mov R3,R1**.



```
    14: ; **********************************
    15: ;  M A I N   P R O G R A M   I N I T
    16: ; **********************************
    17: ;
    18: Main:
>   19: 000000   2C31  mov R3,R1 ; Copy number 1 to R3
    20: 000001   0C32  add R3,R2 ; Add number 2
```

That is the instruction that will be executed when we now click **Step** in the simulation window.

If we do that we'll see nothing but an increasing value of the program counter and the number of executed instructions. The reason for that is that all registers are set to zero when the controller re-starts. And 0 plus 0 yields zero, and that was exactly the content of R3 after executing the first instruction and therefore is not highlighted. So even doing step 2 does not change much.

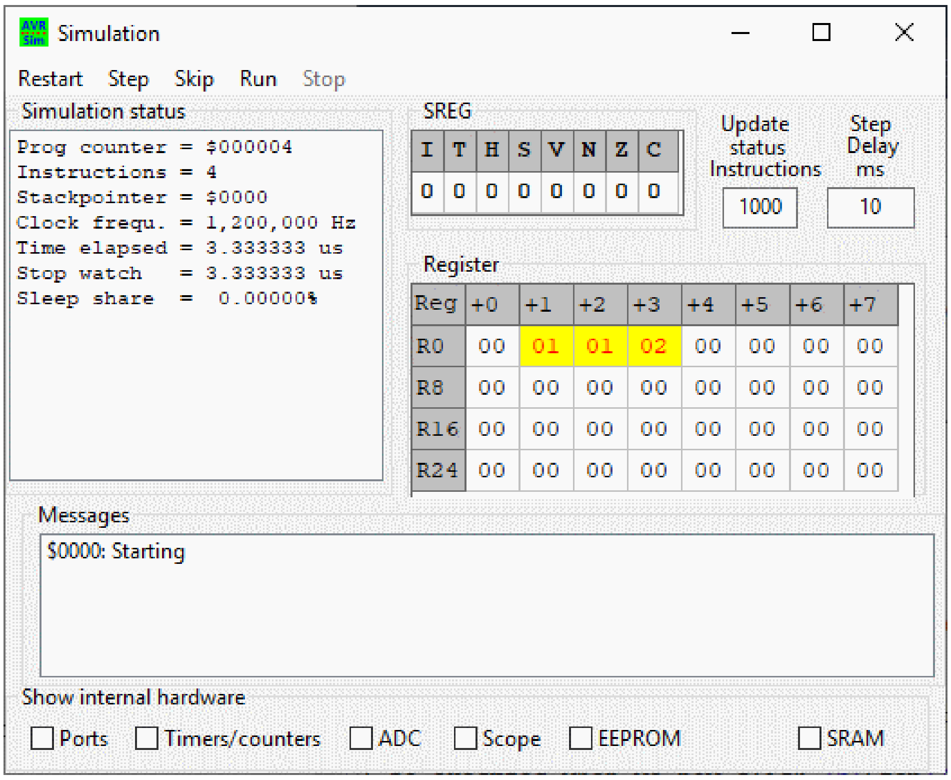
This changes if we add the two instructions before the **MOV** instruction:

```
inc R1 ; R1 to one
inc R2 ; R2 to one
```

The instruction **INC** increases the content of the given register by one. As it was zero, a one is now in R1 and R2.

The registers now show that their content has changed: R1 and R2 are now one, the adder result is 2.

As each step sets the previous value the fact that all three registers are yellow was reached by setting a breakpoint on the rjmp (set the cursor to that line, right-click and select **Breakpoint**, the breakpoint is displayed as **B** in that line) and starting the simulation with the Run entry in the menu. So all changes to registers remain yellow.



```
Prog counter = $000004
Instructions = 4
Stackpointer = $0000
Clock frequ. = 1,200,000 Hz
Time elapsed = 3.333333 us
Stop watch   = 3.333333 us
Sleep share  =  0.00000%
```

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

Update status Instructions: 1000 — Step Delay ms: 10

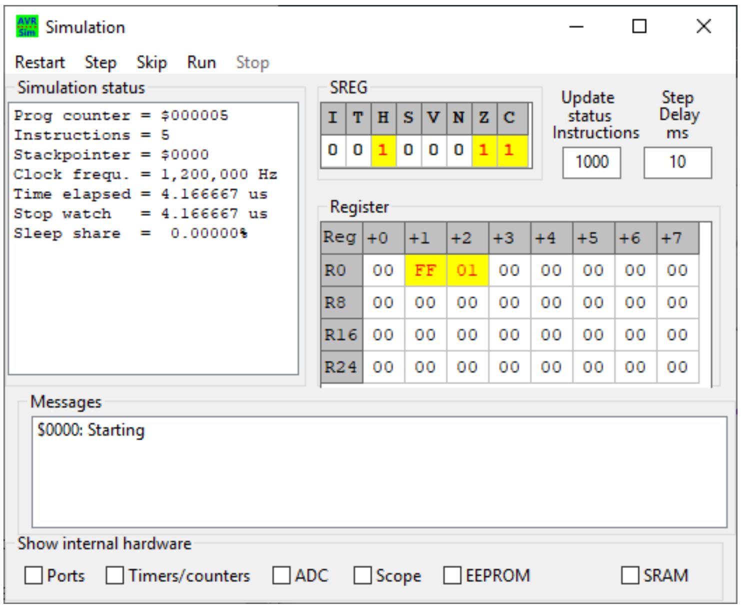
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	01	01	02	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

$0000: Starting

3.4 Using the flags

If we exchange the line **inc R1** by **dec R1** we provoke some new effects. The **DEC** decreases the content of register R1. The zero yields a 255 or hexadecimal FF. As the result will be larger than 255 we also add R4 for the higher byte. The source code now:

```
  dec R1 ; Number 1 to 255
  inc R2 ; Number 2 to 1
  clr R4 ; Result MSB to zero
  mov R3,R1 ; Copy number 1 to R3
  add R3,R2 ; Add number 2
  adc R4,R4 ; Add carry to MSB
Loop:
  rjmp Loop
```



```
Prog counter = $000005
Instructions = 5
Stackpointer = $0000
Clock frequ. = 1,200,000 Hz
Time elapsed = 4.166667 us
Stop watch   = 4.166667 us
Sleep share  =  0.00000%
```

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

Update status Instructions: 1000 — Step Delay ms: 10

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

$0000: Starting

The **CLR** instruction clears the register to all zeros.

If we now execute the first five instructions, before the adc, we'll get the following register content.

The two numbers FF and 01 are ok, adding those has yielded zero in R3. But: in contrary to our previous adding operations the C flag in the status register SREG is now 1, indicating that the adding has caused an overflow or **CARRY**. And: as the result is zero the Z (or zero) flag is also set one.

The next instruction, **adc R4,R4** now adds the content of R4 with itself (which is zero) and the carry flag. As the carry flag is set, the result is 1.

That is what the flags are for: they indicate relevant states that the CPU detects, but can also be set or cleared by the programmer (the carry flag with the instructions **SEC** or **CLC**).

Another use of the flags are conditional branches. The instructions **BRCS label** or **BRCC label** jump to a label if the carry flag is one (SET) of zero (CLEAR). The same example can also be formulated like this:

```
  dec R1 ; Number 1 to 1
  inc R2 ; Number 2 to 1
  clr R4 ; Result MSB to zero
  mov R3,R1 ; Copy number 1 to R3
  add R3,R2 ; Add number 2
  brcc Loop ; Branch if carry is clear
  inc R4 ; Increase MSB
Loop:
        rjmp loop
```

That has the same effect: the increase of the MSB is not executed if the carry flag is not set.

Note that not all instructions change flags. If and which flags are altered by the CPU can be seen from the Instruction Summary in the data-book of the device. So, e. g. the INC instruction sets the Z flag, but not the C flag. Consult the data-book or the Instruction Set Manual if you are not sure or in doubt.

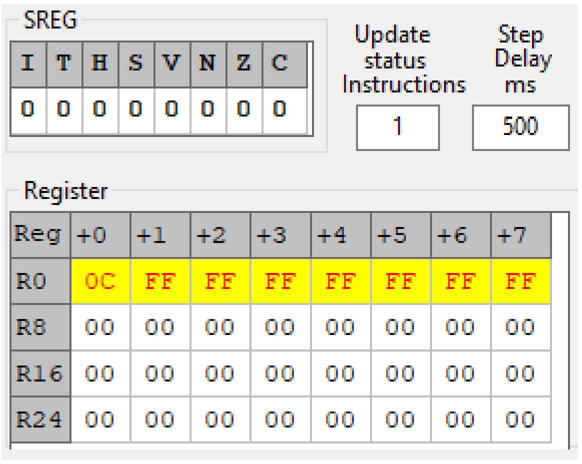
I	T	H	S	V	N	Z	C
–	–	–	⇔	⇔	⇔	⇔	–

With these instruments now learned we can easily construct a 64 bit counter. We increase the lowest byte in R1 and each time, when the zero flag is set, we increase the next higher byte in R2, too. And so on for the bytes up to R8. The source code for that:

```
Count:
  inc R0 ; Increase the lowest byte
  brne Count ; If Z flag is clear count on
  inc R1 ; Increase the second byte
  brne Count ; If Z flag is clear count on
  inc R2 ; Increase the third byte
  brne Count ; If Z flag is clear count on
  inc R3 ; Increase the fourth byte
  brne Count ; If Z flag is clear count on
  inc R4 ; Increase the fifth byte
  brne Count ; If Z flag is clear count on
  inc R5 ; Increase the sixth byte
  brne Count ; If Z flag is clear count on
  inc R6 ; Increase the seventh byte
  brne Count ; If Z flag is clear count on
  inc R7 ; Increase the highest byte
  rjmp Count ; Count on
```

If we now assemble and start the simulation, we can set the value in **Update status** to 1 instruction and the **Step Delay** to 500 milliseconds we can see our counter at work, slowly increasing its values in registers R0 to R7.

As you'll need 0.5*256*256*256*256*256*256*256*256 seconds (more than 292 billion years) to advance the counter's 64 bits to 0 again, this counter is virtually infinite. Even at the highest simulation speed the end is not reached, so I used a trick to come to this displayed count. Guess what I did.



SREG

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

Update status Instructions: 1

Step Delay ms: 500

Register

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

The load instruction, that sets a register value to a given constant value, is **LDI r,decimal**. But, like a few other instructions, only works with registers R16 and higher. So, to set R1 to 255, use a register from R16 upwards to load the constant (here as hexadecimal formatted number) and then with **MOV** to copy it to R1, like this:

```
 ldi R16,0xFF ; Load 255 decimal to register R16
 mov R1,R16 ; Copy the content of R16 to register R1
```

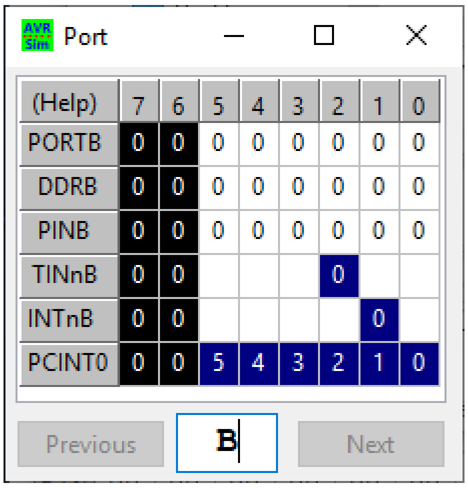
You can also use the LDI instruction to set the value in binary format, if you formulate **ldi R16,0b11111111**.

Another opportunity is to not use the register with its number but to define a name that represents that register. So you are more flexible when it comes to re-arrange the 32 registers. Just define the name of the register with the assembler directive **.def rMyReg = R16** and use that register to set it to 255: **ldi rMyReg,255**. The assembler knows this name now and displays it as type **R** in the symbol list at the end of the listing.

Six of the registers have already such an alias name: R26 is named XL, R27 is XH, R28 is YL, R29 is YH, R30 is ZL and R31 is ZH. Those six registers are very often used as 16-bit pointers, so the lower byte is in L and the higher in H. A few instructions allow 16-bit wide operations, such as **ADIW ZL,1** or **SBIW XL,1** to increase and decrease the 16-bit value, automatically updating the MSB if adding or subtracting yields an over- or underflow to the MSB.

4 Manipulating I/O pins

Now we switch some external pins of the controller on and off. Each AVR type has such pins that can be manipulated. In the simulator window click on the **Ports** selection field to take a look at those. For the start look at the **PORTB**, the **DDRB** and the **PINB** lines and ignore the other lines.



Port

(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

4.1 Setting and clearing single I/O bits

An I/O pin can be input or output. This is controlled by clearing or setting their bit in the port register **DDRp**. As the ATtiny13 has only one I/O port and only five I/O pins, you can use the following instructions:

```
Loop:
  cbi DDRB,DDB0 ; Clear direction bit of pin PB0
  sbi DDRB,DDB0 ; Set direction bit of pin PB0
  rjmp Loop
```

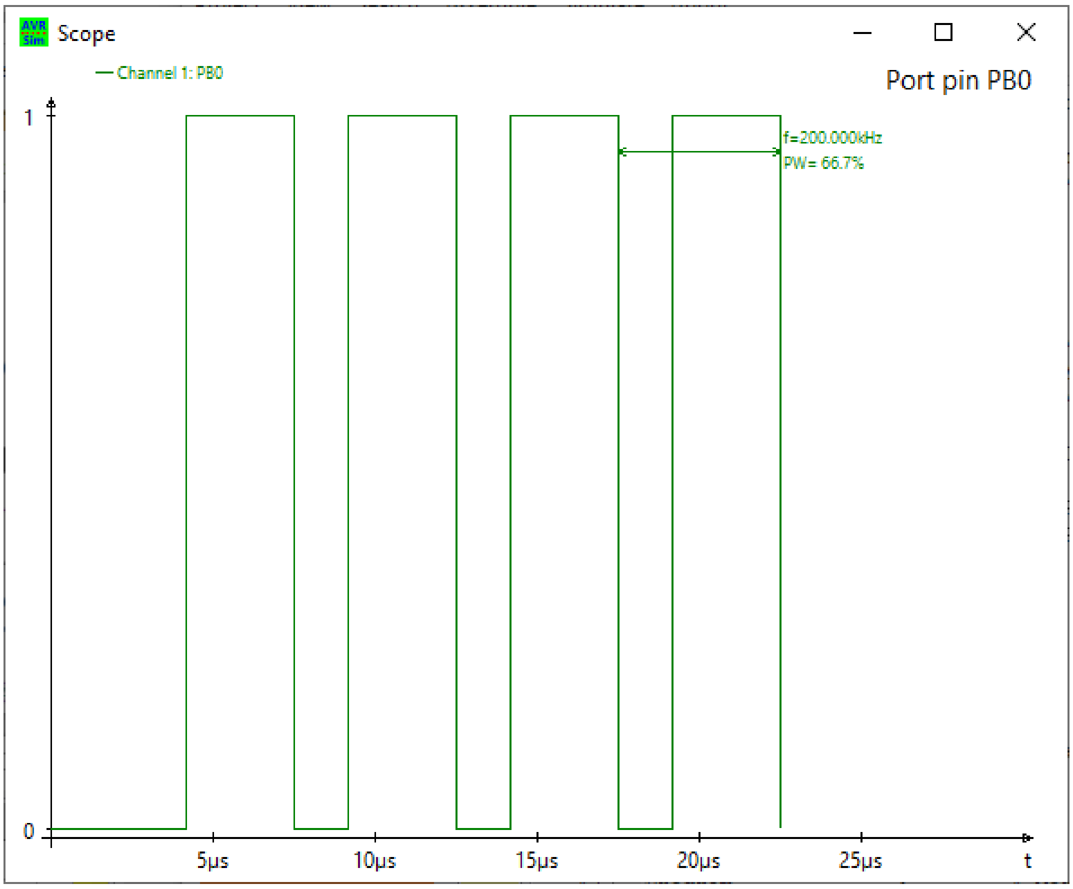
This makes the external pin PB0 first input (**CBI** means Set Bit I/O) and then output (**CBI** means Clear Bit I/O), then restarts again. If you would attach a LED to external pin PB0 and tie it with a resistor of a few 100 Ohms to the operating voltage, the LED would go on (as long as the output is active) and off (as long as the output is not active). If you assemble that code, simulate it step-by-step you'll see the effect in the port view window.

A second possibility is to make PB0 an output permanently and to clear and set the PORTB bit. The source code for that:

```
  sbi DDRB,DDB0 ; Make PB0 an output
Loop:
  cbi PORTB,PORTB0 ; Make PB0 low
  sbi PORTB,PORTB0 ; Make PB0 high
  rjmp Loop ; Repeat all over
```

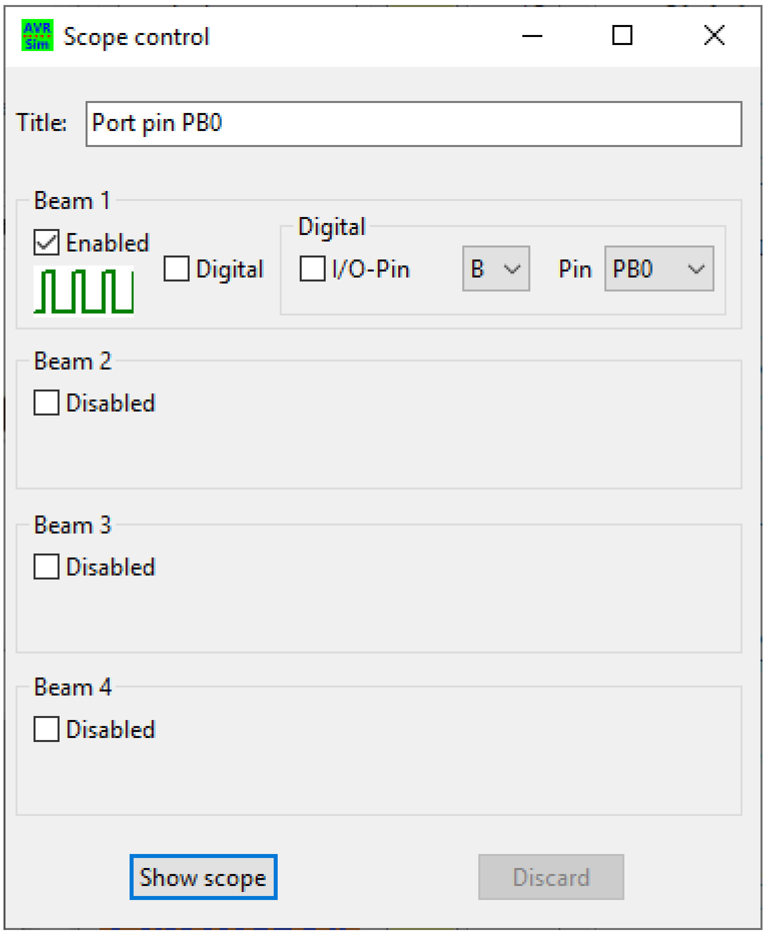
Now the output pin PB0 always drives the output pin: first to low (an attached LED to the operating voltage is on), then to high (the LED is off).

That is what you get on the PB0 pin: a



very fast signal of a few micro-seconds duration, the low period being half as long as the high period.

BTW: you'll get those pulse displays in avr_sim if you enable **Scope**, then in the scope control window choose these selections and click **Show scope**. A click on **Time elapsed** in the simulation status window clears the scope and restarts the time.


Scope control

Title: Port pin PB0

Beam 1
☑ Enabled ☐ Digital
Digital: ☐ I/O-Pin B Pin PB0

Beam 2
☐ Disabled

Beam 3
☐ Disabled

Beam 4
☐ Disabled

An additional case occurs if you clear the direction bit in a port bit and set the port output bit: The port pin now switches an internal resistor of approximately 50 kΩ to the operatig voltage to the input pin. This ties the input pin to the operating voltage ("pull-up"), so reading it by the instruction **IN R16,PINB** shows a one at this bit location. If you connect a switch or a button to this input pin, you can read a zero at this bit if the switch is on, tieing the input pin to ground and over-riding the pull-up, or a one if the switch or button is off.

4.2 Execution times

To exactly find out,

1. at which frequency the pin toggles, and
2. what causes the 66.7% pulse width

we have to go deeper into the details of instruction execution of the AVRs.

Each instruction in an AVR requires one, some require two clock cycles. At a clock rate of 1.2 MHz, on which the ATtiny13 works by default, each single clock cycle is 1 / 1.2 MHz = 0.833 µs long. The instructions in our PB0-toggle program are these clock cycles long:

```
  sbi DDRB,DDB0 ; Make PB0 an output, 2 clock cycles
Loop:
  cbi PORTB,PORTB0 ; Make PB0 low, 2 clock cycles
  sbi PORTB,PORTB0 ; Make PB0 high, 2 clock cycles
  rjmp Loop ; Repeat all over, 2 clock cycles
```

So our loop is 6 clock cycles long. The frequency is therefore

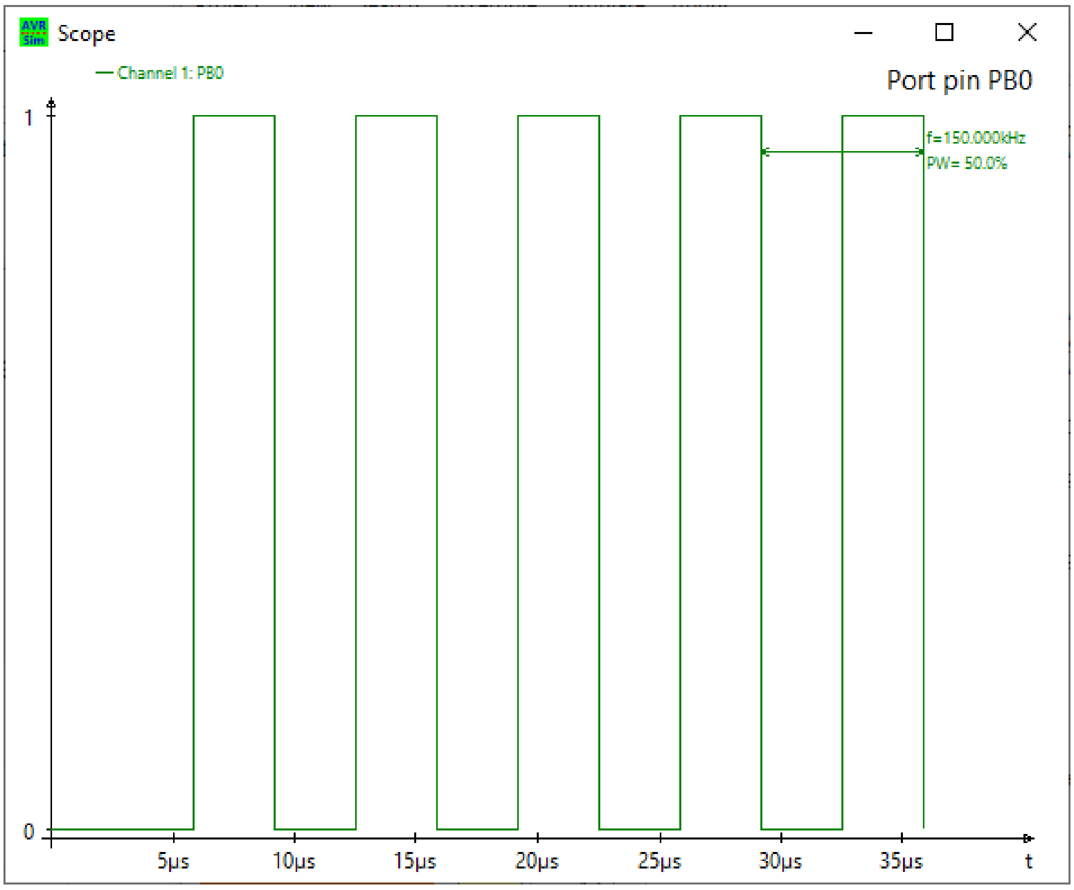
$$f_{rectangle} = 1.2\text{ MHz} / 6 = 200\text{ kHz}$$

Just what the simulator found out in the pulse diagram.

The low period, following the **CBI** with two cycles, is immediately changed to high, but the next instruction **RJMP** delays this high period by two additional clock cycles, so the high period is twice as long as the low period. That results in the 2/3 pulse width.

If you want the pulse width to be exactly 50 percent, you'll only have to insert two clock cycles delay between the **CBI** and the **SBI**. An ideal instruction to do this is the **NOP**: it does absolutely nothing else than wasting one clock cycle. So the source code changes to:

```
  sbi DDRB,DDB0 ; Make PB0 an output, 2 clock cycles
Loop:
  cbi PORTB,PORTB0 ; Make PBO low, 2 clock cycles
  nop ; 1 clock cycle
  nop ; 1 clock cycle
  sbi PORTB,PORTB0 ; Make PB0 high, 2 clock cycles
  rjmp Loop ; Repeat all over, 2 clock cycles
```



Now the whole loop is eight cycles long, with a frequency of

$f_{rectangle}$ = 1.2 MHz / 8 = 150 kHz

but the pulse width is exactly 50%.

Another feature that makes assembler an extremely useful programming language: you are able to exactly determine the duration that each instruction requires. Do not try this in a different language: you will not be able to find that out without going onto the instruction level, just because your compiler decides whether he has to insert other instructions or to use different instructions to switch PB0 on and off.

5 Using the timer hardware

It is not a good idea to use a controller to delay signals as this is wasting the controller's intelligence. For this the AVRs have timers or counters on board. The ATtiny13 has only one of those, others have a second, a few have even more of these. A timer or counter is a piece of internal hardware that (normally) counts up. Those counters can be eight bit wide (as in the ATtiny13), so can count until 255, or 16 bit wide and count until 65,535. When they reach this **TOP** value and the next pulse comes in, they restart at zero.

5.1 Switching the timer on

Switching the timer on is simple, just use the following source code:

```
  ldi R16,1<<CS00 ; Set the clock select bit 0 for the timer 0
  out TCCR0B,R16 ; and write it to the timer 0's control register
Loop:
  rjmp Loop
```

Now assemble that and start the simulation. In the "Show internal hardware" section of the simulation window enable **Timers/counters**. This displays the internals of the timer 0. The timer is inactive.

Your first step writes 0x01 to register R16. Where does that come from? First, take a look at the timer TC0 description in the ATtiny13's device data book and search for the term CS00. You'll find that CS00 is bit 0 in the Timer/Counter Control Register B (TCCR0B), which is eight bit wide and located at the address 0x33. If you use a different AVR type, CS00 and TCCR0B might be at a different location.

The term **1<<CS00** takes a binary 1 (0b00000001) and shifts it CS00 times to the left. As CS00 is zero, no **shifting left** is done, the result is 1. If you would take CS02, which is in bit 2 of TCCR0B, two shifts on 0b00000001 would lead to 0b00000100 or decimal 4. Note that the shifting is done solely by the assembler software, not by the controller (there are shift instructions for the controller, too, but those are different).

Now the R16 register is at 0x01 and the second step writes this to the TCCR0B register, which is done by the **OUT** instruction. The **OUT** instruction writes the 8 bits in a register to a port register. Port registers are 64 different storages that control internal hardware, such as the timer. With **OUT** we can control all 8 bits in TCCR0B at once, in one instruction.

From where does the assembler know that TCCR0B in an ATtiny13 is at address 0x33? Now, that is part of the include file "tn13def.inc" which was included on top of our source code. This file defines the symbols of the ATtiny13 (e. g. TCCR0B) and also provides their addresses (0x33) and is part of a Studio installation. But the file is not really read by avr_sim, it has all symbols of all AVRs on board and finds TCCR0B in its symbol list. So you do not need to install the Studio. And we do not have to care at which address TCCR0B actually is: the symbol list knows it.

By the way: we used PORTB and DDRB as well as PORTB0 and DDB0 in the previous example to switch the PB0 output on and off and to set its direction. Their values were takes from the same symbol list, so we don't have to care for port addresses and their bits. If you need those addresses, you'll find them in the section **Register summary** in any device data book.

After you executed the **OUT** instruction, the timer/counter 0

- is in mode **Normal**, which means: it is counting up,
- its prescaler is switched to **CK/1**,
- its counter value in **TCNT0** is 1.

With each further step, which are **RJMP**s back to the label **Loop:**, the counter value advances by two (because each **RJMP** takes two cycles).

After 128 single steps, the counter restarts at zero (or better: 1). If you want to stop the timer/counter when it has counted to 100, you can use the following source code:

```
  ldi R16,1<<CS00 ; Set the clock select bit 0 for the timer 0
  out TCCR0B,R16 ; and write it to the timer 0's control register
Loop:
  in R16,TCNT0 ; Read the counter value
  cpi R16,100 ; Compare with decimal 100
  brcs Loop ; Not yet equal or larger, go back to loop
  ; Now 100 has been reached
  clr R16 ; Set the timer/counter off
  out TCCR0B,R16 ; in its control port
  rjmp Loop
```

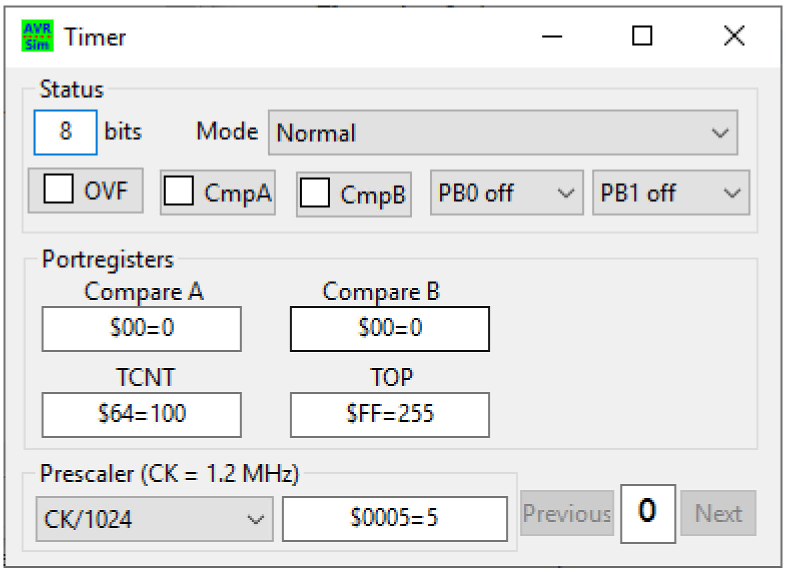
If you run that in the simulator, TCNT0 will stop at 105, not at 100 as desired. The reason for that is that the decision part requires its own clock cycles, and the stop is by five clocks later.

With other combinations of the three CS bits, the prescaler value changes. The values are

1. 1 with CS00 set,
2. 8 with CS01 set,
3. 64 with both both CS00 and CS01 set,
4. 256 with CS02 set, and
5. 1,024 with CS02 and CS00 set.

Setting two bits at once goes like this: **ldi R16,(1<<CS02) | (1<<CS00)**. This sets the prescaler to 1,024. The first term sets bit 2 to one, the second sets bit 0 to one, and the "|" or's both in binary, so that R16 will be set to 0b00000101 or decimal 5.

Now the 1.2 MHz clock is divided by 1,024 before the timer advances by one, f_{timer} = 1.2 MHz / 1,024 = 1.17 kHz. The timer


Timer

Status: 8 bits, Mode: Normal

OVF [ ] CmpA [ ] CmpB [ ] PB0 off, PB1 off

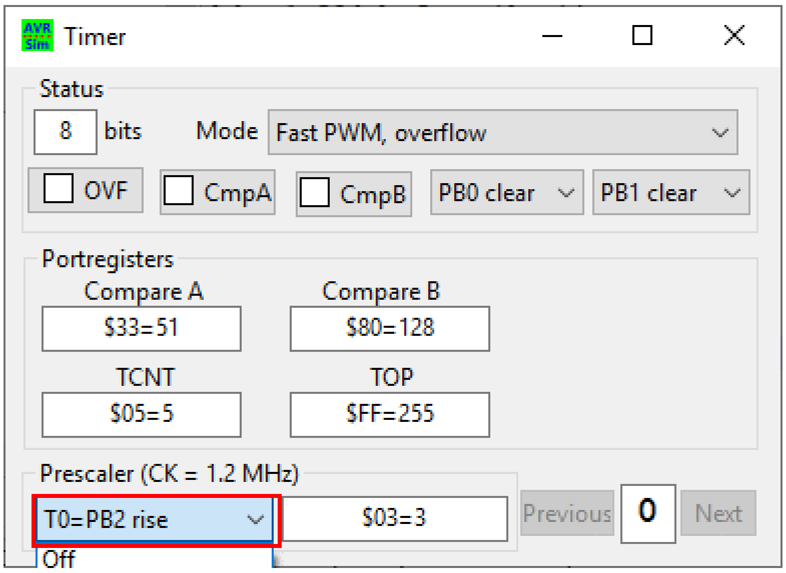
Portregisters
Compare A: $00=0
Compare B: $00=0
TCNT: $64=100
TOP: $FF=255

Prescaler (CK = 1.2 MHz): CK/1024, $0005=5

Previous 0 Next

counts roughly milliseconds now. The simulator now needs more than an hour until the 100 is reached.

The two CS combinations left can be used for counting external signals on the pin **T0** of the ATtiny13 (pin 7 of the DIP or SOIC package): the counter advances either on falling or rising edges on this pin. This really makes TC0 a counter.


Timer

Status: 8 bits, Mode: Fast PWM, overflow

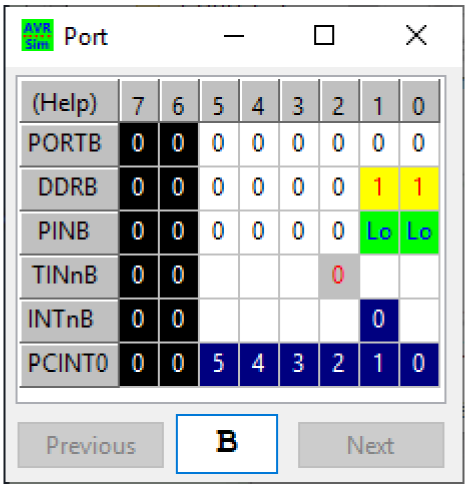
OVF, CmpA, CmpB, PB0 clear, PB1 clear

Portregisters
Compare A: $33=51
Compare B: $80=128
TCNT: $05=5
TOP: $FF=255

Prescaler (CK = 1.2 MHz): T0=PB2 rise, $03=3

Previous 0 Next

Off


Port

(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	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

You can try that by changing the prescaler in the dropdown field to the last entry, "T0=PB2 rise".

If you open the port display window, you'll find that bit 2 of port B in the line **TINnB** is now in red with a gray background, which signals that this bit is now an active input.

5.2 The timer in CTC mode

The timer can not only count to 255, but can be programmed to count to 99 only and then to restart with the next pulse. This timer mode is called CTC (Clear Timer on Compare). This requires to

1. set the compare value in port register OCR0A to 99, and
2. to switch the Waveform Generation bit WGM01 in the TC0 Control Register TCCR0A to one.

In this mode the clock cycle that follows a match between OCR0A and TCNT0 restarts the timer/counter. This divides the prescaled clock signal by (OCR0A + 1), e. g. 1.2 MHz / 1,024 / (99 + 1) = 11.72 Hz.

Another hardware component of the timer offers the opportunity to switch output pins of the controller. When OCR0A is equal to TCNT0 the COM0A0 and COM0A1 bits in TCCR0A control what happens with the OC0A pin (=PB0, pin 5 of the PDIP or SOIC package):

1. COM0A0 = 0, COM0A1 = 0: do nothing with PB0,
2. COM0A0 = 1, COM0A1 = 0: toggle PB0 (if it is low make it high, if it is high make it low),
3. COM0A0 = 0, COM0A1 = 1: make PB0 low,
4. COM0A0 = 1, COM0A1 = 1: make PB0 high.

The same can be done with PB1, for which

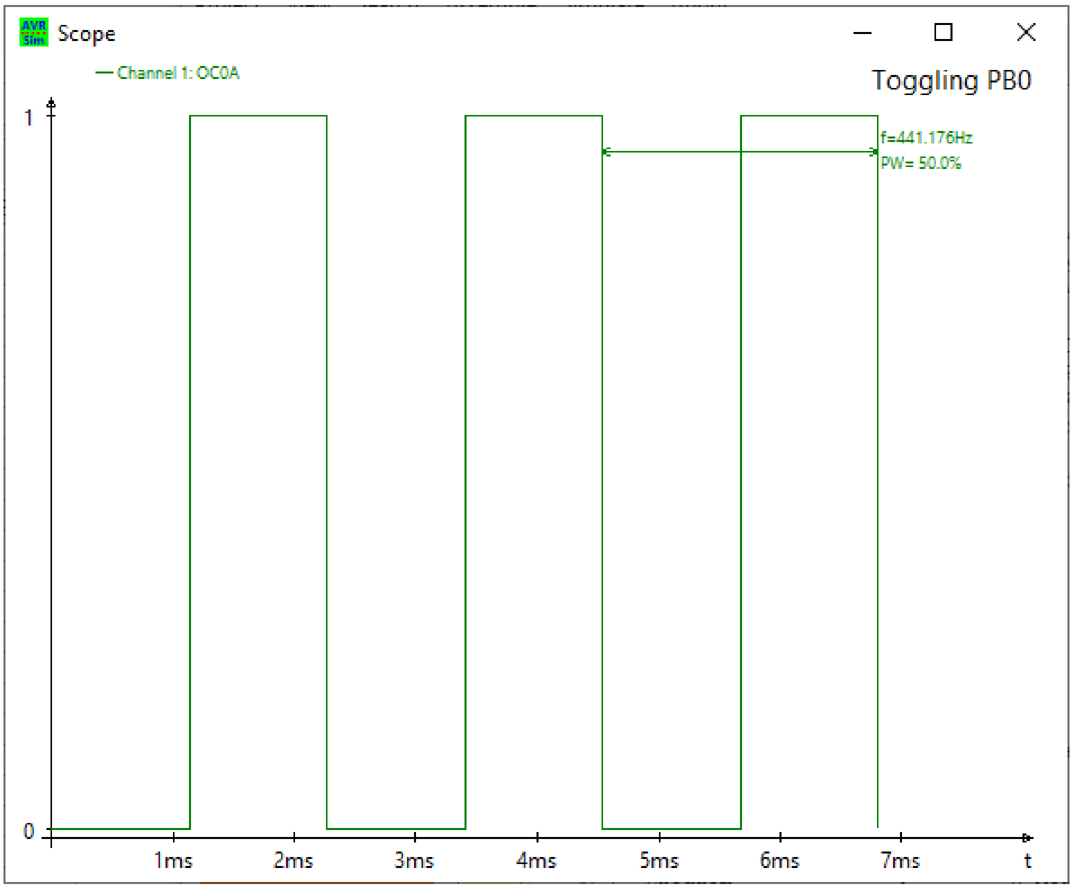
- the COM0B0 and COM0B1 bits determine the action, and
- OCR0B determines the counter value for the match.

With that pulses of nearly any duration or frequencies between clock / 2 and clock / 1,024 / 256 / 2 on PB0 can be generated, depending from OCR0A, with a pre-phased second signal with the same frequency on PB1. As toggling means that one timer period plus another timer period generate a rectangle, the frequency is half the divider rate.

The following source code divides the clock of 1.2 MHz by a prescaler value of 8 and by an OCR0A value of 169 (= 170) to generate a tone on PB0:

$$f_{PB0} = 1.2\ \text{MHz} / 8 / 170 / 2 = 441.18\ \text{Hz}$$

```
  sbi DDRB,DDB0 ; Make PB0 output
  ldi R16,169 ; Compare A
  out OCR0A,R16
  ldi R16,(1<<COM0A0)|(1<<WGM01) ; Toggle PB0, CTC mode
  out TCCR0A,R16 ; to TC0 control port A
  ldi R16,1<<CS01 ; Prescaler = 8
  out TCCR0B,R16 ; to TC0 control port B
Loop:
  rjmp Loop
```



That is the output on PB0. Looks like anything is fine, and the pulse width is exactly 50%.

5.3 The counter in PWM modes

That is not all the counter can do. It can also work in two different PWM modes:

1. in fast PWM mode: the counter sets or clears the output pin on restart and clears or sets the output pin on TOP (255 or OCR0A).
2. in phase-correct PWM mode: the counter counts up, if a compare match occurs, the output pin is either set or cleared. After reaching the TOP value, the counter counts down. If a compare match occurs, the output pin is either cleared or set.

Both differ by a factor of two in the PWM's frequency.

In both cases TOP can either be 255 (full counting range, 8-bit PWM) or can be set to OCR0A (limited range, <8-bit PWM). In the ladder case only compare B and PB1 can create a valid PWM signal, OCR0B in that case must be smaller than OCR0A.

In 8-bit-PWM mode COM0A0/1 and COM0B0/1 determine the polarity of the PB0/PB1 pins:

1. COM0A0=0: Set output PB0 on TOP, clear output on compare match.
2. COM0A0=1: Clear output PB0 on TOP, set output on compare match.

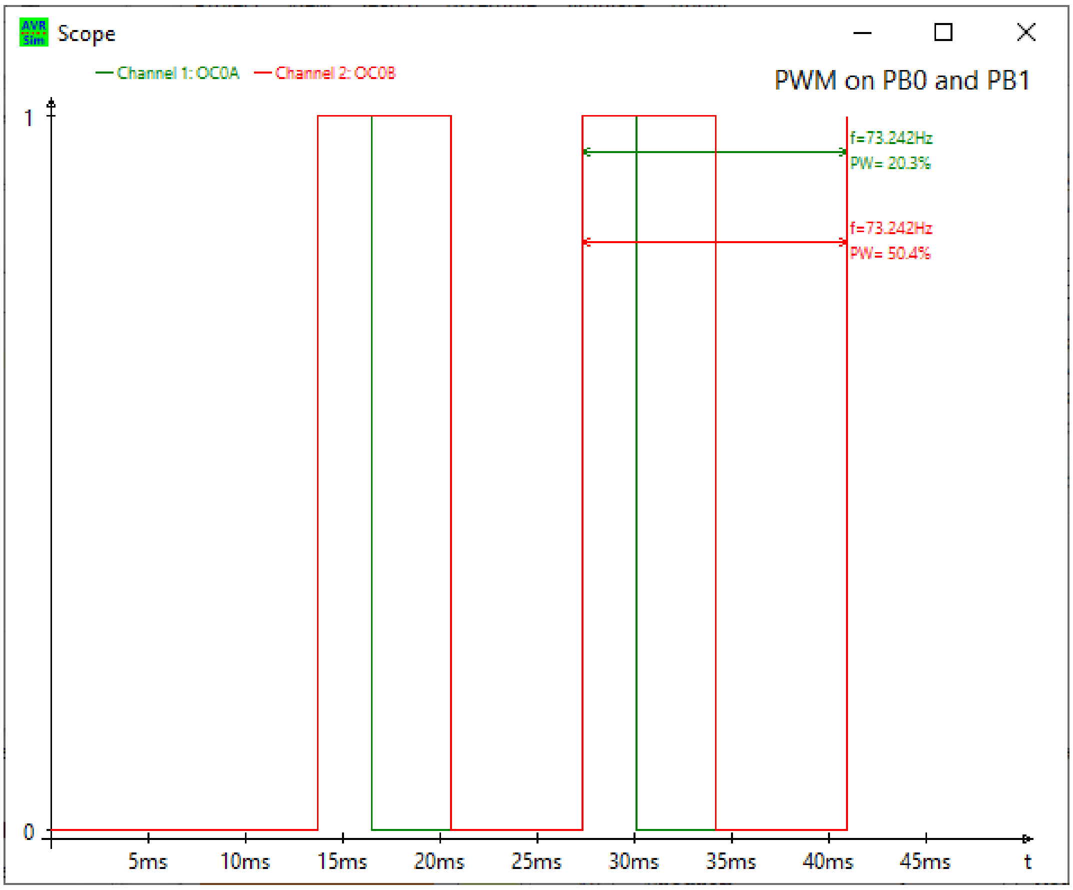
The same applies to COM0B0.

The following source code generates a 20% pulse width output on PB0 and a 50% pulse width on PB1 in fast mode with 73.24 Hz.

```
  sbi DDRB,DDB0 ; PB0 as output
  sbi DDRB,DDB1 ; PB1 as output
  ldi R16,20*256/100 ; PB0 with 20% pulse width
  out OCR0A,R16 ; to compare port A
  ldi R16,50*256/100 ; PB1 with 50% pulse width
  out OCR0B,R16 ; to compare port B
  ldi R16,(1<<COM0A1)|(1<<COM0B1)|(1<<WGM01)|(1<<WGM00) ; PB0/PB1 polarity, fast
PWM
  out TCCR0A,R16 ; to control port A
  ldi R16,(1<<CS01)|(1<<CS00) ; Prescaler to 64
  out TCCR0B,R16 ; to control port B
Loop:
  rjmp Loop
```

These are the signals produced: a short signal on PB0 and a longer signal on PB1. Just as pre-calculated.

We could attach two LEDs with two resistors to PB0 and PB1 and tie them to ground. And we would see that the LED on PB0 is only on during 20% of the time, while the LED on PB1 is half on half off.



Note that the generation of these two PWM signals is fully automatic: it does not need any overhead by the CPU any more. Ideal, because the CPU can perform other things while the timer produces these signals on its own.

5.4 Timer interrupts

Now, what if we want the LEDs to turn off after one minute? We would need a mechanism that counts one minute and then shut the PWM signal off.

Of course, we could construct a loop that counts 1,200,000 * 60 = 72,000,000 clock cycles and then shut it off. That would require a 32-bit counter and some compares to see if the endpoint is reached.

But that is not really clever. Our PWM already is at 73.24 Hz, so we would have to count to 73.24 * 60 = 4,394. If only we could find out the exact time when a PWM cycle is completed. Now: this is the task for an interrupt: whenever a PWM cycle is complete the TC0 overflows and restarts. We can just set the timer's overflow interrupt enable bit TOIE0 in the TIMSK0 port and we are done?

No, it is not that easy. The reason for that is that interrupts are by far more flexible. When such an interrupt condition occurs, the CPU does the following:

1. It switches further interrupts of by clearing the I-bit in its status register SREG.
2. It then stores the current execution address counter PC on a stack, so it can resume execution exactly at this location where the interrupt occurred.
3. It then writes an execution address to the PC which is specific for that interrupt and so jumps to that location.

4. At this address, an **RJMP** or a **JMP** (in larger devices) to a routine is expected, that is then executed.
5. If the routine is completed a **RETI** instruction is executed, which
    - reads back the previous address from the stack and writes this to the PC to resume execution there, and
    - sets the I bit in the status register, so that further interrupts are executed again.

Sounds complicated, but is a simple and reliable mechanism.

5.4.1 The stack

The question is: what is a stack that can store an execution address? Now, it is simply located in the SRAM memory of the device. At start-up we have to initialize that at the end of the SRAM storage: a port register named stackpointer (SP) points to this location. If the device has less or equal 256 - 96 = 160 bytes SRAM, the stack pointer is a single port register called SPL. If it has more SRAM the upper address byte is located in the port register SPH. To set the stackpointer to that address in an ATtiny13, we use the following instructions:

```
  ldi R16,LOW(RAMEND) ; Load the last SRAM address to R16
  out SPL,R16 ; and write it to the LSB of the stackpointer
```

The function **LOW** isolates the lower eight bits of the constant **RAMEND**. The same can be achieved by use of the function **BYTE1**.

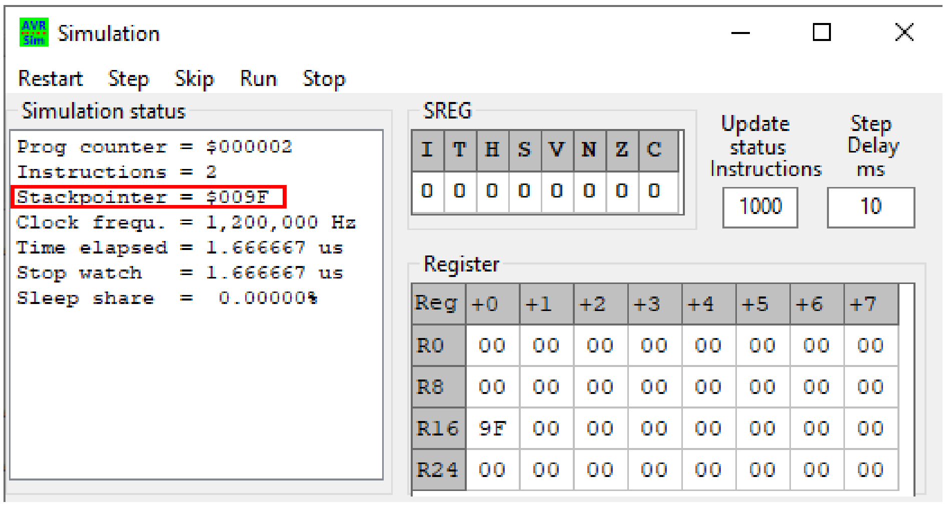
In devices with more SRAM we formulate the following:

```
  ldi R16,HIGH(RAMEND) ; Load the MSB of the last SRAM address to R16
  out SPH,R16 ; and write it to the MSB of the stackpointer
  ldi R16,LOW(RAMEND) ; Load the last SRAM address to R16
  out SPL,R16 ; and write it to the LSB of the stackpointer
```

With that we have set-up the stackpointer and we can use the stack now as an interim storage place. To see what is going on on the stack we write the following source code for an ATtiny13:

```
  ; Setting up the stack
  ldi R16,LOW(RAMEND) ; Load the last SRAM address to R16
  out SPL,R16 ; and write it to the LSB of the stackpointer
  ; Writing a register with a characteristic pattern
  ldi R16,'A' ; ASCII character A to R16
  ; Pushing the register to the stack
  push R16 ; Pushing onto the stack
  ; Popping the register value from the stack
  pop R0 ; Reading the last entry from the stack
Loop:
  rjmp Loop ; Indefinite loop
```

If we assemble this and start simulation we can see the effect of the first two instructions in the **Simulation status**: the stackpointer has been set to 0x009F.



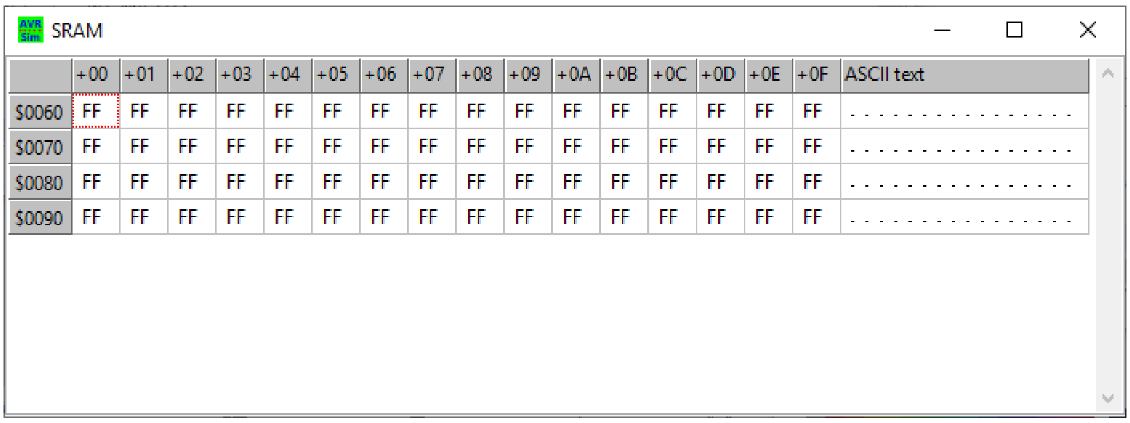
```
Prog counter = $000002
Instructions = 2
Stackpointer = $009F
Clock frequ. = 1,200,000 Hz
Time elapsed = 1.666667 us
Stop watch   = 1.666667 us
Sleep share  =  0.00000%
```

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

Update status Instructions	Step Delay ms
1000	10

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	9F	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

Before we execute further we open the SRAM view window in the **Show internal hardware** section of the simulation window. We see that the SRAM is at 0xFF, nothing has been written there yet.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . .

This changes if we execute the next two instructions: An A is written to register R16 and then to the last location in the SRAM. And: the stackpointer is now at 0x009E and advanced backwards by one location. If we execute the next instruction, the **POP** the opposite happens: the last value, an 'A', is popped to register R0 and the stackpointer returns to 0x009F. The SRAM content remains the same, but the next **PUSH** would override the 'A' there.

Pushing and popping registers is only one use of the stack. During interrupts the stack receives the return address. As addresses are 16 bit wide, two push operations write the address onto the stack. When the **RETI** instruction is executed, these two bytes are popped back to the PC.

The same mechanism can be used to call subroutines and return back at the end. The two instructions are **RCALL (relative address)** and **RET** to return to the next instruction behind the call. The forllowing source code demonstrates that:

```
  ; Setting up the stack
  ldi R16,LOW(RAMEND) ; Load the last SRAM address to R16
  out SPL,R16 ; and write it to the LSB of the stackpointer
  ; Calling a subroutine
  rcall MySub ; Call a subroutine
Loop:
  rjmp Loop ; Indefinite loop
;
; The subroutine
```

```
MySub:
  nop ; doing something
  ret ; and return to the call
```

If we assemble and start simulation the first two instructions set up the stack again. The **RCALL** then

1. writes the address of the current PC to the stack, LSB first, MSB second, and
2. jumps to the label **MySub:**.

Program execution resumes there and the **NOP** is executed. If **RET** occurs, the PC is popped from the stack (MSB first, then LSB) and execution resumes at the address following the **RCALL**.

With that we can call **MySub:** from different locations in the source code and, by use of the stack, the routine always returns back to the correct location. Ideal mechanism also for interrupts: no matter at which point in time they occur, the **RETI** always returns back to the correct location.

5.4.2 Interrupt vectors

From where do we know to which location the AVR jumps if a timer/counter overflows? To understand this, we create a new project with the ATtiny13, but enable interrupts and disable comprehensive. The created frame now has a section with all interrupt call addresses, all equipped with a **RETI** instruction so that it returns back if accidentally enabled:

```
; **********************************
; R E S E T  &  I N T - V E C T O R S
; **********************************
      rjmp Main ; Reset vector
      reti ; INT0
      reti ; PCI0
      reti ; OVF0
      reti ; ERDY
      reti ; ACI
      reti ; OC0A
      reti ; OC0B
      reti ; WDT
      reti ; ADCC
```

The TC0 overflow **OVF0** is our desired address at address 0x000003. If we would enable compare match interrupts for the timer, the **OC0A** on address 0x000006 and **OC0B** one address behind would be jumped to in case of an interrupt. The row, in which those vectors (a one-instruction list) are ordered has a meaning: in case two different interrupts occur at the same time, the interrupt that is higher in the list (with a lower address) is executed first. Only after this has been finished with **RETI** the second one still pending will be executed.

Note that the number of interrupts and also their rowing differs and is specific for each AVR type. So if you change the type make sure that you also use a different vector list.

On address 0x000000, which is the first executed address after a reset, a **RJMP** jumps over the interrupt vectors and to the new init routines. Similarly each **RETI** will be replaced by such an **RJMP** for any interrupt type you'd like to enable, because there is only one instruction that can be there.

In larger devices, that have beyond 4 kB of flash memory, the vectors are two instructions wide. So either use **JMP**, which is a two-word instruction, or **RJMP** followed by an **NOP**. Inactive vectors can have a **RETI** followed by an **NOP**.

5.4.3 Preserving resources in interrupts

Within so-called interrupt service routines any registers that you use can have a by-effect: if the interrupt executes, a register (or any other resource you use) can change. You'll have to ensure that this does not have an effect on the rest of your program execution. So, either use resources in interrupts exclusively or, if that is not possible, save those resources on the stack when entering the interrupt routine and restore those before leaving the interrupt routine.

One of the resources that you'll have to use is the status register SREG: any flag change that an instruction within the interrupt routine causes can have an unplanned side-effect to your program execution. So, you'll have to save SREG on entering, and to restore it before leaving. Saving can be done in an exclusive register. I call the register rSreg and place it always to R15. the source code for that would be:

```
; **********************************
;       R E G I S T E R S
; **********************************
;
; free: R0 to R14
.def rSreg = R15 ; SREG storage register
;
; **********************************
; R E S E T  &  I N T - V E C T O R S
; **********************************
      rjmp Main ; Reset vector
      reti ; INT0
      reti ; PCI0
      rjmp Ovf0Isr ; OVF0
      reti ; ERDY
      reti ; ACI
      reti ; OC0A
      reti ; OC0B
      reti ; WDT
      reti ; ADCC
;
; **********************************
;  I N T - S E R V I C E  R O U T .
; **********************************
;
; TC0 overflow interrupt service routine
Ovf0Isr:
  in rSreg,SREG ; Save the SREG
  ; ... further code
  out SREG,rSreg ; Restore the SREG
  reti ; Return from interrupt
;
; **********************************
;  M A I N  P R O G R A M  I N I T
; **********************************
```

```
;
Main:
  ; ... further code
```

That is all that is needed to use interrupts, and we can now step to our task to solve the 1-minute-PWM-active task.

5.4.4 Measuring one minute

To measure one minute, we'll have to count 4,394 timer overflow interrupts. If that is reached, the PWM shall be switched off.

Now, 4,394 is a 16-bit number. We could use the 16-bit registers R27:R26 (X), R29:R28 (Y) or R31:R30 (Z) for that, but better is to use R25:R24 for that. This is a 16-bit register that has no name, but can execute **ADIW** and **SBIW** instructions. If we set R25:R24 to 4,394 on start-up and count one down each time the overflow interrupt occurs, we can use the Z flag to find the end. The source code for the interrupt service routine is then:

```
Ovf0Isr:
  in rSreg,SREG ; Save SREG
  sbiw R24,1 ; Decrease 16 bit counter
  brne Ovf0IsrReti ; Not yet zero, continue
  clr R16 ; Disable timer interrupts
  out TIMSK0,R16 ; in the TC0 interrupt mask
  ldi R16,(1<<WGM01)|(1<<WGM00) ; Disconnect the PB0 and PB1 pin
  out TCCR0A,R16 ; in the control port A
  clr R16 ; Stop timer counting
  out TCCR0B,R16 ; in the control port B
  ldi R16,0 ; Clear the output lines
  out PORTB,R16 ; in port output register
  ldi R16,0 ; Switch of the output drivers
  out DDRB,R16 ; in the direction port
Ovf0IsrReti:
  out SREG,rSreg ; Restore SREG
  reti ; End of the service routine
```

In the **Main:** section we have to

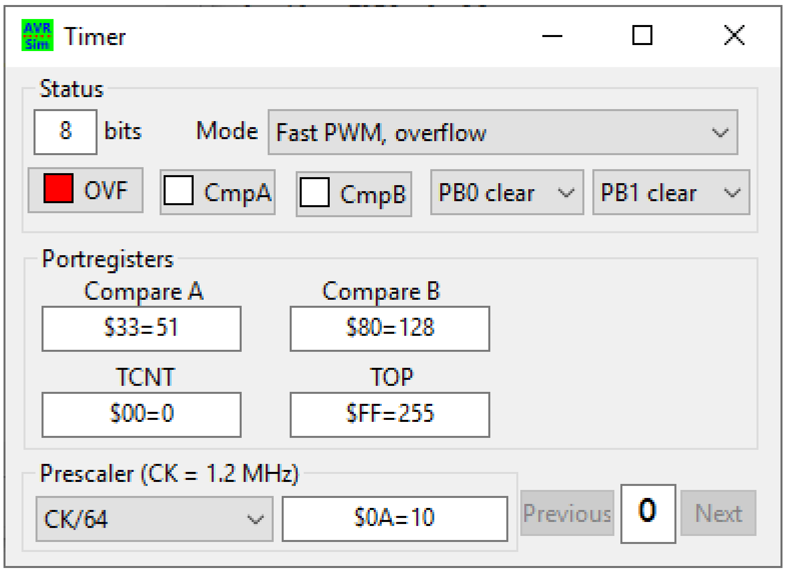
1. set-up the stack,
2. have to set R25:R24 to 4,194 (by use of the two instructions **ldi R25,HIGH(4394)** and **ldi R24,LOW(4394)**),
3. start the PWM operation as above,
4. enabling overflow interrupts with **ldi R16,1<<TOIE0**, and
5. before entering the indefinite loop enable the interrupts by setting the I flag in SREG with **SEI**.

The previously green overflow interrupt indication, signaling an active overflow interrupt enable bit,

- turns yellow if the interrupt is requested,
- turns red if the interrupt is executed, and
- returns to green if the interrupt execution ends with the **RETI** instruction.

If you'll have enough time you can wait for the minute to complete and to see that the overflow interrupt returns to white again and the timer and output finally stops.

You can download the assembler source code as asm file here, so you'll not have to type in anything on your own.


Timer

Status: 8 bits, Mode: Fast PWM, overflow

🟥 OVF, ☐ CmpA, ☐ CmpB, PB0 clear, PB1 clear

Portregisters: Compare A $33=51, Compare B $80=128, TCNT $00=0, TOP $FF=255

Prescaler (CK = 1.2 MHz): CK/64, $0A=10

6 Conclusions

As you can see from that:

1. Not assembler is complicated but the internal hardware of the AVRs is a little bit complicated due to the many features offered.
2. Switching that hardware on in the desired manner requires only a few instructions that configure this hardware.
3. The best resource to understand this hardware are the data books provided, so always keep those at hand when you write source code in assembler.
4. By use of the simulator avr_sim the internal hardware can be inspected step-by-step, to see if it does behave as planned. It is a powerful tool to learn and inspect.
5. High-level languages like C or Basic are of no use if you want to understand how a controller really works as they hide all relevant things from you. No one needs them, they rather drill you as a permanent searcher for libraries that do not exactly fit to what you really want and they prevent you from learning more about AVRs.