



Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors

by

Gerhard Schmidt

<http://www.avr-asm-tutorial.net>

September 2021

History:

Added chapters on binary floating points and on memory access in September 2021

Added chapter on code structures in April 2009

Additional corrections and updates as of January 2008

Corrected version as of July 2006

Original version of December 2003

Content

Why learning Assembler?.....	1
Short and easy.....	1
Fast and quick.....	1
Assembler is easy to learn.....	1
AVRs are ideal for learning assembler.....	1
Test it!.....	2
Hardware for AVR-Assembler-Programming.....	3
The ISP-Interface of the AVR-processor family.....	3
Programmer for the PC-Parallel-Port.....	3
Experimental boards.....	4
Experimental board with an ATtiny13.....	4
Experimental board with an AT90S2313/ATmega2313.....	5
Ready-to-use commercial programming boards for the AVR-family.....	6
STK200.....	6
STK500.....	6
AVR Dragon.....	7
Tools for AVR assembly programming.....	8
From a text file to instruction words in the flash memory.....	8
The editor.....	8
Structuring assembler code.....	9
Comments.....	9
Things to be written on top.....	9
Things that should be done at program start.....	10
Structuring of program code.....	10
The assembler.....	13
Programming the chips.....	14
Simulation in the studio.....	14
What is a register?.....	18
Different registers.....	20
Pointer-registers.....	20
Accessing memory locations with pointers.....	20
Reading program flash memory with the Z pointer.....	20
Tables in the program flash memory.....	21
Accessing registers with pointers.....	21
Recommendation for the use of registers.....	22
Ports.....	23
What is a Port?.....	23
Write access to ports.....	23
Read access to ports.....	24
Read-Modify-Write access to ports.....	24
Memory mapped port access.....	24
Details of relevant ports in the AVR.....	25
The status register as the most used port.....	25
Port details.....	26
SRAM.....	27
Using SRAM in AVR assembler language.....	27
What is SRAM?.....	27
For what purposes can I use SRAM?.....	27
How to use SRAM?.....	27
Direct addressing.....	27
Pointer addressing.....	28
Pointer with offset.....	28
Use of SRAM as stack.....	28
Defining SRAM as stack.....	29
Use of the stack.....	29
Bugs with the stack operation.....	30
Jumping and Branching.....	31
Controlling sequential execution of the program.....	31
What happens during a reset?.....	31
Linear program execution and branches.....	32
Branching.....	32
Timing during program execution.....	33
Macros and program execution.....	33

Subroutines.....	34
Interrupts and program execution.....	35
Calculations.....	38
Number systems in assembler.....	38
Positive whole numbers (bytes, words, etc.).....	38
Signed numbers (integers).....	38
Binary Coded Digits, BCD.....	38
Packed BCDs.....	39
Numbers in ASCII-format.....	39
Bit manipulations.....	39
Shift and rotate.....	40
Adding, subtracting and comparing.....	41
Adding and subtracting 16-bit numbers.....	41
Comparing 16-bit numbers.....	41
Comparing with constants.....	41
Packed BCD math.....	42
Format conversion for numbers.....	43
Conversion of packed BCDs to BCDs, ASCII or Binaries.....	43
Conversion of Binaries to BCD.....	43
Multiplication.....	43
Decimal multiplication.....	43
Binary multiplication.....	44
AVR-Assembler program.....	44
Binary rotation.....	45
Multiplication in the studio.....	45
Hardware multiplication.....	47
Hardware multiplication of 8-by-8-bit binaries.....	47
Hardware multiplication of a 16- by an 8-bit-binary.....	48
Hardware multiplication of a 16- by a 16-bit-binary.....	50
Hardware multiplication of a 16- by a 24-bit-binary.....	52
Division.....	53
Decimal division.....	53
Binary division.....	53
Program steps during division.....	54
Division in the simulator.....	54
Number conversion.....	56
Decimal Fractions.....	56
Linear conversions.....	56
Example 1: 8-bit-AD-converter with fixed decimal output.....	57
Example 2: 10-bit-AD-converter with fixed decimal output.....	58
Floating point numbers in assembler language.....	59
Floating points, if necessary.....	59
The format of floating point numbers.....	59
Conversion of binary to decimal number format.....	60
Conclusion:.....	60
Converting floating point numbers to decimal in assembler language.....	60
Allocation of numbers.....	60
Converting the mantissa to decimal.....	61
Converting the exponent bits.....	64
Rounding the decimal mantissa.....	66
Conversion from BCD to ASCII.....	67
Execution times.....	67
Faster than above: converting a 40-bit-binary to decimal.....	67
Conclusion.....	68
Floating point arithmetic in assembly language.....	68
Converting decimals to binary floating point numbers in assembler language.....	68
Decimal number formats.....	68
The assembler software for the conversion.....	69
Detecting the negative sign.....	69
Read the decimal mantissa and convert it to a binary integer.....	69
Calculate the binary mantissa.....	69
Determine the decimal exponent and convert it.....	69
Normalization and sign processing.....	70
Results.....	70
Conclusion.....	70

Address modes in AVRs.....	71
Accessing SRAM, registers and port registers.....	71
Accessing SRAM locations with fixed addresses.....	71
Accessing SRAM location with pointers.....	72
Accessing SRAM location with increasing pointers.....	73
Accessing SRAM location with decreasing pointers.....	74
Accessing SRAM locations with displacement addressing.....	76
Accessing port registers.....	78
Accessing classical port registers.....	78
Access to extended port registers.....	79
Access with pointers, example: the circular LED light.....	79
Accessing EEPROM.....	82
EEPROM initiation with the .ESEG directive.....	82
EEPROM port registers.....	82
Writing the EEPROM address.....	83
Reading from the EEPROM.....	84
Write access to the EEPROM.....	84
Flash memory accesses.....	86
The .CSEG directive.....	86
The LPM instruction.....	87
Advanced LPM instructions.....	88
Use examples for LPM.....	88
Annex.....	92
Instructions sorted by function.....	92
Directives and Instruction lists in alphabetic order.....	94
Assembler directives in alphabetic order.....	94
Instructions in alphabetic order.....	95
Port details.....	97
Status-Register, Accumulator flags.....	97
Stackpointer.....	97
SRAM and External Interrupt control.....	97
External Interrupt Control.....	98
Timer Interrupt Control.....	98
Timer/Counter 0.....	99
Timer/Counter 1.....	100
Watchdog-Timer.....	101
EEPROM.....	101
Serial Peripheral Interface SPI.....	102
UART.....	103
Analog Comparator.....	103
I/O Ports.....	104
Ports, alphabetic order.....	104
List of abbreviations.....	104

Why learning Assembler?

Assembler or other languages, that is the question. Why should I learn another language, if I already learned other programming languages? The best argument: while you live in France you are able to get through by speaking English, but you will never feel at home then, and life remains complicated. You can get through with this, but it is rather inappropriate. If things need a hurry, you should use the country's language.

Many people that are deeper into programming AVR's and use higher-level languages in their daily work recommend that beginners start with learning assembly language. The reason is that sometimes, namely in the following cases:

- if bugs have to be analyzed,
- if the program executes different than designed and expected,
- if the higher-level language doesn't support the use of certain hardware features,
- if time-critical in line routines require assembly language portions,

it is necessary to understand assembly language, e.g. to understand what the higher-level language compiler produced. Without understanding assembly language you do not have a chance to proceed further in these cases.

Short and easy

Assembler instructions translate one by one to executed machine instructions. The processor needs only to execute what you want it to do and what is necessary to perform the task. No extra loops and unnecessary features blow up the generated code. If your program storage is short and limited and you have to optimize your program to fit into memory, assembler is choice 1. Shorter programs are easier to debug, every step makes sense.

Fast and quick

Because only necessary code steps are executed, assembly programs are as fast as possible. The duration of every step is known. Time critical applications, like time measurements without a hardware timer, that should perform excellent, must be written in assembler. If you have more time and don't mind if your chip remains 99% in a wait state type of operation, you can choose any language you want.

Assembler is easy to learn

It is not true that assembly language is more complicated or not as easy to understand than other languages. Learning assembly language for whatever hardware type brings you to understand the basic concepts of any other assembly language dialects. Adding other dialects later is easy. As some features are hardware-dependent optimal code requires some familiarity with the hardware concept and the dialect. What makes assembler sometimes look complicated is that it requires an understanding of the controller's hardware functions. Consider this an advantage: by learning assembly language you simultaneously learn more about the hardware. Higher level languages often do not allow you to use special hardware features and so hide these functions.

The first assembly code does not look very attractive, with every 100 additional lines programmed it looks better. Perfect programs require some thousand lines of code of exercise, and optimization requires lots of work. The first steps are hard in any language. After some weeks of programming you will laugh if you go through your first code. Some assembler instructions need some months of experience.

AVR's are ideal for learning assembler

Assembler programs are a little bit silly: the chip executes anything you tell it to do, and does not ask you if you are sure overwriting this and that. All protection features must be programmed by you, the chip does exactly anything like it is told, even if it doesn't make any sense. No window warns you, unless you programmed it before.

To correct typing errors is as easy or complicated as in any other language. Basic design errors, the more tricky type of errors, are also as complicated to debug like in any other computer language. But: testing programs on ATMEL chips is very easy. If it does not do what you expect it to do, you can easily add some diagnostic lines to the code, reprogram the chip and test it. Bye, bye to you EPROM programmers, to the UV lamps used to erase your test program, to you pins that don't fit into the socket after having them removed some dozen times.

Changes are now programmed fast, compiled in no time, and either simulated in the studio or checked in-circuit. No pin is removed, and no UV lamp gives up just in the moment when you had your excellent idea about that bug.

Test it!

Be patient doing your first steps! Most of the special features of other computer languages don't make any sense in assembler, so if you are familiar with another (high-level) language: forget it for the first time, it blocks you in learning. Behind every assembler language there is a certain hardware concept, so learn hardware AND software simultaneously.

The first five instructions are not easy to learn, after that your learning speed rises fast. After you had your first lines: grab the instruction set list and lay back in the bathtub, wondering what all the other instructions are like.

Serious warning: Don't try to program a mega-machine to start with. This does not make sense in any computer language, and only produces frustration and hurdles. Start with the small „Hello world“-like examples, e.g. turning some LEDs on and off for a certain time, then explore the hardware features a bit deeper.

Recommendation: Comment your subroutines and store them in a special directory, if debugged: you will need them, or the ideas behind that, again in a short time.

If you need a good tool to learn assembler for AVR: the simulator `avr_sim`, that can be downloaded [here](#), is an easy to use software that executes self-made assembler software and displays the internal hardware as if you are yourself inside the controller.

Have success!

Hardware for AVR-Assembler-Programming

Learning assembler requires some simple hardware equipment to test your programs, and see if it works in practice.

This section shows two easy schematics that enable you to home brew the required hardware and gives you the necessary hints on the required background. This hardware really is easy to build. I know nothing easier than that to test your first software steps. If you like to make more experiments, leave some more space for future extensions on your experimental board.

If you don't like the smell of soldering, you can buy a ready-to-use board, too. The available boards are characterized in this section below.

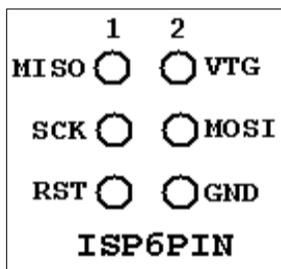
The ISP-Interface of the AVR-processor family

Before going into practice, we have to learn a few essentials on the serial programming mode of the AVR family. No, you don't need three different voltages to program and read an AVR flash memory. No, you don't need another pre-programmed microprocessor to program the AVRs. No, you don't need 10 I/O lines to tell the chip what you like it to do. And you don't even have to remove the AVR from the socket on your your experimental board, before programming it. It's even easier than that.

All this is done by a build-in interface in the AVR chips, that enable you to write and read the content of the program flash and the built-in-EEPROM. This interface works serially and needs only three signal lines:

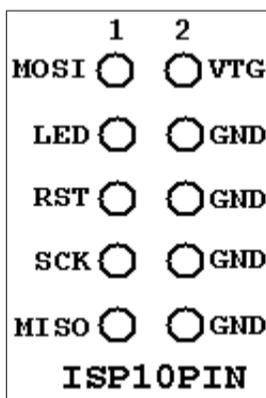
- SCK: A clock signal that shifts the bits to be written to the memory into an internal shift register, and that shifts out the bits to be read from another internal shift register,
- MOSI: The data signal that sends the bits to be written to the AVR,
- MISO: The data signal that receives the bits read from the AVR.

These three signal pins are internally connected to the programming machine only if you change the RESET (sometimes also called RST or restart) pin to zero. Otherwise, during normal operation of the AVR, these pins are programmable I/O lines like all the others.



If you like to use these pins for other purposes during normal operation, and for in-system-programming, you'll have to take care, that these two purposes do not conflict. Usually you then decouple these by resistors or by use of a multiplexer. What is necessary in your case, depends from your use of the pins in the normal operation mode. You're lucky, if you can use them for in-system-programming exclusively.

Not necessary, but recommendable for in-system-programming is, that you supply the programming hardware out of the supply voltage of your system. That makes it easy, and requires two additional lines between the programmer and the AVR board. GND is the common ground or negative pole of the supply voltage, VTG (target voltage) the supply voltage (usually +5.0 volts). This adds up to 6 lines between the programmer hardware and the AVR board. The resulting ISP6 connection, as defined by AMEL, is shown on the left.

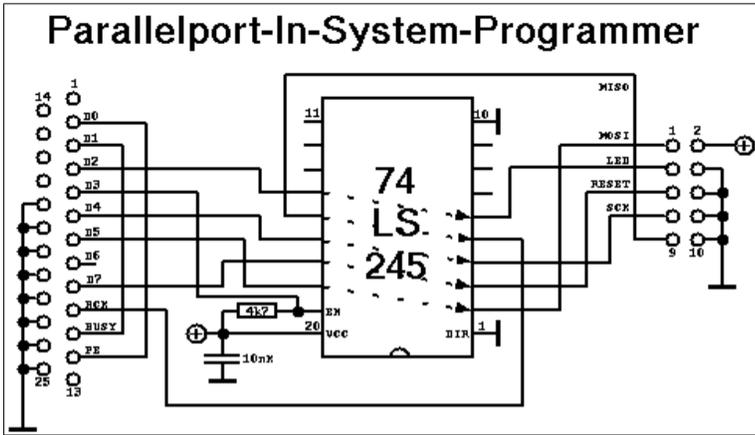


Standards always have alternative standards, that were used earlier. This is the technical basis that constitutes the adapter industry. In our case the alternative standard was designed as ISP10 and was used on the STK200 board, sometimes also called CANDIA interface. It's still a very widespread standard, and even the more recent STK500 board is equipped with it. ISP10 has an additional signal to drive a red LED. This LED signals that the programmer is doing his job. A good idea. Just connect the LED to a resistor and clamp it the positive supply voltage.

Programmer for the PC-Parallel-Port

Now, heat up your soldering iron and build up your programmer. It is a quite easy schematic and works with standard parts from your well-sorted experiments box.

Yes, that's all you need to program an AVR. The 25-pin plug goes into the parallel port of your PC, the 10-pin-ISP goes to your AVR experimental board. If your box doesn't have a 74LS245, you can also use a 74HC245 (with no hardware changes) or a 74LS244/74HC244 (by changing some pins and signals). If you use HC, don't forget to tie unused inputs either to GND or the supply voltage, otherwise the buffers might produce extra noise by capacitive switching.



The necessary program algorithm is done by the ISP software. Be aware that this parallel port interface is not supported by ATMEL's studio software any more. So, if you want to program your AVR directly from within the studio, use different programmers. The Internet provides several solutions.

If you already have a programming board, you will not need to build this programmer, because you'll find the ISP interface on some pins. Consult your handbook to locate these.

Experimental boards

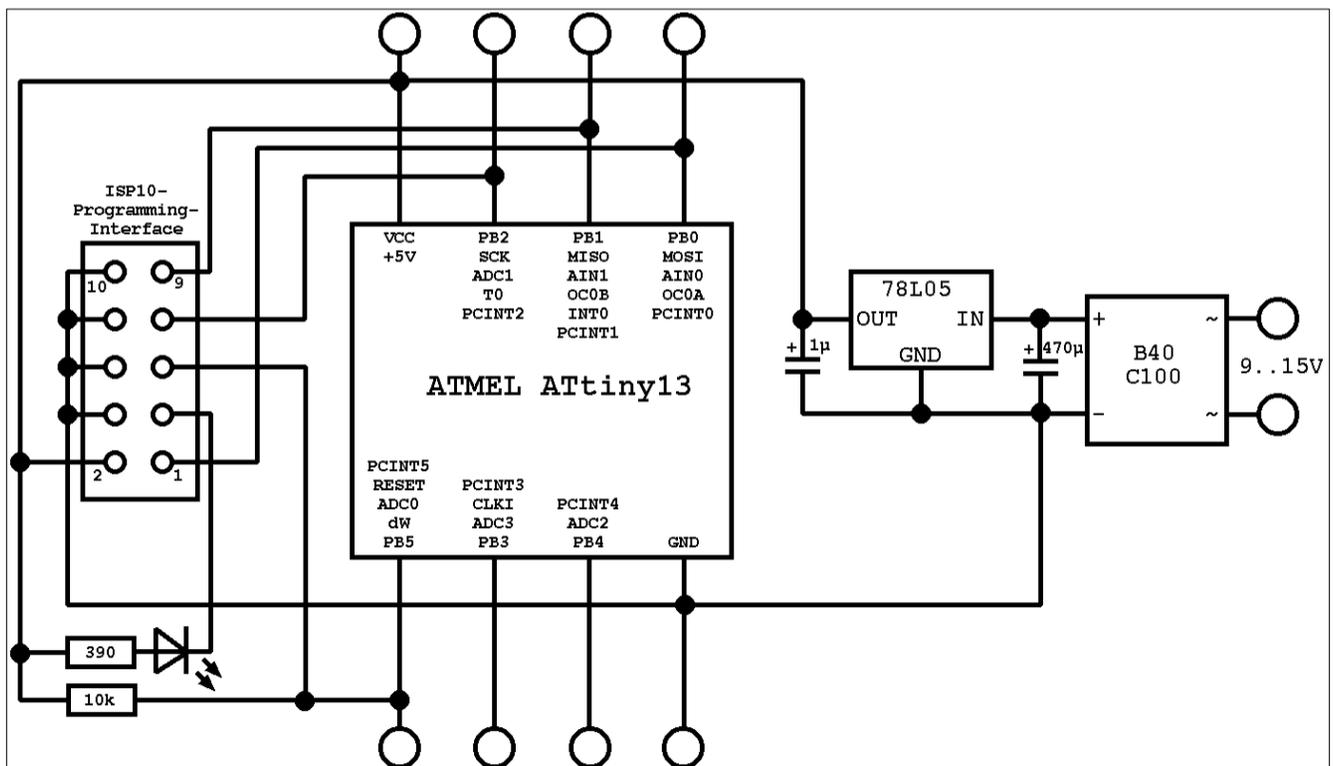
You probably want to do your first programming steps with a self-made AVR board. Here are two versions offered:

- A very small one with an ATtiny13, or
- a more complicated one with an AT90S2313 or ATmega2313, including a serial RS232 interface.

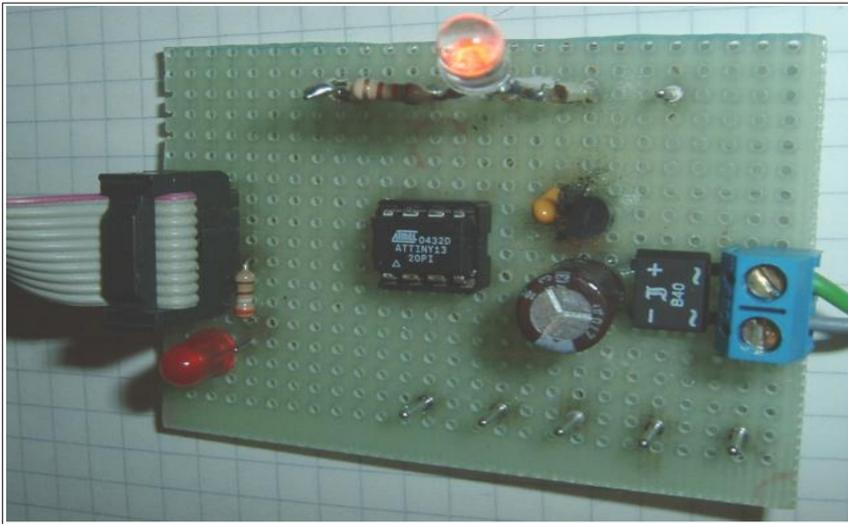
Experimental board with an ATtiny13

This is a very small board that allows experiments with the ATtiny13's internal hardware. The picture shows

- the ISP10 programming interface on the left, with a programming LED attached via a resistor of 390 Ohms,
- the ATtiny13 with a pull-up of 10k on its RESET pin (pin 1),
- the supply part with a bridge rectifier, to be supplied with 9..15V from an AC or DC source, and a small 5V regulator.



The ATtiny13 requires no external XTAL or clock generator, because it works with its internal 9.6 Mcs/s RC generator and, by default, with a clock divider of 8 (clock frequency 1.2 Mcs/s).



The hardware can be build on a small board like the one shown in the picture. All pins of the tiny13 are accessible, and external hardware components, like the LED shown, can be easily plugged in.

This board allows the use of the ATtn13's hardware components like I/O-ports, timers, AD converters, etc.

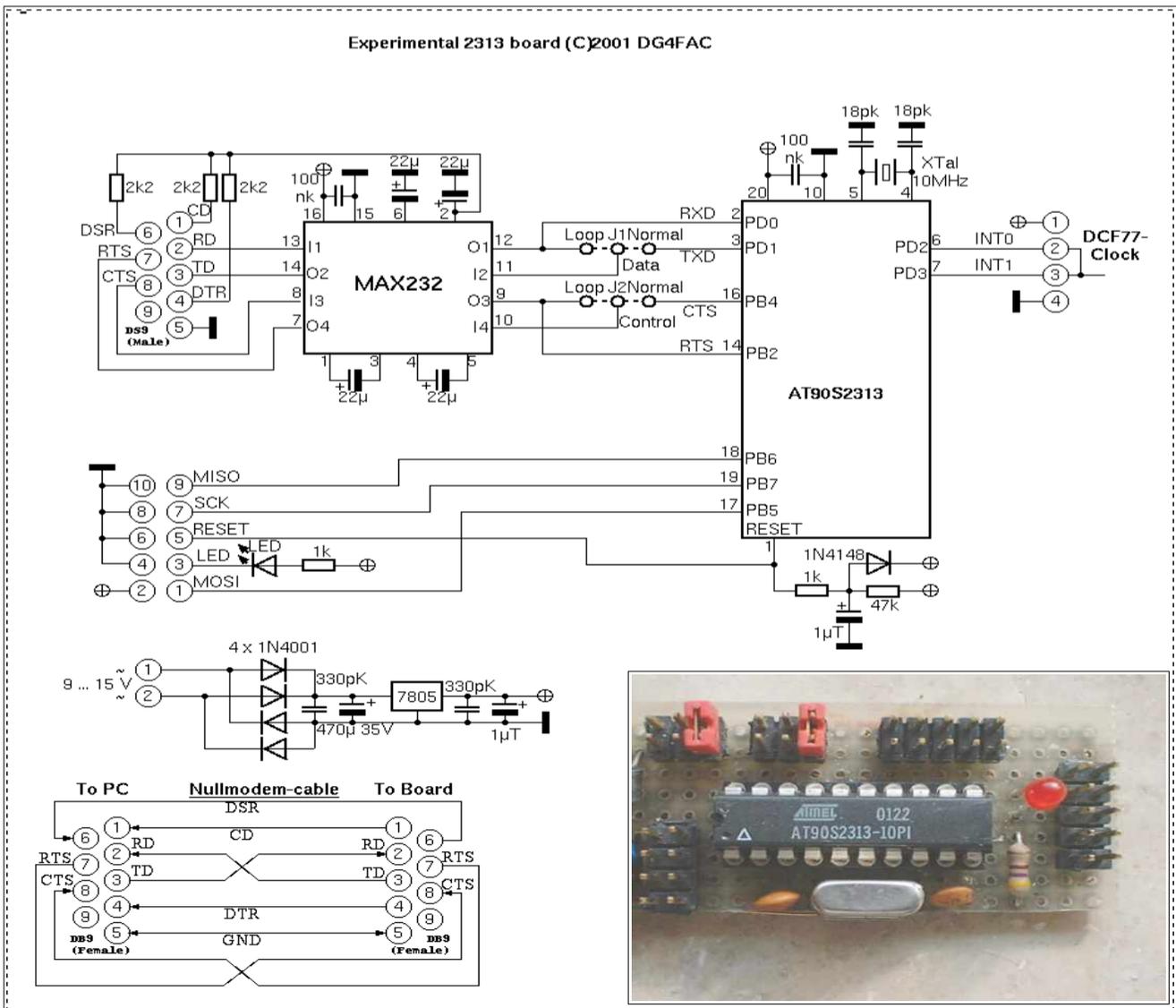
Experimental board with an AT90S2313/ATmega2313

For test purposes, were more I/O-pins or a serial communication interface is necessary, we can use a AT90S2313 or ATmega2313 on an experimental board. The schematic shows

- a small voltage supply for connection to an AC transformer and a voltage regulator 5V/1A,
- a XTAL clock generator (here with a 10 Mcs/s XTAL, all other frequencies below the maximum for the 2313 will also work),
- the necessary parts for a safe reset during supply voltage switching,
- the ISP-Programming-Interface (here with a ISP10PIN-connector).

So that's what you need to start with. Connect other peripheral add-ons to the numerous free I/O pins of the 2313.

The easiest output device can be a LED, connected via a resistor to the positive supply voltage. With that, you can start writing your first assembler program switching the LED on and off.



If you

- do not need the serial communication interface, just skip the hardware connected to pins 2/3 and 14/16,
- if you do not need hardware handshake signals, skip the hardware on the pins 14/16 and connect RTS on the 9-pin-connector over a 2.2k resistor to +9V.

If you use an ATmega2313 instead of an AT90S2313, the following changes are resulting:

- the external XTAL is not necessary, as the ATmega has an internal RC clock generator, so just skip all connections to pins 4 and 5,
- if you want to use the external XTAL instead of the build-in RC as clock source, you will have to program the fuses of the ATmega accordingly.

Ready-to-use commercial programming boards for the AVR-family

If you do not like homebrewed hardware, and if have some extra money left that you don't know what to do with, you can buy a commercial programming board. Depending from the amount of extra money you'd like to spend, you can select between more or less costly versions. For the amateur the following selection criteria should be looked at:

- price,
- PC interface (preferably USB, less convenient or durable: 9-pin RS232, requiring additional software: interfaces for the parallel port of the PC),
- support reliability for newer devices (updates are required from time to time, otherwise you sit on a nearly dead horse),
- hardware features (depends on your foreseeable requirements in the next five years).

The following section describes the three standard boards of ATMEL, the STK200, the STK500 and the Dragon. The selection is based on my own experiences and is not a recommendation.

STK200

The STK200 from ATMEL is a historic board. If you grab a used one you'll get

- a board with some sockets (for 8, 20, 28 and 40 pin devices),
- eight keys and LEDs, hard connected to port D and B,
- an LCD standard 14-pin interface,
- an option for attaching a 28-pin SRAM,
- a RS232 interface for communication,
- a cable interface for a PC parallel port on one side and a 10-pin-ISP on the other side.

HV programming is not supported.

The board cannot be programmed from within the Studio, the programming software is no longer maintained, and you must use external programs capable of driving the PC parallel port.

If someone offers you such a board, take it only for free and if you're used to operate software of the necessary kind.

STK500

Easy to get is the STK500 (e.g. from ATMEL). It has the following hardware:

- Sockets for programming most of the AVR types (e.g. 14-pin devices or TQFP packages require additional hardware),
- serial and parallel programming in normal mode or with high voltage (HV programming brings devices back to life even if their RESET pin has been fuse-programmed to be normal port input),
- ISP6PIN- and ISP10PIN-connection for external In-System-Programming,
- programmable oscillator frequency and supply voltages,
- plug-in switches and LEDs,
- a plugged RS232C-connector (UART),
- a serial Flash-EEPROM (only older boards have this),
- access to all port pins via 10-pin connectors.

A major disadvantage of the board is that, before programming a device, several connections have to be

made manually with the delivered cables.

The board is connected to the PC using a serial port (COMx). If your laptop doesn't have a serial interface, you can use one of the common USB-to-Serial-Interface cables with a software driver. In that case the driver must be adjusted to use between COM1 and COM8 and a baud rate of 115k to be automatically detected by the Studio software.

Programming is performed and controlled by recent versions of AVR studio, which is available for free from ATMEL's web page after registration. Updates of the device list and programming algorithm are provided with the Studio versions, so the support for newer devices is more likely than with other boards and programming software.

Experiments can start with the also supplied AVR (older versions: AT90S8515, newer boards versions include different types). This covers all hardware requirements that the beginner might have.

AVR Dragon

The AVR dragon is a very small board. It has an USB interface, which also supplies the board and the 6-pin-ISP interface. The 6-pin-ISP-Interface is accompanied by a 20-pin HV programming interface. The board is prepared for adding some sockets on board, but doesn't have sockets for target devices and other hardware on board.

The dragon is supported by the Studio software and is updated automatically.

Its price and design makes it a nice gift for an AVR amateur. The box fits nicely in a row with other precious and carefully designed boxes.

Tools for AVR assembly programming

Four basic programs are necessary for assembly programming. These tools are:

- the editor,
- the assembler program,
- the chip programming interface, and
- the simulator.

Two different basic routes are possible:

1. anything necessary in one package,
2. each task is performed with a specific program, the results are stored as specific files.

Usually route #1 is chosen. But because this is a tutorial, and you are to understand the underlying mechanism first, we start with the description of route #2 first.

From a text file to instruction words in the flash memory

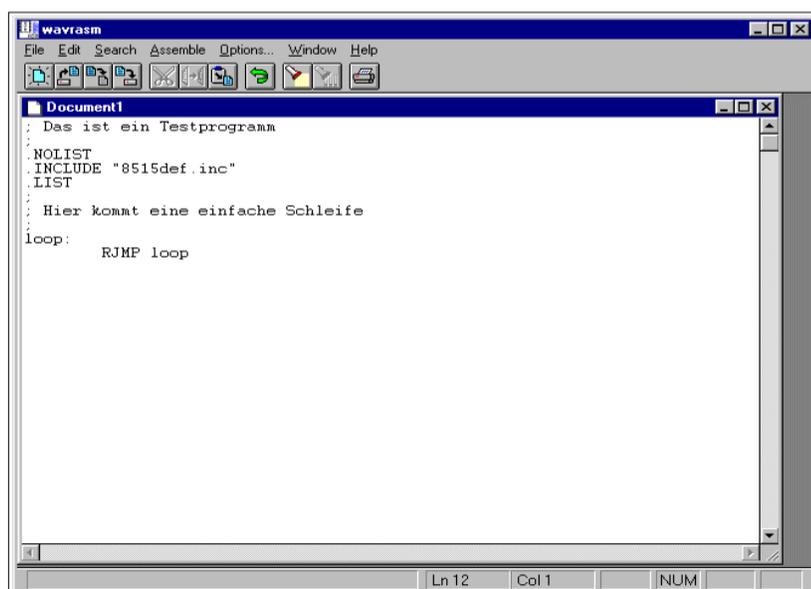
The editor

Assembler programs are written with an editor. The editor just has to be able to create and edit ASCII text files. So, basically, any simple editor does it.

Some features of the editor can have positive effects:

- Errors, that the assembler later detects, are reported along with the line number in the text file. Line numbers are also a powerful invention of the computer-age when it comes to discussions on your code with someone else. So your editor should be able to display the line number. Unfortunately nearly all editors, that a mighty software company provides as part of its operating systems, are missing that feature. Probably Widows 2019 re-invents that feature, and sells better among assembler freaks.
- Typing errors are largely reduced, if those errors are marked with colors. It is a nice feature of an editor to highlight the components of a line in different colors. More or less intelligent recognition of errors ease typing. But this is a feature that I don't really miss.
- If your editor allows the selection of fonts, chose a font with fixed spacing, like Courier. Headers look nicer with that.
- Your editor should be capable of recognizing line ends with any combination of characters (carriage returns, line feeds, both) without producing unacceptable screens. Another item on the wishlist for Widows 2013.

If you prefer shooting with cannons to kill sparrows, you can use a mighty word processing software to write assembler programs. It might look nicer, with large bold headings, gray comments, red warnings, changes marked, and reminders on To-Do's in extra bubble fields. Some disadvantages here: you have to convert your text to plain text at the end, losing all your nice design work, and your resulting textfile should not have a single control byte left. Otherwise this single byte will cause an error message, when you assemble the text. And remember: Line numbers here are only correct on page one of your source code.



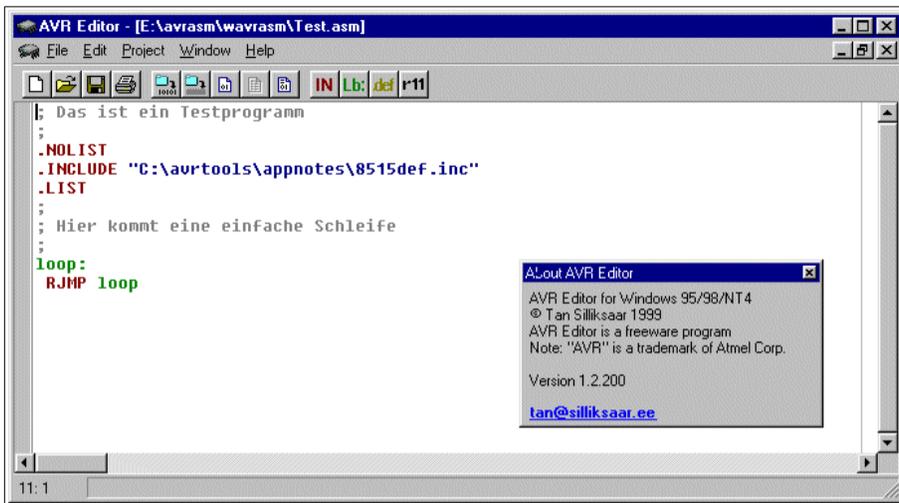
So, whatever text program you chose, it's up to you. The following examples are written in wavrasn, an editor provided by ATMEL in earlier days.

In the plain editor field we type in our directives and assembly instructions. It is highly recommended that lines come together with some comments (starting with ;). Later understanding of what we've planned here will be helpful in later debugging.

Now store the program text, named to something.asm into a dedicated directory, using the file menu. The assembly program is complete now.

If you'd like to see what syntax-highlighting means, I have a snapshot of such an AVR editor here.

The editor recognizes instructions automatically and uses different colors (syntax highlighting) to signal user constants and typing errors in those instructions (in black). Storing the code in an .asm file provides



nearly the same text file, colors are not stored in the file.

Don't try to find this editor or its author; the editor is history and no longer maintained.

Structuring assembler code

This page shows the basic structure of an assembler program. These structures are typical for AVR assembler.

This text discusses

- [comments](#),
- [header informations](#),
- [code at program start](#) and
- the [general structure of programs](#).

Comments

The most helpful things in assembler programs are comments. If you need to understand older code that you wrote, sometimes years after, you will be happy about having some or more hints what is going on in that line. If you like to keep your ideas secret, and to hide them against yourself and others: don't use comments. A comment starts with a semicolon. All that follows behind on the same line will be ignored by the compiler. If you need to write a comment over multiple lines, start each line with a semicolon. So each assembler program should start like that:

```

;
; Click.asm, Program to switch a relais on and off each two seconds
; Written by G.Schmidt, last change: 7.10.2001
;

```

Put comments around all parts of the program, be it a complete subroutine or a table. Within the comment mention the special nature of the routine, pre-conditions necessary to call or run the routine. Also mention the results of the subroutine in case you later will have to find errors or to extend the routine later. Single line comments are defined by adding a semicolon behind the command on the line. Like this:

```

LDI R16,0x0A ; Here something is loaded
MOV R17,R16 ; and copied somewhere else

```

Things to be written on top

Purpose and function of the program, the author, version information and other comments on top of the program should be followed by the processor type that the program is written for, and by relevant constants and by a list with the register names. The processor type is especially important. Programs do not run on other chip types without changes. The instructions are not completely understood by all types, each type has typical amounts of EEPROM and internal SRAM. All these special features are included in a header file that is named xxxdef.inc, with xxx being the chip type, e.g. 2313, tn2323, or m8515. These files are available by [ATMEL](#). It is good style to include this file at the beginning of each program. This is done like that:

```

.NOLIST ; Don't list the following in the list file
.INCLUDE "m8515def.inc" ; Import of the file
.LIST ; Switch list on again

```

The path, where this file can be found, is only necessary if you don't work with ATMEL's Studio. Of course you have to include the correct path to fit to your place where these files are located. During assembling, the output of a list file listing the results is switched on by default. Having listing ob might result in very long list file (*.lst) if you include the header file. The directive .NOLIST turns off this listing for a while, .LIST turns it on again. Let's have a short look at the header file. First these files define the processor type:

```

.DEVICE ATMEGA8515 ; The target device type

```

The directive .DEVICE advises the assembler to check all instructions if these are available for that AVR type. It results in an error message, if you use code sequences that are not defined for this type of processor. You don't need to define this within your program as this is already defined within the header file. The header file also defines the registers XH, XL, YH, YL, ZH and ZL. These are needed if you use the 16-bit-pointers X, Y or Z to access the higher or lower byte of the pointer separately. All port locations are

also defined in the header file, so PORTB translates to a hex number where this port is located on the defined device. The port's names are defined with the same names that are used in the data sheets for the respective processor type. This also applies to single bits in the ports. Read access to port B, Bit 3, can be done using its bit name PINB3, as defined in the data sheet. In other words: if you forget to include the header file you will run into a lot of error messages during assembly. The resulting error messages are in some cases not necessarily related to the missing header file. Others things that should be on top of your programs are the register definitions you work with ,e. g.:

```
.DEF mpr = R16 ; Define a new name for register R16
```

This has the advantage of having a complete list of registers, and to see which registers are still available and unused. Renaming registers avoids conflicts in the use of these registers and the names are easier to remember. Further on we define the constants on top of the source file, especially those that have a relevant role in different parts of the program. Such a constant would, e. g., be the Xtal frequency that the program is adjusted for, if you use the serial interface on board. With

```
.EQU fq = 4000000 ; XTal frequency definition
```

at the beginning of the source code you immediately see for which clock you wrote the program. Very much easier than searching for this information within 1482 lines of source code.

Things that should be done at program start

After you have done the header, the program code should start. At the beginning of the code the reset- and interrupt-vectors (their function see in the [JUMP](#) section) are placed. As these require relative jumps, we should place the respective interrupt service routines right behind. In case of ATmega types with larger flash memory JUMP instructions can be used here, so be careful here. There is some space left then for other subroutines, before we place the main program. The main program always starts with initialization of the stack pointer, setting registers to default values, and the init of the hardware components used. The following code is specific for the program.

Structuring of program code

The described standardized structure is included in a program written for Windows Operating Systems, which can be downloaded at http://www.avr-asm-download.de/avr_head.zip.



Unzip the executable file, and simply run it. It shows this: Here you can choose **ATtiny** by clicking on it, and then select **ATtiny13** in the dropdown field **AVR-Type**.


```

; =====
;   H A R D W A R E   I N F O R M A T I O N
; =====
;
; [Add all hardware information here]
;
; =====
;   P O R T S   A N D   P I N S
; =====
;
; [Add names for hardware ports and pins here]
; Format: .EQU Controlportout = PORTA
;         .EQU Controlportin = PINA
;         .EQU LedOutputPin = PORTA2
;
; =====
;   C O N S T A N T S   T O   C H A N G E
; =====
;
; [Add all constants here that can be subject
;  to change by the user]
; Format: .EQU const = $ABCD
;
; =====
;   F I X + D E R I V E D   C O N S T A N T S
; =====
;
; [Add all constants here that are not subject
;  to change or calculated from constants]
; Format: .EQU const = $ABCD
;
; =====
;   R E G I S T E R   D E F I N I T I O N S
; =====
;
; [Add all register names here, include info on
;  all used registers without specific names]
; Format: .DEF rmp = R16
; .DEF rmp = R16 ; Multipurpose register
;
; =====
;   S R A M   D E F I N I T I O N S
; =====
;
; .DSEG
; .ORG 0X0060
; Format: Label: .BYTE N ; reserve N Bytes from Label:
;
; =====
;   R E S E T   A N D   I N T   V E C T O R S
; =====
;
; .CSEG
; .ORG $0000
;     rjmp Main ; Reset vector
;     reti ; Int vector 1
;     reti ; Int vector 2
;     reti ; Int vector 3
;     reti ; Int vector 4
;     reti ; Int vector 5
;     reti ; Int vector 6
;     reti ; Int vector 7
;     reti ; Int vector 8
;     reti ; Int vector 9
;
; =====
;   I N T E R R U P T   S E R V I C E S
; =====
;
; [Add all interrupt service routines here]
;
; =====
;   M A I N   P R O G R A M   I N I T
; =====

```

```

;
Main:
; Init stack
    ldi rmp, LOW(RAMEND) ; Init LSB stack
    out SPL,rmp
; Init Port B
    ldi rmp,(1<<DDB2)|(1<<DDB1)|(1<<DDB0) ; Direction of Port B
    out DDRB,rmp
; [Add all other init routines here]
    ldi rmp,1<<SE ; enable sleep
    out MCUCR,rmp
    sei
;
; =====
;          P R O G R A M   L O O P
; =====
;
Loop:
    sleep ; go to sleep
    nop ; dummy for wake up
    rjmp loop ; go back to loop
;
; End of source code
;

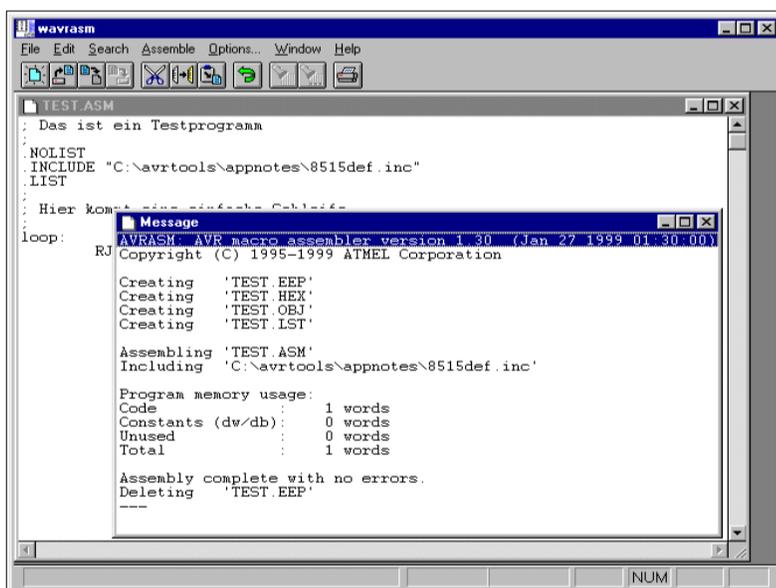
```

The assembler

Now we have a text file, with blank ASCII characters. The next step is to translate this code to a machine-oriented form well understood by the AVR chip. Doing this is called assembling, which means „put together the right instruction words“. The program that reads the text file and produces some kind of output files is called Assembler. In the easiest form this is a program for the instruction line that, when called, expects the address of the text file and some optional switches, and then starts assembling the instructions found in the text file.

If your editor allows calling external programs, this is an easy task. If not (another item on the wish list for the editor in Windows 2010), it is more convenient to write a short batch file (again using an editor). That batch file should have a line like this:

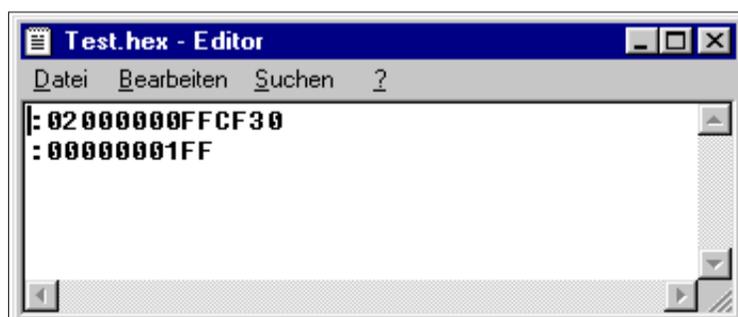
PathToAssembler\Assembler.exe -options PathToTextfile\Textfile.asm



Klicking on the editor's external program caller or on the batch file starts the command line assembler. That piece of software reports the complete translation process (in the smaller window), here with no errors. If errors occur these are notified, along with their type and line number. Assembling resulted in one word of code which resulted from the RJMP instruction that we used. Assembling our single asm text file now has produced four other files (not all apply here).

The first of these four new files, TEST.EEP, holds the content that should be written to the EEPROM of the AVR. This is not very interesting in our case, because we didn't program any content for the EEPROM. The assembler has therefore deleted this file when he

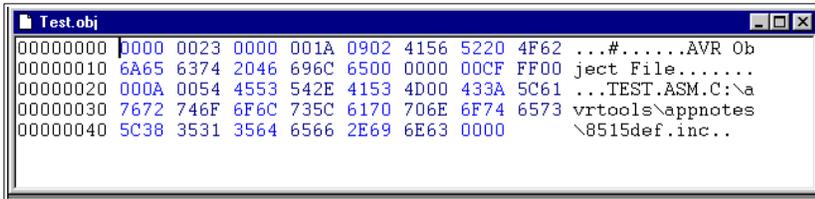
completed the assembly run, because it is empty.



The second file, TEST.HEX, is more relevant because this file holds the instructions later programmed into the AVR chip. This file looks like this.

The hex numbers are written in a special ASCII form, together with address informations and a checksum for each line. This form is called Intel-hex-format, and is very old and stems from the early world of computing. The form is well understood by

the programming software.



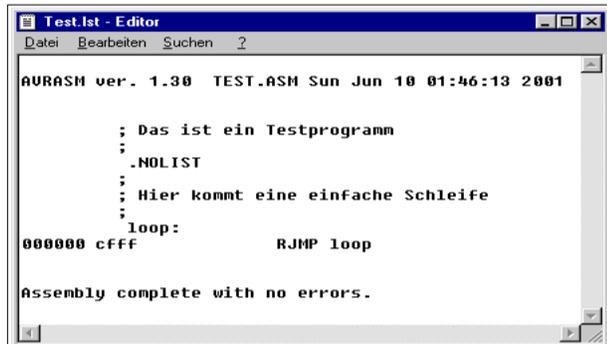
```

00000000 0000 0023 0000 001A 0902 4156 5220 4F62 ...#.....AVR Ob
00000010 6A65 6374 2046 696C 6500 0000 00CF FF00 ject File.....
00000020 000A 0054 4553 542E 4153 4D00 433A 5C61 ...TEST.ASM.C:\n
00000030 7672 746F 6F6C 735C 6170 706E 6F74 6573 vrtools\appnotes
00000040 5C38 3531 3564 6566 2E69 6E63 0000 \8515def.inc..

```

The third file, TEST.OBJ, will be introduced later, this file is needed to simulate an AVR. Its format is hexadecimal and defined by ATMEL. Using a hex-editor its content looks like this. Attention: This file format is not compatible with the programmer

software, don't use this file to program the AVR (a very common error when starting). OBJ files are only produced by certain ATMEL assemblers, don't expect these files with other assemblers.



```

AVRASM ver. 1.30 TEST.ASM Sun Jun 10 01:46:13 2001
; Das ist ein Testprogramm
;
; .NOLIST
;
; Hier kommt eine einfache Schleife
;
; loop:
000000 cfff RJMP loop
Assembly complete with no errors.

```

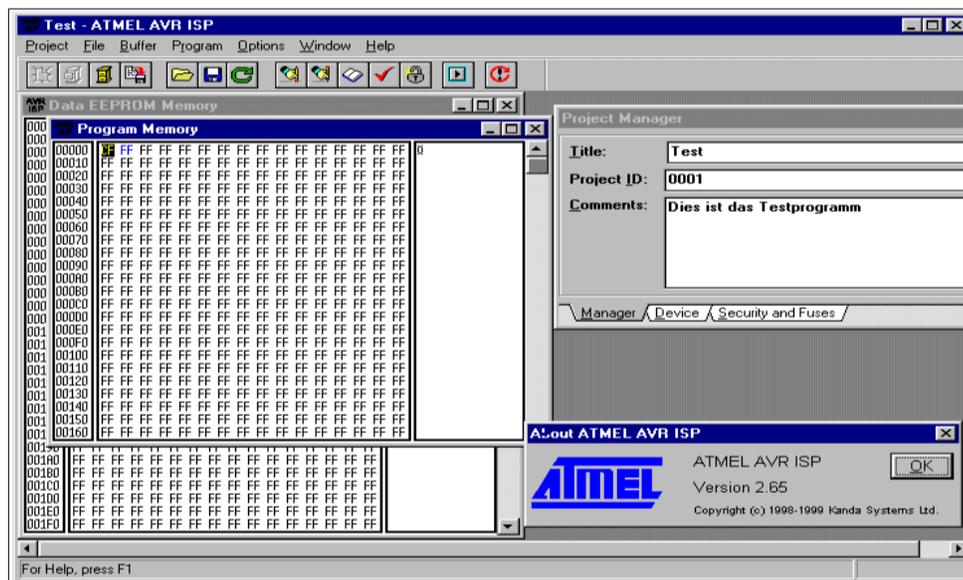
The fourth file, TEST.LST, is a text file. Display its content with a simple editor. The following results.

The program with all its addresses, instructions and error messages are displayed in a readable form. You will need that file in some cases to debug errors.

List files are generated only if the appropriate option is selected on the command line options and if the .NOLIST directive doesn't suppress listing.

Programming the chips

To program our hex code, as coded in text form in the .HEX-file, to the AVR a programmer software is necessary. This software reads the .HEX-file and transfers its content, either bit-by-bit (serial programming) or byte-by-byte (parallel programming) to the AVR's flash memory. We start the programmer software and load the hex file that we just generated.



In an example that looks like this. Please note: the displayed window stems from ISP.exe, a historic program no longer distributed by ATMEL. Other programmer software looks similar.

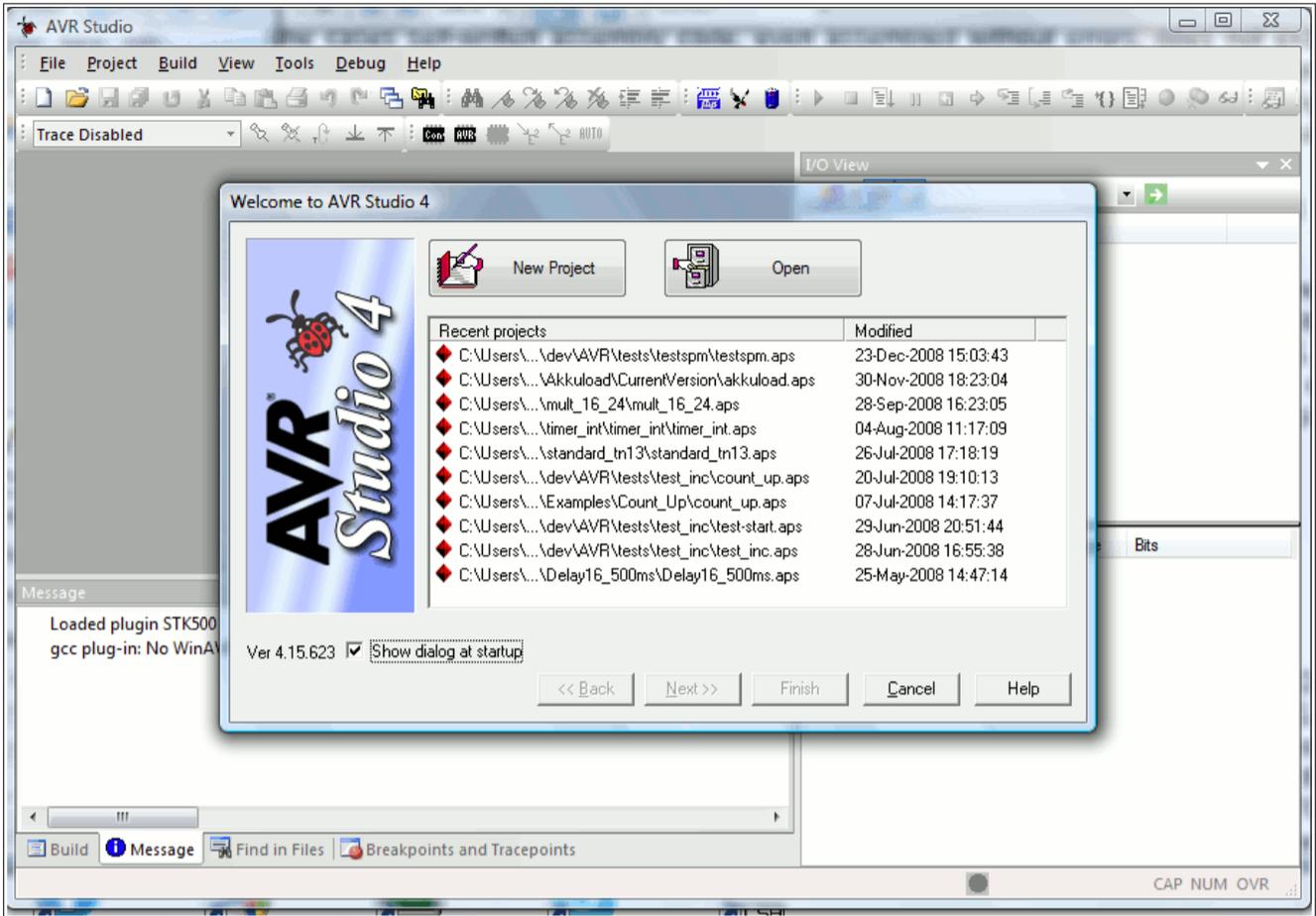
The software will burn our code in the chip's program store. There are a number of preconditions necessary for this step and several reasons possible, if this step fails. Consult your programmer software help, if problems occur.

Programming hardware and appropriate software alternatives for different PC operating systems are available on the Internet. As an example for programming over the PC's parallel or serial communication port, PonyProg2000 should be mentioned here.

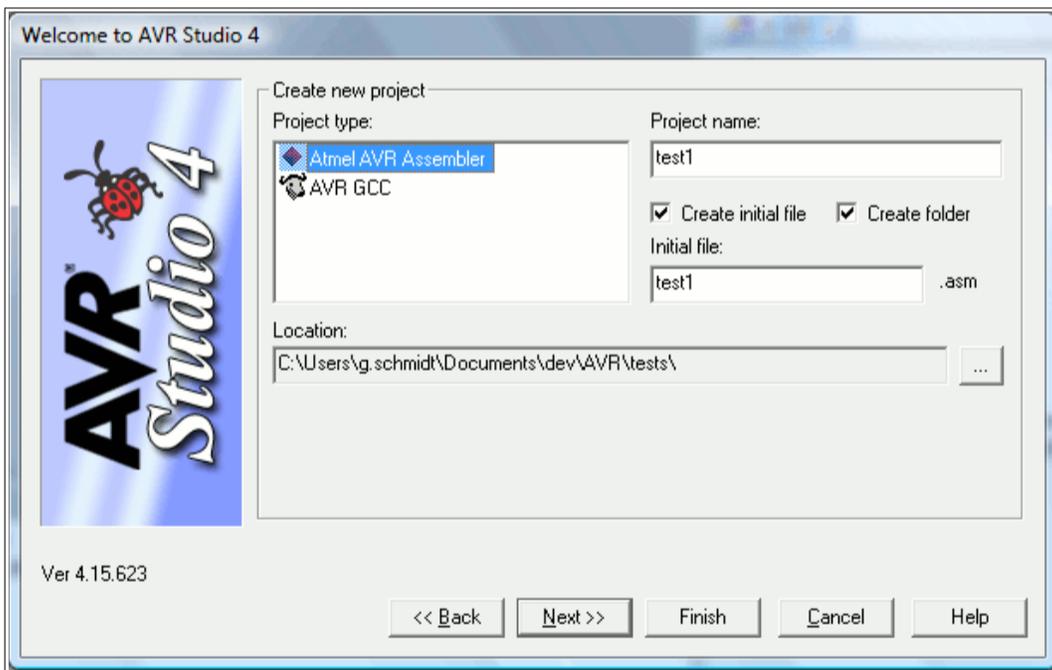
Simulation in the studio

In some cases self-written assembly code, even assembled without errors, does not exactly do what it should do when burned into the chip. Testing the software on the chip could be complicated, esp. if you have a minimum hardware and no opportunity to display interim results or debugging signals. In these cases the Studio software package from ATMEL provides ideal opportunities for debugging. Testing the software or parts of it is possible, the program code could be tested step-by-step displaying results.

The pictures shown here are taken from Version 4 of the Studio, that is available for free on ATMEL's website. Older versions looks different, but do nearly the same. The Studio is a software that has all you need to develop, debug, simulate and burn your assembler programs into the AVR type of your choice. The studio is started and looks like this.

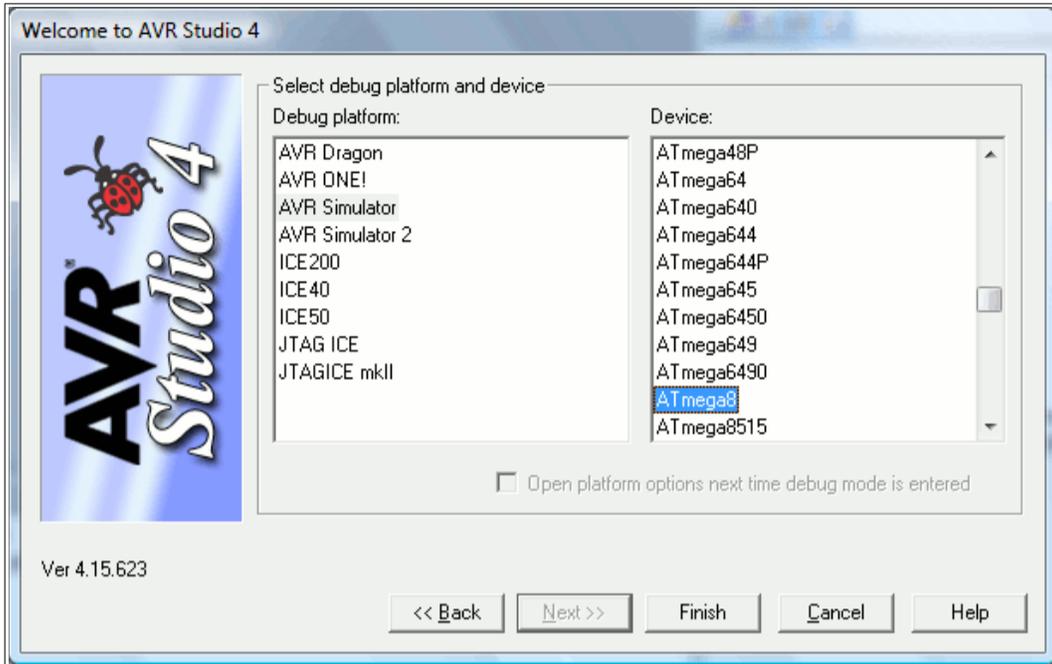


The first dialog asks whether an existing project should be opened or a new project is to be started. In case of a newly installed Studio “New Project” is the correct answer. The Button “Next>>” brings you to the settings dialog of your new project.

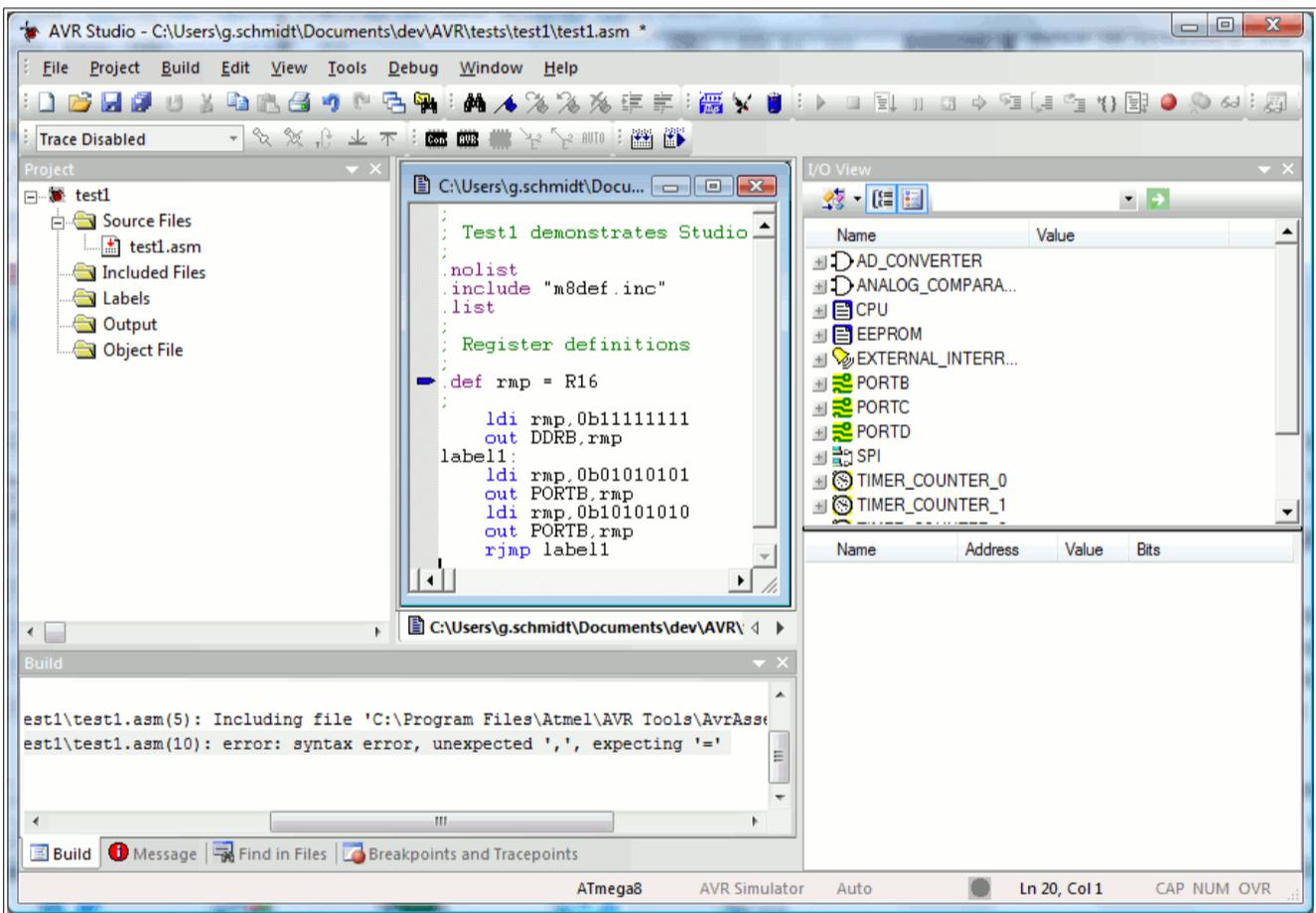


Here you select “Atmel AVR Assembler” as your project type, give that project a name (here “test1”) and let the Studio create an initial (empty) file for your source code, let it create a folder and select a location for that project, where you have write access to.

The button “Next>>” opens the platform and device selection dialog:



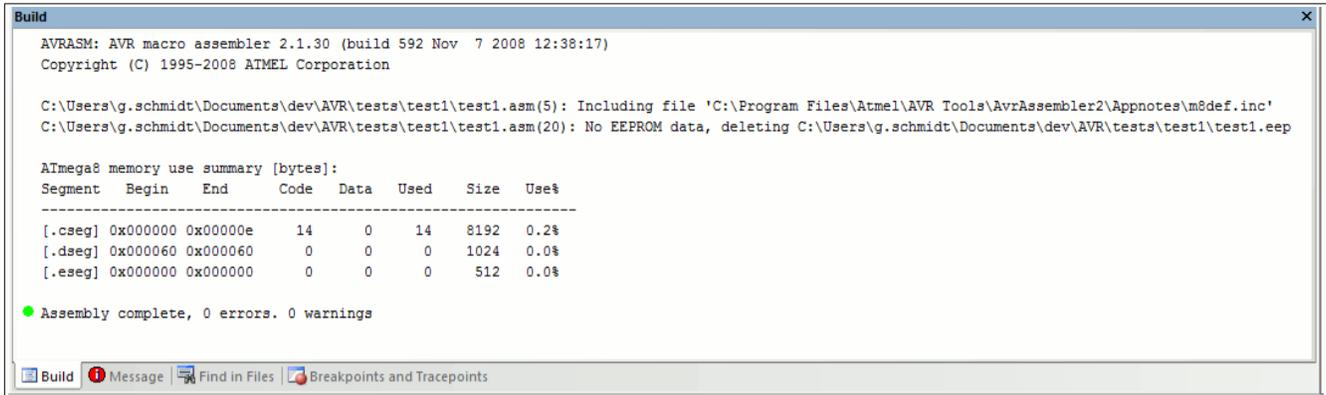
As debug platform select either “AVR simulator” or “AVR simulator 2”. As Device select your AVR type, here an ATmega8 was selected. If your desired type is grayed out, select another simulator platform. Close this window with the “Finish” button. Now a large window pops up, which has lots of different sub-windows.



On the left, the project window allows you to manipulate and view all your project files. In the middle, the editor window, allows you to write your source code (try typing its content to your editor window, don't care about the colors – these are added by the editor – remember syntax-highlighting?). On the left bottom is a “Build” section, where all your error messages go to. On the right side is a strange I/O view and below a rather white field, we'll come to that later on.

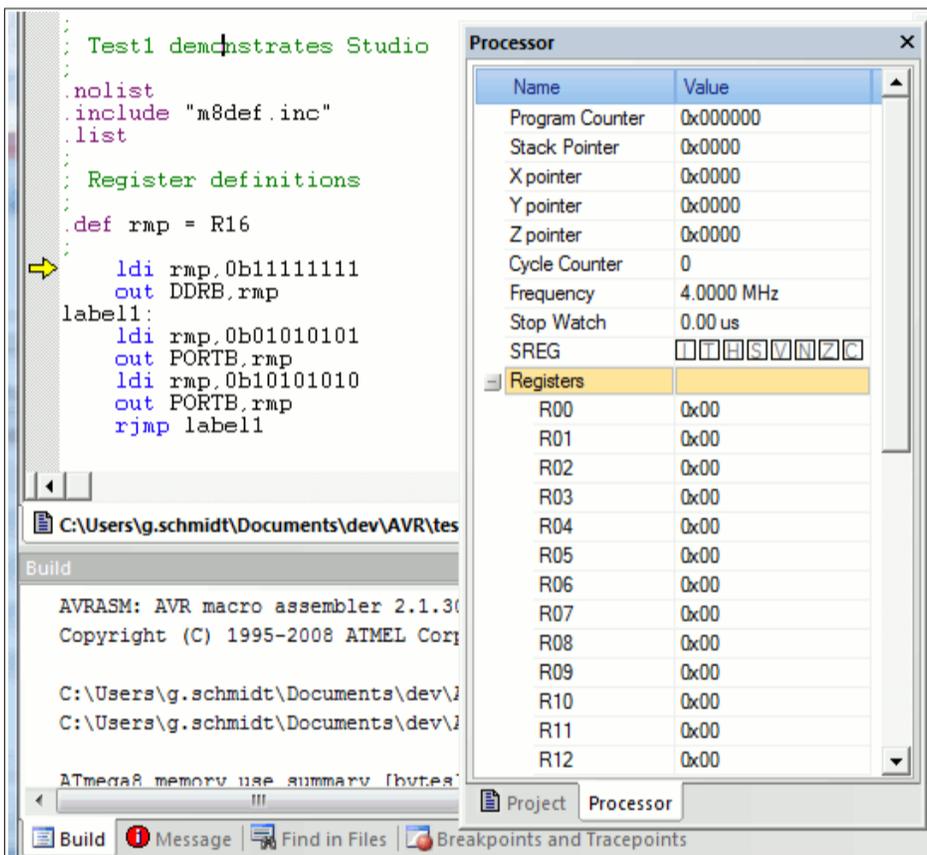
All window portions can be made larger and smaller and even can be shifted around on the screen. Try mixing these windows! The next pictures show some differently looking windows, but they are all the same as here.

After typing the source code shown above to your source file in the editor completely, push the menu “Build” and its sub-menu “Build”. If you typed correctly, the following shows up in your “Build” window:



Make sure, you read all window content once for the first time, because it gives you a lot more info besides the small green circle. All that should be fine, otherwise you typed errors into the code and the circle is red.

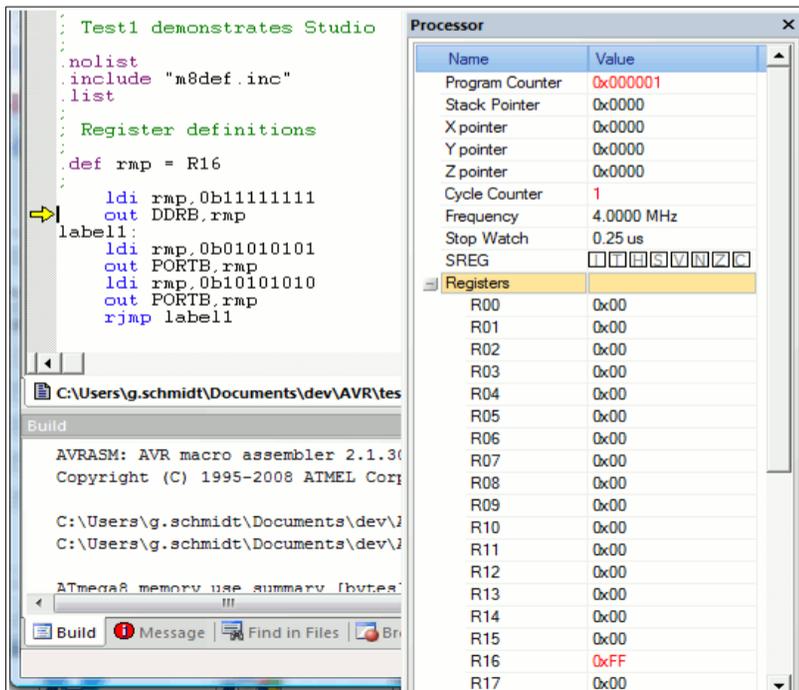
You can now push the menu item “Debug” and some windows change their content, size and position. If you also push the menu item “View”, “Toolbars” and “Processor” and shift around windows, it should look like this:



The former editor window has a yellow arrow now. This arrow points to the next instruction that will be executed (not really executed, but rather “simulated”).

The processor window shows the current program counter value (yes, the program starts at address 0), the stack pointer (no matter what that might be – wait for that later in the course), a cycle counter and a stop watch. If you push on the small “+” left to the word “Registers”, the content of the 32 registers is displayed (yes, they are all empty when you start the processor simulation).

Now let us proceed with the first instruction. Menu item “Debug” and “Step into” or simply F11 executes the first instruction.



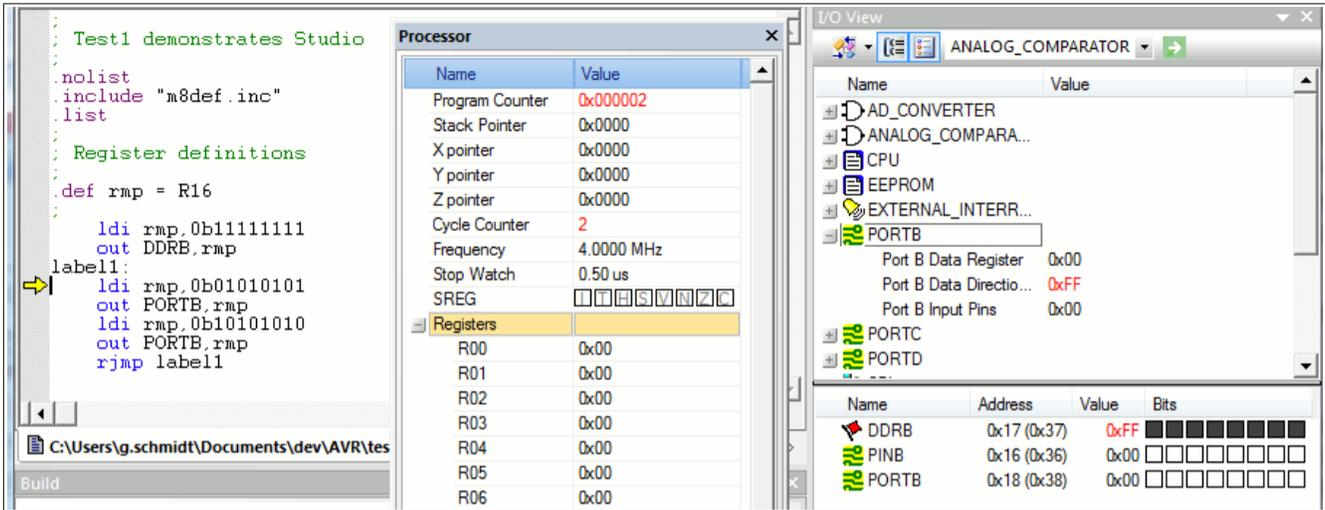
The instruction “ldi rmp,0b11111111” loads the binary value 1111.1111 to register R16. An instruction we will learn more about later on in the course.

The yellow arrow now has advanced one instruction down, is now at the OUT instruction.

In the processor window, the program counter and the cycle counter are both at 1 now.

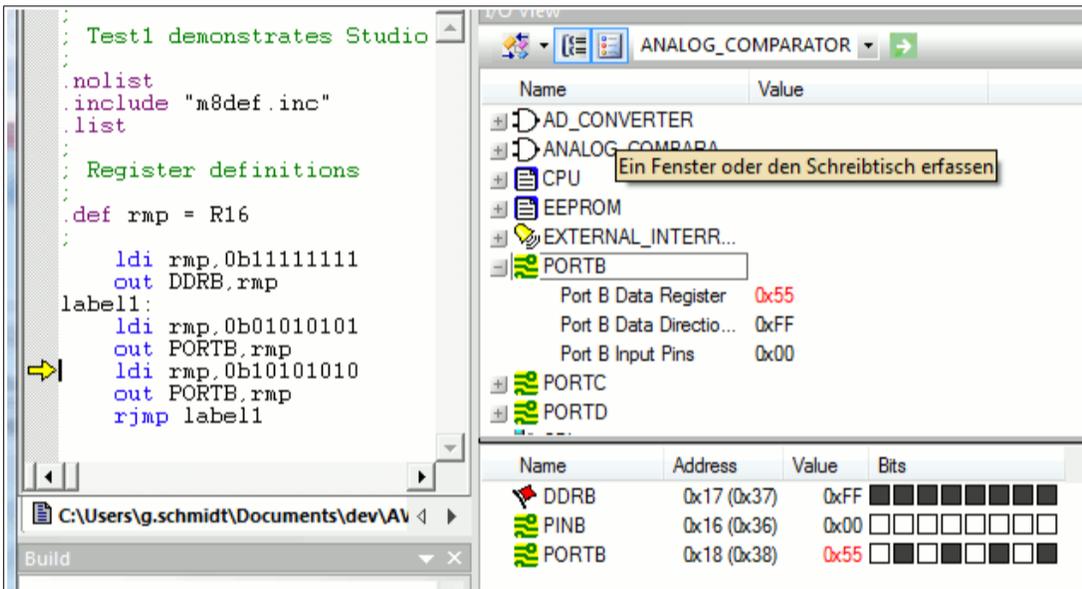
And register 16, down the list of registers, is red now and shows 0xFF, which is hexadecimal for binary 1111.1111.

To learn about another simulator window just advance simulation one step further to execute the OUT instruction (e. g. by pushing the key F11).



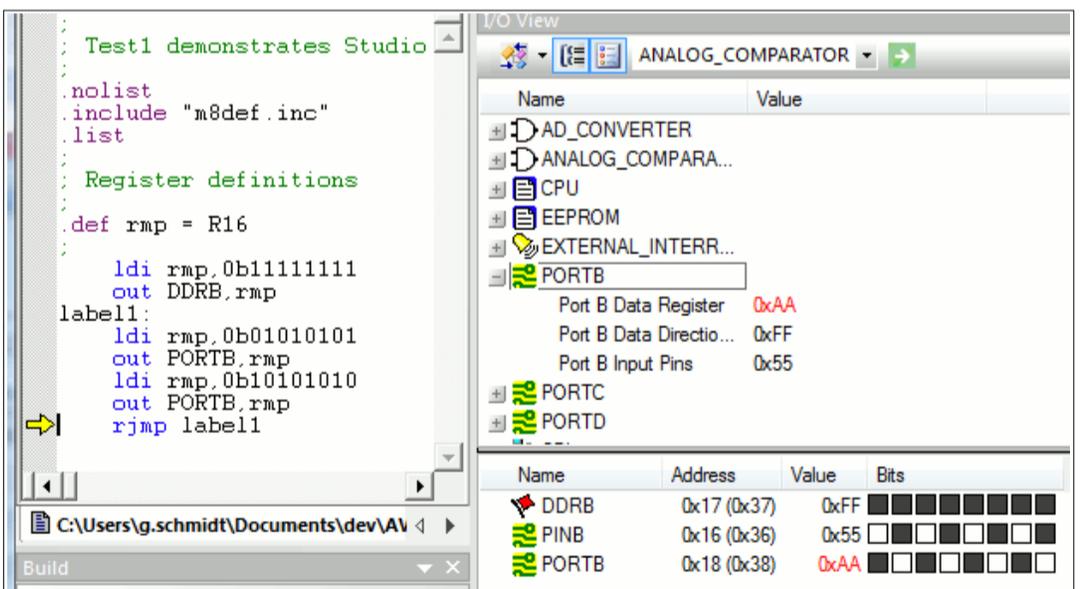
The instruction “Out DDRB,rmp” writes 0xFF to a port named DDRB. Now the action is on the I/O view window. If you push on PORTB and the small “+” left of it, this window displays the value 0xFF in the port DDRB in two different forms: as 0xFF in the upper window portion and as 8 black squares in the lower window section.

To make it even more black, we push F11 two times and write 0x55 to the port PORTB.



As expected, the port PORTB changes its content and has four black and four white squares now.

Another two F11, writing 0xAA to PORTB, changes the black and white squares to the opposite color.



All what has been expected, but what happened to port PINB? We didn't write something to PINB, but it has the opposite colors than PORTB, just like the colors before in PORTB.

PINB is an input port for external pins. Because the direction ports in DDRB are set to be outputs, PINB follows the pin status of PORTB, just one cycle

later. Nothing wrong here. If you like to check this, just press F11 several times and you see that this is correct.

That is our short trip through the simulator software world. The simulator is capable to much more, so it should be applied extensively in cases of design errors. Visit the different menu items, there is much more than can be shown here. In the mean time, instead of playing with the simulator, some basic things have to be learned about assembler language, so put the Studio aside for a while.

What is a register?

Registers are special storages with 8 bits capacity and they look like this:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-------	-------	-------	-------	-------	-------	-------	-------

Note the numeration of these bits: the least significant bit starts with zero (mathematically: $2^0 = 1$).

A register can either store numbers from 0 to 255 (positive number, no negative values), or numbers from -128 to +127 (whole number with a sign bit, located in bit 7), or a value representing an ASCII-coded character (e. g. 'A'), or just eight single bits that do not have something to do with each other (e. g. for eight single flags, used to signal eight different yes/no decisions).

The special character of registers, compared to other storage sites, is that

- they are connected directly to the central processing unit called the accumulator,
- they can be used directly in assembler instructions, either as target register for the result or as read register for a calculation or transfer,
- operations with their content require only a single instruction word.

There are 32 registers in an AVR. They are originally named R0 to R31, but you can choose to name them to more meaningful ones using a so-called assembler directive. An example:

```
.DEF MyPreferredRegister = R16
```

Assembler directives always start with a dot. Instructions or labels do NEVER start with a dot. Note that assembler directives like this are only meaningful for the assembler but do not produce any code that is executable in the AVR target chip. The name "MyPreferredRegister" will not show up in the assembled hex code, and therefore this name cannot be derived from that hex code.

Instead of using the register name R16 we can now use our own name "MyPreferredRegister", if we want to use R16 within an instruction. So we write a little bit more text each time we use this register, but we have an association what might be the content of this register.

Using the instruction line

```
LDI MyPreferredRegister, 150
```

which means: load the number 150 immediately to the register R16, Load Immediate. This loads a fixed value or a constant to that register. Following the assembly, or translation of this code into binary or hex, the program storage written to the AVR chip looks like this:

```
000000 E906
```

This will show up in the listing, a file called *.lst produced by the assembler software, which is a simple text file. All numbers are in hex format: The first hex number is the address (000000), where the instruction is written to in the program flash memory of the AVR, the second is the instruction code (E906). E906 tells the processor three different things in one word, even if you don't see this directly:

- a basic load instruction code, that stands for LDI,
- the target register (R16) where the value 150 is to be written to,
- the value of the constant (150).

Don't be afraid: you don't have to remember this coding because the assembler knows how to translate all this to finally yield E906 and the AVR executes it.

Within one instruction two different registers can play a role. The easiest instruction of this type is the copy instruction, MOV. The naming of this instruction MOV deserves a price for the most confusing definition, because the content of a register cannot be moved (what would be left in a register, if you MOVE its content to somewhere else?). It should better be named COPY, because it copies the content of one register to another register. Like this:

```
.DEF MyPreferredRegister = R16
.DEF AnotherRegister = R15
LDI MyPreferredRegister, 150
MOV AnotherRegister, MyPreferredRegister
```

The first two lines of this monster program are directives that define the new names of the registers R16 and R15 for the assembler. Again, these lines do not produce any code for the AVR. The instruction lines with LDI and MOV produce code:

```
000000 E906
000001 2F01
```

The instruction write the value 150 into register R16 and copy its content to the target register R15. Very IMPORTANT NOTICE:

The first register is always the target register where the result is written to!

(This is unfortunately different from what one expects or from how we speak, think and write – left to right. It is a simple convention, probably inspired by some Asian languages where writing is from right to left.

That was once defined that way to confuse the beginners learning assembler. That is why assembly language is that complicated.)

Different registers

The beginner might want to write the above instructions like this:

```
.DEF AnotherRegister = R15
    LDI AnotherRegister, 150
```

And: you lost. Only the registers from R16 to R31 load a constant immediately with the LDI instruction, R0 to R15 don't do that. This restriction is not very fine, but could not be avoided during construction of the instruction set for the AVRs.

There is one exception from that rule: setting a register to Zero. This instruction

```
CLR MyPreferredRegister
```

is valid for all registers.

Besides the LDI instruction you will find this register class restriction with the following additional instructions:

- ANDI Rx,K ; Bit-And of register Rx with a constant value K,
- CBR Rx,M ; Clear all bits in register Rx that are set to one within the constant mask value M,
- CPI Rx,K ; Compare the content of the register Rx with a constant value K,
- SBCI Rx,K ; Subtract the constant K and the current value of the carry flag from the content of register Rx and store the result in register Rx,
- SBR Rx,M ; Set all bits in register Rx to one, that are one in the constant mask M,
- SER Rx ; Set all bits in register Rx to one (equal to LDI Rx,255),
- SUBI Rx,K ; Subtract the constant K from the content of register Rx and store the result in register Rx.

In all these instructions the register must be between R16 and R31! If you plan to use these instructions you should select one of these registers for that operation. It is shorter and easier to program. This is an additional reason why you should use the directive to define a register's name, because you can easier change the registers location later on, if required.

Pointer-registers

A very special extra role is defined for the register pairs R27:R26, R29:R28 and R31:R32. The role is so important that these pairs have extra short names in AVR assembler: X, Y and Z. These short names are understood by the assembler. These pairs are 16-bit pointer registers, able to point to addresses with max. 16 bit length, e. g. into SRAM locations (X, Y or Z) or into locations in program memory (Z).

Accessing memory locations with pointers

The lower byte of the 16-bit-address is located in the lower register, the higher byte in the upper register. Both parts have their own names, e.g. the higher byte of Z is named ZH (=R31), the lower Byte is ZL (=R30). These names are defined within the assembler. Dividing a 16-bit-word constant into its two different bytes and writing these bytes to a pointer register is done like follows:

```
.EQU address = RAMEND ; RAMEND is the highest 16-bit address in SRAM, defined in the *def.inc header file,
    LDI YH,HIGH(address) ; Load the MSB of address
    LDI YL,LOW(address) ; Load the LSB of address
```

Accesses via pointer registers are programmed with specially designed instructions. Read access is named LD (LoaD), write access named ST (STore), e. g. with the X-pointer:

Similarly you can use Y and Z for that purpose.

Pointer	Sequence	Examples
X	Read/Write from address X, don't change the pointer	LD R1,X or ST X,R1
X+	Read/Write from/to address X, and increment the pointer afterwards by one	LD R1,X+ or ST X+,R1
-X	First decrement the pointer by one and read/write from/to the new address afterwards	LD R1,-X or ST -X,R1

Reading program flash memory with the Z pointer

There is only one instruction for the read access to the program storage space. It is defined for the pointer

pair Z and it is named LPM (Load from Program Memory). The instruction copies the byte at program flash address Z to the register R0. As the program memory is organized word-wise (one instruction on one address consists of 16 bits or two bytes or one word) the least significant bit selects the lower or upper byte (0=lower byte, 1= upper byte). Because of this the original address must be multiplied by 2 and access is limited to 15-bit or 32 kB program memory. Like this:

```
LDI ZH,HIGH(2*address)
LDI ZL,LOW(2*address)
LPM
```

Following this instruction the address must be incremented to point to the next byte in program memory. As this is used very often a special pointer incrementation instruction has been defined to do this:

```
ADIW ZL,1
LPM
```

ADIW means ADD Immediate Word and a maximum of 63 can be added this way. Note that the assembler expects the lower of the pointer register pair ZL as first parameter. This is somewhat confusing as addition is done as 16-bit- operation.

The complement instruction, subtracting a constant value of between 0 and 63 from a 16-bit pointer register is named SBIW, Subtract Immediate Word. (SuBtract Immediate Word). ADIW and SBIW are possible for the pointer register pairs X, Y and Z and for the register pair R25:R24, that does not have an extra name and does not allow access to SRAM or program memory locations. R25:R24 is ideal for handling 16-bit values.

In some later types of AVR the automatic incrementation of Z following the LPM instruction has an extra instruction, LPM Z+. Please consult the instruction list in the data sheet of your AVR type to see if this applies to the type you are working with.

Tables in the program flash memory

Now that you know how to read from flash memory you might wish to place a list of constants or a string of text to the flash and read these. How to insert that table of values in the program memory? This is done with the assembler directives .DB and .DW. With that you can insert byte wise or word wise lists of values. Byte wise organized lists look like this:

```
.DB 123,45,67,89 ; a list of four bytes, written in decimal form
.DB "This is a text. " ; a list of byte characters, written as text
```

You should always place an even number of bytes on each single line. Otherwise the assembler will add a zero byte at the end, which might be unwanted.

The similar list of words looks like this:

```
.DW 12345,6789 ; a list of two word constants
```

Instead of constants you can also place labels (e. g. jump targets) on that list, like that:

```
Label1:
[ ... here are some instructions ... ]
Label2:
[ ... here are some more instructions ... ]
Table:
.DW Label1,Label2 ; a word wise list of labels
```

Labels should start in column 1, but have to be ending with a ":". Note that reading the labels from that table with LPM (and subsequent incrementation of the pointer) first yields the lower byte of the word, then the upper byte.

Accessing registers with pointers

A very special application for the pointer registers is the access to the registers themselves. The registers are located in the first 32 bytes of the chip's address space (at address 0x0000 to 0x001F). This access is only meaningful if you have to copy the register's content to SRAM or EEPROM or read these values from there back into the registers. More common for the use of pointers is the access to tables with fixed values in the program memory space. Here is, as an example, a table with 10 different 16-bit values, where the fifth table value is read to R25:R24:

```
MyTable:
.DW 0x1234,0x2345,0x3456,0x4568,0x5678 ; The table values, word wise
.DW 0x6789,0x789A,0x89AB,0x9ABC,0xABCD ; organized
Read5: LDI ZH,HIGH(MyTable*2) ; address of table to pointer Z
      LDI ZL,LOW(MyTable*2) ; multiplied by 2 for bitwise access
      ADIW ZL,10 ; Point to fifth value in table
      LPM ; Read least significant byte from program memory
      MOV R24,R0 ; Copy LSB to 16-bit register
      ADIW ZL,1 ; Point to MSB in program memory
      LPM ; Read MSB of table value
      MOV R25,R0 ; Copy MSB to 16-bit register
```

This is only an example. You can calculate the table address in Z from some input value, leading to the

respective table values. Tables can be organized byte- or character-wise, too.

Recommendation for the use of registers

The following recommendations, if followed, decide if you are an effective assembler programmer:

- Define names for registers with the `.DEF` directive, never use them with their direct name `Rx`.
- If you need pointer access reserve `R26` to `R31` for that purpose.
- A 16-bit-counter is best located in `R25:R24`.
- If you need to read from the program memory, e. g. fixed tables, reserve `Z` (`R31:R30`) and `R0` for that purpose.
- If you plan to have access to single bits within certain registers (e. g. for testing flags), use `R16` to `R23` for that purpose.
- Registers necessary for math are best placed to `R1` to `R15`.
- If you have more than enough registers available, place all your variables in registers.
- If you get short in registers, place as many variables as necessary to SRAM.

Ports

What is a Port?

Ports in the AVR are gates from the central processing unit to internal and external hard- and software components. The CPU communicates with these components, reads from them or writes to them, e. g. to the timers or the parallel ports. The most used port is the flag register, where flags from previous operations are written to and branching conditions are read from.

There are 64 different ports, which are not physically available in all different AVR types. Depending on the storage space and other internal hardware the different ports are either available and accessible or not. Which of the ports can be used in a certain AVR type is listed in the data sheets for the processor type. Larger ATmega and ATXmega have more than 64 ports, access to the ports beyond #63 is different then (see below).

Ports have a fixed address, over which the CPU communicates. The address is independent from the type of AVR. So e.g. the port address of port B is always 0x18 (0x stands for hexadecimal notation, 0x18 is decimal 24). You don't have to remember these port addresses, they have convenient aliases. These names are defined in the include files (header files) for the different AVR types, that are provided from the producer. The include files have a line defining port B's address as follows:

```
.EQU PORTB, 0x18
```

So we just have to remember the name of port B, not its location in the I/O space of the chip. The include file 8515def.inc is involved by the assembler directive

```
.INCLUDE "C:\Somewhere\8515def.inc"
```

and the registers of the 8515 are all defined there and easily accessible.

Ports usually are organized as 8-bit numbers, but can also hold up to 8 single bits that don't have much to do with each other. If these single bits have a meaning they have their own name associated in the include file, e. g. to enable the manipulation of a single bit. Due to that name convention you don't have to remember these bit positions. These names are defined in the data sheets and are given in the include file, too. They are provided here in the port tables.

Write access to ports

As an example the MCU General Control Register, called MCUCR, consists of a number of single control bits that control the general property of the chip. Here are the details of port MCUCR in the AT90S8515, taken from the device data book. Other ports look similar.

Bit	7	6	5	4	3	2	1	0	
\$35 (\$55)	SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

It is a port, fully packed with 8 control bits with their own names (ISC00, ISC01, ...). Those who want to send their AVR to a deep sleep need to know from the data sheet how to set the respective bits. Like this:

```
.DEF MyPreferredRegister = R16
LDI MyPreferredRegister, 0b00100000
OUT MCUCR, MyPreferredRegister
SLEEP
```

The Out instruction brings the content of my preferred register, a Sleep-Enable-Bit called SE, to the port MCUCR. SE enables the AVR to go to sleep, whenever the SLEEP instruction shows up in the code. As all the other bits of MCUCR are also set by the above instructions and the Sleep Mode bit SM was set to zero, a mode called half-sleep will result: no further instruction execution will be performed but the chip still reacts to timer and other hardware interrupts. These external events interrupt the big sleep of the CPU if they feel they should notify the CPU.

The above formulation is not very transparent, because "0b00100000" is not easy to remember, and no one sees easily what bit exactly has been set to one by this instruction. So it is a good idea to formulate the LDI instruction as follows:

```
LDI MyPreferredRegister, 1<<SE
```

This formulation tells the assembler to

- take a one ("1"),
- to read the bit position of the Sleep Enable bit ("SE") from the symbol list, as defined in the header file 8515def.inc, which yields a value of "5" in that case,
- to shift ("<<") the "1" five times left ("1<<5"), in steps:

1. initial: 0000.0001,
2. first shift left: 0000.0010,
3. second shift left: 0000.0100, and so on until
4. fifth shift left: 0010.0000.

- to associate this value to *MyPreferredRegister* and to insert this LDI instruction into the code.

To make it clear again: This shifting is done by the assembler software only, not within the code in the AVR. It is pure convention to increase the readability of the assembler source text.

How does this change, if you want to set the Sleep Mode bit ("SM") and the Sleep Enable bit ("SE") within the same LDI instruction? SM=1 and SE=1 enables your AVR to react to a SLEEP instruction by going to a big sleep, so only do this if you understand what the consequences are. The formulation is like this:

```
LDI MyPreferredRegister, (1<<SM) | (1<<SE)
```

Now, the assembler first calculates the value of the first bracket, (1<<SM), a "1" shifted four times left (because SM is 4) and that yields 0001.0000, then calculates the second bracket, (1<<SE), a "1" shifted five times left (because SE is 5). The "|" between the two brackets means BIT-OR the first and the second value, each bit one by one. The result of doing this with 0001.0000 and 0010.0000 in that case is 0011.0000, and that is our desired value for the LDI instruction. Even though the formulation

$$(1 \ll SM) | (1 \ll SE)$$

might, on the first look, not be more transparent than the resulting value

```
0011.0000
```

for a beginner, it is easier to understand which bits of MCUCR are intended to be manipulated in this LDI instruction. Especially if you have to read and understand your code some months later, SM and SE are a better hint that the Sleep Mode and Enable bits are targeted here. Otherwise you would have to consult the device's data book much more often.

Read access to ports

Reading a port's content is in most cases possible using the IN instruction. The following sequence

```
.DEF MyPreferredRegister = R16
    IN MyPreferredRegister, MCUCR
```

reads the bits in port MCUCR to the register named *MyPreferredRegister*. As many ports have undefined and unused bits in certain ports, these bits always read back as zeros.

More often than reading all 8 bits of a port one must react to a certain status bit within a port. In that case we don't need to read the whole port and isolate the relevant bit. Certain instructions provide an opportunity to execute instructions depending on the level of a certain bit of a port (see the JUMP section).

Read-Modify-Write access to ports

Setting or clearing certain bits of a port, without changing the other port bits, is also possible without reading and writing the other bits in the port. The two instructions are SBI (Set Bit I/O) and CBI (Clear Bit I/O). Execution is like this:

```
.EQU ActiveBit=0 ; The bit that is to be changed
    SBI PortB, ActiveBit ; The bit "ActiveBit" will be set to one
    CBI PortB, ActiveBit ; The bit "ActiveBit" will be cleared to zero
```

These two instructions have a limitation: only ports with an address smaller than 0x20 can be handled, ports above cannot be accessed that way. Because MCUCR in the above examples is at hex address \$38, the sleep mode and enable bits can't be set or cleared that way. But all the port bits controlling external pins (PORTx, DDRx, PINx) are accessible that way.

Memory mapped port access

For the more exotic programmer and the "elephant-like" ATmega and ATXmega (where ATMEL ran out of accessible port addresses): the ports can also be accessed using SRAM access instructions, e.g. ST and LD. Just add 0x20 to the port's address (remember: the first 32 addresses are associated to the registers!) and access the port that way. Like demonstrated here:

```
.DEF MyPreferredRegister = R16
    LDI ZH,HIGH(PORTB+32)
    LDI ZL,LOW(PORTB+32)
    LD MyPreferredRegister,Z
```

That only makes sense in certain cases, because it requires more instructions, execution time and assembler lines, but it is possible. It is also the reason why the first address location of the SRAM is 0x60 or 0x100 in some larger AVR types.

Details of relevant ports in the AVR

The following table holds the most used ports in a “small” AT90S8515. Not all ports are listed here, some of the MEGA and AT90S4434/8535 types are skipped. If in doubt see the original reference.

Component	Port name	Port-Register
Accumulator	SREG	Status Register
Stack	SPL/SPH	Stackpointer
External SRAM/External Interrupt	MCUCR	MCU General Control Register
External Interrupts	GIMSK	Interrupt Mask Register
	GIFR	Interrupt Flag Register
Timer Interrupts	TIMSK	Timer Interrupt Mask Register
	TIFR	Timer Interrupt Flag Register
8-bit Timer 0	TCCR0	Timer/Counter 0 Control Register
	TCNT0	Timer/Counter 0
16-bit Timer 1	TCCR1A	Timer/Counter Control Register 1 A
	TCCR1B	Timer/Counter Control Register 1 B
	TCNT1	Timer/Counter 1
	OCR1A	Output Compare Register 1 A
	OCR1B	Output Compare Register 1 B
	ICR1L/H	Input Capture Register
Watchdog Timer	WDTCR	Watchdog Timer Control Register
EEPROM Access	EEAR	EEPROM address Register
	EEDR	EEPROM Data Register
	EECR	EEPROM Control Register
Serial Peripheral Interface SPI	SPCR	Serial Peripheral Control Register
	SPSR	Serial Peripheral Status Register
	SPDR	Serial Peripheral Data Register
Serial Communication UART	UDR	UART Data Register
	USR	UART Status Register
	UCR	UART Control Register
	UBRR	UART Baud Rate Register
Analog Comparator	ACSR	Analog Comparator Control and Status Register
I/O-Ports	PORTx	Port Output Register
	DDRx	Port Direction Register
	PINx	Port Input Register

The status register as the most used port

By far the most often used port is the status register with its 8 bits. Usually access to this port is only by automatic setting and clearing bits by the CPU or accumulator, some access is by reading or branching on certain bits in that port, in a few cases it is possible to manipulate these bits directly (using the assembler instructions SE_x or CL_x, where x is the bit abbreviation). Most of these bits are set or cleared by the accumulator through bit-test, compare- or calculation-operations.

The most used bits are:

- Z: If set to one, the previous instruction yielded a zero result.
- C: If set to one, the previous instruction caused a carry of the most significant bit.

The following list has all assembler instructions that set or clear status bits depending on the result of the previous instruction execution.

Bit	Calculation	Logic	Compare	Bits	Shift	Other
Z	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR Z, BSET Z, CLZ, SEZ, TST	ASR, LSL, LSR, ROL, ROR	CLR

Bit	Calculation	Logic	Compare	Bits	Shift	Other
C	ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW	COM, NEG	CP, CPC, CPI	BCLR C, BSET C, CLC, SEC	ASR, LSL, LSR, ROL, ROR	-
N	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR N, BSET N, CLN, SEN, TST	ASR, LSL, LSR, ROL, ROR	CLR
V	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR V, BSET V, CLV, SEV, TST	ASR, LSL, LSR, ROL, ROR	CLR
S	SBIW	-	-	BCLR S, BSET S, CLS, SES	-	-
H	ADD, ADC, SUB, SUBI, SBC, SBCI	NEG	CP, CPC, CPI	BCLR H, BSET H, CLH, SEH	-	-
T	-	-	-	BCLR T, BSET T, BST, CLT, SET	-	-
I	-	-	-	BCLR I, BSET I, CLI, SEI	-	RETI

Port details

Port details of the most common ports are shown in an extra table (see annex).

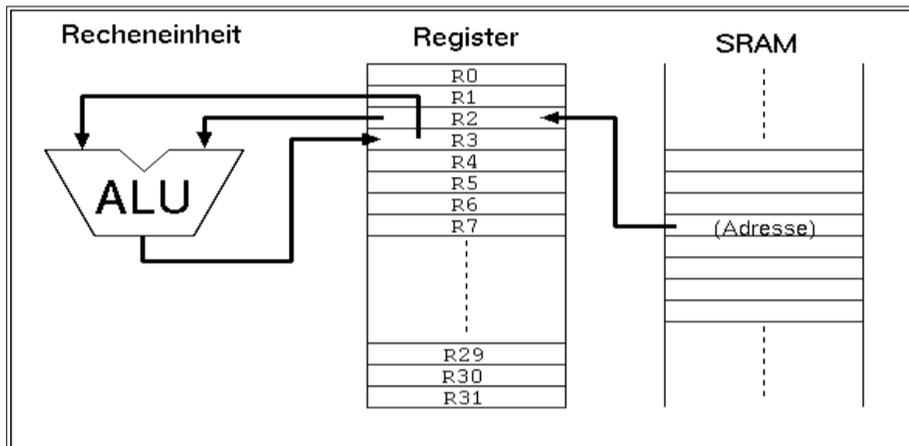
SRAM

Using SRAM in AVR assembler language

Nearly all AVR-types have static RAM (SRAM) on board (only very few old devices don't). Only very simple assembler programs can avoid using this memory space by putting all necessary information into registers. If you run out of registers you should be able to program the SRAM to utilize more space.

What is SRAM?

SRAM are memories that are not directly accessible by the central processing unit (Arithmetic and Logical



Unit ALU, sometimes called accumulator) like the registers are. If you access these memory locations you usually use a register as interim storage. In the example displayed here a value in SRAM will be copied to the register R2 (1st instruction), a calculation with the value in R3 is made and the result is written to R3 (second instruction). After that this value is written back to the same SRAM location (instruction 3, not shown here).

So it is clear that operations with values stored in the SRAM are slower to perform than those using registers alone. On the other hand: even the smallest AVR types have 128 bytes of SRAM available, much more than the 32 registers can hold.

The types from the old AT90S8515 upwards offer the additional opportunity to connect additional external RAM, expanding the internal 512 bytes. From the assembler point-of-view, external SRAM is accessed like internal SRAM. No extra instructions must be learned for accessing that external SRAM.

For what purposes can I use SRAM?

Besides simple storage of values, SRAM offers additional opportunities for its use. Not only access with fixed addresses is possible, but also the use of pointers, so that floating access to subsequent locations in SRAM can be programmed. This way you can build up ring buffers for interim storage of values or calculated (variable) tables. This is not very often used with registers, because they are too few and prefer fixed access.

Even more relative is the access using an offset to a fixed starting address in one of the pointer registers. In that case a fixed address is stored in a pointer register, a constant value is added to this address and read/write access is made to that address with an offset. With that kind of access, tables are very more effective.

But the most relevant use for SRAM is the so-called stack. You can push values (variables) to that stack. Be it the content of a register, that is temporarily needed for another purpose. Be it a return address prior to calling a subroutine, or the return address prior to a hardware-triggered interrupt.

How to use SRAM?

Direct addressing

To copy a value to a memory location in SRAM you have to define the address. The SRAM addresses you can use reach from the start address (very often 0x0060 in smaller AVRs, 0x0100 in larger ATmega) to the end of the physical SRAM on the chip (in the AT90S8515 the highest accessible internal SRAM location is 0x025F, see the device data sheet of your AVR type for more details on this).

With the instruction

```
STS 0x0060, R1
```

the content of register R1 is copied to the first SRAM location in address 0x0060. With

```
LDS R1, 0x0060
```

the SRAM content at address 0x0060 is copied to the register. This is the direct access with an address that has to be defined by the programmer.

The symbols defined in the *def.inc include file, SRAM_START and RAMEND, allow to place your variables within the SRAM space. So it is better to use these definitions to access the 15th memory byte,

like this:

```
LDS R1,SRAM_START+15
```

Symbolic names can be used to avoid handling fixed addresses, that require a lot of work, if you later want to change the structure of your data in the SRAM. These names are easier to handle than hex numbers, so give that address a name like:

```
.EQU MyPreferredStorageCell = SRAM_START
STS MyPreferredStorageCell, R1
```

Yes, it isn't shorter, but easier to remember. Use whatever name that you find to be convenient.

Pointer addressing

Another kind of access to SRAM is the use of pointers. You need two registers for that purpose, that hold the 16-bit address of the location. As we learned in the Pointer-Register-Division, pointer registers are the register pairs X (XH:XL, R27:R26), Y (YH:YL, R29:R28) and Z (ZH:ZL, R31:R30). They allow access to the location they point to directly (e. g. with ST X, R1), after prior decrementing the address by one (e. g. ST - X, R1) or with subsequent auto-incrementation of the address (e. g. ST X+, R1). A complete access to three cells in a row looks like this:

```
.EQU MyPreferredStorageCell = SRAM_START
.DEF MyPreferredRegister = R1
.DEF AnotherRegister = R2
.DEF AndYetAnotherRegister = R3
LDI XH, HIGH(MyPreferredStorageCell)
LDI XL, LOW(MyPreferredStorageCell)
LD MyPreferredRegister, X+
LD AnotherRegister, X+
LD AndYetAnotherRegister, X
```

Easy to operate, those pointers. And as easy as in other languages than assembler, that claim to be easier to learn.

Pointer with offset

The third construction is a little bit more exotic and only experienced programmers use this in certain cases. Let's assume we very often in our program need to access three consecutive SRAM locations. Let's further assume that we have a spare pointer register pair, so we can afford to use it exclusively for our purpose. If we would use the ST/LD instructions we always have to change the pointer if we access another location of the three. Not very convenient.

To avoid this, and to confuse the beginner, the access with offset was invented. During that access the register value isn't changed. The address is calculated by temporarily adding the fixed offset. In the above example the access to location 0x0062 would look like this. First, the pointer register is set to our central location SRAM_START:

```
.EQU MyPreferredStorageCell = SRAM_START
.DEF MyPreferredRegister = R1
LDI YH, HIGH(MyPreferredStorageCell)
LDI YL, LOW(MyPreferredStorageCell)
```

Somewhere later in the program I'd like to write to cell 2 above SRAM_START:

```
STD Y+2, MyPreferredRegister
```

The corresponding instruction for reading from SRAM with an offset

```
LDD MyPreferredRegister, Y+2
```

is also possible.

Note that the 2 is not really added to Y, just temporarily during the execution of this instruction. To confuse you further, this can only be done with the Y- and Z-register-pair, not with the X-pointer!

Of about 100 cases, the use of this opportunity is more effective in one single case. So don't care if you don't understand this in detail. It is only for experts, and only necessary in a few cases.

That's it with the SRAM, but wait: the most relevant use as stack is still to be learned.

Use of SRAM as stack

The most common use of SRAM is its use as stack. The stack is a tower of wooden blocks. Each additional block goes onto the top of the tower, each recall of a value removes the most upper block from the tower. Removal of blocks from the base or from any lower portion of the tower is too complicated and confuses your whole tower, so never try this. This structure is called Last-In-First-Out (LIFO) or easier: the last to go on top will be the first coming down from the top.

Defining SRAM as stack

To use SRAM as stack requires the setting of the stack pointer first. The stack pointer is a 16-bit-pointer, accessible like a port. The double register is named SPH:SPL. SPH holds the most significant address byte, SPL the least significant. This is only true, if the AVR type has more than 256 byte SRAM. If not, SPH is not necessary, is undefined, and must not and cannot be used. We assume we have more than 256 bytes SRAM in the following examples.

To construct the stack, the stack pointer is loaded with the highest available SRAM address. (In our case the tower grows downwards, towards lower addresses, just for historic reasons and to confuse the beginner!).

```
.DEF MyPreferredRegister = R16
    LDI MyPreferredRegister, HIGH(RAMEND) ; Upper byte
    OUT SPH,MyPreferredRegister ; to stack pointer
    LDI MyPreferredRegister, LOW(RAMEND) ; Lower byte
    OUT SPL,MyPreferredRegister ; to stack pointer
```

The value RAMEND is, of course, specific for the processor type. It is defined in the INCLUDE file for the processor type. The file 8515def.inc has the line:

```
.equ RAMEND = $25F ; Last On-Chip SRAM Location
```

The file 8515def.inc is included with the assembler directive

```
.INCLUDE "C:\somewhere\8515def.inc"
```

at the beginning of our assembler source code.

So we defined the stack now, and we don't have to care about the stack pointer any more, because manipulations of that pointer are mostly automatic.

Use of the stack

Using the stack is easy. The content of registers are pushed onto the stack like this:

```
PUSH MyPreferredRegister ; Throw that value on top of the stack
```

Where that value goes to is totally uninteresting. That the stack pointer was decremented after that push, we don't have to care. If we need the content again, we just add the following instruction:

```
POP MyPreferredRegister ; Read back the value from the top of the stack
```

With POP we just get the value that was last pushed on top of the stack. Pushing and popping registers makes sense, if

- the content is again needed some lines of the code later,
- all registers are in use, and if
- no other opportunity exists to store that value somewhere else.

If these conditions are not given, the use of the stack for saving registers is useless and just wastes processor time.

More sense makes the use of the stack in subroutines, where you have to return to the program location that called the routine. In that case the calling program code pushes the return address (the current program counter value) onto the stack and temporarily jumps to the subroutine. After its execution the subroutine pops the return address from the stack and loads it back into the program counter. Program execution is continued exactly one instruction behind the instruction, where the call happened:

```
RCALL Somewhat ; Jump to the label "somewhat:"
[...] here we will later continue with the program.
```

Here the jump to the label "somewhat:" somewhere in the program code,

```
Somewhat: ; this is the jump address
[...] Here we do something
[...] and we are finished and want to jump back to the calling location:
RET
```

During execution of the RCALL instruction the already incremented program counter, a 16-bit-address, is pushed onto the stack, using two pushes (the LSB and the MSB). By reaching the RET instruction, the content of the previous program counter is reloaded with two pops and execution continues there.

You don't need to care about the address of the stack, where the counter is loaded to. This address is automatically generated. Even if you call a subroutine within that subroutine the stack function is fine. This just packs two return addresses on top of the stack, the nested subroutine removes the first one, the calling subroutine the remaining one. As long as there is enough SRAM, everything is fine.

Servicing hardware interrupts isn't possible without the stack. Interrupts stop the normal execution of the program, wherever the program currently is. After execution of a specific service routine as a reaction to that interrupt program execution must return to the previous location, to before the interrupt occurred. This would not be possible if the stack is not able to store the return address.

The enormous advances of having a stack for interrupts are the reason, why even the smallest AVRs without having SRAM have at least a very small hardware stack.

Bugs with the stack operation

For the beginner there are a lot of possible bugs, if you first learn to use stack.

Very clever is the use of the stack without first setting the stack pointer. Because this pointer is set to zero at program start, the pointer points to the location 0x0000, where register R0 is located. Pushing a byte results in a write to that register, overwriting its previous content. An additional push to the stack writes to 0xFFFF, an undefined position (if you don't have external SRAM there). A RCALL and RET will return to a strange address in program memory. Be sure: there is no warning, like a window popping up saying something like „Illegal access to memory location xxxx“.

Another opportunity to construct bugs is to forget to pop a previously pushed value, or popping a value without pushing one first.

In a very few cases the stack overflows to below the first SRAM location. This happens in case of a never-ending recursive call. After reaching the lowest SRAM location the next pushes write to the ports (0x005F down to 0x0020), then to the registers (0x001F to 0x0000). Funny and unpredictable things happen with the chip hardware, if this goes on. Avoid this bug, it can even destroy your external hardware!

Jumping and Branching

Here we discuss all instructions that control the sequential execution of a program. It starts with the starting sequence on power-up of the processor, continues with jumps, interrupts, etc.

Controlling sequential execution of the program

What happens during a reset?

When the power supply voltage of an AVR rises and the processor starts its work, the hardware triggers a reset sequence. The ports are set to their initial values, as defined in the device data sheet. The counter for the program steps will be set to zero. At this address the execution always starts. Here we have to have our first word of code. But not only during power-up this address is activated:

- During an external reset on the reset pin of the device a restart is executed.
- If the Watchdog counter reaches its maximum count, a reset is initiated. A watchdog timer is an internal clock that must be reseted from time to time by the program, otherwise it restarts the processor.
- You can call reset by a direct jump to that address (see the jump section below).

The third case is not a real reset, because the automatic resetting of register- and port-values to a well-defined default value is not executed. So, forget that for now.

The second option, the watchdog reset, must first be enabled by the program. It is disabled by default. Enabling requires write instructions to the watchdog's port. Setting the watchdog counter back to zero requires the execution of the instruction

```
WDR
```

to avoid a reset.

After execution of a reset, with setting registers and ports to default values, the code at address 0000 is word wise read to the execution part of the processor and is executed. During that execution the program counter is already incremented by one and the next word of code is already read to the code fetch buffer (Fetch during Execution). If the executed instruction does not require a jump to another location in the program the next instruction is executed immediately. That is why the AVRs execute extremely fast, each clock cycle executes one instruction (if no jumps occur).

The first instruction of an executable is always located at address 0000. To tell the compiler (assembler program) that our source code starts now and here, a special directive can be placed at the beginning, before the first code in the source is written:

```
.CSEG
.ORG 0000
```

The first directive, `.CSEG`, lets the compiler switch his output to the code section. All following is translated as code and is later written to the program flash memory section of the processor. Another target segment would be the EEPROM section of the chip, where you also can write bytes or words to.

```
.ESEG
```

The third segment is the SRAM section of the chip.

```
.DSEG
```

Other than with EEPROM content, where content is really going to the EEPROM during programming of the chip, the DSEG segment content is not programmed to the chip. There is no opportunity to burn any SRAM content. So the `.DSEG` is only used for correct label calculation during the assembly process. An example:

```
.DSEG ; The following are label definitions within the SRAM segment
MyFirstVariableIsAByte:
.BYTE 1 ; the DSEG-Pointer moves one byte upwards
MySecondVariableIsAWord:
.BYTE 2 ; the DSEG-Pointer moves two bytes upwards
MyThirdVariableIsAFieldForABuffer:
.BYTE 32; the DSEG-Pointer moves 32 bytes upwards
```

So, only three labels are defined within the assembler, no content is produced.

The `ORG` directive within the code segment, `.ORG`, above stands for the word "origin" and manipulates the address within the code segment, where assembled words go to. As our program always starts at 0x0000 the `CSEG/ORG` directives are trivial, you can skip these without getting into an error. We could start at 0x0100, but that makes no real sense as the processor starts execution at 0000. If you want to place a table exactly to a certain location of the code segment, you can use `ORG`. But be careful with that: Only jump forward with `.ORG`, never backwards. And be aware that the flash memory space that you skipped in between your current code location and the one you forced with `.ORG` is always filled with the instruction word 0xFFFF. This instruction does nothing, just goes to the next instruction. So be sure your execution

never jumps into such undefined space in between.

If on the beginning of your code section you want to set a clear sign within your code, after first defining a lot of other things with `.DEF-` and `.EQU-` directives, use the `CSEG/ORG` sequence as a signal for yourself, even though it might not be necessary to do that.

As the first code word is always at address zero, this location is also called the reset vector. Following the reset vector the next positions in the program space, addresses `0x0001`, `0x0002` etc., are interrupt vectors. These are the positions where the execution jumps to if an external or internal interrupt has been enabled and occurs. These positions called vectors are specific for each processor type and depend on the internal hardware available (see below). The instructions to react to such an interrupt have to be placed to the proper vector location. If you use interrupts, the first code, at the reset vector, must be a jump instruction, to jump over the other vectors. Each interrupt vector, that is planned to be enabled, must hold a jump instruction to the respective interrupt service routine. If the vector is not used, a dummy instruction like `RETI` (RETurn from Interrupt) is best placed here. The typical program sequence at the beginning is like follows:

```
.CSEG
.ORG 0000
    RJMP Start ; the reset vector
    RJMP IntServRout1 ; the interrupt service routine for the first interrupt
    RETI ; a dummy for an unused interrupt
    RJMP IntServRout3 ; the interrupt service routine for the third interrupt
[...] here we place all the other interrupt vector instructions

[...] and here is a good place for the interrupt service routines themselves
IntServRout1:
    [...] Code of the first int service routine
    RETI ; end of service routine 1
IntServRout2:
    [...] Code of the third int service routine
    RETI ; end of service routine 2
[...] other code
Start: ; This here is the program start
[...] Here we place our main program
```

The instruction “`RJMP Start`” results in a jump to the label `Start:`, located some lines below. Remember, labels always end with a “`:`”. Labels, that don’t fulfill these conditions are not taken for serious, but interpreted as instructions. Missing labels result in an error message (“Undefined label”), and compilation is interrupted.

Linear program execution and branches

Program execution is always linear, if nothing changes the sequential execution. These changes are the execution of an interrupt or of branching instructions.

Branching

Branching is very often depending on some condition, called conditional branching. As an example we assume we want to construct a 32-bit-counter using the registers `R1` to `R4`. The least significant byte in `R1` is incremented by one. If the register overflows during that operation ($255 + 1 = 0$), we have to increment `R2` similarly. If `R2` overflows, we have to increment `R3`, and so on.

Incrementation by one is done with the instruction `INC`. If an overflow occurs during that execution of `INC R1`, the zero bit in the status register is set to one (the result of the operation is zero). The carry bit in the status register, as usually set when something overflows, is not changed during an `INC`. This is not to confuse the beginner, but carry can be used for other purposes instead. The Zero-Bit or Zero-flag in this case is enough to detect an overflow. If no overflow occurs we can just leave the counting sequence.

If the Zero-bit is set, we must execute additional incrementation of the next upper register. To confuse the beginner the branching instruction, that we have to use, is not named `BRNZ` but `BRNE` (BRanch if Not Equal). A matter of taste ...

The whole count sequence of the 32-bit-counter should then look like this:

```
    INC R1 ; increase content of register R1
    BRNE GoOn32 ; if not zero, branch to GoOn32:
    INC R2 ; increase content of register R2
    BRNE GoOn32
    INC R3
    BRNE GoOn32
    INC R4
GoOn32:
```

So that's about it. An easy thing. The opposite condition to `BRNE` is `BREQ` or BRanch EQual.

Which of the status bits, also called processor flags, are changed during execution of an instruction is listed in instruction code tables, see the List of Instructions. Similarly to the Zero-bit you can use the other status bits like that:

```
    BRCC label/BRCS label; Carry-flag 0 (BRCC) or 1 (BRCS)
```

BRSR label; Equal or greater
BRLO label; Smaller
BRMI label; Minus
BRPL label; Plus
BRGE label; Greater or equal (with sign bit)
BRLT label; Smaller (with sign bit)
BRHC label/BRHS label; Half overflow flag 0 or 1
BRTC label/BRTS label; T-Bit 0 or 1
BRVC label/BRVS label; Two's complement flag 0 or 1
BRIE label/BRID label; Interrupt enabled or disabled

to react to the different conditions. Branching always occurs if the condition is met. Don't be afraid, most of these instructions are rarely used. For the beginner only Zero and Carry are relevant.

Timing during program execution

Like mentioned above the required time to execute one instruction is equal to the processor's clock cycle. If the processor runs on a 4 MHz clock frequency then one instruction requires 1/4 μ s or 250 ns, at 10 MHz clock only 100 ns. The required time is as exact as the internal or external or xtal clock is. If you need exact timing an AVR is the optimal solution for your problem. Note that there are a few instructions that require two or more cycles, e. g. the branching instructions (if branching occurs) or the SRAM read/write sequence. See the instruction table for details.

To define exact timing there must be an opportunity that does nothing else than delay program execution. You might use other instructions that do nothing, but more clever is the use of the no-operation instruction NOP. This is the most useless instruction:

NOP

This instruction does nothing but wasting processor time. At 4 MHz clock we need just four of these instructions to waste 1 μ s. No other hidden meanings here on the NOP instruction. For a signal generator with 1 kHz we don't need to add 4000 such instructions to our source code, but we use a software counter and some branching instructions. With these we construct a loop that executes for a certain number of times and are exactly delayed. A counter could be a 8-bit-register that is decremented with the DEC instruction, e. g. like this:

CLR R1 ; one clock cycle

Count:

DEC R1 ; one clock cycle

BRNE Count ; two for branching, one for not branching

This sequence wastes $(1) + (255*2) + (1*3) = 514$ clock cycles or 128.5 μ s at 4 MHz.

16-bit counting can also be used to delay exactly, like this

LDI ZH,HIGH(65535) ; one clock cycle

LDI ZL,LOW(65535) ; one clock cycle

Count:

SBIW ZL,1 ; two clock cycles

BRNE Count ; two for branching, one for not branching

This sequence wastes $(1+1) + (65534*4) + (1*3) = 262,141$ clock cycles or 65,535.25 μ s at 4 MHz.

If you use more registers to construct nested counters you can reach any delay. And the delay is as exact as your clock source is, even without a hardware timer.

Macros and program execution

Very often you have to write identical or similar code sequences on different occasions in your source code. If you don't want to write it once and jump to it via a subroutine call you can use a macro to avoid getting tired writing the same sequence several times. Macros are code sequences, designed and tested once, and inserted into the code by its macro name. As an example we assume we need to delay program execution several times by 1 μ s at 4 MHz clock. Then we define a macro somewhere in the source:

```
.MACRO Delay1
    NOP
    NOP
    NOP
    NOP
.ENDMACRO
```

This definition of the macro does not yet produce any code, it is silent. Code is produced only if you call that macro by its name:

[...] somewhere in the source code

Delay1

[...] code goes on here

This results in four NOP instructions inserted to the code at that location. An additional "Delay1" inserts additional four NOP instructions.

If your macro has longer code sequences, or if you are short in code storage space, you should avoid the

use of macros and use subroutines instead.

By calling a macro by its name you can add some parameters to manipulate the produced code. But this is more than a beginner has to know about macros.

Subroutines

In contrary to macros a subroutine does save program storage space. The respective sequence is only once stored in the code and is called from whatever part of the code. To ensure continued execution of the sequence following the subroutine call you need to return to the caller. For a delay of 10 cycles you need to write this subroutine:

```
Delay10: ; the call of the subroutine requires some cycles
        NOP ; delay one cycle
        NOP ; delay one cycle
        NOP ; delay one cycle
        RET ; return to the caller
```

Subroutines always start with a label, otherwise you would not be able to jump to it, here named "Delay10:". Three NOPs follow and a RET instruction. If you count the necessary cycles you just find 7 cycles (3 for the NOPs, 4 for the RET). The missing 3 are for calling that routine:

```
[...] somewhere in the source code:
        RCALL Delay10
```

```
[...] further on with the source code
```

RCALL is a relative call. The call is coded as relative jump, the relative distance from the calling routine to the subroutine is calculated by the compiler. The RET instruction jumps back to the calling routine. Note that before you use subroutine calls you must set the stack pointer (see Stack), because the return address must be packed on top of the stack during the RCALL instruction.

If you want to jump directly to somewhere else in the code you have to use the jump instruction:

```
[...] somewhere in the source code
        RJMP Delay10
```

Return:

```
[...] further on with source code
```

Note that RJMP is also a relative jump instruction with limited distance. Only ATmega AVR's have a JMP instruction allowing jumps over the complete flash memory space, but these instructions require two words and more instruction time than RJMP, so avoid it if possible.

The routine that you jumped to can not use the RET instruction in that case, because RJMP does not place the current execution address to the stack. To return back to the calling location in the source requires to add another label and the called routine to jump back to this label. Jumping like this is not like calling a subroutine because you can't call this routine from different locations in the code.

RCALL and RJMP are unconditioned branches. To jump to another location, depending on some condition, you have to combine these with branching instructions. Conditioned calling of a subroutine can best be done with the following (confusing) instructions. If you want to call a subroutine depending on a certain bit in a register use the following sequence:

```
        SBRC R1,7 ; Skip the next instruction if bit 7 in register 1 is 0
        RCALL UpLabel ; Call that subroutine
```

SBRC reads „Skip next instruction if Bit 7 in Register R1 is Clear (=Zero)“. The RCALL instruction to "UpLabel:" is only executed if bit 7 in register R1 is 1, because the next instruction is skipped if it would be 0. If you like to call the subroutine in case this bit is 0 then you use the corresponding instruction SBRS. The instruction following SBRS/SBRC can be a single word or double word instruction, the processor knows how far he has to jump over it. Note that execution times are different then. To jump over more than one following instruction these instructions cannot be used.

If you have to skip an instruction in case two registers have the same value you can use the following exotic instruction:

```
        CPSE R1,R2 ; Compare R1 and R2, skip next instruction if equal
        RCALL SomeSubroutine ; Call SomeSubroutine
```

A rarely used instruction, forget it for the beginning. If you like to skip the following instruction depending on a certain bit in a port use the following instructions SBIC and SBIS. That reads "Skip if the Bit in I/O space is Clear (or Set)", like this:

```
        SBIC PINB,0 ; Skip next instruction if Bit 0 on input port B is 0
        RJMP ATarget ; Jump to the label ATarget
```

The RJMP-instruction is only executed if bit 0 in port B is high. This is something confusing for the beginner. The access to the port bits is limited to the lower half of ports, the upper 32 ports are not usable here.

Now, another exotic application for the expert. Skip this if you are a beginner. Assume we have a bit switch with 4 switches connected to port B. Depending on the state of these 4 bits we would like to jump to 16 different locations in the code. Now we can read the port and use several branching instructions to find out, where we have to jump to today. As alternative you can write a table holding the 16 addresses, like this:

MyTab:

```
RJMP Routine1
RJMP Routine2
[...]
RJMP Routine16
```

In our code we copy that address of the table to the Z pointer register:

```
LDI ZH,HIGH(MyTab)
LDI ZL,LOW(MyTab)
```

and add the current state of the port B (in R16) to this address.

```
ADD ZL,R16
BRCC NoOverflow
INC ZH
```

NoOverflow:

Now we can jump to this location in the table, either for calling a subroutine:

```
ICALL ; call the subroutine which address is in Z
```

or as a jump with no way back:

```
IJMP ; jump to address in Z
```

The processor loads the content of the Z register pair into its program counter and continues operation there. More clever than branching over and over?

Interrupts and program execution

Very often we have to react on hardware conditions or other events. An example is a change on an input pin. You can program such a reaction by writing a loop, asking whether a change on the pin has occurred. This method is called polling, its like a bee running around in circles searching for new flowers. If there are no other things to do and reaction time does not matter, you can do this with the processor. If you have to detect short pulses of less than a μs duration this method is useless. In that case you need to program an interrupt.

An interrupt is triggered by some hardware conditions. All hardware interrupts are disabled at reset time by default, so the condition has to be enabled first. The respective port bits enabling the component's interrupt ability are set first. The processor has a bit in its status register enabling him to respond to the interrupt of all components, the Interrupt Enable Flag. Enabling the general response to interrupts requires the following instruction:

```
SEI ; Set Int Enable Bit
```

Each single interrupt requires additional port manipulation to be enabled.

If the interrupting condition occurs, e. g. a change on the port bit, the processor pushes the actual program counter to the stack (which must be enabled first! See initiation of the stackpointer in the Stack section of the SRAM description). Without that, the processor wouldn't be able to return back to the location, where the interrupt occurred (which could be any time and anywhere within program execution). After that, processing jumps to the predefined location, the interrupt vector, and executes the instructions there. Usually the instruction there is a JUMP instruction to the interrupt service routine, located somewhere in the code. The interrupt vector is a processor-specific location and depending from the hardware component and the condition that leads to the interrupt. The more hardware components and the more conditions, the more vectors. The different vectors for some older AVR types are listed in the following table. (The first vector isn't an interrupt but the reset vector, performing no stack operation!)

Name	Interrupt Vector Address			Triggered by
	2313	2323	8515	
RESET	0000	0000	0000	Hardware Reset, Power-On-Reset, Watchdog Reset
INT0	0001	0001	0001	Level change on the external INT0 pin
INT1	0002	-	0002	Level change on the external INT1 pin
TIMER1CAPT	0003	-	0003	Capture event on Timer/Counter 1
TIMER1COMPA	-	-	0004	Timer/Counter 1 = Compare value A
TIMER1 COMPB	-	-	0005	Timer/Counter 1 = Compare value B
TIMER1 COMP1	0004	-	-	Timer/Counter 1 = Compare value 1
TIMER1 OVF	0005	-	0006	Timer/Counter 1 Overflow
TIMER0 OVF	0006	0002	0007	Timer/Counter 0 Overflow
SPI STC	-	-	0008	Serial Transmit Complete
UART TX	0007	-	0009	UART char in receive buffer available

Name	Interrupt Vector Address			Triggered by
UART UDRE	0008	-	000A	UART transmitter ran empty
UART TX	0009	-	000B	UART All Sent
ANA_COMP	-	-	000C	Analog Comparator

Note that the capability to react to events is very different for the different types. The addresses are sequential, but not identical for different types. Consult the data sheet for each AVR type.

The higher a vector in the list the higher is its priority. If two or more components have an interrupt condition pending at the same time, the up most vector with the lower vector address wins. The lower int has to wait until the upper int was served. To disable lower ints from interrupting during the execution of its service routine the first executed int disables the processor's I-flag. The service routine must re-enable this flag after it is done with its job.

For re-setting the I status bit there are two ways. The service routine can end with the instruction:

```
RETI
```

This return from the int routine restores the I-bit after the return address has been loaded to the program counter.

The second way is to enable the I-bit by the instruction

```
SEI ; Set Interrupt Enabled
RET ; Return
```

This is not the same as the RETI, because subsequent interrupts are already enabled before the program counter is re-loaded with the return address. If another int is pending, its execution is already starting before the return address is popped from the stack. Two or more nested addresses remain on the stack. No bug is to be expected, but it is an unnecessary risk doing that. So just use the RETI instruction to avoid this unnecessary flow to the stack.

An Int-vector can only hold a relative jump instruction to the service routine. If a certain interrupt is not used or undefined we can just put a RETI instruction there, in case an erroneously enabled int happens before we wrote an interrupt service routine. In a few cases it is absolutely necessary to react to these false ints. That is the case if the execution of the respective service routine does not automatically reset the interrupt condition flag of the peripheral. In that case a simple RETI would reset the otherwise never-ending interrupts. This is the case with some of the UART interrupts.

As, after an interrupt is under service, further execution of lower-priority interrupts are blocked, all int service routines should be as short as possible. If you need to have a longer routine to serve the int, use one of the two following methods. The first is to allow ints by SEI within the service routine, whenever you're done with the most urgent tasks. This is not very clever. More convenient is to perform the urgent tasks, setting a flag somewhere in a register for the slower reaction portions and return from the int immediately.

A very serious rule for int service routines is:

The first instruction is always to save the processor status flags in a register or on the stack.

Do this before you use instructions that might change flags in the status flag register. The reason is that the interrupted main program might just be in a state using the flag for a branch decision, and the int would just change that flag to another state. Funny things would happen from time to time. The last instruction before the RETI therefore is to copy the saved flags from the register back to status port or to pop the status register content from the stack and restore its original content. The following shows examples how to do that:

Saving in a register:

```
Isr:
    IN R15,SREG ; save flags
    [... more instructions...]
    OUT SREG,R15 ; restore flags
    RETI ; return from interrupt
```

Saving on the stack:

```
Isr:
    PUSH R15 ; save register on stack
    IN R15, SREG
    [...more instructions...]
    OUT SREG,R15 ; restore flags
    POP R15
    RETI ; return from interrupt
```

The method on the right is slower, the method on the left requires a register exclusively for that purpose.

Generally: All used registers in a service routine should either be exclusively reserved for that purpose or saved on stack and restored at the end of the service routine. Never change the content of a register within an int service routine that is used somewhere else in the normal program without restoring it.

Because of these basic requirements a more sophisticated example for an interrupt service routine here.

```
.CSEG ; Code-Segment starts here
.ORG 0000 ; Address is zero
    RJMP Start ; The reset-vector on Address 0000
    RJMP IService ; 0001: first Int-Vector, INTO service routine
```

[...] here other vectors

Start: ; Here the main program starts

[...] here is enough space for defining the stack and other things

IService: ; Here we start with the Interrupt-Service-Routine

PUSH R16 ; save a register to stack

IN R16,SREG ; read status register

PUSH R16 ; and put on stack

[...] Here the Int-Service-Routine does something and uses R16

POP R16 ; get previous flag register from stack

OUT SREG,R16 ; restore old status

POP R16 ; get previous content of R16 from the stack

RETI ; and return from int

Looks a little bit complicated, but is a prerequisite for using ints without producing serious bugs. Skip PUSH R16 and POP R16 if you can afford reserving the register for exclusive use within the service routine. As an interrupt service routine cannot be interrupted (unless you allow interrupts within the routine), all different int service routines can use the same register.

You understand now, why allowing interrupts within an interrupt service routine, and not at its end with RETI, is not a good idea?

That's it for the beginner. There are some other things with ints, but this is enough to start with, and not to confuse you.

Calculations

Here we discuss all necessary instructions for calculating in AVR assembler language. This includes number systems, setting and clearing bits, shift and rotate, and adding/subtracting/comparing and the format conversion of numbers.

Number systems in assembler

The following formats of numbers are common in assembler:

- Positive whole numbers (Bytes, Words, Longwords, etc.),
- Signed whole numbers (ShortInts, Integers, LongInts, etc.),
- Binary Coded Digits (BCD),
- Packed BCDs,
- ASCII-formatted numbers.

If you come from a high-level language: forget pre-defined number formats. Assembler doesn't have that concept nor its (sometimes frustrating) limitations. What you earn is: you are the master of your own format!

Positive whole numbers (bytes, words, etc.)

The smallest whole number to be handled in assembler is a byte with eight bits. This codes numbers between 0 and 255. Such bytes fit exactly into one register of the MCU. All larger numbers must be based on this basic format, using more than one register. Two bytes yield a word (range from 0 .. 65,535), three bytes form a longer word (range from 0 .. 16,777,215) and four bytes form a double word (range from 0 .. 4,294,967,295).

The single bytes of a word or a double word can be stored in whatever register you prefer. Operations with these single bytes are programmed byte by byte, so you don't have to put them in a row. In order to form a row for a double word we could store it like this:

```
.DEF r16 = dw0
.DEF r17 = dw1
.DEF r18 = dw2
.DEF r19 = dw3
```

Registers dw0 to dw3 are in a row, but don't need to be. If we need to initiate this double word at the beginning of an application (e. g. to 4,000,000), this should look like this:

```
.EQU dwi = 4000000 ; define the constant
    LDI dw0,LOW(dwi) ; The lowest 8 bits to R16
    LDI dw1,BYTE2(dwi) ; bits 8 .. 15 to R17
    LDI dw2,BYTE3(dwi) ; bits 16 .. 23 to R18
    LDI dw3,BYTE4(dwi) ; bits 24 .. 31 to R19
```

So we have splitted this decimal number, called dwi, to its binary portions BYTE4 to BYTE1 and packed them into the four byte packages. Now you can calculate with this double word.

Signed numbers (integers)

Sometimes, but in rare cases, you need negative numbers to calculate with. A negative number is defined by interpreting the most significant bit of a byte as sign bit. If it is 0 the number is positive. If it is 1 the number is negative. If the number is negative we usually do not store the rest of the number as is, but we use its inverted value. Inverted means that -1 as a byte integer is not written as 1000.0001 but as 1111.1111 instead. That means: subtract 1 from 0 (and forget the overflow). The first bit is the sign bit, signaling that this is a negative number. Why this different format (subtracting the number from 0) is used is easy to understand: adding -1 (1111.1111) and +1 (0000.0001) yields exactly zero, if you forget the overflow that occurs during that operation (to the ninth bit).

In one byte the largest integer number to be handled is +127 (binary 01111111), the smallest one is -128 (binary 1,0000000). In other computer languages this number format is called short integer. If you need a bigger range of values you can add another byte to form a larger integer value, ranging from +32,767 .. -32,768, four bytes provide a range from +2,147,483,647 .. -2,147,483,648, in other languages called a LongInt or DoubleInt.

Binary Coded Digits, BCD

Positive or signed whole numbers in the formats discussed above use the available space most effectively. Another, less dense number format, but easier to handle and understand is to store decimal numbers in a byte for one digit each. The decimal digit is stored in its binary form in a byte. Each digit from 0 .. 9 needs four bits (binary values 0000 .. 1001), the upper four bits of the byte are always zeros, blowing a lot of hot air into one byte. For to handle the value 250 we would need at least three bytes, e. g.:

Bit value	128	64	32	16	8	4	2	1
R16, Digit 1 =2	0	0	0	0	0	0	1	0
R17, Digit 2 = 5	0	0	0	0	0	1	0	1
R18, Digit 3 = 0	0	0	0	0	0	0	0	0

```

;Instructions to use:
LDI R16,2
LDI R17,5
LDI R18,0

```

You can calculate with these numbers, but this is a bit more complicated in assembler than calculating with binary values. The advantage of this format is that you can handle as long numbers as you like, as long as you have enough storage space. The calculations are as precise as you like (if you program AVRs for banking applications), and you can convert them very easily to character strings.

Packed BCDs

If you pack two decimal digits into one byte you don't lose that much storage space. This method is called packed binary coded digits. The two parts of a byte are called upper and lower nibble. The upper nibble usually holds the more significant digit, which has advantages in calculations (special instructions in AVR assembler language). The decimal number 250 would look like this when formatted as a packed BCD:

Byte	Digits	Value	8	4	2	1	8	4	2	1
2	4 & 3	02	0	0	0	0	0	0	1	0
1	2 & 1	50	0	1	0	1	0	0	0	0

```

; Instructions for setting:
LDI R17,0x02 ; Upper byte
LDI R16,0x50 ; Lower byte

```

To set this correct you can use the binary notation (0b...) or the hexadecimal notation (0x...) to set the proper bits to their correct nibble position.

Calculating with packed BCDs is a little more complicated compared to the binary form. Format changes to character strings are nearly as easy as with BCDs. Length of numbers and precision of calculations is only limited by the storage space.

Numbers in ASCII-format

Very similar to the unpacked BCD format is to store numbers in ASCII format. The digits 0 to 9 are stored using their ASCII (ASCII = American Standard Code for Information Interchange) representation. ASCII is a very old format, developed and optimized for teletype writers, unnecessarily very complicated for computer use (do you know what a char named End Of Transmission EOT meant when it was invented?), very limited in range for other than US languages (only 7 bits per character), still used in communications today due to the limited efforts of some operating system programmers to switch to more effective character systems. This ancient system is only topped by the European 5-bit long teletype character set called Baudot set or the Morse code, still used by some finger-nervous people.

Within the ASCII code system the decimal digit 0 is represented by the number 48 (hex 0x30, binary 0b0011.0000), digit 9 is 57 decimal (hex 0x39, binary 0b0011.1001). ASCII wasn't designed to have these numbers on the beginning of the code set as there are already instruction chars like the above mentioned EOT for the teletype. So we still have to add 48 to a BCD (or set bit 4 and 5 to 1) to convert a BCD to ASCII. ASCII formatted numbers need the same storage space like BCDs. Loading 250 to a register set representing that number would look like this:

```

LDI R18,'2'
LDI R17,'5'
LDI R16,'0'

```

The ASCII representation of these characters are written to the registers.

Bit manipulations

To convert a BCD coded digit to its ASCII representation we need to set bit 4 and 5 to a one. In other words we need to OR the BCD with a constant value of hex 0x30. In assembler this is done like this:

```

ORI R16,0x30

```

If we have a register that is already set to hex 0x30 we can use the OR with this register to convert the BCD:

```

OR R1,R2

```

Back from an ASCII character to a BCD is as easy. The instruction

```
ANDI R16,0x0F
```

isolates the lower four bits (= the lower nibble). Note that ORI and ANDI are only possible with registers above R15. If you need to do this, use one of the registers R16 to R31!

If the hex value 0x0F is already in register R2, you can AND the ASCII character with this register:

```
AND R1,R2
```

The other instructions for manipulating bits in a register are also limited for registers above R15. They would be formulated like this:

```
SBR R16,0b00110000 ; Set bits 4 and 5 to one
CBR R16,0b00110000 ; Clear bits 4 and 5 to zero
```

If one or more bits of a byte have to be inverted you can use the following instruction (which is not possible for use with a constant):

```
LDI R16,0b10101010 ; Invert all uneven bits
EOR R1,R16 ; in register R1 and store result in R1
```

To invert all bits of a byte is called the One's complement:

```
COM R1
```

inverts the content in register R1 and replaces zeros by one and vice versa. Different from that is the Two's complement, which converts a positive signed number to its negative complement (subtracting from zero). This is done with the instruction

```
NEG R1
```

So +1 (decimal: 1) yields -1 (binary 1.1111111), +2 yields -2 (binary 1.1111110), and so on.

Besides the manipulation of the bits in a register, copying a single bit is possible using the so-called T-bit of the status register. With

```
BST R1,0
```

the T-bit is loaded with a copy of bit 0 in register R1. The T-bit can be set or cleared, and its content can be copied to any bit in any register:

```
CLT ; clear T-bit, or
SET ; set T-bit, or
BLD R2,2 ; copy T-bit to register R2, bit 2
```

Shift and rotate

Shifting and rotating of binary numbers means multiplying and dividing them by 2. Shifting has several sub-instructions.

Multiplication with 2 is easily done by shifting all bits of a byte one binary digit left and writing a zero to the least significant bit. This is called logical shift left or LSL. The former bit 7 of the byte will be shifted out to the carry bit in the status register.

```
LSL R1
```

The inverse division by 2 is the instruction called logical shift right, LSR.

```
LSR R1
```

The former bit 7, now shifted to bit 6, is filled with a 0, while the former bit 0 is shifted into the carry bit of the status register. This carry bit could be used to round up and down (if set, add one to the result). Example, division by four with rounding:

```
LSR R1 ; division by 2
BRCC Div2 ; Jump if no round up
INC R1 ; round up
```

Div2:

```
LSR R1 ; Once again division by 2
BRCC DivE ; Jump if no round up
INC R1 ; Round Up
```

DivE:

So, dividing is easy with binaries as long as you divide by multiples of 2.

If signed integers are used the logical shift right would overwrite the sign-bit in bit 7. The instruction „arithmetic shift right“ ASR leaves bit 7 untouched and shifts the 7 lower bits, inserting a zero into bit location 6.

```
ASR R1
```

Like with logical shifting the former bit 0 goes to the carry bit in the status register.

What about multiplying a 16-bit word by 2? The most significant bit of the lower byte has to be shifted to yield the lowest bit of the upper byte. In that step a shift would set the lowest bit to zero, but we need to shift the carry bit from the previous shift of the lower byte into bit 0 of the upper byte. This is called a rotate. During rotation the carry bit in the status register is shifted to bit 0, the former bit 7 is shifted to the carry during rotation.

```
LSL R1 ; Logical Shift Left of the lower byte
ROL R2 ; ROTate Left of the upper byte
```

The logical shift left in the first instruction shifts bit 7 to carry, the ROL instruction rolls it to bit 0 of the upper byte. Following the second instruction the carry bit has the former bit 7 of the upper byte. The carry bit can be used to either indicate an overflow (if 16-bit-calculation is performed) or to roll it into more upper bytes (if more than 16 bit calculation is done).

Rolling to the right is also possible, dividing by 2 and shifting carry to bit 7 of the result:

```
LSR R2 ; Logical Shift Right, bit 0 to carry
ROR R1 ; ROTate Right and shift carry in bit 7
```

It's easy dividing with big numbers. You see that learning assembler is not THAT complicated.

The last instruction that shifts four bits in one step is very often used with packed BCDs. This instruction shifts a whole nibble from the upper to the lower position and vice versa. In our example we need to shift the upper nibble to the lower nibble position. Instead of using

```
ROR R1
ROR R1
ROR R1
ROR R1
```

we can perform that with a single

```
SWAP R1
```

This instruction exchanges the upper and lower nibble. Note that the content of the upper nibble will be different after applying these two methods.

Adding, subtracting and comparing

The following calculation operations are too complicated for the beginners and demonstrate that assembler is only for extreme experts, hi. Read on your own risk!

Adding and subtracting 16-bit numbers

To start complicated we add two 16-bit-numbers in R1:R2 and R3:R4. (In this notation, we mean that the first register is the most significant byte, the second the least significant).

```
ADD R2,R4 ; first add the two low-bytes
ADC R1,R3 ; then the two high-bytes
```

Instead of a second ADD we use ADC in the second instruction. That means add with carry, which is set or cleared during the first instruction, depending from the result. Already scared enough by that complicated math? If not: take this!

We subtract R3:R4 from R1:R2.

```
SUB R2,R4 ; first the low-byte
SBC R1,R3 ; then the high-byte
```

Again the same trick: during the second instruction we subtract another 1 from the result if the result of the first instruction had an overflow. Still breathing? If yes, handle the following!

Comparing 16-bit numbers

Now we compare a 16-bit-word in R1:R2 with the one in R3:R4 to evaluate whether it is bigger than the second one. Instead of SUB we use the compare instruction CP, instead of SBC we use CPC:

```
CP R2,R4 ; compare lower bytes
CPC R1,R3 ; compare upper bytes
```

If the carry flag is set now, R1:R2 is larger than R3:R4.

Comparing with constants

Now we add some more complicated stuff. We compare the content of R16 with a constant: 0b10101010.

```
CPI R16,0xAA
```

If the Zero-bit in the status register is set after that, we know that R16 is equal to 0xAA. If the carry-bit is set, we know, it is smaller. If Carry is not set and the Zero-bit is not set either, we know it is larger.

And now the most complicated test. We evaluate whether R1 is zero or negative:

TST R1

If the Z-bit is set, the register R1 is zero and we can follow with the instructions BREQ, BRNE, BRMI, BRPL, BRLO, BRSH, BRGE, BRLT, BRVC or BRVS to branch around a little bit.

Packed BCD math

Still with us? If yes, here is some packed BCD calculations. Adding two packed BCDs can result in two different overflows. The usual carry shows an overflow, if the higher of the two nibbles overflows to more than 15 decimal. Another overflow, from the lower to the upper nibble occurs, if the two lower nibbles add to more than 15 decimal.

To take an example we add the packed BCDs 49 (=hex 49) and 99 (=hex 99) to yield 148 (=hex 0x0148). Adding these in binary math, results in a byte holding hex 0xE2, no byte overflow occurs. The lower of the two nibbles should have an overflow, because $9+9=18$ (more than 9) and the lower nibble can only handle numbers up to 15. The overflow was added to bit 4, the lowest significant bit of the upper nibble. Which is correct! But the lower nibble should be 8 and is only 2 ($18 = 0b0001.0010$). We should add 6 to that nibble to yield a correct result. Which is quite logic, because whenever the lower nibble reaches more than 9 we have to add 6 to correct that nibble.

The upper nibble is totally incorrect, because it is 0xE and should be 3 (with a 1 overflowing to the next upper digit of the packed BCD). If we add 6 to this 0xE we get to 0x4 and the carry is set (=0x14). So the trick is to first add these two numbers and then add 0x66 to correct the 2 digits of the packed BCD. But halt: what if adding the first and the second number would not result in an overflow to the next nibble? And not result in a digit above 9 in the lower nibble? Adding 0x66 would then result in a totally incorrect result. The lower 6 should only be added if the lower nibble either overflows to the upper nibble or results in a digit larger than 9. The same with the upper nibble.

How do we know, if an overflow from the lower to the upper nibble has occurred? The MCU sets the H-bit in the status register, the half-carry bit. The following shows the algorithm for the different cases that are possible after adding two nibbles and adding hex 0x6 after that.

1. Add the nibbles. If overflow occurs (C for the upper nibbles, or H for the lower nibbles), add 6 to correct, if not, do step 2.
2. Add 6 to the nibble. If overflow occurs (C resp. H), you're done. If not, subtract 6.

To program an example we assume that the two packed BCDs are in R2 and R3, R1 will hold the overflow, and R16 and R17 are available for calculations. R16 is the adding register for adding 0x66 (the register R2 cannot add a constant value), R17 is used to correct the result depending from the different flags. Adding R2 and R3 goes like that:

```
LDI R16,0x66 ; for adding 0x66 to the result
LDI R17,0x66 ; for later subtracting from the result
ADD R2,R3 ; add the two two-digit-BCDs
BRCC NoCy1 ; jump if no byte overflow occurs
INC R1 ; increment the next higher byte
ANDI R17,0x0F ; don't subtract 6 from the higher nibble
```

```
NoCy1:
BRHC NoHc1 ; jump if no half-carry occurred
ANDI R17,0xF0 ; don't subtract 6 from lower nibble
```

```
NoHc1:
ADD R2,R16 ; add 0x66 to result
BRCC NoCy2 ; jump if no carry occurred
INC R1 ; increment the next higher byte
ANDI R17,0x0F ; don't subtract 6 from higher nibble
```

```
NoCy2:
BRHC NoHc2 ; jump if no half-carry occurred
ANDI R17,0xF0 ; don't subtract 6 from lower nibble
```

```
NoHc2:
SUB R2,R17 ; subtract correction
```

A little bit shorter than that:

```
LDI R16,0x66
ADD R2,R16
ADD R2,R3
BRCC NoCy
INC R1
ANDI R16,0x0F
```

```
NoCy:
BRHC NoHc
ANDI R16,0xF0
```

```
NoHc:
SUB R2,R16
```

Question to think about: Why is that equally correct, half as long and less complicated and where is the trick?

Format conversion for numbers

All number formats can be converted to any other format. The conversion from BCD to ASCII and vice versa was already shown above (Bit manipulations).

Conversion of packed BCDs to BCDs, ASCII or Binaries

Conversion of packed BCDs is not very complicated either. First we have to copy the number to another register. With the copied value we change nibbles using the SWAP instruction to exchange the upper and the lower one. The upper part is cleared, e. g. by ANDing with 0x0F. Now we have the BCD of the upper nibble and we can either use as is (BCD) or set bit 4 and 5 to convert it to an ASCII character. After that we copy the byte again and treat the lower nibble without first SWAPPING and get the lower BCD.

A little bit more complicated is the conversion of BCD digits to a binary. Depending on the numbers to be handled we first clear the necessary bytes that will hold the result of the conversion. We then start with the highest BCD digit. Before adding this to the result we multiply the result with 10. (Note that in the first step this is not necessary, because the result is zero either).

In order to do the multiplication by 10, we copy the result to somewhere else. Then we multiply the result by four (two left shifts resp. rolls). Adding the previously copied number to this yields a multiplication with 5. Now a multiplication with 2 (left shift/roll) yields the 10-fold of the result. Finally we add the BCD and repeat that algorithm until all decimal digits are converted. If, during one of these operations, there occurs a carry of the result, the BCD is too large to be converted. This algorithm handles numbers of any length, as long as the result registers are prepared.

Conversion of Binaries to BCD

The conversion of a binary to BCDs is more complicated than that. If we convert a 16-bit-binary we can subtract 10,000 (0x2710), until an overflow occurs, yielding the first digit. Then we repeat that with 1,000 (0x03E8) to yield the second digit. And so on with 100 (0x0064) and 10 (0x000A), then the remainder is the last digit. The constants 10,000, 1,000, 100 and 10 can be placed to the program memory storage in a word wise organized table, like this:

DezTab:

```
.DW 10000, 1000, 100, 10
```

and can be read word-wise with the LPM instruction from the table.

An alternative is a table that holds the decimal value of each bit in the 16-bit-binary, e. g.

```
.DB 0,3,2,7,6,8
```

```
.DB 0,1,6,3,8,4
```

```
.DB 0,0,8,1,9,2
```

```
.DB 0,0,4,0,9,6
```

```
.DB 0,0,2,0,4,8 ; and so on until
```

```
.DB 0,0,0,0,0,1
```

Then you shift the single bits of the binary left out of the registers to the carry. If it is a one, you add the number in the table to the result by reading the numbers from the table using LPM. This is more complicated to program and a little bit slower than the above method.

A third method is to calculate the table value, starting with 000001, by adding this BCD with itself, each time after you have shifted a bit from the binary to the right, and added to the BCD result.

Many methods, much to optimize here.

Multiplication

Multiplication of binary numbers is explained here.

Decimal multiplication

In order to multiply two 8-bit-binaries we remind ourselves, how this is done with decimal numbers:

```
1234 * 567 = ?
```

```
-----
      1234 *   7 =   8638
+   1234 *  60 =  74040
+   1234 * 500 = 617000
-----
      1234 * 567 = 699678
=====
```

In single steps decimal:

- We multiply the first number with the lowest significant digit of the second number and add this to the result.
- We multiply the first number with 10 and then with the next higher digit of the second number and add this to the result.
- We multiply the first number with 100, then with the third-highest digit, and add this to the result.

Binary multiplication

Now in binary. Multiplication with the single digits is not necessary, because there are only the digits 1 (add the number) and 0 (don't add the number). Multiplication by 10 in decimal goes to multiplication by 2 in binary mode. Multiplication by 2 is done easily, either by adding the number with itself, or by shifting all bits one position left and writing a 0 to the void position on the right. You see that binary math is very much easier than decimal. Why didn't mankind use this from the beginning?

AVR-Assembler program

The following source code demonstrates realization of multiplication in assembler.

```

; Mult8.asm multiplies two 8-bit-numbers to yield a 16-bit-result
;
;
.NOLIST
.INCLUDE "C:\avrtools\lappnotes\8515def.inc"
.LIST
;
;
; Flow of multiplication
;
; 1.The binary to be multiplied with is shifted bitwise into the carry bit. If it is a one, the binary number is added to the
; result, if it is not a one that was shifted out, the number is not added.
; 2.The binary number is multiplied by 2 by rotating it one position left, shifting a 0 into the void position.
; 3.If the binary to be multiplied with is not zero, the multiplication loop is repeated. If it is zero, the multiplication is done.
;
;
; Used registers
;
.DEF rm1 = R0 ; Binary number to be multiplied (8 Bit)
.DEF rmh = R1 ; Interim storage
.DEF rm2 = R2 ; Binary number to be multiplied with (8 Bit)
.DEF rel = R3 ; Result, LSB (16 Bit)
.DEF reh = R4 ; Result, MSB
.DEF rmp = R16 ; Multi purpose register for loading
;
.CSEG
.ORG 0000
;
    rjmp START
;
START:
    ldi rmp,0xAA ; example binary 1010.1010
    mov rm1,rmp ; to the first binary register
    ldi rmp,0x55 ; example binary 0101.0101
    mov rm2,rmp ; to the second binary register
;
; Here we start with the multiplication of the two binaries in rm1 and rm2, the result will go to reh:rel (16 Bit)
;
MULT8:
;
; Clear start values
    clr rmh ; clear interim storage
    clr rel ; clear result registers
    clr reh
;
; Here we start with the multiplication loop
;
MULT8a:
;
; Step 1: Rotate lowest bit of binary number 2 to the carry flag (divide by 2, rotate a zero into bit 7)
;
    clc ; clear carry bit
    ror rm2 ; bit 0 to carry, bit 1 to 7 one position to the right, carry bit to bit 7
;
; Step 2: Branch depending if a 0 or 1 has been rotated to the carry bit
;
    brcc MULT8b ; jump over adding, if carry has a 0
;
; Step 3: Add 16 bits in rmh:rm1 to the result, with overflow from LSB to MSB
;
    add rel,rm1 ; add LSB of rm1 to the result
    adc reh,rmh ; add carry and MSB of rm1
;
MULT8b:
;
; Step 4: Multiply rmh:rm1 by 2 (16 bits, shift left)
;

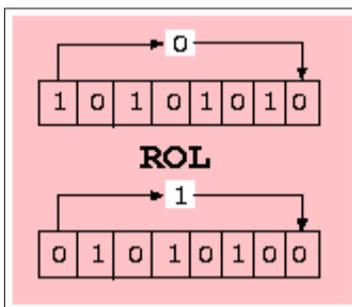
```

```

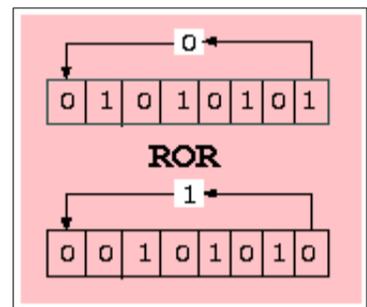
    clc ; clear carry bit
    rol rm1 ; rotate LSB left (multiply by 2)
    rol rmh ; rotate carry into MSB and MSB one left
;
; Step 5: Check if there are still one's in binary 2, if yes, go on multiplying
;
    tst rm2 ; all bits zero?
    brne MULT8a ; if not, go on in the loop
;
; End of the multiplication, result in reh:rel
;
; Endless loop
;
LOOP:
    rjmp loop

```

Binary rotation



For understanding the multiplication operation, it is necessary to understand the binary rotation instructions ROL and ROR. These instructions shift all bits of a register one position left (ROL) resp. right (ROR). The void position in the register is filled with the content of the carry bit in the status register, the bit that rolls out of the register is shifted to this carry bit. This operation is demonstrated using 0xAA as an example for ROL and 0x55 as an example for ROR.



Multiplication in the studio

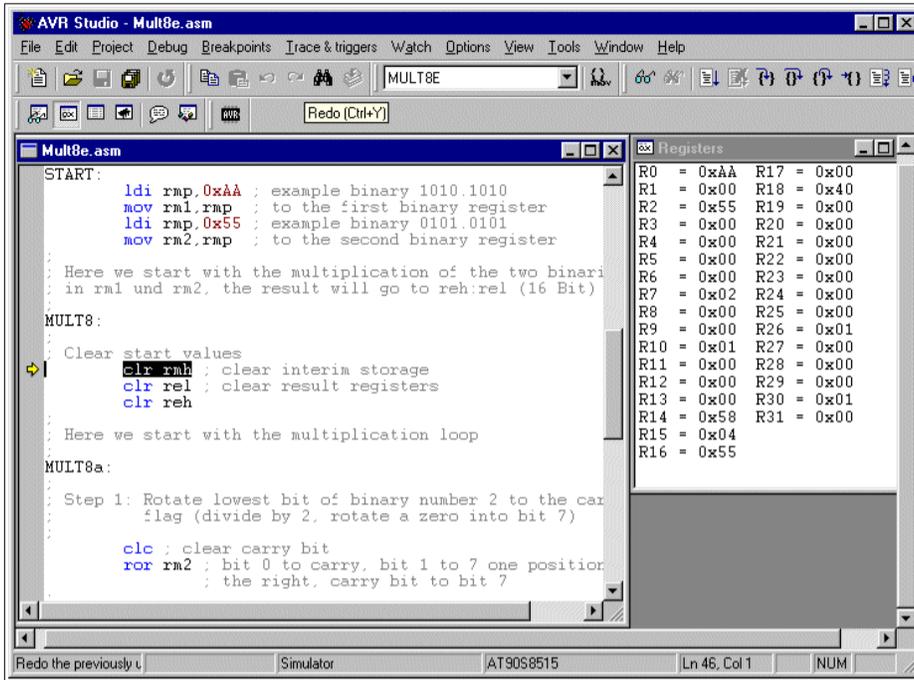
The following screen shots show the multiplication program in the simulator (to make a difference: here Studio version 3).

```

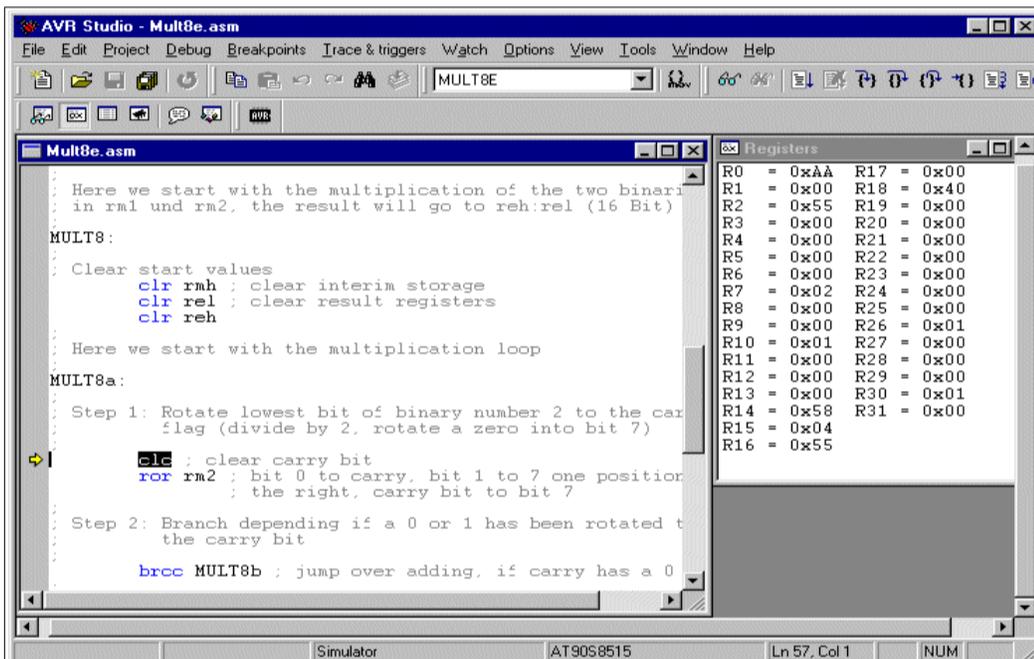
AVR Studio - Mult8e.asm
File Edit Project Debug Breakpoints Trace & triggers Watch Options View Tools Window Help
MULT8E
Mult8e.asm
: Used registers
DEF rm1 = R0 ; Binary number to be multiplied (8 Bit)
DEF rmh = R1 ; Interim storage
DEF rm2 = R2 ; Binary number to be multiplied with (8 Bit)
DEF rel = R3 ; Result, LSB (16 Bit)
DEF reh = R4 ; Result, MSB
DEF rmp = R16 ; Multi purpose register for loading
;
CSEG
ORG 0000
;
rjmp START
START:
    ldi rmp, 0xAA ; example binary 1010.1010
    mov rm1, rmp ; to the first binary register
    ldi rmp, 0x55 ; example binary 0101.0101
    mov rm2, rmp ; to the second binary register
;
; Here we start with the multiplication of the two binaries
; in rm1 und rm2, the result will go to reh:rel (16 Bit)
MULT8:
; Clear start values
    clr rmh ; clear interim storage

```

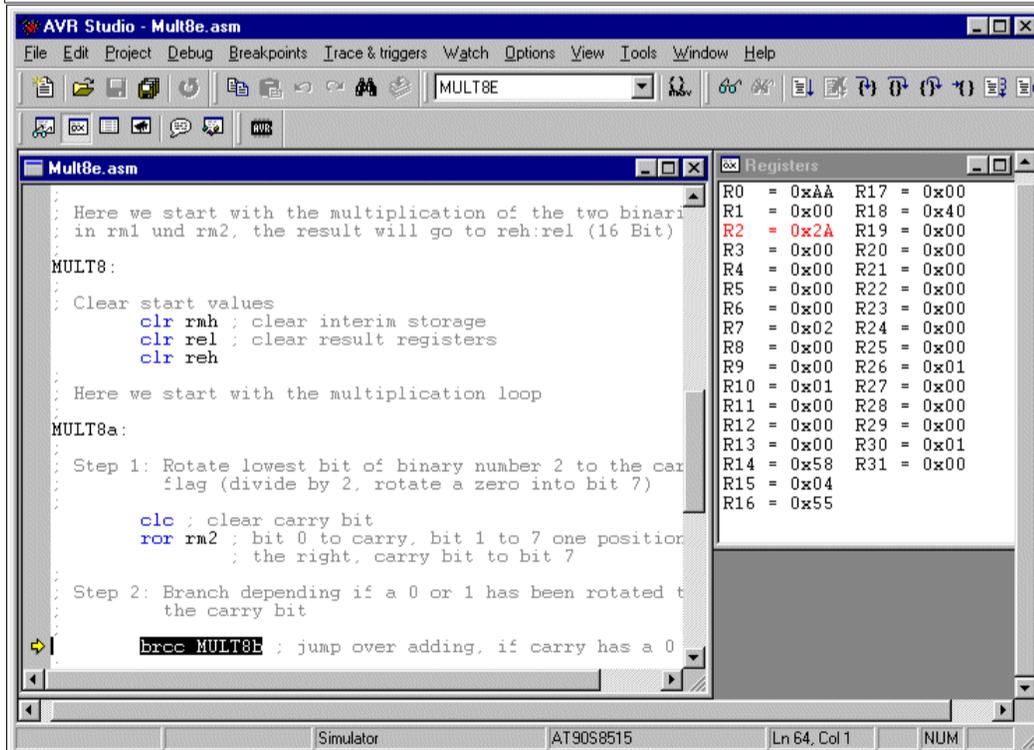
The object-code has been opened, the cursor is placed on the first executable instruction. F11 does single steps.



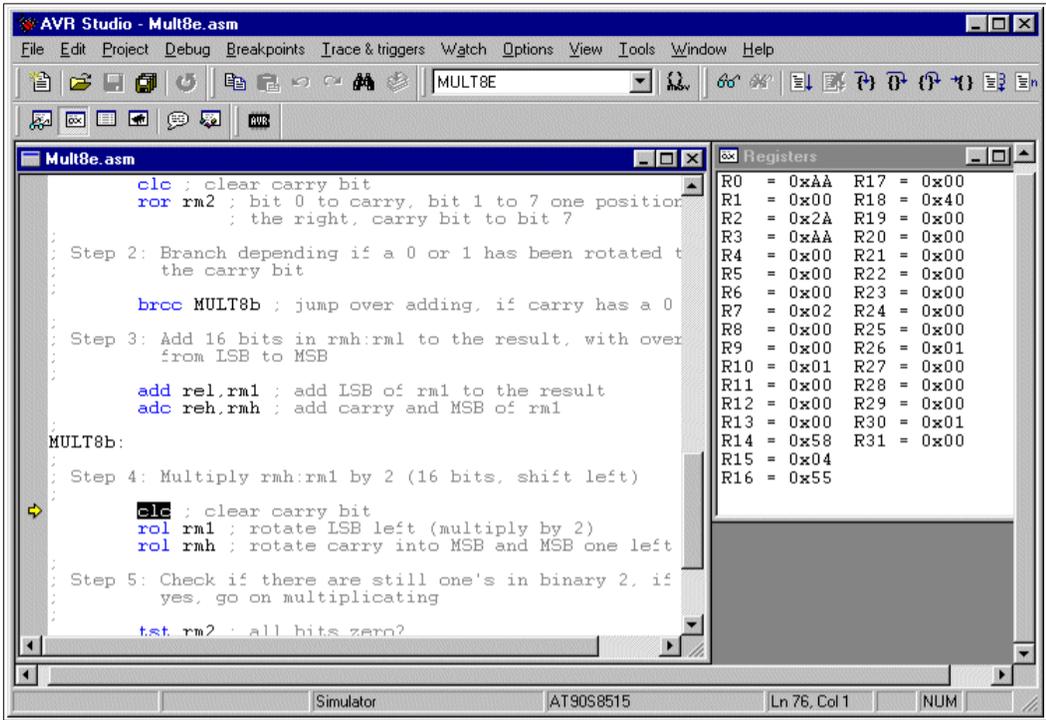
The registers R0 and R2 are set to 0xAA and 0x55, our test binaries, to be multiplied.



R2 is rotated to the right, to roll the least significant bit into the carry bit. 0x55 (0101.0101) yielded 0x2A (0010.1010).

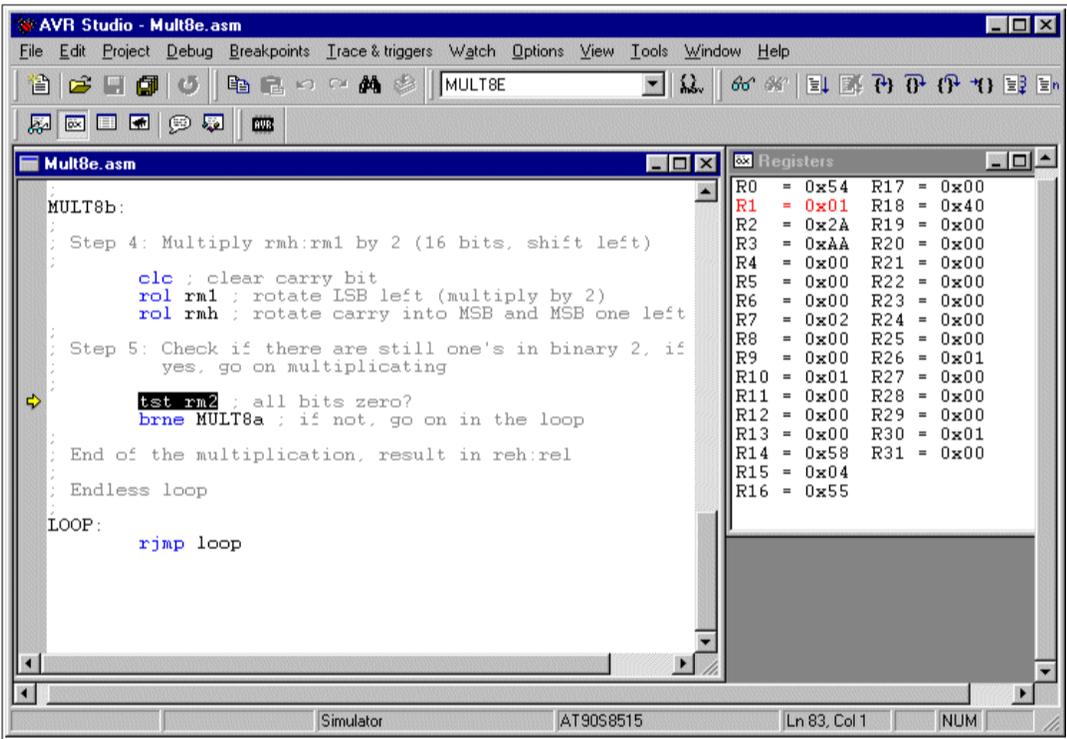


Because the carry bit had a one, the content of the registers R1:R0 is added to the (empty) register pair R4:R3, resulting in 0x00AA there.



Now the register pair R1:R0 is rotated one position left to multiply this binary by 2. From 0x00AA, multiplication by 2 yields 0x0154.

The whole multiplication loop is repeated as long there is at least one binary 1 in register R2. These following loops are not shown here.



Using key F5 of the studio we multi-stepped over these loops to a breakpoint at the end of the multiplication routine. The result register pair R4:R3 has the result of the multiplication of 0xAA by 0x55: 0x3872.

This wasn't that complicated, just remind yourself on the similar decimal operations. Binary multiplication is much easier than decimal.

Hardware multiplication

All ATmega, ATXmega, AT90CAN and AT90PWM have an on-board hardware multiplier, that performs 8 by 8 bit multiplications in only two clock cycles. So whenever you have to do multiplications and you are sure that this software never ever needs not to run on an AT90S- or ATtiny-chip, you can make use of this hardware feature.

The following shows how to multiply

- 8-by-8-binaries,
- 16-by-8-binaries,
- 16-by-16-binaries,
- 16-by-24-binaries.

Hardware multiplication of 8-by-8-bit binaries

The use is simple and straight-forward: if the two binaries to be multiplied are in the registers R16 and R17, just type

```

Tests 8-by-8-bit hardware multiplication with ATmega8
Define Registers
.def ResL = R0
.def ResM = R1
.def m1 = R16
.def m2 = R17

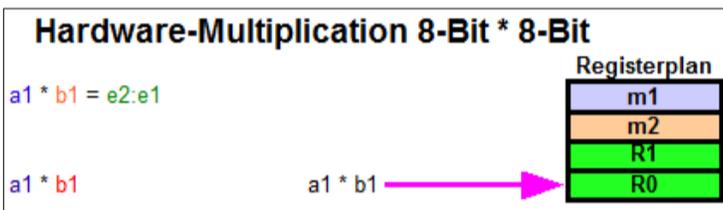
Load multipliers
ldi m1,250
ldi m2,100

Perform multiplication
mul m1,m2

16-bit result is in R1:R0
    
```

mul R16,R17

As the result of these two 8-bit binaries might be up to 16 bits long, the result will be in the registers R1 (most significant byte) and R0 (least significant byte). That's all about it.



The program demonstrates the simulation in the Studio. It multiplies decimal 250 (hex FA) by decimal 100 (hex 64), in the registers R16 and R17.

Register			
R00= 0xA8	R01= 0x61	R02= 0x00	
R03= 0x00	R04= 0x00	R05= 0x00	
R06= 0x00	R07= 0x00	R08= 0x00	
R09= 0x00	R10= 0x00	R11= 0x00	
R12= 0x00	R13= 0x00	R14= 0x00	
R15= 0x00	R16= 0xFA	R17= 0x64	
R18= 0x00	R19= 0x00	R20= 0x00	
R21= 0x00	R22= 0x00	R23= 0x00	
R24= 0x00	R25= 0x00	R26= 0x00	
R27= 0x00	R28= 0x00	R29= 0x00	
R30= 0x00	R31= 0x00		

Processor	
Program Counter	0x000003
Stack Pointer	0x0000
X pointer	0x0000
Y pointer	0x0000
Z pointer	0x0000
Cycle Counter	2
Frequency	1.0000 MHz
Stop Watch	2.00 us
SREG	TTTSSNNZC
Registers	

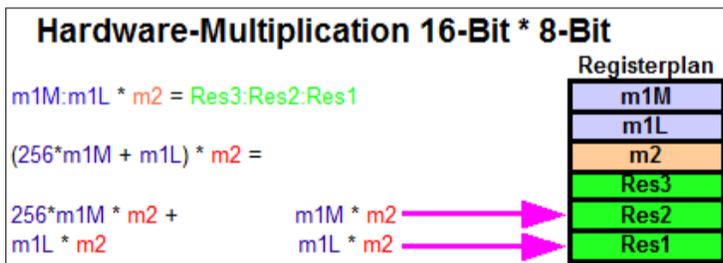
After execution, the registers R0 (LSB) and R1 (MSB) hold the result hex 61A8 or decimal 25,000.

And: yes, that requires only two cycles, or 2 microseconds with a 1 Mcs/s clock.

Hardware multiplication of a 16- by an 8-bit-binary

You have a larger binary to multiply? Hardware is limited to 8, so we need to invest some genius ideas instead. To solve the problem with larger binaries, we just look at this combination of 16 and 8 first. Understanding this concept helps understanding the method, so you will be able to solve the 32-by-64-bit multiplication problem later.

First the math: a 16-bit-binary m1M:m1L are simply two 8-bit-binaries m1M and m1L, where the most significant one m1M of these two is multiplied by decimal 256 or hex 100. (For those who need a reminder: the decimal 1234 is simply (12 multiplied by 100) plus 34, or (1 multiplied by 1000) plus (2 multiplied by 100) plus (3 multiplied by 10) plus 4.



So the 16-bit-binary m1 is equal to 256*m1M plus m1L, where m1M is the MSB and m1L is the LSB. Multiplying m1 by 8-bit-binary m2 so is, mathematically formulated:

- $m1 * m2 = (256*m1M + m1L) * m2$, or
- $256*m1M*m2 + m1L*m2$.

So we just need to do two multiplications and to add both results. Sorry, if you see three asterisks in the formula: the multiplication with 256 in the binary world doesn't require any hardware at all, because it is a simple move to the next higher byte. Just like the multiplication by 10 in the decimal world is simply moving the number one left and write a zero to the least significant digit.

So let's go to a practical example. First we need some registers to

- load the numbers m1 and m2,
- provide space for the result, which might have 24 bits length.

```

; Test hardware multiplication 16-by-8-bit
;
; Register definitions:
;
.def Res1 = R2
.def Res2 = R3
.def Res3 = R4
.def m1L = R16
.def m1M = R17
.def m2 = R18

```

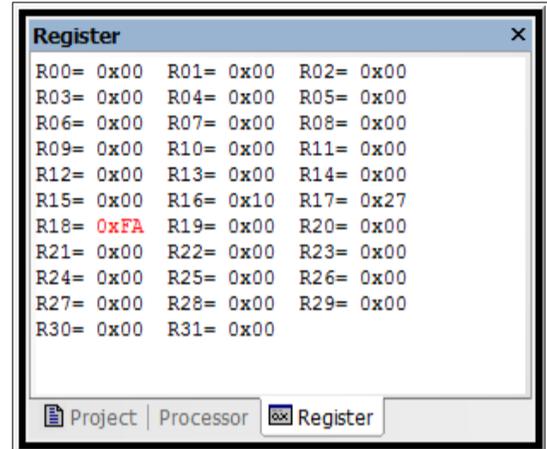
First we load the numbers:

```

;
; Load Registers
;
.equ m1 = 10000
;
    ldi m1M,HIGH(m1) ; upper 8 bits of m1 to m1M
    ldi m1L,LOW(m1) ; lower 8 bits of m1 to m1L
    ldi m2,250 ; 8-bit constant to m2

```

The two numbers are loaded into R17:R16 (dec 10000 = hex 2710) and R18 (dec 250 = hex FA).



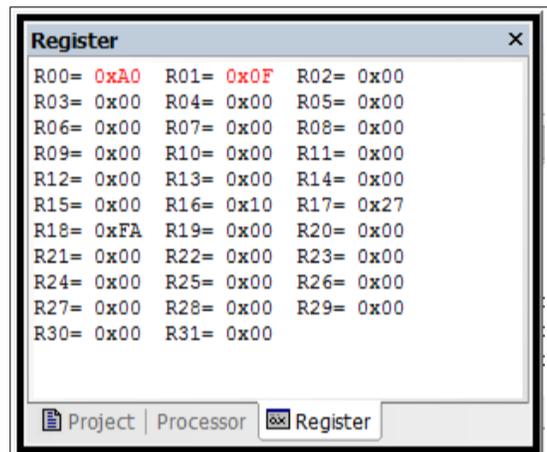
Then we multiply the LSB first:

```

;
; Multiply
;
    mul m1L,m2 ; Multiply LSB
    mov Res1,R0 ; copy result to result register
    mov Res2,R1

```

The LSB multiplication of hex 27 by hex FA yields hex 0F0A, written to the registers R00 (LSB, hex A0) and R01 (MSB, hex 0F). The result is copied to the lower two bytes of the result register, R3:R2.



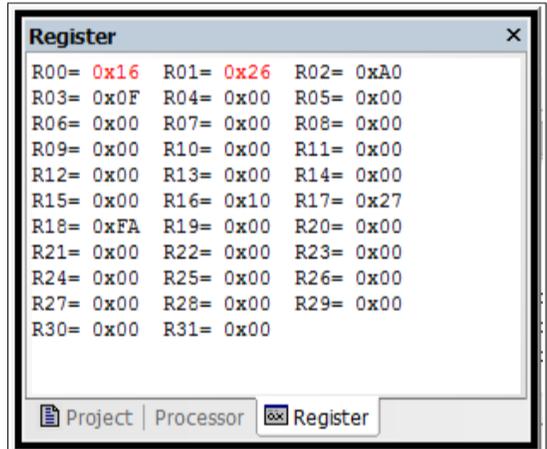
Now the multiplication of the MSB of m1 with m2 follows:

```

    mul m1M,m2 ; Multiply MSB

```

The multiplication of the MSB of m1, hex 10, with m2, hex FA, yields hex 2616 in R1:R0.



Now two steps are performed at once: multiplication by 256 and adding the result to the previous result. This is done by adding R1:R0 to Res3:Res2 instead of Res2:Res1. R1 can just be copied to Res3. R0 is added to Res2 then. If the carry is set after adding, the next higher byte Res3 is increased by one.

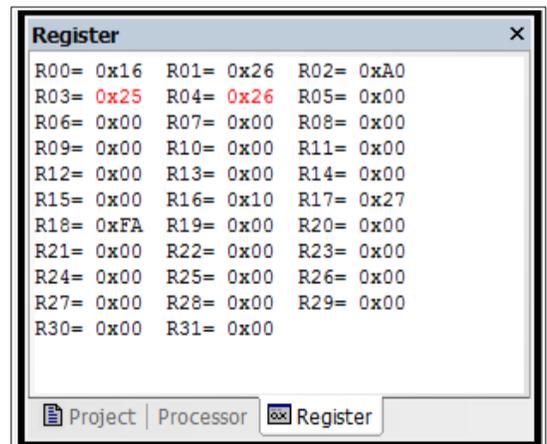
```

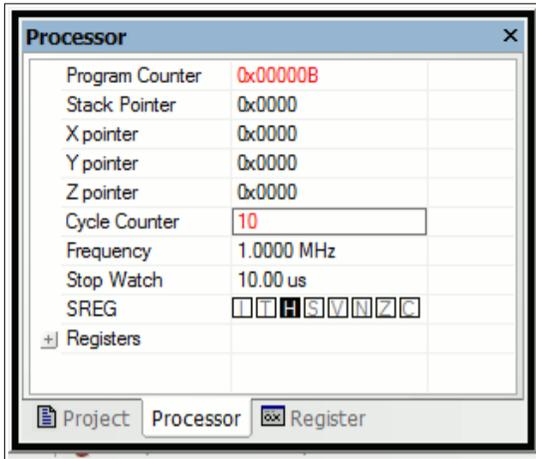
    mov Res3,R1 ; copy MSB result to result byte 3
    add Res2,R0 ; add LSB result to result byte 2
    brc NoInc ; if not carry, jump
    inc Res3

```

NoInc:

The result in R4:R3:R2 is hex 2625A0, which is decimal 2500000 (as everybody knows), and is obviously correct.

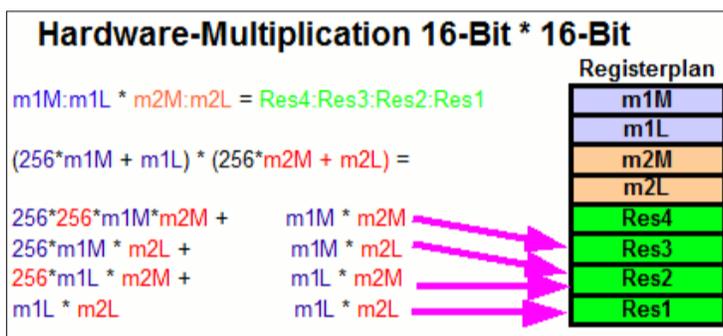




The cycle counter of the multiplication points to 10, at 1 MHz clock a total of 10 microseconds. Very much faster than software multiplication!

Hardware multiplication of a 16- by a 16-bit-binary

Now that we have understood the principle, it should be easy to do 16-by-16. The result requires four bytes now (Res4:Res3:Res2:Res1, located in R5:R4:R3:R2). The formula is:



$$m1 * m2 = (256*m1M + m1L) * (256*m2M + m2L)$$

$$= 65536*m1M*m2M + 256*m1M*m2L + 256*m1L*m2M + m1L*m2L$$

Obviously four multiplications now. We start with the first and the last as the two easiest ones: their results are simply copied to the correct result register positions. The results of the two multiplications in the middle of the formula have to be added to the middle of our result registers, with possible carry overflows to the most significant byte of the result. To do that, you will see a simple trick that is easy to understand. The software:

```

;
; Test Hardware Multiplication 16 by 16
;
; Define Registers
;
.def Res1 = R2
.def Res2 = R3
.def Res3 = R4
.def Res4 = R5
.def m1L = R16
.def m1M = R17
.def m2L = R18
.def m2M = R19
.def tmp = R20
;
; Load input values
;
.equ m1 = 10000
.equ m2 = 25000
;
    ldi m1M,HIGH(m1)
    ldi m1L,LOW(m1)
    ldi m2M,HIGH(m2)
    ldi m2L,LOW(m2)
;
; Multiply
;
    clr R20 ; clear for carry operations
    mul m1M,m2M ; Multiply MSBs
    mov Res3,R0 ; copy to MSW Result
    mov Res4,R1
    mul m1L,m2L ; Multiply LSBs
    mov Res1,R0 ; copy to LSW Result
    mov Res2,R1
    mul m1M,m2L ; Multiply 1M with 2L
    add Res2,R0 ; Add to Result
    adc Res3,R1
    adc Res4,tmp ; add carry
    mul m1L,m2M ; Multiply 1L with 2M

```

```

    add Res2,R0 ; Add to Result
    adc Res3,R1
    adc Res4,tmp
;
; Multiplication done
;

```

Simulation shows the following steps.

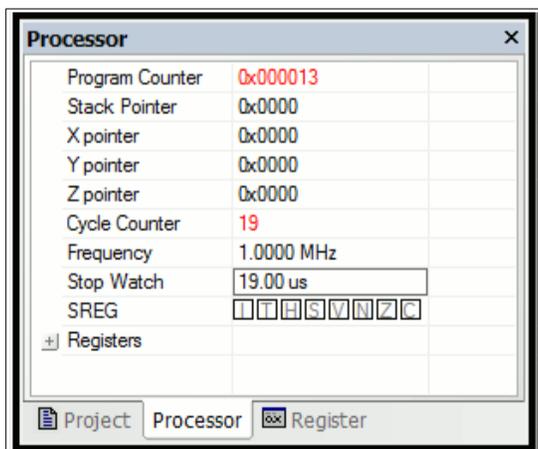
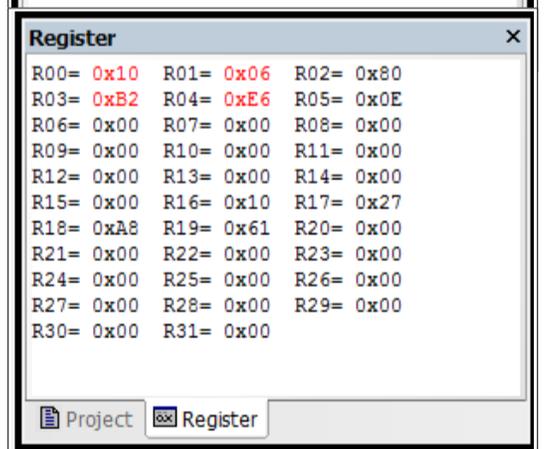
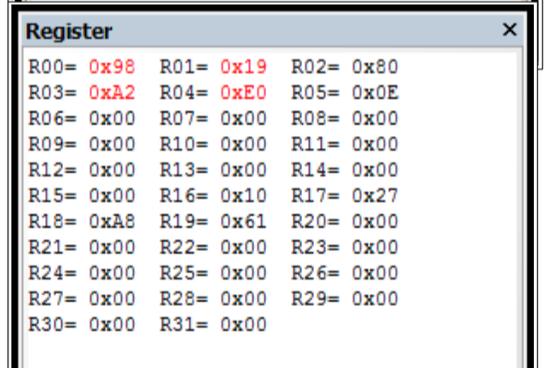
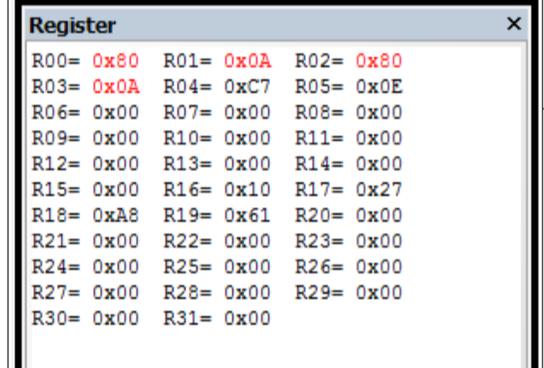
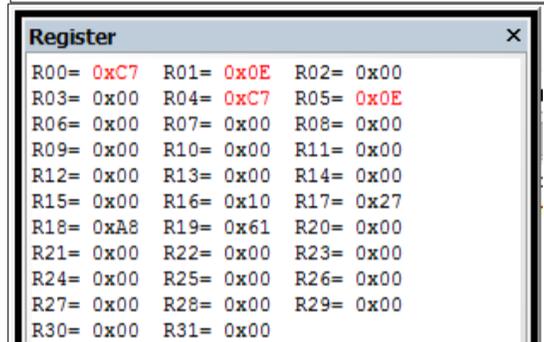
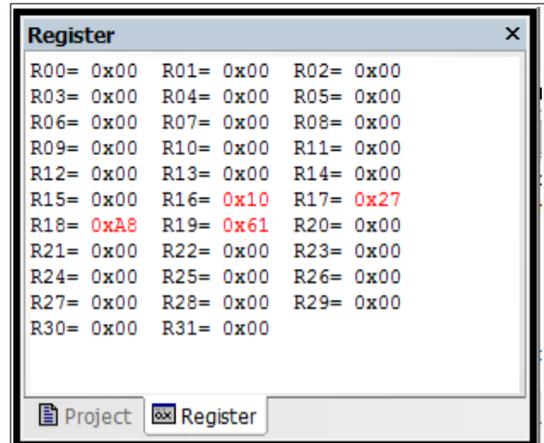
Loading the two constants 10000 (hex 2710) and 25000 (hex 61A8) to the registers in the upper register space ...

Multiplying the two MSBs (hex 27 and 61) and copying the result in R1:R0 to the two most upper result registers R5:R4 ...

Multiplying the two LSBs (hex 10 and A8) and copying the result in R1:R0 to the two lower result registers R3:R2 ...

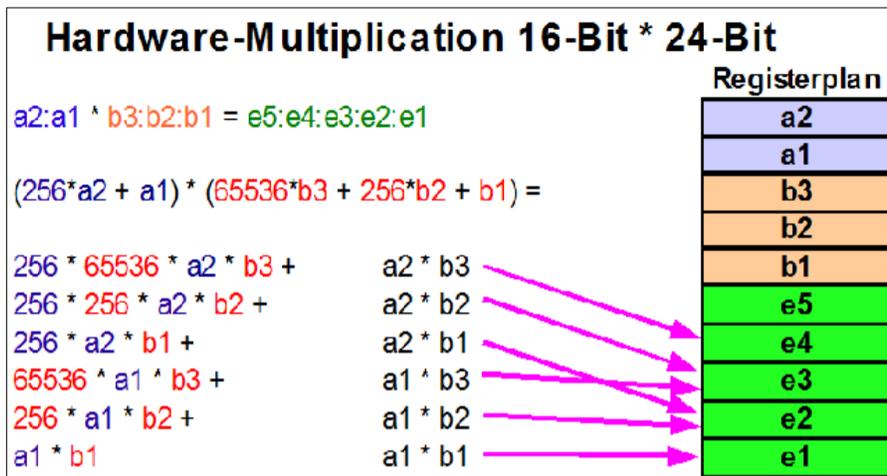
Multiplying the MSB of m1 with the LSB of m2 and adding the result in R1:R0 to the result register's two middle bytes, no carry occurred ...

Multiplying the LSB of m1 with the MSB of m2 and adding the result in R1:R0 to the result register's two middle bytes, no carry occurred. The result is hex 0EE6B280, which is 250000000 and obviously correct ...



Multiplication needed 19 clock cycles, which is very much faster than with software multiplication. Another advantage here: the required time is ALWAYS exactly 19 cycles, and it doesn't depend on the input numbers (like is the case with software multiplication and on overflow occurrences (thanks to our small trick of adding zero with carry). So you can rely on this ...

Hardware multiplication of a 16- by a 24-bit-binary



The multiplication of a 16 bit binary "a" with a 24 bit binary "b" leads to results with up to 40 bit length. The multiplication scheme requires six 8-by-8-bit multiplications and adding the results to the appropriate position in the result registers.

The assembler source code for this:

```

; Hardware Multiplication 16 by 24 bit
.include "m8def.inc"
;
; Register definitions
.def a1 = R2 ; define 16-bit register
.def a2 = R3
.def b1 = R4 ; define 24-bit register
.def b2 = R5
.def b3 = R6
.def e1 = R7 ; define 40-bit result register
.def e2 = R8
.def e3 = R9
.def e4 = R10
.def e5 = R11
.def c0 = R12 ; help register for adding
.def r1 = R16 ; load register
;
; Load constants
.equ a = 10000 ; multiplicator a, hex 2710
.equ b = 1000000 ; multiplicator b, hex 0F4240
    ldi r1,BYTE1(a) ; load a
    mov a1,r1
    ldi r1,BYTE2(a)
    mov a2,r1
    ldi r1,BYTE1(b) ; load b
    mov b1,r1
    ldi r1,BYTE2(b)
    mov b2,r1
    ldi r1,BYTE3(b)
    mov b3,r1
;
; Clear registers
clr e1 ; clear result registers
clr e2
clr e3
clr e4
clr e5
clr c0 ; clear help register
;
; Multiply
mul a2,b3 ; term 1
add e4,R0 ; add to result
adc e5,R1
mul a2,b2 ; term 2
add e3,R0
adc e4,R1
adc e5,c0 ; (add possible carry)
mul a2,b1 ; term 3
add e2,R0
adc e3,R1
adc e4,c0
adc e5,c0
mul a1,b3 ; term 4
add e3,R0
adc e4,R1
adc e5,c0
mul a1,b2 ; term 5
add e2,R0
adc e3,R1
adc e4,c0
adc e5,c0
mul a1,b1 ; term 6
add e1,R0
adc e2,R1
adc e3,c0

```

```

        adc e4,c0
        adc e5,c0
;
; done.
        nop
; Result should be hex 02540BE400

```

The complete execution requires

- 10 clock cycles for loading the constants,
- 6 clock cycles for clearing registers, and
- 33 clock cycles for multiplication.

Division

No, unfortunately there is no hardware division. You need to do this in software!

Decimal division

Again we start with the decimal division, to better understand the binary division. We assume a division of 5678 by 12. This is done like this:

```

                5678 : 12 = ?
-----
- 4 * 1200 = 4800
              ----
- 7 *  120 =  840
              ---
- 3 *   12 =   36
              --
                2
Result: 5678 : 12 = 473 Remainder 2
=====

```

Binary division

In binary the multiplication of the second number in the above decimal example (4 * 1200, etc.) is not necessary, due to the fact that we have only 0 and 1 as digits. Unfortunately binary numbers have much more single digits than their decimal equivalent, so transferring the decimal division to its binary equivalent is a little bit inconvenient. So the program works a bit different than that.

The division of a 16-bit binary number by a 8-bit binary in AVR assembler is listed in the following section.

```

; Div8 divides a 16-bit-number by a 8-bit-number (Test: 16-bit-number: 0xAAAA, 8-bit-number: 0x55)
.NOLIST
.INCLUDE "C:\avrtools\appnotes\8515def.inc" ; adjust the correct path to your system!
.LIST
; Registers
.DEF rd1l = R0 ; LSB 16-bit-number to be divided
.DEF rd1h = R1 ; MSB 16-bit-number to be divided
.DEF rd1u = R2 ; interim register
.DEF rd2 = R3 ; 8-bit-number to divide with
.DEF rel = R4 ; LSB result
.DEF reh = R5 ; MSB result
.DEF rmp = R16; multipurpose register for loading
;
.CSEG
.ORG 0
    rjmp start
start:
; Load the test numbers to the appropriate registers
    ldi rmp,0xAA ; 0xAAAA to be divided
    mov rd1h,rmp
    mov rd1l,rmp
    ldi rmp,0x55 ; 0x55 to be divided with
    mov rd2,rmp
; Divide rd1h:rd1l by rd2
div8:
    clr rd1u ; clear interim register
    clr reh ; clear result (the result registers
    clr rel ; are also used to count to 16 for the
    inc rel ; division steps, is set to 1 at start)
; Here the division loop starts
div8a:
    clc ; clear carry-bit
    rol rd1l ; rotate the next-upper bit of the number
    rol rd1h ; to the interim register (multiply by 2)
    rol rd1u
    brcs div8b ; a one has rolled left, so subtract

```

```

    cp rdlu,rd2 ; Division result 1 or 0?
    brcs div8c ; jump over subtraction, if smaller
div8b:
    sub rdlu,rd2; subtract number to divide with
    sec      ; set carry-bit, result is a 1
    rjmp div8d ; jump to shift of the result bit
div8c:
    clc      ; clear carry-bit, resulting bit is a 0
div8d:
    rol rel ; rotate carry-bit into result registers
    rol reh
    brcc div8a ; as long as zero rotate out of the result registers: go on with the division loop
; End of the division reached
stop:
    rjmp stop ; endless loop

```

Program steps during division

During execution of the program the following steps are ran:

- Definition and preset of the registers with the test binaries,
- presetting the interim register and the result register pair (the result registers are presetted to 0x0001! After 16 rotations the rolling out of the one stops further division steps.),
- the 16-bit-binary in rd1h:rd1l is rotated bitwise to the interim register rd1u (multiplication by 2), if a 1 is rotated out of rd1u, the program branches to the subtraction step in step 4 immediately,
- the content of the interim register is compared with the 8-bit binary in rd2, if rd2 is smaller it is subtracted from the interim register and the carry-bit is set to one, if rd2 is greater the subtraction is skipped and a zero is set to the carry flag,
- the content of the carry flag is rotated into the result register reh:rel from the right,
- if a zero rotated out of the result register, we have to repeat the division loop, if it was a one the division is completed.

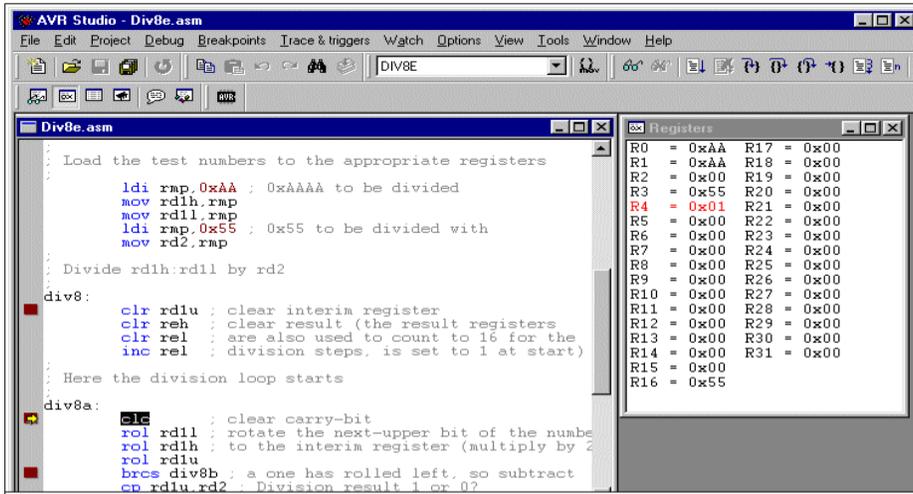
If you don't understand rotation yet you'll find this operation discussed in the multiplication section.

Division in the simulator

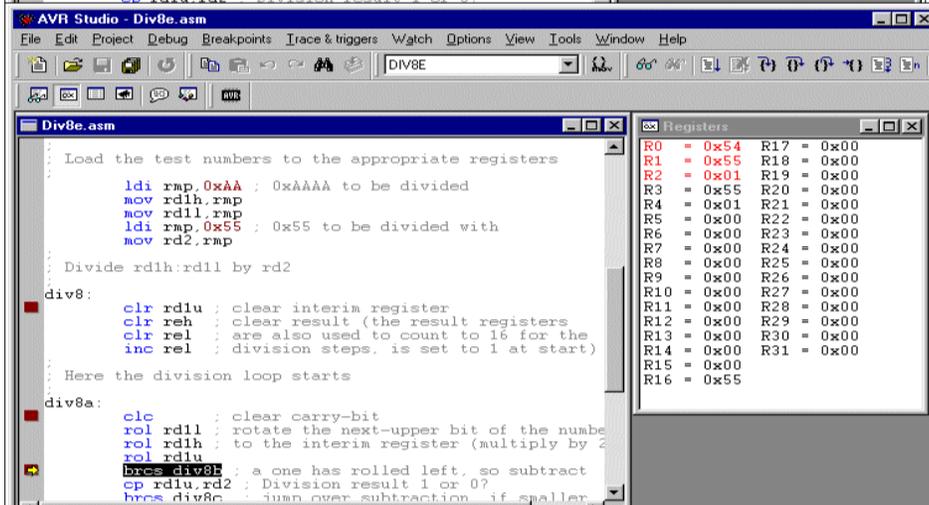
The following screen shots demonstrate the program steps in the studio (here in version 3, so it looks different). To do this, you have to assemble the source code and open the resulting object file in the studio.

The object code has been started, the cursor (yellow arrow) is on the first executable instruction. The key F11 performs single steps.

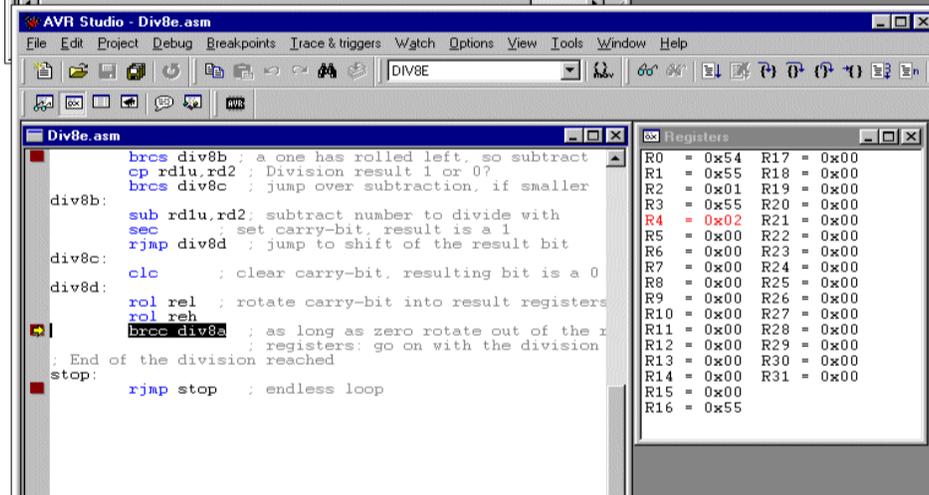
The test binaries 0xAAAA and 0x55, to be divided, have been written to the registers R1:R0 and R3.



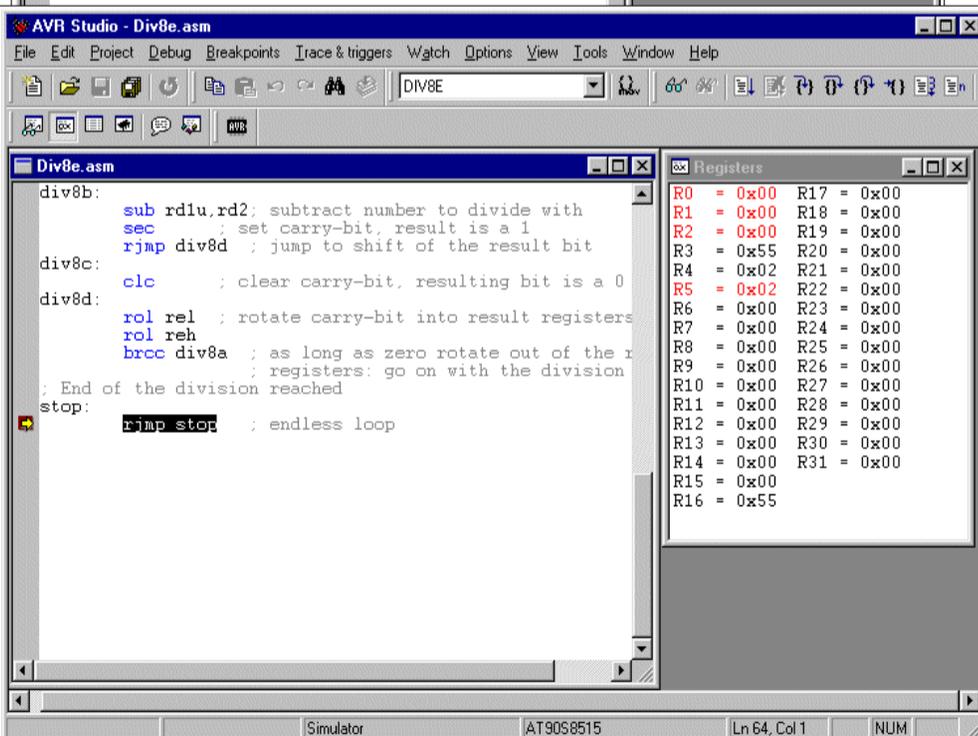
The interim register R2 and the result register pair are set to their predefined values.



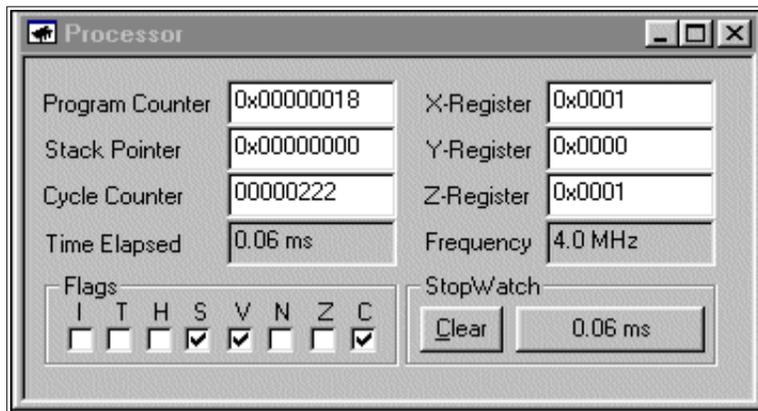
R1:R0 was rotated left to R2, from 0xAAA the doubled value of 0x015554 was yielded.



No overflow from rotation into carry has occurred and 0x01 in R2 was smaller than 0x55 in R3, so subtraction was skipped. A zero in the carry is rotated into the result register R5:R4. The former content of the result register, a single 1-bit in position 0 has rotated to position 1 (content now: 0x0002). As a zero was rotated out of the result register pair, the next step to be executed is a branch to the beginning of the division loop start and the loop is repeated.



After executing the loop 16 times we have reached the breakpoint set at the end of the division routine. The result register in R5:R4 holds 0x0202, the result of the division. The registers R2:R1:R0 are empty, so we do not have a remainder left. If a remainder would have been resulted we can use it to decide whether an incrementation of the result should take place, rounding of the result up. This step is not coded here.



The whole division needs 60 micro-seconds processor time (open a processor view in the studio menu). A rather long time for a division.

Number conversion

Number conversion routines are not included here. Please refer to the website at

http://www.avr-asm-tutorial.net/avr_en

if you need the source code or a better understanding.

Decimal Fractions

First: Do not use any floating points, unless you really need them. Floating points are resource killers in an AVR, lame ducks and need extreme execution times. Run into this dilemma, if you think assembler is too complicated, and you prefer Basic or other languages like C or Pascal.

Not so, if you use assembler. You'll be shown here, how you can perform the multiplication of a fixed point real number in less than 60 micro-seconds, in special cases even within 18 micro-seconds, at 4 MHz clock frequency. Without any floating point processor extensions and other expensive tricks for people too lazy to use their brain.

How to do that? Back to the roots of math! Most tasks with floating point reals can be done using integer numbers. Integers are easy to program in assembler and perform fast. The decimal point is only in the brain of the programmer, and is added somewhere in the decimal digit stream. No one realizes, that this is a trick.

Linear conversions

As an example the following task: an 8-Bit-AD-Converter measures an input signal in the range from 0.00 to 2.55 Volt, and returns as the result a binary in the range from \$00 and \$FF. The result, a voltage, is to be displayed on a LCD display. Silly example, as it is so easy: The binary is converted to a decimal ASCII string between 000 and 255, and just behind the first digit the decimal point has to be inserted. Done!

The electronics world sometimes is more complicated. E. g., the AD-Converter returns an 8-Bit-Hex for input voltages between 0.00 and 5.00 Volt. Now we're tricked and do not know how to proceed. To display the correct result on the LCD we would have to multiply the binary by 500/255, which is 1.9608. This is a silly number, as it is almost 2, but only almost. And we don't want that kind of inaccuracy of 2%, while we have an AD-converter with around 0.25% accuracy.

To cope with this, we multiply the input by $500/255 \cdot 256$ or 501.96 and divide the result by 256. Why first multiply by 256 and then divide by 256? It's just for enhanced accuracy. If we multiply the input by 502 instead of 501.96, the error is just in the order of 0.008%. That is good enough for our AD-converter, we can live with that. And dividing by 256 is an easy task, because it is a well-known power of 2. By dividing with numbers that are a power of 2, the AVR feels very comfortable and performs very fast. By dividing with 256, the AVR is even faster, because we just have to skip the last byte of the binary number. Not even shift and rotate!

The multiplication of an 8-bit-binary with the 9-bit-binary 502 (hex 1F6) can have a result larger than 16 bits. So we have to reserve 24 bits or 3 registers for the result. During multiplication, the constant 502 has to be shifted left (multiplication by 2) to add these numbers to the result each time a one rolls out of the shifted input number. As this might need eight shifts left, we need further three bytes for this constant. So we chose the following combination of registers for the multiplication:

Number	Value (example)	Register
Input value	255	R1
Multiplicand	502	R4 : R3 : R2
Result	128,010	R7 : R6 : R5

After filling the value 502 (00.01.F6) to R4 : R3 : R2 and clearing the result registers R7 : R6 : R5, the multiplication goes like this:

1. Test, if the input number is already zero. If yes, we're done.
2. If no, one bit of the input number is shifted out of the register to the right, into the carry, while a zero is stuffed into bit 7. This instruction is named Logical-Shift-Right or LSR.
3. If the bit in carry is a one, we add the multiplicand (during step 1 the value 502, in step 2 it's 1004, a. s. o.) to the result. During adding, we care for any carry (adding R2 to R5 by ADD, adding R3 to R6 and R4 to R7 with the ADC instruction!). If the bit in the carry was a zero, we just don't add the multiplicand to the result and jump to the next step.
4. Now the multiplicand is multiplied by 2, because the next bit shifted out of the input number is worth double as much. So we shift R2 to the left (by inserting a zero in bit 0) using LSL. Bit 7 is shifted to the carry. Then we rotate this carry into R3, rotating its content left one bit, and bit 7 to the carry. The same with R4.
5. Now we're done with one digit of the input number, and we proceed with step 1 again.

The result of the multiplication by 502 now is in the result registers R7 : R6 : R5. If we just ignore register R5 (division by 256), we have our desired result. To enhance accuracy, we can use bit 7 in R5 to round the result. Now we just have to convert the result from its binary form to decimal ASCII (see Conversion bin to decimal-ASCII on the website). If we just add a decimal point in the right place in the ASCII string, our voltage string is ready for the display.

The whole program, from the input number to the resulting ASCII string, requires between 79 and 228 clock cycles, depending from the input number. Those who want to beat this with the floating point routine of a more sophisticated language than assembler, feel free to mail me your conversion time (and program flash and memory usage).

Example 1: 8-bit-AD-converter with fixed decimal output

```

; Demonstrates floating point conversion in Assembler, (C)2003 www.avr-asm-tutorial.net
;
; The task: You read in an 8-bit result of an analogue-digital-converter, number is in the range from hex 00 to FF.
; You need to convert this into a floating point number in the range from 0.00 to 5.00 Volt
; The program scheme:
; 1. Multiplication by 502 (hex 01F6). That step multiplies by 500, 256 and divides by 255 in one step!
; 2. Round the result and cut the last byte of the result. This step divides by 256 by ignoring the last byte of the result.
;    Before doing that, bit 7 is used to round the result.
; 3. Convert the resulting word to ASCII and set the correct decimal sign. The resulting word in the range from 0 to 500
;    is displayed in ASCII-characters as 0.00 to 5.00.
; The registers used:
; The routines use the registers R8..R1 without saving these before. Also required is a multipurpose register called rmp,
; located in the upper half of the registers. Please take care that these registers don't conflict with the register use in the
; rest of your program.
; When entering the routine the 8-bit number is expected in the register R1. The multiplication uses R4:R3:R2 to hold
; the multiplier 502 (is shifted left max. eight times during multiplication). The result of the multiplication is calculated
; in the registers R7:R6:R5. The result of the so called division by 256 by just ignoring R5 in the result, is in R7:R6. R7:R6
; is rounded, depending on the highest bit of R5, and the result is copied to R2:R1.
; Conversion to an ASCII-string uses the input in R2:R1, the register pair R4:R3 as a divisor for conversion, and places the
; ASCII result string to R5:R6:R7:R8 (R6 is the decimal char).
; Other conventions:
; The conversion uses subroutines and the stack. The stack must work fine for the use of three levels (six bytes SRAM).
; Conversion times:
; The whole routine requires 228 clock cycles maximum (converting $FF), and 79 clock cycles minimum (converting $00).
; At 4 MHz the times are 56.75 microseconds resp. 17.75 microseconds.
; Definitions:
; Registers
.DEF rmp = R16 ; used as multi-purpose register
; AVR type: Tested for type AT90S8515, only required for stack setting, routines work fine with other AT90S-types also
.NOLIST
.INCLUDE "8515def.inc"
.LIST
; Start of test program
; Just writes a number to R1 and starts the conversion routine, for test purposes only
.CSEG
.ORG $0000
    rjmp main
main:
    ldi rmp,HIGH(RAMEND) ; Set the stack
    out SPH,rmp
    ldi rmp,LOW(RAMEND)
    out SPL,rmp
    ldi rmp,$FF ; Convert $FF
    mov R1,rmp
    rcall fpconv8 ; call the conversion routine
no_end: ; unlimited loop, when done
    rjmp no_end
; Conversion routine wrapper, calls the different conversion steps
fpconv8:
    rcall fpconv8m ; multiply by 502
    rcall fpconv8r ; round and divide by 256
    rcall fpconv8a ; convert to ASCII string

```

```

    ldi rmp,',' ; set decimal char
    mov R6,rmp
    ret ; all done
; Subroutine multiplication by 502
fpconv8m:
    clr R4 ; set the multiplicand to 502
    ldi rmp,$01
    mov R3,rmp
    ldi rmp,$F6
    mov R2,rmp
    clr R7 ; clear the result
    clr R6
    clr R5
fpconv8m1:
    or R1,R1 ; check if the number is all zeros
    brne fpconv8m2 ; still one's, go on convert
    ret ; ready, return back
fpconv8m2:
    lsr R1 ; shift number to the right (div by 2)
    brcc fpconv8m3 ; if the lowest bit was 0, then skip adding
    add R5,R2 ; add the number in R6:R5:R4:R3 to the result
    adc R6,R3
    adc R7,R4
fpconv8m3:
    lsl R2 ; multiply R4:R3:R2 by 2
    rol R3
    rol R4
    rjmp fpconv8m1 ; repeat for next bit
; Round the value in R7:R6 with the value in bit 7 of R5
fpconv8r:
    clr rmp ; put zero to rmp
    lsl R5 ; rotate bit 7 to carry
    adc R6,rmp ; add LSB with carry
    adc R7,rmp ; add MSB with carry
    mov R2,R7 ; copy the value to R2:R1 (divide by 256)
    mov R1,R6
    ret
; Convert the word in R2:R1 to an ASCII string in R5:R6:R7:R8
fpconv8a:
    clr R4 ; Set the decimal divider value to 100
    ldi rmp,100
    mov R3,rmp
    rcall fpconv8d ; get ASCII digit by repeated subtraction
    mov R5,rmp ; set hundreds string char
    ldi rmp,10 ; Set the decimal divider value to 10
    mov R3,rmp
    rcall fpconv8d ; get the next ASCII digit
    mov R7,rmp ; set tens string char
    ldi rmp,'0' ; convert the rest to an ASCII char
    add rmp,R1
    mov R8,rmp ; set ones string char
    ret
; Convert binary word in R2:R1 to a decimal digit by subtracting the decimal divider value in R4:R3 (100, 10)
fpconv8d:
    ldi rmp,'0' ; start with decimal value 0
fpconv8d1:
    cp R1,R3 ; Compare word with decimal divider value
    cpc R2,R4
    brcc fpconv8d2 ; Carry clear, subtract divider value
    ret ; done subtraction
fpconv8d2:
    sub R1,R3 ; subtract divider value
    sbc R2,R4
    inc rmp ; up one digit
    rjmp fpconv8d1 ; once again
; End of conversion test routine

```

Example 2: 10-bit-AD-converter with fixed decimal output

This example is a bit more complicated. Refer to the website if you need it.

Floating point numbers in assembler language

Floating points, if necessary

Those who want to make life more complicated than necessary: besides whole numbers (integers), signed integers and fixed dot numbers floating point numbers are available. What comes in higher-level languages simply as 1.234567 is rather complicated in assembler. And that comes as follows.

The format of floating point numbers

Binary floating points consist of two constituents:

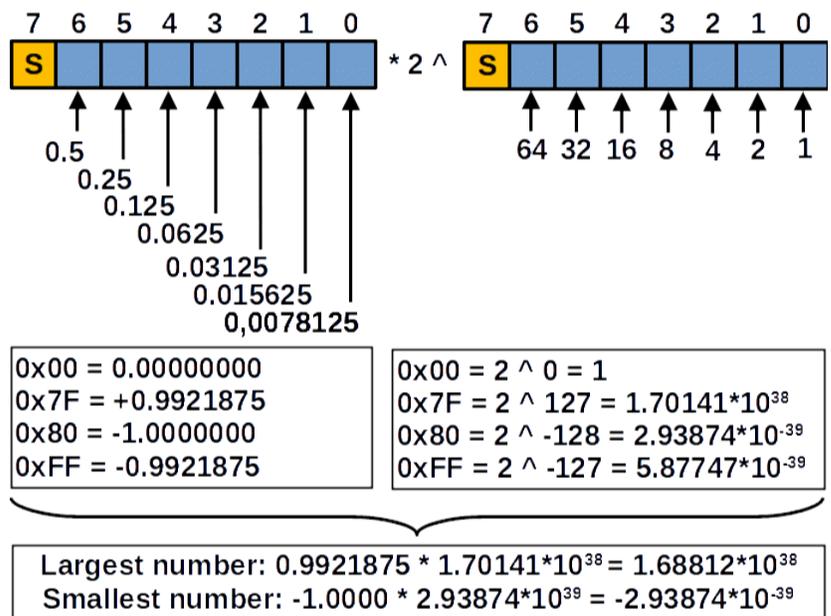
1. a mantissa, and
2. an exponent.

In the decimal world the mantissa gives the normal number part, in 1.234567 this is the 1.234567. The precision of the number is given by the seven digits. The exponent says how often the mantissa has to be multiplied by 10 (the base in the decimal world). In our example this would be a zero. The number could also be written as $0.1234567 \cdot 10^1$ or shorter as 0.1234567E+01, which says: shift the mantissa **one time left**. It could also be written as 12.34567E-01 to say: shift the mantissa **one time right**. The formulation 1.234567E+00 is called normalized in that it

- it has only one single digit left of the dot, and
- this digit is not zero.

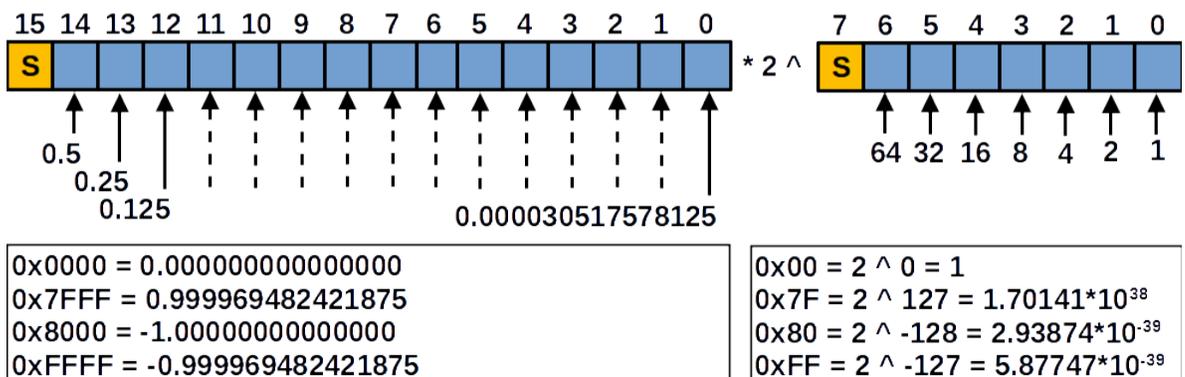
Numbers larger than 9.999999 are repeatedly divided by 10, by that increasing the exponent. Numbers smaller than 1 are repeatedly multiplied by 10, by that decreasing the exponent. Numbers smaller than one have a negative exponent. That is why the exponent has to be a signed integer.

Numbers themselves also can be negative, such as -1.234567. Because multiplying and dividing does not change the sign, the mantissa also needs a sign bit. So we can handle positive as well as negative numbers, such as -182.162°C as the boiling point of oxygen. Of course we'll have to divide this boiling point by 100 to get a normalized mantissa, and its exponent will be plus two. Normalized we'll get -1.82162E+02 for that boiling point.



Converted to the binary world, where the base is 2, the floating point numbers need at least two bytes: one for the mantissa and one for the exponent. Both are signed integers. The meaning of one bit in the mantissa and one bit in the exponent is very different:

1. In the mantissa each bit, starting from the dot, or better: from its highest non-sign-bit, stands for 1 divided by 2, powered by n, where n is its position in the mantissa. So the first bit is $1 / 2^1 = 1 / 2$, or in decimal 0.5. Each further bit stands for half of the previous bit, so the next in the line is 0.25, the over-next is 0,125 etc. etc.
2. The exponent is simpler to understand: in an 8-bit exponent it reaches from zero to 127 (hexadecimal 0x00 to 0x7F) for positive exponents and from -1 to -128 (hexadecimal 0xFF for -1, 0x80 for -128) for negative exponents. This says that for each positive number the mantissa has to be shifted n positions to the left, for negative ones shifted one position to the right. A left shift means multiplying the mantissa by two, right shift a division by two.



Because the exponent shifts the number by its power of two ($\cdot 2^n$), each bit of it is more powerful than a bit in the mantissa. So 2^{127} is 1.7-multiplied-by-10-power-38 or $1.7 \cdot 10^{38}$ or even shorter 1.7E38. Vice versa,

negative exponents make the exponent part of the number very small: 2^{-128} is decimal $2.9E-39$. With eight bit exponent only we can cover the range of numbers between $2.9E-39$ to $1.7E+38$. That should be enough large or small, not for an astronomer but for most of the rest of the calculating mankind. So an 8-bit exponent is sufficient.

Very small are the variations that come with the mantissa: as can be seen from an 8-bit mantissa's $0x7F$ that its decimal value is only 0.992 and by only 0.008 below the one. So we can one handle numbers with slightly more than two digit ($2\frac{1}{2}$) precision in an 8-bit mantissa. By far not enough for calculating interest rates or other commercial stuff or in engineering, only suitable for rather rough technical measurements. 8-bit mantissa's are of the same accuracy as an ancient slide rule (for those who are still familiar with that kind of calculating machines).

To increase the precision we add additional eight bits to the mantissa. The lowest of the mantissa's bits stands now for 0.0000305. This increases the precision to slightly more than four digits. If we would add another byte to the mantissa we are at slightly more than six decimal digits, the complete number has already 32 bits or four bytes. 16-bit mantissas are not precise enough to calculate Mandelbrot-sets, but are suitable for most technical applications.

If you need higher resolutions, pick a needed style from this table.

Total bits	Bits		Maximum		Resolution
	Exponent	Mantissa	Exponent	Mantissa	Dec. Digits
16	8	8	127	1,28E+02	3 ½
24	8	16	127	3,28E+04	4 ½
32	8	24	127	8,39E+06	6 ½
40	8	32	127	2,15E+09	9 ½
48	8	40	127	5,50E+11	11 ½
	16	32	32767	2,15E+09	9 ½
56	8	48	127	1,41E+14	14
	16	40	32767	5,50E+11	11 ½
64	8	56	127	3,60E+16	16 ½
	16	48	32767	1,41E+14	14
	24	40	8388607	5,50E+11	11 ½

Because one additional mantissa bit can increase precision by roughly a half decimal digit, the inventors of binary floats increased it by one with a trick: because a normalized binary mantissa always starts with a one, this bit can be skipped and an additional bit fits into the 16 bit mantissa at the end. These kind of tricks increase the variability of floating number formats and make it more and more complicated to understand: of course the skipped one-bit on top has to be added when calculating with the mantissa. It can replace the mantissa's sign bit, if that bit sign bit is stored elsewhere.

An advantage do those floats have: they simplify the multiplication and division of two floats. If we have to multiply two floats with their mantissas $M1$ and $M2$, we can simply multiply the two mantissas and, even more simple, add their two exponents $E1$ and $E2$. When dividing, we have to subtract $E2$ from $E1$.

The simplification when multiplying is associated by a higher effort when adding or subtracting. Before we can add the two mantissas we have to bring their exponents to the same value (by shifting the mantissa of the smaller number to the right). Only when both are equal, we can add both mantissas.

Conversion of binary to decimal number format

To demonstrate that handling binary float numbers is rather extensive, I have shown the conversion of a 24-bit float with a 16-bit mantissa in detail. The software for doing that has 410 code lines and needs a few milli-seconds in an AVR. How this is done is documented on [this page here](#). If you want to learn assembler: this is a more high-level example, with lots of pointers. I hope that you enjoy the understanding of a more complex task.

Conclusion:

Those who are clever and do not need numbers up to 10^{38} (or even larger) avoid floats and rather use integers or fixed floating point numbers (Pseudo-floats). Those are by far simpler to handle, easier to understand and it is rather simpler to adjust their precision to the given practical needs.

Converting floating point numbers to decimal in assembler language

To convert floating point binaries into decimal (ASCII) numbers we need, of course, some binary and exponential math. If you are weak in both disciplines and if you are lazy, do not try to understand the following, pick a floating point library instead. If you really want to know how it works: go on reading, it is not too complicated to understand.

Allocation of numbers

As has been shown on the previous pages, a 24-bit binary consists of 16 bits for the mantissa and of 8 bits for the exponent. Both components take their most significant bit as sign. We can easily store these two components in three bytes, e. g. in three registers of the AVR.

The decimal resolution of such a binary number is $4\frac{1}{2}$ digits. To convert these back to decimal we need some more space as each digit needs one byte. So we better place the decimal result, together with some interim numbers that are needed during conversion to the SRAM, so we do not need to mess around with register needs and shortages. You can also increase the resolution simply by extending the SRAM

reservation, and the software adds more steps.

15 bits mantissa in binary format corresponds with five decimal digits ($2^{15} = 32,768$). The format of the result is as follows:

0	1	2	3	4	5	6	7	8	9	10	11
	1	.	0	6	6	6	5	E	+	0	1

We need the mantissa's sign bit (if positive we use a blank), the normalized first digit, then the decimal dot, four significant and one insignificant digits, then the **E**, the exponent's sign (+ or -) and two exponent digits. In total we are at 12 bytes result.

So this is the space that should be reserved for the result. To reserve space for that in assembler, we write:

```
.dseg
.org SRAM_START
DecAsc:
.byte 12 ; Reserve 12 bytes for result
```

The conversion involves adding decimal numbers with

- a resolution of six digits, so we can handle more precision than needed,
- an additional space for three further digits, with which we can do the rounding at the end,
- to the left we add another digit to allow for overflows when multiplying the decimal by two during conversion of the binary exponent.

So we are at 10-digit numbers for handling the decimal digits. We will need two buffers for that: one for calculation of the mantissa's value and one to prepare the adders of the mantissa's bits. We add some space to place those numbers on the beginning of a line in SRAM to ease reading in simulation, but can leave these reservations aside when space gets scarce. Our SRAM space now looks like this:

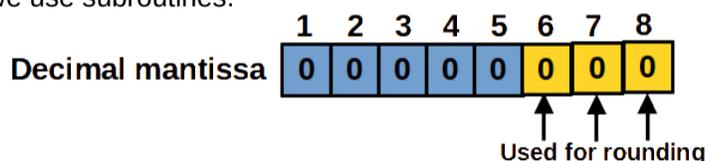
```
.dseg
.org SRAM_START
.equ MantLength = 10
sMant:
.byte MantLength
sMantEnd:
.byte 16-MantLength
sAdder:
.byte MantLength
sAdderEnd:
.byte 16-MantLength
sDecAsc:
.byte 12
```

The two **End:** labels are for checking if the end has been reached, or, in case we have to start from the end of the number, to place a pointer right behind the number.

A basic decision is to handle the calculations in simple binary format, where 0 to 9 are handled as binaries 0 to 9. This requires one byte per digit and does not involve the H flag (in case of packed BCD) or the ASCII format bits when handling ASCII numbers. This is much simpler than in other formats, but needs slightly more time.

In the first step we init the stackpointer, because we use subroutines.

The second step is to get rid of the mantissa's sign bit. If bit 7 of the mantissa is zero we can skip the following. If it is one we subtract the LSB from zero and invert the MSB. This makes a positive number from the negative. The decimal mantissa, and also the adder space, with its eight bytes each now look like this.



Converting the mantissa to decimal



Conversion starts with bit 14 of the binary mantissa. As this bit is always a one, we can skip this by setting the result as well as the adder to 0.50000000. We would formulate this in assembler as follows:

```
; Initiate the decimal mantissa
InitMant:
ldi ZH,High(sMant) ; Point Z to mantissa space, MSB
ldi ZL,Low(sMant) ; dto., LSB
clr rmp ; Clear the complete mantissa space
ldi rCnt,MantLength
```

```

InitMant1:
  std Z+dAddMant,rmp ; Clear the adder
  st Z+,rmp ; Clear the mantissa
  dec rCnt ; At the end?
  brne InitMant1
  ldi rmp,5 ; Start with bit 15
  sts sMant+2,rmp ; Set start value, Mantissa
  sts sAdder+2,rmp ; and adder
  ret

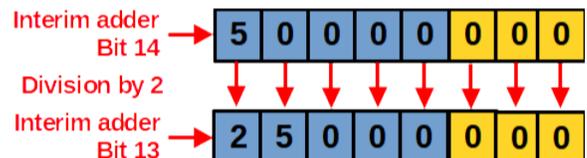
```

Note that both buffers are filled simultaneously with the use of the **STD** instruction. At the end the two **STS** instructions set the **5** to the right place in both buffers.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
S0060	00	00	05	00	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF
S0070	00	00	05	00	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF
S0080	FF															
S0090	FF															
S00A0	FF															
S00B0	FF															
S00C0	FF															
S00D0	FF	00	0B													

The init process has been executed in the simulator [avr_sim](#). Both numbers are set to 0.5 now.

The next step is to divide the decimal adder by two to get the adder for bit 13. Procedure starts with the 5 in the buffer and proceeds over the whole buffer length. If the division by two leaves a remainder (as is already the case with the first digit $5 / 2 = 2$, remainder = 1), 10 has to be added to the next digit. The division by two is a simple task, as the whole algorithm goes like this:



```

; Divide the adder by two
DivideBy2:
  ldi ZH,High(sAdder+1) ; Point to end of the adder, MSB
  ldi ZL,Low(sAdder+1)
  clc ; Clear carry for overflows
  ldi rCnt,MantLength-2 ; Mantissa length minus one to counter
DivideBy2a:
  ld rmp,Z ; Read byte from adder
  brcc DivideBy2b ; Carry is not set, don't add 10
  subi rmp,-10 ; Add ten
DivideBy2b:
  lsr rmp ; Divide by two
  st Z+,rmp ; Store division result
  dec rCnt ; Count down
  brne DivideBy2a
  ld rmp,Z ; Read last byte from adder
  lsr rmp ; Divide by 2
  st Z,rmp
  brcc Divideby2e
  inc rmp ; Round last digit up
Divideby2c:
  st Z,rmp ; Correct last digit
  subi rmp,10 ; Digit > 10?
  brcs DivideBy2e ; Nope
DivideBy2d:
  st Z,rmp ; Correct last digit
  ld rmp,-Z ; Read pre-last digit
  inc rmp ; Increase digit
  st Z,rmp ; and store
  subi rmp,10 ; Check digit >= 10
  brcc DivideBy2d ; Yes, repeat
DivideBy2e:
  ret

```

These show the simulation of the first two divisions of the adder.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	00	00	06	02	05	00	00	00	00	00	FF	FF	FF	FF	FF	FF
\$0070	00	00	01	02	05	00	00	00	00	00	FF	FF	FF	FF	FF	FF
\$0080	FF															
\$0090	FF															
\$00A0	FF															
\$00B0	FF															
\$00C0	FF															
\$00D0	FF	00	15													

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	00	00	05	00	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF
\$0070	00	00	01	02	05	00	00	00	00	00	FF	FF	FF	FF	FF	FF
\$0080	FF															
\$0090	FF															
\$00A0	FF															
\$00B0	FF															
\$00C0	FF															
\$00D0	FF	00	11													

The special here is to increase the last digit if the division of the last digit shifted a one out to carry. In that case the last digit (as accessible with `ld R16,-Z`) has to be increased. If that yields equals or more than ten (**subi R16,10** does not set the carry flag), the overflow has to go back to the previous byte. This has to be repeated with all previous digits until the **INC** does not lead to a digit reaching or exceeding 10 any more.

To the upper right is the next step seen: if the respective mantissa bit is one, the divided adder has to be added to the decimal mantissa. When adding, the procedure starts at the end of the mantissa and adder buffer and proceeds to the left of the buffer. Each digit has to be checked if, by adding the adder byte and the carry, the 10 has been reached or exceeded. If so, ten has to be subtracted and this overflow has to be added to the next digit.

If the mantissa bit is not one, then the next division takes place without adding. The source code for this is here:

```
; Add the adder to the decimal mantissa
MantAdd:
    ldi ZH,High(sMantEnd) ; Point Z to decimal mantissa, MSB
    ldi ZL,Low(sMantEnd) ; dto., LSB
    ldi rCnt,MantLength-1 ; Mantissa length to R16
    clc ; Start with carry clear
MantAdd1:
    ld rmp,-Z ; Read last mantissa byte
    ldd rmp2,Z+dAddMant ; Read corresponding adder byte
    adc rmp,rmp2 ; Add both with carry
    st Z,rmp ; Store in SRAM
    subi rmp,10 ; Subtract 10
    brcs MantAdd2 ; Carry set, smaller than 10
    st Z,rmp ; Overwrite digit
    sec ; Set carry for next digit
    rjmp MantAdd3 ; Count down
MantAdd2:
    clc ; Clear carry for next adding
MantAdd3:
    dec rCnt ; Count down
    brne MantAdd1 ; Not yet complete
    ret
```

This shows the treatment of all 15 bits of the mantissa: dividing in any case, and adding only if the mantissa bit is one. Here shown for a mantissa of `0x5555`, where every second bit is set one.


```

; Normalize the decimal mantissa
Normalize:
  lds rmp,sMant ; Read mantissa overflow byte
  tst rmp ; Not zero?
  brne NormalizeRight ; Shift to the right
Normalize1:
  lds rmp,sMant+1 ; Read the first digit
  tst rmp ; Zero?
  breq NormalizeLeft ; If yes, shift left
  ret ; No normalization necessary
; Shift exponent one position left
NormalizeLeft:
  ldi ZH,High(sMant+1) ; Point to first digit, MSB
  ldi ZL,Low(sMant+1) ; dto., LSB
  ldi rCnt,MantLength-2 ; Shift counter
NormalizeLeft1:
  ldd rmp,Z+1 ; Read the next byte
  st Z+,rmp ; Copy it to the current position
  dec rCnt ; Count down
  brne NormalizeLeft1 ; Additional bytes to move
  clr rmp ; Clear the last digit
  st Z,rmp ; in the last buffer
  dec rDecExp ; Decrease decimal exponent
  rjmp Normalize1 ; Check if further shifts necessary
; Shift number to the right
NormalizeRight:
  ldi ZH,High(sMantEnd-1) ; Place Z to the end, MSB
  ldi ZL,Low(sMantEnd-1) ; dto., LSB
  ldi rCnt,MantLength-1 ; Counter for digits
NormalizeRight1:
  ld rmp,-Z ; Read digit left
  std Z+1,rmp ; Store one position to the right
  dec rCnt ; Count down
  brne NormalizeRight1 ; Further digits
  clr rmp ; Clear the first digit (overflow digit)
  st Z,rmp
  inc rDecExp ; Increase decimal exponent
  ret

```

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
S0060	00	06	06	06	06	05	06	03	00	00	FF	FF	FF	FF	FF	FF
S0070	00	00	00	00	00	00	03	00	05	00	FF	FF	FF	FF	FF	FF
S0080	FF															
S0090	FF															
S00A0	FF															
S00B0	FF															
S00C0	FF															
S00D0	FF	00	55	00	17U..											

The decimal mantissa has been shifted one position to the left and is now normalized.

As the binary exponent is four, now the mantissa has to be multiplied by two. This decreases the binary exponent by one.

The source code for multiplication by 2 is the following:

10 bytes mantissa

0	6	.	6	6	6	5	6	6	2	0
<i>Multiply by two</i>										
1	3	.	3	3	3	1	3	2	4	0

Binary exponent

0	0	0	0	0	0	1	0	0
<i>Decrease exponent</i>								
0	0	0	0	0	0	0	1	1

```

; Multiply number by 2
Multiply2:
  ldi ZH,High(sMantEnd) ; Z to end of mantissa, MSB
  ldi ZL,Low(sMantEnd) ; dto., LSB
  ldi rCnt,MantLength ; Over the complete length
  clc ; No carry on start
Multiply2a:
  ld rmp,-Z ; Read last digit

```

```

rol rmp ; Multiply by 2 and add carry
st Z,rmp ; Overwrite last digit
subi rmp,10 ; Subtract 10
brcs Multiply2b ; Carry set, smaller than 10
st Z,rmp ; Overwrite last digit
sec ; Set carry for next higher digit
rjmp Multiply2c ; To count down
Multiply2b:
clc ; Clear carry for next higher digit
Multiply2c:
dec rCnt ; Count down
brne Multiply2a ; Further digits to process
ret

```

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

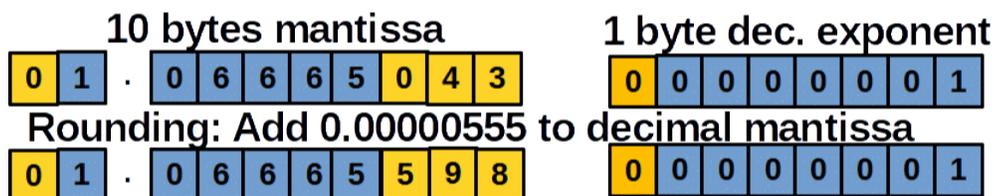
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	01	03	03	03	03	01	02	06	00	00	FF	FF	FF	FF	FF	FF

This is the simulated multiplication by 2. Note that the overflow byte is at one now, so following each multiplication a check whether another normalization has to be performed. If so, a right-shift is performed to normalize the decimal mantissa again.

The same happens if the binary exponent is negative (bit 7 = one). In that case the mantissa has to be divided by two and the normalization check should repair any losses of the first digit, by shifting the mantissa one or more positions to the left.

These steps are repeated until the binary exponent reaches zero.

Rounding the decimal mantissa



We reserved the three last (insignificant) digits for the repeated shifting in the previous phase, but now we use them for rounding the result. To do that we add 0.0000555 to our interim result. This should round these three digits sufficiently.

```

; Round the mantissa up
RoundUp:
ldi ZH,High(sMantEnd)
ldi ZL,Low(sMantEnd)
ldi rmp2,5
ldi rCnt,3
clc
RoundUp1:
ld rmp,-Z
adc rmp,rmp2
st Z,rmp
subi rmp,10
brcs RoundUp2
st Z,rmp
sec
rjmp RoundUp3
RoundUp2:
clc
RoundUp3:
dec rCnt
brne RoundUp1
ldi rmp2,0
ldi rCnt,MantLength-3
RoundUp4:
ld rmp,-Z
adc rmp,rmp2
st Z,rmp
subi rmp,10
brcs RoundUpRet

```

```

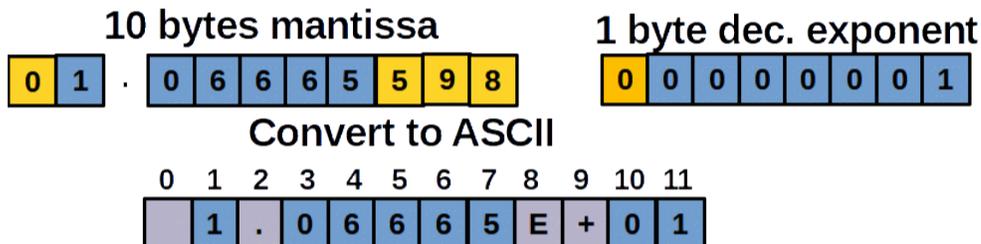
dec rCnt
brne RoundUp4
rcall Normalize
RoundUpRet:
ret

```

Note that, under rare circumstances, rounding can lead to an overflow even to the byte 0. Therefore we finally have to check if an additional normalization is necessary. This is not the case if the up-rounding chain ends already in a lower byte position.

With that we have our float now complete for its conversion to ASCII format.

Conversion from BCD to ASCII



All numbers in our decimal are BCDs. We have to add 0x30 (or subtract -'0') to get ASCII characters. Of course we'll have to add

1. the sign of the decimal mantissa, if it is negative (a blank if otherwise),
2. the decimal dot,
3. if the decimal exponent is not zero, we'll have to add **E**, the sign of the decimal exponent, and the exponent in two-digit format. If not we add four blanks.

Execution times

If you are short in time, because your AVR has more urgent things to do than converting floats to decimals: here are the execution times.

The complete procedure needs roughly the following times:

Mantissa	Exponent	Duration
0x4000	0x00	448 µs
	0x01	668 µs
	0x02	816 µs
	0x10	2.15 ms
	0x7F	23.2 ms
0xFFFF	0x00	449 µs
0x5555		2.88 ms
0x7FFF		3.87 ms

The cases with negative mantissas or exponents are not differing much from the positive cases as there are only two additional instructions (a NEG and a COM).

If you need the assembler source code (419 lines) for own experiments or extensions to 32/40/48/56/64 bit floats, here is it: [float_conv16.asm](#).

Faster than above: converting a 40-bit-binary to decimal

This above was not very effective because we used lots of slow SRAM and used a whole byte per decimal digit. The following shows the more effective way to do conversion of a 40-bit-binary, consisting of 32 bits mantissa and 8 bits exponent, to a decimal. With the above method this would last at least 50 ms, so we need a faster method for this.

We do that in the following way:

1. It first converts the 32-bit mantissa to an integer value. Because a 32 bit binary can hold decimal numbers of up to 4 billion and hence with 10 digits accuracy, we need an integer that can hold up to five bytes, but as we will have some overflow during multiplications, we use six bytes. For each of the 32 mantissa bits, the decimal representation of the weight of this bit is added to the result. Again, like demonstrated above, we start with 0.5, which is decimal 50.000.000.000 or hexadecimal 0B,A4,3B,74,00. These five bytes are repeatedly divided by two to get the next bit's weight factor as decimal. If the mantissa bit is one, the decimal is added to the result in rAdd5:4:3:2:1:0.
2. The integer is then multiplied with the exponent: each positive exponent multiplies the integer by

two. If the left-shift shifts a one to byte 6 in rAdd5, the number is divided by 10 and the decimal exponent is increased by one. If the exponent is negative, the number in rAdd5:4:3:2:1:0 is divided by two. If rAdd4 gets empty by shifting, the number is multiplied by 10 (*4 by two left shifts, *5 plus original value, *10 by an additional left shift and the decimal exponent is decreased by one.

3. If follows normalization of the decimal integer: If byte 6 in rAdd5 is not zero, the number is divided by 10 and the decimal exponent is increased. If the number in rAdd4:3:2:1:0 is larger than or equal to 1.000.000.000.000 or 0xE8.D4.A5.10.00 (the maximum integer that the following integer-to-decimal conversion can handle), the number is also divided by 10 and the decimal exponent is increased. In case that the number did not exceed the maximum, it is checked whether it is smaller than 100.000.000.000 or 0x17.48.76.E8.00. If that is the case, the number is multiplied by 10 and the decimal exponent is decreased. That ensures that the first digit is at least a one (normalization of the decimal).
4. The integer is then converted to a decimal value. This is done by subtracting 100.000.000.000 repeatedly from the integer until an underflow occurs. This leads to the first digit and the decimal subtractor is added again. The decimal dot then follows. The following digits are derived by repeatedly subtracting the next lower decade, and down until 10. The last digit is the rest of the number.

Note that dividing a 6-byte integer by 10 requires shifting the 48 bits bitwise to the left into another register. If that gets larger or equal 10, a one is shifted into the result, if not a zero is shifted. The division routine is a bit lengthy and consumes lots of execution time. As this routine is repeatedly executed if large positive binary exponents have to be processed, their time consumption is higher than for all other cases. But: compare these times with the ones above and consider that we have doubled the mantissa bits (from 16 to 32).

Binary		Decimal	Execution time
Mantissa	Exponent	Result	at 1MHz, ms
0x40000000	0x00	5.000000000000E-01	0.800
0x40000000	0x01	6.2500004800E-02	1.678
0x40000000	0x7F	8.50705917304E+37	40.145
0x40000000	0xFF	1.56250001200E-02	1.639
0x40000000	0x80	1.4936794650E-39	6.810
0x7FFFFFFF	0x00	9.9999999963E-01	2.239
0x7FFFFFFF	0x7F	1.701411833989E+38	41.840
0xBFFFFFFF	0x00	-5.00000000240E-01	1.480
0xBFFFFFFF	0x01	-1.00000000048	1.396
0x80000000	0x00	-9.99999999870E-01	2.310
0xBFFFFFFF	0x7F	-8.50705917716E+37	40.832
0xBFFFFFFF	0x80	-1.46936794650E-39	6.808

The table on the right shows the results for various input combinations.

The source code in assembler format can be downloaded from [here](#). If you like to use it for serious applications: add another byte to the right to get increased accuracy and reduce the Div10 routine from 48 down to 40 bits in cases where no overflow is in rAdd5 to increase the execution speed.

Conclusion

Keep away from those fractional numbers. They eat your performance and blow up your code with, in most cases, completely unneeded trash.

Floating point arithmetic in assembly language

Converting decimals to binary floating point numbers in assembler language

Following the introduction to binary floating point numbers and the conversion of binary floats to decimal format we need the opposite of the last: the conversion of decimals to float. And that goes like this.

Decimal number formats

There are lots of different decimal number formats:

- **1** or **123**: decimal fixed integers without decimal dot,
- **12.3**: decimal floating point numbers with decimal dots,
- **-1.234**: negative decimal floating point numbers with decimal dot,
- **1.2345E2**, **1.2345E+2**, **1.2345E+02**: decimal floating point numbers with one or two decimal exponents, and with or without "+",
- **1.23456E-12**: decimal floating point numbers with negative decimal exponents,
- **-1.234567E-13**: negative decimal floating point numbers with negative decimal exponent.

To convert all these formats of decimals to binary floating point numbers, the software has to:

1. check whether the decimal is negative (the string starts with "-"),
2. convert the unsigned decimal mantissa to a binary number format,
3. get the decimal exponent (number of decimal digits before the decimal dot plus the number of digits after the decimal dot plus the negative or positive number following "", if any) and to multiply (positive exponent) or divide (negative exponent) the binary mantissa, including any changes to the binary exponent, if necessary,
4. normalize the binary mantissa (highest mantissa bit = 1), and to
5. invert the mantissa's sign bit and it's content if the input number is negative.

The assembler software for the conversion

The assembler source code [here](#) has the decimal number to be converted to a binary float as a string with ASCII-formatted characters in its flash memory, together with the conversion code. This string is copied to a location in SRAM first. This step is necessary to avoid multiple mixed accesses to the flash later on. If your decimal is already located in the SRAM (e.g. because you received it via serial communication) you can skip this step. Only ensure that the decimal ends with a null byte at the end (null-terminated string).

The software is written for binary mantissas of up to 40 bit length and uses an 8-bit binary exponent. That corresponds to a 48-bit binary float. Those who need less accuracy can remove the last or the two last bytes and save some execution time with that.

Detecting the negative sign

By default a positive sign is assumed. The pointer X (XH:XL = R27:R26) points to the beginning of the string and reads the first ASCII character. If that is a minus character, the flag **bMneg** is set. The procedure later on uses this flag to format the mantissa as negative.

Read the decimal mantissa and convert it to a binary integer

The software then starts reading the decimal mantissa. This is written for english format (decimal dot), but this is ignored in this stage. The null terminator 0x00 or a "E" character ends reading the decimal mantissa. Other characters than ASCII-Zero to ASCII-Nine lead to a jump to the error loop routine. In that case register R16 holds an ASCII character that characterizes the reason for the failure:

- "0": Character smaller than ASCII-zero,
- "9": Character larger than ASCII-nine,
- "E": Exponent larger than +/-39,
- "b": Binary exponent smaller than -128,
- "B": Binary exponent larger than 127.

The digits read are, starting with 10,000,000,000 (0x02540BE400), multiplied (by repeated addition to the result) and added. The next digit reads the next lower decimal as hex. This is repeated until either the string ends or an "E" occurs or the decimal reaches zero (all other characters following are ignored). To read the decimals from a table in flash memory avoids to divide the number by 10 and accelerates execution.

Calculate the binary mantissa

To convert the binary integer that was read in to a binary mantissa, all mantissa bits are first cleared. Only the least significant bit in the mantissa is set, which signals that all 40 bits have been converted.

Starting with the decimal 1,000,000,000,000 (0xE8D4A51000) this decimal is repeatedly divided by two. The integer is then compared with this divided number. If the integer is equal or larger than the divided decimal, the decimal is subtracted and a one is shifted into the result registers. If not, a zero is shifted into the result registers. If, after shifting, a zero is shifted out the division by two and the comparison is repeated. If a one is shifted out, the conversion of 40 bits is complete.

Determine the decimal exponent and convert it

Now the decimal exponent is determined. First the position of the decimal dot is searched for in the string: any digit left to the dot increases the decimal exponent. Then the "E" is searched for. If the string ends without this character, the decimal exponent is already correct. If not, the maximal two decimal digits are read, converted to a binary byte and this is either added (if the exponent sign is missing or "+" or is subtracted, if negative (flag **bEneg** is set).

Now the decimal exponent is checked if it is larger than 40 or smaller than -40. If so, the error loop is executed.

If the decimal exponent is positive (bit 7 of **rDExp** is clear), the mantissa is multiplied by 10. This is performed in a subroutine named **Mult10:**. To do this it is first checked if the most significant byte of the mantissa (**rR4**) reaches or exceeds 25. If that is the case, the mantissa is shifted right and the binary exponent is increased. The mantissa is then copied, rotated to the left two times, then the copy is added and another shift left is performed. This multiplication is repeated as often as the decimal exponent says.

If the decimal exponent is negative, the mantissa is divided by 10, as often as the decimal exponent says. Division by 10 can be done in two ways, both are included in the source code following the label **Div10:**. Just change the respective switch either to zero or one. The first type of division by 10 is to shift out the 40 bits one by one and to subtract 10 from the shifted-out bits. If no carry occurs, a one is shifted into the result registers. If a carry occurred, the subtraction of the 10 is undone and a zero is shifted in.

The second version of dividing by 10 needs a little bit more source lines, but performs faster. The accelerated mode copies the previous mantissa, adds five to the mantissa, then the copy is repeatedly divided by two. The first, the second and the third divided copies are subtracted from the uprounded mantissa, the fourth and fifth are not subtracted. Then the following two divided copies are subtracted and the next following two are not subtracted. The dividing and subtracting ends when the divided copy is empty.

To switch to the accelerated div10 version is useful if many divisions by 10 have to be performed (in case of a negative decimal exponent). In case of 1E-30 the classical div10 method needs 24.55 ms, the accelerated method only 14 ms, and so is nearly double as fast.

The accelerated method is further described [here](#) for divisions by N and [here](#) for 10 only.

Normalization and sign processing

Finally the binary mantissa is normalized. It is either shifted to the right (if bit 39 is set) or is shifted left as long as bit 38 of the mantissa is not one. Of course shifting decreases or increases the binary exponent accordingly.

Those who need normalization with an extra mantissa bit, shift the mantissa left until bit 39 is one and then clear bit 39. This shifts the most significant bit out and adds one bit to the mantissa.

Finally: if the flag **rDneg** is set, the complete mantissa, including the sign bit 39, is inverted.

After all these operations the result binary mantissa is in **rR4:rR3:rR2:rR1:rR0**, the binary exponent in **rBExp** and all is completed.

Results

The table to the right shows results of such conversions from decimal to binary for selected cases. Displayed is the decimal number, its binary mantissa and exponent, the result of the conversion of the back to decimal format, as well as the execution times needed. In all cases the accelerated DIV10 method has been switched on.

Decimal	Converted to		Decimal Effective	Time ms @1MHz
	Mantissa	Exponent		
„1“	0x40000000B5	0x02	1.000007630	1.198
„-1“	0xBFFFFFFF4B	0x02	-1.000007630	1.234
„12“	0x60000000EF	0x05	12.00006104	1.361
„123“	0x7B000000CA	0x08	123.0007324	1.526
„1.23“	0x4EB851EC0B	0x02	1.230064850	1.364
„.12345678901“	0x7E6B74DE18	0xFE	0.123456551	2.076
„1.2345678901“	0x4F03290ACF	0x02	1.234510670	2.122
„1.2345678901E+01“	0x62C3F34D7F	0x05	12.34564839	2.291
„1.23456678901E+30“	0x7CA8D4FDBC	0x65	1.2346090E+30	5.016
„1.2345678901E-30“	0x6428F8D0D8	0x9E	1.2346130E-30	14.228

As can be seen, the re-converted numbers differ in the fifth or sixth decimal digit. So is 0.12345651 incorrect for the second "5" in the seventh digit, which would round up falsely to the sixth digit. This is as expected because with 40 bits LOG2(40) is little more than 5. If you need it more accurate, use a 56 or 64 bit mantissa instead. The method is the same, the extensions are pretty small.

Conclusion

Those who want to send the controller into deep bit shifting and away from the relevant things that also can happen in the controller's life, use floating point conversion. Conversion of a decimal to a 40-bit float can replace delay routines of around 10 ms. If you need maximum delay, then let your controller convert 1E-36 and switch the accelerated method off.

Have much fun with playing with this software.

Address modes in AVR

Here you find all about accessing locations in assembler in AVR.

Accessing SRAM, registers and port registers

The first thing to learn when accessing memory types in AVR is that there are two types of addressing:

1. Physical addresses, and
2. Pointer addresses.

Both are in most cases not identical. In case of SRAM, the two address types have the following values (see the diagram):

- The physical address of the SRAM starts with 0x0000. If the device has 1 kB SRAM, its physical address ends at 0x03FF.
- The pointer address starts at an address SRAM_START, which is defined in a constant in the def.inc-file. In devices that have no extended port registers this is the address 0x0060, in other cases 0x0100 or even higher.

Note that when accessing SRAM you'll never use the physical address of this memory space, only and exclusively the pointer address. This is a major difference to accessing port registers, where you can use both address types.

Consequently, if you switch the assembler to the data segment using the **.dseg** directive, its address pointer starts at SRAM_START (in many cases 0x0060, in other cases beyond that address).

How can you find out that address? Now, either you search the def.inc file for that address or, more convenient, you use [gavrasm](#) as assembler or use the [avr_sim](#) simulator. Place the following lines into your source code:

```
.nolist
#include "m324Pbdef.inc" ; Define device ATmega324PB
.list
.dseg
TheFirstSramLocation:
.cseg
```

After assembling with gavrasm, with the -s option active, or with avr_sim you'll see in the symbol table within the lower section of the listing that the label placed behind the .dseg directive: the pointer address of the first SRAM location is in that case 0x0100. If you would have selected an ATtiny13 (with .include "tn13def.inc" in the second line), you would get a different value.

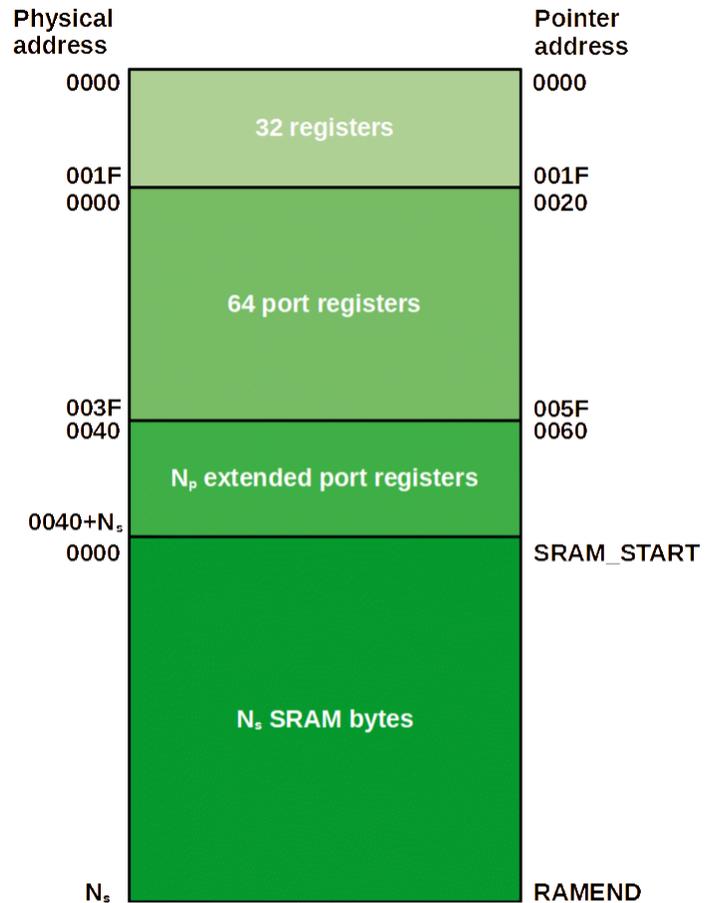
List of symbols:				Decimalval	Hexval	Name
Type	nDef	nUsed				
T	1	1		165	A5	ATMEGA324PB
L	1	0		256	0100	THEFIRSTSRAMLOCATION
L	1	0		0	00	MAIN
L	1	2		0	00	LOOP
C	1	0		21	15	NOW_Y
C	1	0		9	09	NOW_M
C	1	0		15	0F	NOW_D
C	1	0		44454	ADA6	NOW_I

If you'll need the constant **RAMEND**: just use **.equ my_ramend = RAMEND** to get the value of RAMEND into the symbol list. To remove the directive **.NOLIST** from the source code is not recommended as it flows your attention with hundreds of information lines that you are not really interested in.

If you are working with avr_sim, you can also use the feature "View" and "Symbols" and filter the list with the term "RAMEND".

Accessing SRAM locations with fixed addresses

SRAM can have a physical size of up to 32,768 bytes. As the pointer address is always higher, the



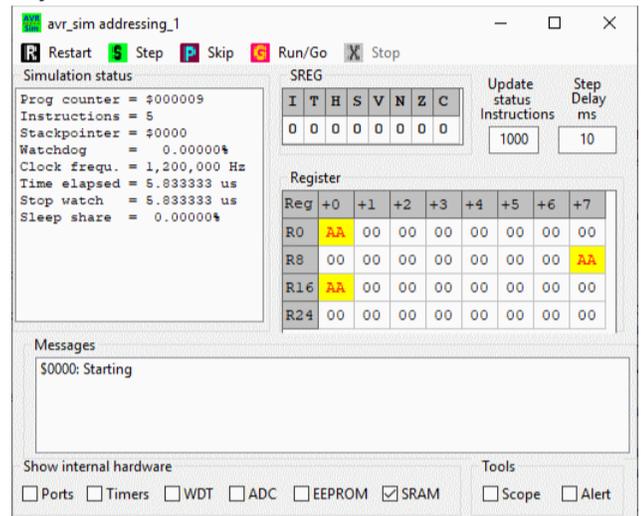
addresses to be handled are therefore 16 bits wide (and not just 15 bits).

Such locations can be addressed directly using the instructions **STS 16-bit-address,register** or **LDS register,16-bit-address**. **Register** can be any register between R0 and R31. The **16-bit-address** can be any 16-bit wide fixed number.

The following code writes 0xAA (or binary 0b10101010) to the first physical SRAM location. This is also written to the 15th byte in SRAM. To demonstrate the usefulness of the pointer addressing, we also write this byte to registers R0 and R15:

```
.dseg
FirstSramLocation: ; Place a label to this address
;
.cseg
ldi R16,0xAA
sts FirstSramLocation,R16 ; Write to first SRAM location
sts FirstSramLocation+15,R16 ; Write to SRAM 15 bytes later
sts 0,R16 ; Write to register R0
sts 15,R16 ; Write to register R15
```

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



To the left, the two bytes written to SRAM can be seen. To the right the two bytes written to those registers can be seen.

If we would have to read from these locations we would use the following:

```
lds R16,FirstSramLocation ; Read from first SRAM location
lds R17,FirstSramLocation+15 ; Read from SRAM 15 bytes later
lds R18,0 ; Read from register R0
lds R19,15 ; Read from register R15
```

If you want your controller to waste some time: lds R16,16 or sts 16,R16 are wonderful operations.

Note that all four locations use fixed addresses. Those addresses are added to the STS instruction word 0x9300, as can be seen from the assembler listing, so that a double-word instruction is resulting.

```
22: 000000 EA0A ldi R16,0xAA
23: 000001 9300 sts FirstSramLocation,R16 ; Write to first SRAM location
    000002 0060
24: 000003 9300 sts FirstSramLocation+15,R16 ; Write to SRAM 15 bytes later
    000004 006F
25: 000005 9300 sts 0,R16 ; Write to register R0
    000006 0000
26: 000007 9300 sts 15,R16 ; Write to register R15
    000008 000F
```

Accessing SRAM location with pointers

To access areas of locations we'll need to address dynamically, in registers. AVR's can handle 16 bit wide addresses in three double registers or so-called register pairs):

- X = XH:XL = R27:R26,
- Y = YH:YL = R29:R28,
- Z = ZH:ZL = R31:R30.



To point double register X to the first SRAM location we use the two following instructions:

```
.dseg
FirstSramLocation: ; Place a label to this address
.cseg
ldi XH,High(FirstSramLocation) ; Set the MSB of the address
```

```
ldi XL,Low(FirstSramLocation) ; Set the LSB of the address
```

The address is now in X. To write 0xAA to this address we add the following:

```
ldi R16,0xAA ; Write AA to register R16
st X,R16 ; and to the first SRAM location
```

Now, that is not very advanced. It is still only one byte to write. But we'll see how we can use the pointer for more.

Accessing SRAM location with increasing pointers

Now we can easily fill the first 16 bytes of SRAM with the 0xAA by using a loop that increases that address:

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

```
.dseg
FirstSramLocation: ; Place a label to this address
.cseg
ldi XH,High(FirstSramLocation) ; Set the MSB of the address
ldi XL,Low(FirstSramLocation) ; Set the LSB of the address
ldi R16,0xAA ; Write AA to register R16
FillLoop:
st X+,R16 ; and to the SRAM location and increase the address
cpi XL,Low(FirstSramLocation+16) ; Check if end of fill area
brne FillLoop
```



This shows the initiation stage:

- The pointer register X (in R27:R26) has been set to the address of the first SRAM location. MSB and LSB are set.
- The register R16 is set to 0xAA.



The first step has been executed, **ST** has written the content of register R16 to the first SRAM location. The plus behind X auto-increases the address right after writing the register to the location in X. It replaces two instructions:

1. **ST X,R16**, plus
2. **ADIW XL,1**,

but consumes only two clock cycles instead of four. That is called Auto-Increment.

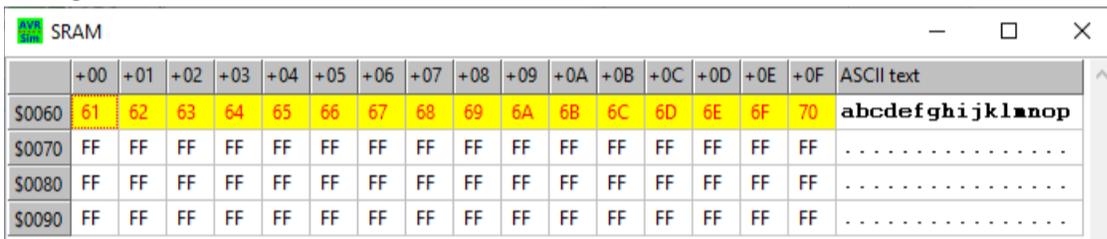
The last step is to check whether X already points to outside of our row. The first location outside our row is **LastLocationPlus1**. Note that this only works for area lengths of up to 256 bytes, because we check only the LSB of the address byte. An alternative way, to be able to fill any desired length of SRAM with that constant, would be:

```
.dseg
FirstSramLocation: ; Place a label to this address
.byte 16 ; Define length of area
LastSramLocationPlus1:
;
.cseg
ldi XH,High(FirstSramLocation) ; Set the MSB of the address
ldi XL,Low(FirstSramLocation) ; Set the LSB of the address
ldi R16,0xAA ; Write AA to register R16
FillLoop:
st X+,R16 ; and to the SRAM location and increase the address
cpi XH,High(LastSramLocationPlus1) ; Check MSB
brne FillLoop
cpi XL,Low(LastSramLocationPlus1) ; Check if end of fill area
brne FillLoop
```

Now: whatever length is defined in the SRAM segment, we'll write our constant to that complete area.

Accessing SRAM location with decreasing pointers

Increasing pointers and repeated read/write allow very fast and effective programs. But what if we need decreasing?

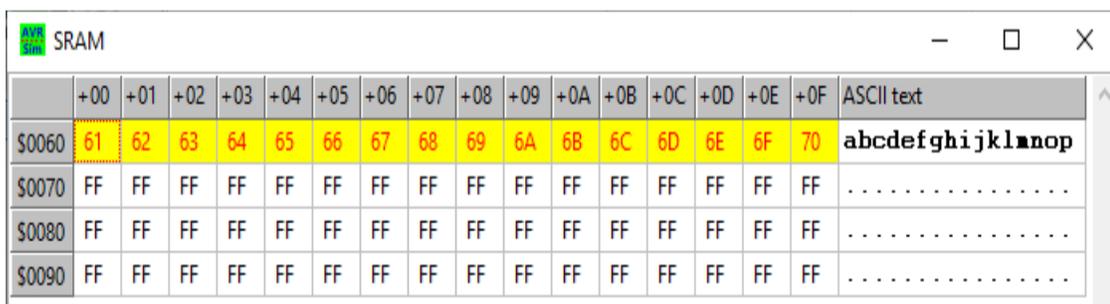


	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	abcdefghijklmnop
\$0070	FF															
\$0080	FF															
\$0090	FF															

As a somewhat weird example: we want to copy an SRAM area to a different area in a reversed row, so that the text in one area appears reversed in another area.

First we have to create a text pattern in a first area. Like this:

```
.dseg
Textarea:
.byte 16
TextareaEnd:
.cseg
ldi XH,High(Textarea)
ldi XL,Low(Textarea)
ldi R16,'a'
FillLoop:
st X+,R16
inc R16
cpi XL,Low(TextareaEnd)
brne FillLoop
```



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	abcdefghijklmnop
\$0070	FF															
\$0080	FF															
\$0090	FF															

That produces the pattern here. In the initiation step the pointer X is set to the beginning of the text area, R16 is set to the ASCII character 'a'. In a loop then this character is written to SRAM, Y is auto-incremented and R16 is also incremented, which produces a 'b', and so on.

Now we'd like to reverse that. Of course we need a second pointer for this, e.g. Y. The first character, that the pointer X points to at the beginning, the "a", goes to the last position of the second area. The next character goes one position left to that, so we have to decrease the second pointer. If you think, that possibly an Y- would be sufficient to avoid a pointer decrease with **SBIW YL,1**, you are on a good assumption, but the assembler complains: **ST Y-,R16** is not a valid instruction:

This is the error message of the assembler when trying to use **ST X-,R16**: the minus is valid on the left

```
Error 027: ST instruction can only use -XYZ+, not Z-
File: C:\Users\gerd\Documents\1_dev\AVR\addressing_x1.asm, Line: 25
Source line: st Z-,R16
```

of Y, not to the right of it. This has a serious consequence: the minus is executed first, before storing. And: the pointer Y starts one position to the right, after the last used target byte.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	abcdefghijklmnop
\$0070	FF															
\$0080	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	ponlkjihgfedcba
\$0090	FF															

This here is the complete source code for reversed copying.

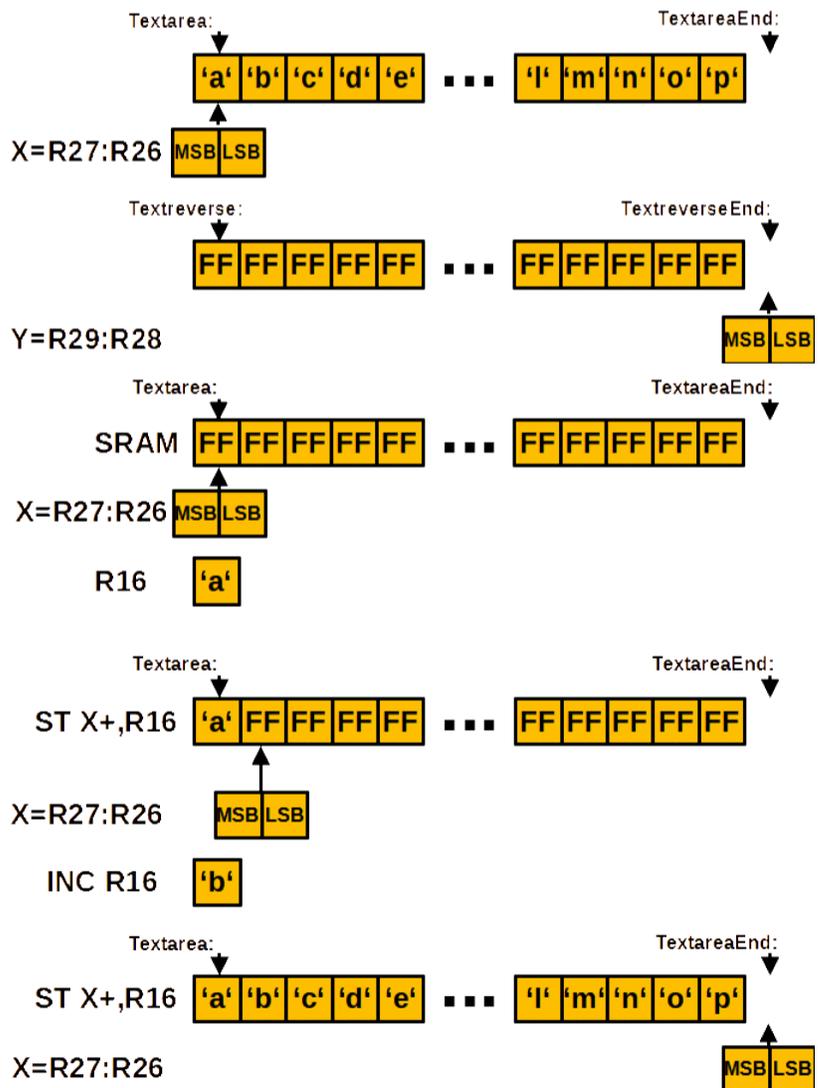
```

.dseg
Textarea:
.byte 16
TextareaEnd:
.org 0x0080 ; Leave some space in between
Textreverse:
.byte 16
TextreverseEnd:
.cseg
ldi XH,High(Textarea)
ldi XL,Low(Textarea)
ldi R16,'a'
FillLoop:
st X+,R16
inc R16
cpi XL,Low(TextareaEnd)
brne FillLoop
ldi XH,High(Textarea)
ldi XL,Low(Textarea)
ldi YH,High(TextreverseEnd)
ldi YL,Low(TextreverseEnd)
CopyLoop:
ld R16,X+
st -Y,R16
cpi XL,Low(TextareaEnd)
brne CopyLoop
    
```

The first part works like filling with a constant, but here we increase the characters in R16 from 'a' to 'p'.

The X-pointer is then set to point to the beginning of that area. Then we add pointer Y, which points at the end of the reversed text area, plus 1.

In a loop, first the next character from the text area is read. Of course, with an auto-increment.



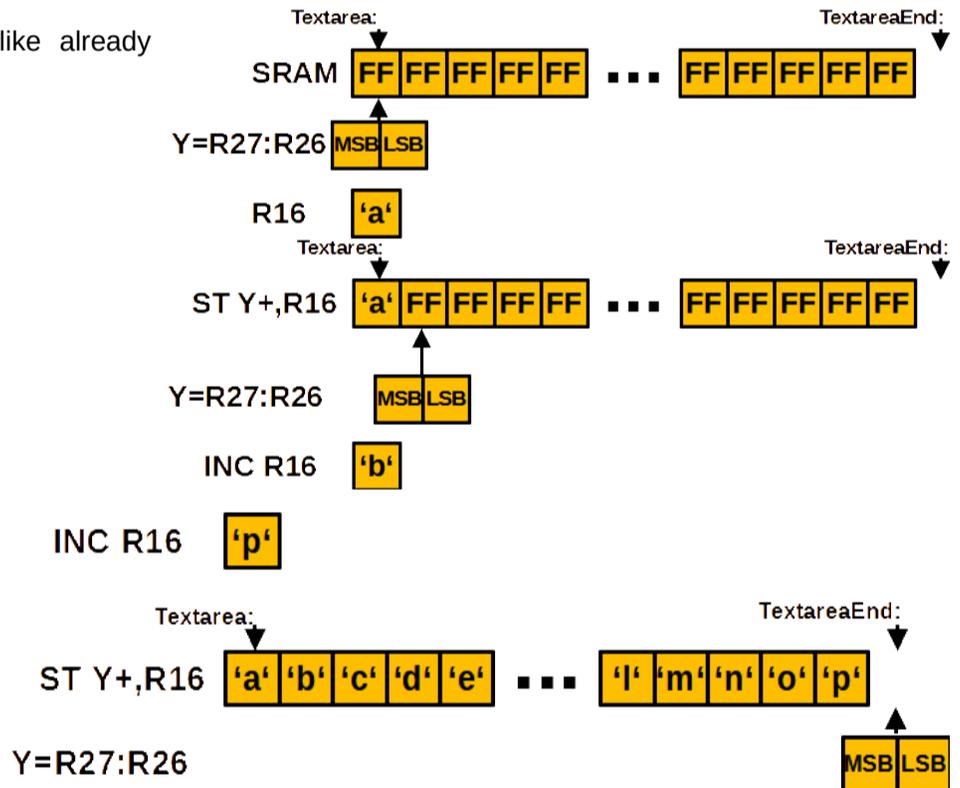
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	ABCDEFGHIJKLMN
\$0070	51	52	53	54	55	56	57	58	59	5A	FF	FF	FF	FF	FF	FF	QRSTUVWXYZ.....
\$0080	FF															
\$0090	FF															

But the displacement access can be useful in many other cases. Here a simple example. Let us assume we have filled an area with characters, such as here.

```
.dseg
TextLocation: ; Place a label to this address
.byte 27
TextLocationEnd:
.byte 1
TextLocationEndPlusOne:
;
.cseg
ldi YH,High(TextLocation) ; Set the MSB of the address
ldi YL,Low(TextLocation) ; Set the LSB of the address
ldi R16,'A' ; Write character A to register R16
FillLoop:
st Y+,R16 ; and to the SRAM location, auto-increase the address
inc R16
cpi YL,Low(TextLocationEnd) ; Check if end of fill area
brne FillLoop
```

Now let us assume further that we need space for an additional character at the start of that string, but we want to keep the original. That means we extend this text by one SRAM location and add the additional character at the beginning.

This is the filling process, like already seen in the examples above.



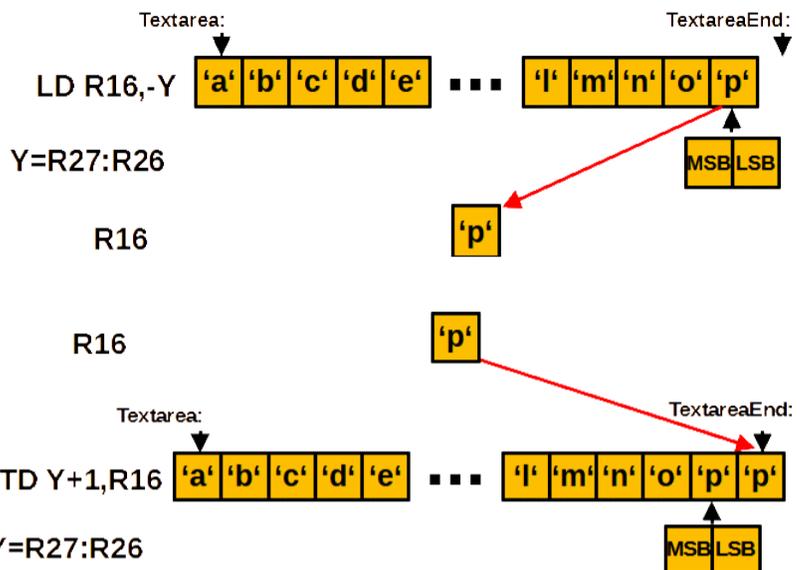
In that case it is clear that we'll have to start from the end of the string, the 'Z': only if we shift the 'Z' one position to the right, we'll have space to shift the next character, the 'Y', also one position to the right.

We see that the last fill operation already ends with the Y pointer pointing to right behind the 'Z'. We already know how to read the 'Z': just with **LD R16,-Z**. That decreases the address in Y by one and then reads the 'Z'.

But we'll have to write the 'Z' now to the next higher address. We can do that by increasing Y with **ADIW YL,1** first, then write the character and then going back with **SBIW YL,1**. As the character is by one location to the left, we can go back by two. That would lead to the following down-up-and-down-orgy:

```
sbiw YL,1 ; 2 clock cycles
Loop:
ld R16,Y ; +2 = 4 clock cycles
adiw YL,1 ; +2 = 6 clock cycles
st Y,R16 ; +2 = 8 clock cycles
sbiw YL,2 ; +2 = 10 clock cycles
; check end of loop here
brne Loop
```

A much more elegant solution uses the "Decrease before reading" feature of AVR addressing: **LD R16,-Y** first decreases the pointer and reads the byte from the already decreased address. This replaces the **SBIW** plus the **LD** and packs it into one instruction. By that it decreases execution from four clock cycles down to two.



The two steps which then increases the pointer again and writes the character there, can be replaced by another intelligent AVR address mode: **STD Y+1,R16**. This adds temporarily a one to Y and writes the character there. As the one is only temporarily added, Y remains the same after the instruction has executed. No further address manipulation is necessary.

SRAM																	
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0060	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	ⓐABCDEFGHIJKLMNO
\$0070	50	51	52	53	54	55	56	57	58	59	5A	FF	FF	FF	FF	FF	PQRSTUVWXYZ.....
\$0080	FF															
\$0090	FF															

With these two addressing tricks we come to the following optimal code for that string shifting:

```

ShiftLoop:
  ld R16,-Y ; 2 clock cycles
  std Y+1,R16 ; +2 = 4 clock cycles
  cpi YL,Low(TextLocation) ; +1 = 5 clock cycles
  brne ShiftLoop ; +1 or +2 = 6/7 clock cycles
  ldi R16,'@'
  st Y,R16
    
```

Now, that is really amazing: no pointer corrections in between: no ADIWs or SBIWs, anything in fast, two cycle long instructions. 153 µs for shifting 28 characters at 1.2 MHz clock in an ATtiny13.

And: more than double as fast than with address manipulation. So if your PIC does not know auto-decrement and displacement access, it needs double as long as an AVR. And: that makes a clock cycle increase in an ATtiny13 with the factor of two, without increasing its power consumption at all.

Conclusion: if your program goes far beyond a blink routine in respect to complexity, think about such advanced instruction capabilities. It makes your code more elegant, executes faster, is much more effective and, if associated with enough comments, it reads simpler.

Accessing port registers

Again, we have to be aware of the two address types here:

1. Physical addresses, and
2. Pointer addresses.

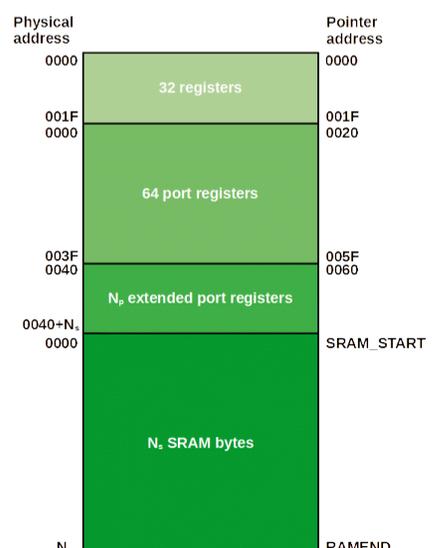
Reading from and Writing to port registers use both address types, which can confuse the beginner.

Accessing classical port registers

This type of access uses the two instructions **OUT outputport,register** to write and **IN register,inport** to read the content on those locations. The address type used here are the physical addresses.

Access with OUT and IN is limited to the 64 classical port registers. The extended port registers are not accessible with those two instructions (see below on how accessing those works).

If you need to set or clear only one of the bits in a port location, you can



use the **SBI port,bit** for setting or **CBI port,bit** for clearing the bit. These instructions replace the following instructions:

```
in R16,port ; Read the port
; Setting
sbr R16,1<<bit ; Setting the bit, or:
ori R16,1<<bit ; alternative for setting the bit in the register
; Clearing
cbr R16,1<<bit ; Clearing the bit, or:
andi R16,256-1<<bit ; alternative for clearing the bit in the register
out port,R16 ; Write to the port
```

While this (longer) method can be used on all 64 port registers, the SBI and CBI works for the lower half of the port registers only.

SBI and CBI require two clock cycles, while the alternative single step method requires three and an additional register.

Toggling a single bit in a port can be done by EXOR-ing the port with a register that has the bit(s) set that are to be toggled:

```
in R16,port ; Read the port
ldi R17,1<<bit ; The port bit to be toggled, or
ldi R17,(1<<bit1)|(1<<bit2) ; two bits to be toggled
eor R16,R17 ; Exclusive or
out port,R16 ; Write the toggled port
```

If the port to be toggled is an I/O port, you can, in most modern AVR devices, alternatively write to the I/O's input port to toggle one or more of the bits. This toggles the bits 1 and 3 of the I/O port PORTA:

```
ldi R16,(1<<1)|(1<<3)
out PINA,R16
```

Access to extended port registers

If the OUT to a port register is ending with the assembler error message that the port is out of range, this port is in the extended port register range beyond physical address 0x3F. That is the case in larger ATtiny or ATmega devices, where 64 port registers did not provide enough address space.

In that case, you'll have to use the pointer address and either the instruction STS or ST to write the data to this port. In that case the address given in the def.inc has already added the 0x20 and is a pointer address, so that you can simply replace the OUT with STS (or an IN with LDS) to that location.

Of course, the SBI/CBI and the SBIC/SBIS instructions cannot be used for these extended port registers. That is why port registers, that require to be changed with a smaller probability are placed into the extended port register area.

Access with pointers, example: the circular LED light

Now let us assume you need a 32-bit light row, where one of the 32 LEDs is pointing towards the next emergency exit. As the whole cycle has to be one second long, a frequency of 32 Hz increases the LED and each LED has to be on for 21.25 ms.

This requires a controller with four complete 8-bit-I/O-ports. The excerpt from avr_sim's device select window shows all those AVR devices. We can use one of those, such as the ATmega324PA.

This is the hardware needed. Looks pretty simple, many resistors and LEDs. If we are sure, that the software will switch only one LED of all 32 at a time on, we can reduce the number of resistors down to one and connect all cathodes with that single resistor. If the number of LEDs on is one in each 8-bit-port we can reduce the number of resistors down to four by connecting all LED cathodes in one 8-bit-port to one resistor.

The first step in software, to make all I/O direction bits output and to set PORTA's bit 0 to one and all others to zero, can be done with or without pointers. The version without pointer would be:

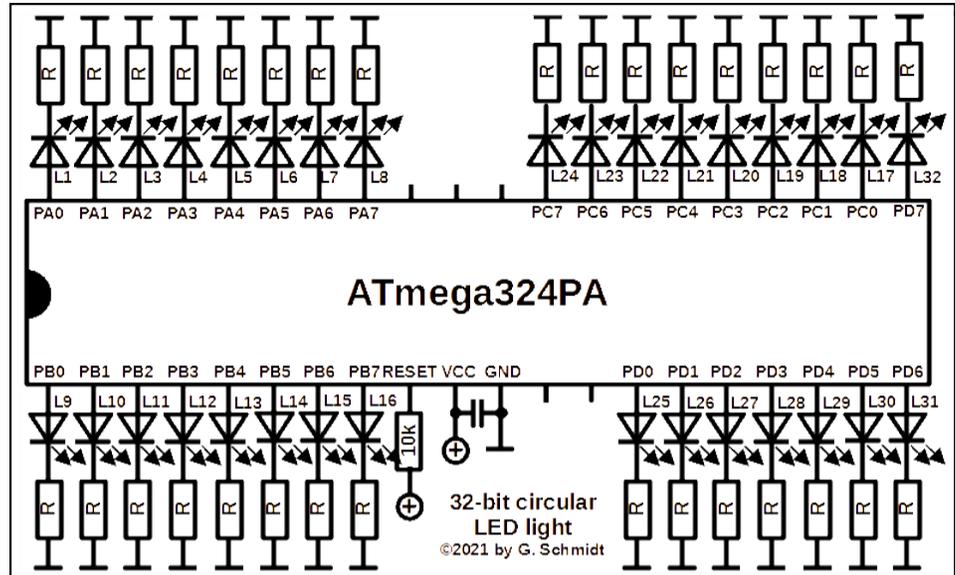
```
ldi R16,0xFF ; All bits as output
out DDRA,R16
out DDRB,R16
out DDRC,R16
out DDRD,R16
```

23 devices out of 478

```
==> ATmega <==
ATmega16 - PDIP40
ATmega16A - PDIP40
ATmega32 - PDIP40
ATmega32A - PDIP40
ATmega83 - PDIP40
ATmega161 - PDIP40
ATmega162 - PDIP40
ATmega163 - PDIP40
ATmega164A - PDIP40
ATmega164P - PDIP40
ATmega164PA - PDIP40
ATmega323 - PDIP40
ATmega324A - PDIP40
ATmega324P - PDIP40
ATmega324PA - PDIP40
ATmega644 - PDIP40
ATmega644A - PDIP40
ATmega644P - PDIP40
ATmega644PA - PDIP40
ATmega1284 - PDIP40
ATmega1284P - PDIP40
ATmega8515 - PDIP40
ATmega8535 - PDIP40
```

```

clr R16 ; All upper
24 bits clear
out PORTD,R16
out PORTC,R16
out PORTB,R16
ldi R16,0x01 ;
Lowest bit set
out PORTA,R16
    
```



These are the register ports controlling I/O ports in an ATmega324PA. All addresses form a row of three register ports: PIN, DDR and PORT. The physical as well as the pointer addresses increase by one. So, with a pointer base address of 0x20 in Y or Z, those register ports can be accessed with displacements of 0, 3, 6 and 9 for the PIN of the I/O port, with 1, 4 and 7 the DDR I/O ports are displaced and with 2, 5 and 8 the PORT I/O ports are displaced. If you wonder what is meant with the term "displacement", see the chapter on displacement in SRAM.

	Symbol	Physical address	Pointer address								
I/O port A	PINA	0x00	0x20	7	6	5	4	3	2	1	0
	DDRA	0x01	0x21	7	6	5	4	3	2	1	0
	PORTA	0x02	0x22	7	6	5	4	3	2	1	0
I/O port B	PINB	0x03	0x23	7	6	5	4	3	2	1	0
	DDRB	0x04	0x24	7	6	5	4	3	2	1	0
	PORTB	0x05	0x25	7	6	5	4	3	2	1	0
I/O port C	PINC	0x06	0x26	7	6	5	4	3	2	1	0
	DDRC	0x07	0x27	7	6	5	4	3	2	1	0
	PORTC	0x08	0x28	7	6	5	4	3	2	1	0
I/O port D	PIND	0x09	0x29	7	6	5	4	3	2	1	0
	DDRD	0x0A	0x2A	7	6	5	4	3	2	1	0
	PORTD	0x0B	0x2B	7	6	5	4	3	2	1	0

The version with a pointer would use these displacements and would look like this:

```

ldi R16,0xFF ; All bits as output
ldi YH,High(PINA+0x20) ; Point Y to PINA's pointer address
ldi YL,Low(PINA+0x20)
std Y+1,R16 ; Access the DDR port registers
std Y+4,R16
std Y+7,R16
std Y+10,R16
clr R16 ; The upper 24 bits clear
std Y+5,R16 ; Access the PORT port registers
std Y+8,R16
std Y+11,R16
ldi R16,0x01 ; The lowest bit set
std Y+2,R16
    
```

Now, that is clearly less efficient, because each STD access consumes two clock cycles instead of one for an OUT. So we would prefer the classical OUT method over the pointer method in that case.

But now, let's write an interrupt service routine for the compare match interrupt of a timer that controls the speed of our LED row. Quick and dirty this would be like:

```

Tc0CmpIsr: ; 7 clock cycles for int+rjmp
in R16,PORTA ; +1 = 8
lsl R16 ; +1 = 9
out PORTA,R16 ; +1 = 10
in R16,PORTB ; +1 = 11
rol R16 ; +1 = 12
out PORTB,R16 ; +1 = 13
in R16,PORTC ; +1 = 14
rol R16 ; +1 = 15
out PORTC,R16 ; +1 = 16
in R16,PORTD ; +1 = 17
rol R16 ; +1 = 18
out PORTD,R16 ; +1 = 19
brcc Tc0CmpIsrReti ; +1/2 = 20/21
in R16,PORTA ; +1 = 21
    
```

```

rol R16 ; +1 = 22
out PORTA,R16 ; +1 = 23
Tc0CmpIsrReti: ; 21/23 cycles
ret ; +4 = 25/27 cycles
; Total cycles: 31 * 25 + 27 = 802 cycles

```

The given clock cycles are for each interrupt. 31 times the interrupt does not require the restart, consuming 25 clock cycles each, once a restart is necessary and consumes 27 cycles. This sums up to 802 cycles in total. Together with 32 wake-ups from sleep and 64 cycles for jumping back to sleep, we are approximately at 866 cycles in one second. At 1 MHz clock, that makes a sleep time share of 99.13%.

But: Three of the four OUT instructions are superfluous, because the active LED is not in that I/O port. So writing only the one port with the currently active LED would be sufficient.

In those rare cases, where the active LED changes the I/O port (every eight's shift), both the old port as well as the next port has to be written. That brings us to a different algorithm, this time with pointers. First the initialization:

```

ldi XH,High(PORTA+0x20) ; Pointer address to X, +1 = 1
ldi XL,Low(PORTA+0x20) ; dto. LSB, +1 = 2
ldi rShift,0x01 ; Start shift register, +1 = 3
st X,rShift ; Write this to PORTA, +2 = 5

```

As these five cycles are to be performed only once during init, we don't really count them. And the interrupt service routine with the moving pointer would be like this:

```

0C0AIsr: ; 7 cycles for int+rjmp
lsl rShift ; +1 = 8
st X,rShift ; +2 = 9
brcc Tc0CmpIsrReti ; +1/2 = 10/11
; Next I/O port
adiw XL,3 ; Point to next channel, +2 = 12
cpi XL,Low(PORTD+3+0x20) ; +1 = 13
brne Tc0CmpIsr1 ; +1/2 = 14/15
ldi XH,High(PORTA+0x20) ; +1 = 16
ldi XL,Low(PORTA+0x20) ; +1 = 17
Tc0CmpIsr1: ; 15/17 cycles
; Restart from the beginning
ldi rShift,0x01 ; Set bit 0, +1 = 16/18
st X,rShift ; +2 = 18/20
Tc0CmpIsrReti: ; 11/18/20
ret ; +4 = 15/22/24
; Total cycles: = 28*15 + 3*22 + 1*24 = 510

```

The number of cycles is slightly more than half of the classical method without pointers. This is a clear indicator that the method with the moving pointer is nearly double as efficient, simply because it doesn't waste time on unnecessary INs and OUTs. That increases the sleep share to 99.43% and also reduces the current consumption of the controller, so that the emergency power supply lasts longer.

Now consider that we connect the LED's cathodes to the I/O pins and the resistor(s) to plus. There are only a few changes that have to be made in the software: **CLR rShift** turns to **SER rShift**, **LDI rShift,0x01** turns to **LDI rShift,0xFE** and **brcc Tc0CmplsrReti** turns to **brcs Tc0CmplsrReti**. These changes concern the initiation of the I/O ports as well as the interrupt service routine. A little more tricky is that the **lsl rShift** in the ISR has to shift a one in, e.g. with **sec** and **rol rShift** instead of **lsl rShift**. Now the software is ready to pull-down the LEDs to GND instead of driving current into the output pins.

The assembler software for this can be downloaded from [here](#). It allows to increase or reduce the circle time between 50 and 2000 milliseconds as well as to select anodes and cathodes connected to the I/O pins (see the adjustable constants on top of the source code).

What if we want to switch more than one LED on? If you allow four LEDs to be on in each LED cycle step, the software is rather simple: just output the register **rShift** to each of the four I/O ports, without the use of pointers. If you want two LEDs to be on at a time (e.g. L1 and L16, L2 and L17, etc.), the algorithm requires pointers and is a little bit more tricky, because the restart of the two pointers, back to PORTA, happens in two different port phases. If eight LEDs shall be on in each phase, consider taking the shift state from a table in flash memory (see the chapter on addressing flash) rather than with LSL or ROL. If more than these LEDs simultaneously on, you might run into current limits of the device: the ATmega324PA can drive only 200 mA via its VCC and GND pins. So make sure that your LED currents do not exceed this limit (e.g. with 16 LEDs on - every second LED - drive those with a maximum current of 12.5 mA per LED, with all LEDs on only 6.25 mA per LED are allowed).

Conclusion:

If your program does require the same operation with ports over and over again, you should consider programming those ports with an algorithm. In many cases this is more efficient than doing all the same all over again and wasting typing and assemble time instead of a more intelligent approach.

Accessing EEPROM

All AVR's have at least 64 bytes and up to 4,096 bytes EEPROM on board. EEPROM is a memory type that keeps its content, even if the power supply of the device is down and, after a short or long period of time, is powered up again.

Un-programmed EEPROM space contains all 0xFF. The content of the EEPROM can be written in two modes:

1. When assembling source code, anything that was written in the .ESEG section of the source code, is copied into a hex file named *.eep. The file's content can be programmed into the EEPROM using a burner software.
2. Within the actively executed program, EEPROM locations can be cleared and re-written with different content. The procedure to clear and re-write a byte requires some time, the end of the write process can initiate an interrupt, if so enabled.

Reading from EEPROM requires a different procedure. Reading EEPROM content is fast and requires only a few clock cycles.

EEPROM initiation with the .ESEG directive

If you use EEPROM you might want to set the initial EEPROM content on the first controller start-up to certain values. This requires a .ESEG directive. The following places different content into the EEPROM.

```
.CSEG ; Code segment
JumpAddress:
; Instructions to be executed
;
.ESEG
.ORG 0x0000 ; The start address of the EEPROM content, default=0
.db 1,2,3,4,'A','a' ; Bytes to be written
.dw 1234, 4567 ; 16-bit Words to be written
.db "Text to be written to EEPROM",0x00 ; Text string
.dw JumpAddress
; End of the ESEG
;
.CSEG ; Code segment
; ... Further code for execution
```

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0000	01	02	03	04	41	61	D2	04	D7	11	54	65	78	74	20	74	...Aa...Text t
\$0010	6F	20	62	65	20	77	72	69	74	74	65	6E	20	74	6F	20	o be written to
\$0020	45	45	50	52	4F	4D	00	D2	04	FF	EEPROM.....						
\$0030	FF															
\$0040	FF															
\$0050	FF															

After assembling and programming the .eep file to the controller, its EEPROM looks like shown in the picture.

Normally programming the flash memory not only erases the flash but erases the EEPROM content as well. If you want to keep the EEPROM content during programming the flash, set the respective preserve fuse of the device. That prevents from erasing the flash. And: if that fuse is set, do not write the .eep file again to the EEPROM. It will fail verification in most of the cases, because zeros in the EEPROM can not be overwritten by ones (only erasing produces ones).

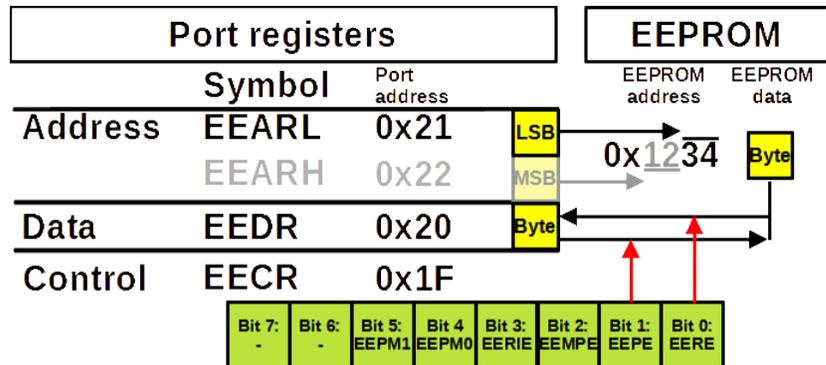
EEPROM port registers

Writing or reading EEPROM uses three or four port registers:

1. An **address**: The lowest 8 bit of the address are to be written to the port register **EEARL**. If the device has more than 256 bytes EEPROM, the most significant 8 bits of the address are to be written to port register **EEARH**.

2. A **data** port: When writing to the EEPROM the 8 bit data to be written are written to the data port register **EEDR**, before the write process is started. When reading from EEPROM, the data at the addressed location is appearing in port register **EEDR**.

3. A **control** port: Write to or read access from the EEPROM is controlled in the control register **EECR** by manipulating bits in this register. When writing, the **EEPE**-Bit is one, when reading the **EERE**-bit is one.



Please note that the addresses given are subject to changes, so always use the def.inc names instead of fixed addresses.

Writing the EEPROM address

If your program shall read or write a single byte to/from the EEPROM, the assembler code to set the address should look like this:

```
.equ EepromAddress = 0x0010
;
; First wait until any write procedure has finished
WaitEep:
  sbic EECR,EEPE ; Check EEPE byte
  rjmp WaitEep ; Wait further
;
; Set the address
ldi R16,Low(EepromAddress)
out EEARL,R16
.ifdef EEARH ; If more than 256 bytes EEPROM
  ldi R16,High(EepromAddress)
  out EEARH,R16
.endif
; Read or write procedure
ReadWriteEep:
  ; Read or write procedure here
```

Note that changing the EEPROM address ALWAYS should check first that no programming is in progress. These are the first two words, before any settings can be made.

The `.IFDEF` directive adds the MSB setting only if the symbol `EEARH` is defined in the def.inc file, which is the case for all AVR's that have more than 256 bytes EEPROM.

If you want to read or write multiple EEPROM locations in a row, you'll have to set the LSB of the address in a loop, and, if the MSB changes, the MSB as well. As the check whether programming is finished has to be made prior to entering any address changes and as the MSB has to be set in any case (if the MSB is physically available), we cannot limit the MSB output only to cases where the MSB changes. Here, we output the MSB in all cases, which are only two additional instructions.

```
.equ EepStartAddr = 0x0010
.equ EepEndAddr = 0x001F
; Set the address to the double register in R1:R0
ldi R16,High(EepStartAddr)
mov R1,R16
ldi R16,Low(EepStartAddr)
mov R0,R16
EepLoop:
  sbic EECR,EEPE
  rjmp EepLoop
; Output the LSB of the address
out EEARL,R0
.ifdef EEARH
  ; Output the MSB of the address
  out EEARH,R1
.endif
EepReadWriteProcedure:
  ; Add Read or write procedure here
```

```

inc R0 ; Increase LSB
brne EepChkEnd
inc R1
EepChkEnd:
ldi R16,Low(EepEndAddr+1)
cp R0,R16
brne EepLoop
ldi R16,High(EepEndAddr+1)
brne EepLoop

```

Reading from the EEPROM

Reading from the EEPROM is initiated by setting the **EERE**-bit in the control register **EEDR**. This halts the CPU for four clock cycles and then writes the EEPROM content at that address to port register **EEDR**. That looks like this:

```

.equ EepAddrs = 0x0010
EepWait:
sbic EECR,EEPE ; Wait until write operation finished
rjmp EepWait
ldi R16,Low(EepAddrs)
out EEARL,R16
.ifdef EEARH
ldi R16,High(EepAddrs)
out EEARH,R16
.endif
sbi EECR,1<<EERE
; Four clock cycles pause
in R16,EEDR ; Read byte to R16

```

The screenshot shows two memory windows. The top window is titled 'EEPROM view' and displays a table of memory addresses and their contents. The bottom window is titled 'SRAM' and displays a similar table. Both windows show the same data for addresses \$0000-\$0020 and \$0100-\$0120 respectively, with the ASCII text '....Aa....Text t', 'o be written to', and 'EEPROM.....'.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0000	01	02	03	04	41	61	D2	04	D7	11	54	65	78	74	20	74Aa....Text t
\$0010	6F	20	62	65	20	77	72	69	74	74	65	6E	20	74	6F	20	o be written to
\$0020	45	45	50	52	4F	4D	00	D2	04	FF	EEPROM.....						

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0100	01	02	03	04	41	61	D2	04	D7	11	54	65	78	74	20	74Aa....Text t
\$0110	6F	20	62	65	20	77	72	69	74	74	65	6E	20	74	6F	20	o be written to
\$0120	45	45	50	52	4F	4D	00	D2	04	FF	EEPROM.....						

In case we have to read more than one byte, we have to have more storage space. If a second byte is to be read, we'll need a second register. If more than three or four bytes are to be read, we use an SRAM space to write the bytes there, e.g. with a pointer to that in X, we would add the instruction **ST X+,R16**.

This here shows how we read the complete EEPROM content into an area in SRAM by using **ST X+,R16**. Note that the whole process needed close to one millisecond, because of delays during access reads, the check of the programming bit, the double byte check of the end and the pointer operations.

Write access to the EEPROM

To avoid unplanned write access to the EEPROM, the procedure to start a write process is a bit more complicated:

1. First check if the last write is finished by testing the **EEPE** bit in the control register **EECR**. Otherwise wait.
2. Then write the correct address to **EEARL/EEARH**.
3. Then write the data byte to be written to **EEDR**.
4. Then set the Master Programming Enable bit **EEMPE** in **EECR**, together with the programming mode bits **EEPM0** (Erase only) and **EEPM1** (Write only), if so desired, and the interrupt enable **EEPIE**-bit, if desired.
5. Within the following four clock cycles (make sure that no interrupt can occur during these four cycles) set the Programming Enable bit **EEPE** in **EECR**.

This starts the programming of the location after two clock cycles.

The programming of the byte lasts 3.4 ms. When finished, the **EEPE**-bit is cleared and, if so enabled, starts an interrupt and jumps to the EEPROM-READY vector.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0100	54	68	69	73	20	74	65	78	74	20	74	6F	20	62	65	20	This text to be
\$0110	77	72	69	74	74	65	6E	20	74	6F	20	45	45	50	52	4F	written to EEPRO
\$0120	4D	2E	00	FF	M.....												

An example: This text in the SRAM has to be copied to EEPROM. We can do that preferably with the interrupt feature of EEPROM write or in a discrete loop that checks whether the **EEPE**-bit is clear and writes the next character in SRAM to the EEPROM.

EEPROM view																	
D=\$77		A=\$0010		<input checked="" type="checkbox"/> MWE		<input checked="" 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	54	68	69	73	20	74	65	78	74	20	74	6F	20	62	65	20	This text to be
\$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

This is the state of the EEPROM after few characters have been written and the 16th character is currently written. In the first half of the write process the location is erased, which means that all bits at that location are set to one. In the second half, the data is written. As this lasts roughly 3.4 ms, the bar shows the progress. Both, the master write/program enable bit **EEMPE** and the write/program enable bit **EEPE** are active (high). When the write process is finished, both bits are cleared by the AVR.

SRAM																	
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0100	54	68	69	73	20	74	65	78	74	20	74	6F	20	62	65	20	This text to be
\$0110	77	72	69	74	74	65	6E	20	74	6F	20	45	45	50	52	4F	written to EEPRO
\$0120	4D	2E	00	FF	M.....												

EEPROM view																	
D=\$00		A=\$0022		<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	54	68	69	73	20	74	65	78	74	20	74	6F	20	62	65	20	This text to be
\$0010	77	72	69	74	74	65	6E	20	74	6F	20	45	45	50	52	4F	written to EEPRO
\$0020	4D	2E	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	M.....

As the whole process lasts more than 140 ms, it is not recommended to perform EEPROM writes in this lengthy way. This was just an example.

One note on interrupts: if the interrupt-enable-bit **EEIE** is set, the EEPROM READY interrupt re-triggers every time the **EEPE**-bit gets clear and if no other higher-ranking interrupt is active. Your whole program can be blocked by this, if you do not write the next byte to the EEPROM in your interrupt service routine. If no more bytes are to be written, clear the interrupt enable bit.

Finally a warning: the number of write operations to EEPROM is limited to several thousand events. One day has over 80,000 seconds, a year has more than 31 million seconds, so if you want to reach the guaranteed number of write accesses in one year, you can write the same EEPROM location every 3,100 seconds or every 53 minutes.

So do not unnecessarily re-write the EEPROM and limit write accesses to several minutes or hours and do it whenever really needed (e.g. because the user just pressed a key or because the controller has re-started). It is a good idea to hold a copy of the EEPROM content in registers or SRAM and to only re-write the EEPROM if the difference is large enough.

An example for this: if you want to keep the current state of a stepper motor in two, three or four bytes in an EEPROM location, do not write the position changes to EEPROM whenever one single step has been made. Write the status only after the complete move has finished. And write only the really changed bytes to EEPROM. And hope that the LSB of the EEPROM lasts long enough.

Conclusions:

Read accesses from EEPROM are very fast, but less fast than SRAM read accesses or accesses to registers. So better read the needed EEPROM content once to a location in SRAM and access this instead of the EEPROM.

Write accesses to EEPROM require longer times, are limited over the life-time of the device and should be limited to the cases where they are useful and really needed. Depending from the rest of your program and

from its overall timing considerations: better use the EEPROM READY interrupt to avoid timing conflicts.

Flash memory accesses

All AVR's have a memory, called flash memory, that holds the executable program. As the executable instructions in AVR's have 16 bits, the memory is 16 bit wide (in words, not in bytes).

The size of the flash memory can be anything between 512 words and up to 393,216 words. The addresses are therefore between 0x01FF and 0x05FFFF. In most cases the constant **FlashEnd** from the def.inc file provides the last or highest address.

Address	MSB								LSB							
0000	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0001	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0002	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
...																
FlashEnd	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The address 0x00000 is special because it is the starting address: after power-up, a reset or a watchdog-reset the instruction word in address 0x000000 is the first that is executed.

The .CSEG directive

Assembling a source code writes all executable instructions and tables to the code segment CSEG by default. When switching the assembler either to the SRAM segment with the directive **.DSEG** or to the EEPROM segment with the directive **.ESEG**, the return back to the code segment can be done with the directive **.CSEG**.

All code that has been assembled is written by the assembler to the .hex file. Its content can be written to the flash memory with the programmer soft- and hardware.

Instructions such as **NOP** write 16-bit words to the .hex file. Words with 16 bits can be written with the directive **.DW 16-bit-value** at any location within the flash.

Bytes can be assembled and written to the .hex file with the **.DB 8-bit-value**. When writing one single 8-bit-value, the upper significant byte MSB is always written to zero. When writing two 8-bit-constants within one **.DB 8-bit-value-1, 8-bit-value_2** directive, the first value is written to the LSB, the second value to the MSB at the current location.

```

E 20   nop ; An instruction
E 21   add R16,R16 ; Another instruction
22 Loop: ; A label
E 23   rjmp loop ; Another instruction: a jump to the label
24 ;
25 ; A table with bytes
26 ByteTable:
27   .db 1 ; One byte
28   .db 1,2 ; Two bytes
29   .db 1,2,3
30   .db "A text string"
31 ;
32 ; Table with words
33 WordTable:
34   .dw 1234 ; One word
35   .dw 1234,5678 ; Two words
36   .dw 1234,5678,9012 ; Three words
37   .dw Loop ; A location address

```

The assembler source code to the left is assembled, the results of the assembling can be viewed in the assembler listing below.

- The **NOP** in line 20 of the source code, which is a valid instruction, has been translated to an address of 0x000000 and an executable hex code of 0x0000. That hex code will be written to the hex file.

```

20: 000000 0000 nop ; An instruction
21: 000001 0F00 add R16,R16 ; Another instruction
22: Loop: ; A label
23: 000002 CFFF rjmp loop ; Another instruction: a jump to the label
24: ;
25: ; A table with bytes
26: ByteTable:
27: .db 1 ; One byte
-> Warning 004: Number of bytes on line is odd, added 00 to fit program memory!
File: C:\Users\gerd\Documents\1_dev\AVR\beginner\test_flash.asm, Line: 27
Source line: .db 1 ; One byte
000003 0001
28: .db 1,2 ; Two bytes
000004 0201
29: .db 1,2,3
-> Warning 004: Number of bytes on line is odd, added 00 to fit program memory!
File: C:\Users\gerd\Documents\1_dev\AVR\beginner\test_flash.asm, Line: 29
Source line: .db 1,2,3
000005 0201 0003
30: .db "A text string"
-> Warning 004: Number of bytes on line is odd, added 00 to fit program memory!
File: C:\Users\gerd\Documents\1_dev\AVR\beginner\test_flash.asm, Line: 30
Source line: .db "A text string"
000007 2041 6574 7478 7320
00000B 7274 6E69 0067
31: ;
32: ; Table with words
33: WordTable:
34: .dw 1234 ; One word
00000E 04D2
35: .dw 1234,5678 ; Two words
00000F 04D2 162E
36: .dw 1234,5678,9012 ; Three words
000011 04D2 162E 2334
37: .dw Loop ; A location address
000014 0002

```

- The **ADD R16,R16** in line 21 has been translated to the executable hex code 0x0F00 at address 0x000001.
- The following line has not been translated, because it is only a label and meaningful for the assembler only.
- The **RJMP loop** has been translated to 0xCFFF at address 0x000002, an executable that jumps back to where it just came from (an indefinite loop).
- The first **.DB 1** has been translated to 0x0001 at address 0x000003, but the assembler complains with a warning, that the number of bytes in the **.DB** line is odd, and that he has added a 0x00 as MSB at address 0x000003.
- The second **.DB 1,2** has been translated to 0x0201 at address 0x000004. Note that the first byte 0x01 is now the LSB of the resulting word in flash while the second byte 0x02 is the MSB of that word.
- The third **.DB 1,2,3** has been splitted into two words: 1 and 2 go to the first word, 3 goes to the second word, and the assembler warns again.
- The line **.DB "A text string"** is translated to seven words in a row: as can be seen from the second character in the string, a blank or 0x20, every second character goes to the MSB and every first character to the LSB. Again, the assembler complains that the number of bytes in the line is odd.
- No such complaints in all lines with **.DW**: all words fit into the 16 bits of the flash memory. The last entry, **.DW Loop** inserts the address of the label **Loop**: into the flash memory at that address 0x000014. We will later on read that address location to jump to such a label.

The LPM instruction

The instruction **LPM** or Load from Program Memory reads one byte from the flash. It takes the flash address from the register pair Z (ZH:ZL = R31:R30) and transfers the result to register R0.

But: each address in flash memory has two bytes, an LSB

and an MSB. Which of the two bytes are to be read, and how to get the second byte at that same address?

Flash memory

Address

0000



LPM (R0,Z)



The trick to do that is to shift the flash address left by one location and to add a zero or a one to the right of the address in bit 0 of Z. A zero to the right addresses the LSB, a one the MSB.

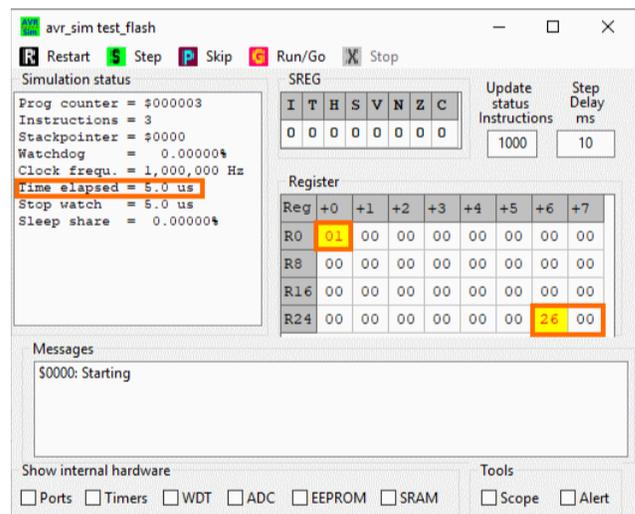
One disadvantage does the trick have: bit 15 of the flash address cannot be used. So better place your lengthy tables with thousands of values into the lower half of your 64k words wide flash.

The following formulations in assembler are all the same and set Z to access the LSB and the MSB of the byte table below:

```
.equ FlashAddr = ByteTable ; Set the flash address
; Formulation 1
ldi ZH,High(FlashAddr+FlashAddr+0) ; Access the LSB, MSB of Z
ldi ZL,Low(FlashAddr+FlashAddr+0) ; dto., LSB of Z
lpm
ldi ZH,High(FlashAddr+FlashAddr+1) ; Access the MSB, MSB of Z
ldi ZL,Low(FlashAddr+FlashAddr+1) ; dto., LSB of Z
lpm
;
; Formulation 2
ldi ZH,High(2*FlashAddr+0) ; Access the LSB, MSB of Z
ldi ZL,Low(2*FlashAddr+0) ; dto., LSB of Z
lpm
ldi ZH,High(2*FlashAddr+1) ; Access the MSB, MSB of Z
ldi ZL,Low(2*FlashAddr+1) ; dto., LSB of Z
lpm
;
; Formulation 3
ldi ZH,High((FlashAddr<<1)|0) ; Access the LSB, MSB of Z
ldi ZL,Low((FlashAddr<<1)|0) ; dto., LSB of Z
lpm
ldi ZH,High((FlashAddr<<1)|1) ; Access the MSB, MSB of Z
ldi ZL,Low((FlashAddr<<1)|1) ; dto., LSB of Z
lpm
Loop:
  rjmp Loop
;
ByteTable:
  .db 1
  .db 1,2
  .db 1,2,3
  .db "This is a text string"
```

Of course, you do not have to define an extra constant named **FlashAddr** but you can directly use the label **ByteTable**: as address in the LDI instructions. And all the +0 and |0 in the formulations are also superfluous because they have no effect.

So, whatever you prefer, it is all the same. The result is always in R0, as the simulated instruction shows. What the simulation also shows is that one access of the flash memory costs three cycles (the LDI are one cycle each). Flash memory therefore is a little bit slower than SRAM and much slower than registers.



Advanced LPM instructions

ATMEL later added the opportunity to use any register as target, the formulation of those instruction are **lpm register,Z**, where register can be any of the 32 registers.

Also a little bit later the auto-increment was implemented. This increases the address in Z after the load has been performed. The effect is that the two instructions **lpm** and **adiw ZL,1** are replaced by the instruction **lpm register,Z+**. Note that this additional step does not increase the access time.

The opposite, the auto-decrease to read tables from the end down to the beginning, was also implemented. Like in the case of SRAM auto-decrement, the decrementation is done prior to the load access. The formulation is **lpm register,-Z** and replaces **sbiw ZL,1** and **lpm**. This additional decrementation does not change access time.

Use examples for LPM

The first example uses LPM to copy a null-terminated text from flash memory to SRAM.

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
\$0100	54	68	69	73	20	74	65	78	74	20	74	6F	20	62	65	20	This text to be
\$0110	63	6F	70	69	65	64	20	74	6F	20	53	52	41	4D	2E	00	copied to SRAM..
\$0120	FF															

```

; Prepare data segment labels
.dseg
sText:
;
.cseg
; Point Z to flash in memory
ldi ZH,High(2*Text)
ldi ZL,Low(2*Text)
; Point X to SRAM target location
ldi XH,High(sText)
ldi XL,Low(sText)
CopyText:
lpm R16,Z+ ; Load from program memory
st X+,R16 ; Store in SRAM
tst R16 ; Null termination?
brne CopyText ; No, go on
; Do'nt run into table
Loop:
rjmp Loop
; Prepare the text in flash memory
Text:
.db "This text to be copied to SRAM.",0x00

```

The whole operation lasts 259 μ s.

The second example is a little bit academic. Guess that your program needs to react to an event with 10 different subroutines, that are of an unequal length: some short ones, some long ones. That can be the case if the user presses one out of ten keys. You can now check whether the initial event was zero, one, two, etc. and up to nine and you can call the ten different subroutines.

Faster and more elegant is it to

- place these ten subroutine addresses into a table,
- to calculate the table address from the given number,
- to read the table entry with LPM, and
- to call the subroutine with ICALL.

What you win here is that you are flexible in extending or reducing the number of subroutines, you are flexible to place them to any address you like, etc.

This is the source code.

```

; Prepare data segment labels
.dseg
sText:
;
.cseg
; Point Z to flash in memory
ldi ZH,High(2*Text)
ldi ZL,Low(2*Text)
ldi XH,High(sText)
ldi XL,Low(sText)
CopyText:
lpm R16,Z+
st X+,R16
tst R16
brne CopyText
; Icall part
.equ select = 0
ldi R16,Low(RAMEND)
out SPL,R16
.ifdef SPH
ldi R16,High(RAMEND)
out SPH,R16
.endif
ldi R16,select ; Load selected routine number here
lsl R16 ; Multiply by two
ldi ZH,High(2*JmpTable) ; Point Z to table
ldi ZL,Low(2*JmpTable)
add ZL,R16 ; Add the doubled selection number

```

```

    ldi R16,0 ; Add carry, if any
    adc ZH,R16
    lpm R16,Z+ ; Read LSB
    lpm ZH,Z ; Read MSB
    mov ZL,R16 ; Copy LSB to ZL
    icall ; Call the routine in Z
Loop:
    rjmp Loop
;
; Routine 0
Routine0:
    nop
    ret
Routine1:
    nop
    nop
    ret
Routine2:
    nop
    nop
    nop
    ret
Routine3:
    nop
    nop
    nop
    nop
    ret
Routine4:
    nop
    nop
    nop
    nop
    nop
    ret
Routine5:
    nop
    nop
    nop
    nop
    nop
    nop
    ret
;
; Jump table
JmpTable:
    .dw Routine0,Routine1,Routine2
    .dw Routine3,Routine4,Routine5
    ; Add additional routines here

```

The simulation has been started with select=0, the table address of this selection has been calculated in Z by adding the left-shifted select to the table's starting address. The address in Z points to the LSB of the first table entry.

```

84: ; Jump table
85: JmpTable:
86: .dw Routine0,Routine1,Routine2
   000033 0018 001A 001D
87: .dw Routine3,Routine4,Routine5
   000036 0021 0026 002C
88: ; Add additional routines here

```

0x0033 is the first entry.

Now the jump address has been read from the table and prepared for an ICALL in Z.

The screenshot shows the AVR simulator interface for 'avr_sim_test_flash'. The simulation status is displayed as follows:

- Prog counter = \$000013
- Instructions = 143
- Stackpointer = \$40FF
- Watchdog = 0.00000%
- Clock frequ. = 1,000,000 Hz
- Time elapsed = 270.0 us
- Stop watch = 270.0 us
- Sleep share = 0.00000%

The SREG register is shown with the following values:

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

The Register window shows the following values:

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	20	01	00	00	66	00

The Messages window shows the following message:

```
$0000: Starting
```

The Register window also shows the following values:

```
R31 = 0
R31:R30= 102
LPM= $0033
```

The Tools window shows the following options:

- Show internal hardware
- Ports
- Timers
- WDT
- ADC
- EEPROM
- SRAM
- Scope
- Alert

```

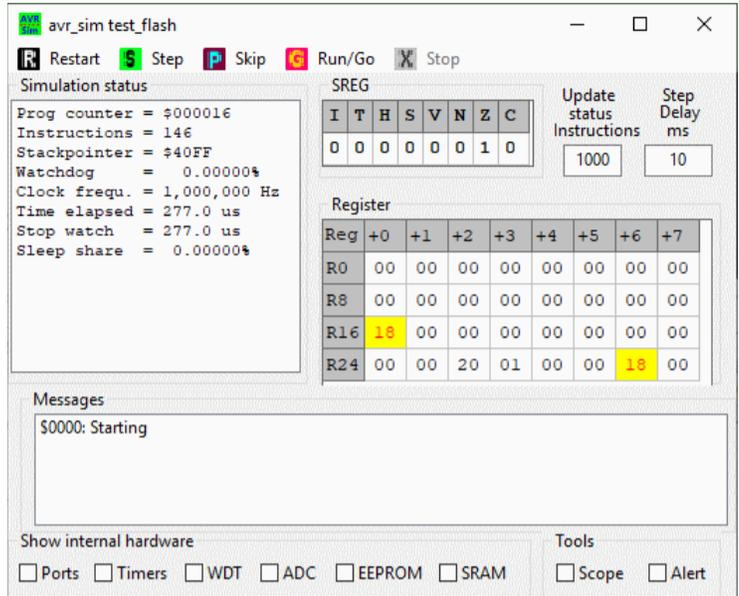
58      50: Routine0:
→ 59      51: 000018  0000  nop
60      52: 000019  9508  ret
    
```

The ICALL has called the Routine0.

With all different possible selects this jumps to the correct routine.

Conclusion:

LPM and its more modern variations offer a wide variety of opportunities to handle texts and to access smaller or larger tables in the large flash memory. Effective programming very often involves such loads from program flash.



Annex

Instructions sorted by function

For the abbreviations used see the list of abbreviations.

<i>Function</i>	<i>Sub function</i>	<i>instruction</i>	<i>Flags</i>	<i>Clk</i>
Register set	0	CLR r1	Z N V	1
	255	SER rh		1
	Constant	LDI rh,c255		1
Copy	Register => Register	MOV r1,r2		1
	SRAM => Register, direct	LDS r1,c65535		2
	SRAM => Register	LD r1,rp		2
	SRAM => Register and INC	LD r1,rp+		2
	DEC, SRAM => Register	LD r1,-rp		2
	SRAM, displaced => Register	LDD r1,ry+k63		2
	Port => Register	IN r1,p1		1
	Stack => Register	POP r1		2
	Program storage Z => R0	LPM		3
	Register => SRAM, direct	STS c65535,r1		2
	Register => SRAM	ST rp,r1		2
	Register => SRAM and INC	ST rp+,r1		2
	DEC, Register => SRAM	ST -rp,r1		2
	Register => SRAM, displaced	STD ry+k63,r1		2
	Register => Port	OUT p1,r1		1
	Register => Stack	PUSH r1		2
Add	8 Bit, +1	INC r1	Z N V	1
	8 Bit	ADD r1,r2	Z C N V H	1
	8 Bit + Carry	ADC r1,r2	Z C N V H	1
	16 Bit, constant	ADIW rd,k63	Z C N V S	2
Subtract	8 Bit, -1	DEC r1	Z N V	1
	8 Bit	SUB r1,r2	Z C N V H	1
	8 Bit, constant	SUBI rh,c255	Z C N V H	1
	8 Bit - Carry	SBC r1,r2	Z C N V H	1
	8 Bit - Carry, constant	SBCI rh,c255	Z C N V H	1
	16 Bit	SBIW rd,k63	Z C N V S	2
Shift	logic, left	LSL r1	Z C N V	1
	logic, right	LSR r1	Z C N V	1
	Rotate, left over Carry	ROL r1	Z C N V	1
	Rotate, right over Carry	ROR r1	Z C N V	1
	Arithmetic, right	ASR r1	Z C N V	1
	Nibble exchange	SWAP r1		1
Binary	And	AND r1,r2	Z N V	1
	And, constant	ANDI rh,c255	Z N V	1
	Or	OR r1,r2	Z N V	1
	Or, constant	ORI rh,c255	Z N V	1
	Exclusive-Or	EOR r1,r2	Z N V	1
	Ones-complement	COM r1	Z C N V	1
	Twos-complement	NEG r1	Z C N V H	1
Bits change	Register, set	SBR rh,c255	Z N V	1
	Register, clear	CBR rh,255	Z N V	1

Function	Sub function	instruction	Flags	Clk
	Register, copy to T-Flag	BST r1,b7	T	1
	Register, copy from T-Flag	BLD r1,b7		1
	Port, set	SBI pl,b7		2
	Port, clear	CBI pl,b7		2
Status bit set	Zero-Flag	SEZ	Z	1
	Carry Flag	SEC	C	1
	Negative Flag	SEN	N	1
	Twos complement carry Flag	SEV	V	1
	Half carry Flag	SEH	H	1
	Signed Flag	SES	S	1
	Transfer Flag	SET	T	1
	Interrupt Enable Flag	SEI	I	1
Status bit clear	Zero-Flag	CLZ	Z	1
	Carry Flag	CLC	C	1
	Negative Flag	CLN	N	1
	Twos complement carry Flag	CLV	V	1
	Half carry Flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt Enable Flag	CLI	I	1
Compare	Register, Register	CP r1,r2	Z C N V H	1
	Register, Register + Carry	CPC r1,r2	Z C N V H	1
	Register, constant	CPI rh,c255	Z C N V H	1
	Register, ≤0	TST r1	Z N V	1
Immediate Jump	Relative	RJMP c4096		2
	Indirect, Address in Z	IJMP		2
	Subroutine, relative	RCALL c4096		3
	Subroutine, Address in Z	ICALL		3
	Return from Subroutine	RET		4
	Return from Interrupt	RETI	I	4
Conditional Jump	Status bit set	BRBS b7,c127		1/2
	Status bit clear	BRBC b7,c127		1/2
	Jump if equal	BREQ c127		1/2
	Jump if not equal	BRNE c127		1/2
	Jump if carry set	BRCS c127		1/2
	Jump if carry clear	BRCC c127		1/2
	Jump if equal or greater	BRSH c127		1/2
	Jump if lower	BRLO c127		1/2
	Jump if negative	BRMI c127		1/2
	Jump if positive	BRPL c127		1/2
	Jump if greater or equal (Signed)	BRGE c127		1/2
	Jump if lower than zero (Signed)	BRLT c127		1/2
	Jump on half carry set	BRHS c127		1/2
	Jump if half carry clear	BRHC c127		1/2
	Jump if T-Flag set	BRTS c127		1/2
	Jump if T-Flag clear	BRTC c127		1/2
	Jump if Twos complement carry set	BRVS c127		1/2
	Jump if Twos complement carry clear	BRVC c127		1/2

Function	Sub function	instruction	Flags	Clk
	Jump if Interrupts enabled	BRIE c127		1/2
	Jump if Interrupts disabled	BRID c127		1/2
Conditioned Jumps	Register bit=0	SBRC r1,b7		1/2/3
	Register bit=1	SBRS r1,b7		1/2/3
	Port bit=0	SBIC pl,b7		1/2/3
	Port bit=1	SBIS pl,b7		1/2/3
	Compare, jump if equal	CPSE r1,r2		1/2/3
Others	No Operation	NOP		1
	Sleep	SLEEP		1
	Watchdog Reset	WDR		1

Directives and Instruction lists in alphabetic order

Assembler directives in alphabetic order

Directive	... means ...
.CSEG	Assemble to the Code segment
.DB	Insert data byte(s)
.DEF	Define a register name
.DW	Insert data word(s)
.ENDMACRO	Macro is complete, stop recording
.ESEG	Assemble to the EEPROM segment
.EQU	Define a constant by name and set its value
.INCLUDE	Insert a file's content at this place as if it would be part of this file
.MACRO	Start to record the following instructions as a macro definition
.ORG	Set the assembler output address to the following number

Instructions in alphabetic order

Instruction	... performs ...
ADC r1,r2	Add r2 with Carry to r1 and store result in r1
ADD r1,r2	Add r2 to r1 and store result in r1
ADIW rd,k63	Add the immediate word constant k63 to the double register rd+1:rd (rd = R24, R26, R28, R30)
AND r1,r2	And bit wise r1 with the value in r2 and store the result in r1
ANDI rh,c255	And bit wise the upper register rh with the constant c255 and store the result in rh
ASR r1	Arithmetic shift the register r1 right
BLD r1,b7	Copy the T-flag in the status register to bit b7 in register r1
BRCC c127	Branch by c127 instructions for- or backwards if the carry flag in the status register is clear
BRCS c127	Branch by c127 instructions for- or backwards if the carry flag in the status register is set
BREQ c127	Branch by c127 instructions for- or backwards if the zero flag in the status register is set
BRGE c127	Branch by c127 instructions for- or backwards if the carry flag in the status register is clear
BRHC c127	Branch by c127 instructions for- or backwards if the half carry flag in the status register is clear
BRHS c127	Branch by c127 instructions for- or backwards if the half carry flag in the status register is set
BRID c127	Branch by c127 instructions for- or backwards if the interrupt flag in the status register is clear
BRIE c127	Branch by c127 instructions for- or backwards if the interrupt flag in the status register is set
BRLO c127	Branch by c127 instructions for- or backwards if the carry flag in the status register is set
BRLT c127	Branch by c127 instructions for- or backwards if the negative and overflow flag in the status register are set
BRMI c127	Branch by c127 instructions for- or backwards if the negative flag in the status register is set
BRNE c127	Branch by c127 instructions for or backwards if the zero flag in the status register is set
BRPL c127	Branch by c127 instructions for- or backwards if the negative flag in the status register is clear
BRSH c127	Branch by c127 instructions for- or backwards if the carry flag in the status register is clear
BRTC c127	Branch by c127 instructions for- or backwards if the transfer flag in the status register is clear
BRTS c127	Branch by c127 instructions for- or backwards if the transfer flag in the status register is set
BRVC c127	Branch by c127 instructions for- or backwards if the overflow flag in the status register is clear
BRVS c127	Branch by c127 instructions for- or backwards if the overflow flag in the status register is set
BST r1,b7	Copy the bit b7 in register r1 to the transfer flag in the status register
CBI pl,b7	Clear bit b7 in the lower port pl
CBR rh,k255	Clear all the bits in the upper register rh, that are set in the constant k255 (mask)
CLC	Clear the carry bit in the status register
CLH	Clear the half carry bit in the status register
CLI	Clear the interrupt bit in the status register, disable interrupt execution
CLN	Clear the negative bit in the status register
CLR r1	Clear the register r1
CLS	Clear the signed flag in the status register
CLT	Clear the transfer flag in the status register
CLV	Clear the overflow flag in the status register
CLZ	Clear the zero flag in the status register
COM r1	Complement register r1 (ones complement)
CP r1,r2	Compare register r1 with register r2
CPC r1,r2	Compare register r1 with register r2 and the carry flag
CPI rh,c255	Compare the upper register rh with the immediate constant c255
CPSE r1,r2	Compare r1 with r2 and jump over the next instruction if equal
DEC r1	Decrement register r1 by 1
EOR r1,r2	Exclusive bit wise Or register r1 with register r2 and store result in r1
ICALL	Call the subroutine at the address in register pair Z (ZH:ZL, R31:R30)
IJMP IN r1,p1	Jump to the address in register pair Z (ZH:ZL, R31:R30)
INC r1	Increment register r1 by 1
LD r1,(rp,rp+,-rp)	Load the register r1 with the content at the location that register pair rp (X, Y or Z) points to (rp+ increments the register pair after loading, -rp decrements the register pair prior to loading)
LDD r1,ry+k63	Load the register r1 with the content at the location that register pair ry (Y or Z), displaced by the constant k63, points to
LDI rh,c255	Load the upper register rh with the constant c255

LDS r1,c65535	Load register r1 with the content at location c65535
LPM LPM r1 LPM r1,Z+ LPM r1,-Z	Load register R0 with the content of the flash memory at the location that register pair Z (ZH:ZL, R31:R30), divided by 2, points to, bit 0 in Z points to lower (0) or upper (1) byte in flash (Load register r1, Z+ increment Z after loading, -Z decrement Z prior to loading)
LSL r1	Logical shift left register r1
LSR r1	Logical shift right register r1
MOV r1,r2	Move register r2 to register r1
NEG r1	Subtract register r1 from Zero
NOP	No operation
OR r1,r2	Bit wise or register r1 with register r2 and store result in register r1
ORI rh,c255	Bit wise or the upper register r1 with the constant c255
OUT p1,r1	Copy register r1 to I/O port p1
POP r1	Increase the stack pointer and pop the last byte on stack to register r1
PUSH r1	Push register r1 to the stack and decrease the stack pointer
RCALL c4096	Push program counter on stack and add signed constant c4096 to the program counter (relative call)
RET	Pop program counter from stack (return to call address)
RETI	Enable interrupts and pop program counter from stack (return from interrupt)
RJMP c4096	Relative jump, add signed constant c4096 to program address
ROL r1	Rotate register r1 left, copy carry flag to bit 0
ROR r1	Rotate register r1 right, copy carry flag to bit 7
SBC r1,r2	Subtract r2 and the carry flag from register r1 and write result to r1
SBCI rh,c255	Subtract constant c255 and carry flag from the upper register rh and write result to rh
SBI pl,b7	Set bit b7 in the lower port pl
SBIC pl,b7	If bit b7 in the lower port pl is clear, jump over the next instruction
SBIS pl,b7	If bit b7 in the lower port pl is set, jump over the next instruction
SBIW rd,k63	Subtract the constant k63 from the register pair rd (rd+1:rd, rd = R24, R26, R28, R30)
SBR rh,c255	Set the bits in the upper register rh, that are one in constant c255
SBRC r1,b7	If bit b7 in register r1 is clear, jump over next instruction
SBRS r1,b7	If bit b7 in register r1 is set, jump over next instruction
SEC	Set carry flag in status register
SEH	Set half carry flag in status register
SEI	Set interrupt flag in status register, enable interrupt execution
SEN	Set negative flag in status register
SER rh	Set all bits in the upper register rh
SES	Set sign flag in status register
SET	Set transfer flag in status register
SEV	Set overflow flag in status register
SEZ	Set zero flag in status register
SLEEP	Put controller to the selected sleep mode
ST (rp/rp+/-rp),r1	Store content in register r1 to the memory location in register pair rp (rp = X, Y, Z; rp+: increment register pair after store; -rp: decrement register pair prior to store)
STD ry+k63,r1	Store the content of register r1 at the location that register pair ry (Y or Z), displaced by the constant k63, points to
STS c65535,r1	Store the content of register r1 at the location c65535
SUB r1,r2	Subtract register r2 from register r1 and write result to r1
SUBI rh,c255	Subtract the constant c255 from the upper register rh
SWAP r1	Exchange upper and lower nibble in register r1
TST r1	Compare register r1 with Zero
WDR	Watchdog reset

Port details

The table of the relevant ports in the ATMEL AVR types AT90S2313, 2323 and 8515. Byte wise accessible ports or register pairs are not displayed in detail. No warranty for correctness, see the original data sheets!

Status-Register, Accumulator flags

Port	Function	Port-Address	RAM-Address
SREG	Status Register Accumulator	0x3F	0x5F

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

Bit	Name	Meaning	Opportunities	Command
7	I	Global Interrupt Flag	0: Interrupts disabled	CLI
			1: Interrupts enabled	SEI
6	T	Bit storage	0: Stored bit is 0	CLT
			1: Stored bit is 1	SET
5	H	Halfcarry-Flag	0: No halfcarry occurred	CLH
			1: Halfcarry occurred	SEH
4	S	Sign-Flag	0: Sign positive	CLS
			1: Sign negative	SES
3	V	Two's complement-Flag	0: No carry occurred	CLV
			1: Carry occurred	SEV
2	N	Negative-Flag	0: Result was not negative/smaller	CLN
			1: Result was negative/smaller	SEN
1	Z	Zero-Flag	0: Result was not zero/unequal	CLZ
			1: Result was zero/equal	SEZ
0	C	Carry-Flag	0: No carry occurred	CLC
			1: Carry occurred	SEC

Stackpointer

Port	Function	Port-Address	RAM-Address
SPL/SPH	Stackpointer	003D/0x3E	0x5D/0x5E

Name	Meaning	Availability
SPL	Low-Byte of Stack pointer	From AT90S2313 upwards, not in 1200
SPH	High-Byte of Stack pointer	From AT90S8515 upwards, only in devices with >256 bytes internal SRAM

SRAM and External Interrupt control

Port	Function	Port-Address	RAM-Address
MCUCR	MCU General Control Register	0x35	0x55

7	6	5	4	3	2	1	0
SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00

Bit	Name	Meaning	Opportunities
7	SRE	Ext. SRAM Enable	0=No external SRAM connected
			1=External SRAM connected

Bit	Name	Meaning	Opportunities
6	SRW	Ext. SRAM Wait States	0=No extra wait state on external SRAM
			1=Additional wait state on external SRAM
5	SE	Sleep Enable	0=Ignore SLEEP instructions
			1=SLEEP on instruction
4	SM	Sleep Mode	0=Idle Mode (Half sleep)
			1=Power Down Mode (Full sleep)
3	ISC11	Interrupt control Pin INT1 (connected to GIMSK)	00: Low-level initiates Interrupt
2	ISC10		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt
1	ISC01	Interrupt control Pin INTO (connected to GIMSK)	00: Low-level initiates interrupt
0	ISC00		01: Undefined
			10: Falling edge triggers interrupt
			11: Rising edge triggers interrupt

External Interrupt Control

Port	Function	Port-Address	RAM-Address				
GIMSK	General Interrupt Maskregister	0x3B	0x5B				
7	6	5	4	3	2	1	0
INT1	INT0	-	-	-	-	-	-
Bit	Name	Meaning	Opportunities				
7	INT1	Interrupt by external pin INT1 (connected to mode in MCUCR)	0: External INT1 disabled				
			1: External INT1 enabled				
6	INT0	Interrupt by external Pin INTO (connected to mode in MCUCR)	0: External INTO disabled				
			1: External INTO enabled				
0...5	(Not used)						

Port	Function	Port-Address	RAM-Address				
GIFR	General Interrupt Flag Register	0x3A	0x5A				
7	6	5	4	3	2	1	0
INTF1	INTF0	-	-	-	-	-	-
Bit	Name	Meaning	Opportunities				
7	INTF1	Interrupt by external pin INT1 occurred	Automatic clear by execution of the Int-Routine or Clear by instruction				
6	INTF0	Interrupt by external pin INTO occurred					
0...5	(Not used)						

Timer Interrupt Control

Port	Function	Port-Address	RAM-Address				
TIMSK	Timer Interrupt Maskregister	0x39	0x59				
7	6	5	4	3	2	1	0
TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-

Bit	Name	Meaning	Opportunities
7	TOIE1	Timer/Counter 1 Overflow-Interrupt	0: No Int at overflow 1: Int at overflow
6	OCIE1A	Timer/Counter 1 Compare A Interrupt	0: No Int at equal A 1: Int at equal A
5	OCIE1B	Timer/Counter 1 Compare B Interrupt	0: No Int at B 1: Int at equal B
4	(Not used)		
3	TICIE1	Timer/Counter 1 Capture Interrupt	0: No Int at Capture 1: Int at Capture
2	(Not used)		
1	TOIE0	Timer/Counter 0 Overflow-Interrupt	0: No Int at overflow 1: Int at overflow
0	(Not used)		

Port	Function	Port-Address	RAM-Address
TIFR	Timer Interrupt Flag Register	0x38	0x58

7	6	5	4	3	2	1	0
TOV1	OCF1A	OCF1B	-	ICF1	-	TOV0	-

Bit	Name	Meaning	Opportunities
7	TOV1	Timer/Counter 1 Overflow reached	Interrupt-Mode: Automatic Clear by execution of the Int-Routine
6	OCF1A	Timer/Counter 1 Compare A reached	
5	OCF1B	Timer/Counter 1 Compare B reached	
4	(Not used)		OR
3	ICF1	Timer/Counter 1 Capture-Event occurred	
2	(not used)		Polling-Mode: Clear by instruction
1	TOV0	Timer/Counter 0 Overflow occurred	
0	(not used)		

Timer/Counter 0

Port	Function	Port-Address	RAM-Address
TCCR0	Timer/Counter 0 Control Register	0x33	0x53

7	6	5	4	3	2	1	0
-	-	-	-	-	CS02	CS01	CS00

Bit	Name	Meaning	Opportunities
2..0	CS02..CS00	Timer Clock	000: Stop Timer
			001: Clock = Chip clock
			010: Clock = Chip clock / 8
			011: Clock = Chip clock / 64
			100: Clock = Chip clock / 256
			101: Clock = Chip clock / 1024
			110: Clock = falling edge of external Pin T0
			111: Clock = rising edge of external Pin T0
3..7	(not used)		

Port	Function	Port-Address	RAM-Address
TCNT0	Timer/Counter 0 count register	0x32	0x52

Timer/Counter 1

Port	Function	Port-Address	RAM-Address
TCCR1A	Timer/Counter 1 Control Register A	0x2F	0x4F

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10

Bit	Name	Meaning	Opportunities
7	COM1A1	Compare Output A	00: OC1A/B not connected 01: OC1A/B changes polarity
6	COM1A0		
5	COM1B1	Compare Output B	10: OC1A/B to zero 11: OC1A/B to one
4	COM1B0		
3	(not used)		
2			
1..0	PWM11 PWM10	Pulse width modulator	00: PWM off 01: 8-Bit PWM 10: 9-Bit PWM 11: 10-Bit PWM

Port	Function	Port-Address	RAM-Address
TCCR1B	Timer/Counter 1 Control Register B	0x2E	0x4E

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10

Bit	Name	Meaning	Opportunities
7	ICNC1	Noise Canceler on ICP-Pin	0: disabled, first edge starts sampling 1: enabled, min four clock cycles
6	ICES1	Edge selection on Capture	0: falling edge triggers Capture 1: rising edge triggers Capture
5..4	(not used)		
3	CTC1	Clear at Compare Match A	1: Counter set to zero if equal
2..0	CS12..CS10	Clock select	000: Counter stopped 001: Clock 010: Clock / 8 011: Clock / 64 100: Clock / 256 101: Clock / 1024 110: falling edge external Pin T1 111: rising edge external Pin T1

Port	Function	Port-Address	RAM-Address
TCNT1L/H	Timer/Counter 1 count register	0x2C/0x2D	0x4C/0x4D

Port	Function	Port-Address	RAM-Address
OCR1AL/H	Timer/Counter 1 Output Compare register A	0x2A/0x2B	0x4A/0x4B hex

Port	Function	Port-Address	RAM-Address
OCR1BL/H	Timer/Counter 1 Output Compare register B	0x28/0x29	0x48/0x49

Port	Function	Port-Address	RAM-Address
ICR1L/H	Timer/Counter 1 Input Capture Register	0x24/0x25	0x44/0x45

Watchdog-Timer

Port	Function	Port-Address	RAM-Address
WDTCR	Watchdog Timer Control Register	0x21	0x41

7	6	5	4	3	2	1	0
-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0

Bit	Name	Meaning	WDT-cycle at 5.0 Volt
7..5		(not used)	
4	WDTOE	Watchdog Turnoff Enable	Previous set to disabling of WDE required
3	WDE	Watchdog Enable	1: Watchdog active
2..0	WDP2..WDP0	Watchdog Timer Prescaler	000: 15 ms 001: 30 ms 010: 60 ms 011: 120 ms 100: 240 ms 101: 490 ms 110: 970 ms 111: 1,9 s

EEPROM

Port	Function	Port-Address	RAM-Address
EEARL/H	EEPROM Address Register	0x1E/0x1F	0x3E/0x3F

EEARH only in types with more than 256 Bytes EEPROM (from AT90S8515 upwards)

Port	Function	Port-Address	RAM-Address
EEDR	EEPROM Data Register	0x1D	0x3D

Port	Function	Port-Address	RAM-Address
EECR	EEPROM Control Register	0x1C	0x3C

7	6	5	4	3	2	1	0
-	-	-	-	-	EEMWE	EEWE	EERE

Bit	Name	Meaning	Function
7..3	(not used)		
2	EEMWE	EEPROM Master Write Enable	Previous set enables write cycle
1	EEWE	EEPROM Write Enable	Set to initiate write
0	EERE	EEPROM Read Enable	Set initiates read

Serial Peripheral Interface SPI

Port	Function	Port-Address	RAM-Address
SPCR	SPI Control Register	0x0D	0x2D

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Bit	Name	Meaning	Function
7	SPIE	SPI Interrupt Enable	0: Interrupts disabled 1: Interrupts enabled
6	SPE	SPI Enable	0: SPI disabled 1: SPI enabled
5	DORD	Data Order	0: MSB first 1: LSB first
4	MSTR	Master/Slave Select	0: Slave 1: Master
3	CPOL	Clock Polarity	0: Positive Clock Phase 1: Negative Clock Phase
2	CPHA	Clock Phase	0: Sampling at beginning of Clock Phase 1: Sampling at end of Clock Phase
1	SPR1	SCK clock frequency	00: Clock / 4
0	SPR0		01: Clock / 16
			10: Clock / 64
			11: Clock / 128

Port	Function	Port-Address	RAM-Address
SPSR	SPI Status Register	0x0E	0x2E

7	6	5	4	3	2	1	0
SPIF	WCOL	-	-	-	-	-	-

Bit	Name	Meaning	Function
7	SPIF	SPI Interrupt Flag	Interrupt request
6	WCOL	Write Collision Flag	Write collision occurred
5..0	(not used)		

Port	Function	Port-Address	RAM-Address
SPDR	SPI Data Register	0x0F	0x2F

UART

Port	Function	Port-Address	RAM-Address
UDR	UART I/O Data Register	0x0C	0x2C

Port	Function	Port-Address	RAM-Address
USR	UART Status Register	0x0B	0x2B

7	6	5	4	3	2	1	0
RXC	TXC	UDRE	FE	OR	-	-	-

Bit	Name	Meaning	Function
7	RXC	UART Receive Complete	1: Char received
6	TXC	UART Transmit Complete	1: Shift register empty
5	UDRE	UART Data Register Empty	1: Transmit register available
4	FE	Framing Error	1: Illegal Stop-Bit
3	OR	Overrun	1: Lost char
2..0		(not used)	

Port	Function	Port-Address	RAM-Address
UCR	UART Control Register	0x0A	0x2A

7	6	5	4	3	2	1	0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

Bit	Name	Meaning	Function
7	RXCIE	RX Complete Interrupt Enable	1: Interrupt on received char
6	TXCIE	TX Complete Interrupt Enable	1: Interrupt at transmit complete
5	UDRIE	Data Register Empty Interrupt Enable	1: Interrupt on transmit buffer empty
4	RXEN	Receiver Enable	1: Receiver enabled
3	TXEN	Transmitter Enable	1: Transmitter enabled
2	CHR9	9-bit Characters	1: Char length 9 Bit
1	RXB8	Receive Data Bit 8	(holds 9 th data bit on receive)
0	TXB8	Transmit Data Bit 8	(write 9 th data bit for transmit here)

Port	Function	Port-Address	RAM-Address
UBRR	UART Baud Rate Register	0x09	0x29

Analog Comparator

Port	Function	Port-Address	RAM-Address
ACSR	Analog Comparator Control and Status Register	0x08	0x28

7	6	5	4	3	2	1	0
ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Bit	Name	Meaning	Function
7	ACD	Disable	Disable Comparators
6		(not used)	
5	ACO	Comparator Output	Read: Output of the Comparators
4	ACI	Interrupt Flag	1: Interrupt request

Bit	Name	Meaning	Function
3	ACIE	Interrupt Enable	1: Interrupts enabled
2	ACIC	Input Capture Enable	1: Connect to Timer 1 Capture
1	ACIS1	Input Capture Enable	00: Interrupt on edge change
0	ACIS0		01: (not used)
			10: Interrupt on falling edge
			11: Interrupt on rising edge

I/O Ports

Port	Register	Function	Port-Address	RAM-Address
A	PORTA	Data Register	0x1B	0x3B
	DDRA	Data Direction Register	0x1A	0x3A
	PINA	Input Pins Address	0x19	0x39
B	PORTB	Data Register	0x18	0x38
	DDRB	Data Direction Register	0x17	0x37
	PINB	Input Pins Address	0x16	0x36
C	PORTC	Data Register	0x15	0x35
	DDRC	Data Direction Register	0x14	0x34
	PINC	Input Pins Address	0x13	0x33
D	PORTD	Data Register	0x12	0x32
	DDRD	Data Direction Register	0x11	0x31
	PIND	Input Pins Address	0x10	0x30

Ports, alphabetic order

ACSR, Analog Comparator Control and Status Register
DDRx, Port x Data Direction Register
EEAR, EEPROM address Register
EECR, EEPROM Control Register
EEDR, EEPROM Data Register
GIFR, General Interrupt Flag Register
GIMSK, General Interrupt Mask Register
ICR1L/H, Input Capture Register 1
MCUCR, MCU General Control Register
OCR1A, Output Compare Register 1 A
OCR1B, Output Compare Register 1 B
PINx, Port Input Access
PORTx, Port x Output Register
SPL/SPH, Stackpointer
SPCR, Serial Peripheral Control Register
SPDR, Serial Peripheral Data Register
SPSR, Serial Peripheral Status Register
SREG, Status Register
TCCR0, Timer/Counter Control Register, Timer 0
TCCR1A, Timer/Counter Control Register 1 A
TCCR1B, Timer/Counter Control Register 1 B
TCNT0, Timer/Counter Register, Counter 0
TCNT1, Timer/Counter Register, Counter 1
TIFR, Timer Interrupt Flag Register
TIMSK, Timer Interrupt Mask Register
UBRR, UART Baud Rate Register
UCR, UART Control Register
UDR, UART Data Register
WDTCR, Watchdog Timer Control Register

List of abbreviations

The abbreviations used are chosen to include the value range. Register pairs are named by the lower of the two registers. Constants in jump instructions are automatically calculated from the respective labels

during assembly.

Category	Abbrev.	Means ...	Value range
Register	r1	Ordinary Source and Target register	R0..R31
	r2	Ordinary Source register	
	rh	Upper page register	R16..R31
	rd	Twin register	R24(R25), R26(R27), R28(R29), R30(R31)
	rp	Pointer register	X=R26(R27), Y=R28(R29), Z=R30(R31)
	ry	Pointer register with displacement	Y=R28(R29), Z=R30(R31)
Constant	k63	Pointer-constant	0..63
	c127	Conditioned jump distance	-64..+63
	c255	8-Bit-Constant	0..255
	c4096	Relative jump distance	-2048..+2047
	c65535	16-Bit-Address	0..65535
Bit	b7	Bit position	0..7
Port	p1	Ordinary Port	0..63
	pl	Lower page port	0..31