

An abstract geometric design featuring three overlapping circles of varying sizes, each composed of three concentric rings in shades of gold and beige. Three thin, intersecting lines in a light gold color cross the page, creating a sense of depth and movement. The circles are positioned in the upper right, middle right, and lower right areas of the page.

# AVR GCC Tutorial (WinAVR)

Early before alpha Version 1.00

This a Google translation for the famous German Wiki tutorial to be found in [www.mikrocontroller.net](http://www.mikrocontroller.net)

By **takashi**

*Some plapla if you do have some time!!! If not send this section to hell!! Don't say I did not warn you man!!!! don't blame me if you found this section silly but after all someone, somewhere may consider this a useful "LIFE" & "Technical" experience. The story contains some fancy not a 100% reality just to be truthful (some salt & sugar to make a nice taste).*

## The "Hard Times" story by takashi

Hello everybody I was asked for a college project using AVR's microcontrollers so I started making some shopping buying the STK 200/300 kit programmer and a parallel port cable (about 100 L.E= 16\$) , surfing the web for necessary software and seeking knowledge that can be found in any site.

I forgot to tell you that it was the first time for me to program a microcontroller ever!! . The decision was made to use the C language as everyone out there is saying it's easier. So again i surfed the web to find some C compiler out there, and so sad every one out there want you to PAY!!!!(-actually I found after that I was wrong-) & this was not in my options. So I was so pleased to find some Gnu solution out there & that was the WinAVR plug-in for the AvrStudio .

I forgot also to tell you that I know almost nothing about how to use the C language (except for an ill course in a previous semester) and of course programming the "AVR" with "C language" is just as ascending to the moon on your foot!!! And of course the WinAVR documentation is useless for me. So again started "googling" for some easy in depth tutorial to help me get rid of this project headache. After several hours of internet mess I failed to find any thing in English except for this [German tutorial](#) and of course all what I know in Germany is (ich liebe dich).

So it seemed like a big nightmare but thanks god I discovered the web page translation tool that Google provides.

For the first moment I thought that this Google tool would add to my misery to be misery= misery++ however to translate a single word sounds good but automatic translation for a complete web page seems a little bit fancy! The least thing you may face is some type of cryptic translation that would make your misery+++. So in a desperate moment I decided to give it a try "I will not lose anything any way" typing the URL pressing the translate button and the big surprise I was in front of a nice neat translation that is far away beyond my dreams

"Gohooooooooo!!!" ; Of course it's not "yahoooooooo!" but after all Google did it. Of course not by 100 percent but 85% is fair enough for me to start playing with the WinAVR. But Oh Oh .....?! Oh no!!?!!

Mr. Google you should have been smart enough not to translate the C Code!!? It might not work this way!!?.

And look to that!! Google smashed the "look and feel" of the tutorial. The colored Code beautiful boxes is now an ugly long line && sadly saying Google translated only 1/9 of the too long (100 paper page fill) webpage. It was a shock but "no desperation with life & no life with desperation" .I decided to finish the road that I had started and started with the help of one my friends who is a web designer a hard work. My friend started to break up the very long webpage in to smaller pages & I took those pages and started uploading them on a free web space on the internet & give the Google webpage translation tool the URL's of the pages by order & one by one. after finishing the translation I started to copy/paste the translation to a word document & started to organize every thing that Google messed ;the "look and feel" , the nice code boxes , the corrupted C Code, and hey this word should mean "INPUT B" not "ENTRANCE B" && "ELF" is just "ELF" not "eleven in Germany" && "Similar" actually means "Analog" && no it's not "EXIT" Google it's "output" & how funny "breathing gas" is actually "ATmegas" & lots of other replacements. I had even translated the words in the pictures. During this I developed

some nice tricks; some words Google could not translate it; I took those words one by one and put them in Google web search, and guess what??? Google answered me with the famous question? Did you mean palapala? I took the **palapala** and try to translate it again and “voila!!!” this time it does have a meaning; Google actually helped me correcting the Germans misspellings I corrected & translated dozens of words by this method.

I guess that I have learned much Germany in those three weeks of this “*tutorial debugging*” ***three weeks is not from the salt & sugar.***

**At last guess what??** I did the job for the project in **assembly** not because the tutorial is bad but because this document debugging consumed all the time & effort. And assembly which I am familiar with, seemed to be a quick practical solution more than debugging the document & printing it. ***I have learned that at some times it's easier to select the hard ways than following what the others tell you that it's the easiest ways.***

THE END

*At last I decided to share this work with the AVR English speaking community.*

*I am not clamming it's a perfect translation (I do not speak Germany remember). But in the hope that the internet community would help proofing this document against spelling & grammar mistakes that is still resident& OF COURSE TRANSLATE THE HEADSTRONG WORDS THAT neither GOOGLE NOR ME COULD TRANSLATE IT.*

*I hope to see the beta version soon.*

[\*http://www.bridgetofaith.com/\*](http://www.bridgetofaith.com/)

# License

---

You can freely distribute and make enhancements to the technical related parts of this document and its next versions under the *creative commons share alike 2.0 license* to be found at the following URL:

<http://creativecommons.org/licenses/by-sa/2.0/>

***Providing that NOT to make any Edits in the first three pages of this Document without direct permission from me.***

***You can contact me at the following E-mail:***

***[Takashi85eg@gmail.com](mailto:Takashi85eg@gmail.com)***

***Please make the message title “AVR GCC Tutorial” or it may be moved to SPAM.***

## Contributors in the translation

---

*Hello.....?? Is any body here!!!*

*If you think you deserve to be mentioned here go ahead and write your name, and make me know at*

*[Takashi85eg@gmail.com](mailto:Takashi85eg@gmail.com)*

*The document still needs more work to reach its final version.*

# Table of contents

- [1\\_preface](#)
- [2\\_needed tools](#)
- [3When you found yourself in trouble DO THIS!!](#)
- [4 Generating the machine code](#)
- [5\\_introduction example](#)
- [6\\_Exploring Makefiles](#)
  - [6,1\\_type of microcontroller set](#)
  - [6,2\\_source coding files register](#)
  - [6,3\\_programming device adjust](#)
  - [6,4\\_application](#)
  - [6.5\\_other parameters](#)
    - [6.5.1\\_optimization degrees](#)
    - [6.5.2 Debug\\_format](#)
    - [6.5.3\\_assembler files](#)
    - [6.5.4\\_clock frequency](#)
  - [6,6\\_input files for simulation in AVR Studio](#)
- [7\\_integral ones \(Integer\) data types](#)
- [8\\_bit fields](#)
- [9\\_fundamental program structure of a  \$\mu\$ C program](#)
  - [9.1\\_sequential program sequence](#)
  - [9.2\\_interrupt-controlled program sequence](#)
- [10\\_general accesses to registers](#)
  - [10,1 I/O\\_registers](#)
    - [10.1.1\\_Read of a I/O register](#)
      - [10.1.1.1\\_Read of a bit](#)

- 10.1.2\_write of a I/O register
    - 10.1.2.1write of bits
  - 10.1.3\_control rooms on a certain condition
- 11\_access to port
  - 11,1\_data direction determines
    - 11.1.1\_whole ones of port
  - 11.2\_pre-defined bit numbers for I/O registers
  - 11.3\_digital signals
  - 11,4\_outputs
  - 11,5\_inputs (as signals come into  $\mu\text{C}$ )
    - 11.5.1\_signal coupling
    - 11.5.2\_keys and switches
      - 11.5.2.1 activation of pull-up resistors
      - 11.5.2.2Debouncing inputs
  - 11.6\_analog
  - 11,7\_16-Bit port register (ADC, ICR1, OCR1, TCNT1, UBRR)
  - 11,8\_IO registers as parameters and variables
- 12 the UART
  - 12.1\_general to the UART
  - 12.2 the hardware
  - 12.3\_sending with the UART
    - 12.3.1\_sending individual indications
    - 12.3.2\_Writes of a character string (stringer)
    - 12.3.3\_Writes of variables' contents
  - 12,4\_indications receiving

- 12,5 software\_UART
- 13\_analog input and output
  - 13,1\_ADC (Analog to digital converter)
    - 13.1.1 the\_internal ADC in the AVR
      - 13.1.1.1 the\_registers of the ADC
      - 13.1.1.2\_activating the ADC
    - 13.1.2\_analog-digital transformation without internal ADC
      - 13.1.2.1\_fairs of a resistance
      - 13.1.2.2\_ADC over comparator
  - 13,2\_DAC (digital analog to converter)
    - 13.2.1\_DAC over several digital outputs
    - 13.2.2\_PWM (pulse width modulation)
- 14 the timers/Counter of the AVR
  - 14.1 the\_prescaler
  - 14,2 8-bits timers/Counter
  - 14,3 16-Bit timer/Counter
    - 14.3.1 the PWM\_mode of operation
    - 14.3.2\_reference value examination
    - 14.3.3\_catching an input signal (input Capturing)
  - 14.4 common registers
- 15 Sleep modes
- 16 the Watchdog
  - 16.1 How now does the Watchdog function?
  - 16.2 Watchdog\_Possible applications
- 17 programming with interrupts



- 17,1 requirements at interrupt routines
- 17,2 sources of interrupt
- 17,3 registers
- 17,4 general over interrupt processing
- 17,5 interrupts with the AVR GCC compiler (WinAVR)
  - 17.5.1\_ISR
  - 17.5.2\_interruptible Interrupt routine
- 17,6\_data exchange with interrupt routines
- 17,7\_interrupt routines and register accesses
- 17,8\_Which makes the main program?
- 18\_memory accesses
  - 18,1\_RAM
  - 18,2\_program memories (Flash)
    - 18.2.1\_byte reads
    - 18.2.2\_word reads
    - 18.2.3\_Floats and Structs read
    - 18.2.4\_simplification for character strings (stringers) in the Flash
    - 18.2.5\_Flash in application write
    - 18.2.6 Why so Complicated in such a way?
  - 18,3\_EEPROM
    - 18.3.1\_bytes read/write
    - 18.3.2\_word reads/writes
    - 18.3.3\_block reads/writes
    - 18.3.4\_EEPROM memory map in.eep file
    - 18.3.5\_EEPROM variable on firm addresses put

- 18.3.6\_acquaintance of problems with the EEPROM functions
- 19\_the\_use of printf
- 20\_assembler and Inline assembler
  - 20,1 Inline\_assembler
  - 20,2\_assembler files
  - 20.3\_global variables for data exchange
    - 20.3.1\_global variables in the assembler file put on
    - 20.3.2\_variables more largely than 1 byte
- 21\_appendix
  - 21,1\_characteristics with the adjustment existing source code
    - To 21.1.1\_functions became outdated for the declaration of interrupt routines
    - To 21.1.2\_functions became outdated to the port access
    - To 21.1.3\_functions became outdated to the access to bits in registers
    - 21.1.4\_self defined (non-standardized) integral data types
  - 21.2\_additional functions in the Makefile
    - 21.2.1\_libraries (Libraries/.a files) add
    - 21.2.2 Fuse\_bits
  - 21.3\_external reference tension of the internal analogue-digital converter

22\_TODO

## Preface

This Tutorial is to facilitate the entrance into the programming of Atmel AVR [Microcontroller](#) in the programming language [C](#) with the free (“free”) C-compiler avr GCC.

This Tutorial presupposes basic knowledge in C. Previous knowledge in the programming of micro-control-learn, neither in assembler nor in another language, are not necessary.

The original version comes from Christian Schifferle, many new sections and current adjustments from Martin Thomas. Many of the functions used in the original document are no more contained in of the current versions of the avr [GCC of C](#)-compiler and the run time library avr libc or are not to be used no more. This Tutorial was adapted to the new functions/methods.

The explanations and examples refer to the versions 3.4.5 of the avr GCC compiler and 1.4.3 avr libc, like that as them in WinAVR 20060125 are contained. The differences to older versions are described in the main text and appendix is however recommended to beginners to use the current versions.

In this text becomes frequently on the standard library for the avr GCC compiler, which referred avr libc. On-line version avr **libc of the documentation** is [here](#). With [WinAVR](#) the documentation belongs to the scope of supply and is along-installed.

A version of this Tutorial as pdf to expressions is here available (not always on current conditions):  
[http://www.siwawi.arubi.uni-kl.de/avr\\_projects/AVR-GCC-Tutorial\\_-\\_www\\_mikrocontroller\\_net.pdf](http://www.siwawi.arubi.uni-kl.de/avr_projects/AVR-GCC-Tutorial_-_www_mikrocontroller_net.pdf)

## Needed tools

In order to provide and test own programs for AVR's by means of avr gcc/avr libc, following hard and software are needed:

- Kit or Experimental board, for an AVR microcontroller which is supported by avr GCC compiler (all ATmegs and most AT90, see documentation avr libc for supported types). This test board can be soldered or also developed on a patch board. Some descriptions of registers in these Tutorials refer at90S2313 to be outdated in the meantime. The by far largest part of the text is however valid for all microcontrollers of the AVR family. Useful test platform are also the [STK500](#) and the [AVR Butterfly](#) of Atmel. [more](#).
- The avr GCC compiler and avr libc. Free of charge available for almost all platforms and operating systems. For Ms-Windows in the package [WinAVR](#); for Unix/Linux see also notes in the article [AVR-GCC](#).
- Programming software and - [hardware](#) e.g. PonyProg (see also: [Pony Prog Tutorial](#)) or [AVRDUDE](#) with [STK200-Dongle](#) or the hard and software available of Atmel ([STK500](#), Atmel AVRISP, AVR [Studio](#)).
- Not necessarily, but for the simulation and for debugging under Ms-Windows quite useful: [AVR Studio](#) (see section [Exploring Makefile](#)).

## When you found yourself in trouble DO THIS!!

- Find out whether it's actually an avr (- GCC) specific problem or you just requires the revitalization of your C-knowledge. One knows possibly general C-questions. Otherwise: (read gbt's “free of charge” on-line) [C-book](#).
- [AVR check list](#)
- Above all [The avr libc documentation](#) (however not only), the section **Related Pages/Frequently Asked Questions** = questions often posted (and answers in addition).
- The article [AVR-GCC](#) in this Wiki read.

- The GCC forum on [www.mikrocontroller.net](http://www.mikrocontroller.net) after comparable problems search.
- The avr GCC forum with [avrfreaks](http://avrfreaks.net) after comparable problems search.
- [Archives of the avr GCC mailing list](http://www.avrfreaks.net/forums/avr-gcc-mailing-list) after comparable problems search.
- Look for example code. Particularly in the Academy of [AVRFREAKS](http://www.avrfreaks.net) (announce).
- Google or yahoos ask never harm.
- For problems with the control of internal AVR functions with C-code: read the data sheet of the microcontroller (completely and at the best one twice). Data sheets are available by the [Atmel web pages](http://www.atmel.com) as pdf files. The complete data sheet (complete) and not the edited version (summary) use.
- In addition, the example programs in the AVR [Tutorial](http://www.avrfreaks.net/forums/avr-gcc-tutorial) are written in AVR assembler, explanations and proceeding are transferable to C-programs.
- A contribution into one of the forums or write a Mail to the mailing list. Give as much as possible information: microcontroller, compiler version, used libraries, cutouts from the source code, exact error messages and/or description of the failure. In the case of control of external devices describe or outline the wiring (e.g. with [Andys ASCII Circuit](http://www.avrfreaks.net/forums/avr-gcc-tutorial)). See in addition also: [“smart questions place”](http://www.avrfreaks.net/forums/avr-gcc-tutorial).

## Generating the machine code

From the C-source code the avr GCC compiler (together with the Preprocessor directives and the linker) produces machine code for the AVR microcontroller. Usually this code lies in the Intel Hex format pronounced (“Hex file”). The programming software (e.g. [AVRDUDE](http://www.avrdude.org), PonyProg or AVRStudio/STK500-plugin) reads this file in and transmits the contained information (the machine code) into the memory of the microcontroller. In theory “only” the avr GCC compiler (and the linker) with the “correct” options or parameters is able to generate a “Hex file” from C-code; however there is two main ways to generate this machine code:

- The use of an integrated development environment (IDE), with which all parameters can be accomplished e.g. in dialogue boxes. Just as AVRStudio which can be used starting from version 4.12 (free of charge with atmel.com) with WinAVR (as plug-in) to form together an integrated development environment for the compiler avr GCC (you must have AVRStudio and WinAVR to be installed on the computer) Further IDEs for the avr GCC (without requirement on completeness): AtmanAvr C (relatively favorable), KamAVR (free) VMLab (starting from version 3.12 likewise free of charge).
- The use of the MAKE program with suitable Makefiles.

Integrated development environments differ strongly in your operation and are not also available for all platforms, on which the compiler is executable (e.g. AVRStudio only for Ms-Windows). The use of the avr GCC compiler with IDEs is referred here to their documentation.

In the following sections the generation of machine code for an AVR (“hex file”) from C-source code (“C-files”) is more near described on the basis of the universal and platform-independent proceeding by means of make and Makefiles. You will find many of the options described here also available at the configuration dialogue of the avr GCC Plugin in AVRStudio (AVRStudio generate a Makefile in a folder of the Project directory). With the change beginning from the Makefile to WinAVR main collecting to AVRStudio you should know that AVRStudio (for: AVRStudio version 4.12SP1) with a new project the optimization option (see section [Exploring Makefiles](http://www.avrfreaks.net/forums/avr-gcc-tutorial), typically: - Os) adjusts; and the mathematical library avr libc (libm.a, linker option - lm) does not merge. Both is standard in use of Makefiles after WinAVR main collecting and should therefore be adjusted “manually” in AVRStudio at the avr GCC Plugins configuration dialogue, in order to produce also with AVRStudio compact code.

## Introduction example

In the beginning we will start with a small example, with which the use of the compiler and the aid programs (the so-called *Toolchain*) is demonstrated. Detailed explanations follow in the further sections of these Tutorials.

The program is to switch some outputs OFF and others ON in an AVR microcontroller. The example is programmed for an ATmega16 ([data sheet](#)), can be modified however in a general manner for other microcontrollers of the AVR family.

First the source code of the application, which in a text file with the name *main.c* one stores.

```
/* all indications between diagonal stroke star and star diagonal stroke are only
comments * /
/ line comments are likewise possible
// all indications of a line following on the two diagonal strokes are comment

#include <avr/io.h>          // (1)

int main (void) {           // (2)

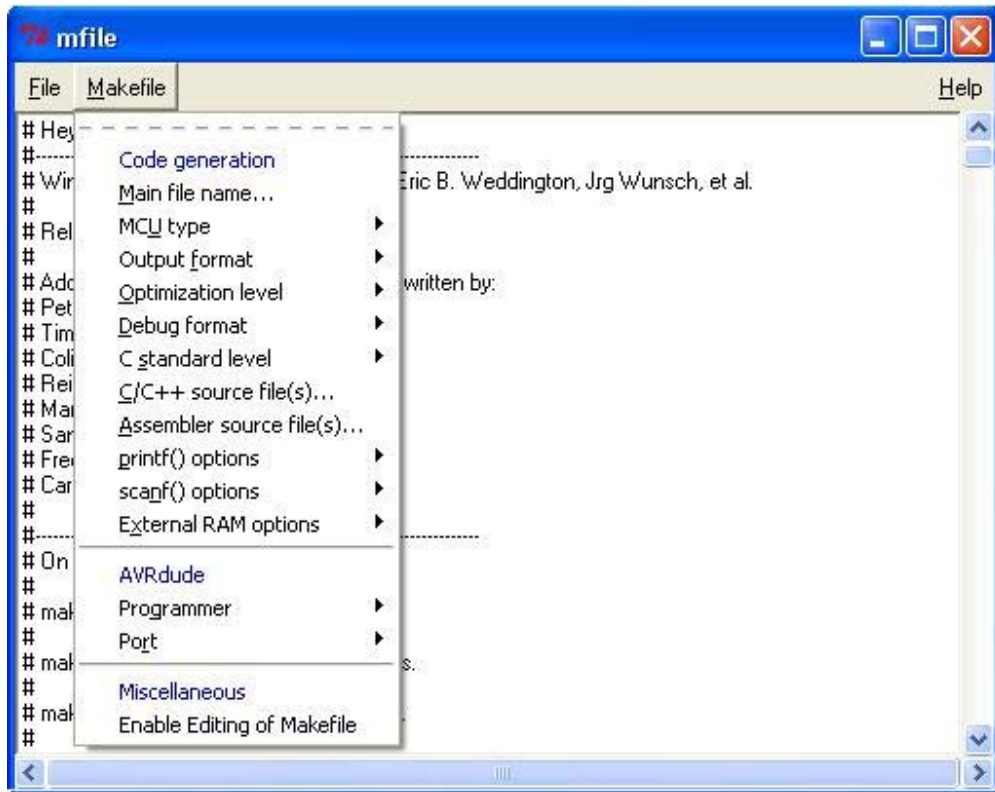
    DDRB = 0xff;             // (3)equal to DDRB=0b11111111
    PORTB = 0x03;            // (4)equal to PORTB=0b00000011

    while(1) {               // (5a)
        /* "empties " Loop*/; // (5b)
    }                         // (5c)

    /* never reached */
    return 0;                // (6)
}
```

- In line marked with (1), a so-called header file is merged. In io.h the register names are defined, which are used in the later process.
- The actual program begins with (2). Each C-program begins with the instructions in the function Main.
- The connections of an AVR (“Pins”) are combined into blocks; such a block is called one port. With the ATmega16 each port has 8 connections; with smaller AVR’s can have a port also with less than 8 connections be assigned. There by definition (data sheet) all set bits in a direction register make the appropriate connection as an output switch, all connections of the port B become with DDRB=0xff as outputs.
- (4) Initialized values of the outputs. Those first two bits of the port assigned port (PB0 and PB1) reach 1, all other connections of the port B (PB2-PB7) 0. If activated outputs (logic 1 or “high”) are on operating voltage (VCC, usually 5 V), non-activated outputs lead 0 V (GND, reference potential).
- (5) Is the major loop in such a way specified (Main loop). This is a program loop, which contains continuously returning instructions. In this example it is empty. The microcontroller goes through the loop again and again, without which something happened (except river one uses). Such a loop is necessary, since there is no operating system, which could take over control after completion of the program on the microcontroller. If the loop would be missing, the condition of the microcontroller became undefined after the program end (in another words the program crashes: TranslaToR cOMMENT)
- (6) Tells us that the program ends here. Contained only for reasons of C compatibility: int Main (void) means that the function returns a value. The instruction is however not reached, since the program never leaves the major loop.

In order to translate this source code into a program executable on the microcontrollers, a Makefile is used here. The used Makefile is on the side [example Makefile](#) and is based on the Makefile generator, the Makefile template in WinAVR is provided and was already adapted (for ATmega16 microcontroller). One can work on and adapt the Makefile to other microcontrollers or “together-click oneself” with the program **Mfile** which is contained in WinAVR Distribution (a Makefile menu controlled).



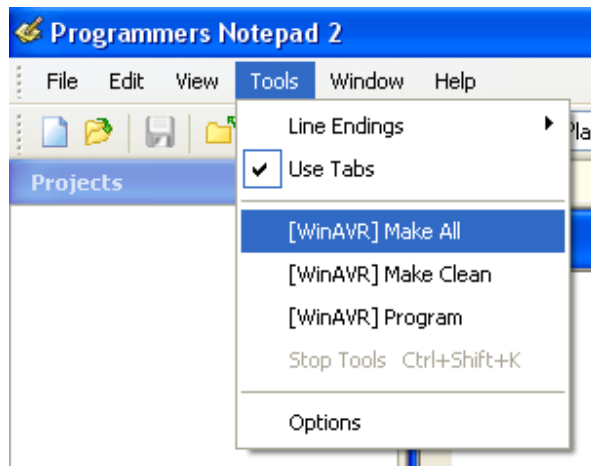
The Makefile is saved under the name Makefile (without extension i.e. .EXE for example) in the same folder into also the file main.c (“the program code”) is put down. More detailed explanations for the function of Makefiles are in the following to section [Exploring Makefiles](#).

```
D:\tmp\gcc_tut\quickstart>dir

quick start of D:\tmp\gcc_tut\quickstart

28.11.2006  22:53    <DIR>          .
28.11.2006  22:53    <DIR>          ..
28.11.2006  20:06                118 main.c
28.11.2006  20:03             16.810 Makefile
                2 files(en)             16.928 Bytes
```

Now one enters **make all**. If the Programmers Notepad installed with WinAVR is used, there is in addition a Menu option in the Tools menu. If all parameters are correct, a file develops main.hex, in which the code for the AVR is contained.



```
D:\tmp\gcc_tut\quickstart>make all

----- begin -----
avr-gcc (GCC) 3.4.6
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Compiling C: main.c
avr-gcc -c -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=1000000UL -Os -funsigned-char -f
unsigned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wundef
-Wa,-adhlns=obj/main.lst -std=gnu99 -Wundef -MD -MP -MF .dep/main.o.d main.c -
o obj/main.o

Linking: main.elf
avr-gcc -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=1000000UL -Os -funsigned-char -funs
igned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wundef -W
a,-adhlns=obj/main.o -std=gnu99 -Wundef -MD -MP -MF .dep/main.elf.d obj/main.o
--output main.elf -Wl,-Map=main.map,--cref -lm

Creating load file for Flash: main.hex
avr-objcopy -O ihex -R .eeprom main.elf main.hex

[...]
```

Contents of the hex file can be transferred now to the microcontroller. This can be made e.g. by in-system-Programming (ISP), described in the AVR [Tutorial: Equipment](#). Makefiles after the WinAVR/Mfile main information collecting are prepared for the use of the program [AVRDUDE](#). If the type and connection of the programming device is adjusted correctly, the transmission can be started with *make program* by means of AVRDUDE. Every other software, which can read hex files and to an AVR transfer (e.g. [PonyProg](#), yapp, AVRStudio), can be naturally likewise used.

If you started the microcontroller now (By Reset pin or Power supply on/off), the program will make PortB Pins PB0 and PB1 set to logic 1. One can measure the operating voltage with a measuring instrument now at this connection or let a LED shine (do not forget anode to the pin, pre-resistor). At the Pins PB2-PB7 one measures 0 V. An LED connected with the anode at one of these connections does not shine.



# Exploring Makefiles

If one is used to work with integrated development environments just as Visual C. The Makefile concept seems at first sight to be somewhat obscure. After short training this proceeding is however becomes very handy. These files (usual name: "Makefile" without file extension) serve the flow control of the program make, which should be installed on all Unix/Linux systems, and in a version for Ms-Windows also in [WinAVR](#)<sup>1</sup> (folder utils/bin) is contained. In the folder *sample in WinAVR* installation you will find one very useful Makefile template, which can be adapted simply to your own project ([local copy conditions September 2004](#))<sup>2</sup>. Alternatively one can use also [mfile](#)<sup>3</sup> of your desire. Mfile produces a Makefile after user selection or choices in the graphic user interface, it's however along-installed with WinAVR, & however as TCL/TK program which is executable on almost all platforms. ***The following remarks refer to the WinAVR example Makefile.***

There are three parameters in the Makefile which are handed over the shell and/or the Windows command prompt (cmd.exe/command.com) as parameters to "make". The program make looks for itself "automatically" the Makefile in the current work directory and accomplishes the operations for the appropriate call parameter, defined therein.

<i>make all</i>	Generates from the source codes a hex <i>file</i> (and if necessary also eep <i>file</i> ), indicated in the Makefile.
<i>make program</i>	The hex file (and alternatively also the eep file for the EEPROM) transferred to the AVR.
<i>make clean</i>	all temporary files, thus also the hex file are deleted

These calls can be merged into the very most editors in "Tool menus". This saves the contact with the command line. (With WinAVR are already in the Tools menu of the provided editor ***Programmers Notepad*** inserted calls.)

Usually the following data in the Makefile are to be adapted:

- Type of microcontroller
- Source coding files (C-files)
- Type and connection of the programming device

Rarer the following parameters are to be accomplished:

- Degree of the optimization
- Method for the production of the Debug symbols (Debug format)
- Assembler source coding files (S-files)

The Makefile cutouts shown in the following subsections are for a program, which is to be implemented on an ATmega8. The source code consists of the C-files *superprog.c* (in it Main ()), *uart.c*, *lcd.c* and *Iwire.c*. In the source code directory are these files: *superprog.c*, *uart.h*, *uart.c*, *lcd.h*, *lcd.c*, *Iwire.h*, *Iwire.c* and the Makefile (The adopted make file to suit our project)

---

<sup>1</sup><http://www.mikrocontroller.net/articles/WinAVR>

<sup>2</sup><http://www.mikrocontroller.net/wikifiles/b/b6/Makefile>

<sup>3</sup><http://www.sax.de/%7Ejoerg/mfile/> also included in the WinAVR distribution (BY THE TRANSLATOR)



The microcontroller is programmed by means of [AVRDUDE](#) over a [STK200-Programmierdongle](#) at the interface lpt1 (and/or /dev/lp0). In the source code also data for the *section* are *.eeprom* defined (see section of [memory accesses](#)), these are when programming equal also in the EEPROM to be written.

## Type of microcontroller set

In addition the “make variable” MCU is set according to the name of the used microcontroller. List of avr GCC and avr libc supported types is in the [documentation avr libc](#).

```
# comments in Makefiles begin...with the number sign("#")

# ATmega8 at work
MCU = atmega8
# or MCU = atmega16
# or MCU = at90s8535
# or ...
...
```

## Source coding files register

The name the source coding file which contains the function Main( ), is assigned to TARGET variable. This is however without the ending **.c**.

```
...
TARGET = superprog
...
```

If the project consists as in the example of more than one source coding file, the further C-files (*don't put the header files at here (INCLUDE files)*) only **.c** are to be assigned by blanks separately with SRC variable. The file defined with TARGET is already contained in the SRC list. Do not delete this entry!

```
...
SRC = $(TARGET).c uart.c lcd.c lwire.c
...
```

Alternatively one can extend the list of the source coding files also with the operator +=.

```
SRC = $(TARGET).c uart.c lwire.c
# LCD code for microcontroller xyz123 (out-commentated)
# SRC += lcd_xyz.c
# LCD code for "standard microcontroller" (used)SRC += lcd.c
```

## Programming device adjust

The collecting mains are adapted on the programming software [AVRDUDE](#), however also different programming software can be merged, if this can be steered over command line (e.g. stk500.exe, uisp, sp12).

```
...
# attitude for STK500 on com1 (out-commentated)
# of AVRDUDE_PROGRAMMER = stk500
# com1 = serial port. Use lpt1 to connect to parallel port.
# AVRDUDE_PORT = com1      # programmer connected to serial device

# attitude for STK200-Dongle on lpt1
AVRDUDE_PROGRAMMER = stk200
AVRDUDE_PORT = lpt1
...
```

If Flash (=hex) and EEPROM (=eep) are to be programmed together on the microcontrollers, the comment symbol is to be deleted forwards AVRDUDE\_WRITE\_EEPROM.

```
... # out-commentates: EERPOM contents not written #AVRDUDE_WRITE_EEPROM = - U EEPROM: w: $
(TARGET) .eep # does not out-commentate: EERPOM contents written AVRDUDE_WRITE_EEPROM = -
U EEPROM: w: $ (TARGET) .eep...
```

## Application

The provided Makefile and the code must be in the same file; also the file name should not be changed.

The input of *make all* in the work dierctory with the Makefile and the source coding files produces (among other things) the files *superprog.hex* and *superprog.eep*. Dependence between the individual C-files are considered automatically thereby. *Superprog.hex* and *superprog.eep* with *make* are transferred *program* to the microcontroller. With *make clean* are deleted all temporary files (= “cleared up”).

## Other parameters

### Optimization degree

The GCC compiler knows different stages of the optimization. Only to test purposes the optimization should be deactivated completely ( $OPT = 0$ ). The further possible options instruct the compiler to produce as compact or as fast a code as possible. Into that most cases  $OPT = s$  is by far the optimal (sic) attitude, thus more compactly and often also the “fastest” machine code is produced.

### Debug format

The formats of stabs and dwarf-2 are supported. The format is stopped behind  $DEBUG =$ . See in addition section *input files for simulation*.

### Assembler files

The assembler files used in the project are separately listed behind ASRC by blanks. Assembler files have always the ending. S (large S) . For example the assembler source code of a software UARTs is in a file *softuart. S* contained reads the line:  $ASRC = softuart. S$

### Clock frequency

Newer versions of the WinAVR/Mfile collecting main for Makefiles contain the definition of a variable  $F\_CPU$  (WinAVR 2/2005). In it the clock frequency of the microcontroller is registered in Hertz. The definition is

available then in the entire project likewise under the designation `F_CPU` e.g. to derive (in order from it UART, MIRROR-IMAGE or ADC frequency settings).

The indication has purely a “informative” character, which becomes actual clock rate over the external clock (e.g. quartz) and/or the attitude of the internal R/C oscillator determines. The use of `F_CPU` makes thus sense only if the indication agrees with the actual clock.

Within `avr libc` starting from version the 1.2 (contained in WinAVR starting from 2/2005) the definition of the clock frequency (`F_CPU`) is used for the computation of the wait functions in `avr/delay.h`. These function correctly only if `F_CPU` agrees with the actual clock frequency. `F_CPU` must be defined in addition however not necessarily in Makefile. It is sufficient, will however with repeated application unclearly to insert *UL* before `#include <avr/delay.h> a #define F_CPU [here clock in cycles per second]`. See in addition the [appropriate section of the documentation](#).

## Input files for simulation in AVR Studio

With older AVR Studio versions one can simulate only on basis of so-called *coff files*. Newer versions of AVR Studio (starting from 4.10.356) support besides the more modern however still experimental dwarf-2-Format, which is produced starting from WinAVR 20040722 (avr GCC 3.4.1/Binutils inclusive Atmel add ons) “directly” by the compiler.

### Proceeding with dwarf-2

- in the Makefile with `DEBUG`:

```
DEBUG=dwarf-2
```

- *make all* (possibly before *make clean*)
- the produced *elf-file* (in the example above *superprog.elf*) into AVR Studio load
- AVR simulator and to simulating microcontrollers select, “finish”
- further see AVR Studio on-line assistance

### Proceeding with extcoff

(Should be only used in exceptional cases)

- in the Makefile with `DEBUG`:

```
DEBUG=stabs
```

- *make extcoff* (possibly before *make clean*)
- the produced *cof-file* (in the example above *superprog.cof*) into AVR Studio load
- AVR simulator and to simulating microcontrollers select, “finish”
- further see AVR Studio on-line assistance

When simulating often “variables seem to be missing”. A cause for it is that the compiler assigns the “variables” directly to registers. This can be avoided, as the optimization is switched off (in Makefile). One simulated then however a program deviating strongly from the optimized code. Switching the optimization off is not recommended.

Instead of the software simulator the AVR Studio can be also used, in order to debug with the ATMEL JTAGICE, a reproduction of it (Boot ICE, Evertool.) or the ATMEL JTAGICE MKII “in system”. In addition

no special parameters are necessary in Makefile. Debugging and/or “in system emulation” with the JTAGICE and JTAGICE MKII is described in the AVR Studio on-line assistance.

The use of Makefiles offers still many far possibilities, some of it in the appendix [additional functions in the Makefile](#) is described.

## Integral (Integer) data types

With the programming of controllers the definition of some integral data types is meaningful, from which clearly the bit length can be read off.

Standardized data types are defined in the header file `stdint.h`. For the use of the standardized types one merges the “definition file” as follows:

```
// off avr libc version 1.2.0 possible and recommended:
#include <stdint.h>
// becomes outdated: #include <inttypes.h>
```

Some the types defined there (avr libc version 1.0.4):

```
typedef signed char      int8_t;
typedef unsigned char    uint8_t;

typedef short            int16_t;
typedef unsigned short   uint16_t;

typedef long              int32_t;
typedef unsigned long    uint32_t;

typedef long long         int64_t;
typedef unsigned long long uint64_t;
```

- `int8_t` stands for 8-bits for a Integer with a range of values -127 to 128.
- `uint8_t` stands for 8-bits for a Integer without signs (unsigned int) with a range of values from 0 to 255
- `int16_t` stands for a 16-Bit Integer with a range of values -32767 to 32768.
- `uint16_t` stands for a 16-Bit Integer without signs (unsigned int) with a range of values from 0 to 65536.

The types without placed in front *u* are stored signed numbers. Types with presented *u* serve the file of postiven numbers (inclusive 0).

See also: [Documentation avr libc the](#) section module it (standard) Integer of type

## Bit fields

When programming micro-control-learn must to each byte or even each bit be paid attention. Often we must store the condition 0 or 1 in a variable only. If we take to the smallest well-known data types, i.e. **unsigned char**, now for the storage of an individual value, then we waste 7 bits, since **unsigned char** an 8-bit is broad.

Here the programming language C offers a powerful tool, with whose assistance we can seize and (nearly to us) like 8 individual variables address 8 bits into an individual byte variable together. The speech is from so-called bit fields. These are defined as structural components. We regard in addition nevertheless best equal an example:

```
struct {
    Unsigned bStatus_1:1; // 1 bit for bStatus_1
    Unsigned bStatus_2:1; // 1 bit for bStatus_2
    Unsigned bNochNBit:1; // and here once again a bit
    Unsigned b2Bits:2;    // this field is 2 bits long
    // universe that has in only one byte variable place.
    // 3 the remaining bits remain unused
} x;
```

That access to such a field effected now as with the structure access admits over the point or the Dereferenzierungs operator:

```
x.bStatus_1 = 1;
x.bStatus_2 = 0;
x.b2Bits = 3;
```

However perhaps bit fields save place in the RAM, debited to of place in the Flash, worsen the Les and maintenance of the code. For beginners, a “whole” byte (uint8\_t) will turn out will use even if only one bit value to be stored is.

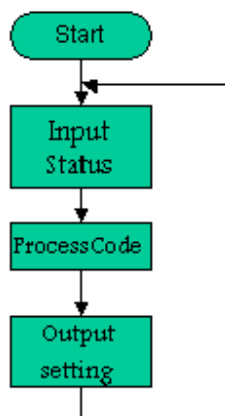
If one liked to use only a few variables of the type bool, one can merge also the header file <stdbool.h> and to be put on then as used a Booltyp. Variable this type need nevertheless 1 byte memory, make however an exact distinction possible between number variable and boolscher variable.

## Fundamental program structure of a $\mu$ C program

We differentiate between 2 different methods, in order to write a microcontroller program, completely independently of in which programming language the program is written.

### Sequential program sequence

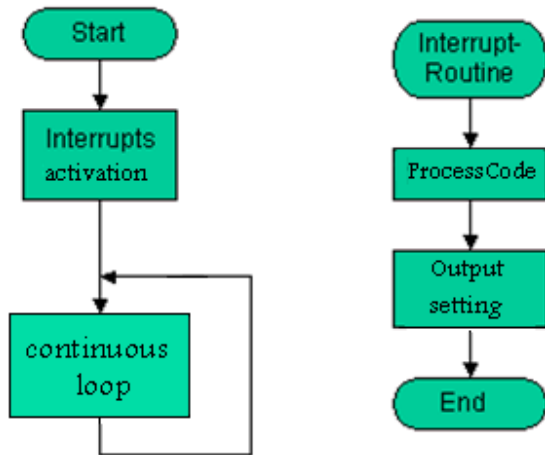
With this programming techniques a continuous loop is programmed, which has essentially always the same structure:



## Interrupt-controlled program sequence

With this method with the program start first the desired sources of interrupt are activated and then go into a continuous loop, in which things to be settled to be able, which are not time-critical.

If an interrupt is released automatically the assigned interrupt function is in such a way implemented.



## General accesses to registers

The AVR microcontrollers have a multiplicity of registers. Most of it are so-called write/read registers. That is, the program can pick out and describe contents of the registers both.

Some registers have special functions, others again can for general purposes (memory of data values) be used.

Individual registers are present with all AVRs, others again only with certain types. So are for example the registers, which are necessary for the access to the UART naturally only with those models available, which have integrated hardware a UART and/or a USART.

The names of the registers are defined in the header files to the appropriate AVR types. If in the Makefile the MCU type is defined, by the system to types fitting definition file is used automatically, as soon as one merges the general “io.h” header file in the code.

```
#include <avr/io.h>
```

If the MCU type is defined e.g. with the contents of atmega8 in the Makefile, when reading the io.h file in implicitly (“automatically”) also the iom8.h-file with the register definitions for the ATmega8 is read in.

## I/O register

The I/O registers have a special value with the AVR control-learn. They serve the access to Port and the interfaces of the microcontroller.

We differentiate between 8-bits and 16-Bit registers. For the time being we treat times the 8-bits registers.

Each AVR implements a different quantity of GPIO registers (GPIO - General Purpose Input/Output). The following examples proceed from an AVR, which possesses both Port A and Port B. Accordingly they must be adapted for other AVRs (for example ATmega8/48/88/168).

## Read an I/O register

For reading one can access I/O registers simply as a variable. In source codes, which were developed for older versions of the avr GCC/avr libc, the read access is made by the function `inp ()`. Current versions of the compiler enables the access now directly and `inp ()` are no longer necessary.

Example:

```
#include <avr/io.h>
#include <stdint.h>

uint8_t foo;

//...

int main(void)
{
    /* copies the status of the input pins at PortB
       Into the variable foo: */
    Foo = PINB;
    //...
}
```

## Read a bit

The AVR library (avr libc) places also functions to the inquiry of an individual bit of a register to the order:

### **bit\_is\_set (<Register>, <Bitnumber>)**

The function *bit\_is\_set* examines whether a bit is set. If the bit is set, a value is returned not equal 0. Exactly taken it is the priority of the queried bit, thus 1 for Bit0, 2 for Bit1, 4 for Bit2 etc.

### **bit\_is\_clear (<Register>, <Bitnumber>)**

The function *bit\_is\_clear* examines whether a bit is deleted. If the bit is deleted, thus on 0, a value is not equal 0 is returned.

The functions **bit\_is\_clear** and/or **bit\_is\_set** are not *necessary*, one can also “simple” C-syntax use, which is universally usable. *bit\_is\_set* e.g. corresponds thereby (register name & (1 << bit number)). The result is set to <>0 (“truly”) if the bit and 0 (“wrongly”) if it is not set. The denial of the expression, thus! (Register name & (1 << bit number)), corresponds *bit\_is\_clear* and returns a value to <>0 (“truly”), if the bit is not set,

- see also [bit manipulation](#)

## Write an I/O register

To the Write one can set I/O registers simply like a variable. In source codes, which were developed for older versions of the avr GCC/avr libc, the write access is made by the function `outp ()`. Current versions of the compiler support the access now directly and `outp ()` are no longer necessary.

Example:

```
Include <avr/io.h>

...

Int main(void)
{
    /* sets the direction register of the PORTA on 0xff
    (all pins as output): */

    DDRA = 0xff;

    /* PortA sets remaining on 0x03, bit 0 and 1 "high", "low": */
    PORTA = 0x03;
    ...
}
```

## Write of bits

One sets individual bits “standard C conformal” by means of more logically (bit) operations.

With the expression:

```
X |= (1 << Bitnumber); // a bit in x is set
X &= ~(1 << Bitnumber); // a bit is deleted in x
```

The least significant bit (for 1) of a byte has the bit number 0, the “most significant” (for 128) the number 7.

Example:

```
#include <avr/io.h>
...
#define MyBIT 2
...
PORTA |= (1 << MyBIT); // sets bit 2 at PortA on 1 * /
PORTA &= ~(1 << MyBIT); // deletes bit 2 at PortA */
```

With this method also several bits of a register can be at the same time set and reset.

Example:

```
#include <avr/io.h>
...
DDRA &= ~( (1<<PA0) | (1<<PA3) ); // PA0 and PA3 as inputs */
PORTA |= (1<<PA0) | (1<<PA3); // internal Pull UP for both switch on */
```

In source codes, which were developed for older version that of the avr GCC/avr libc, individual bits are set and/or reset by means of the functions **sbi** and **cbi**. Both functions are not necessary any longer.

See also:

- [Bit manipulation](#)
- [Documentation avr libc the](#) section Modules/Special Function of register



## Control rooms on a certain condition

There are functions, which wait in the library even, until a certain condition on a bit is reached. It is however normally a rather unpleasant programming techniques, since in these functions becomes “blocking waited”. That is called the program sequence stops here, until the masked event took place. If one uses the Watchdog, one must make certain that this is also still triggered (to put back the Watchdog timers).

The function **loop\_until\_bit\_is\_set** waits in a loop, until the defined bit is set. If the bit is already set with the call of the function, the function is again left immediately. The least significant bit has the bit number 0.

```
#include <avr/io.h>
...

/* control rooms to bit No. 2 (the third bit) in register PINA (1) */

#define WaitingPin PINA
#define WaitingBit PA2

// with avr libc the function is set:
loop_until_bit_is_set(WARTEPIN, WARTEBIT);

// meaning in "C-standard":
// goes through (empties) the loop the WAITING BIT in register waiting Pin
// _ NOT however _ so long 0 (thus 0) is not equal.
While ( !( WaitingPin & (1 << WaitingBit)) ) ;
...
```

The function **loop\_until\_bit\_is\_clear** waits in a loop, until the defined bit is deleted. If the bit is already deleted with the call of the function, the function is again left immediately. The least significant bit has the bit number 0.

```
#include <avr/io.h>
...

/* control rooms to bit No. 4 (the fifth bit) in register PINB (0) */
#define WaitingPin PINB
#define WaitingBit PB4

// avr libc function is deleted:
loop_until_bit_is_clear(WARTEPIN, WARTEBIT);

// meaning in "C-standard":
// goes through (empties) the loop the WAITINGBIT in register WaitingPin
// set (1) is so long
While ( WaitingPin & (1 << WaitingBit) ) ;
...
```

More universally and also to other platforms the use of C-Standard operators is better transferable.

See also:

- [Documentation avr libc the](#) section Modules/Special Function of register
- [Bit manipulation](#)

## Access to Port

All Port the AVR microcontroller is steered via registers. In addition registers are assigned to each Port 3:

<b>DDRx</b>	Data directions register for PORTx.  x corresponds to <b>A, B, C, D</b> etc. (dependent on the amount of the Port of the used AVR). Bit in the register set (1) for output, bit deleted (0) for input.
<b>PINx</b>	Input address for PORTx.  Condition of the Port. The bits in PINx correspond to the condition of the Port pins defined as input. Bit 1 if pin “high”, bit 0 if Port pin low.
<b>PORTx</b>	Data pointer for PORTx.  This register is used, in order to head for the outputs of a Port. With pins, which were switched by means of DDRx to input, the internal Pull UP of resistors can be activated or deactivated over PORTx (1 = actively).

## Data direction determination

First the data direction of the used pins must be determined. In order to reach this, the data direction register of the appropriate Port is described.

For each pin, which is to be used as output, thereby the appropriate bit must be set on the Port. If the pin is to be used as input, the appropriate bit must be deleted.

We want to define thus for example pin 0 to 4 of Port B as outputs to write in such a way we the following line:

```
#include <avr/io.h>

...

// setting the bits 0.1.2.3 and 4
// // binary 00011111 = hexadecimal 1F

DDRB = 0x1F;    /* direct assignment - unclearly */

/* more tapping work however more clearly: */
DDRB = (1 << DDB0) | (1 << DDB1) | (1 << DDB2) | (1 << DDB3) | (1 << DDB4);

/* some compilers permit the input of constants in the binary format (e.g. GCC in WinARM
starting from 1/2006 and in WinAVR, NOT however from GNU GCC sources themselves
erstellter compiler). These writing wise thus do not use, if code with others is to be
exchanged. */
DDRB = 0b00011111;    /* direct assignment
    - Clearer
    - Not standard
    - Conformal
    - Rarely portably
    - Only with modified GCC */

...
```

The pins 5 to 7 are switched (there 0) as inputs.

## Whole one of Port

In order to define a whole Port as output, the following instruction can be used:

```
DDRB = 0xff;
```

In the example the Port B is switched as a whole as output. In addition the header file must be merged `avr/io.h` (in it `DDRB` is defined among other things).

## Pre-defined bit numbers for I/O registers

The bit numbers (e.g. `PCx`, `PINCx` and `DDCx` for the Port C) are defined in the `io*.h`-files `avr/libc` and serve only the better legibility. One must `PAx`, `PBx`, `PCx` etc. these definitions not use or can also simply “always” use, even if that takes place access to bits in `DDRx` or `PINx` registers. For the compiler the expressions `(1<<PC7)` are identical, `(1<<DDC7)` and `(1<<PINC7)`: `(1<<7)` (more exactly: the Preprocessor directives replaces the expressions `(1<<PC7)`... too `(1<<7)`). A cutout of the definitions for Port C of a ATmega32 from the `iom32.h`-file to the elucidation (similar for further Port):

```
...
/* PORTC */
#define PC7      7
#define PC6      6
#define PC5      5
#define PC4      4
#define PC3      3
#define PC2      2
#define PC1      1
#define PC0      0

/* DDRC */
#define DDC7      7
#define DDC6      6
#define DDC5      5
#define DDC4      4
#define DDC3      3
#define DDC2      2
#define DDC1      1
#define DDC0      0

/* PINC */
#define PINC7     7
#define PINC6     6
#define PINC5     5
#define PINC4     4
#define PINC3     3
#define PINC2     2
#define PINC1     1
#define PINC0     0
```

# Digital signals

It is simplest to seize and/or spend digital signals with the microcontroller.

## Outputs

If one wants to set defined pins (appropriate DDRx bits = 1) as output on logic 1, one sets the appropriate bits in the Port register.

With the instruction

```
#include <avr/io.h>
...
PORTB = 0x04; /* better PORTB= (1<<PB2)*/*
```

The output is thus set at pin PB2 (consider that the bits are always counted *from 0*, the least significant bit is thus bit number 0 and not bit number 1).

One notes that for the assignment by means of = all pins are always indicated at the same time. One should read thus, if only certain outputs are to be switched, first the current value in of the Port and let the bit of the desired Port into this value flow. If one wants to set thus only the third pin (bit No. 2) at Port B on “high” and to leave the status of the other outputs unchanged, one uses this form:

```
#include <avr/io.h>
...
PORTB = PORTB | 0x04; /* better: PORTB = PORTB | (1<<PB2) */
/* simplifies by use |= of the operator:*/
PORTB |= (1<<PB2);

/* also several "at the same time":/
PORTB |= (1<<PB4) | (1<<PB5); /* pin PB4 and PB5 "high" */
```

A “switching off”, thus outputs on “low” set, affected similarly:

```
#include <avr/io.h>
...
PORTB &= ~(1<<PB2); /* bit resets 2 in PORTB and sets thereby for pin PB2 on low */
PORTB &= ~( (1<<PB4) | (1<<PB5) ); /* pin PB4 and pin PB5 "low" */
```

In source codes, which were developed for older version that of the avr GCC/avr libc, individual bits are set and/or reset by means of the functions sbi and cbi. Both functions are in current versions avr libc the no more contained and also not more necessarily.

*If the initial condition of outputs is critical, the sequence must be considered, with which the data direction (DDRx) is stopped and the expenditure value (PORTx) is set. For output pins, which are initialized with Initial value “high”, first the bits in the PORTx register set and afterwards the data direction on output place. From it the succession input results - > set PORTx: internal Pull UP actively - > set DDRx: Output “high”. With the order only DDRx and PORTx, can come it to a short “low pulse”, the also exterene of Pull UP resistors “outvoted”. (Unfavorable) the succession: Input - > set DDRx: Output (on “low”, there PORTx after RESET 0) - > set PORTx: Output on high. Comparisons to it also the data sheet section Configuring the pin.*

# Inputs (as signals come into $\mu\text{C}$ )

The digital input signals can arrive at different kinds at our logic.

## Signal coupling

It is simplest, if the signals can be taken over directly from another digital circuit. The output of the appropriate circuit TTL level knew then we even directly the output of the circuit with an input pin of our microcontroller to connect.

The output of the other circuit a TTL level did not have in such a way we the level over appropriate hardware (e.g. optocoupler, [voltage divider](#), "Level shifter" aka [level transducers](#)) to adapt.

The mass of the two circuits must be interconnected naturally. It is naturally finally no matter to the software, how the signal is fed. We can examine anyway only whether against a pin of our microcontroller a logic 1 (tension more largely approx.  $0,7 \cdot V_{cc}$ ) rests or logic 0 (tension smaller approx.  $0,2 \cdot V_{cc}$ ). Detailed information about it, starting from which tension an input as 0 ("low") and/or 1 ("high") is recognized, the table DC Characteristics supplies in the data sheet of the used microcontroller. The inquiry of the conditions of the Port pins is made directly by the register name. **About is important to use for the inquiry of the inputs *not* Port register **PORTx** but to input register **PINx** The inquiry of the pin conditions over PORTx instead of PINx is a frequent error with the AVR "first contact".** (Otherwise one reads not the condition of the inputs, but the status of the internal Pull UP resistances.) If one wants to query signal statuses of Port D and into a variable named bPortD store current, one writes the following command lines:

```
#include <avr/io.h>
#include <stdint.h>
...
uint8_t bPortD;
...
bPortD = PIND;
...
```

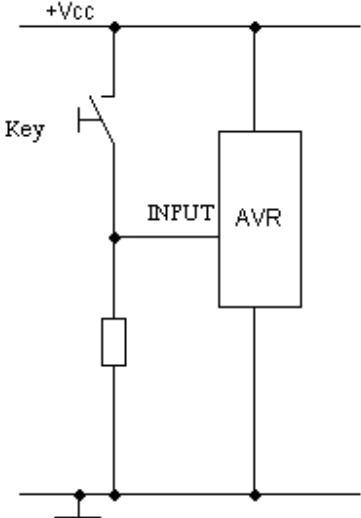
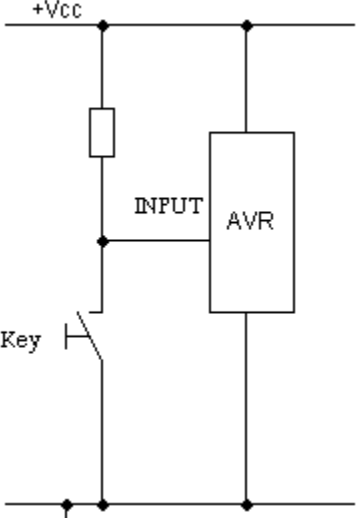
With the C-Bit operation one can query the status of the bits.

```
#include <avr/io.h>
...
/* implements action, if bit No. 1 (the "second" bit) is set in PINC (1) */
if ( PINC & (1<<PINC1) ) {
    /* action */
}

/* implements action, if bit No. 2 (the "third" bit) is deleted in PINB (0) */
if ( !(PINB & (1<<PINB2)) ) {
    /* action */
}
...
```

## Keys and switches

The connection of mechanical contacts at the microcontrollers becomes likewise completely simple, whereby we must differentiate between two different methods (*Active Low* and *Active High*):

Active High	Active Low
	
<p>Here the contact is switched between supply voltage and the input pin.</p> <p>So that during open switching position no undefined signal at the microcontroller is applied, between the input pin and the mass a Pull down resistance is switched. This serves to hold the level during opened switching position logic 0.</p>	<p>With this method the contact is switched between the input pin of the microcontroller and mass.</p> <p>Thus with open switch the microcontroller an undefined signal does not get between supply voltage and the input pin so-called Pull UP a resistance is switched. This serves to draw the level with opened switch on logic 1.</p> <p>The resistance value of the Pull UP of resistance is not critical actually. If it is however too highly selected, the effect is possibly not given. As usual value 10 Kilo ohms were in- practice.</p> <p>The AVR's has even at most pins by software insert able internal Pull UP of resistances, which we can naturally use.</p>

### Pull UP resistors activation

The internal Pull UP of resistors from Vcc to the individual Port pins are activated and/or deactivated via register **PORTx**, if a pin is switched as **input**.

If the value of the appropriate Port pin is set to 1, then the Pull UP resistance is activated. At a value of 0 the Pull UP resistance is not active. One should use in each case either the internal or external Pull UP a resistance, but both together.

In the example all pins of the Port D are switched as inputs and all Pull UP of resistors are activated. Further pin PC7 is switched as input and its internal Pull UP resistance is activated, without changing the parameters for the other Port pins (PC0-PC6).

```
#include <avr/io.h>
...
DDRD = 0x00; /* all pins von Port D as input */
PORTD = 0xff; /* internal Pull Ups at all Port pins activated */
...
DDRC &= ~(1<<DDC7); /* Pin PC7 als Eingang */
PORTC |= (1<<PC7); /* internal Pull UP at PC7 activated */
```

## Debouncing inputs

Now all mechanical contacts have, are it from switches, buttons or also from relays to bounce the unpleasant characteristic. This means that when closing the contact the same manufactures not directly contact, but several times switches on and off up to final manufacturing of the contact.

It is to be counted now with a fast microcontroller how often such a contact is switched, then we have a problem, because bouncing is counted as repeated impulses. And this phenomenon Must be absolutely considered during the writing of the program.

```
#include <avr/io.h>
#include <inttypes.h>
#ifndef F_CPU
#define F_CPU 3686400UL /* quartz with 3,6864 MHz */
#endif
#include <avr/delay.h> /* defines _delay_ms () off avr libc version 1.2.0 */

/* simple function to denounce a button */
inline uint8_t debounce(volatile uint8_t *port, uint8_t pin)
{
    if ( ! (*port & (1 << pin)) )
    {
        /* pin was pulled on mass, 100ms waits */
        _delay_ms(100);
        if ( *port & (1 << pin) )
        {
            /* give user time for releasing the button */
            _delay_ms(100);
            return 1;
        }
    }
    return 0;
}

int main(void)
{
    DDRB &= ~( 1 << PB0 ); /* pin PB0 on input (button)*/
    PORTB |= ( 1 << PB0 ); /* Pull-up resistance activate */
    ...
    if (denounce(&PINB, PB0)) /* case button pressed to pin PB0..*/
        PORTD = PIND ^ ( 1 << PD7 ); /* .. LED at Port PD7 on and/or switch off */
    ...
}
```

With this example it is to be considered that the AVR waits in case of a depressing the key 200ms, therefore fallow lies. Time-critical applications should select another procedure.

To the topic debouncing see also:

- Article [Debouncing](#)

## Analogue

The processing of analog input values and the expenditure of analog values in chapters [analog input and output](#) are treated.

## 16-Bit Port registers (ADC, ICR1, OCR1, TCNT1 and UBRR)

Some of the Port registers in the AVR Controller are 16 bits long. In the data sheet these registers are usually with the suffix “L” (LSB) and “H” (MSB) provide. Avr libc additionally most define this variables the designation without “L” or “H”. These can be assigned and/or accessed directly. The conversion from 16-bit word after 2\*8-bit byte takes place internally.

```
#include <avr/io.h>
#include <stdint.h>
...
uint16_t foo;

foo=ADC; /* the word variable sets foo to the value of the last AD-transformation*/
```

If needs, a 16-Bit variable can be divided also quite simply manually into its two 8-bits of components. The following example demonstrates this on the basis the pseudo 16-Bit of register UBRR.

```
#include <avr/io.h>
#include <stdint.h>
#ifndef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

typedef union {
    uint16_t i16;
    struct {
        uint8_t i8l;
        uint8_t i8h;
    };
} convert16to8;

...
convert16to8 baud;
baud.i16 = F_CPU / (UART_BAUD_RATE * 16L) - 1;
UBRRH = baud.i8h;
UBRRL = baud.i8l;
...

/*alternatively:*/
#include <inttypes.h>

uint16_t wFoo16;
uint8_t bFooLow, bFooHigh;

...

wFoo16 = 0xAA55; /* "dividing" 16Bit-Integer */
```



```
bFooHigh = (uint8_t)(wFoo16 >> 8); /* MS-Byte */
bFooLow  = (uint8_t)(wFoo16);      /* LS-Byte */
...
```

With some AVR types (e.g. ATmega8) UBRRH and UCSRC the same MEMORY address divide. So that the AVR can differentiate nevertheless between the two registers, the Bit7 (URSEL) determines be actually described which register is. *1000 0011* (0x83) addressed therefore UCSRC and the value 3 and *0000 0011* (0x3) addresses UBRRH hands over to 3 and hands likewise the value over.

With some 16-bit registers (in particular those 16-bit timer) write accesses are made by the so-called *temporary register*. The sequence of the accesses determines, when the value is actually written in the register. Typically only the High byte is described (xxxH) and internally in temporary register buffered. After the Low byte (xxxL) was written, the microcontroller with this sets and in temporary the register of buffered value for the High byte the 16-bit register. It is to be noted that internally only temporary a register is available, which in interrupt routines mglw. With another value one overwrites, if 16-bit registers are likewise described there. avr gcc/avr libc, if the registers with its “16-bit label” (without H and/or L) are addressed, is automatic the protection temporary of the register from overwriting considers the correct order by interrupt routines to consider nevertheless (in the doubt with the write access the interrupts briefly global deactivate).

In handling 16-Bit registers see also:

- [Documentation avr libc the](#) section Related Pages/Frequently Asked Questions/No. 8
- Data sheet section *Accessing 16-bit of register*

## I/O registers as parameters and variables

In order to be able to hand registers over as parameters for own functions, one must hand it as a volatile uint8\_t to pointers over. For example:

```
uint8_t key_pressed(const volatile uint8_t *inputreg, uint8_t inputbit)
{
    static uint8_t last_state = 0;

    if ( last_state == ( *inputreg & (1<<inputbit) ) ) {
        return 0; /* no change */
    }

    /* if nevertheless, wait to any bouncing past is: */
    _delay_ms(20);

    /* condition for next call notice: */
    last_state = ( *inputreg & (1<<inputbit) );

    /* depressing and the key return: */
    return ( *inputreg & (1<<inputbit) );
}

/* example of a function call: */

#include <avr/io.h>

uint8_t i;

//...
i = key_pressed( &PINB, PB1 );
//...
```

A call of the function with call by VALUE would cause the following: With the function input only one copy of the momentary Port condition is made, which does not change independently of the actual condition Port any longer, with which the function would be ineffective. The delivery of a pointer would be the solution, if the compiler would not optimize. Because one reads thus in the program not by the hardware, but again only by an image in the memory. The result would be the same as above. With the keyword volatile one says now to the compiler that the appropriate variable can be changed either by other software routines (interrupts) or by the hardware.

In all other respects volatile characterized variables can be defined also as const, in order to guarantee with that they are alterable by the hardware only.

# The UART

## General to the UART

By the [UART](#) an AVR can be connected easily with a [RS-232-Schnittstelle](#) PC or other devices with “serial interface”.

Possible applications of the UART:

- Debug interface: e.g. to the announcement of intermediate results (“printf debugging” - here better “UART debugging”) on a PC. On the computer in addition a terminal software (Ms-Windows is enough: HyperTerminal or better Bray terminal, [HTerm](#); Unix/Linux e.g. minicom). A direct connection is not possible due to different levels, however appropriate interface ICs is as to be integrated e.g. a MAX232 favorably and easy. Computers without serial interface can be attached over finished USB serial adapters.
- “Man-machine interface”: e.g. configuration and status inquiry over a “command line” or menu
- Transferred by stored values: e.g. with a Data logger
- Connection of devices with serial interface (e.g. (radio) modem, mobile telephones, printer, sensors, “intelligent” LC-displays, government inspection department receivers).
- “Field penalty” on RS485/RS422-Basis by means of appropriate bus driver components (e.g. MAX485)
- DMX, Midi etc.
- LIN bus (**local Interconnect network**): Inexpensive sensors/actuators in the automotive engineering and beyond that

Some AVR microcontrollers already inserted to two full-duplexable UART (**universal Asynchronous Receiver** and **transmitter**) (“hardware UART”). By the way: Full-duplex is called nothing else, the component can send and receive at the same time.

Newer AVR (ATmega, ATtiny) has two USART (s) or, these differ from the UART mainly by internal FIFO buffers for input and output and extended configuration options. The buffer size is however only 1 byte.

The UART is addressed via four separate registers. USA RTS of the ATMEGAs have several additional configuration registers. The data sheet gives over it information. The following table shows only the registers (that became outdated) the UARTs.

## UART Control Register.

In this register we adjust, as we would like to use the UART .The register is developed as follows:

0	1	2	3	4	5	6	7	Bit
<b>TXB8</b>	<b>RXB8</b>	<b>CHR9</b>	<b>TXEN</b>	<b>RXEN</b>	<b>UDRIE</b>	<b>TXCIE</b>	<b>RXCIE</b>	<b>Name</b>
W	R	R/W	R/W	R/W	R/W	R/W	R/W	<b>R/W</b>
0	1	0	0	0	0	0	0	<b>Initial value</b>

### **RXCIE (RX Complete interrupt Enable)**

If this bit is set, a UART RX Complete interrupt is released, if an indication became to receive from the UART. Global the Enable interrupt flag must be naturally also set.

### **TXCIE (TX Complete interrupt Enable)**

If this bit is set, a UART TX Complete interrupt is released, if an indication were sent by the UART. Global the Enable interrupt flag must be naturally also set.

### **UDRIE (UART DATA register Empty interrupt Enable)**

If this bit is set, a UART data pointer is released empty interrupts, if the UART is again ready around a new indication which can be sent to take over. Global the Enable interrupt flag must be naturally also set.

### **RXEN (Receiver Enable)**

Only if this bit is set, the receiver of the UART works at all. If the bit is not set, the appropriate pin of the AVR can be used as normal I/O pin.

### **TXEN (transmitter Enable)**

Only if this bit is set, the transmitter of the UART works at all. If the bit is not set, the appropriate pin of the AVR can be used as normal I/O pin.

### **CHR9 (9 bits of character)**

If this bit is set, 9 bits long indications can be transferred and received. 9. Bit can be used if necessary as additional stop bit or as parity bit. One speaks then of a 11-bit indication framework: *( 1 start element + 8 data bits + 1 stop bit + 1 parity bit = 11 bit)*

### **RXB8 (Receive DATA bit 8)**

If the above-mentioned CHR9-Bit is set, then this bit contains of 9. Data bit of a received indication.

### **TXB8 (Transmit DATA bit 8)**

If the above-mentioned CHR9-Bit is set, then must into this bit 9. Bit of the indication which can be sent to be written before the actual data byte into the data pointer is written.

UCR

## UART Status Register.

Here the UART communicates us, which it makes in such a way straight.

0	1	2	3	4	5	6	7	Bit
-	-	-	OR	FE	UDRE	TXC	RXC	Name
R	R	R	R	R	R	R/W	R	R/W
0	0	0	0	0	1	0	0	Initial value

### RXC (UART Receive Complete)

This bit is set by the AVR, if a received indication were transferred by the receipt shift register into the receive data register.

The indication must be picked out now as fast as possible from the data pointer. If this affected before a further indication completely to receive became an overflow error situation will arrive. With the selection of the data pointer the bit is deleted automatically.

### TXC (UART Transmit Complete)

This bit is set by the AVR, if in the transmission shift register indications present completely one spent and no further indication in the transmit data register lines up. This means thus, if communication is at full extent final.

This bit is important with half duplex connections, if the program must switch after sending data to receipt. In the full-duplex enterprise we do not need to consider this bit.

The bit is deleted automatically only then, if the appropriate Interrupt handler is called, otherwise must we the bit delete.

### UDRE (UART DATA register Empty)

This bit indicates whether the send buffer is ready, in order to take up an indication which can be sent. The bit is set by the AVR (1), if the send buffer is empty. It is deleted (0), if an indication is present in the transmit data register and were not transferred yet to the transmission shift register. Atmel recommends for compatibility reasons with coming  $\mu C$  to set UDRE to 0 if the UCSRA register is described.

The bit is deleted automatically, if an indication is written into the transmit data register.

### FE (Framing error)

This bit is set by the AVR, if the UART detects an indication framework error, i.e. if the stop bit of a received character 0 is.

The bit is deleted automatically, if the stop bit of the received character is 1.

### OR (Overrun)

This bit is set by the AVR, if our program does not fetch the indication lying ready in the receive data register before the following indication completely to receive became. The following indication is rejected.

The bit is deleted automatically, if the received indication could be transferred into the receive data register.

USR

<b>UDR</b>	<p><b>UART DATA Registers.</b></p> <p>Here data are transmitted between UART and CCU. Since the UART in the full-duplex enterprise can receive and send at the same time, it concerns here physically 2 registers, which are addressed however via the same I/O address. According to whether a reading or a write access on the UART taken place the correct UDR is addressed automatically.</p>
<b>UBRR</b>	<p><b>UART Baud Rate Register.</b></p> <p>In this register we must communicate to the UART, how quickly we would like to communicate gladly. The value, which must be written into this register, is calculated according to the following formula:</p> $\mathrm{UBRR} = \frac{\mathrm{Taktfrequenz}}{\mathrm{Baudrate}} \cdot 16 - 1$ <p>There is more highly possible for Baud rates up to 115200 Baud and.</p>

## The hardware

The UART is based on normal TTL level with 0V (LOW) and 5V (HIGH). The interface specification for RS-232 defines however -3V... -12V (LOW) and +3... +12V (HIGH). Besides the signal exchange between AVR and partner equipment must be inverted. For the adjustment of the levels and inverting the signals there are finished interface modules. The most well-known of it is probably the MAX232.

If communication strikes by UART, then an incorrect attitude of the Baud rate is often the cause. The configuration on a certain Baud rate depends on the clock frequency of the microcontroller. Straight ones with again developed circuits (and/or again bought control-learn) one should therefore make sure again that the microcontroller also actually works with the assumed clock rate and not e.g. uses the internal [oscillator](#) adjusted factory-installed with some models instead of an external quartz. The values of the different fuse bits in the event of an error thus for example with [AVRDUDE](#) control and if necessarily adapt. In principle also always a view is recommended into the [AVRChecklist](#).

## Send with the UART

We want to spend now data with the UART on the serial interface.

In addition we must initialize the UART first times. In addition we set the necessary bits in the **UART control register** depending upon desired function mode.

Since we would like to send only and want (still) no interrupts to evaluate for the time being, turn out the initialization really very simply, since we only the **Enable** bit **transmitters** must set:

```
UCR |= (1<<TXEN);
```

Newer AVR's with USART (s) has several configuration registers and to require something other configuration. For a ATmega16 e.g.:

```
UCSRB |= (1<<TXEN); // UART TX switch
UCSRC |= (1<<URSEL) | (3<<UCSZ0); // asynchronous 8N1
```

Now is still that and/or is still the Baud rate registers to adjust (UBRR and/or UBRRH/UBRRH). The value for it results from the indicated formula from using the clock frequency and the desired data transmission rate. (One knows the calculation of the formula the [Preprocessor directives](#) left).

```
/* UART INIT with at90S2313 */

#ifndef F_CPU
/* in newer version of the WinAVR/Mfile Makefile collecting main can be
   defined F_CPU in the Makefile, a repeated definition to a compiler warning
   Will lead here. Therefore "protection" through #ifndef/endif */
/* for example 4 MHz-Quartz (if not already in the Makefile defines): */
#define F_CPU 4000000
#endif
#define UART_BAUD_RATE 9600 // 9600 Baud

int main(void)
{
    UBRR = F_CPU / (UART_BAUD_RATE * 16L) - 1;
    //...
}
```

Again for the Mega16 with a 16bit-Register something other programming.

```
/* USART blank INIT with the ATmega16 */
#ifndef F_CPU
#define F_CPU 3686400 // Oscillator-Frequency in Hz */
#endif

// auxiliary macro for UBRR computation ("formula" according to data sheet)
#define UART_UBRR_CALC(BAUD_,FREQ_) ((FREQ_)/((BAUD_)*16L)-1)

#define UART_BAUD_RATE 9600

int main(void)
{
    UBRRH = (uint8_t)( UART_UBRR_CALC( UART_BAUD_RATE, F_CPU ) >> 8 );
    UBRRH = (uint8_t)UART_UBRR_CALC( UART_BAUD_RATE, F_CPU );

    /* alternatively with avr libc "directly the 16bit" : */
    UBRR = UART_UBRR_CALC( UART_BAUD_RATE, F_CPU );
    //...
}
```

Partly one can look up the appropriate value in the data sheet of the respective CCU also. An example of a ATmega32 with 16MHz and of 19200 Baud: in the data sheet of the ATmega32 table “Examples OF UBRR Settings” one reads off the value 51 for these defaults. This is distributed now on registers UBRRH and UBRRH. The initialization for the USART could look then in such a way:

```
int main(void)
{
    /* USA blank INIT 19200 Baud with 16MHz for Mega32 */
    UCSRB |= ( 1 << TXEN ); // UART TX switch
    UCSRC |= ( 1 << URSEL ) | ( 3 << UCSZ0 ); // asynchrone 8N1
    UBRRH = 0; // High byte is 0
    UBRRH = 51; // Low byte is 51 (decimally)
    //...
}
```

The above code is #define in relation to the versions with “...” not so portably, but however somewhat more clearly.

## Send individual indications

In order to send now an indication on the interface, we must write the same only into the **UART DATA registers**. Before it is to be examined whether the UART module is ready the indication which can be sent to receive. The designations/that of status register with the bit UDRE depends on microcontroller types (see data sheet).

```
// with AVR with a UART ("classic AVR" e.g. AT90S8515)
while (!(USR & (1<<UDRE))) /* waits for possible sending */
{
}

UDR = 'x'; /*the indication x writes */

/** OR **/

// with newer AVRs stands the status in UCSRA/UCSR0A/UCSR1A, here e.g. for ATmega16:
while (!(UCSRA & (1<<UDRE))) /* waits for possible sending */
{
}

UDR = 'x'; /*the indication x writes on the interface */
```

## Write to a character string (stringer)

The task “stringer send” by two functions one processes. The universal/microcontroller-independent function `uart_puts` hands in each case an indication to the character string over to a function `uart_putc`, which must be implemented dependent on the existing hardware. In the function for sending an indication is to be made certain that before sending it is examined whether the UART is ready the “transmission job” to receive.

```

// putc for AVR with a UART (e.g. AT90S8515)
int uart_putc(unsigned char c)
{
    while(!(USR & (1 << UDRE))) /* wait, to UDR ready */
    {
    }

    UDR = c; /* send to indications */
    return 0;
}

/** OR */

// with newer AVRs other name for the status registers, here ATmega16:
int uart_putc(unsigned char c)
{
    while (!(UCSRA & (1<<UDRE))) /* waits for possible sending */
    {
    }

    UDR = c; /* send to indications */
    return 0;
}

/* PUTs is independently of the type of microcontroller */
void uart_puts (char *s)
{
    while (*s)
    { /* so long *s! = "\ 0" thus unequally the "string end-character" */
        uart_putc(*s);
        s++;
    }
}

```

Waiting loops are to that extent somewhat critical, since during sending a string no more will not react to other events can. The use of FIFO (first in first out) is more universal - buffers, in which and/or the received indications/byte buffered and by means of interrupt routines at the U (S) KIND which can be sent are passed on and/or selected. In addition finished components (libraries, Libraries) exist, which one can integrate quite simply into own developments. It is advisable to use these components and not to invent the wheel again.

## Write of variables' contents

If contents of variable (integers, floating point) in “more human being-readably form” are to be sent, a transformation is necessary in indication (“ASCII”) before the transfer. With only one number this transformation is relatively simple: one adds the ASCII value from zero to the number and can this value directly send.

```

///...

// here uart_putc (s.o.)
// expenditure of 0123456789
char c;

for (uint8_t i=0; i<=9; ++i) {
    c = i + '0';
    uart_putc( c );
    // shortens: uart_putc (i + "0");
}

```



If more than one number is to be spent, one avails oneself of appropriately existing functions for the transformation of numbers in character strings/stringers. The function *avr libc* for the transformation of signed 16bit-Ganzzahlen (*int16\_t*) in character strings is called *itoa* (Integer ton of ASCII). One must the function a storage area for processing (more buffers) with place for all numbers, which stringer end-characters (“\0”) and possibly the sign make available.

```
#include <stdlib.h>

//...

// here uart_init, uart_putc, uart_puts (s.o.)

int main(void)
{
    char s[7];
    int16_t i = -12345;

    uart_init();

    itoa( i, s, 10 ); // 10 for radix - > decimal system

    uart_puts( s );

    // itoa a pointer on the beginning of s returns there shortened also:
    uart_puts( itoa( i, s, 10 ) );

    while (1) {
        ;
    }

    return 0; // never reached
}
```

For unsigned 16bit-Ganzzahlen (*uint16\_t*) existed *utoa*. The functions for 32bit-Ganzzahlen (*int32\_t* and *uint32\_t*) are called *ltoa* and/or *ultoa*. Since 32bit-Ganzzahlen can exhibit more places, an accordingly larger buffer memory is to be planned.

Also floating point numbers (float/doubles) can be converted with breits existing functions into character sequences, in addition the functions *dtostre* exist and *dtostrf*. *dtostre* uses exponential way of writing (“engineering” - format). (Reference: z.Zt. existed counted in the *avr GCC* no “genuine” double, internally always on “simple accuracy”, accordingly float.)

*dtostrf* and *dtostre* need *libm.a* *avr libc*. In use of Makefiles the parameter is *-lm* in in *LDFLAGS* to indicate (standard in the *WinAVR/mfile* Makefile were present). One uses *AVRStudio* as IDE for the GNU compiler (*GCC Plugins*) is *libm.a* under Libraries to be selected: Project - > Configurations option - > Libraries - merge to the right > *libm.a* with the arrow.

```

#include <stdlib.h>

//...

// uart_init here, uart_putc, uart_puts (s.o.)

/* lt. avr-libc documentation:
char* dtostrf(
    double __val,
    char __width,
    char __prec,
    char * __s
)
*/

int main(void)
{
    // buffer memory sufficiently largely
    // sign + width + end-characters:
    char s[8];
    float f = -12.345;

    uart_init();

    dtostrf( f, 6, 3, s );
    uart_puts( s );

    // shortens: uart_puts (dtostrf (f, 7, 3, s));

    while (1) {
        ;
    }

    return 0; // never reached
}

```

See also:

- [Documentation avr libc/stdlib.h](#)
- [The use of printf](#)

## Receiving Indication

To the receipt of indications the receiving element of the UART must be activated with the initialization, as the RXEN bit in the respective configuration register (UCSRB and/or UCSR0B/UCSR1B) is set. Waited in the simplest case so long, until an indication became to receive, this is located then in the UART data pointers (UDR and/or UDR0 and UDR1 with AVRs with 2 UARTS) for the order (suctions. “Polling enterprise”). An example of the ATmega16:

```

#include <inttypes.h>
#include <avr/io.h>

/* in addition to the Baud rate attitude and the further initialization: */
void Usart_EnableRX()
{
    UCSRB |= ( 1 << RXEN );
}

/* indications receive */
uint8_t Usart_Rx(void)
{
    while (!(UCSRA & (1<<RXC))); // wait for available indications
    return UDR;                  // indication from UDR at Call return }

```

This function blocks the program sequence. Alternatively the RXC bit in a program loop can be queried and be picked out then only with set RXC bit UDR. More elegantly and in most applications the proceeding is “more stable” to read in and buffer for later processing in an Eingangs buffer (FIFO Buffer) the received indications in an interrupt routine. In addition manufacture and well tested [libraries](#) and source code components (e.g. UART LIBRARY of P. Fleury, Procyon avrlib and some in the “Academy” of avrfreaks.net) exist.

TODO: 9bit

## Software UART

If the number of the existing hardware UARTs is not sufficient, further interfaces can be supplemented over so-called software UARTs. There are in addition (at least) two beginnings:

- The beginning most usual with AVRs is based on the principle that an external interrupt pin for the receipt (“RX”) is used. The start element releases the interrupt, in which interrupt routine (ISR) the external interrupt deactivated and a timer activated. In the interrupt routine of the timer the condition of the receipt pin is scanned according to the Baud rate. After receipt of the stop bit the external interrupt is again activated. To send can be made by any pin (“TX”), which is set according to the Baud rate and the indication which can be sent to 0 or 1. The implementation is not completely simple, it exists in addition however finished libraries (e.g. with [avrfreaks](#) or in the [Procyon avrlib](#)).
- A further beginning does not require a pin with “interrupt function” however needs more computing time. Each input pin can serve as receipt pin (RX). Over a timer the condition of the RX-pin with one is scanned the multiples Baud rate (three-way seems usual) and High and/or Low bits identified on the basis a minimum number. (Example: “Gene Eric software Uart” Application note of IAR)

Newer AVRs (e.g. ATtiny26 or ATmega48,88,168,169) has universal a Serial interface (USI), which can take over partial UART function. Atmel makes a Application note available, in which the use of the USI is described as UART (in principle “hardware supported software UART”).

- See also: Resuming information inclusive examples of the use of stdio functions (printf etc.) in the AVR [Tutorial - UART](#).
- [Peter Fleurys UART](#) Bibliotheca for avr gcc/avr libc

# Analog input and output

Analog input values are usually read in over the AVR analog-digital converter (AD-transducer, ADC), which is available in many types (typically 10bit dissolution). Analog signals (tensions) are changed by this into digital numerical values. With AVRs, which have no internal AD-transducer (e.g. ATmega162, ATtiny2313), can by external wiring (R/C network and “time measurement”) the function of the AD-transducer become “emulated”.

No AVRs with inserted digital analog converter (DAC) exists. This function must be copied by external components (e.g. PWM and “smoothing”).

Independently of it the possibility naturally always exists of using special components for the analog-digital and/or digital analog transformation and of heading for these over a digital interface (e.g. MIRROR-IMAGE ONE or I2C) with a AVR.

## ADC (analog digital converter)

The analog-digital converter (ADC) converts analog signals into digital values, which can be interpreted by the microcontroller. Some AVR types already inserted a more-channel analog-digital converter. The accuracy, with which a analog signal can be dissolved, is indicated by the dissolution of the ADC in number of bits, one hears and/or reads in each case from 8-Bit-ADC or 10-Bit-ADC or still more highly.

A ADC with 8-bit dissolution can represent thus the analog signal with an accuracy of  $1/256$  of the maximum value. If we assume now times, we would have a tension between 0 and 5 V and a dissolution of 3 bits, then the values 0V, 0.625V, 1,25, 1.875V, 2.5V, 3.125V, 3,75, 4,375, 5V could come along, see in addition the following table:

Appropriate measured value	Input voltage at the ADC
0	0... <0.625V
1	0.625... <1.25V
2	1.25... <1.875V
3	1.875... <2.5V
4	2.5... <3.125V
5	3.125... <3.75V
6	3.75... <4.375V
7	4.375... 5V

The data are naturally only approximate. The more highly now the dissolution of the analog-digital converter is, thus the more bits it has, all the more exactly can the value be seized.

## The internal ADC in the AVR

If it should be somewhat more exact, then we must fall back to a AVR with inserted analogue-digital converter (ADC), which has several channels. It is called channels in this connection that up to ten analog inputs at the AVR are available, but only “more genuinely” an analogue-digital converter is available, before the actual measurement is thus to be adjusted, which channel (“pin”) is connected and measured with the transducer.

The transformation within the AVR is based on the gradual approximation. With the AVR the pins **AGND** and **AVCC** must be wired. For precise measurements AVCC should be connected by a L-C network with VCC, in order voltage peaks and - break-downs away from the analogue-digital converter to keep. In the data sheet in addition a circuit which and 100 plans 10 MicroHenry nF, is.

The result of the analog-digital transformation is referred to a reference tension. Current AVR offers three possibilities for the choice of this tension:

- An external reference tension of maximally **Vcc** at the connection port **AREF**. The minimum (external) reference tension may be however not arbitrarily low, see in addition (most current) the data sheet of the used microcontroller.
- If the AVR has an internal reference tension, this can be used. All current AVR with internal AD-transducer should be equipped with it (see data sheet: 2,56V or 1,1V depending upon type). The data sheet gives also over the accuracy information to this tension.
- **The internal** reference tension is referred to **Vcc**, an external reference tension to **GND (mass)**. Independently of it digitized tensions are always referred to GND. With the ATMEGA8 the pin AREF is connected e.g. by 32kOhm with GND, i.e. one must this nevertheless extremely low feed impedance also into the computation for a voltage divider include, and/or can this resistance as R2 equal with use. Formula for voltage dividers:  $U_{div} = U / ((R1 + R2) / R2)$

In use of Vcc or the internal reference one recommends to arrange a condenser between the AREF pin and GND. The definition, which tension reference is used, takes place e.g. with the ATmega16 with the bits REFS1/REFS0 in the ADMUX register. The tension which can be measured must lie in the range between **AGND** and **AREF** (all the same whether internally or externally).

**The ADC** can be used in two different modes of operation:

Simple transformation (single Conversion)

In this mode of operation the transducer if necessary knocked against by the program for in each case a measurement.

Freely constantly (Free Running)

In this mode of operation the transducer seizes permanently the lying close tension and writes these into the **ADC DATA registers**.

## The registers of the ADC

**The ADC** has own registers. In the following the description of register of a ATMega16, which has 8 ADC channels. The registers differ however not substantially from those other AVR's (see data sheet).

**ADC control and status register A.**

In this register we adjust, as we would like to use the **ADC**.  
The register is developed as follows:

0	1	2	3	4	5	6	7	Bit
<b>ADPS0</b>	<b>ADPS1</b>	<b>ADPS2</b>	<b>ADIE</b>	<b>ADIF</b>	<b>ADATE</b>	<b>ADSC</b>	<b>ADEN</b>	<b>Name</b>
W	R	R/W	R/W	R/W	R/W	R/W	R/W	<b>R/W</b>
0	1	0	0	0	0	0	0	<b>Initial value</b>

**ADEN (ADC Enable)**

This bit must be set, in order to activate the **ADC** at all. If the bit is not set, the pins can be used like normal I/O pins.

**ADSC (ADC start Conversion)**

With this bit a measuring procedure is started. In the freely running mode of operation the bit must be set, in order to activate the continuous measurement.

If the bit is for the first time set after setting the ADEN **bit**, the microcontroller implements first an additional transformation and only then the actual transformation. This additional transformation is accomplished for initialization purposes.

The bit remains now so long on 1, until the transformation is final, in the case of initialization to the second transformation took place accordingly and goes to it on 0.

**ADATE (ADC Ato trigger Enable)**

With this bit the mode of operation is stopped.

A logic 1 activates the ADC with a trip (trigger), adjusted in the SFIOR register. If no bits are written in the SFIOR register, the ADC in the freely running mode works. **The ADC** now constantly measures the selected channel and writes the measured value into the **ADC DATA registers**.

**ADIF (ADC interrupt flag)**

This bit is set by the **ADC**, if a transformation taken place and the **ADC DATA registers** is updated.

If the **ADIE** bit as well as the I-bit is set in the AVR **status register**, the **ADC interrupt** is released and the interrupt handling routine is called.

The bit is deleted automatically, if the interrupt handling routine is called. It can be however also deleted, as logic 1 is written into the register (in such a way it is located in the AVR documentation).

**ADCSRA**

**ADIE (ADC interrupt Enable)**

If this bit is set and likewise the I-bit in the status register **SREG**, then the ADC **interrupts** activated.

**ADPS2... ADPS0 (ADC Prescaler SELECT bit)**

These bits determine the division factor between the clock frequency and the input clock of the **ADC**.

**The ADC** needs its own clock, for which it produces itself from the CCU Clock frequency. **The ADC clock** should be between 50 and 200kHz.

The prescaler must be adjusted thus in such a way that the CCU clock frequency results in a value between 50-200kHz divided by the division factor.

With a CCU clock frequency of 4MHz for example we count

$$\begin{matrix} \text{TF}_{\min} = \frac{\text{CLK}}{200\,\mathrm{kHz}} = \frac{4000000}{200000} = \mathbf{20} \\ \text{TF}_{\max} = \frac{\text{CLK}}{50\,\mathrm{kHz}} = \frac{4000000}{50000} = \mathbf{80} \end{matrix}$$

Thus the division factor 32 or 64 can be used here. In the interest of the faster transformation time we will stop the factor 32 here.

Division factor	ADPS0	ADPS1	ADPS2
2	0	0	0
2	1	0	0
4	0	1	0
8	1	1	0
16	0	0	1
32	1	0	1
64	0	1	1
128	1	1	1

**ADC DATA registers**

If a transformation is final, the measured value is in these two registers. Only the two least significant bits are used by **ADCH**. Always both registers must be picked out, always **in the order: ADCL, ADCH**. The effective measured value results then too:

```
x = ADCL; // with uint16_t x
x += (ADCH<<8); // in two lines (LSB/MSB sequence and
// C-Operator priority guaranteed)
```

or

```
x = ADCW; // depending upon AVR also x = ADC (see avr/ioxxx.h)
```

### ADC multiplexer SELECT register

With this register the channel which can be measured is selected. With the 90S8535 each pin von Port A can be used as ADC **input** (=8 of channels).

The register is developed as follows:

0	1	2	3	4	5	6	7	Bit
<b>MUX0</b>	<b>MUX1</b>	<b>MUX2</b>	<b>MUX3</b>	<b>MUX4</b>	<b>ADLAR</b>	<b>REFS0</b>	<b>REFS1</b>	<b>Name</b>
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	<b>R/W</b>
0	0	0	0	0	0	0	0	<b>Initial value</b>

### REFS1... REFS0 (ReferenceSelection bit)

With these bits the reference tension can be stopped:

<b>Reference cutting</b>	<b>REFS0</b>	<b>REFS1</b>
External AREF	0	0
AVCC as reference	1	0
Reserved	0	1
Internal 2.56 V	1	1

### ADLAR (ADC Left Adjust Result)

## ADMUX

The ADLAR bit changes the appearance of the suppl. bite of the AD-transformation. With logic 1 the result is left justified spent, with 0 right-justified. A change in this bit affects result immediately, all the same whether a transformation is running.

### MUX4... MUX0

With these 5 bits the channel which can be measured is determined. If a simple 1-channel ADC is used write simply the appropriate pin number of the port indexing the bits 0... 2.

If the register is described, while that a transformation runs, and then first the current transformation on the past channel is terminated. This is to be considered particularly with the freely running enterprise.

A recommendation is therefore this that the freely running enterprise should be used only with an individual analogue input which can be used, if one wants to save problems with the Change over switch.

## Activate the ADC

In order to activate the **ADC**, we must set the ADEN **bit** in the ADCSR **register**. In the same step we specify also equal the mode of operation.

A small example of the “single conversion” - mode with a ATmega169 and a use of the internal reference tension (with '169 1,1V with other AVRs also the 2,56V). I.e. the input signal may not exceed, if necessary with voltage dividers stop this tension. Result of the routine is the ADC value, thus 0 for 0-Volt and 1023 for V<sub>ref</sub>-Volt.



```

uint16_t ReadChannel(uint8_t mux)
{
    uint8_t i;
    uint16_t result = 0;           // Initialization importantly, since local variables
                                   // not are initialized automatically and
                                   // coincidental values to have. Otherwise knows rubbish get a
long
    ADCSRA = (1<<ADEN) | (1<<ADPS1) | (1<<ADPS0); // frequency prescaler
                                   // set to 8 (1) and ADC activate (1)

    ADMUX = mux;                   // channel select
    ADMUX |= (1<<REFS1) | (1<<REFS0); // internal reference tension use
    /* after activating the ADC a "dummy Readout" are recommended, one read thus
    a value and reject this, around the ADC "warm up to let"*/
    ADCSRA |= (1<<ADSC);           // a ADC transformation
    while ( ADCSRA & (1<<ADSC) ) {
        ; // on conclusion of the conversion wait
    }

    /* actual measurement - average value from 4 following each other transformations */
    for(i=0;i<4;i++)
    {
        ADCSRA |= (1<<ADSC);       // a transformation "single conversion"
        while ( ADCSRA & (1<<ADSC) ) {
            ; // on conclusion of the conversion wait
        }
        result += ADCW;             // transformation results add
    }
    ADCSRA &= ~(1<<ADEN);           // ADC deactivate (2)

    result /= 4;                   // sum by four divide = arithm. Average value
    return result;
}

...

/* example calls: */

void foo(void)
{
    uint16_t adcval;

    adcval = ReadChannel(0); /* MUX bits on 0b0000 - > Channel 0 */
    ...
    adcval = ReadChannel(2); /* MUX bits on 0b0010 - > Channel 2 */
    ...
}

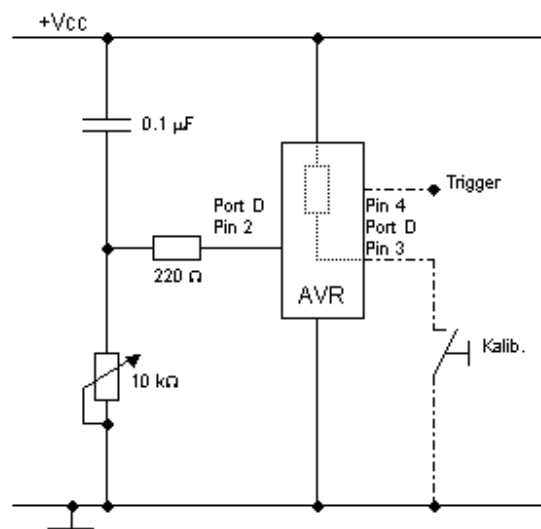
```

In the example with each call of the ADC and after the transformation again switched off, saves river is activated. If one does not want this, one shifts with (1) characterized lines into a function `adc_init()` and deletes with (2) marked lines.

## Analog-digital transformation without internal ADC

### Measure a resistance

Analog values can be determined without analogue-digital converters also indirectly. In the following the measurement of the resistance stopped at a potentiometer is described on the basis the load curve of a condenser. With this method only one port pin is needed, an analogue-digital converter or a analog to Comparator is not necessary. In addition condenser and the resistance (the potentiometer) is switched into row between taking precautions tension and Masse/GND (suctions. RC-network). Additionally a connection of the line between condenser and potentiometer to a port pin of the microcontroller is made. The following illustration clarifies the necessary circuit.



The port pin of the microcontroller on output one configures (in the example  $DDRD \text{ /= } (1 < PD3)$ ) and this output on logic 1 ("High",  $PORTD \text{ /= } (1 < PD3)$ ) switched, the same potential VCC rests and the condenser thus unloaded against both "plates" of the condenser. (Sounds amusingly, with Vcc unloaded, is at both sides of the condenser the same potential fits however in such a way, there and thus a difference of potential of 0V exists => condenser is unloaded).

After a certain time the condenser is unloaded and the port pin as input is configured ( $DDRD \text{ \&= } \sim (1 < PD3)$ ;  $PORTD \text{ \&= } (1 < PD3)$ ), whereby this becomes high impedance. The status input pin (in PIND) is logic 1 (High). The condenser loads itself now over the potentiometer up, the voltage drop over the condenser and that rise over the potentiometer sink. Now if the voltage drop over the potentiometer falls under the Threshold spanning of the input pin ( $2/5 V_{cc}$ , thus approx. 2V), the input signal is recognized as LOW (bit in PIND becomes 0). The time interval between the change-over of unloading on loading and the change of the input signal of High on Low is a measure for the resistance stopped at the potentiometer. For time measurement one can be used in the microcontroller existing timer. The 220 ohms resistance serves the protection of the microcontroller. It would flow otherwise during maximum attitude of the Potentiometers (here 0 ohms) a too high river, which destroys the input stage of the microcontroller.

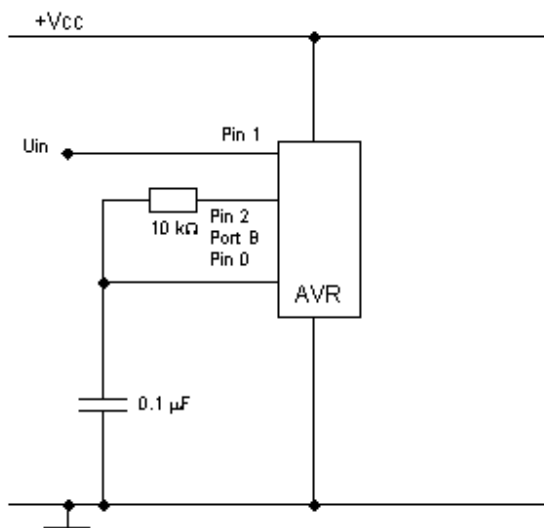
With a further input pin and little software can we also a calibration realize, around the measured value into a reasonable range (e.g.: to convert 0... 100% or in such a way). A sample program is on [Christian Schifferles](#)

[web page](#) in archives *ATMEL.ZIP*, which under the title *Tutorial “programming with C for Atmel microcontroller* can be downloaded.

### ADC over comparator

There is a further way to measure a analog tension with the help of the comparator, which is integrated in nearly each AVR. See in addition also the Application note AVR400 von Atmel.

The signal which can be measured is led on the inverting input of the comparator. Additionally a reference signal is attached to the not inverting input of the comparator. The reference signal is produced here also again over an RC element, however with firm values for R and C.



The principle of the measurement is now quite analog to the preceding. By creation of a LOW level at pin 2 the condenser is first once unloaded. Also here it must be paid attention to the fact that the discharge process lasts sufficient long.

Now pin 2 is put on HIGH. The condenser is loaded. If the tension over the condenser the tension resting against the input pin reached connects through the comparator. The time, which is needed, around the condenser to load can be consulted now also again as measure for the tension of pin 1.

I saved it for me to also develop this circuit for several reasons:

1. 3 pin necessarily.
2. Accuracy comparably with simpler solution.
3. Was simply too putrid.

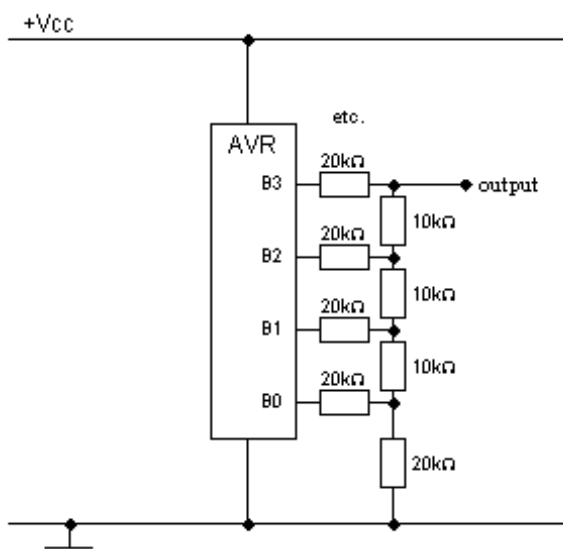
The advantage of this circuit is however in the fact that thereby directly tensions can be measured.

## DAC (digital to analog converter)

With the help of a digital analog converter (**DAC**) we can spend now also analog signals. There are here several procedures.

### DAC over several digital outputs

If we develop at the outputs of the microcontroller an appropriate resistance network have we the possibility of developing by the control of the outputs over the resistances an adder with whose assistance we can produce a proportional tension for the numerical value. The diagram in addition can look about in such a way:

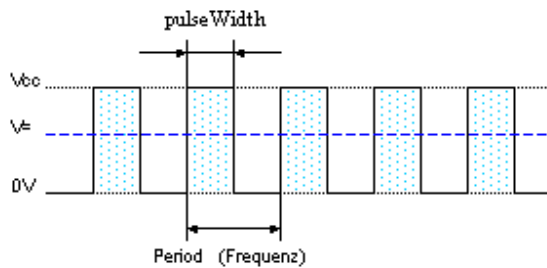


Naturally if possible details of resistors should be used, thus not necessarily such with a tolerance of 10% or more. Further it is advisable to strengthen depending upon application output current over an operation amplifier.

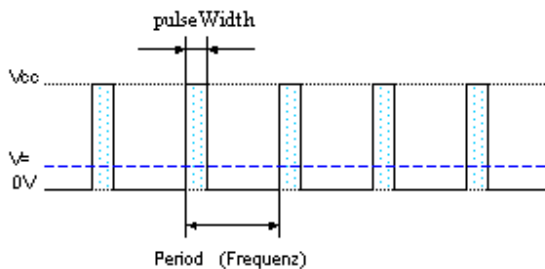
### PWM (pulse width modulation)

We come now to a topic, which is in all mouth, but many users understand not completely, how **PWM** actually functions.

As we know all, a microcontroller is a purely digital construction unit. If we define a pin as output, then we can set this output either on HIGH, on which against the output supply voltage **Vcc** rests, or however we set the output on LOW, according to which then **0V** is because of the output. What happens however now, if we switch periodically with a firm frequency between HIGH and LOW? - Correct, we receive a square wave voltage, as the following illustration shows:



This square wave voltage has now a geometrical average value, which is depending upon Pulse width smaller or larger.



If we filter now this pulsating output voltage still over an RC element “smooth”, then have we already appropriate DC voltage produced.

With the AVRs we can produce directly PWM **signals**. In addition the 16-Bit serves counter, which can be operated in the so-called PWM **mode**.

Note:

In the following considerations as microcontroller of the 90S2313 one presupposes. The theory is comparable with other AVR Controller, the pin allocation however not necessarily, why a view is urgently recommended in the appropriate data sheet.

In order to activate the PWM **mode**, control register A **TCCR1A** the pulse width modulator bits **PWM10** and/or **PWM11** must be set according to following table in the timer/Counter1:

PWM11	PWM10	Meaning
0	0	PWM mode of the timer is not active.
0	1	8-Bit PWM.
1	0	9-Bit PWM.
1	1	10-Bit PWM.

The timer/Counter counts now permanently from 0 to the upper limit and again back, it as so-called up/down counters is thus operated. The upper limit depends on whether we want to work with 8, 9 or 10-bit PWM:

Dissolution	Upper limit	Frequency
8	255	$f_{TC1}/510$
9	511	$f_{TC1}/1022$
10	1023	$f_{TC1}/2046$

Additionally **COM1A0** of the same register the desired output mode of the signal must be defined with the bits **COM1A1** and:

COM1A1	COM1A0	Meaning
0	0	No effect, pin is not switched.
0	1	No effect, pin is not switched.
1	0	Not inverting PWM. The output pin deleted during counting up and set during the Herunterzählen.
1	1	Inverting PWM. The output pin deleted during the Herunterzählen and set during counting up.

The appropriate instruction, in order to use for example the timer/Counter as not inverting 10-bits PWM, is called then:

Old way of writing (PWMxx is not any more accepted)

```
TCCR1A = (1<<PWM11) | (1<<PWM10) | (1<<COM1A1);
```

New way of writing

```
TCCR1A = (1<<WGM11) | (1<<WGM10) | (1<<COM1A1);
```

So that the timer/Counter runs at all, we must stop in the control register B **TCCR1B** still the desired clock (prescaler) and thus also the frequency of the PWM **signal** determine.

CS12	CS11	CS10	Meaning
0	0	0	Stop. The timer/Counter is stopped.
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	External pin 1, negative flank
1	1	1	External pin 1, positive flank

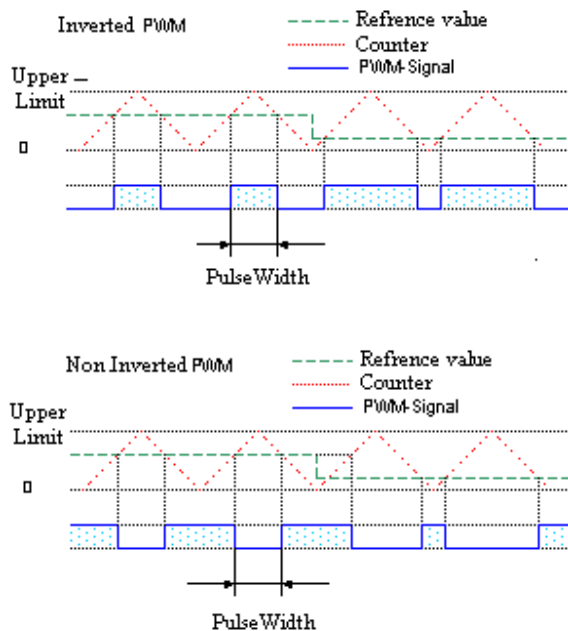
Thus in order to generate a clock from CK/1024 to, we use the following instruction:

```
TCCR1B = (1<<CS12) | (1<<CS10);
```

Now the reference value must be only specified. We write these into the 16-Bit timer/Counter output Compare register **OCR1A**.

```
OCR1A = xxx;
```

The following diagram is to point the connection out between the reference value and the generated PWM signal.



Oh, nearly I would have forgotten. The generated **PWM-signal** spent at the output Compare pin **OC1 of the** timer and unfortunately to know we therefore also with AT90S2313 only an individual **PWM-signal** with this method to generate. Other AVR types order over up to four PWM outputs.

## The timers/Counter of the AVR

The today's microcontrollers and in particular the RISC AVR's are too fast for many control problems. If we want to let an LED or a lamp flash for example, we cannot use naturally the CCU frequency, since then nothing more would be to be noticed from flashing to.

We need thus a possibility of accomplishing procedures in time intervals which are smaller than the clock frequency of the microcontroller. Of course the resulting frequency should be also still as exact and stable as possible.

Here the timer/Counter existing in the AVR is used.

A timer is completely simply a certain register in  $\mu C$  that completely without effort of the program, by hardware, one counts up. That alone would not be yet all too useful; if not this hardware register could release an interrupt with certain counts. Such an event is the OVERFLOWS: Since those is limited bit width of the register, it naturally also occurs that the counter counts so highly that the next count with this bit width is no longer representable and is put back the counters 0. This event one calls the OVERFLOWS and it is possible to this event an interrupt to be coupled.

Another area of application is the counting of signals, which can be supplied by way of a I/O pin.

The following remarks refer to AT90S2313. For other model types you must “if necessary” make some adjustments from the data sheets of the appropriate microcontrollers pick out. We differentiate in principle between 8-bit timers, which exhibit a dissolution of 256 and 16-Bit timers with (logical-proves) a dissolution of 65536. When input clock for the timers/Counter knows either the CCU clock frequency, the output prescaler or a signal set on a I/O pin is used. If an external signal is used, then its frequency may be not higher than half of the CCU clock.

## The prescaler

The prescaler serves to reduce the CCU clock for the time being by an adjustable factor. In such a way divided frequency is supplied to the inputs of the timers.

If we work with a CCU clock of 4 MHz and adjust the prescaler to 1024, the timer is thus supplied with a frequency of 4 MHz/1024, thus with approx. 4 kHz. If thus the timer runs, then the data and/or counting register (TCNTx) with this frequency is incremented.

## 8-bit timer/Counter

All AVR models have two, 8-bit timer at least, partly even.

The 8-bits timer is addressed e.g. with AT90S2313 via the following registers (with other types as far as possible similar):

TCCR0

Timer/Counter Control Register

Timer 0

In this register we adjust, as we would like to use the timer/Counter.

The register is developed as follows:

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	CS02	CS01	CS00
R/W	R	R	R	R	R	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

CS02, CS01, CS00 (Clock SELECT bit)

These 3 bits intends the source for the timer/Counter:

CS00	CS01	CS02	Result
0	0	0	Stop, the timer/Counter are stopped.
1	0	0	CCU clock
0	1	0	CCU clock/8
1	1	0	CCU clock/64
0	0	1	CCU clock/256
1	0	1	CCU clock/1024



	0	1	1	External pin <b>TON</b> , falling flank				
	1	1	1	External pin <b>TON</b> , rising flank				
	If as source the external pin <b>TON</b> is used, then a flank change is also recognized, if the pin <b>TON</b> is switched as output.							

TCNT0	<b>Timer/Counter</b> data register timer <b>0</b> This is realized as 8-bit upward counter with writing and read access. If the counter the value 255 reached begins it with the next cycle again with 0.								
	Bit	7	6	5	4	3	2	1	0
	Name	MSB							LSB
	R/W	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

In order to thus put and it with a frequency of 1/1024-tel of the CCU clock count let now the Timer0 into operation, we write the following command line:

```
TCCR0 |= (1<<CS00) | (1<<CS02);
```

The counter counts now upward to 255, in order to then begin again with 0. The current count stands in TCNT0. With each overflow of 255 to 0 timer is set the OVERFLOW flag **TOV0** in the timer interrupt flag TIFR **register** and released, if configures in such a way, an appropriate timer OVERFLOW interrupt and processed the interrupt routine bound to it. The TOV flag leaves itself by writing a 1 and not as expects 0 again to put back.

## 16-Bit timer/Counter

Many AVR models possess 16-Bit timer except the 8-bits timers also. The 16-Bit timer/Counter is somewhat more complex developed more possibilities than the 8-bits timers/Counter, offers for it in addition, many, than are there:

- [The PWM mode of operation](#) production of a pulse-far-modulated output signal.
- Reference value examination with production of an output signal (output Compare match).
- Catch an input signal with storage of the current counter value (input Capturing), with insertable intercarrier noise suppressor (Noise Filtering).

The following registers are assigned to the timer/Counter 1:

## Timer/Counter Control Register A Timer 1

In this and the following register we adjust, as we would like to use the timer/Counter. The register is developed as follows:

Bit	7	6	5	4	3	2	1	0
Name	COM1A1	COM1A0	-	-	-	-	PWM11	PWM10
R/W	R/W	R/W	R	R	R	R	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

### COM1A1, COM1A0 (Compare match control bit)

These 2 bits determine the action, which is to be implemented at the output pin **OC1**, if the value of the data pointer reaches timers/Counter 1 the value of the comparison register, thus a Compare so mentioned match arises.

The pin **OC1** (**PB3** with 2313) must be configured with the data direction register as output.

COM1A1	COM1A0	Result
0	0	Output pin <b>OC1</b> is not headed for.
0	1	The signal at the pin <b>OC1</b> is inverted (Toggle).
1	0	The output pin <b>OC1</b> is set to 0.
1	1	The output pin <b>OC1</b> is set to 1.

## TCCR1A

In the PWM mode of operation these bits have another function.

COM1A1	COM1A0	Result
0	0	Output pin <b>OC1</b> is not headed for.
0	1	Output pin <b>OC1</b> is not headed for.
1	0	During counting up if the value in the comparison register is reached, and then the pin <b>OC1</b> is set to 0.  During the Herunterzählen if the value in the comparison register is reached, then the pin is set to 1.  One calls this <i>not inverting PWM</i> .
1	1	During counting up if the value in the comparison register is reached, and then the pin <b>OC1</b> is set to 1.  During the Herunterzählen if the value in the comparison register is reached, then the pin is set to 0.  One calls this <i>inverting PWM</i> .

### PWM11, PWM10 (PWM mode SELECT bit)

With these 2 bits the PWM mode of operation timers/Counter 1 is steered.

PWM10	PWM11	Result
0	0	The PWM mode of operation is not activated. Timers/Counter 1 works as normal timers and/or counters.
1	0	8-bit PWM mode of operation activated.
0	1	9-Bit PWM mode of operation activated.
1	1	10-bit PWM mode of operation activated.

#### Timer/Counter Control Register B timer 1

Bit	7	6	5	4	3	2	1	0
Name	<b>ICNC1</b>	<b>ICES1</b>	-	-	<b>CTC1</b>	<b>CS12</b>	<b>CS11</b>	<b>CS10</b>
R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

#### ICNC1 (Input Capture Noise Canceler (4 CKs) Timers/Counter 1

or on German intercarrier noise suppressor of the input signal.

If this bit is set and with the input Capture signal is worked after triggering the signal with the appropriate flank (rising or falling) at the input Capture pin **ICP** in each case 4 measurements with the CCU frequency of the input signal is in such a way queried. Then only if all 4 measurements exhibit the same condition apply the signal as recognized.

#### ICES1 (Input Capture Edge Select Timer/Counter 1)

With this bit it is determined whether the rising (**ICES1=1**) or falling (**ICES1=0**) flank is near drawn for the evaluation of the input Capture of signal on pin **ICP**.

#### CTC1 (Clears Timer/Counter on Compare match Timer/Counter 1)

If this bit is set, then after an agreement of the data pointer **TCNT1H/TCNT1L** with the reference value in **OCR1H/OCR1L** the data pointer **TCNT1H/TCNT1L** is set to 0.

Since the agreement because of the clock is treated according to the comparison, a something else results counting behavior depending upon adjusted prescaler:

If the prescaler is set to 1 and C is the respective counter value, then the data pointer, in the CCU clock regarded, takes the following values:

... | C-2 | c1 | C | 0 | 1 | ...

If the prescaler is adjusted to 8 e.g., then the data pointer takes the following values:

... | C-2, C-2, C-2, C-2, C-2, C-2, C-2, C-2 | c1, c1, c1, c1, c1, c1, c1, c1 | C, 0, 0, 0, 0, 0, 0, 0 | ...

In the PWM mode of operation this bit does not have a function.

#### CS12, CS11, CS10 (Clock SELECT bit)

**TCCR1B**

These 3 bits intends the source for the timer/Counter:

CS12	CS11	CS10	Result
0	0	0	Stop, the timer/Counter are stopped.
0	0	1	CCU clock
0	1	0	CCU clock/8
0	1	1	CCU clock/64
1	0	0	CCU clock/256
1	0	1	CCU clock/1024
1	1	0	External pin T1, falling flank
1	1	1	External pin T1, rising flank

If as source the external pin T1 is used, then a flank change is also recognized, if the pin T1  
Is switched as output.

#### Timer/Counter Data Registers Timer/Counter 1

This is realized as 16-Bit upward counter with writing and read access. If the counter reached the value 65535, it begins with the next cycle again with 0.

Bit	7	6	5	4	3	2	1	0	
Name	MSB								TCNT1H
Name								LSB	TCNT1L
R/W	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	

**TCNT1H**  
**TCNT1L**

In the PWM mode of operation the register is used as up/down counter, i.e. the value rises first from 0, until it reached the overflow of 65535 to 0. Then the register counts backwards again to 0.

For the selection of the register by the CCU an internal temp register is used. The same register is also used, if **OCR1** or **ICR1** one accesses.

Therefore all interrupts must become closed, because otherwise the possibility of the simultaneous access to the temporary register is given, what incorrect behavior of the program naturally lead. Before the access to one of these registers. Besides first **TCNT1L** and only thereafter **TCNT1H** must be selected.

If into the register is to be written, all Interrupts must likewise become closed. Then first the **TCNT1H-Register** and only thereafter the **TCNT1L-Register** must be written, thus exactly the reverse sequence as during the reading of the register.

#### Timer/Counter Output Compare Registers timer/Counter 1

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

<b>OCR1H</b> <b>OCR1L</b>	<b>Name</b>	<b>MSB</b>								<b>OCR1H</b>
	<b>Name</b>								<b>LSB</b>	<b>OCR1L</b>
	<b>R/W</b>	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	<b>Initial value</b>	0	0	0	0	0	0	0	0	
	<p>The value in the output Compare register is constantly compared with the current value in the data pointer <b>TCNT1H/TCNT1L</b>. If the two values agree, then a so-called output Compare match is released. The appropriate actions are stopped over the timers/Counter 1 control and status register.</p> <p>For the selection of the register by the CCU an internal temp register is used. The same register is also used, if <b>OCR1</b> or <b>ICR1</b> one accesses. Therefore all interrupts must become closed, because otherwise the possibility of the simultaneous access to the temporary register is given, what incorrect behavior of the program naturally lead. Before the access to one of these registers. Besides first <b>OCR1L</b> and only thereafter <b>OCR1H</b> must be selected.</p> <p>If into the register is to be written, all interrupts must likewise become closed. Then first the <b>OCR1H-Register</b> and only thereafter the <b>OCR1L-Register</b> must be written, thus exactly the reverse sequence as during the reading of the register.</p>									
<b>ICR1H</b> <b>ICR1L</b>	Timer/Counter <b>Input Capture Registers timer/Counter 1</b>									
	<b>Bit</b>	7	6	5	4	3	2	1	0	
	<b>Name</b>	<b>MSB</b>								<i>ICR1'H'</i>
	<b>Name</b>								<b>LSB</b>	<i>ICR1'L'</i>
	<b>R/W</b>	R	R	R	R	R	R	R	R	
	<b>Initial value</b>	0	0	0	0	0	0	0	0	
	<p>The input Capture register is a 16-Bit register with read access. It cannot be described.</p> <p>If by the input Capture pin <b>ICP</b> the flank defined in accordance with parameters in the <b>TCCR1B</b> is recognized, then current contents of the data pointer <b>TCNT1H/TCNT1L</b> are copied immediately into this register and the input Capture flag <b>ICF1</b> in the timer interrupt flag register <b>TIFR</b> is set.</p> <p>As mentions already above, all interrupts must become closed before the access to this register. Besides Low and High byte of the register in the correct order must be worked on:</p> <p>Read: <b>ICR1L - &gt; ICR1H</b> Write: <b>ICR1H - &gt; ICR1L</b></p>									

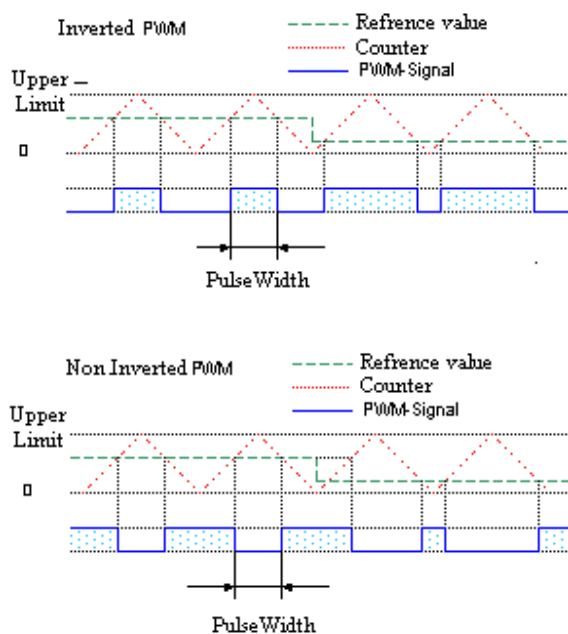
## The PWM mode of operation

If the timer/Counter 1 in the PWM mode of operation is operated, then the data pointer **TCNT1H/TCNT1L** and the comparison register **OCR1H/OCR1L** forms 8, 9 - or 10-bit, freely running PWM modulator, which can be measured as PWM signal at the **OC1-Pin (PB3 with 2313)**. The data pointer **TCNT1H/TCNT1L** is operated thereby as on/off counter which counts from 0 to upward up to the upper limit and afterwards again back on 0. The upper limit results from it, whether 8, 9 - or 10-bit PWM is used, in accordance with the following table:

Dissolution	Upper limit	Frequency
8	255	$f_{TC1}/510$
9	511	$f_{TC1}/1022$
10	1023	$f_{TC1}/2046$

If now the counter value in the data pointer achieves the value stored in **OCR1H/OCR1L**, the expenditure pin **OC1** is set and/or reset, depending upon attitude of **COM1A1** and **COM1A0** in the **TCCR1A-Register**.

I tried to summarize the appropriate signals in the following diagram



## Reference value examination

Here into a special reference value register (**OCR1H/OCR1L**) a value is written, which is constantly compared with the current counter value. If the counter achieves the value registered in this register, then a signal (0 or 1) at the pin **OC1** can be produced and/or an interrupt be released.

## Catch an input signal (inputs Capturing)

With this mode of operation at the inputs Capturing pin (ICP) of the microcontroller a signal source is attached. Now knows depending upon configuration either a signal change from 0 to 1 (rising flank) or from 1 to 0 (falling flank) to be recognized and at this time current count into a special register be put down. At the same time also an appropriate interrupt can be released. If the signal source contains a strong noise, the intercarrier noise suppressor can be switched on. Then when recognizing the flank over 4 clock cycles, configured, the signal is supervised and released then only if equal all 4 measurements are, the appropriate action.

## Common registers

Different registers contain conditions and parameters, which are both for the 8-bits and for the 16-Bit timer/Counter in the same register to be found.

**Timer/Counter interrupt MASK**

Register

Bit	7	6	5	4	3	2	1	0
Name	<b>TOIE1</b>	<b>OCIE1A</b>	-	-	<b>TICIE</b>	-	<b>TOIE0</b>	-
R/W	R/W	R/W	R	R	R/W	R	R/W	R
Initial value	0	0	0	0	0	0	0	0

**TOIE1 (Timer/Counter Overflow Interrupt Enable timer/Counter 1)**

If this bit is set, with an overflow of the data pointer timer/Counter 1 a timer OVERFLOW 1 interrupt is released. Global the Enable interrupt flag must be naturally also set.

**TIMSK OCIE1A (Output Compare Match Interrupt Enable timer/Counter 1)**

With the timer/Counter 1 additionally a reference value can be defined to the overflow. If this bit is set, with reaching the reference value a Compare match interrupt is released. Global the Enable interrupt flag must be naturally also set.

**TICIE (Timer/Counter Input Capture Interrupt Enable)**

If this bit is set, a Capture Event interrupt is released, if an appropriate signal event at the pin PD6 (ICP) arises. Global the Enable interrupt flag must be naturally also set, even if an appropriate interrupt is to be released.

**TOIE0 (Timer/Counter Overflow Interrupt Enable Timer/Counter 0)**

If this bit is set, with an overflow of the data pointer timer/Counter 0 a timer OVERFLOW 0 interrupts is released. Global the Enable interrupt flag must be naturally also set.

**Timer/Counter Interrupt flag Register**

Bit	7	6	5	4	3	2	1	0
Name	<b>TOV1</b>	<b>OCF1A</b>	-	-	<b>ICF1</b>	-	<b>TOV0</b>	-
R/W	R/W	R/W	R	R	R/W	R	R/W	R
Initial value	0	0	0	0	0	0	0	0

**TOV1 (Timer/Counter Overflow flag timer/Counter 1)**

**TIFR** This bit is set by the microcontroller, if at the timer 1 an overflow of the data pointer takes place.  
In the PWM mode of operation the bit is set, if the counting direction from open-closed downward and in reverse is changed (counter value = 0).  
The flag is deleted automatically, if the associated interrupt vector is called. It can be however also deleted, by a logic 1 (!) into the appropriate bit one writes.

**OCF1A (Output Compare flag timer/Counter 1)**

This bit is set, if the current value of the data pointer of timer/Counter agrees 1 with that in the

comparison register **OCR1**.

The flag is deleted automatically, if the associated interrupt vector is called. It can be however also deleted, by a logic 1 (!) into the appropriate bit one writes.

#### **ICF1 (Input Capture flag timer/Counter 1)**

This bit set, if an Capture event arose, which indicates that the value of the data pointer became to transfer timers/Counter 1 into the input Capture register ICR1.

The flag is deleted automatically, if the associated interrupt vector is called. It can be however also deleted, by a logic 1 (!) into the appropriate bit one writes.

#### **TOV0 (Timer/Counter Overflow flag timer/Counter 0)**

This bit is set by the microcontroller, if at the timer 0 an overflow of the data pointer takes place.

The flag is deleted automatically, if the associated interrupt vector is called. It can be however also deleted, by logic 1 (!) into the appropriate bit one writes.

## Sleep mode

AVR microcontrollers have a set of so-called *Sleep modes* (“sleep modes”). These make it possible to switch parts off of the microcontroller. On the one hand thereby particularly power can be saved; on the other hand can components of the microcontroller be deactivated with battery operation, which affects the accuracy of the analogue-digital converter and/or the analog to Comparators negatively. The microcontroller is waked by interrupts from the sleep. Which interrupts terminate the respective sleep mode, is to be taken from a table in the data sheet of the respective microcontroller. The functions `avr libc` are available after merging the header file *sleep.h*.

- **set\_sleep\_mode (uint8\_t mode)**

Sets the sleep mode, which is activated with call of `sleep ()`. In *sleep.h* some constants are defined

(e.g. `SLEEP_MODE_PWR_DOWN`). The defined modes are not supported however by all simultaneous AVR Controller.

- **sleep\_mode ()**

Shifts the microcontroller in with `set_sleep_mode` selected sleep mode.

```
#include <avr/io.h>
#include <avr/sleep.h>
...
set_sleep_mode(SLEEP_MODE_PWR_DOWN);
sleep_mode();

// code here only after occurrence of an appropriate
// "waking up interrupt" finished...
...
```



In older Versions of avr libc not all AVR microcontrollers were headed for correctly by the sleep functions. With avr libc 1.2.0 the number of supported types was extended however clearly. With non-supported types one achieves desired functionality by direct “bit manipulation” of the appropriate registers (see data sheet) and call of the Sleep instruction via Inline assembler:

```
#include <avr/io.h>
...
// Sleep mode "power-save" with the ATmega169 "manually" activated
SMCR = (3<<SM0) | (1<<SE);
asm volatile ("sleep::");
...
```

- see also: [Documentation avr libc the](#) section Modules/power management and Sleep mode

## The Watchdog

And here the ultimate means comes against the incompleteness of us programmers, the **Watchdog**.

So much we are also exerted, for us it is hard to succeed developing absolutely perfect and error free program. The Watchdog cannot help us also to better programs however it can ensure that our program, if it said good-bye again it is started again, as a RESET of the microcontroller is released. We regard nevertheless once the following code sequence:

```
uint8_t x;

x = 10;

while (x >= 0)
{
    // do which

    x--;
}
```

If we look at the loop to we should notice that the same loop is never terminated. Why not? Completely simply, because as **unsigned** defined variable can never become smaller than zero (the compiler should spend however a nice warning).

The program would hang itself up in here and would turn eternally in the loop. And here the Watchdog comes exactly to the course.

## How now does the Watchdog function?

The Watchdog contains a separate timer/Counter, which with an internally produced clock by 1 MHz with 5V Vcc one clocks.

Some microcontrollers have their own Watchdog oscillator, e.g. the Tiny2313 with 128 kHz. After the Watchdog was activated and the desired prescaler was adjusted, the Counter from 0 begins to count up. If now the number of cycles stopped depending upon prescaler were reached, the Watchdog releases a RESET.

In order to thus prevent now in the normal operation the RESET, we must regularly again start and/or reset the Watchdog (Watchdog RESET). This should happen within our major loop.

In order to prevent an unintentional switching of the Watchdogs off, a special Procedure must be used, in order to switch the WD off. First the two bits WDTOE and WDE in an individual operation (thus not with sbi) must be set to 1. Then the bit WDE must be set to 0 within the next 4 clock cycles.

The Watchdog control register:

WDTCR

Watchdog timer control register

In this register we adjust, as we would like to use the Watchdog.

The register is developed as follows:

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
R/W	R	R	R	R/W	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

WDTOE (Watchdog turn off Enable)

This bit must be set, if the bit **WDE** is deleted, otherwise one the Watchdog is not switched off. If the bit is set, it is automatically again deleted by the hardware after 4 clock cycles.

WDE (Watchdog Enable)

If this bit is set, then the Watchdog is activated.  
The bit can be only deleted, as long as the bit **WDTOE** stands to 1.

WDP2, WDP1, WDP0 (Watchdog timer **Prescaler** bit)

These 3 bits intends the number of oscillator cycles for the Watchdog, thus, how long it lasts, until a RESET is released:

WDP2	WDP1	WDP0	Number of cycles	Type. Timeout period with Vcc = 3V	Type. Timeout period with Vcc = 5V
0	0	0	16K	47ms	15ms
0	0	1	32K	94ms	30ms
0	1	0	64K	0.19s	60ms
0	1	1	128K	0.38s	0.12s
1	0	0	256K	0.75s	0.24s
1	0	1	512K	1.5s	0.49s
1	1	0	1024K	3s	0.97s
1	1	1	2048K	6s	1.9s

In order to use the Watchdog with the AVR-GCC compiler, the header file must be merged *wdt.h* (*#include <avr/wdt.h>*) into the source file. Afterwards the following functions can be used:

- **wdt\_enable (uint8\_t timeout)**

Activate the Watchdog and adjust the prescaler to the desired value and/or in timeout handed over value into the WDTCR register one registers. Some Timeout values are pre-defined as constants

Possible Time out Worth:

Constant one	Worth	Time Out
WDTO_15MS	0	15 ms
WDTO_30MS	1	30 ms
WDTO_60MS	2	60 ms
WDTO_120MS	3	120 ms
WDTO_250MS	4	250 ms
WDTO_500MS	5	500 ms
WDTO_1S	6	1 S
WDTO_2S	7	2s

- **wdt\_disable ()**

With this function the Watchdog can be switched off. The necessary Procedure, as, is described above implemented automatically.

- **wdt\_reset ()**

This is probably the most important of the Watchdog functions. It produces a Watchdog RESET, which must be implemented periodically, before expiration of the Time out time, so that the Watchdog does not put the AVR back.

Of course the WDTCR **register** can be programmed also with us the already well-known functions for the access to registers.

## Watchdog possible applications

Whether now the Watchdog is to be used as protective function in first place, depends strongly upon application, used periphery , the range and the quality assurance of the code. If one wants to get surely that a program does not get caught in a continuous loop, the Watchdog is the suitable mean for this to be prevented. Further it can be used with skillful programming to implement certain current savings functions. With some newer AVRs (e.g. the ATtiny13) the Watchdog can be used also directly than timer, which wakes the microcontroller up from a sleep mode. Also this can be adjusted in the WDTCR **register**. In addition the WD offers the only possibility an intended system RESET (a “correct RESET”, none “**jmp 0x0000**”) without external wiring to release, which e.g. with the implementation of a Bootloaders is useful. With certain applications the use of the WD can naturally always serve as “ultimate deadlock safety device for not considered conditions” as additional safety device.

It exists to find the possibility out whether a RESET was released by the Watchdog (with the ATmega16 e.g. bit WDRF in MCUCSR). This information should be also used, if a WD-RESET were not implemented according to plan in application. For example one can hang an LED to a free pin, which lights up only with a RESET by

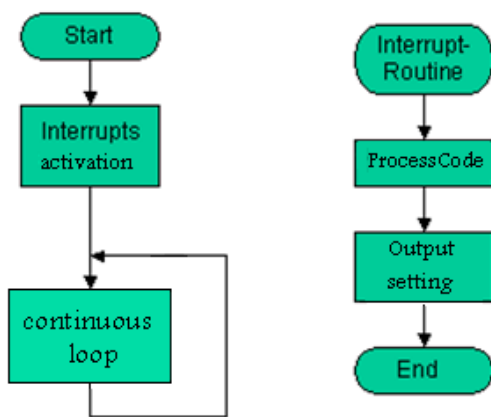
the WD or secures however the “event WD-RESET” in the internal EEPROM of the AVR, in order the information e.g. over UART or a display to spend later (or simply pick EEPROM contents out over the ISP/JTAG interface).

- see also: [Documentation avr libc the](#) section Modules/Watchdog timer handling

## Program with interrupts

After we taught now all that worth to be known for serial program preparation we now will completely start different topic, i.e. *programming with help of the interrupts of the AVR*.

As the first we want to lead ourselves again the general program sequence during interrupt programming to mind.



One sees that the interrupt routine runs off quasi parallel to the main program. There we only one CCU have are natural it no genuine parallelism, but the main program is interrupted with the arrival of an interrupt, the interrupt routine is implemented and to it only again to the main program returns.

[Detailed Thread in the forum](#) تم نسخها في ملف اللينكات المهمة

## Requirements at interrupt routines

In order to prevent unpleasant surprises, some basic rules should be considered with the implementation of the interrupt routines. Interrupt routines should be as short and fast processable as possible, from this follows:

- No extensive computations within the interrupt routine. (\*)
- No long program loops.
- Although it is possible to permit during the processing of an interrupt routine other one or even the same interrupt again without exact knowledge of the internal operational sequence urgently one advises against.

Interrupt routines (ISRs) should be as short as possible thus and contain no loops with many runs. Longer operations can be usually isolated into a “interrupt part” in a ISR and a “working part” in the main program. E.G. memory of the condition of all inputs in the EEPROM in certain time intervals: ISR part: Time comparison (timer, RTC) with log time interval. During agreement a global flag set (volatile with flag declaration forgotten,

do not see below). Examine then in the main program whether the flag is set. If: the data in the EEPROM placing and flag delete.

(\*) Note: There is however the rare situation that one must process straight read in ADC values immediately. Particularly if one gets several values very fast clean other. Then nothing else remains for one to process as the values still in the ISR. Occurs however very rarely and should be avoided by suitable choice of the system clock and/or selection of the microcontroller!

## Sources of interrupt

The following events can release an interrupt on a AVR AT90S2313, whereby the sequence of the listing points out also the priority of the interrupts.

- RESET
- External interrupt 0
- External interrupt 1
- Timer/Counter 1 Capture event
- Timer/Counter 1 Compare match
- Timer/Counter 1 overflow
- Timer/Counter 0 overflow
- UART indication receive
- UART data pointer empty
- UART indication sent
- Analog comparator

The number of possible sources of interrupt varies between the different types. In the doubt a view helps in the data sheet (“interrupt Vectors”).

# Register

At90S2313 had 2 registers with the interrupts together hang.

<b>GIMSK</b>	<b>General Interrupt Mask Register.</b>								
	<b>Bit</b>	7	6	5	4	3	2	1	0
	<b>Name</b>	<b>INT1</b>	<b>INT0</b>	-	-	-	-	-	-
	<b>R/W</b>	R/W	R/W	R	R	R	R	R	R
	<b>Initial value</b>	0	0	0	0	0	0	0	0

**INT1 (External Interrupt Request 1 Enable)**

If this bit is set, an interrupt is released, if by the **INT1-Pin** a rising or falling (depending upon configuration in the **MCUCR**) flank is recognized.  
Global the Enable interrupt flag must be naturally also set.  
The interrupt is also released, if the pin is switched as output. In this way the possibility is offered of realizing software interrupts.

### INT0 (External Interrupt Request 0 Enable)

If this bit is set, an interrupt is released, if by the **INT0-Pin** a rising or falling (depending upon configuration in the **MCUCR**) flank is recognized.

Global the Enable interrupt flag must be naturally also set.

The interrupt is also released, if the pin is switched as output. In this way the possibility is offered of realizing software interrupts.

### General Interrupt Flag Register.

Bit	7	6	5	4	3	2	1	0
Name	INTF1	INTF0	-	-	-	-	-	-
R/W	R/W	R/W	R	R	R	R	R	R
Initial value	0	0	0	0	0	0	0	0

### INTF1 (External Interrupt Flag 1)

This bit is set, if by the **INT1-Pin** an interrupt condition, according to the configuration, is recognized. If that is set global to Enable interrupt flag, the interrupt routine is started. The flag is deleted automatically, if the interrupt routine is terminated. Alternatively the flag can be deleted, by the value **1 (!)** one writes.

### INTF0 (External Interrupt Flag 0)

This bit is set, if by the **INT0-Pin** an interrupt condition, according to the configuration, is recognized. If that is set global to Enable interrupt flag, the interrupt routine is started. The flag is deleted automatically, if the interrupt routine is terminated. Alternatively the flag can be deleted, by the value **1 (!)** one writes.

### MCU control register.

The MCU control register contains control bits for general MCU functions.

Bit	7	6	5	4	3	2	1	0
Name	-	-	SE	SM	ISC11	ISC10	ISC01	ISC00
R/W	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

### SE (Sleep Enable)

This bit must be set, in order to be able to shift the microcontroller with the **SLEEP instruction** into the sleep condition.

In order not to switch the sleep mode on erroneously, is recommended, to set the bit only directly before execution of the **SLEEP instruction**.

### SM (Sleep Mode)

This bit determines the sleep mode.

If the bit is deleted, then the Idle **mode** is implemented. If the bit is set, then the power down mode is implemented. (for other AVR microcontroller see section “Sleep mode”)

### ISC11, ISC10 (Interrupt Scythe Control of 1 bit)

These two bits determine whether the rising or the falling flank for the interrupt recognition at the **INT1-Pin** is evaluated.

ISC11	ISC10	Meaning
0	0	Low level at <b>INT1</b> produces an interrupt.  In the description it is called, the interrupt is triggered, as long as the pin to 0 remains, thus actually uselessly.
0	1	Reserved
1	0	The falling flank at <b>INT1</b> produces an interrupt.
1	1	The rising flank at <b>INT1</b> produces an interrupt.

### ISC01, ISC00 (interrupt scythe control 0 bits)

These two bits determine whether the rising or the falling flank for the interrupt recognition at the **INT0-Pin** is evaluated.

ISC01	ISC00	Meaning
0	0	Low level at <b>INT0</b> produces an interrupt.  In the description it is called, the interrupt is triggered, as long as the pin to 0 remains, thus actually uselessly.
0	1	Reserved
1	0	The falling flank at <b>INT0</b> produces an interrupt.
1	1	The rising flank at <b>INT0</b> produces an interrupt.

## General look over interrupt processing

If an interrupt arrives, **global the interrupt Enable** bit in the status register **SREG** is deleted automatically and all further interrupts is prevented. Although it is possible to set at this time already again the GIE bit I advise against urgently. This is set automatically, if the interrupt routine is terminated. If in the meantime further interrupts arrive, the associated interrupt bits are set and the interrupts on completion of the current interrupt routine in the order of their priority are implemented. This can actually lead to problems only if a highly priority interrupt arises constantly and in short consequence. This then possibly closes all other interrupts with lower priority. This is one of the reasons, why the interrupt routines are to be kept very short.

## Interrupt with the AVR GCC compiler (WinAVR)

Functions for interrupt processing are made available in the Inc. loading files *interrupt.h* *avr* libc (with older source code additionally *signal.h*).

```
// for sei(), cli() und ISR():  
#include <avr/interrupt.h>
```

The macro **sei ()** switches on the interrupts. Actually nothing else is made, set as **global the interrupt Enable** bit in the status register.

```
sei();
```

The macro **cli ()** switches the interrupts off, or differently said, **global the interrupt Enable** bit in the status register is deleted.

```
cli ();
```

Often one stands before the task that a code sequence may not be interrupted. It is obvious then, at the beginning of this sequence `cli ()` and at the end is to be inserted `()`. This is however unfavorable, if the interrupts before call of the sequence were deactivated and afterwards are to remain also further deactivated. Is `()` regardless of the previous condition the Interrupts would activate, which can lead to unwanted side effects. The proceeding evident from the following example is to be preferred in such cases:



```

#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>

//...

Void NichtUnterbrechenBitte(void)
{
    uint8_t tmp_sreg; // of temporary memory for the status register
    tmp_sreg = SREG; // status register (thus also the I-flag therein) secure
    cli();           // interrupts global deactivate

    /*more unterbrechnungsfreier" code */
    /* example at the beginning of JTAG interfaces of a ATmega16 by software deactivate
    here and thus the JTAG pins at PORTC for "general I/O" usable make without the JTAG Fuse
    bits to change. In addition one is "timed sequence" to keep (vgl data sheet ATmega16,
    conditions 10/04, P. 229): The JTD bit must be written twice within 4 clock cycles. An
    interrupt between the two write accesses the necessary sequence "to break", the JTAG
    interface would remain further active and the IO pins further reserved for JTAG.
    MCUCSR |= (1<<JTD);
    MCUCSR |= (1<<JTD); ); // 2 times in consequence, see data sheet for more information

    /* example end */

    SREG = tmp_sreg; // status registers repair
                    // thus also the I-flag on secured condition set
}

void NichtSoGut(void)
{
    cli();

    /* "more unterbrechnungsfreier" code here */

    sei();
}

int main(void)
{
    //...

    cli();
    // interrupt global deactivates
    NichtUnterbrechenBitte();
    // also after call of the function deactivated
    sei();
    // interrupt global activates

    NichtUnterbrechenBitte();
    // further activates.
    //...

    /* elucidation of the unfavorable proceeding with cli/sei: *
    cli();
    // interrupt global deactivates now

    NichtSoGut();
    after call of the function are interrupts // this are global activated mglw.
    Inadvertently! .
    //...

```

After now the interrupts are activated, it needs naturally still the code which can be required, which is to run off, if an interrupt arrives. In addition the definition (a macro) exists to **ISR**. **SIGNAL** should not be used no more, to the Portierung from **SIGNAL** to **ISR** sees section (TODO: left) in the appendix.

## ISR

(*ISR* ()) replaces with newer versions avr libc the *SIGNALS* (). see [appendix](#))

```
#include <avr/interrupt.h>
//...
ISR (Vectorname) /* before times: SIGNAL (siglabel) with it Vectorname! = siglabel! * /
{
    /* Interrupt Code */
}
```

With *ISR* a function for the treatment of an interrupt is introduced. As argument for it the designation of the appropriate interrupt vector must be indicated. These are to be found in the respective Inc. loading files IOxxxx.h. The designation corresponds to the name from the data sheet, with which through would underline the blanks are replaced and is *\_vect* attached.

As example a cutout from the file of the ATmega8 (with WinAVR standard installation in C:\WinAVR\avr\include\avr\iom8.h) in the meantime the *SIGNAL* (SIG\_\*), no longer current apart from the current names for *ISR* (\*\_vect) still the names for that, are contained.

```

/* $Id: iom8.h,v 1.13 2005/10/30 22:11:23 joerg_wunsch Exp $ */

/* avr/iom8.h - definitions for ATmega8 */
//...

/* Interrupt vectors */

/* External Interrupt Request 0 */
#define INT0_vect          _VECTOR(1)
#define SIG_INTERRUPT0    _VECTOR(1)

/* External Interrupt Request 1 */
#define INT1_vect          _VECTOR(2)
#define SIG_INTERRUPT1    _VECTOR(2)

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect   _VECTOR(3)
#define SIG_OUTPUT_COMPARE2 _VECTOR(3)

/* Timer/Counter2 Overflow */
#define TIMER2_OVF_vect    _VECTOR(4)
#define SIG_OVERFLOW2      _VECTOR(4)

/* Timer/Counter1 Capture Event */
#define TIMER1_CAPT_vect   _VECTOR(5)
#define SIG_INPUT_CAPTURE1 _VECTOR(5)

/* Timer/Counter1 Compare Match A */
#define TIMER1_COMPA_vect   _VECTOR(6)
#define SIG_OUTPUT_COMPARE1A _VECTOR(6)

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect   _VECTOR(7)
#define SIG_OUTPUT_COMPARE1B _VECTOR(7)
//...

```

Possible ones of function trunks for interrupt functions are for example:

```

#include <avr/interrupt.h>
/* becomes outdated: #include <avr/signal.h> */

ISR(INT0_vect) /* becomes outdated: SIGNAL(SIG_INTERRUPT0) */
{
    /* Interrupt Code */
}

ISR(TIMER0_OVF_vect) /* becomes outdated: SIGNAL(SIG_OVERFLOW0) */
{
    /* Interrupt Code */
}

ISR(USART_RXC_vect) /* becomes outdated: SIGNAL(SIG_UART_RECV) */
{
    /* Interrupt Code */
}

// and so on and so on...

```

To the correct way of writing of the vector designation is to be paid attention. The GCC compiler examines only starting from version 4.x whether a signal/an interrupt of the indicated designation is defined in the Inc. loading file actually and spends otherwise a warning. With WinAVR (starting from 2/2005) the examination was

integrated also into the provided compiler of the version 3.x. From the GCC source code version 3.x provided compilers do not contain the examination (see [AVR-GCC](#)).

During the execution of the function all further interrupts are automatically closed. When leaving the function the interrupts become again certified.

If a further interrupt (same or other source of interrupt) should arise during the processing of the interrupt routine, then the appropriate bit in the assigned interrupt flag register is set and the appropriate interrupt routine after terminating the current function is called automatically.

A problem actually results then only if during the processing of the current interrupt routine several homogeneous interrupts arise. The appropriate interrupt routine afterwards called however to know we not whether now the appropriate interrupt arose twice or still more frequently once. Therefore it is to be again stressed here that interrupt routines will possible again leave as fast as only at all should.

## Interruptible Interrupt routine

“Rule of thumb”: in the doubt **ISR**. Those in the following described method use only if one realizes oneself the different function mode.

```
#include <avr/interrupt.h>
//...
void XXX_vect(void) __attribute__((interrupts));
void XXX_vect(void) {
    //...
}
```

Here stands for XXX for described the name of the vector the above (thus e.g. *void TIMER0\_OVF\_vect (void)*...). The difference compared with ISR is that when calling the function **global the Enable interrupts** bits become certified here automatically again set and thus further interrupts. This can to not insignificant problems of in the simplest case a stack OVERFLOW up to other unexpected effects lead and should really be used only if one is absolutely safe itself to have the whole also in the grasp.

See also: References in [AVR-GCC](#)

See in addition: [http://www.nongnu.org/avr-libc/user-manual/group\\_\\_avr\\_\\_interrupts.html](http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html)

## Data exchange with interrupt routines

Variable one both in interrupt routines (ISR = interrupt service routine (s)), and by the remaining program code to be written or read, **volatile** must be defined with. Thus it is communicated to the compiler that contents of the variables before each read access are read from the memory and written after each write access into the memory. Otherwise the compiler could not optimize the code so that the value of the variables becomes only buffered in processor registers, those “anything by the change elsewhere received”.

For illustration a code fragment for a push button denounce with recognition “is enough for pressed” key.

```

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
//...

// threshold values
// Debouncing:
#define CNTDEBOUNCE 10
// "is enough pressed:"
#define CNTREPEAT 200

// here e.g. key to Pin2 PortA "active low" = 0 if pressed
#define KEY_PIN PINA
#define KEY_PINNO PA2

// considers: volatile!
volatile uint8_t gKeyCounter;

// timer Compare interrupt ISR, here e.g. all 10ms implemented
ISR(TIMER1_COMPA_vect)
{
    // is gKeyCounter changed or Remaining
    // program sections must "see" this change:
    // volatile -> current value always into the memory write
    if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
        if (gKeyCounter < CNTREPEAT) gKeyCounter++;
    }
    else {
        gKeyCounter = 0;
    }
}

//...

int main(void)
{
    //...
    /* here: Initialization of the ports and the timer interrupt */
    //...
    // one accesses here on more gKeyCounter. In addition must the value written in
    // ISR admits to be:
    // volatile -> key always read current value from the memory
    if ( gKeyCounter > CNTDEBOUNCE ) { // at least 10*10 ms "bounce-free"
        if (gKeyCounter == CNTREPEAT) {
            /* here: Code for "key is enough pressed" */
        }
        else {
            /* here: Code for "key briefly pressed" */
        }
    }
    //...
}

```

With variables more largely a byte, which in interrupt routines and in the main program it is accessed must be paid attention to the fact that the accesses are not interrupted the individual bytes outside of the ISR by an interrupt. (General place: AVR's are 8-bit microcontroller). To the illustration a code fragments:

```

//...
volatile uint16_t gMyCounter16bit
//...
ISR(...)
{
//...
    gMyCounter16Bit++;
//...
}

int main(void)
{
    uint16_t tmpCnt;
//...
    // not well: Mglw. here an error, if a byte of MyCounter
    // is in tmpCnt copied already however before copying the second byte
    // // an interrupt arises, which changes contents of MyCounter.
    tmpCnt = gMyCounter16bit;

    // better: Changes "outside" prevent - > all "partial bytes"
    // // remain consistent
    cli(); // Interrupts deactivate
    tmpCnt = gMyCounter16Bit;
    sei(); // again activate

    // or: previous status of the global interrupt flag remains received
    uint8_t sreg_tmp;
    sreg_tmp = SREG; /* protection */
    cli()
    tmpCnt = gMyCounter16Bit;
    SREG = sreg_tmp; /* restoring */
//...
}

```

## Interrupt routines and register accesses

If registers are changed both in the main program and in interrupt routines, it is to be made certain that these accesses do not overlap themselves. Only few instructions can be translated into so-called “atomic” accesses, which cannot be interrupted by interrupt routines.

To the illustration an instruction, with which a bit and in the connection three bits in a register are set:

```

#include <avr/io.h>

int main(void)
{
//...
    PORTA |= (1<<PA0);

    PORTA |= (1<<PA2) | (1<<PA3) | (1<<PA4);
//...
}

```

The compiler translates these instructions for an ATmega128 with optimization stage “S” after:

```
...
    PORTA |= (1<<PA0);
d2:  d8 9a          sbi      0x1b, 0 ; 27 (a)

    PORTA |= (1<<PA2)|(1<<PA3)|(1<<PA4);
d4:  8b b3          in       r24, 0x1b      ; 27 (b)
d6:  8c 61          ori      r24, 0x1C      ; 28 (c)
d8:  8b bb          out      0x1b, r24      ; 27 (d)
...
```

Setting of the individual bit is translated during switched on optimization for registers in the lower storage area into only one assembler directive (sbi) and is not susceptible to Unterbrechnungen by interrupts. The instruction for setting three bits is translated however into three dependent assembler directives and offers thereby two “points of attack” for Unterbrechnungen. An interrupt routine could change the value of the register to the shop of the starting situation into the buffer (here register 24), delete e.g. a bit. Thus the buffer no more was not written back however nevertheless with the actual condition to agree after the bit operation (here ori) into the register.

Example: PORTA is at the beginning of 0b00000000. The first instruction (A) sets bit 0, PORTA is thereafter 0b00000001. Now read in the first part of the second instruction the port condition into a register (B). Directly on it (before (C)) an interrupt “fires”, in whose interrupt routine bit 0 is deleted by PORTA. After leaving the interrupt routine PORTA has the value 0b00000000. In the two still following instructions of the main program now the buffered “old” condition 0b00000001 with 0b00011100 logical or verknüft (C) and the result 0b00011101 in PortA written (D). Although in the meantime bit 0 was deleted, it is again set according to (D).

Solution types:

- Registers without special precautions in interrupt routines *and* in the main program do not change.
- Interrupts before changes in registers, which are changed also in ISRs, deactivate (“cli”).
- Bits individually reset or set. sbi and cbi cannot be interrupted. Caution: only registers in the lower storage area are accessible by means of sbi/cbi. The compiler can generate only for these sbi/cbi instructions. For registers outside of this address range (“MEMORY Mapped” - registers) also for the manipulation of individual bits dependent instructions are produced (lds,..., sts).
- see also: [Documentation avr libc of the](#) Frequently asked Questions/questions No. 1 and 8 (conditions: avr libc verse. 1.0.4)

## What makes the main program?

In the simplest (exception) case nothing at all more. It is thus quite conceivable to write a program which remains in the Main function only still the interrupts activated and then in a continuous loop. All functions are then processed in the ISRs. This proceeding is however bad with most applications: in addition one gives away one processing level and has possibly problems by interrupt routines, which need too much working time.

Normally one will let the operations absolutely necessary with the occurrence of the respective interrupt event implement in the interrupt routines only. All less critical tasks are then processed in the main program.

- see also: [Documentation avr libc the](#) section Modules/interrupt and of signal

# Memory accesses

Atmel AVR microcontrollers have typically three memories:

- [RAM](#): In the RAM (more exactly static RAM/CRAM) by the GCC compiler place for variables is reserved. Also the stack is in the RAM. This memory is “volatile”, i.e. contents of the variables are lost with switching off or a collapse of voltage supply.
- Program memory: Implemented as FLASH memory, again-record ably by page. In it the application program is put down.
- [EEPROM](#): Non volatile memory, i.e. once written contents remain also without current supply. Byte by byte write/readably. In the EEPROM typically device-specific values are e.g. put down like calibration values of sensors.

Some AVRs do not only possess a RAM memory, the registers can as “work variables” be used. Since the application of the avr GCC to such “small” control-learn is anyway rarely meaningful and also only with some RAM lots types after “[amateur handicrafts](#)” is possible, these microcontrollers are not continued to consider here. Also EEPROM memory is not available on all types. Generally the following explanations should be transferable to all ATmega microcontrollers and the larger AT90-Typen. To the types ATtiny2313 and ATtiny26 of the “tiny row” apply the remarks likewise.

## RAM

The administration of the RAM memory affected via the compiler, as a rule is not to be considered with the access to variables in the RAM anything special. The explanations in each useful C-book apply also to the code produced by the avr GCC compiler.

## Program memory (Flash)

An access to constants in the program memory is not possible by means of avr GCC “transparency”. I.e. there are necessary special access functions, in order to read data from this memory. In principle all access functions are based on the assembler directive **lpm** (load program MEMORY). The standard run time library of the avr GCC (avr libc) makes these functions available after merging the header file pgmspace.h. With these functions individual bytes, data items know (16bit) and data blocks to be read.

Declarations of variables in the Flash memory are supplemented by the “attribute” **PROGMEM**. Local one variable (actually constants) within functions can be likewise put down in the program memory. In addition is to be placed in front with the definition however *static*, since such “variables” cannot be administered on the stack and/or (during optimization) in registers. The compiler “throws” a warning if static is missing.



```

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <inttypes.h>

//...

/* Byte */
const uint8_t pgmFooByte PROGMEM = 123;

/* Word */
const uint16_t pgmFooWord PROGMEM = 12345;

/* Byte-Field */
const uint8_t pgmFooByteArray1[] PROGMEM = { 18, 3, 70 };
const uint8_t pgmFooByteArray2[] PROGMEM = { 30, 7, 79 };

/* pointer */
const uint8_t *pgmPointerToArray1 PROGMEM = pgmFooByteArray1;
const uint8_t *pgmPointerToArray2[] PROGMEM = {pgmFooByteArray1, pgmFooByteArray2 };
//...

Void foo(void)
{
    static /*const*/ uint8_t pgmTestByteLocal PROGMEM = 0x55;
    static /*const*/ char pgmTestStringLocal[] PROGMEM = "in the Flash";
    // so not (static is missing): char pgmTestStingLocalFalsch [] PROGMEM = "so not";

    // ...
}

```

## Byte read

With the function out `pgmspace.h` affected that `pgm_read_byte` access to the data. Parameter of the function is the address of the byte in the Flash memory.

```

// value of the RAM variables myByte to the value of pgmFooByte sets:
uint8_t myByte;

myByte = pgm_read_byte(&pgmFooByte);
// myByte has now the value 123.

//...

// loop over an array from byte values in the Flash
uint8_t i;

for (i=0;i<3;i++) {
    myByte = pgm_read_byte(&pgmFooByteArray1[i]);
    // mach' which with myByte....
}

```

## Word read

For “simple” 16-bit broad variable that takes place access similarly to the byte example, however with the function `pgm_read_word`.

```
uint16_t myWord;

myWord = pgm_read_word(&pgmFooWord);
```

Pointers on values in the Flash are likewise 16 bits “largely” (conditions avr GCC 3.4.x). Thus the possible storage area for “Flash constants” is limited on 64kB.

```
uint8_t *ptrToArray;

ptrToArray = (uint8_t*)(pgm_read_word(&pgmPointerToArray1));
// ptrToArray does not contain now the start address the byte array pgmFooByteArray1
// however direct accesses with this pointer (e.g. temp=*ptrToArray) a
// ''not'' contents of pgmFooByteArray1 [0] would supply, but of a storage location
// in '' the RAM '', which has the same address as pgmFooByteArray1 [0]
// therefore must now the function pgm_read_byte () is used, which uses the address
contained in ptrToArray
// and the Flash accesses.
for (i=0;i<3;i++) {
    myByte = pgm_read_byte(ptrToArray+i);
    // mach' which with myByte... (18, 3, 70)
}

ptrToArray = (uint8_t*)(pgm_read_word(&pgmPointerArray[1]));

// ptrToArray contains now the address of the first element of the byte array
pgmFooByteArray2
// is there in the second element of the pointer array pgmPointerArray the address
// of pgmFooByteArray2 put down

for (i=0;i<3;i++) {
myByte = pgm_read_byte(ptrToArray+i);
    // mach' which with myByte... (30, 7, 79)
}
```

## Floats and Structs read

In order to pick complex data types out (Structs), not integer data types (floats) from the Flash, auxiliary functions are necessary. Some examples:

```

/* example float from Flash */

float pgmFloatArray[3] PROGMEM = {1.1, 2.2, 3.3};
//...

/* floats from Flash Address ADDR reads and gives these as return VALUE back */
inline float pgm_read_float(const float *addr)
{
    union
    {
        uint16_t i[2]; // 2 16-bit-Worte
        float f;
    } u;

    u.i[0]=pgm_read_word((PGM_P)addr);
    u.i[1]=pgm_read_word((PGM_P)addr+2);

    return u.f;
}
...

void egal(void)
{
    int i;
    float f;

    for (i=0;i<3;i++) {
        f = pgm_read_float(&pgmFloatArray[i]); // according to "f = pgmFloatArray [i];"
        mach' which with f
    }
}

```

TODO: Examples for Structs and pointers out flash on Structs in flash (menu, state machines etc.)

## Simplification for character strings (stringers) in the Flash

Character strings can be proven within the source code as “Flash constants”. In addition the macro serves PSTR out pgmspace.h. This saves the separate declaration with PROGMEM attribute.

```

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <string.h>
#define MAXLEN 30

char StringImFlash[] PROGMEM = "Erwin Lindemann"; // in the "Flash"
char StringImRam[MAXLEN];

//...
strcpy(StringImRam, "Mueller-Luedenscheidt");

if (!strncmp_P(StringImRam, StringImFlash, 5)) {
    // mach' which, if the first 5 indications identically - here not
}
else {
    // the code in here is implemented
}

if (!strncmp_P(StringImRam, PSTR("Mueller-Schmitt"), 5)) {
    // the code was implemented here, the first 5 indications agrees
}
else {
    // with not Uebereinstimmung implemented
}
//...

```

But caution: Replaces one for example

```

const char textImFlashOK[] PROGMEM = "with[]";
// = data in the "Flash", textImFlashOK* points to Flashadresse

```

through

```

const char* textImFlashProblem PROGMEM = "with*";
// conflict: Data in the BSS (read: RAM), textImFlashFAIL* points to Flashadresse

```

then it can come to problems with AVR-GCC. To recognize to the fact that the initialization stringer is put by “textImFlashProblem” to the constants to the end of the program code (BSS), from which it should actually be copied for use in RAM (and becomes). Since nevertheless the reading code (by means of `pgm_read*`) searches in front in a place in the Flash, nonsense is read. This seems to be further a problem of the AVR-GCC (seen with avr GCC 3.4.1 and 3.4.2) with the adjustment to Harvard architecture (constant pointer on variable data?!). Remedy (“Workaround”): Initialization with character strings with `[]` or directly in the code `PSTR()` (“...”) use.

One hands over character strings (more exact: the address of the first indication), those in the Flash are abgelegt to a function, must this be accordingly programmed. The function does not have a possibility to differentiate between whether it concerns an address in the Flash or in the RAM. Avr libc and many other avr GCC libraries adhere to the convention that names of functions expect the Flash addresses with the suffix `_p` (or `_P`) are provided.

A function, which spends a stringer put down in the Flash e.g. to a UART, would look in such a way then:

```

void uart_puts_p(const char *text)
{
    char indications;

    while (indications = pgm_read_byte(text))
    { /* so long, as by means of pgm_read_byte an indication could be read by the
Flash, which does not represent the "string end-character"*/

        /* the read indication over the normal channels sends away */
        uart_putc(indications);
        text++;
    }
}

```

Macros are defined by some libraries, which “automatically” a PSTR inserts on use of a function. A view into the header file of the library shows whether this is the case. An example from **P. Fleury’s** LCD LIBRARY:

```

// cutout from the header file lcd.h the "Fleury LCD Lib."
//...
extern void lcd_puts_p(const char *progmem_s);
#define lcd_puts_P(__s) lcd_puts_p(PSTR(__s))
//...

// // in an application (wieauchimmmmer.c)
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <string.h>
#include "lcd.h"

char StringImFlash[] PROGMEM = "Erwin Lindemann"; // in the "Flash"

//...
    lcd_puts_p(StringImFlash);
    lcd_puts_P("Dr. Kloeber");
    // from it becomes wg. #define lcd_put_P...: lcd_puts_p (PSTR ("Dr. Kloeber"));
//...

```

## Flash in application write

With AVR's with “self programming” - option (also admits as boot loader support) parts of the Flash memory can be described also by the application program. This is possible only if the write functions are put down in a special storage area (boot section) of the program memory/Flash. With few “small” AVR's there is no separate boot section, with these can the Flashspeicher by each place of the program be written. For details is here referred to the respective microcontroller data sheet and the explanations to the module boot.h **avr libc**. Also Application Notes exist in addition with [atmel.com](http://atmel.com) which is transferable to avr GCC code.

## Why so Complicated in such a way?

To the topic, why the Verarbeitung is “complicated” by values from the Flash memory like that, it is here only briefly described: The Harvard architecture of the AVR exhibits separate address areas for program (Flash) - and data memory (RAM). The C-standard and the GCC compiler do not plan different address areas. One string\_an\_uart for example a function (const char\* s) and hands over to this function the address of a character string (a pointer, e.g. 0x01fe), “white” the function not whether the address shows to the Flash memory or/RAM. However from the pointer value (the number) cannot be closed, whether a “simple”

zeichen\_an\_uart (s [i]) or zeichen\_an\_uart (pgm\_read\_byte (&s [i])) must to be used, in order to spend the ith indication.

Some AVR compilers “cheat” somewhat, in which they do not only put on the address for a pointer, but additionally to each pointer the file place (Flash or RAM) internally secure. With call with pointer parameters apart from the address also the storage area, to which the pointer points, will then hand over to a function. This does not however only have advantages; Explanations why this like that is, lead here too far.

see also: [Documentation avr libc of the](#) sections Modules/Program space stringer utilities and section Modules/Bootloader support utilities

## EEPROM

One notes that the EEPROM memory permits only a limited number of write accesses. If one describes an EEPROM cell more frequently than the number assured in the data sheet (typical 100,000), the function of the cell is not any longer guaranteed. This applies to each individual cell. With skillful programming (e.g. ring buffer), with which the cells which can be described is regularly changed, can one a clearly higher number of write accesses, related to the overall memory, achieve.

Writing and read accesses on the EEPROM memory are made by the functions defined in the module eeprom.h. With these functions individual bytes, data items know (16bit) and data blocks to be written and read.

In the case of use of the EEPROM it is to be noted that before the access for this memory it is tested whether the MICROCONTROLLER locked the previous EEPROM operation. The avr libc functions contain this examination, one must it not implements. One should also prevent that that is interrupted access by the processing of an interrupt routine, there determines command successions is given, which must within fewer clock cycles sequences (“timed sequence”). Also this must be implemented on use of the functions from avr libc/eeprom.h file not. Within the functions interrupts before the “EEPROM sequence” is global deactivated and in the connection, if already switched on, new activates before.

With the declaration variable in the EEPROM, is to be supplemented the attribute for the section “.eeprom”. See in addition the following example:

```

#include <avr/io.h>
#include <avr/eeprom.h>
#include <avr/interrupt.h>
#include <inttypes.h> // in current versions avr lib with xx.h merged

// of the EEMEM with current versions avr lib in eeprom.h defined
// here: define if yet does not admit ("old" avr libc)
#ifndef EEMEM
// all passages in the text EEMEM in the source code through __attribute__... replace
#define EEMEM__attribute__ ((section (".eeprom")))
#endif

//...

/* Byte */
uint8_t eeFooByte EEMEM = 123;

/* word */
uint16_t eeFooWord EEMEM = 12345;

/* float */
float eeFooFloat EEMEM;

/* Byte-Field */
uint8_t eeFooByteArray1[] EEMEM = { 18, 3, 70 };
uint8_t eeFooByteArray2[] EEMEM = { 30, 7, 79 };

/* 16-bit unsigned short field */
uint16_t eeFooWordArray1[4] EEMEM;
//...

```

With the tools belonging to the compiler can be written from the variable declaration derived EEPROM contents into a file (usual file dung: .eep, data in Intel the Hex format). Makefiles after WinAVR/MFile collecting main contain already the necessary parameters. Contents of the eep file must be likewise transferred to the microcontroller (Write EEPROM), if the start-up values from the declaration are expected by the program. Otherwise the EEPROM memory contains after the transmission the Programmers by means of ISP on the attitude of the EESAVE Fuse (see data sheet section Fuse bit) the previous data (EESAVE programmed = 0), their position possibly no longer with the allocation in the current program agrees dependent or the default value after “chip Erase”: 0xFF (EESAVE unprogrammed = 1). As safety device one can reproach, during the reading for 0xFF examine the default values new in the program and use a default value if necessary.

## Bytes read/write

Avr libc the function for reading a byte is called `eeprom_read_byte`. Parameter is the address of the byte in the EEPROM. Over the function `eeprom_write_byte` with the parameters address and contents are written. Example of use:

```
//...
uint8_t myByte;

myByte = eeprom_read_byte(&eeFooByte); // read
// myByte have now the value 123
//...
myByte = 99;
eeprom_write_byte(&eeFooByte, myByte); // write
//...
myByte = eeprom_read_byte(&eeFooByteArray1[1]);
// myByte has now the value 3
//...

// example for the "safety device" newst empty EEPROM after "chip Erase"
// (e.g. if the.eep file after programming of a new version
// of the program is not into the EEPROM it became necessary to transfer it and
// EESAVE is deactivated (unprogrammed/1)
//
// caution: if EESAVE is "programmed", this safety device does not continue to help
// since, the memory address meals in a new extended/program
// Then &eeFooByte were possibly shifted.
// in its place stands possibly the value of another variable from a "old" version.

#define EEPROM_DEF 0xFF
uint8_t fooByteDefault = 222;
if ( ( myByte = eeprom_read_byte(&eeFooByte) ) == EEPROM_DEF ) {
    myByte = fooByteDefault;
}
```

## Word read/write

Letter and reading from data items take place similarly to the proceeding with bytes:

```
//...
uint16_t myWord;

myWord = eeprom_read_word(&eeFooWord); // read
// myWord have now the value 12345.
//...
myWord = 2222;
eeprom_write_word(&eeFooWord, myWord); // write
//...
```

## Block read/write

Vintages and letters from data blocks is made by the functions `eeprom_read_block()` and/or `eeprom_write_block()`. The functions expect three parameters: the address of the pouring and/or target data in the RAM, EEPROM-ADDR-eats and the length of the data block in bytes (`size_t`).

TODO: **Caution!** the following examples are not yet examined, first times only as reference to “the principle”. Possibly are missing “casts” and possibly still more.



```
//...
uint8_t myByteBuffer[3];
uint16_t myWordBuffer[4];

/* data block from EEPROM VINTAGES */

/* reads 3 bytes however the EEPROM address defined by eeFooByteArray1
   into the RAM array more myByteBuffer */
eeprom_read_block(myByteBuffer, eeFooByteArray1, 3);

/* meaning somewhat more descriptive however "useless tapping work": */
eeprom_read_block(&myByteBuffer[0], &eeFooByteArray[0], 3);

/* meaning with something security the length concerned */
eeprom_read_block(myByteBuffer, eeFooByteArray1, sizeof(myByteBuffer));

/* and now with "16bit" */
eeprom_read_block(myWordBuffer, eeFooWordArray1, sizeof(myWordBuffer));

/* data LOCK in EEPROM LETTER */
eeprom_write_block(myByteBuffer, eeFooByteArray1, sizeof(myByteBuffer));
eeprom_write_block(myWordBuffer, eeFooWordArray1, sizeof(myWordBuffer));
//...
```

“Not Integer” - data types like e.g. floating point numbers can be converted quite practically over a *union* in “byte arrays” and “back-changed” new. This proves here (however not only here) as useful.

```
//...
float myFloat = 12.34;

union {
    float r;
    uint8_t i[sizeof(float)];
} u;

u.r = myFloat;

/* float in EEPROM */
eeprom_write_block(&(u.i), &eeFooFloat, sizeof(float));

/* float from EEPROM */
eeprom_read_block(&(u.i), &eeFooFloat, sizeof(float));
/* so new 12.34 */
//...
```

Also compound types can be processed with the block routines.

```

//...
typedef struct {
    uint8_t    label[8];
    uint8_t    rom_code[8];
} tMyStruct;

#define MAXSENSORS 3
tMyStruct eeMyStruct[MAXSENSORS] EEMEM;

//...

void egal(void)
{
    tMyStruct work;

    strcpy(work.label, "Flur");
    GetRomCode(work.rom_code);    // dummy for illustration - Rom code sets

    /* protection of "work" in the EEPROM */
    eeprom_write_block(&work, &eeMyStruct[0], sizeof(tMyStruct)); // f. Index 0
    strcpy(work.label, "Bad");
    GetRomCode(work.rom_code);
    eeprom_write_block(&work, &eeMyStruct[1], sizeof(tMyStruct)); // f. Index 1
//...
    /* read of the data EEPROM index 0 in "work" */
    eeprom_read_block(&work, &eeMyStruct[0], sizeof(tMyStruct));
    // work. label has now the contents of "corridor".
//...
}
//...

```

## EEPROM memory map in.eep file

A special function of the avr GCC is that with appropriate options in the Makefile from the start-up values of the variables in the source code a file can be produced, which one can program on the microcontrollers (.eep file). Thus very elegantly default values for EEPROM contents in the source code can be defined. The proceeding becomes evident from the WinAVR example Makefile. See in addition the explanations in the section Exploring Makefiles.

## EEPROM variable on firm addresses put

Equal to beginning I would like to refer to that this procedure is only a Workaround, with which one the problem that “coincidental” distribution of the EEPROM variables by the compiler something into the grasp apparent can get.

Helpfully this can be above all if one e.g. over a command interpreter (o.ä. Functions) directly determined EEPROM addresses to manipulate would like. Even if one liked to manipulate the program sequence over an JTAG adapter (mk I or mkII), by changing the EEPROM values directly, this technology can be helpful.

In the following now two SOURCE listings with an example:

```

////////////////////////////////////
// file "eeprom.h" their own project
////////////////////////////////////

#include <avr/eeprom.h>      // the EEPROM definitions/macros avr libc merge

#define    EESIZE    512      // maximum size of the EEPROM the

#define    ee_dummy    0x000  // dummy element (address 0 should not be used)
#define    ee_value1    0x001  // a byte variable
#define    ee_word1L    0x002  // a Wordvariable (Lowbyte)
#define    ee_word1H    0x003  // a Wordvariable (Highbyte)
#define    ee_value2    0x004  // a entree byte variable

```

With the macros “#define “ee\_value1” one specifies the names and the address of the variables.  
**IMPORTANTLY:** The addresses must be sequential!

Theoretically the compiler should produce an error message, if one uses inadvertently twice the same address for two different variables.

```

////////////////////////////////////
// file "eeprom.c" their own project
////////////////////////////////////

#include "eeprom.h"          // own EEPROM header file merge
uint8_t ee_mem[EESIZE] EEMEM =
{
    [ee_dummy]    = 0x00,
    [ee_value1]   = 0x05,
    [ee_word1L]   = 0x01,
    [ee_word1H]   = 0x00,
    [ee_value2]   = 0xFF
};

```

By the use array, which covers the gesamte EEPROM, does not remain for the compiler different one to platzieren than the array in such a way that element corresponds to 0 of the array of the address 0 of the EEPROM. *(I hope only that those change compiler farmer in it nothing!)*

As one can also see in the above code listing, the procedure has a small hook. Variable those are larger than 1 byte, must somewhat pedantically be defined. That access to the EEPROM of values can e.g. take place then .so:

```

uint8_t    temp1;
uint16_t   temp2;

temp1 = eeprom_read_byte(ee_value1);
temp2 = eeprom_read_word(ee_word1L);

```

Whether in avr libc the existing functions for it can be used, I do not know. But in some cases one must build oneself anyway own functions, which the specific requirements (interrupts - atom problem, etc.) to fulfill.

The possibility described above is only one possibility, how one can realize this. It offers a relatively simple kind the EEPROM values in arbitrary addresses to put or addresses change. The other possibility consists of occupying the EEPROM values as follows:

```
////////////////////////////////////
// file "eeprom.c" their own project
////////////////////////////////////

#include "eeprom.h"          // own EEPROM header file merge

uint8_t ee_mem[EESIZE] EEMEM =
{
    0x00,                    // ee_dummy
    0x05,                    // ee_value1
    0x01,                    // ee_word1L
    0x00,                    // (ee_word1H)
    0xFF,                    // ee_value2
};
```

Here one knows variables, which are larger than more simply define 1 byte and one must define only the Highbyte or Lowbyte address in “eeprom.h”. However one must watch out here höllisch that one slips one or more positions not around!

Which of the two possibilities one begins, it depends above all on the fact how many byte, Word and other variable one use. To accustom one should be able oneself to both variants;)

Small concluding remark:

- The avr GCC supports the variant 1 and the variant 2
- Icc avr the compiler supports only the variant 2!

## Admitted problems with the EEPROM functions

Caution: With old versions avr libc not all AVR MICROCONTROLLERS were supported. E.G. with avr libc the version 1.2.3 with AVRs “the functions do not function to the new generation” (ATmega48/88/168/169) in particular correctly (a cause: different storage addresses of the EEPROM registers). In newer versions (e.g. avr libc 1.4.3 from WinAVR 20050125) the number of the supported MICROCONTROLLERS was extended clearly and a method for easy adjustment to future MICROCONTROLLERS was introduced.

In each data sheet to AVR Microcontroller with EEPROM short example codes for the writing and read access are contained. Wants or if one cannot update on the new version, the code shown there can be used also with the avr GCC (without avr libc/eeprom.h) (“Copy/paste”, if necessary protection from Unterbrechnung/interrupts supplements `uint8_t sreg; sreg=SREG; cli (); [EEPROM code]; SREG=sreg; return; ,` see section interrupt). In the doubt a view helps into the assembler code (lst/iss files), produced by the compiler.

- see also: [Documentation avr libc the](#) section module it/EEPROM handling

## The use of printf

In order to transact comfortably expenditures on a display or the serial interface, offers itself printf/sprintf.

```

#include <stdio.h>
#include <stdint.h>

// ...
// not represented: Implementation of uart_puts (see section UART)
// ...

uint16_t counter;

void uart_puti( uint16_t value )
{
    uint8_t s[20];

    sprintf( s, " count: %d", value );
    uart_puts( s );
}

int main()
{
    counter = 5;

    uart_puti( counter );
    uart_puti( 42 );
}

```

A further elegant possibility, is offered of bending the expenditure stdout on own functions:

```

#include <stdio.h>
int uart_putchar(char c, FILE *stream);
static FILE mystdout = FDEV_SETUP_STREAM( uart_putchar, NULL, _FDEV_SETUP_WRITE );

int uart_putchar( char c, FILE *stream )
{
    if( c == '\n' )
        uart_putchar( '\r', stream );

    loop_until_bit_is_set( UCSRA, UDRE );
    UDR = c;
    return 0;
}

int main(void)
{
    init_uart();
    stdout = &mystdout;
    printf( "Hello, world!\n" );
    return 0;
}

// source: avr-libc-user-manual-1.4.3.pdf, S.74

```

If floating point numbers are to be spent, another (larger) version must be merged printflib in the Makefile.

The disadvantage of all printf variants: They are quite memory-intensive.

# Assembler and Inline assembler

Occasionally it proves as useful to use C and assembler code in an application. Typically the main program in C is written and few, extremely time-critical or hardware near operations in assembler.

The “gnu Toolchain” offers for it two possibilities:

- Inline assembler: The assembler directives are integrated directly into the C-code. A source coding file contains thus C and assembler directives
- Assembler files: The assembler code is in own source coding files. These are assembled by the gnu assembler (avr as) to Object files (“compiles”) and with the Object files provided from the C-code tied together (linked).

## Inline assembler

Inline assembler offers itself, if only few assembler directives are needed. Typical application are short code sequences for time-critical operations in interrupt routines or very precise waiting loops (e.g. 1-Wire). Inline assembler will become with **asm volatile** introduced, the assembler directives in a character string summarized, which is handed over as “parameter”. By colons to be separated the input and output as well as the “Clobber list” indicated. A simple example of Inline assembler is inserting an NOP instruction (NOP stands for NO operation). This assembler instruction necessarily exactly one clock cycle, otherwise “does not do anything”. Meaningful applications for NOP are exact Delay (=Warte) - functions.

```
...
/* retarding the further program execution around
   exactly 3 clock cycles */
asm volatile ("nop");
asm volatile ("nop");
asm volatile ("nop");
...
```

Further it can be prevented with a NOP that empty loops, which are meant as waiting loops are away-optimized. The compiler recognizes otherwise the allegedly useless loop and produces for it no code in the executable program.

```
...
uint16_t i;

/* loop empties - during switched on compiler optimization away-optimized */
for (i = 0; i < 1000; i++)
    ;

...

/* loop force (no optimization): "NOP method" */
for (i = 0; i < 1000; i++)
    asm volatile("NOP");

...

/* alternative method (no optimization): */
volatile uint16_t j;
for (j = 0; j < 1000; j++)
    ;
```

A further more useful “assembler inline” is the call of sleep (*asm volatile (“sleep”);*), since for this no own function exists in avr libc.

As example of Inline assembler of several lines a precise Delay function. The function receives a 16-bit word as parameter, examines the parameter for 0 and terminates the function in this case or goes through the following loop whenever as indicated in the value of the parameter. Inline assembler has here the advantage that the running time independently of the optimization stage (parameter - O, see makefile) and the compiler version is.

```
static inline void delayloop16 (uint16_t count)
{
    asm volatile ( "cp  %A0, __zero_reg__ \n\t"
                  "cpc %B0, __zero_reg__ \n\t"
                  "breq 2f \n\t"
                  "1: \n\t"
                  "sbiw %0,1 \n\t"
                  "brne 1b \n\t"
                  "2: \n\t"
                  : "=w" (count)
                  : "0" (count)
    );
}
```

- Each instruction is locked with `\n\t`. The line-makeup communicates to the assembler that a new instruction begins.
- As branch marks (labels) numbers are used. These special labels are several times in the code usable. One jumps back in each case (B) or forward (f) to the next ausffindbaren label.

The result points a view to the assembler file, which the compiler with the option `- save temps` does not delete:

```
cp  r24, __zero_reg__      ; count
cpc r25, __zero_reg__      ; count
breq 2f
1:
sbiw r24,1                  ; count
brne 1b
2:
```

Detailed remarks about Inline assembler are in the documentation avr libc in the section of the [Related Pages/Inline Asm](#).

See also:

- [AVR assembler directive list](#)
- [German introduction to Inline assembler](#)

## Assembler files

Assembler files receive the ending. S (*large S*) and are listed separately into makefile according to WinAVR/mfile collecting main behind *ASRC*= by blanks.

In the example a function *superFunc*, which switches all pins of the Port D to “exit”, a function *ultraFunc*, which switches the exits according to the handed over parameter, a function *gigaFunc*, which returns the status of Port A and a function *addFunc*, which adds two bytes to a 16-bit-Wort. The assignments in the C-code (PORTx =...) prevent the fact that the compiler calls away-optimized and serves only for the illustration of the parameter transfers.

First the assembler code. The file name is useful. S:



```

#include "avr/io.h"

//; Work register (without "r")
workreg = 16
workreg2 = 17

//; Constant one:
ALLOUT = 0xff

//; ** Set all pins von PortD on exit **
//; for no parameters, no return
.global superFunc
.func superFunc
superFunc:
    push workreg
    ldi workreg, ALLOUT
    out _SFR_IO_ADDR(DDRD), workreg // beachte: _SFR_IO_ADDR()
    pop workreg
    ret
.endfunc

//; ** Set for PORTD to handed over value **
//; Parameter in r24 (LSB always with "degrees" of numbers)
.global ultraFunc
.func ultraFunc
ultraFunc:
    out _SFR_IO_ADDR(PORTD), 24
    ret
.endfunc

//; ** return condition of PINA **
//; Return values in r24: r25 (LSB: MSB), here only LSB used
.global gigaFunc
.func gigaFunc
gigaFunc:
    in 24, _SFR_IO_ADDR(PINA)
    ret
.endfunc

//; ** Two bytes add and return 16-bit-Word;
//; Parameter in r24 (Summand1) and r22 (Summand2)
//; Parameters are always Word "aligned" i.e. LSB of on "degrees"
//; Register numbers. With 8-bits and 16-Bit Paramtern thus
//; beginning with r24 then r22 then r20 etc.
//; Return value in r24: r25
.global addFunc
.func addFunc
addFunc:
    push workreg
    push workreg2
    clr workreg2
    mov workreg, 22
    add workreg, 24
    adc workreg2, 1 // r1 - assumed to be always zero ...
    movw r24, workreg
    pop workreg2
    pop workreg
    ret
.endfunc

//; oh ever - sorry - my AVR assembler is in-rusted, hopes is correct in such a way...

.end

```

In the Makefile the name of the assembler source coding file is to be registered:

```
ASRC = useful. S
```

The call takes place then in the C-code in such a way:

```
extern void superFunc(void);
extern void ultraFunc(uint8_t setVal);
extern uint8_t gigaFunc(void);
extern uint16_t addFunc(uint8_t w1, uint8_t w2);

int main(void)
{
[... ]
    superFunc();

    ultraFunc(0x55);

    PORTD = gigaFunc();

    PORTA = (addFunc(0xF0, 0x11) & 0xff);
    PORTB = (addFunc(0xF0, 0x11) >> 8);
[... ]
}
```

The result becomes new evident in the lss file:

```

[...]  

    superFunc();  

148:  0e 94 f6 00      call    0x1ec  
  

    ultraFunc(0x55);  

14c:  85 e5             ldi     r24, 0x55      ; 85  

14e:  0e 94 fb 00      call    0x1f6  
  

    PORTD = gigaFunc();  

152:  0e 94 fd 00      call    0x1fa  

156:  82 bb             out     0x12, r24      ; 18  
  

    PORTA = (addFunc(0xF0, 0x11) & 0xff);  

158:  61 e1             ldi     r22, 0x11      ; 17  

15a:  80 ef             ldi     r24, 0xF0      ; 240  

15c:  0e 94 ff 00      call    0x1fe  

160:  8b bb             out     0x1b, r24      ; 27  

    PORTB = (addFunc(0xF0, 0x11) >> 8);  

162:  61 e1             ldi     r22, 0x11      ; 17  

164:  80 ef             ldi     r24, 0xF0      ; 240  

166:  0e 94 fc 00      call    0x1f8  

16a:  89 2f             mov     r24, r25  

16c:  99 27             eor     r25, r25  

16e:  88 bb             out     0x18, r24      ; 24  
  

[...]  

000001ec <superFunc>:  

// sets all pins of PortD on exit  

.global superFunc  

.func superFunc  

superFunc:  

    push workreg  

1ec:  0f 93             push    r16  

    ldi workreg, ALLOUT  

1ee:  0f ef             ldi     r16, 0xFF      ; 255  

    out _SFR_IO_ADDR(DDRD), workreg  

1f0:  01 bb             out     0x11, r16      ; 17  

    pop workreg  

1f2:  0f 91             pop     r16  

    ret  

1f4:  08 95             ret  
  

000001f6 <ultraFunc>:  

.endfunc  
  

// sets PORTD to handed over value  

.global ultraFunc  

.func ultraFunc  

ultraFunc:  

    out _SFR_IO_ADDR(PORTD), 24  

1f6:  82 bb             out     0x12, r24      ; 18  

    ret  

1f8:  08 95             ret  
  

000001fa <gigaFunc>:  

.endfunc  
  

// condition of PINA return  

.global gigaFunc  

.func gigaFunc  

gigaFunc:

```

```

    in 24, _SFR_IO_ADDR(PINA)
1fa:  89 b3          in      r24, 0x19      ; 25
    ret
1fc:  08 95          ret

000001fe <addFunc>:
.endfunc

// two bytes add and 16-bit-Word return
.global addFunc
.func addFunc
addFunc:
    push workreg
1fe:  0f 93          push    r16
    push workreg2
200:  1f 93          push    r17
    clr workreg2
202:  11 27          eor     r17, r17
    mov workreg, 22
204:  06 2f          mov     r16, r22
    add workreg, 24
206:  08 0f          add     r16, r24
    adc workreg2, 1 // r1 - assumed to be always zero ...
208:  11 1d          adc     r17, r1
    movw r24, workreg
20a:  c8 01          movw   r24, r16
    pop workreg2
20c:  1f 91          pop     r17
    pop workreg
20e:  0f 91          pop     r16
    ret
210:  08 95          ret

[ ... ]

```

The assignment from registers to parameter number and the registers for the return values are described in “registers Usage Guidelines” the avr libc documentation.

See also:

- [Avr libc documentation: Related Pages/avr libc and assembler programs](#)
- [Avr libc documentation: Related Pages/FAQ/“What of register of acres used by the C compilers?”](#)

## Global variables for data exchange

Often one does not come around global variables to e.g. realize in order data exchange between main program and interrupt routines. For this one must know in the assembler, where exactly the variable is stored by the C-compiler.

For this the variable, here “counters”, must be called be defined first in the C-code as global, e.g. so:

```

#include <avr/io.h>

volatile uint8_t zaehler;

int16_t main (void)
{
    // any code, in which counter can be used
}

```

In the following assembler example the external Interrupt0 is used, in order to count up the counter. The initializations of the interrupt and the interrupt enable are missing, so correctly meaningfully are the code also not, but it shows (hopefully) like it goes.

In handling interrupt vectors the same applies, as with C with the GCC assembler: One must consider the accurate way of writing, otherwise not the interrupt vector is put on, but a new function - and one is surprised that nothing functioning (see the AVR GCC manual).

```

#include "avr/io.h"

temp = 16

.extern zaehler

.global INT0_vect
INT0_vect:

    push temp                                ;; importantly: Register used and
    in temp, _SFR_IO_ADDR(SREG)             ;; Status register (SREG) secure!
    push temp

    lds temp, zaehler                        ;; Worth from the memory read
    inc temp                                ;; work on
    sts zaehler, temp                        ;; and new write back

    pop temp                                ;; the used registers re-establish
    out _SFR_IO_ADDR(SREG), temp
    pop temp
    reti

.end

```

## Global variables in the assembler file put on

Alternatively variables can be put on in addition, in the assembler file. Thus can be done without a.c-file. For the above example the source text would know then the files zaehl\_asm. S and zaehl\_asm.h to be put down, so that only **zaehl\_asm. S** with to be compiled would have.

Instead of referring in the assembler file over the keyword *.extern* to an existing variable, in addition with the keyword *.com m* the necessary number of bytes for a variable is reserved.

### zaehl\_asm.S

```
#include "avr/io.h"

temp = 16

//; 1 byte in the RAM for the counter reserve
.comm zaehler, 1

.global INT0_vect
INT0_vect:

...
```

In the header file then to the variable only one refers (keyword *extern*):

### zaehl\_asm.h

```
#ifndef ZAEHL_ASM_H
#define ZAEHL_ASM_H

extern volatile uint8_t zaehler;

#endif
```

Contrary to global variables in C variables so put on are not initialized automatically with the value 0.

### Variable one more largely than 1 byte

Variable ones, which are larger than a byte, can be responded in assembler to similar kind. For this enough bytes must be only requested, in order to take up the variable. If a variable is to be used of the type *unsigned long*, thus *uint32\_t* e.g. for the counter, then 4 bytes must be reserved:

```
...
// 4 byte in the RAM for the counter reserve
.comm zaehler, 4
...
```

The pertinent declaration in the header file would then be:

```
...
extern volatile uint32_t zaehler;
...
```

With variables, which are larger than a byte, the values are put down beginning with the least significant byte in the RAM. The following Codes shows, how can be accessed under assembler the individual bytes. In addition in the interrupt now a 32-Bit counter is increased:

```

#include "avr/io.h"

temp = 16

// 4 byte in the RAM for the counter reserve(zaehler = counter "just as notice "Ahmed)
.comm zaehler, 4

.global INT0_vect
INT0_vect:

    push temp                                // importantly: Used registers in temp
    in temp, _SFR_IO_ADDR(SREG)             // status register (SREG) secure!
    push temp

    // 32-Bit-counter increment
    lds temp, (zaehler + 0)                 // 0. Byte (low order width unit byte)
    inc temp
    sts (zaehler + 0), temp
    brne RAUS

    lds temp, (zaehler + 1)                 // 1. Byte
    inc temp
    sts (zaehler + 1), temp
    brne RAUS

    lds temp, (zaehler + 2)                 // 2. Byte
    inc temp
    sts (zaehler + 2), temp
    brne RAUS

    lds temp, (zaehler + 3)                 // 3. Byte (most significant Byte)
    inc temp
    sts (zaehler + 3), temp
    brne RAUS

RAUS:
    pop temp                                // the used registers re-establish
    out _SFR_IO_ADDR(SREG), temp
    pop temp
    reti

.end

```

**TODO:** 16-Bit/32-Bit variable, access to arrays (stringers)

# Appendix

## Characteristics with the adjustment existing source code

Some functions, which were present in früheren versions avr libc, are regarded in the meantime as outdated. They are missing any longer or than *deprecated* (disapproved) proven and definitions in <compat/deprected.h> shifted. It is advisable not to use existing code to portieren any longer and even the old functions if these are still available.

### Became outdated functions for the declaration of interrupt routines

The functions (actually macros) *SIGNAL* and *INTERRUPT* for the declaration of interrupt routines should not be used no more.

In current versions avr libc (e.g. avr libc 1.4.3 from WinAVR 20060125) interrupt routines, which cannot **be interrupted** by other interrupts, with ISR are defined (see section in the main part). Also the designation were standardized and adapted to the usual designations into the AVR data sheets. In the documentation avr libc old and new designations in the table are confronted. The necessary steps to the Portierung:

- #include from avr/signal.h remove
- SIGNAL durch ISR replace
- Name of the interrupt vector adapt (SIG\_\* \*\_vect by appropriate)

As example of the adjustment first a “old” code:

```
#include <avr/interrupt.h>
#include <avr/signal.h>

...

/* Timer2 output Compare with a ATmega8*/
SIGNAL(SIG_OUTPUT_COMPARE2)
{
    ...
}
```

In the data sheet the vector with TIMER2 COMP is designated. The designation in avr libc corresponds to the name in the data sheet, blank through would underline (\_) replaced and \_vect attached.

The new code looks then in such a way:

```
#include <avr/interrupt.h>
/* signal.h escapes */

ISR(TIMER2_COMP_vect)
{
    ...
}
```

With ambiguity concerning the new vector labels (still) a view helps into the header file of the appropriate MICROCONTROLLER. For the previous example thus the view into the file iom8.h for the ATmega8, there one finds the outdated designation below the current.



```

...
/* $Id: iom8.h,v 1.13 2005/10/30 22:11:23 joerg_wunsch Exp $ */
/* avr/iom8.h - definitions for ATmega8 */

...

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect      _VECTOR(3)
#define SIG_OUTPUT_COMPARE2   _VECTOR(3)
...

```

For **interruptible** interrupt routines, which are defined by means of *INTERRUPTS*, there is no direct replacement in form macros. Such routines are to be defined according to documentation avr libc in the following form:

```

void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
    ...
}

```

Example:

```

/* ** old ** */
#include <avr/io.h>
#include <avr/interrupt.h>

//...

INTERRUPT(SIG_OVERFLOW0)
{
    ...
}

/* ** new: ** */
#include <avr/io.h>

//...

void TIMERO_OVF_vect(void) __attribute__((interrupt));
void TIMERO_OVF_vect(void)
{
    ...
}

```

Wants or if one cannot portieren the code, the header file is to be merged *compat/deprecated.h* for the further use of *INTERRUPTS*. One should however new examine on this occasion whether the functionality of *INTERRUPTS* is actually intended. In many cases *INTERRUPT* was used, where *SIGNAL* (now *ISR*) should actually have been used.

## Became outdated functions to the Port access

*inp* and *outp* to reading in and/or writing registers are not necessary any longer, the compiler supports this without this detour.

```

unsigned char i, j;

// old:
i = inp(PINA);
j = 0xff;
outp(PORTB, j);

// new (no longer really new...):
i = PINA
j = 0xff;
PORTB = j;

```

Wants or if one cannot portieren the code, the header file is to be merged **compat/deprecated.h** for the further use of inp and outp.

## Became outdated functions to the access to bits in registers

*cbi* and *sbi* to resetting and setting bits are not necessary any longer, the compiler supports this without this detour. The designation is the functions only for registers with addresses in the lower storage area into the assembler directives *cbi* and *sbi* to be actually translated anyway misleading there.

```

// old:
sbi(PORTB, PB2);
cbi(PORTC, PC1);

// new (also not more really new...):
PORTB |= (1<<PB2);
PORTC &= ~(1<<PC1);

```

Wants or if one cannot portieren the code, the header file is to be merged **compat/deprecated.h** for the further use of *sbi* and *cbi*. Who wants absolutely, can naturally own macros with more meaningful names define itself. For example:

```

#define SET_BIT(PORT BITNUM) ((PORT) |= (1<<(BITNUM)))
#define CLEAR_BIT(PORT BITNUM) ((PORT) &= ~(1<<(BITNUM)))
#define TOGGLE_BIT(PORT BITNUM) ((PORT) ^= (1<<(BITNUM)))

```

## Self defined (non-standardized) integral data types

With in the following the type definitions mentioned it is to be noted that the names for “words” are partly used depending upon processor platform differently. The indicated definitions refer to the “bit widths” usual in connection with AVR/8-bit-Microcontrollern (in explanations concerning the ARM7TDMI often 32-bit Integer with “word” are e.g. designated without further addition). It is advisable to use during the revision of old code the *integral data types standardized* in the section described data types (stdint.h) and prevent thus “misunderstandings”, which can occur e.g. with the Portierung of C-code between different platforms.

```

typedef unsigned char    BYTE;        // better: uint8_t out <stdint.h>
typedef unsigned short   WORD;        // better: uint16_t out <stdint.h>
typedef unsigned long     DWORD;      // better: uint32_t out <stdint.h>
typedef unsigned long     QWORD;      // better: uint64_t out <stdint.h>

```

## BYTE

The data type BYTE defines a variable with 8-bit width for the representation of whole numbers within the range between 0... 255.

## WORD

The data type WORD defines a variable with 16 bits long for the representation of whole numbers within the range between 0... 65535.

## DWORD

The data type DWORD (spoken: Double Word) defines a variable with 32 bits long for the representation of whole numbers within the range between 0... 4294967295.

## QWORD

The data type QWORD (spoken: Quad Word) defines a variable with 64 bits long for the representation of whole numbers within the range between 0... 18446744073709551615.

# Additional functions into the Makefile

## Libraries (Libraries/.a files) ADDs

In order to use functions from Libraries (“genuine” Libraries, \*.a-files), the linker the names OF the Libraries of are passed over as parameter. In addition the option is intended -l (small L), which the name OF the LIBRARY is attached.

It is noted that the name the LIBRARY and the file name OF the LIBRARY of are emergency identical. Behind -l indicated name corresponds to the file name OF the LIBRARY without the characters sequence *lib* at the beginning OF the file name and without the ending *.a*. being supposed functions from A LIBRARY with the file name *libefsl.a* (linked) becomes merged e.g., READ the appropriate parameters -l efsl (like thus -lm for tying up libm.a).

In traditional Makefiles A make variable LDLIBS is used, into which “l-parameter” of are PUT down. The WinAVR Makefile collecting Main does emergency contain this variable, this represents however NO restriction, since all into the make variable LDFLAGS PUT down parameter to that the linker of are passed on.

Examples:

```
# merging functions from a LIBRARY efsl (file name libefsl.a)
LDFLAGS += -l efsl
# merging functions from a LIBRARY xyz (file name libxyz.a)
LDFLAGS += -l xyz
```

If the LIBRARY files do not lie in the standard LIBRARY search path, the paths are by means of parameters -L to likewise indicate. (The pre-defined search path can by means of *avr GCC --print search dirs* to be indicated.)

As example a project (“superapp2”), in which the source code by two Libraries (efsl and xyz) and the source code of actual application are put down in different listings with the following “tree structure”:

```
superapp2
|
+----- efslsource (in it libefsl.a)
|
+----- xyzsource (in it libxyz.a)
|
+----- firmware (in it application source code and Makefile)
```

Hence it follows that in the Makefile the listing and xyzsource are to be taken up efsIsource to the LIBRARY search path: .

```
LDFLAGS += L.. /efsIsource/ L.. /xyzsource/
```

## Fuse bit

For the computation of the Fuse bits also the [AVR Fuse Calculator](#) offers itself apart from the study of the data sheet. To be warned must before the use of PonyProg, because by the negated representation gladly errors are made there.

If the programming of Fuse and LOCK bits is to be automated, one can likewise make this by entries in the Makefile, which is handed over with the call of “make program” to the used programming software. In the Makefile collecting main of WinAVR (and mfile) there is for it however no “filling out assistance” (conditions 9/2006). The following remarks apply to the programming software [AVRDUDE](#) (standard in the WinAVR collecting main), can be transferred however in a general manner to other programming software, which supports the indication of the Fuse and LOCK bit parameters by command line parameter (e.g. stk500.exe). In the simplest case one supplements some variables, whose values naturally depend on the used MICROCONTROLLER and the desired parameters in the Makefile (see data sheet Fuse /Lockbits):

```
#----- Programming Options (avrdude) -----
#...
#example! f. ATmega16 - not simply take over! Numerical values is more near described on
#----- the data sheet reconstruct and if necessary change.
#
AVRDUDE_WRITE_LFUSE = -U lfuse:w:0xff:m
AVRDUDE_WRITE_HFUSE = -U hfuse:w:0xd8:m
AVRDUDE_WRITE_LOCK = -U lock:w:0x2f:m
#...
```

So that these variables are also used, should the call of avrdude in the Makefile accordingly to be supplemented:

```
# Program the device.
program: $(TARGET).hex $(TARGET).eep
# without Fuse /Lock parameters (after WinAVR collecting main conditions 4/2006)
#     $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_FLASH) \
#     $(AVRDUDE_WRITE_EEPROM)
# with Fuse /Lock parameters
#     $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_LFUSE) \
#     $(AVRDUDE_WRITE_HFUSE) $(AVRDUDE_WRITE_FLASH) \
#     $(AVRDUDE_WRITE_EEPROM) $(AVRDUDE_WRITE_LOCK)
```

Far possibility consists of letting the Fuse and LOCK bit parameters of the pre-processor/compiler generate. The Fuse bits are then written when using AVRDUDE into own Hex files. For this one can use the e.g. following Construct:

Into one the C-Source is defined a variable for each Fuse byte of the type *unsigned char* and packed into extra a section. This can be passed either in an existing file or be written into a new (e.g. fuses.c). The file must be compiled and linked in the Makefile however in any case with.

```

// tiny 2313 fuses low byte
#define CKDIV8 7
#define CKOUT 6
#define SUT1 5
#define SUT0 4
#define CKSEL3 3
#define CKSEL2 2
#define CKSEL1 1
#define CKSEL0 0

// tiny2313 fuses high byte
#define DWEN 7
#define EESAVE 6
#define SPIEN 5
#define WDTON 4
#define BODLEVEL2 3
#define BODLEVEL1 2
#define BODLEVEL0 1
#define RSTDISBL 0

// tiny2313 fuses extended byte
#define SELFPRGEN 0

#define LFUSE __attribute__((section("lfuses")))
#define HFUSE __attribute__((section("hfuses")))
#define EFUSE __attribute__((section("efuses")))

// select ext crystal 3-8Mhz
unsigned char lfuse LFUSE =
    ( (1<<CKDIV8) | (1<<CKOUT) | (1<<CKSEL3) | (1<<CKSEL2) |
      (0<<CKSEL1) | (1<<CKSEL0) | (0<<SUT1) | (1<<SUT0) );
unsigned char hfuse HFUSE =
    ( (1<<DWEN) | (1<<EESAVE) | (0<<SPIEN) | (1<<WDTON) |
      (1<<BODLEVEL2) | (1<<BODLEVEL1) | (0<<BODLEVEL0) | (1<<RSTDISBL) );
unsigned char efuse EFUSE =
    ((0<<SELFPRGEN));

```

NOTE: The bit locations were not completely tested !

A “1” means here that the Fuse bit is not programmed - the function thus is in general not activated. “0” however activate most functions. This is as in the data sheet (1 = unprogrammed, 0 = programmed).

The Makefile must be extended now still by the following targets (with tabulator to engage - not with blanks):

```

lfuses: build
-$(OBJCOPY) -j lfuses --change-section-address lfuses=0 \
-O ihex $(TARGET).elf $(TARGET)-lfuse.hex
@if [ -f $(TARGET)-lfuse.hex ]; then \
$(AVRDUDE) $(AVRDUDE_FLAGS) -U lfuse:w:$(TARGET)-lfuse.hex; \
fi;

hfuses: build
-$(OBJCOPY) -j hfuses --change-section-address hfuses=0 \
-O ihex $(TARGET).elf $(TARGET)-hfuse.hex
@if [ -f $(TARGET)-hfuse.hex ]; then \
$(AVRDUDE) $(AVRDUDE_FLAGS) -U hfuse:w:$(TARGET)-hfuse.hex; \
fi;

efuses: build
-$(OBJCOPY) -j efuses --change-section-address efuses=0 \
-O ihex $(TARGET).elf $(TARGET)-efuse.hex
@if [ -f $(TARGET)-efuse.hex ]; then \
$(AVRDUDE) $(AVRDUDE_FLAGS) -U efuse:w:$(TARGET)-efuse.hex; \
fi;

```

The target “clean” must still around the lines

```

$(REMOVE) $(TARGET)-lfuse.hex
$(REMOVE) $(TARGET)-hfuse.hex
$(REMOVE) $(TARGET)-efuse.hex

```

are extended, even if the Fuse files are to be deleted.

Around now the Fuse bits of the attached MICROCONTROLLER to program must be only started “make lfuses”, “make hfuses” or “make efuses”. With the Fuse bits special caution is required, since these can make a programming of the MICROCONTROLLER impossible. Program thus only if one has a HV-programmer handy or a few reserve AVR; -)

In order to be able to describe further the “normal” Flash, it is important, for the target “\*.hex” in the Makefile not only “- R .eeprom ” as parameters to hand over separately still “- to R of lfuses - R of efuses - R of hfuses”. Otherwise AVRDUDE of problems gets these sections in the Flash (where they do not belong) to write.

See also :[Comparison of the Fuses with different programs](#)

## External reference tension of the internal analogue-digital converter

The minimum (external) reference tension of the ADC may be not arbitrarily low, see in addition (most current) the data sheet of the used MICROCONTROLLER. e.g. with the ATMEGA8 it may not fall below 2,0V according to data sheet (S.245, table 103, line “VREF”). NOTE: this information is only in the last revision (Rev. 2486O-10/04) of the data sheet.

After my own experience one can however (on own danger and naturally not for standard sets) small little far fallings, with the ATMEGA8L taken by me under the magnifying glass (thus the Low Voltage variant) quite still functions the ADC with 5V operating voltage with up to VREF=1,15V down correctly, starting from 1,1V

and among them digitizes he however only Blödsinn). I would go for safety's sake not under 1,5V and with lower operating voltages like itself the lower bound for VREF at the pin AREF if necessary upward (!) shift.

In addition in the last revision of the data sheet it is corrected that ADC4 and ADC5 very probably offer 10 bits accuracy (and not only 8-bits, as indicated in older revisions erroneously.)