Application Notes

This page and all linked documents have been written by Leitner Harald.

The creation of makefiles

Makefiles provide the possibility of translating own programs efficiently.
Here I describe how you have to alter the makefiles for your own programs.
makedoc.pdf

The combination of C and assembler instructions

This document describes the possibility of inserting the assembler instructions in your C-programs directly or through macros.
asmdoc.pdf

The creation of variables, fields and constants how to use them?

The Atmel AVR processor series exist of 3 different memory forms. Here you find the emphasis on working with the different possibilities.

SRAM: sramdoc.pdf
Program memory: progmemdoc.pdf
Eeprom: eepromdoc.pdf

How can you use I/O - registers and ports ?

Here you can find the description of the programming of I/O - registers and especially ports.
iodoc.pdf

How can you create an interrupt routine ?

Which interrupt routines can be created ?
Explainations are shown on the example of an external interrupt (INT0 and INT 1).
irqdoc.pdf

Properties and use of the timer / counter

How can you program and use the different modes of the timer/counter peripherial ?

8 bit Timer / Counter 0 : t0doc.pdf
16 bit Timer / Counter 1: t1doc.pdf
8 bit Timer 2 : t2doc.pdf

How are data transmitted through a seriell interface ?

Receiving and transmitting of data with the UART module in the AVR controller.
uartdoc.pdf

Analog values processing digital

Using the 10bit Analog/Digital converter in the single conversion or free running mode.
adcdoc.pdf

Comparing of analog values

How are two analog values compared with means of a comparator.
acompdoc.pdf

How to adapt a *makefile* for your programms:

In order to create an executable program (*ownprog.rom*) from a source file (*ownprog.c*) one ought to have some instructions.
They need to be repeated after each change of program.
To avoid this there is a more efficient possibility.
The instructions are summarized and stored in a so called *makefile*.
In order to carry out this *makefile* you need to have the program *make*, which has already been integrated in this compiler area.
The *makefile* and the source file should be in the same directory.

The *makefile* is carried out step by step through calling up the instruction make and so one gets the wanted program.
The *makefile* needs two include files (*make1,make2*) as well. Both files contain parts of the whole makefile, which don´t have to be changed for the use for the most part. Both files you can find in the directory *($avr)/include.*

For further information look up the program *make* and its helpfile.

The structure of a possible makefile:

This part includes the *make1* file of the include directory
include $(AVR)/include/make1

#Put the name of the controller you have used here.#
MCU = at90s8535

#At this place you can insert a new name for the program but you should better keep the original name.#
TRG = ownprog

#These are names of your programs which should be translated.#
SRC = owninc1.c owninc2.c ownprog.c

#Put additional assembler source files here#
ASRC =

#Additional libraries and object files to link#
LIB =

#Additional includes to compile#
INC =

#Compiler flags#
CPFLAGS =-g -O3 -Wall -Wstrict-prototypes -Wa,-hlms=$(<:.c=.lst)

#Assembler flags#
ASFLAGS = -gstabs

#Linker flags#
LDFLAGS =

#Here the makefile includes the *make2* file - which
don´t need to be changed.#
include $(AVR)/include/make2

#Dependencies: The program *make* knows with its dependencies
which parts of the program have to be translated or not.
If *owninc1* is changed, only the program itself and the program
which includes *owninc1*, is translated. *owninc2* is not translated
again because there is no dependency to *owninc1*.#

ownprog.o :ownprog.c owninc1.h owninc2.h
owninc1.o :owninc1.c owninc1.h
owninc2.o :owninc2.c owninc2.h

How to integrate assembler instruction:

Sometimes you need functions for programming that cannot be carried out through a C instruction or it seems easier to write a few assembler instructions. There are two possibilities to add an assembler code into the C sourcefile.
The first possibility is to write the assembler instructions directly into the sourcecode and the second one is to write macros.
Macros are always used in case you need the implemented function more often. The macros are always directly added at the place where they are called. Suitable assembler instructions you need for programming you can take out of the instructional sentence of the AVR processor series. The keyword *asm* must be together with the instructions so that the compiler realises that assembler instructions are given.
You can put multiple assembler instructions together in a single asm template, separated wither with „*n*“ or with semicolons.
Further labels can be defined that can be called up through jump instructions.
It’s not possible to jump from one *asm* to another *asm* area.
After the input of the instructions, which are put in inverted commas, you can specify the characteristics of the used input variables, output variables or registers.
A colon separates the assembler template from the first output operand and another separates the last output operand from the first input operand.

If there are no output operands but there are input operands, you must place two consecutive colons that surround the place where the output operands would go.

In addition to that the qualities of the operands can be determined through constrains. Constrains can say whether an operand may be a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; wheter the operand may be an immediate constant, an which possible values it may have. Constrains can also require two operands to match. („=*r*“)
To make sure the instruction operates on the correct data type you need to use a typecast in the asm statements.
(for example: *:“r“ ((uint8_t)(val))*)

Constrains:

The ordinary output operands must be of the state „write-only“.

Often used constrains:

„*r*“ The value is handed through a general register.

„*i*“ An immediate integer operand (0-255) is allowed. This includes symbolic constants whose values will be known only at assembly time.

„*I*“ „I“ is defined to stand for the range of values 0 to 63.

„*z*“ The transfered value is loaded into the *z*-register
(low byte -> R30, high byte R31)

„*y*“ The transfered value is loaded into the *y*-register
(low byte -> R28, high byte R29)

„*x*“ The transfered value is loaded into the *x*-register
(low byte -> R26, high byte R27)

For the output parameters you have to put a „=“ in front of the constrain parameter.
Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be re-used later if it happens to be found in a register. You can prevent an asm instruction from being deleted, moved significantly or combined by writing the keyword volatile after the asm command.If you write an asm instruction with no output the compiler knows that the instruction has side effects and will not delete them. If you write a header file write __*asm*__ instead of asm.

Filling in the assembler instructions directly:

Construction:

```
...
asm volatile („operator operand;operator operand....“
        : input variables
        : output variables
        : used registers)
...
```

example:

```
/* a simple program to show the usage of assembler code in
        your c- programs

        8/00 Leitner Harald */

#include<io.h>

uint8_t val1, val2;

int main(void)
{
outp(0xff,DDRB);                /* use all pins on PortB for output */
val1 = 3;
val2 = 4;

asm volatile
("LDS R30,val1;                 /* start of the asm part*/
        LDS R31,val2;
        ADD R30,R31;
        RJMP do;                /* definition of jump labels */
        LDI R30,0xFF;
        do:STS val1,R30;"
        ::);

outp(~val1,PORTB);              /* write result of val1+val2 to Port */
for (;;){}
        }
```

Insert the assembler instructions through macros:

The structure is similar to the normal insertion of assembler instructions. Through the creation of such macros and the integration under C you can easily make complex instructions.

Construction:

```
# define macroname(variablelist) \
        ({ variables declaration list \
        asm („       operator operand \
                operator operand“\
                : output parameters\
                : input  parameters\
                : used registers)\ })
```

The essential difference is that no fixed variables are indicated but placeholder who refer to the variables.
These placeholders could also stand for bigger types of variables. For

example:

1Byte :

```
#define xyz (val) (\
        asm volatile (
        „mov r30, %0“ \
        : /* no output */ \
        : „r“ (val)\
        )
```

The placeholder *%0* stands for the variable *val*.

2Byte:

```
#define xyz (address) (\
        asm volatile (
        „mov r30, %A1 \
        mov r30, %B1“ \
        : /* no output */ \
        : „r“ (address)\
        )
```

address is a 16 bit value which is seperated through %A1 and %B1 to a low and high byte.

Example:

```
/* simple program to show the usage of assembler code in
your c-programs and how to define and run macros

 8/00 Leitner Harald*/

#define outport(val, port)\
        asm volatile (\
                        "out %1, %0"\
                        : /* no outputs */ \
                        : "r" ((uint8_t)(val)),\
                          "I" ((uint8_t)(port))\
        )

#define DDRB    0x17
#define PORTB  0x18

typedef unsigned char uint8_t;

uint8_t val1;
uint8_t val2;

int main(void)
{                                       /* call output macro and */
outport(0xff,DDRB);             /* use all pins on PortB for output */
val1 = 2;
val2 = 4;

asm volatile (
                "LDS R30,val1;
        LDS R31,val2;
                ADD R30,R31;
        STS val2,R30":: );

outport(~val2,PORTB);           /* call output macro and write result to PortB
(Leds) */
for (;;){}
}
```

The macro *outport* has two input parameters *val* and *port.*
The placeholder *%0* stands for the variable *val* and *%1* for *port*.

For the input parameters the type of both is mentioned (*(uint8_t)*).
The field you can mention optionally for the used registers is not used.

Variables in the SRAM:

Variables that are defined in the program without any attributes are placed by the compiler into the SRAM of the controller.
This memory is directly integrated, not with I/O registers.
The advantage of this is that the reading and writing is faster than in other memory forms and you don't need other approaching functions.

Frequently used integer data types are summarized in the include file <*inttypes.h*>.
As soon as the file <*io.h*> is included, the data types of <*inttypes.h*> are automatically included as well.
Constantes are filed trough giving the keyword *const*. This variables are of the state „read only" and cannot be altered.

Pre-defined variables in the include file <*inttypes.h*>:

name of the data typ	length in byte	value range
Int8_t	1	-128 to 127
Uint8_t	1	0 to 255
Int16_t	2	-32768 to 32767
Uint16_t	2	0 to 65535
Int32_t	4	-2147483648 to 2147483647
Uint32_t	4	0 to 4294967295
Int64_t	8	-9.22*10^18 to 9.22*10^18
Uint64_t	8	0 to 1.844*10^19

Definition:

Creation of a variable in the SRAM:

```
Uint_8 val = 8;
```

Through this instruction a new variable *val* in the SRAM is filed and initialized with value 8. The variable reserves a one byte value.

Alteration of the variable in the SRAM:

Val = 10;

Because of that the value that is on the given address that is mentioned through val is altered.

Creation of a constante in the SRAM:

Const uint8_t october = 10;

Through this instruction a constante in SRAM is filed that is of the value 10. The name of this constante is october and can only be read but not written, because of that an alteration in the value is not possible.

The filing of register variables:

Register uint8_t alwaysused = 10;

Through this instruction variables are created that are assigned to a register. This is especially useful for variables that are used frequently.

Variables in the program memory:

The memory of data in the program memory is very useful because the data are filed from the beginning and don‘t have to be created first.
A possible application is the filing of the different sentences which have to be shown on a lcd display, in the program memory because they don’t have to be altered.
You could file the constants in the SRAM as well, but as however the program memory is larger than the SRAM and as the data don’t have to be altered the memory in the program memory is a better alternative.
When you download the program, the defined constantes are filed as a part of the program.
The creation of these data is through the keyword __*attribute*__ *((progmem)).* In order to avoid mentioning the keyword after the creation of a data record, suitable data types have already been designed in the include file <*progmem.h*>*.*
This file also includes all access procedures that are necessary for the reading of the data.

8 bit data typ:

prog_char

Singular value:

The creation of a constante in the program memory:

prog_char LINE = {1};

Here value 1 is filed in the program memory.
LINE obtains the address of where the value is situated.

The reading of the constants:

char res = PRG_RDB(&LINE)

Through the address of LINE the value can be read and memorized in the variable *res* that is situated in the SRAM.

Array:

The creation of an array in the program memory:

prog_char TEN[10] = {0,1,2,3,4,5,6,7,8,9};

Here you have the possibility of defining an open array.
(*prog_char TEN[] = {0,1,2,3,4,5,6,7,8,9};*)

Reading of the value *TEN[5]*:

char res = PRG_RDB(&value[5])

With this instruction the value of TEN in position 5 is memorized into the variable res.

16bit data typ: prog_int
32bit data typ: prog_long
64bit data typ: prog_long_long

The applications of the 16 bit(2Byte), 32 bit(4Byte), 64 bit(8Byte) data types correspond with those of the 8 bit data type.
The only difference is that these types reserve more memory.

Further applications:

The filing of a character string in the program memory.

Version1:

- **The creation of a character string:**

*Char *LINE1 = PSTR(„The first line of my LCD display“);*

This definition files the sentence in the program memory and leads the pointer back, that points to the first sign of the character string.

- **The reading of a sign of the character string:**

Char lastchar = PRG_RDB (LINE1+42);

The variable (in the RAM) lastchar now contains the 42th letter of the character string *LINE1*.

Version2:

- **The creation of a character string:**

Char LINE2[] __attribute__ ((progmem)) = „this is an other version“;

With this definition a field is created in which the sentence *„this is an other version“* is filed.

- **The reading of a sign of the character string:**

Char firstchar = PRG_RDB (&[LINE2[0]);

The variable *firstchar* (in the SRAM) now contains the first letter of the character string LINE2.

Special operation of approach in <*progmem.h*>:

uint PRG_RDB(uint16_t addr);

This function contains as an input the address where you can find the value in the program memory and as a return the value itself.

PSTR(s);

PSTR files a string in the program memory and provides a starting address.

Variables in the eeprom (electrical erasable programmable read only memory):

In contrast to the simple definition of variables in SRAM for the usage of the eeprom which is integrated through I/O registers you need special access procedures.
This procedures can be found in the include file *<eeprom.h>.*

Procedures:

eeprom_wb (unsigned int addr, unsigned char val);

This procedure allows to write the value *val* to a defined address *addr*.

unsigned char eeprom_rb (unsigned int addr);

With the *eeprom_rb* you can read a value (one byte) out of the eeprom address *addr* and this is directly handed over in the form

value = eeprom_rb (addr);

unsigned int eeprom_rw (unsigned int addr);

With the eeprom_rw you can read a 16bit value (two byte) out of the eeprom address addr and this is directly handed over in the form

value = eeprom_rw (addr);

eeprom_read_block (void *buf, unsigned int addr, size_t n);

With this procedure *n* values are read out of the address *addr* and written into the SRAM. The starting address in SRAM is transmitted with the **buf*.

int eeprom_is_ready (void);

This function returns one if the eeprom is ready and zero if the eeprom is busy.

The use of the procedures:

- **First kind of use:**

It is possible to memorize a value directly on the eeprom. One only has to name an address and a value.You will soon lose back of things, if you draw up various values of this kind because you have to manage the addresses to the variables.

example:

```
/* This program stores the value $AA
on the eeprom at the address $0.
Then the value at address $0 is read
and written to PORTB

10/00
Leitner Harald*/

#include <io.h>
#include <eeprom.h>
int main(void)
{

uint8_t val1 = 0xAA, val2;

eeprom_wb(0x00, val1);        /* file val1 at eeprom */

val2 = eeprom_rb(0x00);       /* read eeprom at address $0 */

outp(0xff,DDRB);

outp(val2,PORTB);             /* show val2 on PORTB */
for (;;) {}
}
```

Here the value of the varaible val1 is memorized on the address 0 of the eeprom. Afterwards the value, you can find on address 0, is read and memorized in variable val2.
This value is shown by the leds.

- **Second kind of use:**

An assembler programmer is familiar with the first way, but it should be easier to handle in a higher language.
Therefore there exists a more useful variation. A variable whose address is located in the surroundings of the eeprom is defined. Because of that you don't need an explicit address, you can take the one of the variable.
The variable can be defined as follows:

example:

```
/* This program writes the value $AA in the
variable val1 that is defined on the eeprom.
Afterwards the variable val1 is read and
written to PORTB

10/00 Leitner Harald*/

#include <io.h>
#include <eeprom.h>

int main(void)
{

/* That's the way to define a eeprom variable */
static uint8_t val1 __attribute__((section(".eeprom")));

uint8_t val2;

outp(0xFF,DDRB);

eeprom_wb ((uint8_t)&val1, 0xAA);   /* writing the val1 */

val2  = eeprom_rb((uint8_t)&val1);  /* reading of val1 */

outp(val2,PORTB);

for (;;) {}
}
```

This example carries out the same function like the example of the first way of use, but the address is mentioned through variable val which is defined in the eeprom.

There is also the possibility to put an array in the eeprom and read again:

example:

```
/* This program creates an array on the eeprom and
writes different values into the array.
Afterwards the val[4] is read and written to PORTB

10/00 Leitner Harald*/

#include <io.h>
#include <eeprom.h>

int main(void)
{
/* That's the way to define an array on the eeprom */
static uint8_t val[5] __attribute__((section(".eeprom")));

uint8_t val2;

outp(0xFF,DDRB);

eprom_wb ((uint8_t)&val[0], 0xAA);     /* writing the val[0] */
eeprom_wb ((uint8_t)&val[1], 0xBB);   /* writing the val[1] */
eeprom_wb ((uint8_t)&val[2], 0xCC);   /* writing the val[2] */
eeprom_wb ((uint8_t)&val[3], 0xDD);   /* writing the val[3] */
eeprom_wb ((uint8_t)&val[4], 0x00);    /* writing the val[4] */

val2  = eeprom_rb((uint8_t)&val[4]);    /* reading of val[4] */

outp(val2,PORTB);

for (;;) {}
}
```

In this example an array field is defined with 5 values. The fields of the arrays get values one after the other and afterwards the whole string is read. In that way the indicator is sent back to the value array as a result.

How to copy an array from the eeprom to the SRAM?

Example:

```
/* This program creates an array eepromval
on the eeprom and writes different
values into the array.
Then the whole array is copied to the SRAM
array sramval
Afterwards the sramval[3] is read and
written to PORTB.

10/00 Leitner Harald*/

#include <io.h>
#include <eeprom.h>

int main(void)
{

/* That's the way to define an array on the eeprom */
static uint8_t eepromval[5] __attribute__((section(".eeprom")));

uint8_t ramval[5];

outp(0xFF,DDRB);

eeprom_wb ((uint8_t)&eepromval[0], 0x1);   /* writing the eepromval[0] */
eeprom_wb ((uint8_t)&eepromval[1], 0x2);   /* writing the eepromval[1] */
eeprom_wb ((uint8_t)&eepromval[2], 0x3);   /* writing the eepromval[2] */
eeprom_wb ((uint8_t)&eepromval[3], 0x4);   /* writing the eepromval[3] */
eeprom_wb ((uint8_t)&eepromval[4], 0x5);   /* writing the eepromval[4] */

/* copy eepromval to ramval - length is 5 */
eeprom_read_block(&ramval,(uint8_t)&eepromval,5);

outp(ramval[3],PORTB);
for (;;) {}
}
```

With this function *eeprom_read_block*, whole blocks that are memorized in the eeprom can be memorized into the SRAM.

For this you only need a starting address in the eeprom and the aim and the length of the blocks to copy.

Programming I/O registers and using Ports:

1. How to use I/O registers:

The peripherie that is used in AVR processor series is implemented through I/O registers. Because of that new values can not be fixed through simple allocations (impossible: *PORTB = 0xFF*).
In order to work with these I/O registers you need special approaching operations, that are defined in the include file <*iomacros.h*> through assembler instructions.
Instead of <*iomacros.h*> the include file <io.h> should be included.
These I/O registers are so called “special function registers” (SFR) and are pre-defined in the processor specific file (for the AT90S8535: <*io8535.h*>*).*
These registers are called through a fix address.

- ***BV(x):***

In the processor specific file (for example: <*io8535.h*> the register bit names are defined through their bit number (0-7) in the correct register (for instance: the constante *PINA6* is the 6th bit of *PORTA* and is of the value 6)
One needs function *BV (x)* in order to come from the bit´s location to the correct value with which the register has to be loaded.(so *BV (PINA6)* creates as a return value 64 2^6)

example:

char result = BV(PINA6);

result is now of the value 64

- ***void sbi (uint8_t port, uint8_t bit):***

With this instruction you can set particular bits in the given register. The other bits stay the same.

examples:

sbi (PORTB, 3);
sbi (PORTB, PINA3);

Both functions set the 3rd bit of register *PORTB*.

- ***void cbi (uint8_t port, uint8_t bit):***

With this instruction you can delete particular bits in the given register.

examples:

cbi (PORTB,3);
cbi (PORTB, PINB3);

Both functions delete the 3rd bit of register *PORTB*.

- ***uint8_t bit_is_set (uint8_t port, uint8_t bit);***

This function checks the bit at the place *bit* in the register *port*.
In case the bit is set this function has as a result 1, otherwise 0.
This function can be used as a check for loops or *if* statements.

example:

uint8_t result = bit_is_set (PORTB, PINB3);

result = 1 if the 3rd bit of PORTB is set
result = 0 ,otherwise.

- ***uint8_t bit_is_clear (uint8_t port, uint8_t bit);***

This function checks the bit at the place *bit* in the register *port*.
In case the bit is not set this function has as a result 1, otherwise 0.
This function can be used as a check for loops or *if* statements.

example:

uint8_t result = bit_is_clear (PORTB, PINB3);

result = 1 if the 3rd bit of PORTB is not set
result = 0 ,otherwise.

- ***uint8_t inp (uint8_t port):***

With this function you can read and return a 8 bit value out of the mentioned register.

example:

uint8_t res = inp (SREG);

Read the statue register *SREG* and put the value in the variable *res*.

- ***uint16_t __inw (uint8_t port):***

With this function you can read a 16 bit value out of the mentioned 16 bit register and bring it back. These registers are *ADC, ICR1, OCR1A, OCR1B, TCNT1* (for processor AT90S8535) and must be read in a special range in order to obtain a correct 16 bit value.
With this function it is possible that an interrupt is carried out during the reading process.

example:

uint16_t res = __inw (TCNT1);

Read register *TCNT1* and put the value into the variable *res*.

- ***uint16_t __inw_atomic (uint8_t port):***

These functions work like *__inw* but an interrupt is not possible.
In this function *__inw_atomic* an assembler instruction (*"cli"*) is integrated that deletes the I-bit (global interrupt enable) of the status register.

- ***outp (uint8_t val, uint8_t port):***

With this function you can write the 8 bit value *val* into register *port.*

example:

outp(0xFF, PORTB);

This instruction writes the value $FF into register *PORTB*.

- ***__outw (uint16_t val, uint8_t port):***

With this function a 16 bit value can be written into the register *port*. This is for initialising of the following registers: *ADC, ICR1, OCR1A, OCR1B, TCNT1* (for processor AT90S8535)
With this function it is possible that an interrupt is carried out during the writing process.

example:

__outw (0xAAAA, OCR1A);

The 16 bit register OCR1A (output compare register A of timer 1) is initialised with value $AAAA.

- ***__outw_atomic (uint16_t val, uint8_t port):***

This function works like *__outw* but an interrupt is not possible.
In this function __out*w_atomic* an assembler instruction (*"cli"*) is integrated that deletes the I-bit (global interrupt enable) in the status register.

2. How to use Ports:

All AVR ports have true "read-modify-write" functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the *sbi (uint8_t port, uint8_t bit)* and the *cbi (uint8_t port, uint8_t bit)*instructions.
Each ports consist of three registers.

registers:

- ***DDRX:***

This is the **D**ata **D**irection **R**egister of port **X** (instead of X use the right port character – for example *DDRA* is the register for Port A).
If you set this register to $FF the whole Port X is defined as output and if the register loaded with $00 the whole Port X is defined as input.
You can set or clear only some pins with the instructions *sbi (uint8_t port, uint8_t bit)* and *cbi (uint8_t port, uint8_t bit).*

example:

outp (0xF0, DDRB);

This instruction sets pin 0-3 of *PORTB* as input and pin 4-7 as output.

- ***PORTX:***

This is the data register of *PORTX* (X stands for A, B, C or D).
If you want to put data to the port or read data from the port you have to use this register.

example:

outp (0xAA, PORTB);

This instruction loads the data register of *PORTB* with the value $AA. In this case *PORTB* is used as output, so you have to initialize *PORTB* as output ($FF -> *DDRB*).

char res = inp (PORTB);

This instruction loads the value of the data register of *PORTB* into the variable *res*. In this case *PORTB* is used as input, so you have to initialize PORTB as input ($00 -> *DDRB*).

- ***PINX:***

PINX (X stands for A, B, C or D) is the input pin address of PORTX and is no register. When reading PORTX the PORTX data latch is read, and when reading PINX, the logical values present on the pins are read. The PORTX input pins are of the state “read only”, while the data register (PORTX) and the data direction register (DDRX) are of the state “read/write”.

example:

outp(0x00,DDRA);
res = inp(PINA);

With this instructions you can read the physical value of *PORTA.*

- **Alternate functions:**

Most of the ports have alternate functions.
Here several pins are used as an entry for the peripherie that is situated in the processor.
These entries of the processor are allocated differently in the Atmel AVR series.

example:

AT90S8535:
pin 0 of PORTB -> external counter input for timer/counter T0

Interruptprogramming with the GNU-C compiler

Interrupts are used when you have to react fast to special incidents. These incidents are actuated either through the internal pheripherie or through external signals. During the running program an interrupt routine is called in order to analyse these incidents.
There the interrupt is processed and jumped back to the place where the program has been left before.
That an interrupt is licensed different bits have to be set in the suitable registers of the peripherie and the status register.
For the use of the interrupt functions include the files <*interrupt.h*> and <*signal.h*>.

<*signal.h*> includes possible interrupt names that have to be mentioned when they are used.

Signal names:

External interrupt0 function name:
SIG_INTERRUPT0

External interrupt1 function name:
SIG_INTERRUPT1

External interrupt2 function name (ATmega):
SIG_INTERRUPT2

External interrupt3 function name (ATmega[16]03)
SIG_INTERRUPT3

External interrupt4 function name (ATmega[16]03):
SIG_INTERRUPT4

External interrupt5 function name (ATmega[16]03):
SIG_INTERRUPT5

External interrupt6 function name (ATmega[16]03):
SIG_INTERRUPT6

External interrupt7 function name (ATmega[16]03):
SIG_INTERRUPT7

Output compare2 interrupt function name:
SIG_OUTPUT_COMPARE2

Overflow2 interrupt function name:
SIG_OVERFLOW2

Input capture1 interrupt function name:
SIG_INPUT_CAPTURE1

Output compare1(A) interrupt function name:
SIG_OUTPUT_COMPARE1A

Output compare1B interrupt function name:
SIG_OUTPUT_COMPARE1B

Overflow1 interrupt function name:
SIG_OVERFLOW1

Output compare0 interrupt function name:
SIG_OUTPUT_COMPARE0

Overflow0 interrupt function name:
SIG_OVERFLOW0

SPI interrupt function name:
SIG_SPI

UART(0) receive complete interrupt function name:
SIG_UART_RECV

UART1 Receive complete interrupt function name (ATmega161):
SIG_UART1_RECV

UART(0) Data register empty interrupt function name:
SIG_UART_DATA

UART1 Data register empty interrupt function name (ATmega161):
SIG_UART1_DATA

UART(0) Transmit complete interrupt function name:
SIG_UART_TRANS

UART1 Transmit complete interrupt function name (ATmega161):
SIG_UART1_TRANS

ADC Conversion complete:
SIG_ADC

Eeprom ready:
SIG_EEPROM

Analog comparator interrupt function name:
SIG_COMPARATOR

The structure of the routine:

For the definition of such an interrupt routine you need to write the keyword „*SIGNAL*“ or „*INTERRUPT*“.

SIGNAL (SIG_NAME)
{
Here the instructions of the interrupt routine are processed.
}

The interrupt routine with the keyword *SIGNAL* is executed with disabled interrupts.

Or:
INTERRUPT (SIG_NAME)
{
Here the instructions of the interrupt routine are processed.
}

The interrupt routine with the keyword *INTERRUPT* is executed with enabled interrupts.

Functionens of <*interrupt.h*>:

Sei():

This function sets the *I*-bit in the status register and therefore enables interrupts.
The individual interrupt enable control is then performed in separate control registers.

Cli():

Deletes the I-bit in the status register and therefore avoids possible interrutps.

enable_external_int (unsiged char ints):

This function sets suitable bits in the *GIMSK* register (Mega series: *EIMSK*) in order to enable external interrupts.

example:

```
/*
        Turn on leds with switch on PD3 (Int1)
        Turn off leds with switch on PD2 (Int0)

        10/00 Leitner Harald
*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

SIGNAL (SIG_INTERRUPT0) /* PD2 */
{

outp(0xFF, PORTB);    /* turn off leds */

}

SIGNAL (SIG_INTERRUPT1) /* PD3 */
{

outp(0x00, PORTB);    /* turn on leds */

}

int main( void )
{

/* define PortB as Output (Leds) and PortD as Input (Switches) */
outp(0xFF, DDRB);
outp(0x00, DDRD);

/* enable interrupt Int0 and Int1 */
outp((1<<INT0)|(1<<INT1), GIMSK);

/* falling edge on Int0 or Int1 generates an interrupt */
outp((1<<ISC01)|(1<<ISC11), MCUCR);

sei();
for (;;){}

}
```

This program switches through switch on *PORTD.2* and *PORTD.3* the leds on and off. The stiches are connected through interrupt entrances *INT0 (PORTD.2)* and *INT1 (PORTD.3)*.
Therefore an interrupt control is possible.
First the bits *INT0 (bit6)* and *INT1 (bit7)* have to be set in the *GIMSK* register. Afterwards in the *MCUCR* register is adjusted to witch signal an interrupt has to be released. For each interrupt 2 bits have to be set.

ISC00 (bit0) , ISC01 (bit1)	...	Interrupt 0
ISC10 (bit2) , ISC11 (bit3)	...	Interrupt 1

ISCX1	*ISCX0*	description
0	0	the low level of *INTX* generates an interrupt
0	1	reserved
1	0	the falling edge of *INTX* generates an interrupt
1	1	the rising edge of *INTX* generates an interrupt

How to use Timer / Counter 0 ?

General information:

Timer0 is a 8 bit timer/counter which can count from *0* to *$FF*. In the timer mode this peripherie uses an internal clock signal and in the counter mode an external signal on *PORTB.0.*
I take both mode of operation into consideration.
Besides the timer can be operated either in the polling mode or in the interrupt mode.

Used registers:

Timer registers:

TCCR0	(Timer/Counter 0 Control Register)
TCNT0	(Timer/Counter 0 Value)

Interrupt registers:

TIFR	(Timer Interrupt Flag Register)
TIMSK	(Timer Interrupt Mask Register)
GIMSK	(General Interrupt Mask Register)

Timer mode:

In this mode of operation the timer is provided by an internal signal. Whereas after each clock cycle the value of the *TCNT0* register is increased by one.This clock signal is produced out of *x* times the amount of the oscillator signal. The factor *x* can have the following values:
1, 8, 64, 256, 1024
(for example: 1024 - the timer is increased after 1024 cycles of the oscillator signal)
This prescaling is controlled by writing one of the following values into the register *TCCR0*:

initial value	used frequency
1	ck
2	ck/8
3	ck/64
4	ck/256
5	ck/1024

Polling mode:

Example:

```
/*

        Testprogramm for Timer/Counter 0 in the Polling Mode
        If the Timer has an overflow the overlow bit in the TIFR register
        will be set and the led variable increased.
        The led variable will be written to the PORTB.

        Leitner Harald
        07/00

*/

#include <io.h>

uint8_t led;
uint8_t state;

int main( void )
{

outp(0xFF, DDRB);           /* use all pins on PORTB for output */

outp(0, TCNT0);             /* start value of T/C0 */
outp(5, TCCR0);             /* prescale ck/1024 */

led = 0;

for (;;)

        {

        do                  /* this while-loop checks the overflow bit in the
                               TIFR register */

                state = inp(TIFR) & 0x01;
```

```
        while (state != 0x01);

        outp(~led,PORTB);

        led++;

        if (led==255)
                led=0;

        outp((1<<TOV0),TIFR);       /* if a 1 is written to the TOV0 bit the
                                       TOV0 bit will be cleared */
        }
}
```

In that way the register *TCCR0* is loaded with *5 (ck/1024)*
and the starting value of the timer in the register *TCNT0* is laid down with *0*.

After each 1024th cycle of the oscillator the value of *TCNT0* is increased by one. The *for(;;){}* defines an endless loop.
In this loop a *do-while* loop is inserted, that constantly checks, if the bit at the place *0* of the *TIFR* register is set or not.
This bit has the name *TOV0* (timer overflow 0) and is set when the 8 bit register *TCNT0* is of the value *$FF* and tries to increase it -> overflow.
In this case the *do-while* loop is left and the bontent of the variable led is written on *PORTB*.
Afterwards the variable *led* is increased by one and checks if *led* is of the value *$FF*. In this case led is fixed to *0*. Otherwise you have to write a one into register *TIFR*, which has as a consequence that the *TOV0* is deleted and the timer starts counting from the beginning.

Interrupt mode:

This mode of operation is used more often than the polling mode. In this case the *TOV0* bit isn't constantly proved if it was set.
Because in case of an overflow the controller jumps from the actual position to the suitable interrupt vector address.
The interrupt is called from this vector address.
After this execussion the program goes on the place, where it was interrupted.

Example:

```
/*

        Test program for Timer/Counter 0 in the Interrupt Mode
        Every time the Timer starts an interrupt routine the led variable
        is written on the PORTB and increased one time.

        Leitner Harald
        07/00

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t led;

SIGNAL (SIG_OVERFLOW0)
{

outp(~led, PORTB);              /* write value of led on PORTB */

led++;

if (led==255)
        led = 0;

outp(0,TCNT0);                  /* reload timer with initial value */
```

```
}

int main( void )
{

outp(0xFF, DDRB);           /* use all pins on PORTB for output */

outp((1<<TOIE0), TIMSK);    /* enables the T/C0 overflow interrupt in
                               the T/C interrupt mask register for */
outp(0, TCNT0);             /* start value of T/C0 */

outp(5, TCCR0);             /* prescale ck/1024 */

led = 0;

sei();                      /* set global interrupt enable */

for (;;){}
}
```

The interrupt routine is introduced trough the keyword *SIGNAL*.
As soon as an overflow occurs, this interrupt routine is carried out.
In the main program you have to fix the bits that enable the interrupt.
In the register *TIMSK* you have to set the bit *TOIE0* and through the command *sei()* the *i*-bit (global interrupt enable) is enabled in the status register (*SREG*).

Counter mode:

In this mode of operation the statue changes on the pin *T0* are counted. Instead of the manual operation of entrance *T0*, a supply trough a frequency generator is equally possible.
Following program counts the status changes on pin *T0* and increase the value of the counter register *TCNT0* by one.
One has to pay attention that the pin *T0* is situated on *PORTB*. Therefore pin 0 of *PORTB* has to be defined as an input and all others as an output.
The next step is the definition of the right mode of operation.
The value *$6* has to be written into the register *TCCR0*.
Now the timer/counter is configurated as a counter of falling edges to pin *T0*.

Polling mode:

Example:

```
/*

    Test program for Counter 0 in the Polling Mode
    If the Counter has an overflow the overlow bit in the TIFR register
    will be set and the led variable increased.
    The led variable will be written to the PORTB.

    Leitner Harald
    07/00

*/

#include <io.h>

uint8_t led;
uint8_t state;
```

```
int main( void )
{

outp(0xFE, DDRB);           /* use pin 1-7 of PORTB as output
                                   and pin 0 (T0) as input  */

outp(0xFE, TCNT0);          /* start value of counter */

outp(6, TCCR0);             /* init the T/C as counter triggered by
                               falling edge on T0 */
led = 2;

for (;;)
        {

        do                  /* this while-loop checks the overflow bit
                               in the TIFR register */

                state = inp(TIFR) & 0x01;

        while (state != 0x01);

        outp(0xFE, TCNT0);  /* start value of counter */

        outp(~led,PORTB);

        led++;

        if (led==255)

                led=0;

        outp((1<<TOV0),TIFR);       /* if a 1 is written to the TOV0 bit
                                       the TOV0 bit will be cleared */
        }
}
```

Interrupt mode:

```
/*

        Test program for Counter 0 in the Interrupt Mode
        Every time a falling edge is set on the T0 input
        the counter is increased for one time.

        Leitner Harald
        07/00

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t led;

SIGNAL (SIG_OVERFLOW0)
{

outp(~led, PORTB);              /* write value of led on PORTB */

led++;

if (led==255)

        led = 2;

outp(0xFE,TCNT0);               /* reload counter with initial value */

}

int main( void )
{

outp(0xFE, DDRB);               /* use all pins on PORTB for output */

outp((1<<TOIE0), TIMSK);    /* enables the T/C0 overflow interrupt in
```

```
                                the T/C interrupt mask register for */

outp(0xFE, TCNT0);              /* start value of counter */

outp(6, TCCR0);                 /* init the T/C as a counter falling edge
                                on Pin T0 */
led = 0;

sei();                          /* set global interrupt enable */

for (;;){}
}
```

How to use Timer / Counter 1 ?

General information

In contrast to timer 0 or timer 2, timer 1 is a 16 bit timer/counter. Because of that you can use it for longer counter procedures. The counting extent is between *$0000* and *$FFFF*. This area is being realised through two registers. Otherwise Timer 1 possesses compare/capture and a PWM.

Used registers:

Timer registers:

TCCR1A	(Timer/Counter Control Register A)
TCCR1B	(Timer/Counter Control Register B)
TCCR1L	(Timer/Counter Value Low Byte)
TCCR1H	(Timer/Counter Value High Byte)
OCR1AL	(Output Compare Register A Low Byte)
OCR1AH	(Output Compare Register A High Byte)
OCR1BL	(Output Compare Register B Low Byte)
OCR1BH	(Output Compare Register B High Byte)
ICR1L	(Input Capture Register Low Byte)
ICR1H	(Input Capture Register High Byte)

Interrupt registers:

TIFR	(Timer Interrupt Flag Register)
TIMSK	(Timer Interrupt Mask Register)
GIMSK	(General Interrupt Mask Register)

Timer mode:

In this mode of operation the timer is supplied by an internal signal. After each takt cycle the meter reading is increased by 1. This signal is produced by a *n* times the amount of the oscillator signal. The factor *x* can have the following result:
1,8,64,256,1024
(for instance: 1024- only after 1024 cycles of the oscillators the timer is raised- the frequency is only fosc/1024)
This results can be set with register *TCCR1B*.
The timer is adjusted through writing the following results into the register

initial value	used frequency
1	ck
2	ck/8
3	ck/64
4	ck/256
5	ck/1024

Polling mode:

Example:

```
/*

        Test program for Timer/Counter 1 in the Polling Mode
        If the Timer has an overflow the overflow bit of the TIFR register
        is set and the led variable increased.
        The led variable is then written to PORTB.

        Leitner Harald
        07/00

*/

#include <io.h>

uint8_t led;
uint8_t state;

int main( void )
{

outp(0xFF, DDRB);           /* use all pins on PORTB for output */

outp(0x00, TCNT1L);         /* start value of T/C1 - low byte */
outp(0x00, TCNT1H);         /* start value of T/C1 – highbyte*/
outp(0, TCCR1A);            /* T/C1 in timer mode */
outp(1, TCCR1B);            /* prescale ck */

led = 0;

for (;;)
        {
        do                  /* this while-loop checks the overflow bit in
                            the TIFR register */
                state = inp(TIFR) & 0x04;

        while (state != 0x04);

        outp(~led,PORTB);

        led++;
```

```
        if (led==255)
                led=0;

        outp(0x00, TCNT1L);         /* start value of T/C1 - low byte */
        outp(0x00, TCNT1H);         /* start value of T/C1 - high byte*/

        outp((1<<TOV1),TIFR);       /* if a 1 is written to the TOV1
                                       bit the TOV1 bit will be cleared */
        }
}
```

The *TCCR1A* register is established with 0 the *TCCR1B* register with 1 (CK-one prescale) and the starting value of the registers *TCNT1L* and *TCNT1H* with 0.
After each cycle of the quartz oscillators the meter reading of the *TCNT1L* register is increased by 1. After reaching the value *$FF* in register *TCNT1L* and renewed increasing the register, *TCNT1H* is increased by 1 and *TCNT1L* is placed to 0.
There is a *do-while* loop addeded, which constantly controlls if the bit is in position 4 of the *TIFR* register or not. This register is called *TOV1* (time overflow1) and is established if both 8 bit registers (*TCNT1L, TCNT1H*) are of the same value *$FF* and if it is tried to raise the value-overflow.
In this case the *do-while* loop is left and the content of the variable led is written on *PORTB*.
Afterwards the variable *led* is increased by one and it must be prooved if led is of the value 255($FF). If this is the case *led* is put to *0*.
Otherwise one is written in position 4 of the register *TIFR*, which has the consequence that *TOV1* bit is deleted and the timer has to begin to count once again.

Interrupt mode:

This mode of operation is used more frequently than polling. The *TOV1* bit is not always controlled if it has been put. Because if there is an overflow the appropriate vector address is mentioned**.** The interrupt routine is called up of this vector address.
After the work of this routine the programm goes at the point where it was interrupted.

Example:

```
/*

        Test program for Timer/Counter 1 in the Interrupt Mode
        Every time the Timer starts an interrupt routine the led variable
        is written on the PORTB and increased one time.

        Leitner Harald
        07/00
*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t led;

SIGNAL (SIG_OVERFLOW1)
{

outp(~led, PORTB);              /* write value of led on PORTB */

led++;
if (led==255)
        led = 0;

outp(0,TCNT1L);                 /* reload timer with initial value */
outp(0XFF, TCNT1H);

}
```

```
int main( void )
{

outp(0xFF, DDRB);              /* use all pins on port B for output */

outp((1<<TOIE1), TIMSK);       /* enables the T/C1 overflow interrupt in the
                                  T/C interrupt mask register f

outp(0xFF, TCNT1H);                  /* start value of T/C1 */

outp(0, TCNT1L);

outp(0, TCCR1A);               /* no compare/capture/pwm mode */

outp(5, TCCR1B);               /* prescale ck/1024 */

led = 0;

sei();                         /* set global interrupt enable */

for (;;){}

}
```

The interrupt routine is introduced through the keyword *SIGNAL*. As soon as an overflow occurs the routine is carried out. In the main program necessary interrupts must be enabled.
In register *TIMSK* the bit *TOIE1* has to be set and through the instruction the *I*-bit in the status register is enabled.
In all other cases the timer is initialised like in the polling mode.

Counter mode:

In this mode of operation the changes of the statue are counted on the external pin *T1*. In case of an overflow an interrupt routine is called up. Instead of the manual handling of entry *T1*, a supply through a frequency generator is equally possible.

The program is in the most parts equal to that of **timer0**. One should pay attention that pin *T1* is situated at *PORTB*. Because of that one has to define Pin1 of *PORTB* as an entry and all the others as an outlet. The next step is the determination of the suitable mode of operation. For that the value *$6* is written into register *TCCR1B*. Now the timer/counter is configurated to pin T1 as a counter of falling edges.

Compare mode:

The Timer/Counter 1 supports two output compare functions using the registers *OCR1A* (low and high byte) and *OCR1B* (low and high byte) as the data sources to be compared to the content of the Timer/Counter register *TCNT1* (low and high byte). If there is a compare match it is possible to clear the content of the Timer/Counter register (only if compare with *OCR1A*) or take effects on the output pins.
This pins are called *OC1A* (PORTD.5) and *OC1B* (PORTD.4).

The different functions are controlled by the register *TCCR1A* as follows:
Bit 0-3: not used
Bit 4: *COM1B0*
Bit 5: *COM1B1*
Bit 6: *COM1A0*
Bit 7: *COM1A1*

Mode Select:

COM1X1	*COM1X0*	Description
0	0	T/C 1 disconnected from pin *OC1X*
0	1	Toggle the value of *OC1X*
1	0	Clear *OC1X*
1	1	Set *OC1X*

The *CS10*, *CS11* and the *CS12* bit (bit0-2) of register *TCCR1B* defines the prescalling source of Timer/Counter 1 as follows:

CS12	*CS11*	*CS10*	Description
0	0	0	T/C 1 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	clocked by the pin T1, falling edge
1	1	1	clocked by the pin T1, rising edge

If you want to clear the content of Timer/Counter 1 on a compareA match, it is necessary to set bit3 in the register *TCCR1B*.

In a compare match the suitable bit (*OCIE1A* -> bit4, *OCIE1B* -> bit3) is set in the *TIMSK* register or the interrupt routine (*SIG_OUTPUT_COMPARE1A, SIG_OUTPUT_COMPARE1B*) is carried out.

example:

```
/*

        Flashes LED on STK200 Board with Compare - Mode of Timer 1
        Pulse width is regulated by switch PD2 and PD3 (Int0 Int1)

        07/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t delay;

SIGNAL (SIG_OUTPUT_COMPARE1A)/* Compare interrupt routine */
{

if (delay == 0)
        outp(0XFF, PORTB);
else
        outp(0XFE, PORTB);

}

SIGNAL (SIG_OVERFLOW1)/* T/C1 overflow interrupt routine */
{

if (delay == 0)
        outp(0XFF, PORTB);
else
        outp(0XFF, PORTB);

outp(0XFF, PORTB);
outp(delay,TCNT1H);
outp(0,TCNT1L);

}
```

```
SIGNAL (SIG_INTERRUPT0) /* PD2 */
{

if (delay < 15)
        delay = 0;
else delay = delay - 15;

}

SIGNAL (SIG_INTERRUPT1) /* PD3 */
{

if (delay > 235)
        delay = 250;
else delay = delay + 15;

}

int main( void )
{

outp(0xFF, DDRB);           /* define PORTB as Output (Leds) and
                               PORTD as Input (Switches) */
outp(0x00, DDRD);

delay = 120;                /* default of timer1 high byte */

/* Switches PB3,PB4 for Interrupt 0 and 1 */
outp((1<<TOIE1)|(1<<OCIE1A), TIMSK);

outp((1<<INT0)|(1<<INT1), GIMSK);

outp((1<<ISC01)|(1<<ISC10)|(1<<ISC11), MCUCR);

outp(delay, TCNT1H);            /* Init  T1 */

outp(0, TCNT1L);

outp(0XFF, OCR1AH);             /* Compare value of T1 */
outp(0X10, OCR1AL);

outp(0, TCCR1A);                /* Timer mode with no output */
outp(1, TCCR1B);                        /* counting with ck */
```

sei();

for (;;){}

}

This program shows a possible use of the compare function. The timer is initialized in the main program. To controll the delay of the compare mode there are two switches *PD2* and *PD3* on the STK 200 board.
A variable delay is decreased by 15 when using *PD2* and when using *PD3* increased by 15.
This value delay is loaded into the high-byte of the timer/counter register. Out of this value the timer tries to reach the overflow.
Before it can reach an overflow a compare match occurs and the suitable interrupt routine is called up.
Because of this routine a led on the board flashes.
Afterwards the timer runs to an overflow and carries out the overflow interrupt routine. The led is switched off again and the timer new is initialised with the value delay.

Capture mode:

This mode of operation enables to memorize the actual value of Timer/Counter 1 through an external signal.
When the rising or falling edge of the signal at the input capture pin *ICP* (*PORTD.6*) is detected, the current value of Timer/Counter 1 is transfered to the 16bit input capture register *ICR1* (*ICR1L, ICR1H*).
This sets the input capture flag *ICF1* and is possible to call up a interrupt routine (*SIG_INPUT_CAPTURE*) if the bit *TICIE1* (bit5 of *TIMSK*) is set.
The most important thing is that you have to read the low byte (*ICR1L*), for a full 16bit register read, first.

example:

```
/*

    Flashes LED on STK200 Board with Capture - Mode of Timer 1
    With each capture signal on PORTD.6 the actual value of timer 1
    (high byte) is written at PORTB.

     07/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

SIGNAL (SIG_OVERFLOW1)
{

outp(0,TCNT1H);                    /* reset timer after overflow */
outp(0,TCNT1L);

}

SIGNAL (SIG_INPUT_CAPTURE1)
{

outp(~inp(ICR1L),PORTB);
                                   /* reading timer value of capture register */

outp(~inp(ICR1H),PORTB);           /* and write to PORTB */
}
```

```
int main( void )
{

outp(0xFF, DDRB);     /* define PORTB as Output (Leds) and */

outp(0x00, DDRD);                /* PORTD as Input (Switches) */

outp(0xFF, PORTB);

/* Enable interrupt for capture and overflow */
outp((1<<TICIE1)|(1<<TOIE1), TIMSK);

outp(0, TCNT1H);                 /* Init  T1 */
outp(0, TCNT1L);

outp(0, TCCR1A);                  /* Timer mode with no output */
outp(5, TCCR1B);                 /* counting with ck/1024 */

sei();

for (;;){}

}
```

This program reads the actual value of Timer/Counter 1 after a signal has occured on the pin *ICP*.
Then the high-byte of this value is written to the Leds on *PORTB*.

PWM mode:

When the PWM (PulseWidthModulation) mode is selected the Timer/Counter 1 can be used as an 8,9 or 10bit, free running PWM. Timer/Counter 1 acts as an up/down counter that is counting up from *$0000* to the selected top (8bit -> *$00FF*, 9bit -> *$01FF*, 10bit -> *$03FF*), where it turns and counts down to *$0000* and repeats this cycle endlessly. When the counter value matches the content of the compare register (*OCR1A, OCR1B*) it has an effect on the output pins *OCA1* and *OCB1* as follows:

COM1X1	*COM1X0*	Effect on *OCX1*
0	0	no
0	1	no
1	0	cleared on compare match, up-counting, set on compare match, down-counting
1	1	cleared on compare match, down-counting, set on compare match, up-counting

These bits are set in the register *TCCR1A.*
(*COM1A1* -> bit7, *COM1A0* -> bit6, *COM1B1* -> bit5, *COM1B0* -> bit4)
The right PWM mode is selected bits *PWM10* (bit0 of *TCCR1A*) and *PWM11* (bit1 of *TCCR1A*) as follows:

PWM11	*PWM10*	Description
0	0	PWM mode disabled
0	1	8bit PWM
1	0	9bit PWM
1	1	10bit PWM

example:

```
/*

    Testprogramm for Timer/Counter 1 PWM Mode 10bit
    The T/C is used as a free running 10bit-PWM.
    The T/C is counting from $00 up to $3FF and after
    reaching $3FF the T/C is counting down to $00.
    This cycle repeats endless.
    When the counter value matches the content of the output
    compare register during counting up, PD5(OC1A) pin is cleared
    and if matches while counting down the PD5(OC1A) is set.
```

```
        When the counter value matches the content of the output
        compare register during counting up, PD4(OC1B) pin is set
        and if matches while counting down the PD4(OC1B) is cleared.

        Leitner Harald
        07/00

*/

#include <io.h>

int main( void )
{

outp(0xFF, DDRD);           /* use all pins on port D for output */
outp(0xB3, TCCR1A);         /* init the counter */
outp(0x5, TCCR1B);          /* init the counter */
outp(0x00, TCNT1L);         /* value of T/C1L */
outp(0x00, TCNT1H);         /* value of T/C1L */
outp(0xFF, OCR1AL); /* value of Compare register A Low-Byte */
outp(0x00, OCR1AH); /* value of Compare register A High-Byte */
outp(0xFF, OCR1BL); /* value of Compare register B Low-Byte */
outp(0x00, OCR1BH); /* value of Compare register B High-Byte */
for (;;){}
}
```

This program creates a free running PWM signal on the two outputs *OCA1* and *OCB1*. The two compare registers *OCR1A* and *OCR1B* define the pulswidth.

Selected Mode:
TCCR1A:
- 10bit PWM
- *Com1A1* -> 1, *COM1A0* -> 0
  - Cleared on compare match, up-counting, set on compare match, down-counting
- *Com1B1* -> 1, *COM1B0* -> 1
  - Cleared on compare match, down-counting, set on compare match, up-counting

TCCR1B:
- prescale: ck/10

How to use Timer 2 ?

General information:

Timer2 is a 8 bit timer that counts from *0* to *$FF*. In the timer mode this peripherie uses an internal clock signal. Besides the timer can be operated either in the polling mode or in the interrupt mode.

Used registers:

Timer registers:

TCCR2	(Timer0 Control Register)
TCNT2	(Timer0 Value)
OCR2	(Output Compare Register of Timer2)
ASSR	(Asynchronous Status Register)

Interrupt registers:

TIFR	(Timer Interrupt Flag Register)
TIMSK	(Timer Interrupt Mask Register)
GIMSK	(General Interrupt Mask Register)

Timer mode:

In this mode of operation the timer is provided by an internal signal.
The value of the *TCNT2* register is increased by one after each clock cycle. This clock signal is produced out of *x* times the amount of the oscillator signal. The factor *x* can have the following values:
1, 8, 32, 64, 128, 256, 1024
(for example: 1024 - the timer is increased after 1024 cycles of the oscillator signal)
This prescaling is controlled by writing one of the following values into the register *TCCR2*:

initial value	used frequency
1	ck
2	ck/8
3	ck/32
4	ck/64
5	ck/128
6	ck/256
7	ck/1024

Timer2 is no Timer/Counter but only a Timer for internal signals.

Polling mode:

Example:

```
/*

        Test program for Timer/Counter 2 in the Polling Mode
        If the Timer has an overflow the overlow bit in the TIFR register
        is set and the led variable increased.
        Then the variable led is written to PORTB.

        Leitner Harald
        07/00

*/

#include <io.h>

uint8_t led;
uint8_t state;

int main( void )
{

outp(0xFF, DDRB);           /* use all pins on PORTB for output */

outp(0, TCNT2);             /* start value of T/C2 */

outp(7, TCCR2);             /* prescale ck/1024 */

led = 0;

for (;;)
        {

        do                  /* this while-loop checks the overflow bit in the
                               TIFR register */
                state = inp(TIFR) & 0x40;

        while (state != 0x40);
```

```
        outp(~led,PORTB);

        led++;

        if (led==255)
                led=0;

        outp((1<<TOV2),TIFR);       /* if a 1 is written to the TOV2 bit
                                           the TOV2 bit will be cleared */
        }
}
```

In that way the register *TCCR2* is loaded with *7* (*ck/1024*)
and the starting value of the timer in register *TCNT2* is laied down with *0*.

After each 1024th cycle of the oscillator the value of TCNT2 is increased by one. The *for(;;){}* defines an endless loop.
In this loop a *do-while* loop is inserted, that constantly checks, if the bit on the place 6 of the *TIFR* register is set or not.
This bit has the name *TOV2* (timer overflow 2) and is set when the 8 bit register *TCNT2* is of the value *$FF* and tries to increase it -> overflow.
In this case the *do-while* loop is left and the bontent of the variable *led* is written on *PORTB*.
Afterwards the variable *led* is increased by one and checks if *led* is of the value *$FF*. In this case *led* is fixed to *0*. Otherwise you have to write a one into register *TIFR on position 6* , what has as a consequence that the *TOV2* is deleted and the timer starts counting from the beginning.

Interrupt mode:

This mode of operation is used more often than the polling mode. In this case the *TOV2* bit isn't constantly proved if it was set.
Because in case of an overflow the controller jumps from the actual position to the suitable interrupt vector address.
The interrupt is called from this vector address.
After this execussion the program goes on the place, where it was interrupted.

Example:

```
/*

        Test program for Timer/Counter 2 in the Interrupt mode
        Every time the Timer starts an interrupt routine the led variable
        is written on the PORTB and increased one time.

        Leitner Harald
        07/00

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t led;

SIGNAL (SIG_OVERFLOW2)
{

outp(~led, PORTB);              /* write value of led on PORTB */

led++;

if (led==255)
        led = 0;

outp(0,TCNT2);                  /* reload timer with initial value */
```

```
}

int main( void )
{

outp(0xFF, DDRB);           /* use all pins on PORTB for output */

outp((1<<TOIE2), TIMSK);    /* enables the T/C2 overflow interrupt in
                               the T/C interrupt mask register for */

outp(0, TCNT2);             /* start value of T/C2 */

outp(7, TCCR2);             /* prescale ck/1024 */

led = 0;

sei();                      /* set global interrupt enable */

for (;;){}
}
```

The interrupt routine is introduced trough the keyword *SIGNAL*.
As soon as an overflow occurs, this interrupt routine is carried out.
In the main program you have to fix the bits that enable the interrupt.
In the register *TIMSK* you have to set the bit *TOIE2* and through the command *sei()* the *i*-bit (global interrupt enable) is enabled in the status register (*SREG*).

Compare mode:

The Timer2 supports one output compare function using the register *OCR2* as the data source to be compared to the content of the Timer2 data register *TCNT2.* If there is a compare match it is possible to clear the content of Timer2 data register *TCNT2* or to take effect on the output compare pin *OC2 (PORTD.7).*

The different functions are controlled by the register *TCCR2* in the following way:

Bit 0-2: used for prescale *(CS20 -> bit0, CS21 -> bit1, CS22 -> bit2)*
Bit 3: *CTC2* (Clear Timer on Compare Match)
Bit 4: *COM20*
Bit 5: *COM21*
Bit 6: *PWM2*
Bit 7: not used

Mode select:

Com21	*Com20*	Description
0	0	Timer2 disconnected from pin *OC2*
0	1	Toggle the value of *OC2*
1	0	Clear *OC2*
1	1	Set *OC2*

Bits 0-2 of register *TCCR2* defines the prescaling source of timer2 in the following way:

CS22	*CS21*	*CS22*	Description
0	0	0	Timer2 stopped
0	0	1	*ck*
0	1	0	*ck/8*
0	1	1	*ck /32*
1	0	0	*ck /64*
1	0	1	*ck /128*
1	1	0	*ck /256*
1	1	1	*ck /1024*

If you want to clear the content of timer2 on a compare match it is necessary to set the *CTC2* bit (bit 3) of the *TCCR2* register.

On a compare match the bit *OCIE2* (bit 7) in the *TIMSK* register is set or a interrupt routine is called up *(SIG_OUTPUT_COMPARE2).*

Example:

```
/*

        Flashes LED on STK200 Board with Compare - Mode of Timer 2
        Timer starts with value $10, if the counter register has the same
        value as the ocr register the Led will be switched on.
        If the timer overflow flag is set the leds will be switched off.

        Leitner Harald
        07/00


*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t delay;

SIGNAL (SIG_OUTPUT_COMPARE2)
{

outp(0X00, PORTB);                  /* turn on leds on PORTB */

}

SIGNAL (SIG_OVERFLOW2)
{

outp(0XFF, PORTB);                  /* turn off leds on PORTB */
outp(delay,TCNT2);

}
```

```
int main( void )
{

outp(0xFF, DDRB);                    /* define PORTB as output (leds) */

delay = 0x10;

outp((1<<TOIE2)|(1<<OCIE2), TIMSK);  /* enables interrupt of timer */

outp(delay, TCNT2);                  /* default value of timer */

outp(0x80, OCR2);                    /* value of compar register */

outp(7, TCCR2);                      /* selected prescale : ck/1024 */

sei();

for (;;){}

}
```

This program shows a possibility of using timer2 in the compare-mode.
The timer2 is initialized in the *main()* function of the program.
Value *delay* ($10) is the starting value for the register *TCNT2* and the value of the compare register *OCR2* is set to *$80.*
Timer2 counts from *$10* to *$80*. At this point the program calls the interrupt routine for the compare match up and turns on the leds on *PORTB*. Then Timer2 counts until an overflow has occured. Then the interrupt routine for the overflow is reached and the leds are turned off.
Then the cycle starts counting endlessly from *delay ($10)*.

PWM mode:

When the PWM (Pulse Width Modulation) mode is selected, timer2 is used as a 8bit free running PWM. Timer2 acts as an up/down counter that is counting up from *$00* to *$FF*, where it turns and counts down to zero and repeats this cycle endlessly.
To choose this mode you have to set the bit *PWM2* (bit 6) of the register *TCCR2*. When the counter value matches the content of the compare register *(OCR2*) there will be effects on the output pin *OC2 (PORTD.7)* in the following way:

COM21	*COM20*	Description
0	0	no effects
0	1	no effects
1	0	cleared on compare match, up-counting, set on compare match, down-counting
1	1	cleared on compare match, down-counting, set on compare match, up-counting

(*COM21 -> bit5 and COM20 -> bit4 of register TCCR2)*

This program creates a free running PWM signal on the output pin *OC2.* The compare register *OCR2* defines the signals form.

Example:

```
/*
        Test program for Timer/Counter 2 PWM Mode
        The T/C is used as a free running PWM.
        The T/C is counting from $00 up to $FF and after
        reaching $FF the T/C is counting down to $00.
        Then the cycle repeats.
        When the counter value matches the content of the output
        compare register during counting up, PD7(OC2) pin is cleared
        and if matches while counting down the PD7(OC2) is set.

        Leitner Harald
        07/00

*/
```

```
#include <io.h>
#include <interrupt.h>
#include <signal.h>

int main( void )
{

outp(0xFF, DDRD);           /* use all pins on PORTD for output */

outp(0, TCNT2);             /* start value of T/C2 */

outp(0x67, TCCR2);          /* init the counter */

outp(0x19, OCR2);           /* value of OCR2 */

for (;;){}

}
```

Selected mode:

Prescale: *ck/1024*
PWM*: COM20 -> 1, COM21 -> 1*

UART (Universal Asynchronous Receiver Transmitter)

In order to receive or send data to a computer you need an UART.
This component enables a transfer of data through the seriell interface and is integrated in the controller.
In the STK200 board the data conductions are directly connected with a 9 pole SUB-D plug.
In order to be connected with the computer you need a seriell cable that is pinned on a free COM port of the computer and the SUB-D plug of the board. Besides a program has to be carried out on the computer that enables a connection through the seriell interface with the controller.

For the programming you need the speed in order to change the data between the program in the controller and the program in the computer. Both programms have to be adjusted to the same speed.
There is also the opportunity to connect two controller boards in order to transfer data.
Possible baud rates are dependent from the used quartz oscillator on the board and are mentioned in the suitable data sheet of the used processor. The value of the data sheet has to be loaded into the baud rate register *UBRR*. Therefore the UART is initialised with the right speed.

For example: *fosz = 4MHz, Baud rate = 9600*
→ *UBRR = 25* (data sheet)

The processor is operated with a 4MHz oscillator and the baud rate of the uart should be 9600. That necessary value that has to be written into register *UBRR* is according to the data sheet 25.

An other way is to calculate the right value with the following formula:

*Value of register UBRR = fosc / (baud rate * 16) – 1*

The actual data are either written or read into the data register *UDR*.

A further register *USR* gives information about the processing of the written or read data of the data register *UDR*.

USR - register:

Bit	name	description
0-2	*no*	no
3	*OR*	OverRun: This bit is set when a character already presented in the *UDR* data register is not read before the next character is shifted into the receiver shift register.
4	*FE*	FramingError: This bit is set when the stop bit of an incoming character is zero (Error).
5	*UDRE*	Uart Data Register Empty: This bit is set when data is transfered from the register *UDR* to the shift register. → ready for transmission of new data
6	*TXC*	UART transmission complete:This bit is set when data is shifted out from the shift register and no new data is written into data register *UDR*.
7	*RXC*	UART receiving complete:This bit is set if new data is in register *UDR*. This bit is cleared by reading the new data from the *UDR* register.

The *UCR* register is the control register of the uart. With this register you can choose the different modes of operation.

UCR - register:

Bit	name	description
0	*TXB8*	9th data bit for transmission, when *CHR9* bit is set.
1	*RXB8*	9th data bit which has been received, when *CHR9* bit is set.
2	*CHR9*	9 bit characters: If this bit is set the received and transmitted charcters are 9bit long. The 9th bit is in *TXB8* for transmission or when you receive a 9bit character the 9th bit is in *RXB8*.
3	*TXEN*	Transmitter Enable: If you set this bit the transmitter is enabled.
4	*RXEN*	Receiver Enable: If you set this bit the reveicer is enabled.
5	*UDRIE*	Uart Data Register Empty Interrupt Enable: When this bit is set, a setting of the *UDRE* bit in register *USR* will cause the uart data register empty interrupt routine to be executed provided that global interrupts are enabled.
6	*TXCIE*	UART Transmission Complete Interrupt Enable: When this bit is set and a transmission is ready the Interrupt routine *SIG_UART_TRANS* will be executed.
7	*RXCIE*	UART Receiving Complete Interrupt Enable: When this bit is set and a character has been received, the Interrupt routine *SIG_UART_RECV* will be executed.

Through adjusting these registers new routines for the data exchange can be created. In this chapter the programming that is controlled by the interrupt is mentioned.

Transmit mode:

Example:

```
/*


        This program transfers the string a of the program
        memory to the uart.

        10/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>
#include <progmem.h>

char a[] __attribute__((progmem)) = "This is a test message!";

uint8_t pos=0;

SIGNAL(SIG_UART_TRANS)
{

if (pos++ < 23)
        outp(PRG_RDB(&a[pos]), UDR);

}

int main(void)
{

outp((1<<TXCIE)|(1<<TXEN),UCR);      /* enable TX interrupt */

outp(25, UBRR);                      /* init transfer speed */

sei();                               /* enable interrupts */

outp(PRG_RDB(&a[pos]), UDR);         /* write byte to data buffer */

for(;;){}
}
```

In this program a character string that is situated in the program memory is written into the data register *UDR* step by step and transmittes through a seriell cable to a reception place.
The baud rate is determined in register *UBRR* with 9600.
Then the first character of the string is given to the data register *UDR*. As soon as the first character has been transmitted, the interrupt routine is called up and the next character has to be transmitted.
This happens as long as the last character is transmitted.

Receive mode:

Example:

```
/*

        This program receives data from the uart and
        shows it through the leds on the PORTB.

        10/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

SIGNAL(SIG_UART_RECV)
{

outp(inp(UDR),PORTB);           /* read data from uart and
                                   transfer it to PORTB */
}

int main(void)
{
outp(0Xff,DDRB);
outp((1<<RXCIE)|(1<<RXEN),UCR);     /* enable RX interrupt */
outp(25, UBRR);                         /* init transfer speed */
sei();                                  /* enable interrupts */
for(;;){}

}
```

This program transmitts with the speed of 9600baud each character sent from the computer into the data register *UDR*. Because of that the correct interrupt routine is called up and the character is read of the register *UDR*.
The ASCII code of the character is shown through the leds on *PORTB*.

Transmitt / Receive mode:

Example:

```
/*

        This program writes the string1 to the uart.
        Then every read data is written to PORTB and
        to the uart back.

        10/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t STRING1[6] = "Input:";
uint8_t pos=0;

SIGNAL(SIG_UART_TRANS)
{

if (pos++ < 7)                          /* character of string to uart */
        outp(STRING1[pos], UDR);
else
        pos = 7;

}

SIGNAL(SIG_UART_RECV)
{
uint8_t recval;

recval = inp (UDR);                     /* read data from uart buffer */
```

```
outp(recval,PORTB);                 /* write data to PORTB */

outp(recval, UDR);                  /* write data to uart buffer */

}

int main(void)
{

outp(0xff,DDRB);

    /* init the uart for receiving and transmitting of data */
outp((1<<RXCIE)|(1<<TXCIE)|(1<<RXEN)|(1<<TXEN),UCR);

outp(25, UBRR);                     /* init speed */

sei();                              /* enable interrupts */

outp(STRING1[pos], UDR);            /* write 1st byte of string1 to data
                                       buffer */
for(;;){}

}
```

In this program the UART is used as a receiver and transmitter. At first the character string „Input:“ is transmitted through the seriell interface. Afterwards the ASCII code of each received character is shown through the leds on *PORTB.* The received character itself is sent back through the seriell interface.

How to use the Analog/Digital Converter ?

An A/D converter in the controller was integrated for the processing of analog signals. This A/D converter is able to show analog signals with a resolution of *10 bit*.
The A/D converter works with the procedure of the successive approximation. In order to process different analog signals, a multiplexer is placed before the converter. This multiplexer is connected with PORTA. Therefore it is possible to process a signal about each of the eight pins on *PORTA*.
At any time only one of the eight signals can be converted.
Register *ADMUX,* with the following value, adjusts which of the eight channels are connected with the A/D converter.

value of ADMUX	used channel
0	*channel 0*
1	*channel 1*
2	*channel 2*
3	*channel 3*
4	*channel 4*
5	*channel 5*
6	*channel 6*
7	*channel 7*

Besides the A/D converter needs an own tact signal that is between *50* and *200kHz*. This signal is produced out of the frequency of the quartz oscillator. Through the use of a prescale function the needed frequency for the A/D converter is created. This prescale value is adjusted through the following three bits *ADPS0, ADPS1 and ADPS2* in register *ADCSR*:

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

for example:

fosz = 4MHz
50kHz < fad < 200kHz
→ prescale = 32
fad = fosz/32 = 4000000/32 = 125000 = 125 kHz
50kHz < 125kHz < 200kHz
→ ADPS0 = 1, ADPS1 = 0, ADPS2 = 1

ADCSR:

bit	name	description
0	*ADPS0*	prescale
1	*ADPS1*	prescale
2	*ADPS2*	prescale
3	*ADIE*	A/D Interrupt Enable: if set, A/D conversion complete interrupt is enabled.
4	*ADIF*	A/D Interrupt Flag: This bit is set, when a A/D conversion is completet.
5	*ADFR*	A/D Free Run Select: If you set this bit, the AD converter free operates in the running mode.
6	*ADSC*	A/D Start Conversion: If you set this bit and the *ADEN* bit is set a Conversion is started.
7	*ADEN*	A/D Enable: To enable the A/D - converter you have to set this bit.

The A/D converter can be used in two different modes of use:
The first is called **„single conversion“** mode. In this case only one A/D conversion is carried out. Afterwards the bit *ADSC* has to be set again for a new conversion.
The second mode of use is the **„free running“** mode. Here the bit *ADFR* has to be set and the conversion with bit *ADSC* started.
Afterwards a conversion is carried out permanently.

The *10* bit value is written into the data registers *ADCL* (low byte) and *ADCH* (high byte). When reading these registers you have to pay attention that register *ADCL* has to be read before *ADCH*.

Single conversion mode:

Example:

```
/*

      Analog to digital conversion of value on analog input 0 in single
      conversion mode. Result lo_val is presented on port B (Leds)

      10/00 Leitner Harald
 */

#include <io.h>
#include <interrupt.h>
#include <signal.h>

SIGNAL(SIG_ADC)
{
uint8_t lo_val, hi_val;

lo_val = inp(ADCL);     /* read low byte first and then the high byte */

hi_val = inp(ADCH);

outp(lo_val,PORTB);          /* write low byte to PORTB */

}

int main( void )

{

outp(0xFF, DDRB);            /* define PORTB as Output (Leds) */

outp(0, ADMUX);              /* Select Analog input 0*/

                             /* Enables ADC and start one conversion */
outp((1<<ADEN)|(1<<ADSC)|(1<<ADIF)|(1<<ADIE),ADCSR);

sei();

for (;;){}

}
```

For this program the A/D converter input zero on *PORTA* is choosen and register *ADCSR* is adjusted to the single conversion mode.
After the first conversion the interrupt routine is carried out and the value is read out of the data registers. The low byte is shown through the leds on *PORTB*.

Free running mode:

Example:

```
/*

        Analog to digital conversion of value on analog input 0
        in free running mode.
        Result is presented on PORTB (Leds)

        7/00 Leitner Harald

*/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

uint8_t lo_val, hi_val;

SIGNAL (SIG_ADC)
{

lo_val = inp(ADCL);                     /* read low byte first */

hi_val = inp(ADCH);

outp(~lo_val, PORTB);

}

int main( void )
{

outp(0xFF, DDRB);               /* define PORTB as Output (Leds) */
```

```
outp(0, ADMUX);                    /* Select Analog input 0*/

/* Enables ADC and start conversion in free running interrupt mode */
outp((1<<ADEN)|(1<<ADSC)|(1<<ADFR)|(1<<ADIE), ADCSR);

sei();

for (;;){}

}
```

This program fulfills the same duty. The converted value is permanently updated through the free running mode.

Analog Comparator:

The analog comparator compares the voltage on the input *AIN0 (PORTB.2)* and the input *AIN1 (PORTB.3)*. When the voltage on pin *AIN0* is higher than the voltage on pin *AIN1* bit *ACO* in register *ACSR* is set. It is also possible to trigger a separate interrupt. You can select interrupt triggering on comparator output rising, fall or toggle.
This function is controlled through the bits *ACIS0* and *ACIS1* in the following way:

ACIS0	*ACIS1*	description
0	0	Comparator on output toggle.
0	1	not used
1	0	Comparator interrupt on falling output edge.
1	1	Comparator interrupt on rising output edge

Register *ACSR*:

bit	name	description
0	*ACIS0*	interrupt select
1	*ACIS1*	interrupt select
2	*ACIC*	Analog Comparator Input Capture Enable: trigger input capture function of timer/counter 1
3	*ACIE*	Analog Comparator Interrupt Enable: When this bit is set interrupt of analog comparator is enabled.
4	*ACI*	Analog Comparator Interrupt Flag: This bit is set if a interrupt is executed with the definition through ACIS0 and ACIS1. Bit is cleared through writing a one on this flag or executing the interrupt routine.
5	*ACO*	Analog Comparator Output: Directly from the comparator.
6	no	Reserved
7	*ACD*	Analog Comparator Disable: If this bit is set the analog comparator is turned off.

Example:

```
/*

    The Analog Compator executes an interrupt if
```

```
        the voltage on pin PORTB.2 is higher than the
        voltage on PORTB.3.
        Then the interrupt routine is started and the
        leds on PORTB are turned on.

        7/00 Leitner Harald

 */

#include <io.h>
#include <interrupt.h>
#include <signal.h>

SIGNAL(SIG_COMPARATOR)
{

 outp(0xFE,PORTB);      /* turn on 1st led on PORTB */

}

int main( void )
{

 outp(0xF3,DDRB);       /* pin 2 and 3 PORTB set as input, other output */
 outp(0xFF,PORTB);
 outp((1<<ACIE)|(1<<ACIS1)|(1<<ACIS1),ACSR); /* init comparator */
 sei();
 for (;;){}

}
```

This program compares both inputs *AIN0* and *AIN1*. The interrupt routine is carried out if the voltage on *AIN0* is bigger than on the input *AIN1*. During that the first led on *PORTB* is turned on.