

Introduction to AVR assembler programming for beginners

Controlling sequential execution of the program

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

What happens during a reset?

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

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

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

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

WDR

to avoid a reset.

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

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

.CSEG
.ORG 0000

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

.ESEG

The third segment is the SRAM section of the chip.

.DSEG

The ORG directive above stands for origin and manipulates the address within the code segment, where assembled words go to. As our program always starts at 0x0000 the CSEG/ORG directives are trivial, you can skip these without getting into an error. We could start at 0x0100, but that makes no real sense. If you want to place a table exactly to a certain location of the code segment, you can use ORG. If you want to set a clear sign within your code, after first defining a lot of other things with .DEF- and .EQU-directives, use the CSEG/ORG sequence, even though it might not be necessary to do that.

As the first code word is always at address zero, this location is also called the reset vector. Following the reset vector the next positions in the program space, addresses 0x0001, 0x0002 etc., are interrupt vectors. These are the positions where the execution jumps to if an external or internal interrupt has been enabled and occurs. These positions called vectors are specific for each processor type and depend on the internal hardware available (see below). The commands to react to such an interrupt have to be placed to the proper vector location. If you use interrupts, the first code, at the reset vector, must be a jump command, to jump over the other vectors. Each interrupt vector must hold a jump command to the respective interrupt service routine. The typical program sequence at the beginning is like follows:

.CSEG

.ORG 0000

RJMP Start

RJMP IntServRout1

[...] here we place the other interrupt vector commands

[...] and here is a good place for the interrupt service routines themselves

Start: ; This here is the program start

[...] Here we place our main program

The command *RJMP* results in a jump to the label *Start:*, located some lines below. Labels always start in column 1 of the source code and end with a *:*. Labels, that don't fulfil these conditions are not taken for serious by the compiler. Missing labels result in an error message ("Undefined label"), and compilation is interrupted.

[To the top of this page](#)

Linear program execution and branches

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

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

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

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

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

```
INC R1  
BRNE GoOn32  
INC R2  
BRNE GoOn32  
INC R3  
BRNE GoOn32  
INC R4  
GoOn32:
```

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

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

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

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

[To the top of that page](#)

Timing during program execution

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

read/write sequence. See the instructions table for details.

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

NOP

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

```
CLR R1  
Count:  
DEC R1  
BRNE Count
```

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

```
LDI ZH,HIGH(65535)  
LDI ZL,LOW(65535)  
Count:  
SBIW ZL,1  
BRNE Count
```

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

[To the top of that page](#)

Macros and program execution

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

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

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

[...] somewhere in the source code

Delay1

[...] code goes on here

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

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

If your macro has longer code sequences, or if you are short in code storage space, you should avoid the use of macros and use subroutines instead.

[To the top of that page](#)

Subroutines

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

Delay10:

NOP

NOP

NOP

RET

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

[...] somewhere in the source code:

RCALL Delay10

[...] further on with the source code

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

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

[...] somewhere in the source code

RJMP Delay10

Return:

[...] further on with source code

The routine that you jumped to can not use the *RET* command in that case. To return back to the calling location in the source requires to add another label and the called routine to jump back to

this label. Jumping like this is not like calling a subroutine because you can't call this routine from different locations in the code.

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

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

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

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

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

A rarely used command, forget it for the beginning.

If you like to skip the following instruction depending on a certain bit in a port use the following instructions *SBIC* and *SBIS*. That reads Skip if the Bit in I/o space is Clear (or Set), like this:

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

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

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

MyTab:
RJMP Routine1
RJMP Routine2
[...]
RJMP Routine16

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

LDI ZH,HIGH(MyTab)

LDI ZL,LOW(MyTab)

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

ADD ZL,R16

BRCC NoOverflow

INC ZH

NoOverflow:

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

ICALL

or as a jump with no way back:

IJMP

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

[To the top of that page](#)

Interrupts and program execution

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

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

SEI ; Set Int Enable

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

Name	Int Vector Address	triggered by ...
------	--------------------	------------------

	2313	2323	8515	
RESET	0000	0000	0000	Hardware Reset, Power-On Reset, Watchdog Reset
INT0	0001	0001	0001	Level change on the external INT0-Pin
INT1	0002	-	0002	Level change on the external INT1-Pin
TIMER1 CAPT	0003	-	0003	Capture event on Timer 1
TIMER1 COMPA	-	-	0004	Timer1 = Compare A
TIMER1 COMPB	-	-	0005	Timer1 = Compare B
TIMER1 COMP1	0004	-	-	Timer1 = Compare 1
TIMER1 OVF	0005	-	0006	Timer1 Overflow
TIMER0 OVF	0006	0002	0007	Timer0 Overflow
SPI STC	-	-	0008	Serial transmit complete
UART RX	0007	-	0009	UART char in receive buffer available
UART UDRE	0008	-	000A	UART transmitter ran empty
UART TX	0009	-	000B	UART All sent
ANA_COMP	-	-	000C	Analog Comparator

Note that the capability to react to events is very different for the different types. The addresses are sequential, but not identical for different types. The higher a vector in the list the higher is its priority. If two components have an interrupt condition at the same time the upmost vector with the lower vector address wins. The lower int has to wait until the upper int was served. To disable lower ints from interrupting during the execution of its service routine the first executed int disables the processor's I-flag. The service routine must re-enable this flag after it is done with its job.

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

RETI

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

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

SEI ; Set Interrupt Enabled

RET ; Return

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

An Int-vector can only hold a relative jump instruction to the service routine. If a certain int is not used or undefined we can just put a *RETI* instruction there, in case a false int happens.

Note that larger devices have a two-word organization of the vector table. In this case the *JMP* instruction has to be used instead of *RJMP*. And *RETI* instructions must be followed by an *NOP* to point to the next vector table address.

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

A very hard rule for int service routines is: First instruction is to save the status register on the stack, before you use instructions that might change flags in the status register. The interrupted main program might just be in a state using the flag for a branch decision, and the int would just change that flag to another state. Funny things would happen from time to time. The last instruction before the *RETI* therefore is to pop the status register content from the stack and restore its original content, prior to the int.

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

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

.CSEG ; Code-Segment starts here

.ORG 0000 ; Address is zero

RJMP Start ; The reset-vector on Address 0000

RJMP IService ; 0001: first Int-Vektor, INT0 service routine

[...] here other vectors

Start: ; Here the main program starts

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

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

PUSH R16 ; save a register to stack

IN R16,SREG ; read status register

PUSH R16 ; and put on stack

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

POP R16 ; get previous flag register from stack

OUT SREG,R16 ; restore old status

POP R16 ; get previous content of R16 from the stack

RETI ; and return from int

Looks a little bit complicated, but is a prerequisite for using ints without producing serious bugs. Skip *PUSH R16* and *POP R16* if you can afford reserving R16 for exclusive use in that service routine.

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

Calculations in assembler language

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

Number systems in assembler

The following formats of numbers are common in assembler:

- [Positive whole numbers \(Integers\)](#),
- [Signed whole numbers](#),
- [Binary Coded Digits, BCD](#),
- [Packed BCDs](#),
- [ASCII-formatted numbers](#).

Positive whole numbers (integers)

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

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

```
.DEF r16 = dw0
```

```
.DEF r17 = dw1
```

```
.DEF r18 = dw2
```

```
.DEF r19 = dw3
```

dw0 to dw3 are in a row in the registers. If we need to initiate this double word at the beginning of an application (e.g. to 4,000,000), this should look like this:

```
.EQU dwi = 4000000 ; define the constant
```

```
LDI dw0,LOW(dwi) ; The lowest 8 bits to R16
```

```
LDI dw1,BYTE2(dwi) ; bits 8 .. 15 to R17
```

LDI dw2,BYTE3(dwi) ; bits 16 .. 23 to R18
LDI dw3,BYTE4(dwi) ; bits 24 .. 31 to R19

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

[To the top of that page](#)

Signed numbers

Sometimes, but in rare cases, you need negative numbers to calculate with. A negative number is defined by interpreting the most significant bit of a byte as sign bit. If it is 0 the number is positive. If it is 1 the number is negative. If the number is negative we usually do not store the rest of the number as is, but we use its inverted value. Inverted means that -1 as an byte integer is not written as 1,0000001 but as 1,1111111 instead. That means: subtract 1 from 0 and forget the overflow. The first bit is the sign bit, signalling that this is a negative number. Why this different format (subtracting the negative number from 0) is used is easy to understand: adding -1 (1,1111111) and +1 (0,0000001) yields exactly zero, if you forget the overflow that occurs during that operation (the ninth bit).

In one byte the biggest number to be handled is +127 (binary 0,1111111), the smallest one is -128 (binary 1,0000000). In other computer languages this number format is called short integer. If you need a bigger range of values you can add another byte to form a normal integer value, ranging from +32,767 .. -32,768), four bytes provide a range from +2,147,483,647 .. -2,147,483,648, usually called a LongInt or DoubleInt.

[To the top of that page](#)

Binary Coded Digits, BCD

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

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

Instructions to use:

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

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

[To the top of that page](#)

Packed BCDs

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

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

Instructions for setting:

LDI R17,0x02 ; Upper byte

LDI R16,0x50 ; Lower byte

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

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

[To the top of that page](#)

Numbers in ASCII-format

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

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

LDI R18,'2'

LDI R17,'5'

LDI R16,'0'

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

[To the top of that page](#)

Bit manipulations

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

ORI R1,0x30

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

OR R1,R2

Back from an ASCII character to a BCD is a bit more complicated in AVR assembler, because the instruction

ANDI R1,0x0F

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

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

AND R1,R2

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

SBR R16,0b00110000 ; Set bits 4 und 5 to one

CBR R16,0b00110000 ; Clear bits 4 and 5 to zero

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

LDI R16,0b10101010 ; Invert all even bits

EOR R1,R16 ; in register R1 and store result in R1

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

COM R1

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

NEG R1

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

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

BLD R1,0

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

CLT ; clear T-bit, or

SET ; set T-bit, or

BST R2,2 ; copy register R2, bit 2, to the T-bit

[To the top of that page](#)

Shift and rotate

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

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

LSL R1

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

LSR R1

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

Example, division by four with rounding:

LSR R1 ; division by 2

BRCC Div2 ; Jump if no round up

INC R1 ; round up

Div2:

LSR R1 ; Once again division by 2

BRCC DivE ; Jump if no round up

INC R1 ; Round Up

DivE:

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

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

ASR R1

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

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

LSL R1 ; Logical Shift Left of the lower byte

ROL R2 ; ROfate Left of the upper byte

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

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

LSR R2 ; Logical Shift Right, bit 0 to carry

ROR R1 ; ROfate Right and shift carry in bit 7

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

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

ROR R1

ROR R1

ROR R1

ROR R1

we can perform that with a single

SWAP R1

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

[To the top of that page](#)

Adding, subtracting and comparing

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

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

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

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

We subtract R3:R4 from R1:R2.

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

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

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

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

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

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

CPI R16,0xAA

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

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

TST R1

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

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

to more than 15 decimal. To take an example we add the packed BCDs 49 (=hex 49) and 99 (=hex 99) to yield 148 (=hex 0x148). Adding these in binary math, results in a byte holding hex 0xE2, no byte overflow occurs. The lower of the two nibbles has had an overflow because $9+9=18$ and the lower nibble can only handle numbers up to 15. The overflow was added to bit 4, the lowest significant bit of the upper nibble. Which is correct! But the lower nibble should be 8 and is only 2 ($18 = 0b0001.0010$). We should add 6 to that nibble to yield a correct result. Which is quite logic, because whenever the lower nibble reaches more than 9 we have to add 6 to correct that nibble.

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

How do we know, if an overflow from the lower to the upper nibble has occurred? The MCU sets a H-bit in the status register, the half-carry bit. The following table shows the different cases that are possible after adding R1 and R2 and adding hex 0x66 after that.

Add R1,R2 (Half)Carry-Bit	Add Nibble,6 (Half)Carry-Bit	Correction
0	0	subtract 6
1	0	none
0	1	none
1	1	(not possible)

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

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

NoCy1:

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

NoHc1:

```
ADD R2,R16 ; add 0x66 to result
BRCC NoCy2 ; jump if no carry occurred
```

INC R1 ; increment the next higher byte
ANDI R17,0x0F ; don't subtract 6 from higher nibble
NoCy2:
BRHC NoHc2 ; jump if no half-carry occurred
ANDI R17,0xF0 ; don't subtract 6 from lower nibble
NoHc2:
SUB R2,R17 ; subtract correction

A little bit shorter than that:

LDI R16,0x66
ADD R2,R16
ADD R2,R3
BRCC NoCy
INC R1
ANDI R16,0x0F
NoCy:
BRHC NoHc
ANDI R16,0xF0
NoCy:
SUB R2,R16

Question to think about: Why is that equally correct and where is the trick?

[To the top of that page](#)

Format conversion for numbers

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

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

A little bit more complicated is the conversion of BCD digits to a binary. Depending on the numbers to be handled we first clear the necessary bytes that will hold the result of the conversion. We then start with the highest BCD digit. Before adding this to the result we multiply the result with 10. In order to do this we copy the result to somewhere else. Then we multiply the result by four (two left shifts resp. rolls). Adding the previously copied result to this yields a multiplication with 5. Now a multiplication with 2 (left shift/roll) yields the 10-fold of the result. Now we add the BCD and repeat that algorithm until all decimal digits are converted. If, during one of these operations, there occurs a carry of the result, the BCD is too big to be converted.

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

DezTab:

.DW 10000, 1000, 100, 10

and can be read wordwise with the *LPM* instruction from the table.

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

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

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

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

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

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

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

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

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

Controlling sequential execution of the program

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

What happens during a reset?

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

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

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

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

WDR

to avoid a reset.

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

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

.CSEG

.ORG 0000

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

.ESEG

The third segment is the SRAM section of the chip.

.DSEG

The ORG directive above stands for origin and manipulates the address within the code segment, where assembled words go to. As our program always starts at 0x0000 the CSEG/ORG directives are trivial, you can skip these without getting into an error. We could start at 0x0100, but that makes no real sense. If you want to place a table exactly to a certain location of the code segment, you can use ORG. If you want to set a clear sign within your code, after first defining a lot of other things with .DEF- and .EQU-directives, use the CSEG/ORG sequence, even though it might not be necessary to do that.

As the first code word is always at address zero, this location is also called the reset vector. Following the reset vector the next positions in the program space, addresses 0x0001, 0x0002 etc., are interrupt vectors. These are the positions where the execution jumps to if an external or internal interrupt has been enabled and occurs. These positions called vectors are specific for each processor type and depend on the internal hardware available (see below). The commands to react to such an interrupt have to be placed to the proper vector location. If you use interrupts, the first code, at the reset vector, must be a jump command, to jump over the other vectors. Each interrupt vector must hold a jump command to the respective interrupt service routine. The typical program sequence at the beginning is like follows:

.CSEG

.ORG 0000

RJMP Start

RJMP IntServRout1

[...] here we place the other interrupt vector commands

[...] and here is a good place for the interrupt service routines themselves

Start: ; This here is the program start

[...] Here we place our main program

The command *RJMP* results in a jump to the label *Start:*, located some lines below. Labels always start in column 1 of the source code and end with a *:*. Labels, that don't fulfil these conditions are not taken for serious by the compiler. Missing labels result in an error message ("Undefined label"), and compilation is interrupted.

[To the top of this page](#)

Linear program execution and branches

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

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

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

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

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

```
INC R1
BRNE GoOn32
INC R2
BRNE GoOn32
INC R3
BRNE GoOn32
INC R4
GoOn32:
```

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

Which of the status bits, also called processor flags, are changed during execution of a command is listed in instruction code tables, see the [List of Instructions](#). Similiarly to the Zero-bit you can use

the other status bits like that:

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

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

[To the top of that page](#)

Timing during program execution

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

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

NOP

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

```
CLR R1
Count:
DEC R1
BRNE Count
```

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

```
LDI ZH,HIGH(65535)
LDI ZL,LOW(65535)
Count:
```

SBIW ZL,1 BRNE Count

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

[To the top of that page](#)

Macros and program execution

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

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

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

```
[...] somewhere in the source code  
  Delay1  
[...] code goes on here
```

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

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

If your macro has longer code sequences, or if you are short in code storage space, you should avoid the use of macros and use subroutines instead.

[To the top of that page](#)

Subroutines

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

```
Delay10:  
  NOP
```

NOP
NOP
RET

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

[...] somewhere in the source code:

RCALL Delay10

[...] further on with the source code

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

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

[...] somewhere in the source code

RJMP Delay10

Return:

[...] further on with source code

The routine that you jumped to can not use the *RET* command in that case. To return back to the calling location in the source requires to add another label and the called routine to jump back to this label. Jumping like this is not like calling a subroutine because you can't call this routine from different locations in the code.

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

SBRC R1,7 ; Skip the next instruction if bit 7 is 0

RCALL UpLabel ; Call that subroutine

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

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

CPSE R1,R2 ; Compare R1 and R2, skip if equal

RCALL SomeSubroutine ; Call SomeSubroutine

A rarely used command, forget it for the beginning.

If you like to skip the following instruction depending on a certain bit in a port use the following instructions *SBIC* und *SBIS*. That reads Skip if the Bit in I/o space is Clear (or Set), like this:

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

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

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

MyTab:
RJMP Routine1
RJMP Routine2
[...]
RJMP Routine16

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

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

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

ADD ZL,R16
BRCC NoOverflow
INC ZH
NoOverflow:

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

ICALL

or as a jump with no way back:

IJMP

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

[To the top of that page](#)

Interrupts and program execution

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

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

SEI ; Set Int Enable

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

Name	Int Vector Address			triggered by ...
	2313	2323	8515	
RESET	0000	0000	0000	Hardware Reset, Power-On Reset, Watchdog Reset
INT0	0001	0001	0001	Level change on the external INT0-Pin
INT1	0002	-	0002	Level change on the external INT1-Pin
TIMER1 CAPT	0003	-	0003	Capture event on Timer 1
TIMER1 COMPA	-	-	0004	Timer1 = Compare A
TIMER1 COMPB	-	-	0005	Timer1 = Compare B
TIMER1 COMP1	0004	-	-	Timer1 = Compare 1
TIMER1 OVF	0005	-	0006	Timer1 Overflow
TIMER0 OVF	0006	0002	0007	Timer0 Overflow
SPI STC	-	-	0008	Serial transmit complete
UART RX	0007	-	0009	UART char in receive buffer available
UART UDRE	0008	-	000A	UART transmitter ran empty
UART TX	0009	-	000B	UART All sent
ANA_COMP	-	-	000C	Analog Comparator

Note that the capability to react to events is very different for the different types. The addresses are sequential, but not identical for different types. The higher a vector in the list the higher is its

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

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

RETI

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

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

SEI ; Set Interrupt Enabled

RET ; Return

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

An Int-vector can only hold a relative jump instruction to the service routine. If a certain int is not used or undefined we can just put a *RETI* instruction there, in case a false int happens.

Note that larger devices have a two-word organization of the vector table. In this case the *JMP* instruction has to be used instead of *RJMP*. And *RETI* instructions must be followed by an *NOP* to point to the next vector table address.

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

A very hard rule for int service routines is: First instruction is to save the status register on the stack, before you use instructions that might change flags in the status register. The interrupted main program might just be in a state using the flag for a branch decision, and the int would just change that flag to another state. Funny things would happen from time to time. The last instruction before the *RETI* therefore is to pop the status register content from the stack and restore its original content, prior to the int.

For the same reason all used registers in a service routine should either be exclusively reserved for that purpose or saved on stack and restored at the end of the service routine. Never change the

content of a register within a int service routine that is used somewhere else in the normal program without restoring it.

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

.CSEG ; Code-Segment starts here

.ORG 0000 ; Address is zero

RJMP Start ; The reset-vector on Address 0000

RJMP IService ; 0001: first Int-Vektor, INT0 service routine

[...] here other vectors

Start ; Here the main program starts

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

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

PUSH R16 ; save a register to stack

IN R16,SREG ; read status register

PUSH R16 ; and put on stack

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

POP R16 ; get previous flag register from stack

OUT SREG,R16 ; restore old status

POP R16 ; get previous content of R16 from the stack

RETI ; and return from int

Looks a little bit complicated, but is a prerequisite for using ints without producing serious bugs. Skip *PUSH R16* and *POP R16* if you can afford reserving R16 for exclusive use in that service routine.

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

What is a register?

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



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

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

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

- they can be used directly in assembler commands,
- operations with their content require only a single command word,

- they are connected directly to the central processing unit called the accumulator,
- they are source and target for calculations.

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

.DEF MyPreferredRegister = R16

Note that assembler directives like this are only meaningful for the assembler but do not produce any code that is executable in the AVR target chip. Instead of using the register name R16 we can now use our own name MyPreferredRegister, if we want to use R16 within a command. So we write a little bit more text each time we use this register, but we have an association what might be the content of this register.

Using the command line

LDI MyPreferredRegister, 150

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

000000 E906

The load command code as well as the target register (R16) as well as the value of the constant (150) is part of the hex value E906, even if you don't see this directly. Don't be afraid: you don't have to remember this coding because the assembler knows how to translate all this to yield E906.

Within one command two different registers can play a role. The easiest command of this type is the copy command MOV. It copies the content of one register to another register. Like this:

.DEF MyPreferredRegister = R16

.DEF AnotherRegister = R15

LDI MyPreferredRegister, 150

MOV AnotherRegister, MyPreferredRegister

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

000000 E906

000001 2F01

The commands write 150 into register R16 and copy its content to the target register R15.

IMPORTANT NOTE:

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

(This is unfortunately different from what one expects or from how we speak. It is a simple convention that was once defined that way to confuse the beginners learning assembler. That is why assembler is that complicated.)

[To the top of that page](#)

Different registers

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

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

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

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

```
CLR MyPreferredRegister
```

is valid for all registers.

Besides the LDI command you will find this register class restriction with the following additional commands:

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

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

[To the top of that page](#)

Pointer-register

A very special extra role is defined for the register pairs R26:R27, R28:R29 and R30:R31. The role is so important that these pairs have extra names in assembler: X, Y and Z. These pairs are 16-bit pointer registers, able to point to addresses with max. 16-bit into SRAM locations (X, Y or Z) or into locations in program memory (Z).

The lower byte of the 16-bit-address is located in the lower register, the higher byte in the upper

register. Both parts have their own names, e.g. the higher byte of Z is named ZH (=R31), the lower Byte is ZL (=R30). These names are defined in the standard header file for the chips. Dividing these 16-bit-pointer-names into two different bytes is done like follows:

.EQU Adress = RAMEND ; RAMEND is the highest 16-bit address in SRAM

LDI YH,HIGH(Adress) ; Set the MSB

LDI YL,LOW(Adress) ; Set the LSB

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

Pointer	Sequence	Example
X	Read/Write from adress X, don't change the pointer	LD R1,X ST X,R1
X+	Read/Write from/to adress X and increment the pointer afterwards by one	LD R1,X+ ST X+,R1
-X	Decrement the pointer by one and read/write from/to the new adress afterwards	LD R1,-X ST -X,R1

Similiarly you can use Y and Z for that purpose.

There is only one command for the read access to the program storage. It is defined for the pointer pair Z and it is named LPM (Load from Program Memory). The command copies the byte at adress Z in the program memory to the register R0. As the program memory is organised word-wise (one command on one adress consists of 16 bits or two bytes or one word) the least significant bit selects the lower or higher byte (0=lower byte, 1= higher byte). Because of this the original adress must be multiplied by 2 and access is limited to 15-bit or 32 kB program memory. Like this:

LDI ZH,HIGH(2*Adress)

LDI ZL,LOW(2*Adress)

LPM

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

ADIW ZL,1

LPM

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

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

As incrementation after reading is very often needed, newer AVR types have the instruction

LPM R,Z+

This allows to transport the byte read to any location R, and auto-increments the pointer register.

How to insert that table of values in the program memory? This is done with the assembler directives **.DB** and **.DW**. With that you can insert byte-wise or word-wise lists of values. Byte-wise organised lists look like this:

.DB 123,45,67,89 ; a list of four bytes

.DB "This is a text. " ; a list of byte characters

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

The similar list of words looks like this:

.DW 12345,6789 ; a list of two words

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

Label1:

[... here are some commands ...]

Label2:

[... here are some more commands ...]

Table:

.DW Label1,Label2 ; a word-wise list of labels

Note that reading the labels with **LPM** first yields the lower byte of the word.

A very special application for the pointer registers is the access to the registers themselves. The registers are located in the first 32 bytes of the chip's address space (at address 0x0000 to 0x001F). This access is only meaningful if you have to copy the register's content to SRAM or EEPROM or read these values from there back into the registers.

More common for the use of pointers is the access to tables with fixed values in the program memory space. Here is, as an example, a table with 10 different 16-bit values, where the fifth table value is read to R25:R24:

MyTable:

.DW 0x1234,0x2345,0x3456,0x4568,0x5678 ; The table values, word-wise

.DW 0x6789,0x789A,0x89AB,0x9ABC,0xABCD ; organised

Read5: LDI ZH,HIGH(MyTable*2) ; Address of table to pointer Z

LDI ZL,LOW(MyTable*2) ; multiplied by 2 for byte-wise access

ADIW ZL,10 ; Point to fifth value in table

LPM ; Read least significant byte from program memory

MOV R24,R0 ; Copy LSB to 16-bit register

ADIW ZL,1 ; Point to MSB in program memory

LPM ; Read MSB of table value

MOV R25,R0 ; Copy MSB to 16-bit register

This is only an example. You can calculate the table address in Z from some input value, leading to the respective table values. Tables can be organised byte- or character-wise, too.

[To the top of that page](#)

Recommendation for the use of registers

1. Define names for registers with the .DEF directive, never use them with their direct name Rx.
2. If you need pointer access reserve R26 to R31 for that purpose.
3. 16-bit-counter are best located R25:R24.
4. If you need to read from the program memory, e.g. fixed tables, reserve Z (R31:R30) and R0 for that purpose.
5. If you plan to have access to single bits within certain registers (e.g. for testing flags), use R16 to R23 for that purpose.

6. What is a Port?

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

There are 64 different ports, which are not physically available in all different AVR types. Depending on the storage space and other internal hardware the different ports are either available and accessible or not. Which of these ports can be used is listed in the data sheets for the processor type.

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

.EQU PORTB, 0x18

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

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

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

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

As an example the MCU General Control Register, called MCUCR, consists of a number of single control bits that control the general property of the chip (see the description in [MCUCR in detail](#)). It is a port, fully packed with 8 control bits with their own names (ISC00, ISC01, ...). Those who want to send their AVR to a deep sleep need to know from the data sheet how to set the respective bits. Like this:

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

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

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

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

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

More often than reading all 8 bits of a port one must react to a certain status of a port. In that case we don't need to read the whole port and isolate the relevant bit. Certain commands provide an opportunity to execute commands depending on the level of a certain bit (see the [JUMP](#) section). Setting or clearing certain bits of a port is also possible without reading and writing the other bits in the port. The two commands are SBI (Set Bit I/o) and CBI (Clear Bit I/o). Execution is like this:

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

These two instructions have a limitation: only ports with an address smaller than 0x20 can be handled, ports above cannot be accessed that way.

For the more exotic programmer: the ports can be accessed using SRAM access commands, e.g. ST and LD. Just add 0x20 to the port's address (the first 32 addresses are the registers!) and access the port that way. Like demonstrated here:

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

That only makes sense in certain cases, but it is possible. It is the reason why the first address location of the SRAM is always 0x60.

[To the top of that page](#)

8. Details of relevant ports in the AVR

9. The following table holds the most used ports. Not all ports are listed here, some of the MEGA and AT90S4434/8535 types are skipped. If in doubt see the original reference. Ports:

Component	Link	Register	Link
Accumulator	SREG	Status Register	SREG
Stack	SPL/SPH	Stackpointer	SPL/SPH
Ext.SRAM/ Ext.Interrupt	MCUCR	MCU General Control Register	MCUCR
Ext.Int.	INT	Interrupt Mask Register	GIMSK
		Flag Register	GIFR
Timer Interrupts	Timer Int.	Timer Int Mask Register	TIMSK
		Timer Interrupt Flag Register	TIFR
Timer 0	Timer 0	Timer/Counter 0 Control Register	TCCR0
		Timer/Counter 0	TCNT0
Timer 1	Timer 1	Timer/Counter Control Register 1 A	TCCR1A
		Timer/Counter Control Register 1 B	TCCR1B
		Timer/Counter 1	TCNT1
		Output Compare Register 1 A	OCR1A
		Output Compare Register 1 B	OCR1B
		Input Capture Register	ICR1L/H
Watchdog Timer	WDT	Watchdog Timer Control Register	WDTCR
EEPROM	EEPROM	EEPROM Address Register	EEAR
		EEPROM Data Register	EEDR
		EEPROM Control Register	EECR
SPI	SPI	Serial Peripheral Control Register	SPCR
		Serial Peripheral Status Register	SPSR
		Serial Peripheral Data Register	SPDR
UART	UART	UART Data Register	UDR
		UART Status Register	USR
		UART Control Register	UCR
		UART Baud Rate Register	UBRR
Analog Comparator	ANALOG	Analog Comparator Control and Status Register	ACSR
I/O-Ports	IO-Ports		

10.

[To the top of that page](#)

11. The status register as the most used port

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

compare- or calculation-operations. The following list has all assembler commands that set or clear status bits depending on the result of the execution.

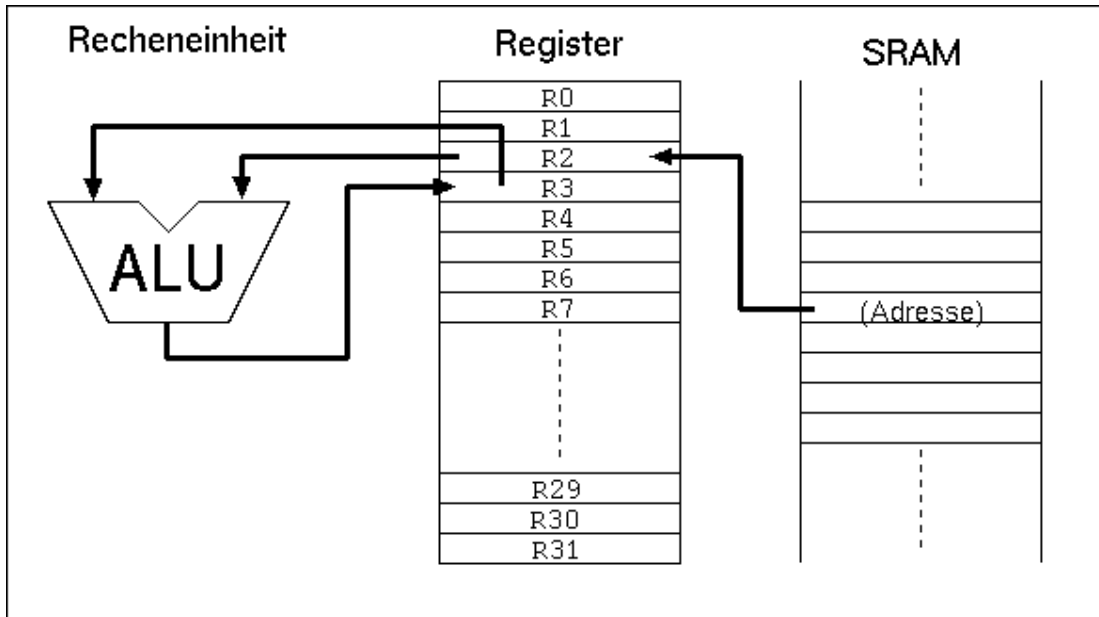
Bit	Calculation	Logic	Compare	Bits	Shift	Others
Z	ADD, ADC, ADIW, DEC, INC, SUB, SUBI, SBC, SBCI, SBIW	AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR	CP, CPC, CPI	BCLR Z, BSET Z, CLZ, SEZ, TST	ASR, LSL, LSR, ROL, ROR	CLR
C	ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW	COM, NEG	CP, CPC, CPI	BCLR C, BSET C,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

Using SRAM in AVR assembler language

All AT90S-AVR-type MCUs have static RAM (SRAM) on board. Only very simple assembler programs can avoid using this memory space by putting all info into registers. If you run out of registers you should be able to program the SRAM to utilize more space.

What is SRAM?

SRAM are memories that are not directly accessible to the central processing unit (Arithmetic and Logical Unit ALU, sometimes called accumulator) like the registers are. If you access these memory locations you usually use a register as interim storage. In the following example a value in SRAM will be copied to the register R2 (1st command), a calculation with the value in R3 is made and the result is written to R3 (command 2). After that this value is written back to the SRAM location (command 3, not shown here).



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

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

[To the top of that page](#)

For what purposes can I use SRAM?

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

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

The most relevant use for SRAM is the so-called stack. You can push values to that stack, be it the content of a register, a return address prior to calling a subroutine, or the return address prior to an hardware-triggered interrupt.

[To the top of that page](#)

How to use SRAM?

To copy a value to a memory location in SRAM you have to define the address. The SRAM addresses you can use reach from 0x0060 (hex notation) to the end of the physical SRAM on the chip (in the AT90S8515 the highest accessible internal SRAM location is 0x025F). With the command

```
STS 0x0060, R1
```

the content of register R1 is copied to the first SRAM location. With

```
LDS R1, 0x0060
```

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

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

```
.EQU MyPreferredStorageCell = 0x0060  
STS MyPreferredStorageCell, R1
```

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

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

```
.EQU MyPreferredStorageCell = 0x0060  
.DEF MyPreferredRegister = R1  
.DEF AnotherRegister = R2  
.DEF AndAnotherRegister = R3  
LDI XH, HIGH(MyPreferredStorageCell)  
LDI XL, LOW(MyPreferredStorageCell)  
LD MyPreferredRegister, X+  
LD AnotherRegister, X+  
LD AndAnotherRegister, X
```

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

The third construction is a little bit more exotic and only experienced programmers use this. Let's assume we very often in our program need to access three SRAM locations. Let's further assume that we have a

spare pointer register pair, so we can afford to use it exclusively for our purpose. If we would use the ST/LD instructions we always have to change the pointer if we access another location. Not very convenient. To avoid this, and to confuse the beginner, the access with offset was invented. During that access the register value isn't changed. The address is calculated by temporarily adding the fixed offset. In the above example the access to location 0x0062 would look like this. First, the pointer register is set to our central location 0x0060:

```
.EQU MyPreferredStorageCell = 0x0060  
.DEF MyPreferredRegister = R1  
    LDI YH, HIGH(MyPreferredStorageCell)  
    LDI YL, LOW(MyPreferredStorageCell)
```

Somewhere later in the program I'd like to access cell 0x0062:

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

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

The corresponding instruction for reading from SRAM with an offset

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

is also possible.

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

[To the top of that page](#)

Use of SRAM as stack

The most common use of SRAM is its use as stack. The stack is a tower of wooden blocks. Each additional block goes onto the top of the tower, each recall of a value removes the upmost block from the tower. This structure is called *Last-In-First-Out (LIFO)* or easier: *the last to go on top will be the first coming down*.

Defining SRAM as stack

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

To construct the stack the stack pointer is loaded with the highest available SRAM address. (In our case the tower grows downwards, towards lower addresses!).

```
.DEF MyPreferredRegister = R16
```

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

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

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

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

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

at the beginning of our assembler source code.

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

Use of the stack

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

```
PUSH MyPreferredRegister ; Throw that value
```

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

```
POP MyPreferredRegister ; Read back the value
```

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

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

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

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

```
RCALL Somewhat ; Jump to the label somewhat
```

[...] here we continue with the program.

Here the jump to the label *somewhat* somewhere in the program code,

Somewhat: ; this is the jump address

[...] Here we do something

[...] and we are finished and want to jump back to the calling location:

RET

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

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

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

Common bugs with the stack operation

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

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

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

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