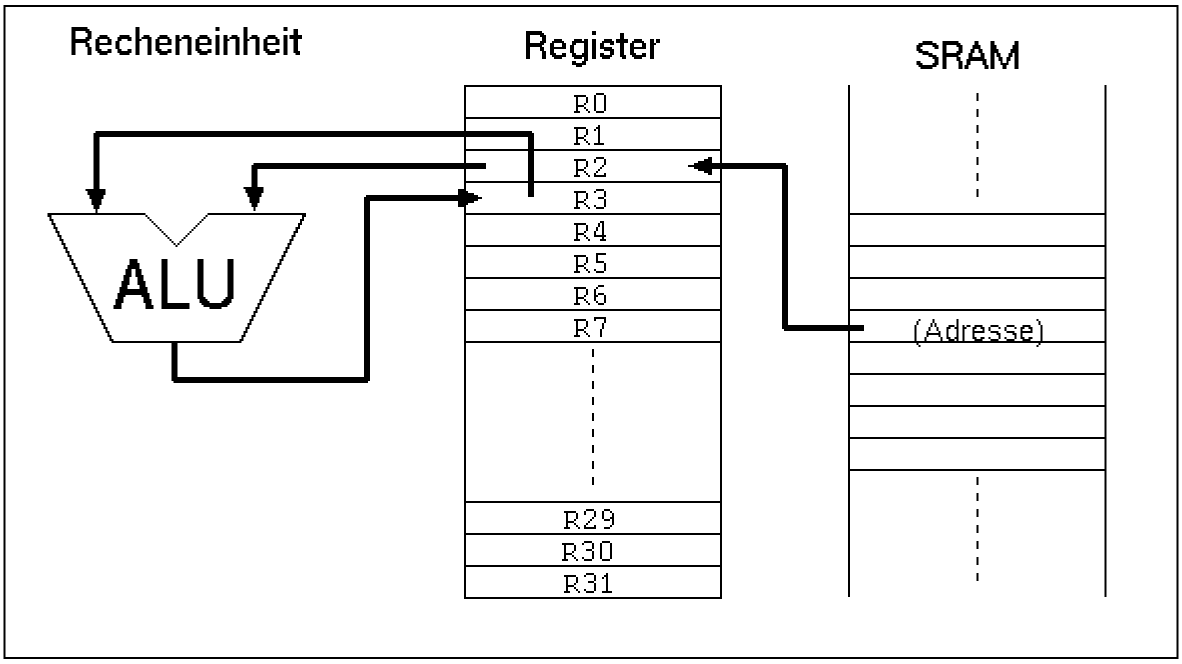
Using SRAM in AVR assembler language

All 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.

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 (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.

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.

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.

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.

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 exection 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!