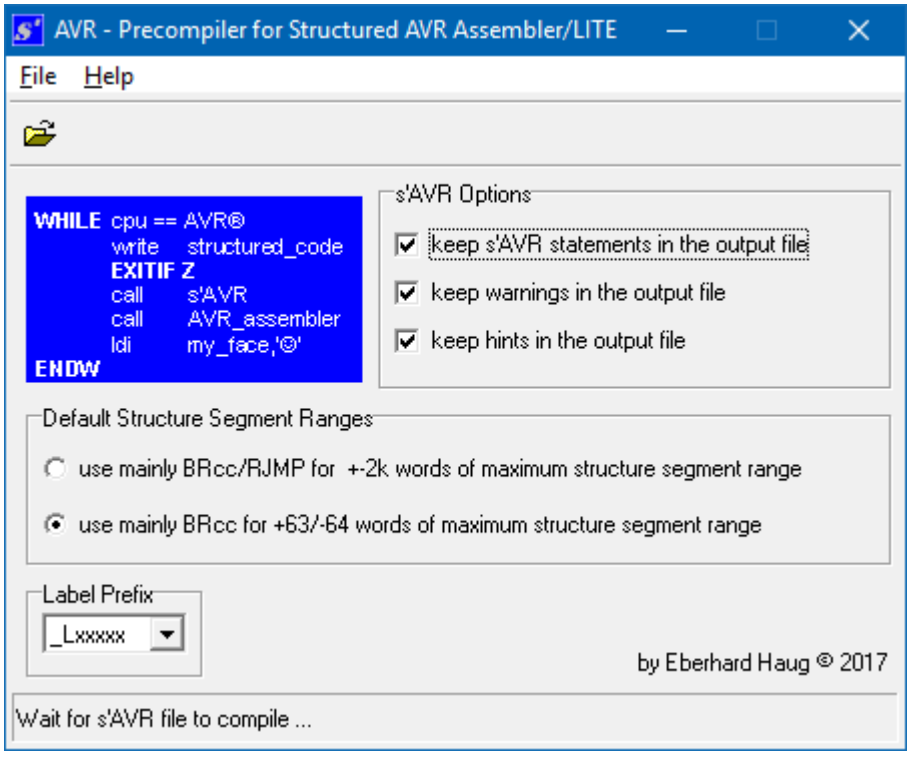
s'AVR-LITE

Structured Assembly Programming for Atmel® AVR®



Contents

Structured Programming

Firstly control structures should not be confused with conditional assembly. Control structures given by **s'AVR** und **s'AVR-Lite**[1] provide directives for structured programming offering the following benefits:

- **Structured programming in AVR assembler environment**
- **Simplified program development and debugging**
- **Extensively improved program readability and documentation**
- **Shorter program design cycles and improved productivity**
- **Improved program reliability and maintenance**
- **More fun writing AVR assembler programs.**

For speed and memory sensitive applications the alternate usage of high level languages like C normally is excluded for compact single-chip microcontrollers like the smaller Atmel AVR devices.

s'AVR is a perfect alternate solution as it combines some advantages of high level languages and all advantages of the AVR assembly language. **s'AVR** provides directives for implementation of various branching and looping tasks commonly found in programming.

Depending on selected Structure Segment Ranges (since **s'AVR** 2.x), control structures provided by **s'AVR** do affect code efficiency little[2] or not at all. That means computing power and memory utilization are equivalent to pure AVR assembler programming, or can show even better results - depending on the software engineer's skills!

Using **s'AVR** all assembly language facilities are still conserved as all instructions other than **s'AVR** directives per definition are regular AVR assembly instructions.

The gist of **s'AVR** in a nutshell:

All instructions in a **s'AVR** source code program (`*.s` or `*.savr`)
other than **s'AVR** directives are regular AVR assembly instructions!

This message understood = **s'AVR** understood!

Therefore even existing pure AVR assembler programs could be extended afterwards[3] by **s'AVR** directives to get structured AVR source code, but still keeping all advantages of the AVR assembly language.

After compiling the **s'AVR** based source code a "flat" AVR assembly source program is provided which can be used to assemble, debug and program the given AVR device using the known AVR design tools, including AVR® Studio.

[1] Within this manual s'AVR always includes s'AVR-Lite unless any restrictions are specified.
[2] Due to the AVR instruction set (missing appropriate skip instructions which refer to the status register and branch instructions which cover a wider address range).
[3] Using the given method initial tests have been done to proof s'AVR.
However, writing s'AVR based source code from scratch is much easier than converting pure AVR assembly source code to s'AVR.

When writing structured **s'AVR** programs explicit jumps could be omitted, but in certain situations, however, avoiding jumps will result in more confusion than using them.

Therefore **s'AVR** offers a unique feature even handling such a situation. On the other hand for purposes of learning structured programming, avoiding jumps will force the mind to exercise the structured mentality.

Compared to high level languages **s'AVR** directives are far less complex, thus much easier to use.

What **s'AVR** is intended to do

s'AVR (running under all current Windows® versions[4] and Linux) is a precompiler translating directives for structured programming into assembly source code for the Atmelr® AVR microcontroller family.

Amongst the **s'AVR** statements any regular AVR assembly statement sequences can be placed (typically for the actual data manipulation and subroutine calls, **s'AVR** even doesn't touch those).

Of course, for nesting structures any number of additional **s'AVR** directives (and AVR assembly instructions) can be used, as long as the program memory of the given AVR microcontroller is sufficient.

The assembly source code generated by **s'AVR** can be used for simulating and debugging (for example using Atmel Studio or any other AVR development tools). For source level debugging the original **s'AVR** statements normally stay as comments within the generated *.asm file.

Optionally, however, the comments can also be removed from the output file. It is also possible to remove **s'AVR** warnings and **s'AVR** hints from the output file.

Finally the code generated by **s'AVR** must be assembled by any Atmel 8-bit AVR compatible assembler into final AVR object code.

Important!

A very essential feature of **s'AVR** is the exclusive use of the AVR registers specified within the **s'AVR** directives.

No other registers are employed or modified by **s'AVR** (except, of course, the program counter and occasionally the status register).

The AVR stack is not used either by **s'AVR**, which means that the AVR program designer has full and exclusive control over all AVR registers and the AVR SRAM data memory. He even is forced to do so (like with every assembler program).

All **s'AVR** structures can be nested to any depth - just limited by the program memory of the AVR microcontroller being used.

[4] All versions from Windows® 98SE through Windows®10 should work. Under Linux s'AVR can be used when combined with WINE.

As s'AVR is not using the AVR stack for return addresses, any jumps could be done to any location of the program, even into other structures[5]. The AVR stack is 100% available for regular subroutine calls and push/pop instructions.

Depending on the method of writing the code, s'AVR should show similar code efficiency compared to pure AVR assembly code, but offer much better readability, better documented code, easier debugging etc. and finally more fun writing successful AVR assembler programs quicker and without hurdles.

What s'AVR doesn't do

s'AVR does not generate any AVR object code directly.
Therefore it's called a ***pre***-compiler.

As s'AVR does not know about any absolute addresses it cannot check address boundary violations among others.

In addition s'AVR cannot check whether data defined in the source code is within certain limits, for example if the ±2k word range of the `RJMP` instruction is exceeded. This also has to be checked by the assembler itself.

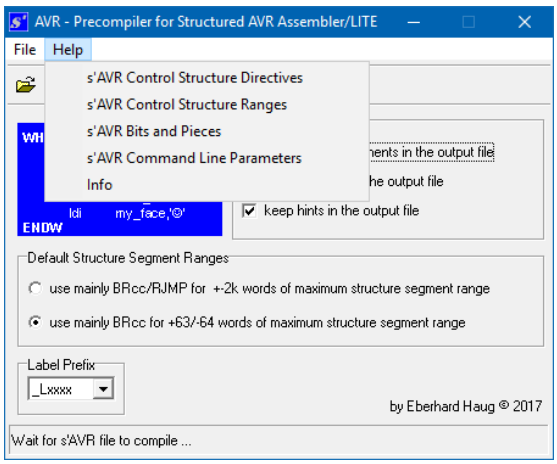
No declarations are required for s'AVR and s'AVR does not analyse any (for example those being declared for the AVR assembler).

s'AVR Installation and Uninstallation

s'AVR is a very compact stand-alone executable Windows® program which does not need any installation. The *.exe file simply is copied to any arbitrary (preferably separate) directory. If wanted, the file name can be changed.

If you like to "uninstall" s'AVR simply erase the given *.exe file.
Manuals (PDF-format) are available in English and German language.

The s'AVR Help menue gives a very compact overview of the s'AVR Control Structure Directives and Ranges, "Bits and Pieces" and Command Line Parameters (in English):



[5] However, such jumps are not recommended. Especially jumps into and out from subroutines always want to be well-considered.

How to use *s'AVR*

- Write your application source code using any preferred text editor taking care of the syntax described in both the AVR assembler manual, the present *s'AVR* documentation and the *s'AVR* Help menue. The file extension of your source code should read ".s", for example `my_source_file.s`. Optionaly ".savr" is accepted, see later in this manual.

- Start *s'AVR* like any Windows® program. After selecting any *s'AVR* compile options, now you can open your *.s file through the program menu and compile it. The generated output file (now having flat AVR assembly source code without any structures) will show the extension ".asm".

Important! Be careful: If any file in the same directory is having the same file name as the source file and its file extensioin is *.asm, it will be replaced without any warning!

Additional *.s files can be compiled one after another without leaving *s'AVR*.
A status line will ask you to wait for a short moment if your *s'AVR* file is quite big or your computer is very slow.

When finished, the *s'AVR* status line will show the number of errors/warnings, if any.

The generated AVR assembly source file (for example `my_source_file.asm`) now can be loaded by any AVR tools, assembled, programmed into AVR and debugged. During these sessions *s'AVR* needs not to be quit.

If any errors/warnings occurred during compilation they will be shown within the *.asm output file, referring to the line numbers of the original *.s source file.

Error messages in the *.asm output file are not shown as comments but as .ERROR lines (including an error text) to indicate AVRASM2 to stop the assembly (and prevent additional assembly error messages).

Since *s'AVR* version 2.1 a changeable Label Prefix[6] from `_Axxxx` through `_Zxxxx` allows to link up to 26 different *s'AVR* source files to a common AVR program without any label conflicts.

s'AVR also has a command line interface to allow calling *s'AVR* from other programs (like Atmel Studio) and passing the various compile options to *s'AVR* too.

A simple example for a command line causing *s'AVR* to compile `my_source_file.s` generating an output file `my_source_file.asm` by using the default *s'AVR* options:

s'AVR.exe /my_source_file.s

The same interface is used when *s'AVR* is called by other tools like Atmel Studio.
Then even any error messages are fed back to Atmel Studio so the affected error line(s) of `my_source_file.asm` can be reached simply by a mouse click.

[6] Label Prefix _Lxxxx is default.

However, detected errors must be corrected in the source file, not in the output file!

Details how to use the command line interface and various options see next section and at the end of this manual, please.

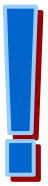
Note:

Although **s'AVR** is not case sensitive, for better readability all **s'AVR** directives/ statements within this manual are shown in upper case letters (like the AVR assembly code generated by **s'AVR**).

But AVR assembly lines being part of examples are shown in lower case letters and in `Courier` font.

Embedding *s'AVR* in Atmel® Studio

First you create your structured assembly source program `My_sAVR_Program` with the file extension `.s` (default setting for the Atmel Studio editor, so Visual Assist will be supported) either in advance using the text editor of your choice or after the project has already been set up under Studio.



Since normally an AVR assembly source file `main.asm` is created for a new Studio assembler project, you must first rename[7] it within the 'Solution Explorer' to `My_sAVR_Program.asm` using the right mouse button and then assign it to the assembler using *'Set As EntryFile'* because otherwise the *.asm file will not be reloaded automatically by the Studio Editor after compiling the *.s file by **s'AVR**!

Then you can open additional text files within Studio for structured **s'AVR** source programs, either the already existing existing source file `My_sAVR_Program.s` or any new assembly files with the file extension[8] *.s).

Now all source files (for example `My_sAVR_Program.s` written with **s'AVR** syntax and the "flat" AVR assembly source program `My_sAVR_Program.asm` compiled by **s'AVR**) could be edited side by side with the Atmel Studio editor - but only the `My_sAVR_Program.s` file should be edited (not so the *.asm file, as this will be automatically updated by **s'AVR**)!

The *.s file must be saved[9] before calling **s'AVR**, otherwise the last saved file will be compiled and you might wonder why the assembled and linked AVR code is not up-to-date[10]!

Normally `My_sAVR_Program.asm` (automatically updated after every **s'AVR** call) is only used to be assembled or (rarely) to check the "flat" assembly code generated by **s'AVR** and, if necessary, any generated error messages.

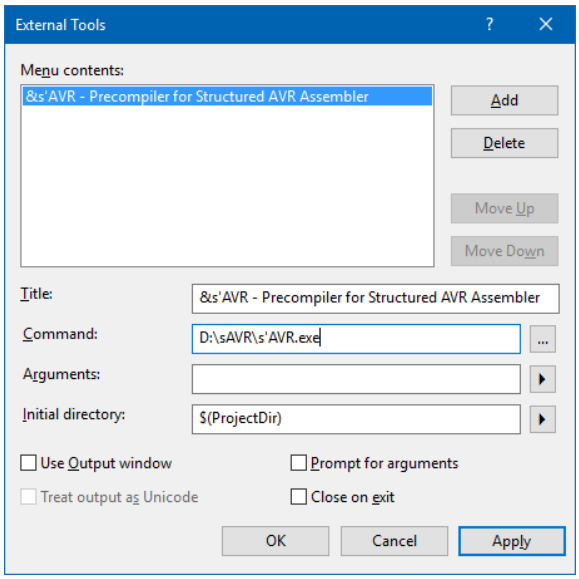
Above all, you should not make any changes in the `*.asm` file!

Finally, you will also want to embed **s'AVR** directly into the Atmel Studio.

[7] `main.asm` (only) cannot be opened and saved by another program (like s'AVR) at the same time.
[8] File extensions other than *.asm and *.s are not supported by Visual Assist.
[9] Newer versions of Atmel Studio indicate files not being saved after any changes by a '*' at the end of the file name.
[10] In case of need it could be verified checking date and time of the *.asm file.

For this purpose, under 'Tools' the submenu 'External Tools' is used to set up 'Title' and 'Command', depending on the location of the s'AVR Program.

Initially[11] no specification for 'Arguments' is required:



After Apply/OK, you can now edit/view the s'AVR source code as well as the assembly source code with the Studio Editor and directly call[12] s'AVR under 'Tools'.

After compiling (as quick as a flash!) by s'AVR normally[13] the generated assembly source code `My_sAVR_Program.asm` is automatically updated within Atmel Studio and can now be assembled from Studio directly by means of 'Build' or 'Rebuild' and (if error-free) via 'Device Programming' programmed into the program memory of the AVR microcontroller.

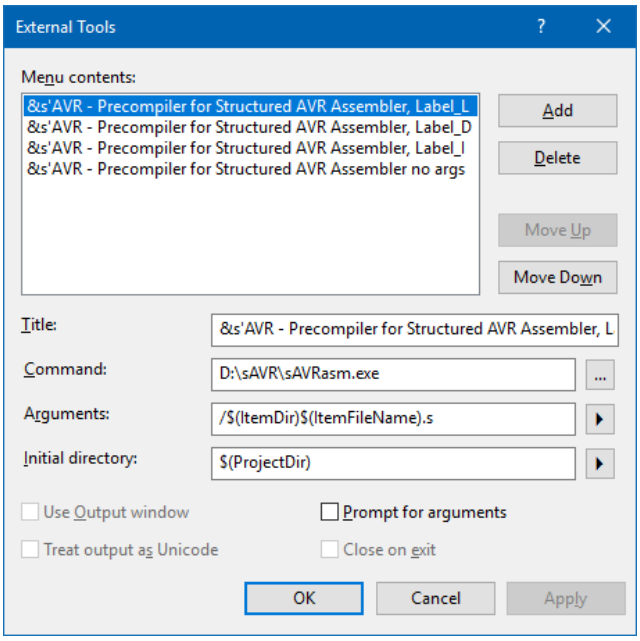
Normally you edit your source code only in the *.s window(s), click (after saving) the s'AVR window or 'External Tools' to start the pre-compiler and then go directly to 'Build'.

Again: The *.asm window is only touched for troubleshooting or simulating/debugging!

[11] If 'Arguments' are passed to s'AVR, the s'AVR windows will not open at all (only if 'Arguments' are not passed correctly).
[12] When updating Atmel Studio 7.0 software, in one case the 'External Tools' setup has disappeared, so it had to be created again.
[13] If the *.asm file in the 'Solution Explorer' is marked by 'Set As Entry File'.

Even more elegant

After some experience dealing with **s'AVR** finally you will also specify 'Arguments' under 'External Tools', possibly even several arguments for different label prefixes[14], so you do not need the **s'AVR** window anymore (but then no help menu is available anymore) and a few extra mouse clicks can be omitted:



Different label prefixes are required if the AVR program consists of several individual **s'AVR** source programs, which are compiled separately and are then linked by .INCLUDE by the AVR assembler.

Again the important note:

s'AVR overwrites a file with the same source file name and the *.asm extension in the same directory (possibly from a previous compilation, but hopefully not a self-written assembly source program) without any warnings!

Only that way unhindered working with **s'AVR** is possible.

[14] Label Prefixes _Axxxx though _Zxxxx are supported since s'AVR version 2.1.

Control Structure Directives implemented in *s'AVR*

```
IF        condition      [THEN]  ; test for first branch
          StatementSequence
ELSEIF    condition      [THEN]  ; additional branch test(s), multiple and optional
          StatementSequence
ELSE                             ; last branch, optional
          StatementSequence
ENDI                             ; end of IF structure
-------------------------------------------------------------------------------------------------------
LOOP                             ; start of endless loop
          StatementSequence      ; exit LOOP by EXIT, EXITIF, assembly jump
                                 ; or branch, RET, interrupt or AVR reset only
ENDL                             ; end of LOOP structure
-------------------------------------------------------------------------------------------------------
WHILE     condition              ; test at start of the loop
          StatementSequence
ENDW                             ; end of WHILE loop
-------------------------------------------------------------------------------------------------------
REPEAT                           ; start of the REPEAT loop
          StatementSequence
UNTIL     condition              ; test at end of the loop
-------------------------------------------------------------------------------------------------------
FOR       RegisterAssign         ; FOR loop with initialized loop counter
          StatementSequence
ENDF                             ; decrement and test for loop counter = zero
-------------------------------------------------------------------------------------------------------
EXIT                             ; leave one structure level unconditionally

EXITIF    condition              ; conditional exit from actual structure (one level)

EXITIF    condition  TO  label    ; conditional jump out of a structure
```

Notes:

- *StatementSequence* stands for any number of AVR assembly instructions and/or ***s'AVR*** statements, therefore ***s'AVR*** statements can be nested without limits (as long as enough memory is available).
- *condition* stands for compare tests (al least one AVR register, register/port bit tests or status register flag tests).
- *RegisterAssign* assigns the number of loop cycles. The specified register can be loaded already or the assignment is part of the statement. The FOR loop register can be loaded with an immediate literal or the contents of another AVR register.
- EXIT and EXITIF are exits to just one lower structure level.
- EXITIF–TO is an exception allowing conditional jumps even into foreign structures.
  Unconditional jumps simply can be done using the assembly instruction RJMP or JMP (not for older ATtiny).
- *label* is any valid AVR address (not local) somewhere in the ***s'AVR*** program (take care when leaving subroutines).
- ***s'AVR*** is using the specified AVR registers only (in most cases the status register too).
- ***s'AVR*** directives are not case sensitive.
  Therefore the directives can be written in lower or upper case (or even both).
- Since ***s'AVR*** 2.0 certain directives allow range extensions .m and .s.

Description of Control Structure Directives

General syntax rules

For simplicity the ***s'AVR*** syntax does not use the 'full-size' structures like IF–THEN–ELSEIF–THEN–ELSE–ENDIF, WHILE–DO–ENDWHILE statements etc.

THEN and DO are not needed (THEN is optional, DO is not allowed) and all ENDs are using just the first character of the initiator only like IF–END**I**, WHILE–END**W**, LOOP–END**L**, FOR–END**F**.

REPEAT ends differently, using UNTIL for the final test.

All ***s'AVR*** structures can be left by EXIT and EXITIF by just one structure level.

EXITIF got an optional TO for jumping out of a structure conditionally.

As mentioned in the "Notes" above, ***s'AVR*** statements may be written in upper or lower case (even mixed). In this manual, ***s'AVR*** statements and assembly instructions generated by ***s'AVR*** are always written in capital letters to distinguish assembly instructions being part of the ***s'AVR*** source program (which are always written in lowercase letters and `Courier font`).

For some very basic syntax rules please also read the notes at the end of this document.

It was the basic intention to allow writing structured ***s'AVR*** source code quite simply and clearly to get reliable AVR assembly code without headache!

Adressing Modes

Since the AVR microcontrollers allow operations with registers, register bits, port bits and constants resulting in different assembly instructions, it must be possible to differentiate between those by the ***s'AVR*** statements.

Without special marking of the operands, registers are assumed or the AVR assembler must check for correctness.

Register bits are specified in the form ***Register,BitPosition*** and constants ("immediate") are always preceded by a **#** character, even if they are symbols!

Complete port bytes must first be loaded into an AVR register (which is an AVR property) by means of an assembly instruction (IN), before structured statements can be executed.

Only port bits can be checked directly using the form ***%Port,BitPosition***.

The **%** character allows ***s'AVR*** to distinguish between AVR registers and AVR I/O ports.

No spaces are allowed after **#** and **%**, just optionally before ***BitPosition***.
Examples will follow.

Control Structure Range by Extensions .m and .s

Originally, s'AVR (in particular all versions 1.x) strictly generated "no pain" AVR code mainly using BRcc/RJMP instructions, which did not always result in best code efficiency, but because of ±2k words of address range normally did not require any intervention by hand within the s'AVR structures.

A more efficient AVR code would predominantly use BRcc instructions (not combined with RJMP). However, for most efficient code the maximum address range is only +64/-63 instruction words, which is not sufficient for larger program structures.

As of s'AVR 2.0, the s'AVR source program can optionally be compiled with the previous "no pain" or the new "efficient" option. However, in order to avoid problems with a too small address range, especially with the "efficient" option, the s'AVR directives can be supplemented with range extension .m (for "medium") to force BRcc being combined with RJMP in order to get a wider address range of ±2k words compared to the "efficient" code, thus avoiding error messages by the assembler.

Conversely, the range extension .s (for "small") for the "no pain" compiler option (namely, for source programs originally written for s'AVR 1.x) allows the enforcement of short jumps per BRcc and skip instructions.

For the following s'AVR statements, the range extensions .m and .s are accepted:

IF, ELSEIF, EXITIF, REPEAT, WHILE and **ENDF**.

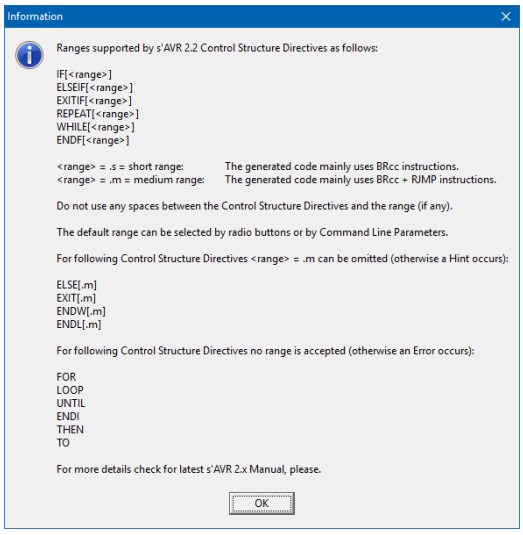
For these statements, a range extension .m is allowed (with a "hint" in the output file), but is redundant[15]:

ELSE, EXIT, ENDW and **ENDL.**

And for these statements, no range extension is permitted (would result in an error message), since elsewhere in the structure the range of the jump will be determined:

FOR, LOOP, UNTIL, ENDI, THEN and **TO.**

The s'AVR Help menue is showing a summary regarding the structure ranges:



[15] Since anyway AVR code is generated for medium range by BRcc/RJMP or even just by RJMP.

Simple and therefore not 100% consistent

Actually, the range extensions are normally not part of the IF, ELSEIF, EXITIF and WHILE statements, where you would normally expect size extensions (for example IF.w for a 16-bit comparison) and the range extension instead should be attached to THEN, DO and TO.

Due to the AVR instruction set (unlike the instruction set of 68000 and other µP/µC), "efficient" AVR code is very different compared to "no pain" AVR code.

Unfortunately, a simple 1-pass compiler (such as **s'AVR**) must know the wanted range while generating the appropriate code, so a range attached to identifiers THEN, DO and TO would already be too late.

Therefore, in **s'AVR**, for the sake of simplicity the range extension has been defined to be placed at the specified practical places instead of the logically correct ones.

Skip and Jump Instructions

Unfortunately, AVR (compared to other microcontrollers) offers only a few skip instructions which directly relate to the status register. This makes a universal procedure much more complicated for the (pre-) compiler, since instead of simple skip instructions, branch instructions need to be used unnecessarily in many places to skip a subsequent RJMP instruction, because it was the goal that the assembly source code generated by **s'AVR** can always be assembled directly by the AVR assembler without any intervention by the program designer.

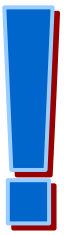
The only restriction for the LITE version is due to the maximum range of RJMP, which is ±2k word addresses. That is, the **s'AVR** structures in the LITE version are allowed to sweep a maximum range of 2k word addresses, which should normally be sufficient.

Predefined Status Bits = all Bits of the AVR Status Register[16]

```
Z       ; Zero bit set
NZ      ; Zero bit clear

C       ; Carry bit set
NC      ; Carry bit clear

H       ; Half/Digit Carry set
NH      ; Half/Digit Carry clear

S       ; Sign bit set, S = N ⊕ V (Negative EXOR Overflow)
NS      ; Sign bit clear

V       ; V bit set (2's Complement Overflow)
NV      ; V bit clear

N       ; N bit set (Negative)
NN      ; N bit clear

T       ; Transfer bit set
NT      ; Transfer bit clear

I       ; Interrupt activated
NI      ; Interrupt deactivated
```

Negations

```
NOT     ; logical negation
!       ; logical negation

~       ; bitwise negation, 1's complement (same as assembly instruction COM)
```



Important notes regarding status bits:

For ease of use those status bits are – like every ***s'AVR*** directive – reserved ***s'AVR*** keywords and should not be used within the AVR assembler program as identifiers (labels, registers and constants).

Instead of NZ, NC, etc., equivalent NOT Z, NOT C or even !Z, !C can be used.

Although AVRASM2 itself also defines Z, C, H, S, V, N and T, it does not bother in this context.

[16] T / NT and I / NI are supported since s'AVR 1.1.

EXIT and EXITIF, common for leaving s'AVR Structures
Description of Conditions and Relations

Syntax:

EXIT				; exit from actual structure unconditionally
EXITIF	*condition*			; exit from actual structure conditionally
EXITIF	*condition*	**TO**	*label*	; jump out of a structure conditionally

Both the unconditional EXIT and the conditional EXITIF can be used to leave the actual s'AVR structure stepping one (and only one) structure level back.

If you need to leave more structure levels (up or down) for certain reasons just use a regular assembly jump (RJMP or JMP [not for older ATtiny]), which is not strict structured programming but in certain situations jumping would offer more program clarity.

But be careful when jumping out of assembly subroutines[17] (which would be unstructured): Your AVR stack might get muddled up!

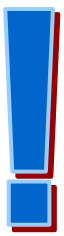
EXITIF allows testing for a certain condition, which could be a certain content of a register or a certain bit status of a register or a port. Optionally EXITIF allows jumping out of s'AVR structures using EXITIF–TO, which is also quite handy, as we will see.

Special status bits can be used directly like EXITIF Z, EXITIF NZ, and so on, as described above under "Predefined Status Bits".

Unfortunately, since the AVR status register is in the I/O range >31 (see next section), its bits can not be used in the form[18] `%SREG,StatusBit`, which would make the generated AVR code simpler and more efficient in many cases.

Register Bits and Port Bits

Generally spoken, for bit queries ***Register,BitPosition*** or ***%Port,BitPosition*** the syntax reads as follows:



Register, BitPosition	Both ***Register*** and ***BitPosition*** may be symbols, ***BitPosition*** may also be a decimal value 0-7.
%Port, BitPosition	Both ***%Port*** and ***BitPosition*** may be symbols, ***BitPosition*** may also be a decimal value 0-7.

[17] Certainly complete s'AVR structures (even nested ones) can be used within assembly subroutines.
[18] Even though the s'AVR syntax would accept it, not so does the AVR assembler.

Note:

A single symbol cannot be used to combine *Register/Port* and *BitPosition* as the ***s'AVR*** syntax for the status flag test requires the comma between *Register/Port* and *BitPosition* (exceptions are only the predefined status flags like Z, NZ etc.).

The reason is, that the generated code must be different for registers, constants, register bits and port bits.

According to AVRASM2 syntax, for ***Register*** or ***%Port*** only the AVR registers or ports 0-31 are allowed (after assembly) and for ***BitPosition*** only the values 0-7, which is partially checked by ***s'AVR*** as far as possible. AVRASM2 checks for other permissions as usual.

In order to check contents or bits of ports with a port address >31, the wanted port must first be copied into one of the AVR registers 0-31.

This is also true (unfortunately) for the status register SREG (see above), if you do not want to use the predefined status bits!

Attention: Behind the % sign and in front of the comma, there must be no spaces[19] in order to obtain a clear assignment of the individual symbols!

On the other hand, ***s'AVR*** leaves enough freedom regarding spaces and TABs.

Examples for checking the status:

EXITIF Reg, 3 ; EXIT ***s'AVR*** structure if bit 3 of register Reg is set

The **NOT** directive or simply the **!** sign allow to check register and port bits which are not set (clear):

EXITIF !r12, 5 ; EXIT if bit 5 of register 12 is clear
EXITIF NOT r12,5 ; same as EXITIF !r12,5
EXITIF !%portd, strobe ; EXIT if bit 'strobe' of port D is clear

EXITIF Z ; EXIT if Z bit is set
EXITIF NC ; EXIT if C bit is clear
EXITIF ! C ; same as EXITIF NC
EXITIF NOT C ; same as EXITIF NC

Multiple usage of NOT and ! are allowed and will be handled correctly:

EXITIF ! NOT !NC ; same as EXITIF C

Attention: For a negated logical test do not use the bitwise negation '~' !

Of course, it is allowed - and sometimes useful - to use one and the same EXITIF condition multiple times in multiple nested structures in order to avoid to leave these structures using an unstructured jump (`RJMP`, `JMP` or EXITIF-TO) over several levels.

[19] Since s'AVR 2.23 spaces are accepted after the comma.

Originally there was the idea of offering EXITs over more than one structure level, but the implementation would have been much more complex and using a regular assembly jump (`RJMP` or `JMP`) is doing almost the same job much more easily and even more clearly. But to get optimized code and to prevent any assembly headache, the unique code saving **EXITIF–TO directive** was created, which is equal to a conditional jump.

An example for a jump table using EXITIF–TO see description of the LOOP directive.

Now there is no longer a need to do any assembly comparisons and you no longer need to worry about these!

But: EXITIF–TO (same for EXIT and EXITIF) must be placed within **s'AVR** structures (LOOP–ENDL, e.g.) otherwise this statement would not be recognized by **s'AVR** and an unbalanced structure error would occur during compilation.

Releations

More general are those conditions, where a **relation** between two items is tested (unsigned 8-bit tests!):

Syntax (example using EXITIF):

EXITIF *a* ***Relation*** *b* ; EXIT **s'AVR** structure by one level if '*a Relation b*' is true

Relations can also be part of IF, ELSEIF, WHILE and UNTIL statements.

Relation is any of the following characters or character strings:

==	**equal**
<>	**not equal (>< is not allowed and results in an error)**
<	**less than**
<=	**less or equal**
>	**greater than**
>=	**greater or equal**

Both ***a*** and ***b*** can be any AVR **register** described by an identifier or a register number (R17, e.g.). ***b*** in addition also can be an immediate **literal** using the **#** delimiter.

Then, however, only the AVR registers 16-31 are supported for comparisons with a literal (an AVR property). Otherwise two AVR registers must be compared instead.

Attention using relations:

- Errors concerning invalid AVR registers are partially detected by ***s'AVR***, but at the latest by the AVR assembler during assembly.
- Due to the 8-bit unsigned comparisons, a comparison to >0xff can not be performed.
- Comparing for 'greater than' with the constant #0xff results in an assembler error.
- Equally (meaningfully) a comparison to <0 is not possible (both with a register with content 0 and with a constant #0).
- For the mentioned error cases, the results of such relations are always (correctly) false, independent of the value of the operand ***a*** during runtime.
- The assembly code generated by ***s'AVR*** shows some subtleties of the different comparisons (looking at them once in a while is worthwhile).
- To compare two signed values (<127) with each other via ***s'AVR*** condition, you can increase both values by an assembly instruction by an offset of 128 or subtract -128 before the comparison[20].

IF – [THEN] – ELSEIF – [THEN] – ELSE – ENDI, Test for alternate Branches

This more complex statement is the very common way of finding decisions for certain branches.

Syntax:

```
IF      condition1      [THEN]   ; initial test, branch if false
        StatementSequence
ELSEIF  condition2      [THEN]   ; optional 2nd test, branch if false
        StatementSequence
ELSEIF  conditionN      [THEN]   ; optional more tests
        StatementSequence
ELSE                             ; optional last (default) branch without test
        StatementSequence        ; no more other ELSEIF can follow now
ENDI                             ; end of IF structure
```

IF and the optional **ELSEIF** (which can be used as often as needed) are followed by the condition to be checked. **ELSE** is the last (or default) branch without testing for any condition. No additional ELSEIF is allowed after ELSE.

Condition is exactly the same as described at **EXITIF**, thus a status bit or a relation.

The **EXIT** and **EXITIF** statements could be used to leave the IF structure.

EXITIF–TO would be the non-structured method leaving an IF structure conditionally to any defined destination within the ***s'AVR*** program.

[20] For the registers R16..R31 instruction `SUBI Reg,-128` can be used. An `ADDI` instruction does not exist with AVR.

In some cases the **EXITIF–TO** method can contribute greatly to the clarity of the program, but should be used with caution, especially if it involves a "direct" EXIT from an assembly subroutine.

THEN can be used optionally after IF and ELSEIF (without any effect, it generates no extra code).

Examples:

```
IF C                        ; branch to ELSEIF if carry bit is clear
        StatementSequence   ; any assembly and/or s'AVR statements
ELSEIF %pinb,3              ; else if port B, bit 3 is clear branch to ELSE
        StatementSequence   ; more assembly and/or s'AVR statements
ELSE
        StatementSequence   ; these are the statements the program
                            ; is executing if no other condition is true
        EXITIF Z            ; exit the IF structure if Z bit is set
        StatementSequence   ; any assembly and/or s'AVR statements
ENDI                        ; end of IF structure reached
```

As the current version of **s'AVR** at the moment does not support a SWITCH-CASE directive, the IF directive can be used instead, offering even more flexibility as every case can test different conditions:

```
IF case1
                StatementsCase1
ELSEIF case2
                StatementsCase2
ELSEIF caseN
                StatementsCaseN
ELSE
                StatementsCaseDefault
ENDI
```

case1, *case2* and *caseN* must be any conditions/relations as described above.

Of course, here too in the various *StatementsCaseX* additional **s'AVR** statements including EXIT and EXITIF can be included.

Because of completely new tests for every case some additional AVR code is needed for the IF based tests compared to a true SWITCH–CASE directive.
However, in the IF structure, the conditions are independent of each other.

Another solution would be using successive EXITIF–TO statements, as will be shown in the next chapter.

LOOP – ENDL, endless Loop

This is the simplest structure repeating the loop until any break occurs. Any EXIT or EXITIF directive could do a break. Or an assembly jump or branch, an AVR interrupt or a reset would break the loop. In any other situation the loop would never end.

LOOP-ENDL often encloses the main program of a microcontroller application (after the usual initialization routines).

Then for **s'AVR-LITE** the main program can only be a maximum of 2k words[21].

The workaround may be to create a main loop without LOOP-ENDL, but via loop label and JMP command (if supported by the AVR µC being used). However, this should be rare, since called subroutines are always outside of such a main LOOP-ENDL and therefore not adversely affect the jump range.

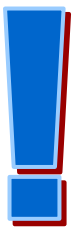
Syntax:

```
LOOP                            ; start of endless loop
        StatementSequence       ; exit LOOP by EXIT, EXITIF, JMP, BRcc, RET,
                                ; RETI, an AVR interrupt or an AVR reset only
ENDL                            ; end of LOOP
```

Examples:

```
LOOP
        StatementSequence
        EXITIF C                ; exit 1st LOOP if carry bit set
        StatementSequence
        LOOP                    ; nested loop
                StatementSequence
                EXITIF !r16,1   ; exit 2nd LOOP if register 16 bit 1 is clear
                StatementSequence
        ENDL                    ; end of 2nd LOOP
        StatementSequence
ENDL                            ; end of 1st LOOP
-------------------------------------------------------------------------------------------
LOOP                            ; the smallest possible s'AVR program!
ENDL                            ; wait for interrupt or reset
```

There is a little pitfall using **EXIT**. Don't write such s'AVR code as shown in the example below[22], as this IF structure wouldn't show any effect at all:



```
LOOP
        StatementSequence
        IF Z EXIT               ; the EXIT will leave the IF structure only,
        ENDI                    ; not the LOOP structure!!
        StatementSequence
ENDL                            ; will loop forever if no other break happens
```

[21] Limited due to the maximum possible RJMP range of ±2k words.
[22] Even the syntax for this very compact code is perfect.

To solve this situation simply use EXITIF:

```
LOOP
        StatementSequence
        EXITIF Z                ; now LOOP will be left if the Z bit is set
        StatementSequence
ENDL
```

Jump List

The usage of EXITIF-TO within a LOOP structure is shown here as a good example of a jump list[23] saving AVR code:

```
LOOP                            ; any structure needed to use EXITIF-TO
                                ; no extra AVR code generated at this point
      EXITIF  cond1   TO  label1
      EXITIF  cond2   TO  label2
      EXITIF  condN   TO  labelN
      jmp     DefaultLabel      ; assembly jump to DefaultLabel
ENDL                            ; never looped back in this example
```

Note: The jump destination addresses *label1/2/N* and *DefaultLabel* will be placed somewhere in the **s'AVR** source code, but they should by no means follow any **s'AVR** statements in the same line, otherwise these addresses/instructions will not be recognized and **s'AVR** will claim unbalanced structures.

WHILE – ENDW, Test at the Beginning of the Loop

Before the loop will start a test for a certain condition is done. If the loop is passed once it starts at the beginning of the loop doing the same test again. If the condition is false the WHILE loop is completely skipped.

Syntax:

```
WHILE condition                 ; test at start of WHILE loop
      StatementSequence
ENDW                            ; end of WHILE loop
```

Example:

```
      in    r17,portd       ; copy port D to register r17
WHILE r17 <> #0             ; as long as r17 <> 0
      out   portb,r17       ; copy register r17 to port B
      StatementSequence     ; other assembly or s'AVR statements
      in    r17,portd       ; copy port D to register r17
ENDW
```

[23] In contrast to an even more compact jump table, where the conditions are successive integers.

If you analyze the generated AVR assembly code, you will see that **s'AVR** is checking the **s'AVR** source code for literals being a constant with the value of zero[24] and finally optimized code using the TST instruction will be generated of the test is for '==' or '<>' (which works for all AVR registers, including registers 0 through 15):

```
        in      r17,portd       ; copy port D to register r17
        ;01// WHILE r17 <> #0   ; as long as r17 <> 0
_L1:
        TST     r17
        BRNE    _L2
        RJMP    _L3
_L2:
        out     portb,r17       ; copy register r17 to port B
        StatementSequence       ; other assembly or s'AVR statements
        in      r17,portd       ; copy port D to register r17
        ;01// ENDW
        RJMP    _L1
_L3:
```

Remarks:

- In this example, the **s'AVR** options "no pain" and "keep **s'AVR** statements in the output file" are used: After the first ";" (comment character), first the current structure level "01" is listed, followed by "//" (something distinctive) and the original **s'AVR** statement including any original comment of the **s'AVR** source line.
- Displaying the structure level by using "; 01//" etc. makes both debugging and trouble-shooting a little easier, especially when nesting several similar **s'AVR** statements (each with its individual structure level).

Any Objections?

Looking closer at the generated "no pain" assembly code, one could argue that the `RJMP` instruction can be completely eliminated at this point, as instead of:

```
        BRNE    _L2
        RJMP    _L3
```

one could smarter program as follows (of course AVR assembly code automatically generated by the precompiler):

```
        BREQ    _L3
```

But this would have the serious disadvantage that in a slightly more complex program, the jump range for BREQ to the address _L3 is greater than the maximum allowable range of the AVR branch instructions of only 63 words.

Therefore, **s'AVR** basically uses branch instructions and (if possible) skip instructions in the "no pain" compiling option to skip the RJMP command, which is ultimately responsible for the "long" jump (in the example to the end of the above WHILE loop).

Thus, the very small range of the AVR branch instructions in the generated "no pain" assembly code never becomes a problem, rarely the RJMP limit, namely, if a **s'AVR** structure covers an address range of more than ±2k words.

[24] Only immediate literals like #0, #$0, #$00, #0x0, #0x00, #0b0, #0b00, #0b000 und #0b0000 are checked.

This compromise on RJMP instead of one of the more efficient BRcc instructions is not really optimal, but it's frustration-free, just "no pain".

The same applies to other s'AVR structures.

Voilà - always efficient Code from s'AVR Version 2!

Even if the benefit in program memory and execution speed may not be that extreme, s'AVR 2.0 now gives the user the choice between "no pain" and "efficient" code, either default for the entire s'AVR program or even individually per structure as compiled here for the above WHILE example by using structure range .s:

```
        in      r17,portd       ; copy port D to register r17
        ;01// WHILE.s r17 <> #0 ; as long as r17 <> 0
_L1:
        TST     r17
        BREQ    _L3
        out     portb,r17       ; copy register r17 to port B
        StatementSequence       ; other assembly or s'AVR statements
        in      r17,portd       ; copy port D to register r17
        ;01// ENDW
        RJMP    _L1
_L3:
```

Conversely, it is possible to enforce the "no pain" code for individual s'AVR structures by structure range .m, provided that "efficient" is selected as the default range option.

REPEAT – UNTIL, test at the End of the Loop

In this case the loop is started and a certain condition is tested for when reaching the end of the loop. If the condition is false the loop is started again until the condition is finally met.

Syntax:

```
REPEAT                          ; start of REPEAT loop
        StatementSequence
UNTIL   condition               ; test at end of the loop
```

Example (here compiled using label prefix '_Mxxxx' and option 'no pain'):

```
REPEAT                          ; loop ...
        rcall get_character     ; ... to read characters to register 'char'
UNTIL char <> #blank            ; but skip all blanks
```

Generated code:

```
        ;01// REPEAT                ; loop ...
_M1:
        rcall get_character         ; ... to read characters to register 'char'
        ;01// UNTIL char <> #blank  ; but skip all blanks
        CPI    char,blank
        BRNE   _M2
        RJMP   _M1
_M2:
```

Hint!

An interesting Programming Hint:

Compared to WHILE–ENDW the REPEAT–UNTIL structure saves code (typically one AVR instruction) if no other instructions (s'AVR or AVR assembly) are within the given structure, for example in a simple I/O wait loop:

```
WHILE %pinb,3                   ; wait as long as port B bit 3 is set
ENDW

REPEAT                          ; wait ...
UNTIL NOT %pinb,3               ; ... until port B bit 3 is clear
```

And then while typing the s'AVR program you have one more time to think about what should be tested for behind UNTIL ... ☺.

Generated AVR assembly code:

```
        ;01// WHILE %pinb,3         ; wait as long as port B bit 3 is set
_L1:
        SBIS   pinb,3
        RJMP   _L3
_L2:
        ;01// ENDW
        RJMP   _L1
_L3:

        ;01// REPEAT                ; wait ...
_L4:
        ;01// UNTIL NOT %pinb,3     ; ... until port B bit 3 is clear
        SBIC   pinb,3
        RJMP   _L4
_L5:
```

Incidentally, with these two structures, there is no difference in the generated code between "no pain" and "efficient". This is - depending on the query - not always true.

After we got the conditional branching, the loops testing at the beginning and at the end of the loop and even looping forever, we also would like to have a loop cycled for a given number of cycles supplied before starting the loop...

FOR – ENDF, Loop with initialized Loop Counter and Decrement Steps of 1

Syntax:

FOR	*RegisterAssign* *StatementSequence*	; loop counter can be any regular AVR register
ENDF		; decrement and test loop counter for zero

RegisterAssign can be done for the loop counter as follows:

Register1 := Register2	; copy another AVR register (including a port)
Register := #Literal	; load a constant into the loop counter
Register	, the loop counter is initialized already

Register can be any of the AVR registers r0 though r31 or a symbol.

The identifier ***Register1*** can be any of the AVR registers r0 though r31 or a symbol, which means that the loop counter can be initialized with another register (*Register2*), a constant (*#Literal*, no space after '#'), or even can has been initialized already (no assignment ':=' then) before the FOR statement is reached.

For loading with a constant, however, for ***Register*** only the AVR registers R16..R31 are supported (an AVR property). In all other cases all the AVR registers R0 though R31 are accepted.

If the FOR step size should be different from –1 then simply a REPEAT–UNTIL structure could be used instead having the loop register initialized by any assembly code and doing the stepping (increment or decrement) in assembly code too just before the UNTIL statement.

Remarks:

If the loop counter is zero at the start of the FOR loop, the loop is looped 256 times due to the 8-bit loop register, because the zero count is only checked after decrementing at the end of the loop being looped through.

Instead of assigning #0, ***s'AVR*** also accepts #256 as the (only) non-8-bit constant translated[25] by ***s'AVR*** in #0 to initialize the loop counter.

s'AVR uses the double equal sign "==" for comparisons (like AVRASM2) and ":=" to initialize FOR loop registers.

[25] AVRASM2 would reject a constant 256 at this point. Other assemblers partially accept values modulo 256.

Programming Tip:

To also use the less flexible registers R0 to R15 for FOR loops with constants, one can load frequently used constants at the beginning of the program into any other AVR register (R0 to R31) and then assign per:

FOR RegisterLower := RegisterHigher

Other examples:

```
FOR     count := #3             ; value 3 assigned to AVR register 'count'
        rcall blink_led         ; the LED subroutine will be called 3 times
ENDF
```

```
FOR     loop := #loops          ; this assignment will result in several errors,
                                ; as 'loop' is a reserved s'AVR keyword!
        StatementSequence
ENDF
```

```
FOR     lp_count                ; register 'lp_count' is initialized already
                                ; before the FOR structure is reached
        StatementSequence
ENDF
```

The loop number zero can be very helpful (but sometimes also a trap):

```
FOR     lp_cnt := #0            ; loop count isn't zero now, it's 256 ...
        StatementSequence
ENDF                            ; ... because lp_cnt is tested at the end of
                                ; the loop after decrementing lp_cnt
```

As noted earlier, for better understanding and with exactly the same result, you could have taken **lp_cnt: = #256** just as well (don't foget the # sign!):

```
FOR     lp_cnt := #256          ; the number of loops is actually 256 ...
        StatementSequence
ENDF                            ; ..., because s'AVR initializes lp_cnt to 0
```

Generated "efficient" code for both cases (no original **s'AVR** statements and comments shown):

```
        CLR     lp_cnt
_L1:
        StatementSequence
        DEC     lp_cnt
        BRNE    _L1
```

If the current value of lp_cnt is used within *StatementSequence*, just remember that the first value is 0, followed by 255, 254, 253 etc.

s'AVR Options

Option "keep *s'AVR* statements in the output file"

Keeping the *s'AVR* statements in the generated output file is the default option. During debugging by Atmel® Studio or for documentation this is quite a helpful feature. If - for any reasons – the pure original assembly lines and the *s'AVR* generated assembly code are sufficient, the "keep..." option should be switched off. Correspondingly fewer lines are generated in the output file. However, then also the comments are missing that are included in the original *s'AVR* lines.

Option "keep warnings in the output file"

This default option keeps warnings of *s'AVR* in the output file.

Option "keep hints in the output file"

This default option keeps hints of *s'AVR* in the output file.

Options for jump range = "Default Structure Segment Ranges"

- *s'AVR* 1.x always generated "no pain" code.
- As of *s'AVR* 2.0, either "efficient" (default) or "no pain" code can be generated (via GUI or via command line, see below).

  In both cases, short or medium jump range (.s or .m) can be enforced individually for each structure (if the program permits this depending on the segment size).
- The range extensions are, of course, not backwards compatible with *s'AVR* 1.x.

Option for Label Prefix

From *s'AVR* 2.1 it is possible to specify which labels are generated by *s'AVR*, namely `_Axxxx` through `_Zxxxx`, which is helpful if different program modules are written as separate *s'AVR* source programs, then compiled individually and assembled together by `.INCLUDE` the AVR assembler.

Collision with conditional and other Assembler Directives

Another (minor) problem is using *s'AVR* code together with various AVR assemblers having different syntax rules. At the moment *s'AVR* is supporting the AVRASM2 from Atmel® and compatible ones.

If assembler directives or conditional assembly are not like the usual .IF, .ELSE, #IF, #ELSE, etc., those might collide with *s'AVR* structured programming directives (which should not be the case with AVRASM2).

To prevent collision with such directives (or any other assembly statements) simply a **question mark** has to be inserted in front of the affected directive.

s'AVR will check for those leading question marks and remove them without taking care about the rest of the line itself. Afterwards the assembler can read the directive as expected. This is a simple trick[26], but very effective ☺.

Example for conditional assembler directives within the **s'AVR** source code, which could collide with **s'AVR** statements without having the questions marks:

```
?IF debug=1                    ; '?' automatically will be removed by s'AVR
        StatementSequence
?ELSE                          ; '?' automatically will be removed by s'AVR
        StatementSequence
ENDIF                          ; this line would not matter, as s'AVR uses ENDI,
                               ; but nevertheless '?ENDIF' could be used too
```

Normally colliding conditional assembler directives should not happen so often within a program compared to **s'AVR** directives (hopefully), so typing the additional question marks should not hurt too much.

If those question marks are forgotten (but would be required) fortunately it will be recognized by **s'AVR** as the begin/end pairs of directives do not match. Therefore unbalanced program structures will be recognized and claimed by **s'AVR**.

Non-AVRASM2 Assembler: (conditional assembly)		**s'AVR:** (structured directive)
`IF`	same syntax	**IF**
	no equivalent	**ELSEIF**
`ELSE`	same syntax	**ELSE**
`ENDIF`	different syntax	**ENDI**

In general the **s'AVR** error messages are showing quite good hints (in the compiled output file!) to fix the problem.

Under Atmel Studio conveniently the line numbers are shown at least in the **s'AVR** source file (must be selected under Options).

And **s'AVR** is designed to synchronize itself after detecting errors to prevent a bunch of additional unexpected error messages which would be confusing only and **s'AVR** doesn't stop compiling after any error is detected.

Collision with Assembler Macros

Since AVR macro directives start with a dot, there should be no conflicts with AVR macros. However, **s'AVR** statements should not be used within macros because of the global addresses being generated by **s'AVR**.

s'AVR will ignore AVR macros like all assembly lines or pass them unchanged into the output file.

[26] The question mark can not be misinterpreted as a "conditional operator" (from AVRASM 2.1), as this is not the first printable character of a line.

Comments

Comments by means of semicolons are treated by *s'AVR* as such and reissued in the output file, unless they are part of *s'AVR* lines and these are to be suppressed by the given *s'AVR* option.

Comments between /* and */ over several lines (as well supported by AVRASM2) are not recognized as comments by *s'AVR*, which causes *s'AVR* statements within such comment areas (but not within lines with /* and */) nevertheless be evaluated and compiled (possibly also with corresponding errors, warning and hints).

But that does not hurt, as the AVR assembler continues to see comments between /* and */. Only a few redundant labels are generated by *s'AVR*.

It is important, however, that in those lines with /* and */ there are no *s'AVR* statements that would not be captured by *s'AVR*, which would lead to unbalanced structures (together with an error message) if additional *s'AVR* statements are elsewhere in the /**/ comment section.

Addresses, Labels

As soon as *s'AVR* recognizes assembly instructions, comments and labels in a source line of the input file, the remainder of this line is not examined for *s'AVR* statements.

In other words:

- In front of *s'AVR* statements, no label may be placed.
- Each AVR assembly instruction must have its own source line.

Multiple *s'AVR* Statements in a single Line

On the other hand, several *s'AVR* statements directly one after the other without assembly instructions in between are allowed in a single line, even if this does not look very clear = unstructured.

Sometimes it can be very compact and then save program lines.
However, for complex lines, it can happen that the original *s'AVR* line is output several times as a comment. The generated AVR assembly code is normally still correct.

Examples:

```
LOOP ENDL                          ; endless loop (until Interrupt or Reset)

REPEAT UNTIL %Portx,4              ; wait until Bit 4 of Portx is set

WHILE rega <> regb EXITIF T ENDW   ; wait until both registers are equal
                                   ; or the Transfer Bit is set
```

If an AVR assembly instruction is in the same line before, between or after *s'AVR* directives, either an unbalanced structure is detected or the assembly instruction is lost. Conclusion: In such cases better use clean structured programming!

Some basic Syntax Rules

As noted earlier, ***S'AVR*** has been designed to work with AVRASM2 and, if applicable, the associated Atmel® Studio development environment.
Therefore, the same rules apply to symbols, numbers and so on:

Identifiers (including addresses, symbols, registers, etc.) begin with a letter or underscore and can be followed by additional letters, underscores, and numbers. The addresses generated by ***S'AVR*** are either preset `_Lxxxx` or optional `_Axxxx` though `_Zxxxx` and may not have been used elsewhere.

A **single comma** between two identifiers is mandatory (with no spaces in front of it) if **register bits** or **port bits** are to be addressed.

Numbers can be decimal, hexadecimal (`$ff` or `0xff`) and binary (`0b1111_1111`) (the latter also with underscores for a clearer representation, but this is not accepted by every AVR assembler). For register and port bits, only decimal numbers 0-7 (or symbols) are allowed.

In addition for *S'AVR*:

Because in ***S'AVR*-LITE** all instructions are 8-bit operations, values >255 are displayed as errors[27] on both decimal, hexadecimal, and binary values.

Texts are recognized by ***S'AVR*** using simple quotes (`'abc -!? ...'`), but only single byte characters are required for comparisons (`'a'`). ***S'AVR*** allows some freedom, which eventually has to be checked by the assembler to be valid.

Ports are marked by prefix % (without spaces after the prefix!).

Constants (immediate literals) need a prefix # and have the formats `#999`, `#$ff`, `#0xff`, `#0b1111_1111` and `#'x'` (each without spaces after the prefix, not accepted by every AVR assembler).

These forms are recognized as value zero:
`#0`, `#$0`, `#$00`, `#0x0`, `#0x00`, `#0b0`, `#0b00`, `#0b000`, `#0b0000`.
They result partly in optimized assembly code.

Other formats may be accepted by AVRASM2 and other AVR assemblers, but not by ***S'AVR*** (and can then be used with assembly instructions).

These rules are mandatory in the context of ***S'AVR*** statements. If an AVR assembler with a different syntax is used, the assembly instructions may be different without ***S'AVR*** being affected, but the code generated by ***S'AVR*** must also be understood by the assembler.

***S'AVR*-LITE** only generates relative jumps via `RJMP` instruction and branch or skip instructions to skip `RJMP` instructions, so ***S'AVR*-LITE** can be used on older ATtinys that

[27] #256 is only accepted to initialize a FOR loop and taken as #0 in the generated code, making the FOR loop pass correctly 256x.

do not support `JMP`. ***S'AVR*** does not use any macros but only generates native AVR assembly instructions.

Command-Line-Interface

Command-Line Call

S'AVR is supporting a DOS-style **command line interface**, which means that ***S'AVR*** also can be started from a DOS window or other WINDOWS® programs. Therefore the GUI no longer is a must for ***S'AVR***.

These command line calls need to meet the following **rules**:

- All parameters are passed to ***S'AVR*** by a preceding slash (/).
- The first parameter must be a valid and existing ***S'AVR*** source code file with extension ".s", else ***S'AVR*** will be started in WINDOWS® mode.
- Following the file name any number of additional parameters - separated by a slash - are accepted, see command line syntax.
- If any parameters are conflicting, the last one will be used.
- Unknown parameters will be ignored.
- Error messages will be supplied within the generated output file (*.asm), also in a separate error file (*.err, from version 2.21) in case any errors are detected (otherwise such an error file does not exist).
- Upper and lower case letters and spaces in between are tolerated in any combination and are even allowed within file names.

Command-Line Syntax:

s'AVR-Lite.exe /filename.s | /filename.savr [/nosavr | /keepsavr | /nowarn | /keepwarn | /nohint | /keephint | {label_a .. label_z} | /listclp]

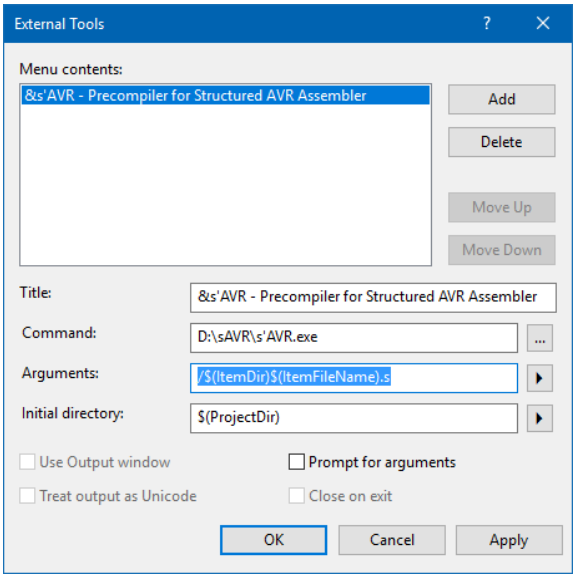
/filename.s **/filename.savr**	vaild ***s'AVR*** source code file name (need extension ".s" **or** extension ".savr")
/nosavr	***s'AVR*** statements are not shown in the output file
/keepsavr	***s'AVR*** statements are shown within the output file as comments (default)
/nowarn	***s'AVR*** warnings are not shown in the output file
/keepwarn	***s'AVR*** warnings are shown within the output file as comments (default)
/nohint	***s'AVR*** hints are not shown in the output file
/keephint	***s'AVR*** hints are shown within the output file as comments (default)
/label_a **... /label_z**	instead of the default Label Prefix `_Lxxxx` Label Prefixes `_Axxxx` though `_Zxxxx` will be used
/listclp	besides the output file an information file called "**sAVR_listclp.txt**" is listing all the parameters found and the corresponding compiler flags (usually in the same directory from which ***s'AVR*** was started).

Command-Line Call per Atmel® Studio

At the beginning the embedding of ***s'AVR*** into Atmel® Studio was described in such a way that the ***s'AVR*** program window appears when the embedded ***s'AVR*** tool is called, so that you can select the respective source program '.s' in the corresponding project directory when calling it up for the first time. If necessary, for every compiler run the options being offered can be individually deselected. With this method, the ***s'AVR*** program window can remain open in the background after the first call.

If you also fill in the Atmel® Studio 'Arguments' when setting up ***s'AVR*** as an external tool, you can pass the source file name and, if necessary, ***s'AVR*** options via command line call to ***s'AVR***, but then without the possibility to intervening via the GUI, which even does not show up.

For default s'AVR options, the 'Tools Arguments' set-up would look like this:



As far as possible the individual argument parameters take place without any spaces between them. Otherwise, Atmel® Studio will pass the parameters individually in single quotes[28].

Several 'Arguments' are separated by another '/$' (in addition to the '/' of the s'AVR parameters). 'Prompt for Arguments' remains deselected[29].

It is important that the file extension '.s' is specified as well, otherwise it may be (depending on the currently active file) that '.asm' is passed to s'AVR instead of '.s' and thus the wrong source file is compiled.

After setting up properly with Atmel® Studio, a single mouse click in the menu item 'Tools' is sufficient to start s'AVR and '.asm' is updated at lightning speed (recognizable by date and time) without the s'AVR program window being visible or even being served - provided you have saved the source program '.s' after any changes before the call.

A '*' at the end of the filename displayed under Atmel® Studio disappears after the source file has been saved (a new and nice feature since a while)!

Linux

s'AVR can be called up and used under Linux using WINE.

By using scripts under Linux you can also compile and assemble complex projects with a few mouse clicks. Specifically, from version 2.21 s'AVR generates a separate error file with the extension '.err' in case any errors have been detected (only then), which lists purely the error messages (if any) line by line.

From version 2.23 s'AVR is also able to directly read and compile Linux text files without the intermediate step `unix2dos`[30].

[28] From version 1.04, s'AVR automatically removes double quotes from the command line.
[29] To check the parameter passing for correctness, you can temporarily enable this option.
[30] Windows and DOS are using CRLF at the end of a text line, while Unix and Linux are only using LF.

Error Messages

s'AVR always stores error messages in the output file at the suspected error location with reference to the line number of the source program and starts the line with `.ERROR`, which forces AVRASM2 to stop and display the **s'AVR** error message under Atmel® Studio. If necessary, you can directly jump to the corresponding line in the output file (not in the source program file) by a double click on the error message. However, the error must be fixed in the source file (`*.s`) and not in the output file!

In case any errors are detected, all error messages will be written to a separate error file (`*.err`, which is helpful when using scripts under Linux, e.g.).

Outlook

The limitation of the LITE version due to the `RJMP` instruction is omitted in the full version (allows optional `JMP` instructions or range extension '`.l`' respectively).
Also, the full version will support 16-bit operations with some **s'AVR** directives.

Apart from that, very complex AVR programs can be written in a well-structured manner using **s'AVR**, and the "efficient" AVR code that can be generated with **s'AVR** version 2 absolutely does not have to shy away from a comparison with a pure assembler program written by an AVR guru.

Using "Label Prefix" (from **s'AVR** 2.1), you can also handle AVR programs consisting of several **s'AVR** modules.

Depending on success and demand of the **s'AVR** users (and my free resources), future **s'AVR** versions to be released - besides fixing bugs (if any) - might offer additional features as follows:

- **AND-OR** combinations within conditions
- **IN** lists within conditions (like in PASCAL/DELPHI)
- improved **FOR** statement (step size)
- combined **WHILE-UNTIL** statement
- **SWITCH-CASE** statement
- any other proposals improving **s'AVR**

Although **s'AVR** has been carefully checked for proper function, programming errors can never be ruled out due to the complexity of the program[31].

Error messages are always welcome by e-mail (if possible with an attached source code snippet as a text file that causes the error and the **s'AVR** options used or the associated output file).

Comments (rants and raves) and suggestions to **s'AVR** are of course also welcome.

And now: Have fun and success with Atmel® AVR® combined with **s'AVR**!

(An index will follow occasionally.)

[31] s'AVR itself has been created using a quite old Delphi 5 package (which means that s'AVR is coded in Pascal), which is still running well under Windows® 10.