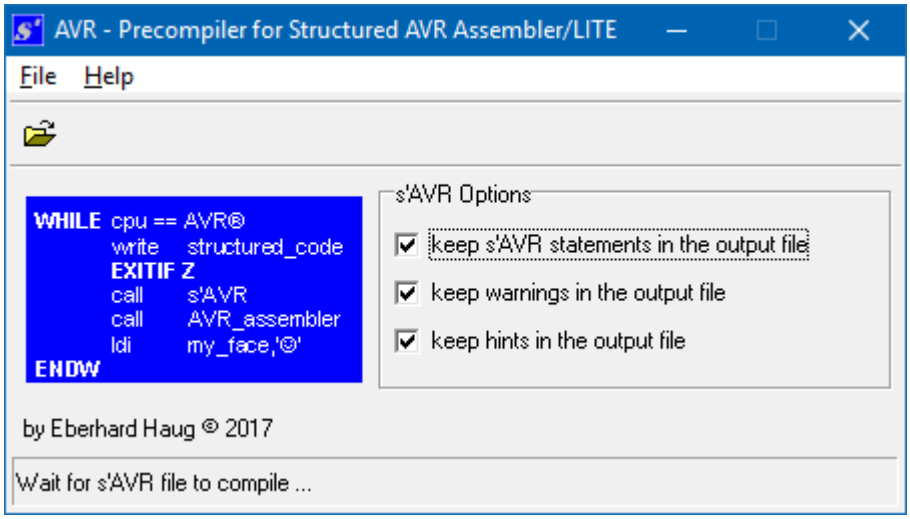
# s'AVR-LITE

Strukturierte Assembler-Programmierung für Atmel® AVR®



# Inhalt

## Strukturierte Programmierung

Zunächst sollte eine typische Begriffsverwechslung klargestellt werden:

> Unter Steueranweisungen für strukturierte Programmierung versteht man **nicht** bedingte Assemblierung (Conditional Assembly).

Steueranweisungen, wie sie von ***S'AVR*** und ***S'AVR*-Lite**[1] zur Verfügung gestellt werden, bieten folgende Vorteile:

- **erlauben strukturierte Programmierung in Assembler-Umgebung**
- **vereinfachen die Programmentwicklung und das Debugging**
- **verbessern die Programmlesbarkeit und die Programmdokumentation**
- **verkürzen die Programmentwicklung und vergrößern die Produktivität**
- **erhöhen die Programmzuverlässigkeit**
- **erleichtern die Programmpflege**
- **bringen mehr Spaß beim Schreiben von AVR-Assembler-Programmen**

***S'AVR*** bietet dem Programmierer wie bei höheren Programmiersprachen Anweisungen für die Realisierung von typischen Verzweigungs- und Schleifensituationen. Allerdings ist die Anwendung der ***S'AVR***-Anweisungen bei Weitem nicht so komplex wie bei höheren Programmiersprachen, sondern ausgesprochen einfach in der Anwendung.

Diese ***S'AVR***-Anweisungen beeinflussen die Code-Effizienz des Programmes im Vergleich zu reiner Assembler-Programmierung nur wenig[2]. Das bedeutet, daß sowohl die Rechenleistung als auch die Speicherausnützung des AVR-Mikrocontrollers vergleichbar sind mit ausschließlicher AVR-Assembler-Programmierung.

Bei Verwendung von ***S'AVR***-Steueranweisungen bleiben alle Vorteile der Assembler-Programmierung erhalten, da alle Nicht-***S'AVR***-Anweisungen ganz normale Assembler-Befehle sind. Das ist der ganze Trick!

## Des Pudels Kern von *S'AVR* kurz und bündig:

**Im *S'AVR*-Quellprogramm (`*.s` oder `*.savr`) sind alle Anweisungen ganz normale AVR-Assembler-Befehle soweit sie keine *S'AVR*-Anweisungen sind.**

Deshalb können bestehende AVR-Assembler-Quellprogramme nachträglich[3] (auch nur teilweise) mit ***S'AVR***-Anweisungen zur strukturierten AVR-Assembler-Programmierung erweitert werden.

Nach dem Compilieren steht ein "flaches" AVR-Assembler-Quellprogramm zum Assemblieren, Debuggen und Programmieren des jeweiligen AVR-Mikrocontrollers mit den vorhandenen Entwicklungs-Tools zur Verfügung.

[1] Mit ***S'AVR*** ist in diesem Handbuch immer auch ***S'AVR*-Lite** gemeint, es sei denn, es sind explizit Einschränkungen genannt.
[2] Dies ist bedingt durch den AVR-Befehlssatz (fehlende geeignete Skip-Befehle zur Statusregister-Abfrage und Branch-Befehle über einen größeren Adressbereich). Sonst könnte die Effizienz des erzeugten Codes noch etwas besser sein.
[3] Mit dieser Methode wurden (neben weiteren detaillierteren Untersuchungen) erste Tests auf Richtigkeit mit ***S'AVR*** durchgeführt.

## Wofür **s'AVR** gedacht ist

**s'AVR** (lauffähig unter allen aktuellen Windows®-Versionen[4]) ist ein Precompiler ("Vorübersetzer"), der Anweisungen für strukturierte Programmierung (genannt **s'AVR**-Anweisungen) in Assembler-Quellsprache für die 8-Bit-AVR-Familie von Atmel® übersetzt.

Zwischen den **s'AVR**-Anweisungen stehen die eigentlichen AVR-Assembler-Befehle (typischerweise für die eigentliche Datenmanipulation und Unterprogrammaufrufe), die vom **s'AVR**-Precompiler unangetastet bleiben.

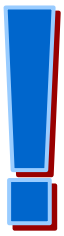
Natürlich können zur Programmverschachtelung beliebig viele weitere **s'AVR** -Anweisungen und AVR-Assembler-Befehle dazwischen stehen.

Der von **s'AVR** erzeugte AVR-Quellcode[5] muss anschließend von irgend einem Atmel®-kompatiblen Assembler in den endgültigen AVR-Objektcode übersetzt werden.

Das von **s'AVR** erzeugte AVR-Assembler-Quellprogramm kann zum Simulieren und Debuggen verwendet werden.

Zur besseren Übersicht können beim Debuggen die **s'AVR**-Anweisungen als Kommentare in derselben Datei stehen bleiben. Optional lassen sie sich aber auch aus der Ausgabedatei entfernen. Ebenso lassen sich wahlweise **s'AVR**-Warnhinweise und **s'AVR** -Tipps aus der Ausgabedatei entfernen.

**Wichtig!**



Eine sehr wesentliche Eigenschaft von **s'AVR** ist die ausschließliche Verwendung der in den **s'AVR**-Anweisungen spezifizierten AVR-Register.

Ansonsten werden keine weiteren AVR-Register[6] verwendet oder verändert!

Auch der AVR-Programm-Stack wird von **s'AVR** nicht angetastet. Das heißt, der AVR-Programmentwickler hat vollständige und alleinige Kontrolle über alle AVR-Register!

**s'AVR**-Strukturen können beliebig tief geschachtelt werden. Das Maximum wird nur durch den Programmspeicher des verwendeten AVR-Mikrocontrollers bestimmt.

Da **s'AVR** für den Rücksprung aus den jeweiligen Strukturen nicht den AVR-Stack[7] verwendet, können bei Bedarf Sprünge an beliebige Stellen im **s'AVR**-Programm gemacht werden, sogar in andere **s'AVR**-Strukturen[8] hinein.

Abhängig von der Art des Programmierens sollte **s'AVR** im Vergleich zu ausschließlicher AVR-Assembler-Programmierung deshalb ähnliche Code-Effizienz ergeben, bei gleichzeitig deutlich besserer Lesbarkeit, weniger Programmieraufwand, erleichtertem Debuggen usw., vor allem verbunden mit mehr AVR-Programmierspaß auf einem stolperfreien Weg zum erfolgreichen AVR-Assembler-Programm!

---

[4] Alle Versionen von Windows® 98 bis Windows®10.
[5] Bei **s'AVR-Lite** werden ausschließlich die von ATtiny unterstützten Befehle verwendet, also `RJMP` statt `JMP`.
[6] Genaugenommen wird natürlich der Befehlszähler und ggf. das Statusregister verändert.
[7] Strukta (für 8080/8085 und Z80 verfügbar) hatte den Stack verwendet, so dass immer Vorsicht angebracht war.
[8] Sprünge in Unterprogramme hinein und von Unterprogrammen heraus wollen immer wohl überlegt sein.

## Was **s'AVR** nicht kann

**s'AVR** erzeugt aus dem **s'AVR**-Quellprogramm nicht direkt AVR-Objektcode.
Deshalb wird **s'AVR** ein Precompiler genannt.

Da **s'AVR** keine absoluten Programmadressen kennt, kann er u.a. keine Adressbereichsüberschreitungen erkennen, so dass man ggf. von Hand eingreifen muss, z.B. falls bei **s'AVR-LITE** der ±2k-Worte-Sprungbereich der `RJMP`-Befehle überschritten wird.

Weiterhin kann **s'AVR** nicht überprüfen, in wieweit definierte und verwendete Symbole und Daten gültig und innerhalb zulässiger Grenzen sind. Dies ist allein Aufgabe des nachgeschalteten Assemblers. **s'AVR** selbst benötigt keine Deklarationen und wertet auch keine aus (wie z.B. jene des Assemblers).

## **s'AVR**–Installation und Deinstallation

**s'AVR** ist ein sehr kompaktes[9] direkt unter Windows® ausführbares Programm, das keine besondere Installation benötigt und deshalb auch nicht in die Windows®-Registry eingreift (und in der Originalversion des Entwicklers auch sonst frei von bösartigen Dingen ist). Das **s'AVR** -Programm wird einfach in ein beliebiges (sinnvollerweise eigenes) Verzeichnis kopiert. Der Dateiname darf dabei problemlos geändert werden.

Die 'Deinstallation' von **s'AVR** erfolgt schlicht durch Löschen von `s'AVR.exe`.

Das ausführliche (hier vorliegende) **s'AVR**-Handbuch steht derzeit in deutscher Sprache im PDF-Format zur Verfügung. Eine englischsprachige Version ist in Vorbereitung.
Das Hilfe-Menü des Programms fasst die Syntax der **s'AVR**-Steueranweisungen, Vergleiche, Status-Bits etc. kurz in englischer Sprache zusammen.

## **s'AVR** im Einsatz

Zunächst wird das Quellprogramm der Applikation unter Verwendung eines bevorzugten Texteditors erstellt unter Beachtung der Syntax von **s'AVR** und des zu benützenden AVR-Assemblers.

Die Dateierweiterung des **s'AVR**-Quellprogrammes sollte "`.s`" heißen, also z.B. `Mein_sAVR_Programm.s`. Wahlweise ist auch "`.savr`" zulässig, siehe weiter unten.

**s'AVR** wird gestartet wie jedes Windows®-Programm (mittels **s'AVR**-Icon, dem Windows®-Explorer, dem Menüpunkt Start – Ausführen ... etc.).

Jetzt kann - bei Bedarf nach Auswahl einiger **s'AVR**-Optionen - die gewünschte `*.s`-Datei über ein übliches Windows®-Dateimenü ausgewählt und compiliert werden.

Die beim Compilieren erzeugte Ausgabedatei besteht aus einem "flachen" (strukturlosen) AVR-Assembler-Quellprogramm und hat nun voreingestellt die Dateierweiterung "`.asm`" für den AVR-Assembler.

Jedoch Vorsicht: Falls Dateien mit demselben Namen und der Dateierweiterung *.asm im selben Verzeichnis vorhanden sind: Sie werden ohne nachzufragen überschrieben!

[9] Die LITE-Version hat eine Dateigröße von weniger als 600KByte.

Weitere **s'AVR**-Dateien können nacheinander compiliert werden ohne **s'AVR** verlassen zu müssen. Eine Statuszeile gibt Auskunft über den Zustand bzw. Verlauf des Compiler-Laufes. Abschließend wird in der Statuszeile u.U. die Anzahl eventueller Fehler bzw. Warnhinweise angegeben.

Die erzeugte Assembler-Quelldatei kann nun zum Assemblieren und Programmieren und/oder Debuggen des AVR z.B. in das AVR-Studio geladen werden.

Falls Fehler bzw. Warnhinweise beim Compilieren aufgetreten sind, erscheinen sie an entsprechender Stelle als Fehlertext in der Ausgabedatei unter Bezug auf Zeilennummern der **s'AVR**-Quelldatei. Der Fehlertext erscheint in der Ausgabedatei nicht als Kommentar, damit bei versehentlichem Assemblieren einer fehlerhaften Compilierung eine weitere Meldung[10] des Assemblers erfolgt.

**s'AVR** kann alternativ auch per **Command-Line-Aufruf** mit Parametern gestartet werden, z.B. direkt vom Editor aus.

Beispiel einer Kommandozeile, die **s'AVR-LITE** aufruft, um ein Quellprogramm zu compilieren:

```
s'AVR-Lite.exe /Mein_sAVR_Programm.s /asm
```

In diesem Fall würde der Aufruf des nachgeschalteten AVR-Assemblers und z.B. eines eines AVR-Debug-Tools ebenfalls mit der selben Methode mit nur einem Mausklick durch den Editor erfolgen.

Details zum Command-Line-Aufruf siehe am Ende dieses Handbuches.

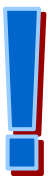
**Anmerkung:**

Obwohl **s'AVR**-Anweisungen in Groß- und Kleinschreibung akzeptiert werden (sogar gemischt), sind **s'AVR**-Anweisungen in diesem Handbuch groß geschrieben (wie auch der von **s'AVR** erzeugte AVR-Assembler-Quellcode).

Soweit AVR-Assembler-Befehle jedoch Teil der Beispiele sind, werden sie der Übersichtlichkeit halber klein und in `Courier`-Schrift geschrieben.

## Einbindung in Atmel®-Studio

Zunächst erstellt man sein strukturiertes Assembler-Quellprogramm mit der Dateierweiterung `*.s` (speziell voreingestellt für den Studio-Editor, damit Visual Assist verwendet wird) oder wahlweise und zur Unterscheidung auch `*.savr`, und zwar entweder vorab mit einem beliebigen Texteditor oder erst, nachdem das Projekt unter Studio bereits eingerichtet wurde.



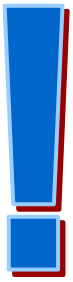
Da bei einem Assembler-Projekt von Studio eine Assembler-Quelldatei `main.asm` angelegt wird, muss man diese im 'Solution Explorer' zunächst mit der rechten Maustaste in `Mein_sAVR_Programm.asm` umbenennen[11] und an derselben Stelle mittels *'Set As EntryFile'* dem Assembler zuordnen, denn sonst wird die `*.asm`-Datei nach dem Compilieren durch **s'AVR** nicht automatisch vom Studio-Editor neu geladen!

Dann kann man von Studio aus eine zusätzliche Textdatei für das strukturierte **s'AVR**-Quellprogramm öffnen, entweder die bereits erstellte vorhandene Quelldatei

[10] Grundsätzlich könnte man für AVRASM2 auch eine `.ERROR`-Zeile erzeugen, was aber derzeit nicht in **s'AVR** implementiert ist.
[11] `main.asm` lässt sich (warum auch immer) nicht von einem anderen Program (wie **s'AVR**) gleichzeitig öffnen und beschreiben.

`Mein_sAVR_Programm.s` oder eine neue Assembler-Datei, die man dann unter dem gewünschten Namen (aber weiterhin mit der Dateierweiterung[12] `*.s`) gleich wieder abspeichert.

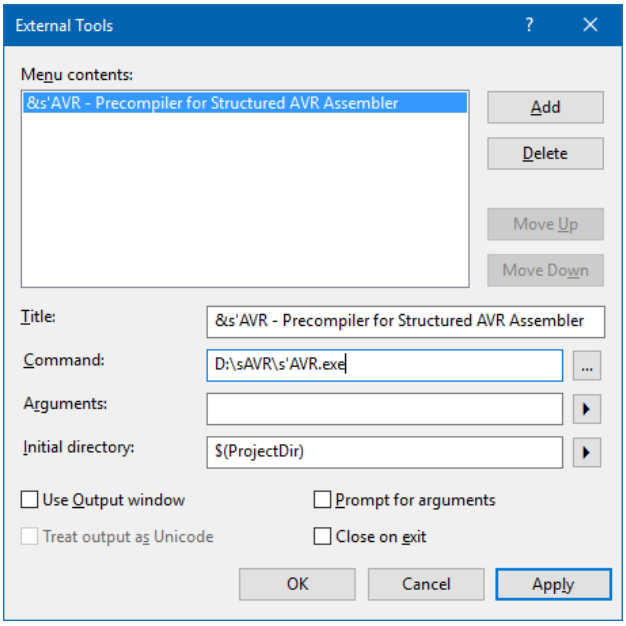
Nun lassen sich beide Quelldateien (also das mit ***s'AVR***-Syntax geschriebene `Mein_sAVR_Programm.s` und das von ***s'AVR*** compilierte "flache" Assembler-Quellprogramm `Mein_sAVR_Programm.asm`) mit dem Studio-Editor nebeneinander bearbeiten - sinnvollerweise aber nur die `Mein_sAVR_Programm.s`-Datei!



Die `*.s`-Datei muss man vor dem Auruf von ***s'AVR*** unbedingt abspeichern, denn sonst wird die zuletzt abgespeicherte Datei compiliert und man wundert sich, warum der AVR-Code nicht aktuell ist (ggf. überprüfbar an Datum und Uhrzeit der `*.asm`-Datei)!

Die `Mein_sAVR_Programm.asm`-Datei wählt man nur zum Assemblieren aus (sie wird normalerweise nach jedem ***s'AVR***-Aufruf automatisch aktualisiert[13]) oder zum Überprüfen des von ***s'AVR*** erzeugten "flachen" Assembler-Codes und ggf. irgendwelcher Fehlermeldungen. Vor allem nimmt man in der `*.asm`-Datei keine Änderungen vor!

Schließlich wird man auch ***s'AVR*** selbst noch direkt in das Atmel-Studio® einbinden wollen. Hierzu ruft man unter 'Tools' einmalig das Untermenü 'External Tools' auf und trägt dort z.B. wie folgt ein ('Command' je nach Speicherort von ***s'AVR***, zunächst[14] keine Angabe bei 'Arguments', mehr siehe unter ' Command-Line-Aufruf per Atmel®-Studio):




Nach Apply/OK kann man mit dem Studio-Editor nun sowohl den ***s'AVR***- als auch den Assembler-Quellcode bearbeiten und ***s'AVR*** unter 'Tools' direkt aufrufen[15].

Nach dem (blitzschnellen!) Compilieren steht der erzeugte Assembler-Quellcode normalerweise[16] als automatisch aktualisiertes `Mein_sAVR_Programm.asm` in Studio zur Verfügung sobald man in das `*.asm`-Fenster klickt und kann nun von Studio aus

[12] Bei Dateierweiterungen abweichend von *.asm und *.s verschwindet die Unterstützung durch Visual Assist.
[13] Unter Tools/Options/Environment/Documents aktivieren: *'Detect when file is changed ...'* und *'Reload modified files ...'*.
[14] Falls auch noch 'Arguments' an s'AVR übergeben werden, kann man ganz ohne s'AVR-Fenster arbeiten.
[15] Bei einem Update von Atmel Studio 7.0 ist dieser Eintrag verschwunden, so dass er wieder neu angelegt werden musste.
[16] Sofern die `*.asm`-Datei im *'Solution Explorer'* mittels *'Set As EntryFile'* markiert wurde.

sofort mittels 'Build' oder 'Rebuild' assembliert und (sofern fehlerfrei) per 'Device Programming' in den Programmspeicher des AVR-Mikrocontrollers übertragen werden.

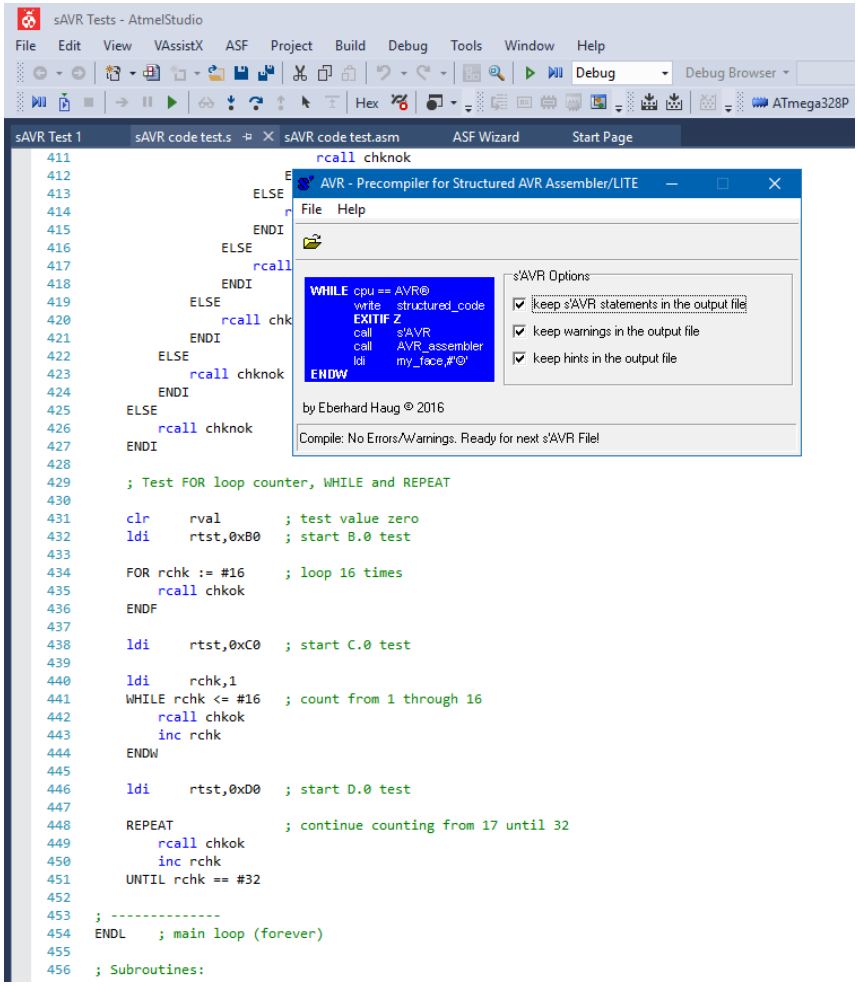
Im Normalfall bearbeitet man seinen Source-Code nur im `*.s`-Fenster, klickt (nach dem Abspeichern) das **s'AVR**-Fenster zum Compiler-Aufruf und geht direkt zu 'Build' und benötigt das `*.asm`-Fenster nur zum Fehlersuchen oder Simulieren/Debuggen.

Nochmals der wichtige Hinweis:

**s'AVR** überschreibt eine im selben Verzeichnis vorhandene Datei mit der Erweiterung `*.asm` (ggf. von einer vorausgehenden Compilierung, hoffentlich nicht ein selbst geschriebenes Assembler-Quellprogramm) ohne nachzufragen!
Nur so macht eine einfache Bedienung mit möglichst wenig Maus-Klicks Sinn.

**s'AVR** bleibt praktischerweise für den nächsten Compilier-Lauf geöffnet (verschwindet in den Hintergrund nach Anklicken von *.s), sofern man **s'AVR** nicht selbst beendet.

Und so schaut schließlich ein Bildschirm-Snippet von strukturierter AVR-Assembler-Programmierung nach einem Aufruf von **s'AVR** innerhalb von Atmel®-Studio aus:




```
411                         rcall chknok
412                     E
413                 ELSE
414                     r
415                 ENDI
416             ELSE
417                 rcall
418             ENDI
419         ELSE
420             rcall chk
421         ENDI
422     ELSE
423         rcall chknok
424     ENDI
425 ELSE
426     rcall chknok
427 ENDI
428
429 ; Test FOR loop counter, WHILE and REPEAT
430
431 clr     rval        ; test value zero
432 ldi     rtst,0xB0   ; start B.0 test
433
434 FOR rchk := #16     ; loop 16 times
435     rcall chkok
436 ENDF
437
438 ldi     rtst,0xC0   ; start C.0 test
439
440 ldi     rchk,1
441 WHILE rchk <= #16   ; count from 1 through 16
442     rcall chkok
443     inc rchk
444 ENDW
445
446 ldi     rtst,0xD0   ; start D.0 test
447
448 REPEAT              ; continue counting from 17 until 32
449     rcall chkok
450     inc rchk
451 UNTIL rchk == #32
452
453 ; --------------
454 ENDL    ; main loop (forever)
455
456 ; Subroutines:
```

Mittels Kommandozeilen-Aufruf (Details siehe dort) kann man das **s'AVR**-Fenster ganz umgehen, muss dann aber ggf. auch gewünschte Optionen mit übergeben.

## **s'AVR**-Anweisungen:

| | | |
|---|---|---|
| **IF** | *Bedingung* **[THEN]**<br>*BefehlsFolge* | ; erste Abfrage |
| **ELSEIF** | *Bedingung* **[THEN]**<br>*BefehlsFolge* | ; weitere Abfrage(n), mehrfach optional |
| **ELSE** | <br>*BefehlsFolge* | ; letzte Verzweigung, optional |
| **ENDI** | | ; Ende der IF-Struktur |
| **LOOP** | | ; Endlosschleife |
| | *BefehlsFolge* | ; Verlassen der LOOP-Schleife nur mittels<br>; EXIT, EXITIF, Assembler-Sprung- und Branch-<br>; Befehlen, RET, Interrupt oder AVR-Reset |
| **ENDL** | | ; Ende der LOOP-Schleife |
| **WHILE** | *Bedingung*<br>*BefehlsFolge* | ; Abfrage zu Beginn der Schleife |
| **ENDW** | | ; Ende der WHILE-Schleife |
| **REPEAT** | <br>*BefehlsFolge* | ; Beginn der REPEAT-Schleife |
| **UNTIL** | *Bedingung* | ; Abfrage am Ende der Schleife |
| **FOR** | *RegisterZuweisung*<br>*BefehlsFolge* | ; FOR-Schleife mit Register-Initialisierung |
| **ENDF** | | ; Dekrementieren des Schleifenzählers und<br>; Überprüfung auf den Wert Null |
| **EXIT** | | ; Verlassen einer Strukturebene |
| **EXITIF** | *Bedingung* | ; bedingtes Verlassen einer Strukturebene |
| **EXITIF** | *Bedingung* **TO** *Adresse* | ; bedingter Sprung aus einer Strukturebene |

**Anmerkungen**:

- *BefehlsFolge* steht für eine beliebige Anzahl von AVR-Assembler-Befehlen und/oder **s'AVR**-Anweisungen.
  **s'AVR**-Anweisungen können demnach beliebig tief geschachtelt werden - solange genügend Programmspeicher vorhanden ist.
- *Bedingung* steht für Vergleiche (mindestens ein AVR-Register), Register-Bit-Abfragen oder Statusregister-Flag-Abfragen
- *RegisterZuweisung* gibt die Zahl der FOR-Schleifendurchgänge an. Das spezifizierte Register kann bereits vorher geladen sein oder die Zuweisung ist Teil des Befehls. Das FOR-Schleifenregister kann entweder mit einer Konstanten oder dem Inhalt eines anderen AVR-Registers geladen werden.
- *Adresse* steht für eine (nicht lokale) Adresse an beliebiger Stelle im **s'AVR**-Programm (jedoch Vorsicht beim Springen aus Unterprogrammen). EXITIF-TO ist die einzige nicht-strukturierte **s'AVR**-Direktive und erlaubt codesparenden und 'schnellen' bedingten Ausstieg aus einer **s'AVR**-Struktur. Ein unbedingter Sprung wird einfach mit dem Assembler-Befehl `RJMP Adresse` bzw. `JMP Adresse` [nicht für ATtiny] durchgeführt.
- **s'AVR** verwendet nur die in den **s'AVR**-Anweisungen spezifizierten Register.
  Verändert werden sonst nur der Befehlszähler und (bedingt durch einige der Abfragen) auch das Statusregister.
- Kollision mit bedingten Assemblier-Anweisungen kann von **s'AVR** vermieden werden: Direkt vor kollidierende Anweisungen wird im **s'AVR**-Quellprogramm ein '?' gestellt, das von **s'AVR** automatisch entfernt wird, so daß die Anweisungen für den Assembler-Lauf wie gewünscht zur Verfügung stehen. AVRASM2 sollte davon nicht betroffen sien.
- **s'AVR**-Anweisungen können mit Groß- oder Kleinbuchstaben (auch gemischt) geschrieben werden

# Beschreibung der *s'AVR*-Anweisungen

## Allgemeine Syntax-Regeln

Der Einfachheit halber verwendet ***s'AVR*** nicht die bei Hochsprachen üblichen 'Full-Size'-Strukturen wie IF–THEN–ELSEIF–THEN–ELSE–ENDIF, WHILE–DO–ENDWHILE etc.

THEN und DO werden nicht benötigt (THEN ist wahlweise, DO ist nicht zulässig) und an alle END wird einheitlich kurz und bündig nur der erste Buchstaben des "Initiators" angehängt, also IF–END**I**, WHILE–END**W**, LOOP–END**L**, FOR–END**F**.

Aufgrund der Assembler-Umgebung können IF-Strukturen (im Gegensatz zu Hochsprachen) auch per EXIT und EXITIF verlassen werden.

REPEAT wird für die abschließende Schleifenabfrage mit UNTIL beendet.

THEN kann wahlweise verwendet werden. EXITIF erhält optional ein TO für einen bedingten absoluten Sprung aus einer Struktur heraus.

Wie bereits unter obigen "Anmerkungen" erwähnt, dürfen ***s'AVR***-Anweisungen mit Groß- oder Kleinbuchstaben (sogar gemischt) geschrieben werden. In diesem Handbuch sind ***s'AVR***-Anweisungen und von ***s'AVR*** generierte Assembler-Befehle zur Unterscheidung von Assembler-Befehlen des Quellprogramms (die im Handbuch immer klein geschrieben sind) jedoch durchweg mit Großbuchstaben geschrieben.

Einige grundsätzliche Regeln bezüglich der Darstellung von Zahlen etc. sind am Ende dieses Handbuches zusammengefasst.

> Die Grundidee von ***s'AVR*** ist möglichst einfaches, schnelles und übersichtliches ***strukturiertes*** Schreiben von AVR-Assembler-Programmen!

## Adressierungsarten

Da die AVR-Mikrocontroller u.a. Operationen mit Registern, Register-Bits, Port-Bits und Konstanten zulassen und diese unterschiedliche Assembler-Befehle zur Folge haben, müssen diese bei den ***s'AVR***-Anweisungen unterschieden werden.

Ohne besondere Kennzeichnung der Operanden werden Register angenommen bzw. dem Assembler wird die Überprüfung auf Richtigkeit überlassen.

Register-Bits werden in der Form ***Register,BitStelle*** angegeben und Konstanten ("Immediate") wird ein **#** vorgestellt.

Komplette Port-Inhalte müssen zunächst per Assembler-Befehl (`IN`) in ein AVR-Register geladen werden (eine AVR-Eigenschaft), bevor damit strukturierte Anweisungen erfolgen können.

Lediglich Port-Bits können per ***%Port,BitStelle*** direkt abgefragt werden.
Das **%**-Zeichen dient ***s'AVR*** zur Unterscheidung zwischen AVR-Register und AVR-I/O-Port (Beispiele folgen).

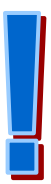
## Sprungbefehle

Leider bietet AVR (im Vergleich zu anderen Mikrocontrollern) nur wenige Skip-Befehle, die sich direkt auf das Status-Register beziehen. Dadurch wird ein universelles Verfahren für den Compiler weniger elegant (und der Compiler damit auch etwas aufwendiger), da statt Skip-Befehlen unnötigerweise Branch-Befehle zum Überspringen eines nachfolgenden `RJMP`-Befehls verwendet werden müssen, denn das Ziel war, dass das von ***s'AVR*** erzeugte Assembler-Programm ohne Eingriff in den erzeugten Source-Code immer direkt anschließend vom AVR-Assembler assembliert werden kann.

Die einzige Einschränkung für die LITE-Version ist die maximale Sprungweite des verwendeten `RJMP`-Befehls von ±2k Wortadressen. D.h., dass die ***s'AVR***-Strukturen bei der LITE-Version maximal 2k Wortadressen überstreichen dürfen.

## Vordefinierte Statusbits

```
Z       ; Zero-Bit gesetzt
NZ      ; Zero-Bit nicht gesetzt

C       ; Carry-Bit gesetzt
NC      ; Carry-Bit nicht gesetzt

H       ; Half/Digit-Carry-Bit gesetzt
NH      ; Half/Digit-Carry-Bit nicht gesetzt

S       ; Sign-Bit gesetzt, S = N ⊕ V (Negative EXOR Overflow)
NS      ; S-Bit nicht gesetzt

V       ; V-Bit gesetzt (Overflow, Überlauf 2er-Komplement)
NV      ; V-Bit nicht gesetzt

N       ; N-Bit gesetzt (Negative)
NN      ; N-Bit nicht gesetzt

T       ; Transfer-Bit gesetzt
NC      ; Transfer-Bit nicht gesetzt

I       ; Interrupt aktiviert
NI      ; Interrupt deaktiviert
```



***Wichtige Anmerkung zu den Statusbits***:

Diese Statusbits sind reservierte ***s'AVR*** –Schlüsselworte und dürfen deshalb innerhalb des ***s'AVR***-Programmes nicht für andere Symbole (Adressen, Register, Konstante) verwendet werden.

AVRASM2 selbst definiert zwar ebenfalls Z, C, H, S, V, N und T, was in diesem Zusammenhang aber nicht stört.

## EXIT und EXITIF, einheitliches Verlassen von s'AVR-Strukturen
## Beschreibung von Bedingungen und Vergleichen

Syntax:

| | |
|---|---|
| **EXIT** | ; Verlassen einer Strukturebene |
| **EXITIF** *Bedingung* | ; bedingtes Verlassen einer Strukturebene |
| **EXITIF** *Bedingung* **TO** *Adresse* | ; bedingter Sprung aus einer Strukturebene |

Sowohl das unbedingte EXIT als auch das bedingte EXITIF können verwendet werden um die aktuelle s'AVR-Struktur in genau eine zurückliegende Ebene zu verlassen.

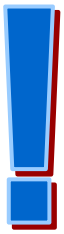
Soll – warum auch immer – in mehr als nur eine Strukturebene tiefer oder höher gesprungen werden, so kann ganz einfach ein unbedingter Assembler-Sprung (JMP [nicht bei ATtiny] bzw. RJMP) zum Einsatz kommen, was natürlich unstrukturiertes Programmieren bedeutet, jedoch in bestimmten Fällen übersichtlicher sein kann. Dabei ist aber äußerste Vorsicht geboten, wenn man auf diese Art Assembler-Unterprogramme[17] verlässt, denn der AVR-Stack (Stapelspeicher) könnte durcheinander geraten!

EXITIF erlaubt die Abfrage einer bestimmten Bedingung. Das kann ein bestimmter Inhalt eines Registers sein oder ein bestimmter Zustand eines Register- oder Port-Bits. Wahlweise kann man auch einen bedingten Sprung aus s'AVR-Strukturen heraus mittels EXITIF – TO durchführen. Dies kann manchmal sehr vorteilhaft sein, wie wir noch sehen werden.

Als Sonderfall von Registerbits können die speziellen Statusbits des Statusregisters direkt angegeben werden, also Z (Zero), NZ (Not Zero), C (Carry), NC (no Carry), u.s.w., siehe oben.

### Register- und Port-Bits

Allgemein ausgedrückt, lautet die Syntax für Bit-Abfragen ***Register,BitStelle*** bzw. ***%Port,BitStelle***, wobei sowohl ***Register*** und ***%Port*** als auch ***BitStelle*** Symbole und ***BitStelle*** auch Zahlenwerte sein können:



| | |
|---|---|
| ***Register,BitStelle*** | ***Register*** und ***BitStelle*** dürfen sowohl Symbole sein, BitStelle auch Zahlenwerte 0-7 (dezimal, hex und binär). |
| ***%Port,BitStelle*** | ***%Port*** und ***BitStelle*** dürfen sowohl Symbole sein, BitStelle auch Zahlenwerte 0-7 dezimal, hex und binär). |

Es darf aber statt dessen nicht ein einziges Symbol für beide verwendet werden.

Der Grund hierfür liegt in der unterschiedlichen AVR-Code-Generierung für Register, Konstante, Register- und Port-Bits.

[17] Natürlich dürfen s'AVR-Anweisungen auch beliebig in Unterprogrammen verwendet werden.

Als ***Register*** bzw. ***%Port*** sind nach dem Assemblieren gemäß AVRASM-Syntax nur die AVR-Register 0-31 bzw. die Ports 0-31 und als ***BitStelle*** nur die Werte 0-7 zulässig, was - sofern möglich - teilweise durch ***s'AVR*** überprüft wird.
AVRASM überprüft - wie gewöhnlich - auf sonstige Zulässigkeiten.

Um Port-Inhalte oder Port-Bits von Ports mit einer Port-Adresse >31 abzufragen, muss der gewünschte Port vorher zwingend in eines der AVR-Register 0-31 kopiert werden. Das gilt deshalb (leider) auch für das Status-Register SREG!

Beispiele für Statusabfragen:

```
EXITIF Z          ; aktuelle s'AVR-Struktur verlassen, falls Z-Bit gesetzt ist

EXITIF NC         ; aktuelle s'AVR-Struktur verlassen, falls C-Bit nicht gesetzt ist

EXITIF !C         ; gleichbedeutend mit EXITIF NC
EXITIF NOT C      ; gleichbedeutend mit EXITIF NC
```

Die **NOT**-Anweisung oder einfach das **!**-Zeichen bieten eine Möglichkeit, Register- und Port-Bits abzufragen, die nicht gesetzt sind:

```
EXITIF !r12,5     ; Exit falls Register 12 Bit 5 nicht gesetzt ist
EXITIF NOT r12,5  ; gleichbedeutend mit EXITIF !r12,5
EXITIF !%portd,1  ; Exit falls Port D Bit 1 nicht gesetzt ist
```

Mehrfache Verwendung von NOT und ! wird korrekt bearbeitet:

```
EXITIF ! NOT !NC  ; gleichbedeutend mit EXITIF C
```

Vorsicht:
Für eine negierte logische Abfrage nicht die bitweise Negierung ~ verwenden!

Ursprünglich bestand die Idee, EXIT-Anweisungen über mehr als eine Struktur-Ebene anzubieten, jedoch wäre die Implementierung sehr komplex und unübersichtlich geworden, und ein konventioneller Assembler-Sprung (`JMP` bzw. `RJMP`) könnte bei Bedarf nahezu dieselbe Aufgabe sehr viel einfacher und übersichtlicher erfüllen.

Um jedoch optimalen Code zu erhalten und jeglichem Nachdenken über Assembler-Befehle und Zustandsbits aus dem Weg zu gehen, wurde die einzigartige **EXITIF–TO-Anweisung** geschaffen.

Ein Beispiel einer Sprungtabelle unter Verwendung von EXITIF – TO folgt im Abschnitt der LOOP-Anweisung.

Durch EXITIF–TO sind also keine der fehlerträchtigen Assembler-Vergleiche mehr nötig und man muß auch keinerlei Gedanken mehr darüber verschwenden.

EXITIF–TO muß allerdings innerhalb von ***s'AVR***-Strukturen stehen (LOOP z.B.), sonst würde diese Anweisung - wie auch EXIT und EXITIF - von ***s'AVR*** nicht erkannt werden und schließlich nach dem Compilieren möglicherweise eine Fehlermeldung wegen nicht ausgeglichenen Strukturen entstehen.

## Vergleich

Etwas allgemeiner sind die Bedingungen, die einen **Vergleich** zwischen zwei Größen durchführen (vorzeichenlose 8-Bit-Vergleiche!):

Syntax:

```
EXITIF a Vergleich b     ; verlasse aktuelle s'AVR-Struktur um eine Ebene
                         ; falls 'a Vergleich b' wahr ist
```

***Vergleich*** ist eines der folgenden Zeichen oder Zeichenfolgen:

| | |
|---|---|
| **==** | **gleich** |
| **<>** | **ungleich (>< ist nicht zulässig und erzeugt einen Fehler)** |
| **<** | **kleiner als** |
| **<=** | **kleiner gleich** |
| **>** | **größer als** |
| **>=** | **größer gleich** |

Sowohl Operand ***a*** als auch Operand ***b*** können bei einem Vergleich irgend ein AVR-Register sein, das entweder durch ein definiertes Symbol (z.B. `ArbeitsReg`) oder durch eine Register-Zahl (z.B. `R17`) beschrieben wird.

Der Operand ***b*** kann zusätzlich auch eine Konstante sein, die laut ***s'AVR***-Syntax mit dem vorangestellten **#**-Zeichen dargestellt wird. Dann werden zum Vergleichen jedoch nur die AVR-Register 16-31 unterstützt (eine AVR-Eigenschaft). Gegebenenfalls müssen statt dessen zwei AVR-Register miteinander verglichen werden.

**Achtung bei Vergleichen**:

- Fehler bezüglich unzulässigen AVR-Registern werden teilweise bereits von ***s'AVR*** erkannt, aber spätestens vom Assembler beim Assemblieren.
- Aufgrund der vorzeichenlosen 8-Bit-Vergleiche kann ein Vergleich auf > 0xff nicht durchgeführt werden.
- Bei >-Vergleich mit der Konstanten #0xff ergibt sich ein Assembler-Fehler.
- Ebenso ist (sinnvollerweise) ein Vergleich auf < 0 nicht möglich (sowohl mit einem Register mit Inhalt 0 als auch mit einer Konstanten #0).
- Für die genannten Fälle sind solche Abfragen während der Laufzeit (korrekterweise) unabhängig vom Wert des Operanden ***a*** immer unwahr.
- Der von ***s'AVR*** generierte Assembler-Code zeigt einige Feinheiten der verschiedenen Vergleichsabfragen auf (diesen hin und wieder anzuschauen, lohnt sich).
- Um zwei vorzeichenbehaftete Werte (<127) miteinander per ***s'AVR***-Bedingung zu vergleichen, kann man vor dem Vergleich beide Werte per Assembler-Befehl um einen Offset von 128 erhöhen bzw. von beiden vorher -128 abziehen[18].

[18] Bei den Registern R16..R31 kann man hierfür den Befehl SUBI Reg,-128 verwenden. Einen ADDI-Befehl gibt es bei AVR nicht.

## IF – [THEN] – ELSEIF – [THEN] – ELSE – ENDI, Verzweigungen

Die etwas komplexere IF-Struktur wird üblicherweise für Verzweigungsabfragen verwendet.

Syntax:

```
IF      Bedingung1   [THEN]   ; erste Abfrage
        BefehlsFolge
ELSEIF  Bedingung2   [THEN]   ; wahlweise 2. Abfrage
        BefehlsFolge
ELSEIF  BedingungN   [THEN]   ; wahlweise weitere Abfragen
        BefehlsFolge
ELSE                          ; optional letzte Verzweigung ohne Abfrage,
        BefehlsFolge          ; keine weiteren ELSEIF sind erlaubt
ENDI                          ; Ende der IF-Struktur
```

**IF** und das optionale **ELSEIF** (das beliebig oft wiederholt werden darf) werden gefolgt von der Bedingung, die geprüft werden soll.

**ELSE** ist bei Bedarf die letzte (sozusagen vorbelegte) Verzweigungsmöglichkeit ohne Abfrage einer Bedingung. Danach darf keine weitere **ELSEIF**-Abfrage mehr kommen, nur noch **EXIT**/**EXITIF** und schließlich **ENDI**.

Unter ***Bedingung*** versteht man genau dasselbe, wie oben unter **EXITIF** beschrieben, also entweder ein Bit-Zustand oder ein Vergleich.

Die **EXIT**- und **EXITIF**-Anweisung kann zum Verlassen einer **IF**-Struktur an beliebiger Stelle verwendet werden.

**EXITIF – TO** wäre ein möglicher nicht-strukturierter Ausstieg aus der IF-Struktur, ein bedingter Sprung an irgend eine Stelle im ***s'AVR***-Programm.

Diese Technik kann in manchen Fällen sehr zur Programmübersichtlichkeit beitragen, sollte aber mit Vorsicht angewandt werden, vor allem, wenn damit ein "Direkt"-Ausstieg aus einem Assembler-Unterprogramm erfolgt.

Beispiele:

```
IF C                    ; Sprung zu ELSEIF falls Carry-Bit nicht gesetzt
    BefehlsFolge        ; beliebige Assembler- und/oder s'AVR-Befehle
ELSEIF %pinb,3          ; falls Port B Pin 3 <> 1, springe weiter zu ELSE
    BefehlsFolge        ; weitere Assembler- und/oder s'AVR -Befehle
ELSE
    BefehlsFolge        ; diese Befehle werden ausgeführt, falls keine
                        ; der vorausgehenden Bedingungen erfüllt war
    EXITIF Z            ; verlasse IF-Struktur, falls Z-Bit gesetzt ist
    BefehlsFolge        ; weitere Assembler- und/oder s'AVR -Befehle
ENDI                    ; Ende der IF-Struktur erreicht
```

Da die vorliegende Version von **s'AVR** keine **SWITCH-CASE**-Anweisung unterstützt, kann statt dessen die **IF**-Anweisung verwendet werden, die sogar noch etwas flexibler (aber geringfügig aufwendiger) ist:

**IF** *case1*
*BefehlsFolgeCase1*
**ELSEIF** *case2*
*BefehlsFolgeCase2*
**ELSEIF** *caseN*
*BefehlsFolgeCaseN*
**ELSE**
*BefehlsFolgeDefault*
**ENDI**

*case1*, *case2* und *caseN* sind irgendwelche der oben beschriebenen Bedingungen. Natürlich können auch hier in den diversen *BefehlsFolgen* weitere **s'AVR**-Anweisungen einschließlich EXIT und EXITIF enthalten sein.

Wegen jeweils völlig neuen Abfragen für jeden einzelnen Fall wird im Vergleich zu einer echten SWITCH-CASE-Abfrage bei einer IF-basierenden Abfrage etwas mehr Code erzeugt. Allerdings sind bei der IF-Struktur die Bedingungen voneinander unabhängig.

Eine weitere Möglichkeit statt dessen wären aufeinanderfolgende EXITIF–TO-Anweisungen, wie im nächsten Abschnitt gezeigt wird.

## LOOP – ENDL, Endlosschleife

Das ist die einfachste Programmstruktur, die eine Schleife solange wiederholt, bis eine Unterbrechung eintritt. Eine Unterbrechung kann irgend eine EXIT- oder EXITIF-Anweisung, ein Assembler-Sprung aus dieser Struktur heraus, ein AVR-Interrupt oder ein Reset sein. In jedem anderen Falle würde die Schleife niemals enden.

LOOP/ENDL schließt häufig das Hauptprogramm einer Mikrocontroller-Applikation ein (nach den üblichen Initialisierungs-Routinen). Dann kann dieses - bedingt durch die maximal mögliche RJMP-Sprungweite von 2k-Worten - bei **s'AVR-LITE** auch nur maximal 2k-Worte groß sein. Abhilfe schafft ggf. eine Hauptschleife ohne LOOP/ENDL per JMP-Befehl (falls vom verwendeten AVR-µC unterstützt). Das sollte aber eher selten vorkommen, da aufgerufene Unterprogramme immer außerhalb einer solchen LOOP-Schleife liegen und deshalb die Sprungweite nicht nachteilig beeinflussen.

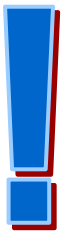
Syntax:

```
LOOP                    ; Endlosschleife
    BefehlsFolge        ; Verlassen der LOOP-Schleife nur mittels
                        ; EXIT, EXITIF, RJMP, RET, Interrupt
                        ; oder AVR-Reset
ENDL                    ; Ende der LOOP-Schleife
```

Beispiele:

```
LOOP
        BefehlsFolge
        EXITIF C                ; Ausstieg, falls das Carry gesetzt ist
        BefehlsFolge
        LOOP                    ; verschachtelte LOOP-Schleife
                BefehlsFolge
                EXITIF !r16,1   ; Ausstieg, falls Register 16 Bit 1 nicht gesetzt
                BefehlsFolge
        ENDL                    ; Ende der 2. LOOP-Schleife
        BefehlsFolge
ENDL                            ; Ende der 1. LOOP-Schleife
-------------------------------------------------------------------------------------------
LOOP                            ; das kleinstmögliche S'AVR-Programm!
ENDL                            ; warten, bis Interrupt oder Reset
```

Da gibt es eine kleine Falle beim Einsatz von **EXIT**. Folgendes Beispiel sollte nicht verwendet werden[19], denn diese IF-Struktur würde überhaupt keinen Effekt zeigen:



```
LOOP
        BefehlsFolge
        IF Z        EXIT        ; dieses EXIT verläßt nur die IF-Struktur,
        ENDI                    ; nicht jedoch die LOOP-Struktur!!
        BefehlsFolge
ENDL                            ; Endlosschleife, falls kein Interrupt oder Reset
```

Diese scheinbar verzwickte Situation wird ganz einfach mittels EXITIF gelöst:

```
LOOP
        BefehlsFolge
        EXITIF Z                ; nun wird LOOP verlassen, falls Z-Bit gesetzt ist
        BefehlsFolge
ENDL
```

Da die Verwendung von EXITIF-TO so gut in die Beschreibung der LOOP-Struktur paßt, soll es hier als codesparendes Beispiel für eine Sprungtabelle[20] gezeigt werden:

```
LOOP                                    ; irgend eine S'AVR –Struktur wird benötigt
                                        ; um EXITIF – TO benützen zu können
        EXITIF  Beding1  TO  Adresse1
        EXITIF  Beding2  TO  Adresse2
        EXITIF  BedingN  TO  AdresseN
        RJMP    AdrDefault              ; Assembler-Sprung nach AdrDefault
ENDL                                    ; in diesem Beispiel wird die Schleife
                                        ; nicht nochmals durchlaufen
```

**Anmerkung**: Die Sprungziele *Adresse1/2/N* werden irgendwo im **S'AVR**-Sourcecode platziert sein, diesen dürfen aber keineswegs irgend welche **S'AVR**-Anweisungen in derselben Zeile folgen, da sonst diese Anweisungen nicht erkannt werden und **S'AVR** schließlich nicht ausgeglichene Strukturen feststellen würde.

[19] Obwohl die Syntax dieses sehr kompakten Programmes korrekt wäre.
[20] Im Unterschied zu einer vereinfachten Sprungtabelle mit aufeinanderfolgenden Ganzzahlen.

## WHILE – ENDW, Abfrage am Anfang der Schleife

Bevor die WHILE-Schleife durchlaufen wird, muß eine bestimmte Bedingung erfüllt sein. Nach dem Durchlaufen wird wieder mit derselben WHILE-Abfrage begonnen und zwar solange, bis die Bedingung nicht mehr erfüllt ist. Dann wird die gesamte Schleife übersprungen. Es kann also sein, daß die WHILE-Schleife nie durchlaufen wird und zwar dann, wenn die Bedingung bereits beim ersten Mal nicht erfüllt ist.

Syntax:

```
WHILE  Bedingung          ; Abfrage zu Beginn der Schleife
       BefehlsFolge
ENDW                      ; Ende der WHILE-Schleife
```

Beispiel:

```
       in     r17,portd      ; kopiere Port D nach Register 17
WHILE r17 <> #0              ; solange Register 17 <> Null
       out    portb,r17      ; kopiere Register 17 nach Port B
       BefehlsFolge          ; weitere Assembler- oder s'AVR -Befehle
       in     r17,portd      ; kopiere Port D nach Register 17
ENDW
```

Beim Analysieren des erzeugten Assembler-Codes erkennt man, daß **s'AVR** den **s'AVR**-Sourcecode bezüglich Konstanten mit dem Wert Null[21] überprüft. In diesen Fällen wird ein optimierter Code unter Verwendung des TST-Befehles erzeugt, aber nur falls auf == oder <> abgefragt wird (denn das klappt auch mit Register 0 bis 15):

```
       in     r17,portd      ; kopiere Port D nach Register 17
       ;01// WHILE r17 <> #0  ; solange Register 17 <> Null
_L1:
       TST    r17
       BRNE   _L2
       RJMP   _L3
_L2:
       out    portb,r17      ; kopiere Register 17 nach Port B
       BefehlsFolge          ; weitere Assembler- oder s'AVR -Befehle
       in     r17,portd      ; kopiere Port D nach Register 17
       ;01// ENDW
       RJMP   _L1
_L3:
```

**Anmerkungen**:

- In diesem Beispiel wird die **s'AVR**-Option ”keep **s'AVR** statements in the output file” verwendet: Nach dem “;” (Kommentarzeichen) wird zunächst die aktuelle Strukturebene “01” aufgeführt, gefolgt von “//” (etwas Markantem) und der ursprünglichen **s'AVR**-Anweisung einschließlich eventuellem Kommentar der **s'AVR**-Quellzeile.

- Durch das Anzeigen der Strukturebene mittels ";01//" etc. wird das Debugging und die Fehlersuche etwas einfacher, insbesondere bei Verschachtelung von mehreren gleichartigen **s'AVR**-Anweisungen (mit jeweils einer eigenen Strukturebene).

[21] Wert Null, jedoch ausschließlich in der Form #0, #$0, #$00, #0x0, #0x00, #0b0, #0b00, #0b000 und #0b0000.

### Einwand?

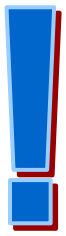
Beim näheren Betrachten des erzeugten Assembler-Codes könnte man einwenden, dass man den RJMP-Befehl an dieser Stelle ganz einsparen kann, indem man statt

```
BRNE   _L2
RJMP   _L3
```

schlauer wie folgt programmiert (dann vom Precompiler erzeugt):

```
BREQ   _L3
```

Das hätte aber den schwerwiegenden Nachteil, dass bei einem geringfügig komplexeren Programm die Sprungweite für BREQ bis zur Adresse _L3: größer als die bei AVR maximal zulässige Sprungweite der Branch-Befehle von 63 Worten wird.



Deshalb verwendet **S'AVR** grundsätzlich Branch-Befehle und (sofern verfügbar) Skip-Befehle lediglich um den RJMP-Befehl zu überspringen, der schließlich für den "weiten" Sprung zuständig ist (im Beispiel bis zum Ende der obigen WHILE-Schleife).

Anstatt dem RJMP-Befehl wird mangels Kenntnis der absoluten Adressen bzw. der tatsächlichen Sprungweite und wegen der genannten Einschränkung der Branch-Befehle bewusst kein Branch-Befehl verwendet, auch wenn es meist einen Befehl einsparen würde (siehe obiges Beispiel).

Und somit wird die reichlich geringe Sprungweite der AVR-Branch-Befehle im erzeugten Assembler-Code nie ein Problem, nur selten das RJMP-Limit, nämlich falls eine **S'AVR**-Struktur einen zu großen Adressbereich von mehr als 2k-Worten überstreicht.
Dieser Kompromiss bezüglich RJMP statt einem der Branch-Befehle ist nicht wirklich optimal, aber dafür frustfrei.

Derselbe Sachverhalt gilt auch bei anderen Abfragen wie ELSEIF und EXITIF, sofern die Abfragen nicht am Ende einer **S'AVR**-Struktur stehen, siehe REPEAT - UNTIL.

Hinweis: **S'AVR** 2.x wird sowohl den geschilderten "schmerzfreien" Code mit überwiegend BRcc/RJMP als auch einen effizienten Code mit überwiegend BRcc erzeugen. ☺

### REPEAT – UNTIL, Abfrage am Ende der Schleife

In diesem Fall wird die Schleife mindestens einmal durchlaufen bevor am Ende nach einer Bedingung abgefragt wird. Falls diese Bedingung nicht erfüllt ist, wird die Schleife wiederholt und zwar solange, bis die Bedingung schließlich erfüllt ist.

Syntax:

```
REPEAT                      ; Beginn der Schleife
       BefehlsFolge
UNTIL  Bedingung            ; Abfrage am Ende der Schleife
```

Beispiel:

```
REPEAT                          ; Schleife ...
       rcall get_character      ; ... um Zeichen nach 'char' zu lesen
UNTIL char <> #blank            ; jedoch alle Leerzeichen überspringen
```

Erzeugter Code:

```
       ;01// REPEAT             ; Schleife ...
_L1:
       rcall get_character      ; ... um Zeichen nach 'char' zu lesen
       ;01// UNTIL char <> #blank ; jedoch alle Leerzeichen überspringen
       CPI    char,blank
       BRNE   _L2
       RJMP   _L1
_L2:
```

**Tipp!**

**Ein interessanter Programmiertipp:**

Die REPEAT-UNTIL-Anweisung spart einen AVR-Befehl im Vergleich zur WHILE-ENDW-Anweisung, wenn – wie bei einer Warteschleife – keine weiteren Befehle innerhalb der Struktur stehen:

```
WHILE %pinb,3                   ; warten solange Port B Pin 3 HIGH ist
ENDW

REPEAT                          ; warten ...
UNTIL NOT %pinb,3               ; ... bis Port B Pin 3 LOW ist
```

Und man hat beim Programmieren eine Zeile mehr Zeit zum Überlegen, was hinter UNTIL abgefragt werden soll ... ☺.

Erzeugter AVR-Assembler-Code:

```
       ;01// WHILE %pinb,3      ; warten solange Port B Pin 3 HIGH ist
_L1:
       SBIS   pinb,3
       RJMP   _L3
_L2:
       ;01// ENDW
       RJMP   _L1
_L3:

       ;01// REPEAT             ; warten ...
_L4:
       ;01// UNTIL NOT %pinb,3  ; ... bis Port B Pin 3 LOW ist
       SBIC   pinb,3
       RJMP   _L4
_L5:
```

Nachdem wir nun bedingte Verzweigungen, Schleifenabfragen zu Beginn und am Ende der Schleife oder sogar Endlosschleifen haben, fehlen uns noch Schleifen, die für eine bestimmte Anzahl durchlaufen werden.

## FOR – ENDF, Schleife mit Schleifenzähler und Schrittweite -1

Syntax:

```
FOR   RegisterZuweisung          ; der Schleifenzähler ist ein AVR-Register
      BefehlsFolge
ENDF                             ; dekrementiere und überprüfe den
                                 ; Schleifenzähler auf den Wert Null
```

---

*RegisterZuweisung* kann für den Schleifenzähler wie folgt ausgeführt werden:

```
Register   := Register2          ; Schleifenzähler mit Register2 laden
Register   := #Konstante         ; Schleifenzähler mit einer Konstanten laden
Register                         ; der Schleifenzähler ist bereits initialisiert
```

***Register*** bedeutet also ein AVR-Register. D.h. der Schleifenzähler kann mit dem Inhalt eines anderen Registers ***Register2*** oder einer Konstanten ***#Konstante*** initialisiert werden. Zum Laden mit einer Konstanten werden jedoch für ***Register*** nur die AVR-Register R16..R31 unterstützt (eine AVR-Eigenschaft), ansonsten alle AVR-Register R0..R31.

Oder das spezifizierte Schleifenregister kann bei Erreichen der FOR-Anweisung bereits initialisiert sein (***Register*** ohne Zuweisungszeichen).

Falls die FOR-Schrittweite ungleich -1 sein soll, müsste man eine REPEAT-UNTIL-Struktur verwenden, wobei das Initialisieren des Schleifenzählers durch Assembler-Befehle vor der REPEAT-Anweisung und die Berechnung der Schleifenschritte vor der UNTIL-Anweisung erfolgt.

**Anmerkungen**:

Hat der Schleifenzähler den Wert Null beim Starten der FOR-Schleife, so wird die Schleife aufgrund des 8 Bit breiten Schleifenregisters 256 mal durchlaufen und zwar weil die Abfrage auf den Wert Null erst nach dem Dekrementieren am Ende der durchlaufenen Schleife erfolgt.

***s'AVR*** akzeptiert zum Zuweisen statt #0 auch #256 als (einzige) Nicht-8-Bit-Konstante, die von ***s'AVR*** in #0 übersetzt[22] wird, um den Schleifenzähler damit zu initialisieren.

***s'AVR*** benützt zum Unterscheiden das doppelte Gleichheitszeichen "==" für Vergleiche (wie AVRASM2) und ":=" um FOR-Schleifenregister zu initialisieren.

[22] AVRASM2 würde an dieser Stelle eine Konstante 256 reklamieren. Andere Assembler übernehmen teilweise Werte modulo 256.

Beispiele:

```
FOR     count := #3             ; AVR-Register 'count' mit Wert 3 initialisiert
        rcall blink_led         ; Unterprogramm wird genau 3 mal durchlaufen
ENDF
```

```
FOR     loop := #loops          ; diese Anweisung erzeugt einige Fehler,
                                ; da LOOP ein s'AVR -Schlüsselwort ist!
        BefehlsFolge
ENDF
```

```
FOR     lp_count                ; Register 'lp_count' ist bereits initialisiert
                                ; bevor die FOR-Schleife erreicht wird
        BefehlsFolge
ENDF
```

Die Schleifenzahl Null kann sehr praktisch (aber auch eine Falle) sein:

```
FOR     lp_cnt := #0            ; die Schleifenzahl ist nicht Null, sondern ...
        BefehlsFolge
ENDF                            ; ... 256, da lp_cnt am Ende der Schleife erst
                                ; nach dem Dekrementieren überprüft wird
```

Erzeugter Code (gezeigt ohne **s'AVR**-Anweisungen und Kommentare):

```
        CLR     lp_cnt
_L1:
        BefehlsFolge
        DEC     lp_cnt
        BREQ    _L2
        RJMP    _L1
_L2:
```

Wie weiter oben angemerkt, hätte man deshalb für besseres Verständnis und mit genau demselben Ergebnis eben so gut **lp_cnt := #256** nehmen können (#-Zeichen!):

```
FOR     lp_cnt := #256          ; die Schleifenzahl ist tatsächlich 256 ...
        BefehlsFolge
ENDF                            ; ..., da s'AVR zum Initialisieren den Wert #0
                                ; (bzw. CLR) verwendet hat und die Abfrage
                                ; am Ende der FOR-Schleife erfolgt
```

Deshalb WICHTIG:

Falls der aktuelle Wert von lp_cnt innerhalb von *BefehlsFolge* verwendet wird, muss man eben daran denken, dass der erste Wert 0 ist, gefolgt von 255, 254, 253 etc.

## *s'AVR*-Optionen

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

Bei dieser voreingestellten Option bleiben die *s'AVR*–Anweisungen in der erzeugten Ausgabedatei als Kommentar erhalten. Beim Debuggen mittels Atmel®-Studio oder zur Dokumentation ist diese Option sehr hilfreich. Falls jedoch - warum auch immer - die reinen von *s'AVR* erzeugten Assembler-Befehle ausreichen, muß diese Option abgewählt werden. Entsprechend weniger Zeilen werden in der Ausgabedatei erzeugt.

### Option "keep warnings in the output file"

Diese voreingestellte Option belässt Warnhinweise von *s'AVR* in der Ausgabe-Datei.

### Option "keep hints in the output file"

Diese voreingestellte Option belässt Tipps von *s'AVR* in der Ausgabe-Datei.

### Kollision mit bedingter Assemblierung oder Assembler-Direktiven

Ein weiteres (aber geringfügiges) Problem ist unter Umständen das korrekte Zusammenspiel zwischen *s'AVR* und dem nachgeschalteten Assembler. Die vorliegende *s'AVR*-Version unterstützt den AVRASM2 und kompatible Assembler.

Falls Assembler-Direktiven oder bedingte Assemblierung nicht wie die üblichen `.IF`, `.ELSE` bzw. `#IF`, `#ELSE` etc. lauten, könnte es u.U. eine Kollision mit *s'AVR* - Anweisungen zur strukturierten Programmierung geben (was mit AVRASM2 nicht der Fall sein sollte).

Um eine solche Kollision zu vermeiden, wird im *s'AVR*-Quellprogramm bei Bedarf ganz einfach ein **Fragezeichen** vor die betreffende Assembler-Anweisung gestellt.

*s'AVR* überprüft das Quellprogramm auf Fragezeichen als erstes druckbares Zeichen in einer Zeile, entfernt dieses und gibt den Rest der Zeile unverändert in die Ausgabedatei. Der Assembler kann dann die Anweisung wie gewünscht interpretieren. Dies ist ein sehr einfacher aber wirkungsvoller Trick[23].

Beispiel für bedingte Assemblierung innerhalb eines *s'AVR*–Sourcecodes, die ohne das Fragezeichen mit *s'AVR*–Anweisungen kollidieren könnte:

```
?IF debug=1                    ; das '?' wird von s'AVR automatisch entfernt
        BefehlsFolge
?ELSE                          ; das '?' wird von s'AVR automatisch entfernt
        BefehlsFolge
ENDIF                          ; diese Zeile stört wegen ENDI nicht, dennoch
                               ; dürfte trotzdem ein '?' vor ENDIF stehen
```

[23] Das Fragezeichen kann nicht als "Conditional Operator" (ab AVRASM 2.1) fehlinterpretiert werden, da dieser nicht als erstes druckbares Zeichen einer Zeile steht.

Normalerweise (wenn überhaupt) kommen kollidierende bedingte Assemblierungsanweisungen im Vergleich zu *s'AVR*-Anweisungen nicht so häufig vor, so daß die Eingabe der zusätzlichen Fragezeichen (sofern überhaupt nötig) keine besondere Mühe bereiten dürfte.

Falls diese Fragezeichen jedoch versehentlich vergessen werden (aber ggf. nötig wären), erkennt dies *s'AVR* daran, dass dann Anfangs- und Endpaare dieser Anweisungen nicht zusammenpassen. *s'AVR* würde dann ggf. nicht ausgeglichene Programmstrukturen erkennen und reklamieren.

| Nicht-AVRASM2-Assembler: (bedingte Assemblierung) | | *s'AVR:* (strukturierte Anweisung) |
|---|---|---|
| `IF` | gleiche Syntax | **IF** |
| | kein Äquivalent | **ELSEIF** |
| `ELSE` | gleiche Syntax | **ELSE** |
| `ENDIF` | ungleiche Syntax | **ENDI** |

Im Allgemeinen geben *s'AVR* –Fehlermeldungen (in der Ausgabedatei!) sehr gute Hinweise um die Fehlerursache schnell beseitigen zu können.

Praktischerweise lässt man hierfür unter Studio die Zeilennummern mindestens in der *s'AVR*-Quelldatei anzeigen (siehe Beispiel).

Und *s'AVR* ist überdies so entworfen, daß sich eine Fehlererkennung selbst synchronisiert und nicht weitere unerwartete Fehlermeldungen erzeugt, die dann mehr irritieren als helfen würden.

## Konflikte mit Assembler-Macros

Da AVR-Macro-Direktiven mit einem Punkt beginnen, sollte es keine Konflikte mit AVR-Makros geben. *s'AVR*-Anweisungen sollten aber wegen den erzeugten globalen Adressen nicht innerhalb von Macros verwendet werden.

*s'AVR* wird AVR-Macros wie alle Assembler-Zeilen ignorieren bzw. unverändert in die Ausgabedatei übergeben.

## Kommentare

Kommentare mittels **;**-Zeichen werden von *s'AVR* als solche behandelt.

Kommentare zwischen /* und */ über mehrere Zeilen (wie von AVRASM2 praktischerweise ebenfalls unterstützt) werden von *s'AVR* jedoch nicht als Kommentar erkannt, wodurch *s'AVR*-Anweisungen innerhalb solcher Kommentarbereiche trotzdem ausgewertet und compiliert werden (ggf. also auch mit Fehler-, Warn- und Tipp-Hinweisen).

Das schadet aber nicht weiter, da es weiterhin Kommentare zwischen /* und */ bleiben. Es werden lediglich einige überflüssige Labels erzeugt.

Wichtig ist dann jedoch, dass in Zeilen mit /* und */ keine *s'AVR*-Anweisungen stehen, die dann nicht erfasst werden, wodurch es zu unausgeglichenen Strukturen (nebst Fehlermeldung) kommt, falls weitere *s'AVR*-Anweisungen an anderer Stelle innerhalb des Kommentarbereichs stehen.

## Einige grundsätzliche Syntax-Regeln

Wie bereits vermerkt, wurde **s'AVR** entworfen, um zusammen mit AVRASM2 und seiner zugehörigen Entwicklungsumgebung Atmel®-Studio verwendet zu werden. Deshalb gelten für Symbole, Zahlen und so weiter dieselben Vorschriften:

**Identifier** (einschließlich Adressen, Symbole, Register etc.) beginnen mit einem Buchstaben oder einem Unterstrich und können von weiteren Buchstaben, Unterstrichen und Ziffern gefolgt werden. Die von **s'AVR** erzeugten Adressen lauten alle **_Lxxxx** und dürfen nicht anderwetig verwendet worden sein.

Ein Komma zwischen zwei Indentifiern ist zwingend (und zwar ohne Leerzeichen dazwischen), wenn **Register-Bits** oder **Ports-Bits** adressiert werden sollen.

**Zahlen** können dezimal, hexadezimal ($ff oder 0xff) und binär (0b1111_1111) dargestellt sein (letztere auch mit Unterstrichen für eine übersichtlichere Darstellung).

## Ergänzend gilt für **s'AVR**:

Da bei **s'AVR-LITE** alle Anweisungen 8-Bit-Operationen sind, werden Werte >255 als Fehler[24] angezeigt, und zwar sowohl bei Dezimal- und Hexadezimal- als auch bei Binärwerten.

**Texte** werden von **s'AVR** unter Verwendung von einfachen Hochkommata erkannt (‘abc-!?...’), jedoch werden für Vergleiche nur einzelne Byte-Zeichen benötigt ('a'). **s'AVR** erlaubt einige Freiheiten, die schließlich der Assembler auf Gültigkeit überprüfen muß.

**Ports** werden mittels vorangestelltem **%** gekennzeichnet.

**Konstante** (immediate literals) benötigen ein vorangestelltes **#** und haben die Formate #999, #$ff, #0xff, #0b1111_1111 und #’x’.

Diese Formen werden als Wert Null erkannt: #0, #$0, #$00, #0x0, #0x00, #0b0, #0b00, #0b000, #0b0000. Sie resultieren teilweise in optimiertem Assembler-Code.

Andere Formate werden u.U. von AVRASM und anderen AVR-Assemblern erkannt, nicht jedoch von **s'AVR** (und können dann zusammen mit Assembler-Befehlen verwendet werden).

Diese Regeln sind vorgeschrieben im Zusammenhang mit **s'AVR**-Anweisungen. Falls ein AVR-Assembler mit abweichender Syntax verwendet wird, können zwar die Assembler-Anweisungen anders lauten, ohne daß **s'AVR** davon berührt wäre, jedoch muß der von **s'AVR** generierte Code vom Assembler ebenfalls verstanden werden.

**s'AVR-LITE** generiert ausschließlich relative Sprünge per RJMP-Befehl und Branch- oder Skip-Befehle zum Überspringen der RJMP-Befehle, so dass **s'AVR-LITE** auch für ATtiny verwendet werden kann. **s'AVR** verwendet keine Macros sondern erzeugt nur native AVR-Assembler-Befehle.

[24] Nur zum Initialisieren einer FOR-Schleife wird auch #256 akzeptiert und als #0 in den erzeugten Code übernommen, wodurch die FOR-Schleife korrekt 256x durchlaufen wird.

# Command-Line-Interface

## Command-Line-Aufruf

***s'AVR*-LITE** unterstützt **Command-Line-Aufrufe** nach DOS-Manier, d.h. ***s'AVR*** kann (insbesondere von fremden Programmen wie z.B. Editoren und Debugger oder auch über das Windows®-Menü "Start - Ausführen...") mit Parametern gestartet werden, ohne ***s'AVR*** über das ***s'AVR***-Programm-Fenster bedienen zu müssen.

Bei Command-Line-Aufrufen gelten folgende **Regeln**:

- Die einzelnen Parameter werden mittels vorangestelltem Schrägstrich (/) an ***s'AVR*** übergeben.
- Der **erste** Parameter **muss** eine gültige und vorhandene ***s'AVR***-Sourcecode-Datei mit der Erweiterung "`.s`" oder "`.savr`" sein, sonst wird im WINDOWS®-Mode gestartet.
- Nach dem Dateinamen können - jeweils mit Schrägstrich getrennt - beliebig viele Parameter folgen, siehe Command-Line-Syntax.
- Bei sich widersprechenden Parametern gilt der jeweils letztgenannte Parameter.
- Nicht definierte bzw. nicht erkannte Parameter werden ignoriert.
- Fehlermeldungen erfolgen nur in der Ausgabedatei (`*.asm`).

Groß-/Kleinschreibung und Leerzeichen werden großzügig toleriert und sind auch innerhalb von Dateinamen zugelassen. Die Parameter-Schlüsselworte können (wie generell bei ***s'AVR***) ebenfalls in Groß- und Kleinschrift geschrieben sein.

## Command-Line-Syntax:

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

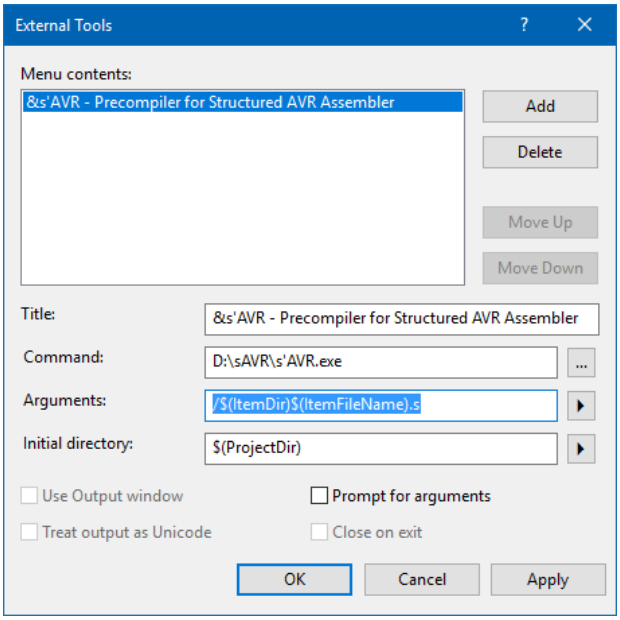
| | |
|---|---|
| **filename.s**<br>**filename.savr** | gültiger ***s'AVR***-Sourcecode-Dateiname (benötigt Erweiterung ".s" **oder** ".savr") |
| **nosavr** | die ***s'AVR***-Anweisungen werden in der Ausgabedatei unterdrückt |
| **keepsavr** | die ***s'AVR***-Anweisungen werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt) |
| **nowarn** | die ***s'AVR***-Warnhinweise werden in der Ausgabedatei unterdrückt |
| **keepwarn** | die ***s'AVR***-Warnhinweise werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt) |
| **nohint** | die ***s'AVR***-Tipps werden in der Ausgabedatei unterdrückt |
| **keephint** | die ***s'AVR***-Tipps werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt) |
| **listclp** | erzeugt außer der Assembler-Ausgabedatei auch eine Kontrolldatei namens "**sAVR_listclp.txt**", die alle gefundenen Parameter und die zugehörigen Flags auflistet (normalerweise im selben Verzeichnis, von dem aus ***s'AVR*** gestartet wurde). |

## Command-Line-Aufruf per Atmel®-Studio

Eingangs wurde die Einbindung von s'AVR so beschrieben, dass bei Aufruf des eingebundenen s'AVR-Tools das s'AVR-Programmfenster erscheint und man so beim ersten Aufruf das jeweilige Quellprogramm '.s' im zugehörigen Projektverzeichnis auswählen und bei jedem Compiler-Lauf ggf. angebotene Optionen erneut einzeln abwählen kann. Das s'AVR-Programmfenster kann bei dieser Methode nach dem ersten Aufruf im Hintergrund geöffnet bleiben.

Falls man beim Einrichten von s'AVR als externes Tool auch noch die 'Arguments' ausfüllt, lassen sich an dieser Stelle Quelldateiname und ggf. s'AVR-Optionen per Kommandozeilenaufruf an s'AVR übergeben, allerdings ohne dass man dann noch eingreifen könnte.

Bei voreingestellten s'AVR-Optionen würde der Arguments-Eintrag wie folgt aussehen:




Die einzelnen Arguments-Parameter erfolgen möglichst alle ohne jegliche Leerzeichen dazwischen. Ansonsten übergibt Atmel®-Studio die Parameter einzeln in doppelten Hochkommata[25].

'Prompt for Arguments' bleibt weiterhin deaktiviert[26].

Wichtig ist, dass die Dateierweiterung '.s' mit angegeben wird, denn sonst kann es sein, dass u.U. (je nach gerade aktiver Datei) '.asm' statt '.s' an s'AVR übergeben und damit die falsche Quelldatei kompiliert wird.

Nach dem richtigen Einrichten unter Atmel®-Studio genügt zum Starten von s'AVR ein einziger Mausklick im Menü-Punkt 'Tools' und '.asm' wird blitzschnell aktualisiert (erkennbar an Datum und Uhrzeit), ohne dass das s'AVR-Programmfenster sichtbar wird oder dieses gar bedient werden müsste - vorausgesetzt, man hat das Quellprogramm '.s' nach eventuellen Änderungen vor dem Aufruf auch abgespeichert!

[25] s'AVR entfernt ab Version 1.04 doppelte Hochkommata automatisch aus der Kommandozeile.
[26] Zum Überprüfen der Parameterübergabe auf Richtigkeit kann man diese Option vorübergehend aktivieren.

## Ausblick

Die durch den `RJMP`-Befehl bedingte Einschränkung der LITE-Version wird bei der Vollversion entfallen (verwendet optional `JMP`-Befehle). Auch wird die Vollversion 16-Bit-Operationen mit einigen ***s'AVR***-Anweisungen erlauben.

***s'AVR*** 2.0 ist bereits in Vorbereitung und wird auch in der LITE-Version sowohl den hier im Handbuch beschriebenen "schmerzfreien" Code mit überwiegend `BRcc/RJMP` als auch optional einen sehr effizienten Code mit überwiegend `BRcc` erzeugen, der einen Vergleich mit einem reinen AVR-Guru-Assembler-Programm nicht scheuen muss.

Abhängig von den Wünschen der ***s'AVR***-Anwender können weitere ***s'AVR***-Versionen entstehen, die möglicherweise einige zusätzliche Funktionen erhalten:

- **AND–OR**-Kombinationen innerhalb von Bedingungen
- **IN**-Listen innerhalb von Bedingungen (wie bei PASCAL/DELPHI)
- verbesserte **FOR**-Schleife (mit Schrittweite)
- kombinierte **WHILE-UNTIL**-Anweisung (Abfrage am Anfang und Ende der Schleife)
- **SWITCH-CASE**-Anweisung
- **ON–GOTO**-Anweisung
- Weitere Vorschläge, die ***s'AVR*** verbessern

Auch wenn ***s'AVR*** sorgfältig auf richtige Funktion überprüft[27] wurde, lassen sich Programmierfehler[28] nie ausschließen.

Fehlermitteilungen sind per eMail immer willkommen (möglichst mit einem angehängten Source-Code-Schnipsel als Textdatei, das den Fehler verursacht).

Kommentare und Anregungen zu ***s'AVR*** sind natürlich ebenso willkommen.

Und nun: Viel Spaß und Erfolg mit Atmel® AVR® kombiniert mit ***s'AVR***!

(Ein Stichwortverzeichnis folgt bei Gelegenheit.)

[27] Eines der ***s'AVR***-Testprogramme für einen ATmega328P mit angeschlossener LCD-Anzeige besteht aus 743 ***s'AVR***- und Assembler-Zeilen.
Daraus wurden von ***s'AVR*** mit der Option ohne ***s'AVR***-Statements 1.087 "flache" Assembler-Zeilen erzeugt, also beinahe 350 Assembler-Zeilen mit strukturierter ***s'AVR***-Programmierung gespart - bei deutlich weniger Risiko von Programmierfehlern etc.
Mit den ***s'AVR***-Statements sind es dagegen 1.284 Zeilen in der Ausgabedatei.

[28] ***s'AVR*** selbst ist mit einem in die Tage gekommenen Delphi 5 erstellt (also in Pascal programmiert), das mit einem kleinen Tweak bei der veralteten (seit Windows® 7 nicht mehr unterstützten) Hilfefunktion immer noch sehr gut unter Windows® 10 läuft.