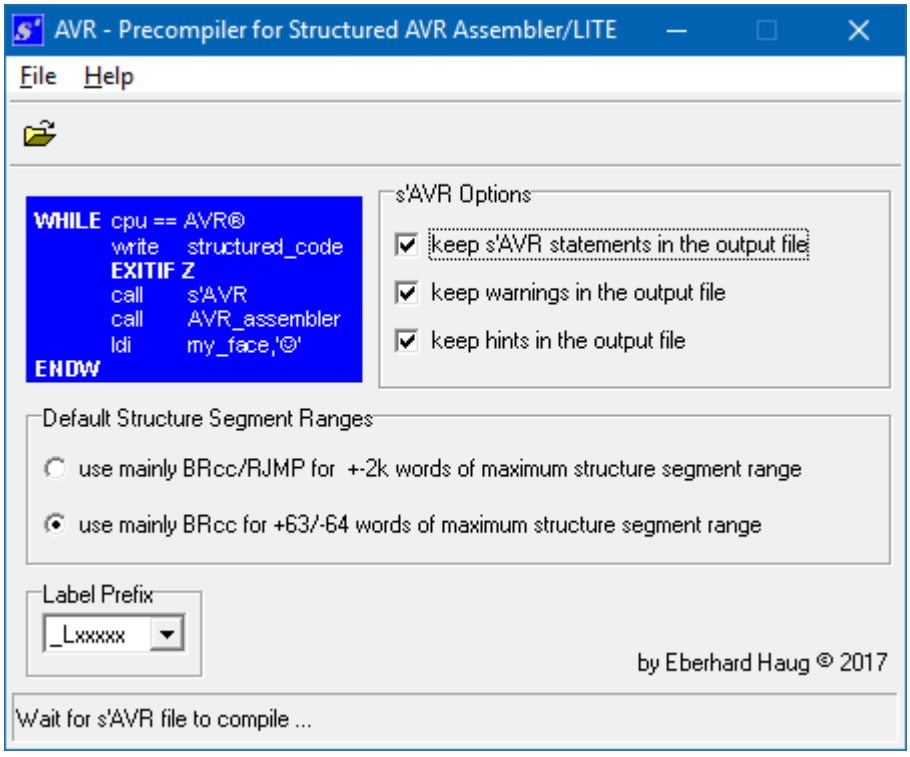
s'AVR-LITE

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 AVR-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.

Je nach gewählter Sprungweiten-Option (ab ***s'AVR*** 2.x) beeinflussen diese ***s'AVR***-Anweisungen die Code-Effizienz des Programmes im Vergleich zu reiner Assembler-Programmierung nur sehr wenig[2] oder gar nicht. 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 bei Bedarf 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).
[3] Mit dieser Methode wurden (neben weiteren detaillierteren Untersuchungen) erste Tests auf Richtigkeit mit s'AVR durchgeführt. Natürlich ist es sehr viel einfacher, AVR-Programme von Anfang an als s'AVR-Quellprogramm zu schreiben.

Wofür **s'AVR** gedacht ist

s'AVR (lauffähig unter allen aktuellen Windows®-Versionen und Linux[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.

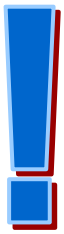
Zwischen den **s'AVR**-Anweisungen stehen die eigentlichen AVR-Assembler-Befehle (typischerweise für die eigentliche Datenmanipulation und Unterprogrammaufrufe), die vom **s'AVR**-Precompiler unangetastet bleiben.

Natürlich können zur Programmverschachtelung beliebig viele weitere **s'AVR**-Anweisungen und AVR-Assembler-Befehle dazwischen stehen, solange dies der Programmspeicher des jeweiligen verwendeten AVR-Mikrocontrollers zulässt.

Das von **s'AVR** erzeugte AVR-Assembler-Quellprogramm[5] kann mit den vorhandenen Entwicklungswerkzeugen zum Simulieren und Debuggen verwendet werden und muss von irgendeinem kompatiblen Assembler in den endgültigen AVR-Objektcode übersetzt 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 von **s'AVR** erzeugte Warnhinweise und Tipps in der Ausgabedatei unterdrücken.

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 Register und den Arbeitsspeicher des AVR!

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 gleiche 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. Unter Linux lässt sich s'AVR mittels WINE aufrufen und anwenden.
[5] Bei s'AVR-Lite werden ausschließlich die von den älteren ATtiny unterstützten Befehle verwendet, also RJMP statt JMP.
[6] Genaugenommen wird natürlich der Befehlszähler und ggf. das Statusregister verändert.
[7] Strukta (gab es für 8080/8085 und Z80) hatte bei LOOP den Stack verwendet, so dass dort 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 in deutscher und englischer Sprache im PDF-Format zur Verfügung.
Das Hilfe-Menü des Programms zeigt in englischer Sprache einen kompakten Überblick über die Syntax der *s'AVR*-Steueranweisungen, der Anweisungen für Sprungweiten, der zulässigen Vergleiche/Status-Bits und der Kommandozeilen-Parameter.

s'AVR im Einsatz

Zunächst wird das Quellprogramm der Applikation unter Verwendung eines bevorzugten Texteditors erstellt, natürlich 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 per üblichem 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 auch in der Ausgabedatei erfasst sind.

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

Ab s'AVR 2.1 wird per variablem Label-Prefix[10] von `_Axxxx` bis `_Zxxxx` ermöglicht, bis zu 26 verschiedene s'AVR-Quellprogramme zu einem gemeinsamen Programm zu verlinken, ohne dass es Adresskollisionen geben würde.

Falls Fehler bzw. Warnhinweise beim Compilieren aufgetreten sind, erscheinen sie an entsprechender Stelle in der Ausgabedatei unter Bezug auf Zeilennummern der s'AVR-Quelldatei. Der Fehlertext erscheint in der Ausgabedatei nicht als Kommentar, sondern als .ERROR-Zeile nebst Fehlertext, den AVRASM2 anzeigt und das Assemblieren stoppt. Durch Anklicken wird der Fehler direkt in der Ausgabedatei angezeigt, muss aber natürlich in der Quelldatei korrigiert werden.

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
```

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.

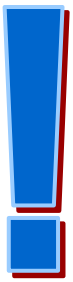
Da bei einem Assembler-Projekt von Studio eine Assembler-Quelldatei `main.asm` angelegt wird, muss man diese im 'Solution Explorer' zunächst mit der rechten Maustaste in `Mein_sAVR_Programm.asm` umbenennen[11] und an derselben Stelle

[10] Label-Prefix _Lxxxx ist voreingestellt.
[11] `main.asm` lässt sich (warum auch immer) nicht von einem anderen Program (wie s'AVR) gleichzeitig öffnen und beschreiben.

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 `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.

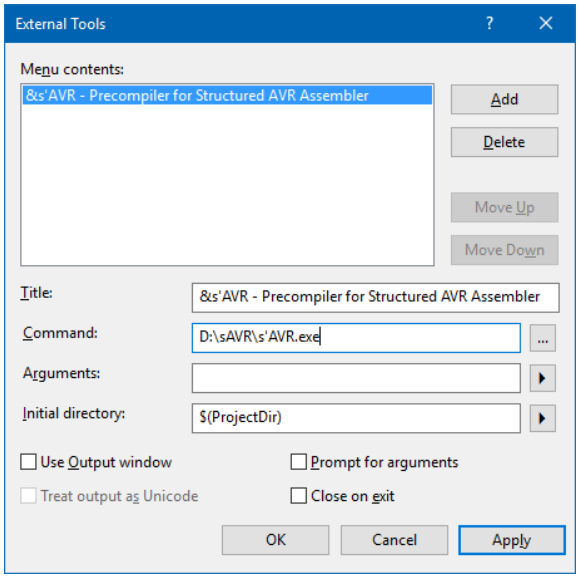
Nun lassen sich beide Quelldateien (also das mit **s'AVR**-Syntax geschriebene `Mein_sAVR_Programm.s` und das von **s'AVR** compilierte "flache" Assembler-Quellprogramm `Mein_sAVR_Programm.asm`) mit dem Studio-Editor nebeneinander bearbeiten - sinnvollerweise aber nur die `Mein_sAVR_Programm.s`-Datei!



Die `*.s`-Datei muss man vor dem Aufruf von **s'AVR** unbedingt abspeichern[13], 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 dient normalerweise nur zum Assemblieren (sie wird nach jedem **s'AVR**-Aufruf automatisch aktualisiert[14]) oder zum Überprüfen des von **s'AVR** erzeugten "flachen" Assembler-Codes und ggf. irgendwelcher generierter 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[15] keine Angabe bei 'Arguments', mehr siehe unter 'Command-Line-Aufruf per Atmel® Studio'):



[12] Bei Dateierweiterungen abweichend von *.asm und *.s entfällt die Unterstützung durch Visual Assist.
[13] Bei neueren Versionen von Atmel Studio wird im geöffneten Fenster ein '*' am Ende des Dateinamens angezeigt, solange Änderungen noch nicht abgespeichert wurden.
[14] Unter Tools/Options/Environment/Documents aktivieren: *'Detect when file is changed ...'* und *'Reload modified files ...'*.
[15] Falls auch noch 'Arguments' an s'AVR übergeben werden, kann man ganz ohne das s'AVR-Programmfenster arbeiten bzw. es wird erst gar nicht geöffnet (nur bei falsch übergebenen 'Arguments').

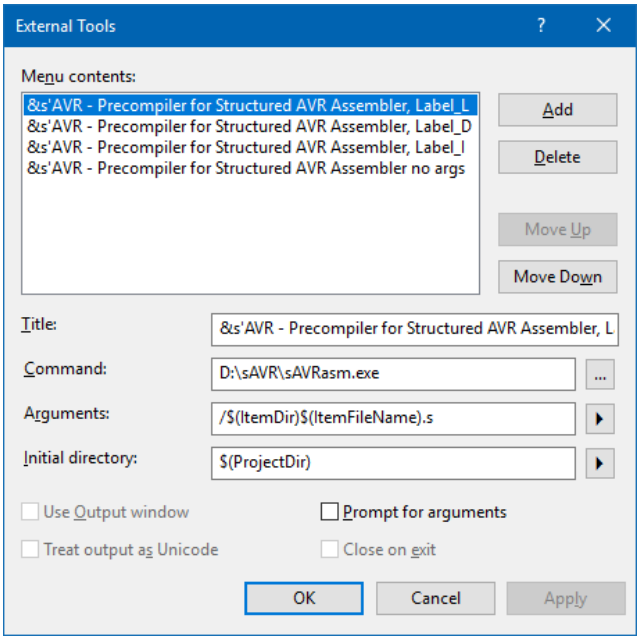
Nach Apply/OK kann man mit dem Studio-Editor nun sowohl den **s'AVR**- als auch den Assembler-Quellcode bearbeiten/anschauen und **s'AVR** unter 'Tools' direkt aufrufen[16].

Nach dem (blitzschnellen!) Compilieren steht der erzeugte Assembler-Quellcode normalerweise[17] als automatisch aktualisiertes `Mein_sAVR_Programm.asm` in Atmel® Studio zur Verfügung und kann nun von Studio aus sofort mittels 'Build' oder 'Rebuild' assembliert und (sofern fehlerfrei) per 'Device Programming' und Programmier-Tool 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'. Man benötigt das `*.asm`-Fenster nur zum Fehlersuchen oder Simulieren/Debuggen.

Noch etwas eleganter

Nach etwas Routine im Umgang mit **s'AVR** wird man unter 'External Tools' auch Argumente angeben, eventuell sogar mehrere für verschiedene Label-Prefixes[18], so dass man das **s'AVR**-Fenster gar nicht mehr benötigt (dann allerdings auch kein Hilfe-Menü mehr hat) und ein paar extra Mausklicks entfallen können:



Unterschiedliche Label-Prefixes werden benötigt, wenn das AVR-Programm aus mehreren einzelnen **s'AVR**-Quellprogrammen besteht, die getrennt compiliert und dann per .INCLUDE durch den AVR-Assembler verknüft werden.

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 ist ein unbehindertes Arbeiten mit **s'AVR** möglich.

[16] Bei einem Update von Atmel Studio 7.0 ist dieser Eintrag verschwunden, so dass er wieder neu angelegt werden musste.
[17] Sofern die `*.asm`-Datei im *'Solution Explorer'* mittels *'Set As EntryFile'* markiert wurde.
[18] Label-Prefixes _Axxxx bis _Zxxxx sind neu ab s'AVR 2.1.

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

IF	*Bedingung* **[THEN]** *BefehlsFolge*	; erste Abfrage
ELSEIF	*Bedingung* **[THEN]** *BefehlsFolge*	; weitere Abfrage(n), mehrfach optional
ELSE *BefehlsFolge*		; letzte Verzweigung, optional
ENDI		; Ende der IF-Struktur
LOOP		; Endlosschleife
BefehlsFolge		; Verlassen der LOOP-Schleife nur mittels ; EXIT, EXITIF, Assembler-Sprung- und Branch- ; Befehlen, RET, Interrupt oder AVR-Reset
ENDL		; Ende der LOOP-Schleife
WHILE	*Bedingung* *BefehlsFolge*	; Abfrage zu Beginn der Schleife
ENDW		; Ende der WHILE-Schleife
REPEAT *BefehlsFolge*		; Beginn der REPEAT-Schleife
UNTIL	*Bedingung*	; Abfrage am Ende der Schleife
FOR	*RegisterZuweisung* *BefehlsFolge*	; FOR-Schleife mit Register-Initialisierung
ENDF		; Dekrementieren des Schleifenzählers und ; 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/Port-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***-Anweisung 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 die älteren 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.
- ***s'AVR***-Anweisungen können mit Groß- oder Kleinbuchstaben (auch gemischt) geschrieben werden.
- Ab ***s'AVR*** 2.0 stehen bei bestimmten ***s'AVR***-Anweisungen Sprungweitenerweiterungen .m und .s zur Verfügung.

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**.

REPEAT wird für die abschließende Schleifenabfrage mit UNTIL beendet.

Alle ***s'AVR***-Strukturen können per EXIT und EXITIF um genau eine Strukturebene verlassen werden.

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 und in `Courier-Schrift` 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 zuverlässigen 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 immer ein **#** vorangestellt (auch wenn es symbolische Konstante sind).

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).

Sprungweite per Anweisungsergänzungen .m und .s

Urspünglich hatte ***s'AVR*** (nämlich bei allen Versionen 1.x) grundsätzlich "schmerzfreien" AVR-Code unter Verwendung von überwiegend BRcc/RJMP-Befehlen generiert, der zwar nicht immer maximal codeeffizient, aber im Normalfall wegen ±2k Worten Sprungweite innerhalb von ***s'AVR***-Strukturen ohne Eingriff wegen zu kleiner Sprungweite möglich war - im Gegensatz zu AVR-Code mit überwiegend BRxx (also ohne Kombination mit RJMP), der dann (wegen BRcc) nur Sprungweiten von +64/-63 Worten erlaubt hätte.

Ab ***s'AVR*** 2.0 kann das ***s'AVR***-Quellprogramm wahlweise mit der bisherigen "schmerzfreien" oder der neuen "effizienten" Option übersetzt werden. Damit es aber speziell bei der "effizienten" Option keine Probleme mit zu kleiner Sprungweite gibt, können bei dieser neuerdings die ***s'AVR***-Anweisungen mit einer Sprungweitenerweiterung .m (für "medium") ergänzt werden, womit dann bei der betroffenen Struktur BRcc mit RJMP kombiniert wird, um trotz sonst "effizientem" Code einen Sprungweitebereich von ±2k Worten zu erzwingen und somit eine Fehlermeldung des Assemblers zu vermeiden.

Umgekehrt erlaubt die Erweiterung .s (für "small") bei der "schmerzfreien" Übersetzungsoption (nämlich bei Quellprogrammen, die ursprünglich für ***s'AVR*** 1.x geschrieben wurden) das Erzwingen von kurzen Sprungweiten per überwiegend BRcc- und Skip-Befehlen.

Bei den folgenden ***s'AVR***-Anweisungen sind die Sprungweitenerweiterungen .m und .s zulässig:

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

Bei diesen Anweisungen ist eine Sprungeweite .m zwar erlaubt (mit einem "Hint" in der Ausgabedatei), aber überflüssig[19]:

ELSE, EXIT, ENDW und **ENDL.**

Und bei diesen Anweisungen ist keine Sprungweitenerweiterung zulässig, da ggf. an anderer Stelle der Struktur über die Sprungweite entschieden wird:

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

Einfach und deshalb nicht ganz konsistent

Eigentlich gehört die Sprungweitenerweiterung nicht bei den Anweisungen IF, ELSEIF, EXITIF und WHILE platziert, denn dort würde man normalerweise[20] Größenerweiterungen (z.B. .w für 16-bit-Wortbreite) und dafür die Sprungweitenerweiterung stattdessen[21] bei THEN, DO und TO erwarten.

[19] Da sowieso AVR-Code für eine mittlere Sprungweite per BRcc/RJMP oder auch nur per RJMP erzeugt wird.
[20] Wie bei anderen strukturierten Assemblern üblich.
[21] THEN ist bei s'AVR optional (ohne weitere Wirkung), DO ist nicht zulässig.

Bedingt durch den AVR-Befehlssatz (im Unterschied zum Befehlssatz von 68000 und anderen µP/µC) ist "effizienter" AVR-Code sehr unterschiedlich im Vergleich zu "schmerzfreiem" AVR-Code. Dummerweise weiß man das bei einem einfachen 1-Pass-Compiler (wie **s'AVR**) aber erst, wenn der abzufragende Ausdruck der betroffenen Anweisungen im Quellprogramm nicht mehr zum Analysieren greifbar ist. Deshalb wurde bei **s'AVR** die Sprungweitenerweiterung der Einfachheit halber an den angegebenen praktischen Stellen definiert statt an den logisch richtigen.

Sprungbefehle

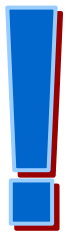
Leider bietet AVR (im Vergleich zu anderen Mikrocontrollern) nur wenige Skip-Befehle, die sich direkt auf das Status-Register beziehen. Dadurch wird ein universelles Verfahren für den Compiler einiges aufwendiger, da statt Skip-Befehlen unnötigerweise an vielen Stellen Branch-Befehle zum Überspringen 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 = alle Bits des AVR-Statusregisters[22]

```
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
NT      ; Transfer-Bit nicht gesetzt

I       ; Interrupt aktiviert
NI      ; Interrupt deaktiviert
```

[22] T / NT und I / NI werden ab s'AVR 1.1 unterstützt.



Wichtige Anmerkungen 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.

Statt NZ, NC etc. darf gleichwertig auch NOT Z, NOT C bzw. !Z, !C verwendet werden.

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

Soll – warum auch immer – in mehr als nur eine Strukturebene höher oder tiefer gesprungen werden, so kann ganz einfach ein unbedingter Assembler-Sprung (RJMP bzw. JMP [nicht bei älteren ATtiny]) zum Einsatz kommen, was natürlich unstrukturiertes Programmieren bedeutet, jedoch in seltenen Fällen übersichtlicher sein kann. Alternativ käme dann ggf. auch ein EXITIF-TO in Frage, s.u.

Man sollte es aber tunlichst unterlassen, mit dieser unstrukturierten Methode Assembler-Unterprogramme[23] zu verlassen, denn der AVR-Stack (Stapelspeicher) könnte dabei 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.

Als Sonderfall von Registerbits können die speziellen Statusbits des Statusregisters direkt angegeben werden, also EXITIF Z, EXITIF NZ u.s.w., Details siehe oben unter "Vordefinierte Statusbits".

Da das AVR-Statusregister aber im I/O-Bereich > 31 liegt (siehe auch nächsten Abschnitt), lassen sich dessen Bits leider nicht in der Form[24] `%SREG,Statusbit` verwenden, womit der erzeugte AVR-Code in vielen Fällen einfacher und effizienter wäre.

[23] Natürlich dürfen vollständige s'AVR-Strukturen auch beliebig innerhalb von Unterprogrammen verwendet werden.
[24] Auch wenn es die s'AVR-Syntax zulassen würde.

Natürlich ist es erlaubt - und manchmal auch zweckmäßig - in mehreren verschachtelten Strukturen ein und dieselbe EXITIF-Bedingung mehrfach hintereinander zu verwenden, um diese Strukturen auch ohne einen unstrukturierten Sprung (`RJMP`, `JMP` oder EXITIF-TO) über mehrere Ebenen hinweg sauber strukturiert zu verlassen, siehe Auszug aus einem *S'AVR*-Programm, das verschiedene Tasten abfrägt:

```
                ...
                EXITIF BUTTONS, S8
                EXITIF BUTTONS, S7
                EXITIF BUTTONS, S3
                ...
              ENDL
            ENDI
            EXITIF BUTTONS, S8
          ENDL
          EXITIF BUTTONS, S8
          ...
        ENDI
        ...
```

Aber schließlich kann man auch einen bedingten Sprung aus *S'AVR*-Strukturen heraus mittels EXITIF-TO durchführen, wofür aber eine Zieladresse nötig ist, die dem strukturierten Programmieren widerspricht.

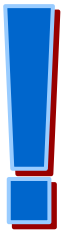
Im obigen Beispiel könnte die unstrukturierte Methode mittels EXITIF-TO also wie folgt aussehen:

```
                ...
                EXITIF BUTTONS, S8 TO exit_s8
                EXITIF BUTTONS, S7
                EXITIF BUTTONS, S3
                ...
              ENDL
            ENDI
          ENDL
          ...
        ENDI
exit_s8:
        ...
```

Dies kann in manchen anderen Situationen sehr vorteilhaft sein, wie wir noch sehen werden.

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 beides Symbole sein, BitStelle auch Zahlenwerte 0-7 (ausschließlich dezimal).
%Port,BitStelle	***%Port*** und ***BitStelle*** dürfen beides Symbole sein, BitStelle auch Zahlenwerte 0-7 (ausschließlich dezimal).

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.

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 (s.o.), falls man nicht die vordefinierten Statusbits verwenden möchte!

Achtung: Nach dem %-Zeichen und vor dem Komma dürfen keine Leerzeichen sein (ab ***s'AVR*** 2.23 sind nach dem Komma Leerzeichen zulässig), um eine eindeutige Zuordnung der einzelnen Symbole zu erhalten!
Ansonsten lässt ***s'AVR*** genügend Spielraum bezüglich Leerzeichen und TABs.

Beispiele für Statusabfragen:

```
EXITIF Reg,3      ; aktuelle s'AVR-Struktur verlassen, falls Register Reg Bit 3 gesetzt ist
```

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, strobe ; Exit falls Port D Bit 'strobe' nicht gesetzt ist

EXITIF Z               ; EXIT falls Z-Bit gesetzt ist
EXITIF NC              ; EXIT falls C-Bit nicht gesetzt ist
EXITIF !C              ; gleichbedeutend mit EXITIF NC
EXITIF NOT C           ; gleichbedeutend mit EXITIF NC
```

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 (`RJMP` bzw. `JMP`) 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.

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

Vergleiche

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

Syntax (Beispiel):

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

Vergleiche können auch Teil der IF-, ELSEIF-, WHILE- und UNTIL-Anweisungen sein.

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 irgendein 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 direkt vorangestellten **#**-Zeichen dargestellt wird. Dann werden zum Vergleichen jedoch nur die AVR-Register 16-31 unterstützt (eine AVR-Eigenschaft). Gegebenenfalls müssen stattdessen zwei AVR-Register miteinander verglichen werden.

Achtung bei Vergleichen:

- Fehler bezüglich unzulässiger AVR-Register 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 Fehlerfälle 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[25].

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 jeweiligen 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**.

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

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

Die **EXIT**- und **EXITIF**-Anweisungen können 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 irgendeine per Label definierte 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 irgendeine 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 das Hauptprogramm - 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 Loop-Label und JMP-Befehl (falls vom verwendeten AVR-µC unterstützt). Das sollte aber eher selten vorkommen, da aufgerufene Unterprogramme immer außerhalb einer solchen Haupt-LOOP-Schleife liegen und deshalb die Sprungweite nicht nachteilig beeinflussen.

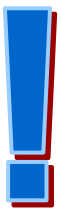
Syntax:

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

Beispiele ausschließlich mit strukturierten LOOP-Schleifen:

```
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[26], 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
```

[26] Obwohl die Syntax dieses sehr kompakten s'AVR-Programmes korrekt wäre.

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[27] 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 irgendwelche **s'AVR**-Anweisungen in derselben Zeile folgen, da sonst diese Anweisungen nicht erkannt werden und **s'AVR** schließlich nicht ausgeglichene Strukturen feststellen würde.

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[28] 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 den AVR-Registern 0 bis 15):

[27] Im Unterschied zu einer vereinfachten Sprungtabelle mit aufeinanderfolgenden Ganzzahlen.
[28] Wert Null, jedoch ausschließlich in der Form #0, #$0, #$00, #0x0, #0x00, #0b0, #0b00, #0b000 und #0b0000.

```
        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-Optionen "no pain" und "keep S'AVR statements in the output file" verwendet: Nach dem ersten ";" (Kommentarzeichen) wird zunächst die aktuelle Strukturebene "01" aufgeführt, gefolgt von "//" (etwas Markantem) und der ursprünglichen S'AVR-Anweisung einschließlich eventuellem ursprünglichem Kommentar der S'AVR-Quellzeile.

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

Einwand?

Beim näheren Betrachten des erzeugten "schmerzfreien" ("no pain") 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 (natürlich automatisch 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 bei der "schmerzfreien" Übersetzungsoption grundsätzlich Branch-Befehle und (sofern möglich) 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).

Somit wird die reichlich geringe Sprungweite der AVR-Branch-Befehle im erzeugten "schmerzfreien" 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, eben "no pain".

Derselbe Sachverhalt gilt auch bei anderen S'AVR-Strukturen.

Voilà - immer effizient ab s'AVR Version 2!

Auch wenn der Gewinn an Programmspeicher und Ausführungsgeschwindigkeit ggf. nicht so extrem ist, hat der Anwender ab s'AVR 2.0 die Wahl zwischen "schmerzfreiem" und "effizientem" Code, entweder pauschal für das gesamte s'AVR-Programm oder sogar gezielt pro Struktur, wie hier für das obige Beispiel per Sprungweite .s compiliert:

```
        in      r17,portd       ; kopiere Port D nach Register 17
        ;01// WHILE.s r17 <> #0 ; solange Register 17 <> Null
_L1:
        TST     r17
        BREQ    _L3
        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:
```

Umgekehrt lässt sich für einzelne s'AVR-Strukturen per Sprungweite .m der "schmerzfreie" Code erzwingen, sofern voreingestellt die "effiziente" Option ausgewählt ist.

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 (hier mit Label-Prefix _Mxxxx und Option "schmerzfrei" compiliert):

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

Erzeugter Code:

```
        ;01// REPEAT            ; Schleife ...
_M1:
        rcall get_character     ; ... um Zeichen nach 'char' zu lesen
        ;01// UNTIL char <> #blank ; jedoch alle Leerzeichen überspringen
        CPI     char,blank
        BRNE    _M2
        RJMP    _M1
_M2:
```

Tipp!

Ein interessanter Programmiertipp:

Die REPEAT-UNTIL-Anweisung spart oft einen AVR-Befehl im Vergleich zur WHILE-ENDW-Anweisung, so z.B. bei einer I/O-Abfrageschleife:

```
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 dann beim Tippen des **s'AVR**-Programms 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:
```

Bei diesen beiden Strukturen ist übrigens kein Unterschied im generierten Code zwischen "schmerzfrei" und "effizient". Das ist - je nach Abfrage - aber nicht immer so.

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 ist eines der AVR-Register.

D.h. der Schleifenzähler kann mit dem Inhalt eines anderen Registers ***Register2*** oder einer Konstanten ***#Konstante*** initialisiert werden (ohne Zwischenraum nach '#').

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 ggf. 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[29] 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.

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

Programmier-Tipp:

Um auch die weniger flexiblen Register R0 bis R15 für FOR-Schleifen mit Konstanten zu verwenden, kann man häufig verwendete Konstante zu Programmbeginn in ein beliebiges anderes AVR-Register (R0 bis R31) laden, um diese dann per

FOR RegisterLower := RegisterHigher

zu verwenden.

Weitere 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
```

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 mit dem Wert 0 initialisiert
```

Erzeugter "effizienter" Code für beide Fälle (ohne s'AVR-Anweisungen und Kommentare):

```
        CLR     lp_cnt
_L1:
        BefehlsFolge
        DEC     lp_cnt
        BRNE    _L1
```

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.
Allerdings fehlen dann auch Kommentare, die in den *s'AVR*-Zeilen enthalten sind.

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.

Optionen für Sprungweiten = "Default Structure Segment Ranges"

- *s'AVR* 1.x hat immer "schmerzfreien" ("no pain") Code erzeugt.
- Ab *s'AVR* 2.0 kann wahlweise "effizient" (voreingestellt) oder "schmerzfrei" erzeugt werden (per GUI oder per Kommandozeile, siehe dort). In beiden Fällen kann dann für jede Struktur zusätzlich auch noch kurze oder mittlere Sprungweite (.s oder .m) erzwungen werden (sofern das Programm dies je nach Segmentgröße zulässt).
- Die Sprungweitenerweiterung ist natürlich nicht rückwärtskompatibel zu *s'AVR* 1.x.

Optionen für Label-Prefix

Ab *s'AVR* 2.1 lässt sich vorgeben, welche Labels von *s'AVR* erzeugt werden, nämlich `_Axxxx` bis `_Zxxxx`, was hilfreich ist, wenn verschiedene Programm-Module als separate *s'AVR*-Quellprogramme geschrieben, diese dann einzeln compiliert und per `.INCLUDE` durch den AVR-Assembler zusammen assembliert werden.

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 Ausgabeda-

tei. Der Assembler kann dann die Anweisung wie gewünscht interpretieren. Dies ist ein sehr einfacher aber wirkungsvoller Trick[30].

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
      BefehlsFolge1
?ELSE                          ; das '?' wird von S'AVR automatisch entfernt
      BefehlsFolge2
ENDIF                          ; diese Zeile stört wegen ENDI nicht, dennoch
                               ; dürfte trotzdem ein '?' vor ENDIF stehen
```

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 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 Atmel Studio die Zeilennummern mindestens in der ***S'AVR***-Quelldatei anzeigen (muss ggf. eingestellt werden).

Und ***S'AVR*** ist überdies so entworfen, daß sich eine Fehlererkennung selbst synchronisiert und nicht viele 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.

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

Kommentare

Kommentare mittels **;**-Zeichen werden von *s'AVR* als solche behandelt und wieder ausgegeben, es sei denn, sie stehen in *s'AVR*-Zeilen und diese sollen per *s'AVR*-Option in der Ausgabedatei unterdrückt werden.

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 (aber außerhalb der Zeilen mit /* und */) trotzdem ausgewertet und compiliert werden (ggf. also auch mit Fehler-, Warn- und Tipp-Hinweisen).

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

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

Adressen, Labels

Sobald *s'AVR* in einer Quellprogrammzeile Assemblerbefehle, Kommentare und auch Labels erkennt, wird der Rest dieser Zeile nicht nach *s'AVR*-Anweisungen untersucht.

Mit anderen Worten:

- Vor *s'AVR*-Anweisungen darf kein Label stehen.
- Jeder Assemblerbefehl muss eine eigene Zeile haben!

Mehrere *s'AVR*-Anweisungen in einer Zeile

Andererseits sind mehrere *s'AVR*-Anweisungen direkt hintereinander ohne Assembler-Befehle dazwischen durchaus in einer einzigen Zeile erlaubt, auch wenn dies nicht besonders übersichtlich = unstrukturiert ausschaut.

Manchmal kann es aber sehr kompakt sein und dann auch Programmzeilen einsparen. Bei komplexen Zeilen kann es passieren, dass die *s'AVR*-Zeile mehrfach als Kommentar ausgegeben wird. Der generierte AVR-Code ist normalerweise dennoch korrekt.

Beispiele:

```
LOOP ENDL                          ; Endlosschleife (bis Interrupt oder Reset)

REPEAT UNTIL %Portx,4              ; warten bis Bit 4 von Portx gesetzt ist

WHILE rega <> regb EXITIF T ENDW   ; warten bis beide Register gleich sind
                                   ; oder das Transfer-Bit gesetzt ist
```

Wenn ein AVR-Assemblerbefehl in derselben Zeile vor, zwischen oder nach *s'AVR*-Anweisungen steht, wird entweder eine nicht ausgeglichene Struktur erkannt oder der Assemblerbefehl geht verloren. Fazit: Doch lieber sauber strukturiert programmieren!

Einige grundsätzliche Syntax-Regeln

Wie bereits vermerkt, wurde *s'AVR* entworfen, um zusammen mit AVRASM2 und ggf. mit der 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 entweder voreingestellt `_Lxxxx` oder optional `_Axxxx` bis `_Zxxxx` und dürfen nicht anderweitig verwendet worden sein.

Ein **Komma** zwischen zwei Indentifiern ist zwingend (und zwar ohne Leerzeichen davor), 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, was aber nicht von jedem AVR-Assembler akzeptiert wird). Für Register- und Ports-Bits sind nur Dezimalzahlen 0-7 (oder Symbole) zulässig.

Ergänzend gilt für *s'AVR*:

Da bei ***s'AVR*-LITE** alle Anweisungen 8-Bit-Operationen sind, werden Werte >255 als Fehler[31] 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 (ohne Leerzeichen!).

Konstante (immediate literals) benötigen ein vorangestelltes **#** und haben die Formate `#999, #$ff, #0xff, #0b1111_1111` und `#'x'` (jeweils ohne Leerzeichen).

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 AVRASM2 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 As-

[31] 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.

sembler-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 die älteren ATtinys verwendet werden kann, die kein `JMP` unterstützen. *s'AVR* verwendet keine Macros sondern erzeugt nur native AVR-Assembler-Befehle.

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" 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 sowohl in der Ausgabedatei (*.asm) als auch in einer separaten Fehlerdatei (*.err, ab Version 2.21). Falls keine Fehler erkannt werden, ist die Fehlerdatei auch nicht vorhanden.

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 | {label_a .. label_z} | /listclp]

/filename.s **/filename.savr**	gültiger *s'AVR*-Sourcecode-Dateiname (benötigt Erweiterung ".s" **oder** ".savr")
/nosavr	*s'AVR*-Anweisungen werden in der Ausgabedatei unterdrückt
/keepsavr	*s'AVR*-Anweisungen werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt)
/nowarn	*s'AVR*-Warnhinweise werden in der Ausgabedatei unterdrückt
/keepwarn	*s'AVR*-Warnhinweise werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt)
/nohint	*s'AVR*-Tipps werden in der Ausgabedatei unterdrückt

/keephint s'AVR-Tipps werden in der Ausgabedatei als Kommentar aufgeführt (voreingestellt)

/label_a ... /label_z statt dem voreingestellten Label-Prefix `_Lxxxx` wird `_Axxxx` bis `_Zxxxx` verwendet

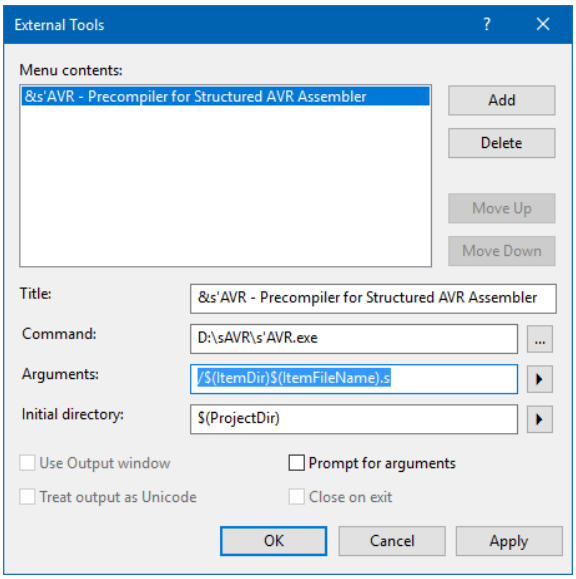
/listclp erzeugt außer der Assembler-Ausgabedatei auch eine Kontrolldatei namens "**sAVR_listclp.txt**", die alle gefundenen Parameter und die zugehörigen Flags auflistet (normalerweise im selben Verzeichnis, von dem aus s'AVR gestartet wurde).

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 per GUI eingreifen könnte (da es nicht mehr erscheint).

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



Die einzelnen Arguments-Parameter erfolgen möglichst alle ohne Leerzeichen dazwischen. Sonst übergibt Atmel® Studio die Parameter einzeln in Hochkommata[32].

Mehrere 'Arguments' werden durch weitere '/$' getrennt (zusätzlich zum '/' der s'AVR-Parameter). 'Prompt for Arguments' bleibt weiterhin deaktiviert[33].

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

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 compiliert 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 - erkennbar, wenn der * am Ende des unter Atmel® Studio angezeigten Dateinamens verschwindet!

Linux

s'AVR lässt sich unter Linux mittels WINE aufrufen und anwenden.

Durch Verwendung von Scripts lassen sich unter Linux auch komplexe Projekte mit wenigen Mausklicks compilieren und assemblieren. Speziell hierfür erzeugt *s'AVR* ab Version 2.21 bei festgestellten Fehlern (und nur dann) eine separate Datei mit der Erweiterung .err, die zeilenweise ausschließlich die Fehlermeldungen auflistet.

Ab Version 2.23 ist *s'AVR* auch in der Lage, Linux-Textdateien ohne den Zwischenschritt `unix2dos`[34] direkt zu lesen und zu compilieren.

Fehlermeldungen

Fehlermeldungen werden von *s'AVR* sonst grundsätzlich in der Ausgabedatei an der vermuteten Fehlerstelle mit Hinweis auf die Zeilennummer des Quellprogramms hinterlegt und beginnen mit `.ERROR`, wodurch AVRASM2 zum Anhalten gezwungen wird und die *s'AVR*-Fehlermeldung unter Atmel Studio anzeigt. Dann kann man bei Bedarf durch Doppelkick auf die Fehlermeldung direkt in die entsprechende Zeile in der Ausgabedatei (nicht in die Quellprogrammdatei) springen. Allerdings müssen Fehler immer im Quellprogramm (`*.s`) und nicht in der Ausgabedatei (`*.asm`) behoben werden!

Falls Fehler entdeckt werden, werden alle Fehlermeldungen in eine separate Fehlerdatei geschrieben (`*.err`, was hilfreich ist, wenn Skripte unter Linux verwendet werden).

Ausblick

Die durch den `RJMP`-Befehl bedingte Einschränkung der LITE-Version wird bei der Vollversion[35] entfallen (erlaubt optional `JMP`-Befehle bzw. große Sprungweite per '`.l`'). Auch wird die Vollversion 16-Bit-Operationen mit einigen *s'AVR*-Anweisungen erlauben.

Ansonsten lassen sich mit *s'AVR* sehr komplexe AVR-Programme sauber strukturiert erstellen und der mit *s'AVR* Version 2 erzeugbare "effiziente" AVR-Code muss einen Vergleich mit einem reinen AVR-Guru-Assembler-Programm absolut nicht scheuen.

Mittels "Label Prefix" (ab *s'AVR* 2.1) lassen sich auch aus mehreren *s'AVR*-Modulen bestehende AVR-Programme erstellen.

[34] Windows/DOS verwendet am Zeilenende einer Textdatei CRLF, Unix/Linux dagegen nur LF.
[35] Die Vollversion ist in Vorbereitung.

Abhängig von den Wünschen der **s'AVR**-Anwender (und meinen freien Resourcen) könnten dennoch weitere **s'AVR**-Funktionen implementiert werden, wie z.B.:

- **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
- Weitere Vorschläge, die **s'AVR** verbessern

Auch wenn **s'AVR** sorgfältig auf richtige Funktion überprüft[36] wurde, lassen sich Programmierfehler[37] aufgrund der Komplexität des Programms nie ausschließen.

Fehlermitteilungen sind per E-Mail immer willkommen (dann möglichst mit einem angehängten Source-Code-Schnipsel als Textdatei, das den Fehler verursacht und die verwendeten **s'AVR**-Optionen oder die zugehörige Ausgabedatei).

Kommentare (Lob&Tadel) 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.)

[36] 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 eingespart - bei deutlich weniger Risiko von Programmierfehlern etc.
Mit den **s'AVR**-Statements sind es dagegen 1.284 Zeilen in der Ausgabedatei.

Ab s'AVR 2.1 (Mitte 2017) wurden erfolgreich komplexere Fremdprogramme getestet (auch unter Linux).

Seit August 2017 wurden an s'AVR keine Änderungen durchgeführt (Stand Dezember 2018).
Damit kompilierte teilweise sehr umfangreiche eigene s'AVR-Programme laufen dauerhaft fehlerfrei in echten Anwendungen.

[37] **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 auch unter Windows® 10 läuft.