Der MCS51 Assembler V4

Andreas Roth, Controllertechnik GbR

Handbuch zum MCS51 Assembler V4 – 4. Auflage 2018

Controllertechnik, Waldstraße 19a, D-69488 Birkenau

Tel.: 06201/393185

Fax: 06201/393184

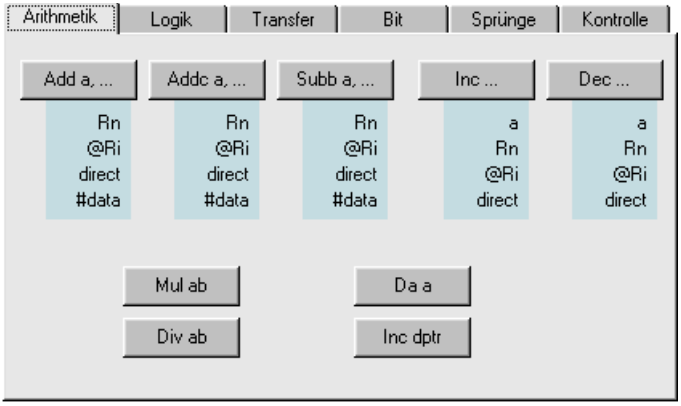
www.Controllertechnik.de

E-Mail: Info@Controllertechnik.de

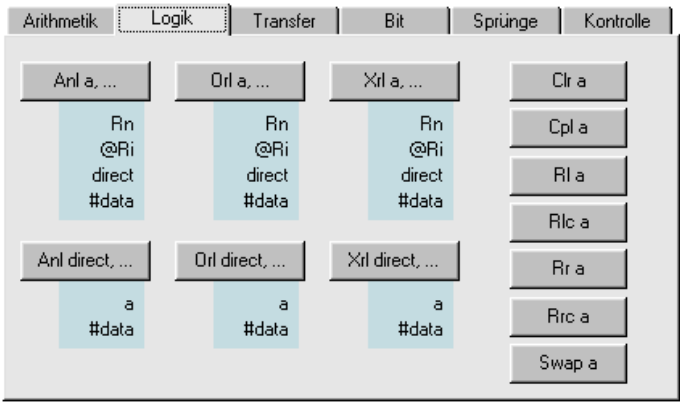
Inhaltsverzeichnis

1. Der Befehlssatz der MCS51 Mikrocontroller

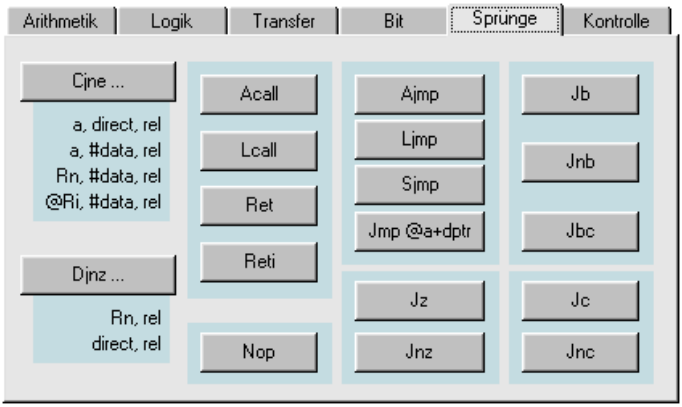
Arithmetische Befehle



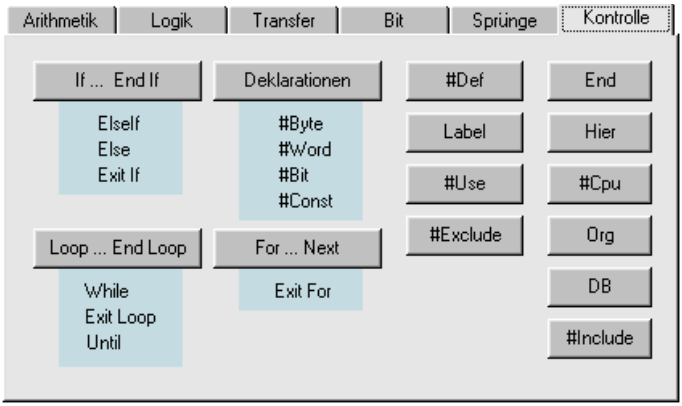
Logische Befehle



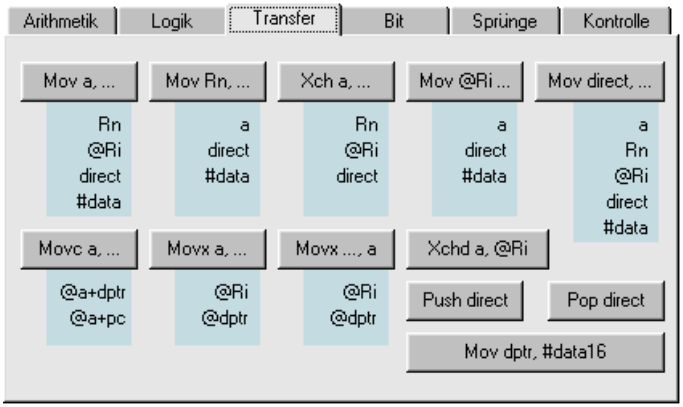
Sprungbefehle



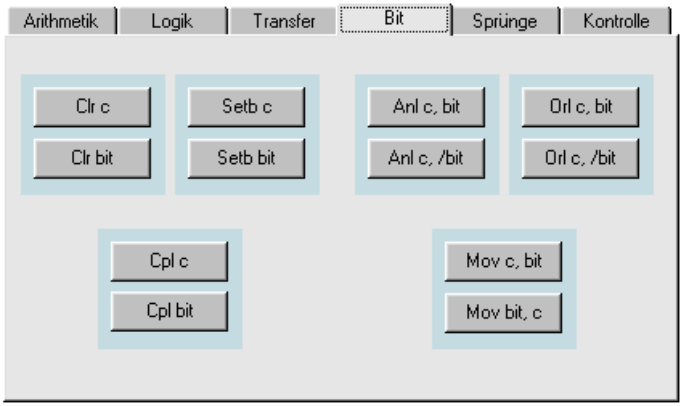
Kontrollstrukturen des Assemblers



Datentransferbefehle



Bitbefehle



Syntax

2. Befehlssyntax

Die Befehle des Assemblers bauen auf der Intelschen Mnemonik auf. Dieser Grundwortschatz wurde um einige Operanden und einige Befehle erweitert, die oberflächlich betrachtet nicht im Befehlssatz zu finden sind und erst beim näheren Hinsehen daraus abgeleitet werden können. Daher sind die Möglichkeiten des vorliegenden Assemblers für Kontrollstrukturen wesentlich umfangreicher.

Format:

Mnemonik *Leerzeichen* Operand1 *Komma* Operand2

z. B. mov P2, #7Fh

Erkennt der Assembler eine gültige Mnemonik, so erwartet er nach einem Leerzeichen oder Tabulator die zugehörigen Operanden. Die Operanden wiederum unterscheidet er am Komma. Den ersten Operanden erwartet der Assembler also zwischen erstem Leerzeichen und erstem Komma; den zweiten Operanden erwartet er zwischen dem ersten und zweiten Komma und den dritten nach dem zweiten Komma. Somit kann ein Operand nahezu alle Zeichen, insbesondere auch Leerzeichen aufweisen. Da jeder Operand von Tabulatoren und Leerzeichen befreit wird, ist deren Verwendung in den Operanden belanglos, außerdem spielt die Groß/Kleinschreibung keine Rolle.

Im Assembler finden Sie im Menü *Hilfe* die zur Verfügung stehenden Befehle zusammen mit deren Operanden.

Mehrere Befehle in einer Zeile

Maschinenbefehle sind in der Regel recht kurz und der Quelltext des Programms verschwindet schnell in einer schmalen Spalte nach unten. Häufiges Umblättern im Editorfenster ist die Folge und eine damit

verbundene Unübersichtlichkeit. Der Assembler kann diesen Nachteil verringern, da er erlaubt, mehrere Befehle in einer Zeile aufzuführen. Die einzelnen Befehle werden durch den Doppelpunkt getrennt. So könnte die folgende Zeile in einer Routine zur A/D-Konvertierung stehen.

```
setb TR1:  setb EA:  dptr = #Tabelle:  mov r0, SBUF
```

Eine Befehlszeile darf nie mit einem Doppelpunkt enden, da sie sonst komplett als Label interpretiert wird.

Kommentare beginnen nach dem Semikolon (;) und enden am Ende der Zeile. Sie werden im Assembler farbig abgehoben, die Kommentarfarbe ist wählbar.

Benutzen Sie Kommentare, um kurze Software-Änderungen zu testen, ohne sie löschen zu müssen. Entscheiden Sie sich danach für die bessere Lösung.

Benutzen Sie Kommentare für die Erklärung einzelner Passagen. Im Moment des Schreibens ist Ihnen der Sinn zwar noch geläufig, nach zwei Wochen aber wissen Sie nicht mehr, wozu diese Zeilen dienen. Tun Sie sich also etwas Gutes!

Gleichheitszeichen

Sie können in der Mnemonik den Mov-Befehl durch das Gleichheitszeichen ersetzen, was das Schreiben und Lesen der Software einfacher macht. Das Ersetzen ist für Movx– und Movc-Befehle nicht möglich.

Zuweisung	Mnemonik
a = #3Eh	mov a, #3Eh
dptr = #1234h	mov dptr, #1234h
c = TR0	mov c, TR0
P1 = #AAh	mov P1, #AAh

3. Zahlenarten

3.1. Dezimalzahlen

Der Assembler fasst zunächst eine jede Zahlenangabe als eine Dezimalzahl auf. Dezimalzahlen dürfen nur die Ziffern 0 bis 9 enthalten. Als Vorzeichen können Sie + oder – verwenden. Zur Betonung des dezimalen Charakters können Sie am Ende ein *d* anfügen.

3.2. Hexadezimalzahlen

Hexadezimalzahlen dürfen nur die Ziffern 0 bis 9 und die Buchstaben A bis F enthalten und müssen mit einem *h* enden. Als Vorzeichen können Sie + oder – verwenden. Folgende Zahlen fasst der Assembler als Hexzahlen auf:

12h = 18d
E3h = 227d
130Ah = 4874d

3.3. Binärzahlen

Binärzahlen dürfen nur die Ziffern 0 und 1 (und strukturierende Leerzeichen) enthalten und müssen mit einem *b* enden. Als Vorzeichen können Sie + oder – verwenden. Es müssen nicht alle acht bzw. sechzehn Stellen vorhanden sein, d.h. Sie können auf führende Nullen verzichten.

Beispiele

10 1101b = 45d
10001011B = 139d

3.4. Text (Zeichenketten)

Zeichenketten können alle ANSI-Zeichen mit Ausnahme des Hochkommas (‘) , des Kommas (,) und des Doppelpunktes (:) enthalten. Eine Zeichenkette darf als Bestandteil eines Befehls maximal ein Zeichen umfassen, als Bestandteil einer Tabelle beliebig viele. Es wird zwischen Groß- und Kleinschreibung unterschieden. Zeichenketten müssen zwischen zwei Hochkommas (‘) stehen, z. B. ‘aBc‘.

Beispiele

'A' = 41h

'3' = 33h

@r0 = #'m'

3.5. Label

Label geben die Adressen der durch Label markierten Speicherstellen oder Interrupts zurück.

Beispiele

mov dptr, #*Labelname*

Die Adresse von *Labelname* wird in den Datenpointer geschrieben.

mov r0, #Low(*Labelname*)

Das Lowbyte der Adresse von Labelname wird in Register 0 geschrieben.

3.6. Bytevariablen

Bytevariablen geben die Adressen oder Inhalte der zuvor implizit oder explizit definierten Speicherstellen zurück.

Beispiele

mov r0, #*Bytevariable*

Die Adresse von *Bytevariable* wird in das Register 0 geschrieben.

mov r0, *Bytevariable*

Der Inhalt der Adresse von *Bytevariable* im internen RAM wird in das Register 0 geschrieben.

3.7. Wordvariablen

Wordvariablen geben die Adressen oder Inhalte der zuvor implizit oder explizit definierten Speicherstellen zurück.

Beispiele

mov dptr, #*Wordvariable*

Die Adresse von *Wordvariable* wird in den Datenpointer geschrieben.

mov r0, *Wordvariable*

mov r1, *Wordvariable* + 1

Der Inhalt der Adresse von *Wordvariable* wird in das Register 0 und der Inhalt von *Wordvariable + 1* in das Register 1 geschrieben.

3.8. Konstanten

Konstanten sind Platzhalter für zuvor definierte Zahlen und dienen der besseren Lesbarkeit und Pflege eines Programms.

Syntax: #CONST *const1 = Wert 1, const 2 = Wert 2* etc.

Beispiele

```
#Const abc = 36h
mov r0, #abc
```

Die Zahl 36h wird in das Register 0 geschrieben.

```
mov r0, abc
```

Der Inhalt der Adresse 36h wird in das Register 0 geschrieben.

3.9. Spezialfunktionsregister SFR

Die Namen der Spezialfunktionsregister sind in der CPU-Datei festgelegt und können dort geändert werden. Sie können nur durch eine Direktadressierung gelesen oder beschrieben werden.

Beispiele

```
mov r0, #P2
```

Die physikalische Adresse von Port 2 = A0h wird in das Register 0 geschrieben.

```
mov r0, P2
```

Der Inhalt von Port 2 wird in das Register 0 geschrieben.

4. Zahlenbereich

Der Zahlenbereich für ein Byte reicht von 0 bis 255, in Hex 00h bis FFh. Für Konstanten ist er von 0 bis 65535, in Hex 0000h bis FFFFh. Wenn die Zuweisung für ein Byte größer als 255 (FFh) ist, wird eine Warnung ausgegeben. Das muss nicht immer auf eine Fehleingabe hinweisen. Intern handhabt der Assembler alle Zahlen als Zwei-Byte-Werte. Zu dieser Warnung kommt es beispielsweise, wenn Sie eine negative Zahl verwenden. Im Befehl mov a, #–13 wandelt der Assembler beispielsweise

die negative Zahl in eine signierte Hexzahl um. –13 hat die Hexdarstellung FFF3h. In diesem Fall verwendet der Assembler nur das Low-Byte F3h. Die –13 wird also zu F3h.

Ausnahme ist der Datenpointer. Ihm kann man eine größere Zahl zuweisen, z.B. mov dptr, #1234h.

Dennoch können Sie Teile von Zahlen größer als 255 einem Byte zuweisen. Mit *#Low(Zahl)* bzw. *#High(Zahl)* wird das Low- bzw. High-Byte der genannten Zahl oder Berechnung verwendet. Am häufigsten wird das bei der Verwendung von Label geschehen.

Beispiel

```
mov P2, #High(Label)
mov r0, #Low(Label)
For r7 = #10
    movx @r0, a
    inc r0
Next
```

5. Berechnungen

Der Assembler kennt die vier Grundrechenarten

Plus	**+**
Minus	**–**
Mal	*
Geteilt	/

Diese Grundrechenarten finden bei Konstanten und Adressen Verwendung. Das macht beim Beschreiben eines Display Sinn, um die auszugebenden Zeichen leichter platzieren zu können. Der Zeilenanfang habe z. B. die Adresse 80h im IRAM, der Messwert soll an der fünften Stelle angezeigt werden, dann lauten die Befehle:

```
r0 = #80h + 5: a = Messwert: acall Messwertanzeige
```

Interessanter wird das bei Variablen, wenn z. B. die halbe Mehrwertsteuer

mit MwSt / 2 errechnet werden soll und MwSt sich im Laufe der Jahre ändert.

Ein anderer Verwendungszweck liegt bei Tabellendaten. Wenn das Label *Tabellenanfang* die Startadresse der Tabelle im IRAM angibt, dann schreibt der Befehl

```
r0 = #Tabellenanfang + 2: a = @r0
```

das dritte Byte der Tabelle in den Akku, und

```
r7 = #Tabellenende – Tabellenanfang
```

schreibt die Länge der Tabelle in das Register 7.

Achten Sie darauf, dass im Falle von Konstanten nur einmal das Doppelkreuz # zu Beginn steht.

Sie können in einer Zeile beliebig viele Berechnungen vornehmen lassen. Dabei führt der Assembler allerdings **ohne** Beachtung der Rechenhierarchie die Berechnungen von links nach rechts durch. Klammern werden nicht unterstützt.

```
mov r1, #317 – 14 * 11 – 2
```

Schreibt in das Register 1 die Zahl 161.

6. Variablen

Variablen müssen vor ihrer Verwendung angemeldet werden. Der Assembler unterscheidet zwischen Bit-, Byte- und Word-Variablen sowie Konstanten. Die Namen von Byte- und Word-Variablen und Konstantennamen dürfen untereinander nicht identisch sein und mit Labelnamen nicht übereinstimmen.

6.1. Bitvariablen

Die Syntax für Bitvariablen ist

```
#Bit Varname [= Bitadresse1][, Varname2 [= Bitadresse2], ...]
```

Varname ist der von Ihnen gewählte Name der Bitvariablen. *Varname* kann alle Zeichen, auch Leerzeichen, mit Ausnahme des Kommas und des

Doppelpunktes enthalten. Groß- und Kleinschreibung ist belanglos. Bitadresse ist üblicherweise die Hexzahl des Bitoffset gefolgt von der Bitnummer, z.B. 24h.4 oder P3.2 oder acc.0.

Ist die Bitadresse für den Varnamen nicht angegeben, vergibt der Assembler selbstständig eine freie Bitadresse absteigend von 7Fh = 2Fh.7. Zuvor prüft der Assembler, ob der Name nicht mit einem Bitnamen aus dem SFR-Bereich oder einem bereits definierten Namen identisch ist, und gibt gegebenenfalls eine Warnung aus.

Als **Bitadresse** können Sie

1. eine Zahl von 0 bis 255 dezimal, hexadezimal oder binär oder
2. ein bitadressierbares Byte im Bereich von 20h bis 2Fh und ab Adresse 80h im SFR-Bereich, wobei die Adresse im SFR-Bereich mit 0h oder 8h enden muss, gefolgt von einem Punkt und der Bitnummer von 0 bis 7, z.B. 25h.3 oder A0h.7, oder
3. ein SFR-Name oder ein Byte-Variablenname aus einem unter Punkt 2 genannten Adressbereich gefolgt von einem Punkt und der Bitnummer von 0 bis 7, z.B. 25h.3 oder A0h.7, angeben.

Beispiel

```
#Bit Status = 22h
      setb Status
#Byte Bitregister = 26h      ; 26h liegt im
      clr Bitregister.5      ; bitadressierbaren Bereich.
c = PSW.1
```

Sie können der Bitvariablen auch eine feste Adresse zuordnen. Für *Bitadresse* kann aber auch eine zuvor definierte Bitvariable oder ein Bitname aus dem SFR-Bereich verwendet werden. Damit können SFB-Namen umdefiniert werden. Durch Adresszuweisung kann ein Bit auch mehrere Namen erhalten.

Beispiel

#Bit Negativ = 3Ah, Status, Motorausgang = P1.1

; Das Bit *Negativ* befindet sich an Adresse 3Ah = 27h.2.

; Die Adresse für die Variable *Status* wird automatisch vergeben und wäre hier 7Fh = 2Fh.7.

; Die Variable Motorausgang hat die Adresse von Port 1:
P1.1 = 91h = 90h.1.

Beispiel

Wenn Sie viele Ausgänge mit derselben Funktion, z. B. zwanzig Fenster steuern wollen und die verschiedenen Zustände bitcodiert speichern müssen, kann der Bitbereich des Controllers schnell erschöpft sein. Dann reservieren Sie ein Byte im bitadressierbaren Bereich und nennen das Byte z. B. *Bitadresse* und geben den einzelnen Bits darin den Namen ihrer Verwendung. In einem externen Speicher bewahren Sie eine Kopie dieser Bitadresse auf. Vor der Verwendung der Bits kopieren Sie das Statusbyte des betreffenden Fensters aus dem externen Speicher in die Variable *Bitadresse*, nach der Verwendung schreiben Sie den Inhalt der Bitadresse wieder zurück. Dann verwendet die Software lediglich acht Bit anstelle von 20 x 8 = 160 Bit. Dann können Sie die folgende Technik anwenden:

```
#Byte Bitadresse = 26h      ; liegt im Bitbereich
#Bit Motor auf = Bitadresse.7, Motor zu = Bitadresse.6
#Bit ist Wind = Bitadresse.5, ist Regen = Bitadresse.4
#Bit ist Verzögerung = Bitadresse.3
#Bit ist Phase = Bitadresse.2
#Bit Fenster öffnet = Bitadresse.1
#Bit Fenster schließt = Bitadresse.0
```

In der Software zur Auswertung steht dann zu Beginn die Sequenz:

```
r0 = #Status Fenster 1: movx a, @r0: Bitadresse = a
. . weitere Befehle
```

In der Auswertungsroutine können Sie dann für jedes Fenster dieselben Bit-Namen verwenden, und am Ende schreiben Sie die eventuell geänderten

Bits wieder zurück, d. h. Sie brauchen für alle Fenster nur eine einzige Routine:

r0 = #Status Fenster 1: a = Bitadresse: movx @r0, a

Beachten Sie die Lage der Bits im Controller:

Bitadresse	**Lage**
00h – 7Fh:	internes RAM, Adressen 20h bis 2Fh
80h – FFh:	Spezialfunktionsregister SFR

Zugriff auf Einzelbits im externen RAM gibt es nicht, Einzelbit im externen RAM können Sie laut vorstehender Methode realisieren.

6.2. Bytevariablen

Die Syntax für Bytevariablen ist

#Byte *Varname1 [= Byteadresse1] [, Varname2 [= Byteadresse2], ...]*

Varname ist der von Ihnen gewählte Name der Bytevariablen. *Varname* kann alle Zeichen, auch Leerzeichen, mit Ausnahme des Kommas und des Doppelpunktes enthalten. Groß- und Kleinschreibung ist belanglos. Byteadresse ist üblicherweise eine Hex- oder Dezimalzahl eines Speicherplatzes im internen (00h bis FFh) bzw. externen RAM.

Ist die Byteadresse für den Varnamen nicht angegeben, vergibt der Assembler selbstständig eine freie Byteadresse absteigend von 7Fh bis 00h im internen RAM. Zuvor prüft der Assembler, ob der Name nicht mit einem SFR-Namen oder einem bereits definierten Bytenamen identisch ist, und gibt gegebenenfalls eine Warnung aus.

Sie können der Bytevariablen auch eine feste Adresse zuordnen. Für *Byteadresse* kann aber auch eine zuvor definierte Bytevariable verwendet werden. Durch Adresszuweisung kann eine Adresse auch mehrere Namen erhalten.

Byte- und Wordvariablen teilen sich denselben Adressraum, ihre Adressen überlagern sich aber nicht. Bei der automatischen Adressvergabe wird darauf geachtet, dass eine Bytevariable keine Adresse einer Wordvariablen einnimmt und umgekehrt.

Beispiel

#Byte abc = 3Ah, Zähler, externer Modus = P1

; Der Variablen abc wird die Adresse 3Ah zugewiesen.

; Die Adresse für die Variable *Zähler* wird automatisch vergeben und ist hier 7Fh.

; Die Variable *externer Modus* hat die Adresse von Port 1 = 90h.

6.3. Wordvariablen

Die Syntax für Wordvariablen ist

#Word *Varname1 [= Wordadresse1] [, Varname2 [= Wordadresse2], ...]*

Varname ist der von Ihnen gewählte Name der Wordvariablen. *Varname* kann alle Zeichen, auch Leerzeichen, mit Ausnahme des Kommas und des Doppelpunktes enthalten. Groß- und Kleinschreibung ist belanglos. Wordadresse ist üblicherweise eine Hex- oder Dezimalzahl eines Speicherplatzes im internen (00h bis FFh) bzw. externen RAM (0000h bis FFFFh).

Ist die Wordadresse für den Varnamen nicht angegeben, vergibt der Assembler selbstständig eine freie Wordadresse absteigend von 7Fh bis 00h im internen RAM. Zuvor prüft der Assembler, ob der Name nicht mit einem SFR-Namen oder einem bereits definierten Wordnamen identisch ist, und gibt gegebenenfalls eine Warnung aus.

Sie können der Wordvariablen auch eine feste Adresse zuordnen. Für Wordadresse kann aber auch eine zuvor definierte Wordvariable verwendet werden. Durch Adresszuweisung kann eine Adresse auch mehrere Namen erhalten.

Byte- und Wordvariablen teilen sich denselben Adressraum, ihre Adressen überlagern sich aber nicht. Bei der automatischen Adressvergabe wird darauf geachtet, dass eine Bytevariable keine Adresse einer Wordvariablen einnimmt und umgekehrt.

Eine Wordvariable beansprucht zwei Byte.

Beispiel

#Word Tabelle = 300Ah, bcd, externer Modus = P1

; Die Variable *Tabelle* befindet sich an der Adresse 0300Ah im externen RAM.

; Die Adresse für die Variable *bcd* wird automatisch vergeben und ist hier 007Eh.

; Die Variable *externer Modus* hat die Adresse von Port 1 = 0090h.

6.4. Konstanten

Konstanten werden im Programm wie Zahlen oder Adressen verwendet. Durch ihre Textform sind sie aussagekräftiger als Zahlen. Sie sollten vor dem Beginn ihrer Verwendung deklariert werden. Die Syntax für Konstanten ist

#Const Constname1 = Wert1[, Constname2 = Wert2, [...]]

Constname ist der von Ihnen gewählte Name der Konstanten. *Constname* kann alle Zeichen, auch Leerzeichen, mit Ausnahme des Kommas enthalten. Groß- und Kleinschreibung ist belanglos. *Wert* muss nach dem Gleichheitszeichen angegeben werden und ist üblicherweise eine Hex- oder Dezimalzahl (ohne führendes #). Der Inhalt einer Konstanten kann nicht redefiniert werden.

Konstanten belegen keinen Speicherplatz im Controller. Sie sind nur im Speicher Ihres Computers zur Zeit des Compilierens vorhanden. Die gleiche Wirkung wie die Verwendung der Konstanten hat der #Def-Befehl. Sie können auch mit der nötigen Vorsicht als Bytevariablen verwendet werden.

Beispiel

```
#Const abc = 3Ah, bcd = 20, Tabellenoffset = 512d
add a, #abc          ; zum Akku wird der Wert 3Ah addiert.
#Const Temperaturwahl = 11h, Baudrate = 12h
#Const Zahlungsadresse = 3CBh
```

7. Label

7.1. Definition

Label sind Sprungziele im Programm. Der Assembler erkennt Label daran, dass als letztes Zeichen vor einem eventuellen Semikolon (;) ein Doppelpunkt (:) steht. Alles, was mit einem Doppelpunkt endet, ist ein Label, d.h. eine virtuelle Adresse, deren physikalischer Wert erst beim Assemblieren ermittelt wird. Label können somit Buchstaben oder Zahlen, ganze Sätze oder Sonderzeichen sein. Label dürfen mit einem Befehl nicht in einer Zeile stehen. Folgende Konstellation führt zu einer Fehlermeldung:

```
Label:  mov a, #23
```

Richtig ist hingegen:

```
Label:
    mov a, #23
```

Bei der Benutzung von Label in Sprungbefehlen ist hingegen der Doppelpunkt wegzulassen.

falsch ist

```
jz  Label 1:
```

richtig ist

```
jz  Label 1
```

7.2. Reservierte Label

Einige Bezeichnungen für Label sind reserviert und weisen dem so bezeichneten Label eine feste Adresse zu. Es handelt sich dabei um die **Reset-** und **Interruptadressen**. Diese reservierten Bezeichnungen sind Bestandteil der CPU-Datei und werden beim Beginn des Assemblierens geladen. Eine Liste der reservierten Label und deren Adressen können Sie dem jeweiligen CPU-Treiber - z. B. 89S8253.CPU - entnehmen.

Das Label *Reset:* ist optional und kann weggelassen werden, da der Assembler in diesem Fall automatisch dem ersten Befehl die Adresse 0000 zuweist.

Die Adressen der reservierten Label können nur durch Änderung in der *. CPU Datei geändert werden.

7.3. Interrupt

Die Kennzeichnung von Interrupts erfolgt durch deren Namen. Möchten Sie z. B. eine Interruptroutine für Timer 0 erzeugen, so schreiben Sie in den Quelltext z.B. die Zeilen:

```
Sjmp Initialisierung
Timer 0: ; Timer 0 Interrupt. Erhält die Adresse 0Bh.
    Befehle ...
reti
Initialisierung:
setb ET0 ; Interrupt Timer 0 freigeben
setb EA  ; globale Interruptfreigabe
setb TR0 ; Start Timer 0
end      ; Endlosschleife, wartet nur auf den
         ; Timer 0 Interrupt.
```

Achten Sie darauf, dass die Nennung des Interruptnamens wie eine ORG-Anweisung funktioniert. Der dem Label *Timer 0* folgende Befehl startet nun an der Adresse 000Bh, der Interruptadresse von Timer 0. Es ist damit möglich, bereits assemblierten Code zu überschreiben. Achten Sie daher bei mehreren Interrupts auf deren richtige Reihenfolge.

7.4. Besonderheiten

1. Trifft der Compiler auf ein Label ohne eine Mnemonik davor und ohne Doppelpunkt dahinter, so ergänzt er den Befehl ACALL. Mit der Assembler-Direktive #use lcall können Sie den Assembler veranlassen, statt dem Befehl ACALL den Befehl LCALL zu verwenden. Das ist erforderlich, wenn Ihr Programm den 2K-Block überschreitet, da sonst die Subroutinen nicht mehr korrekt aufgerufen werden können.

2. Das Wort **END** wird mit SJMP *auf der Stelle* übersetzt. Diese Endlosschleife kann nur mit einem Interrupt unterbrochen werden.

3. Sprünge, die auf das Eintreten einer Bedingung warten, können den Label-Namen **HIER** verwenden.

```
jb P3.2, hier        ; wartet, bis der Eingang P3.2 an Masse geht.
```

4. Labeladressen können an Variablen und an den Datenpointer übergeben werden. Da jede Adresse einen 16Bit-Wert darstellt, muss man mitteilen, welcher Teil übergeben werden soll:

```
a = #High(Label)  ; schreibt das High-Byte des Label
                  ; in den Akku.
@r0 = #Low(Label) ; schreibt das Low-Byte des Label an
                  ; die Adresse, die in R0 steht.
```

5. Labelnamen dürfen mit Byte- Word- und Constantennamen nicht identisch sein.

8. END

Verwendung: Abschluss eines Programms

Sie sollten bei der Erstellung der Software vermeiden, dass das Programm nach dem letzten Befehl in den nicht benutzten Speicherbereich weiter läuft, da es an dessen Ende angekommen an der Adresse 0000 weitermacht und die dort stehenden Befehle unbeabsichtigt ausführt. Ferner können Sie nicht sicher sein, dass in dem unbenutzten Speicherbereich nur FFh oder 00h als Befehlscode auftritt. Beenden Sie daher Ihr Programm mit einer Endlosschleife oder der Anweisung *END*.

Der Assembler weist dem Wort ***END*** den Maschinencode 80h FEh zu. Dieser Code bewirkt eine Endlosschleife auf der Stelle. Er ersetzt folgende Anweisung:

```
Ende:
    sjmp Ende
```

Diese Endlosschleife kann nur durch einen Interrupt unterbrochen werden. Daher wird *END* in Interrupt basierenden Anwendungen verwendet.

9. HIER

Funktion: Rückgabe der aktuellen Befehlsadresse

Das Wort ***HIER*** ersetzt die relative Adresse bzw. einen Label bei bedingten Sprüngen. Der Assembler ermittelt die Befehlslänge des relativen Sprungbefehls und ergänzt als relative Adresse den Wert FEh bei 2-Byte-Befehlen oder FDh bei 3-Byte-Befehlen.

HIER gibt ferner die Adresse des aktuellen Befehls zurück, z.B. mov dptr, #hier oder r0 = #Tabellenanfang - hier.

Beispiel

```
jb P1.3, hier
```

Das Programm wartet so lange auf der Stelle, bis der Pin P1.3 an Masse geht. Danach wird mit dem weiteren Programm fortgefahren. Das Wort *HIER* ersetzt also folgende Anweisung:

```
Warte:
    jb P1.3, Warte
```

10. Definitionen

Definitionen oder Makros sind Textteile, die beim Compilieren durch andere Textteile ersetzt werden. Ein Zweck des Vorgangs ist es, wenig aussagekräftige Mnemoniks oder Operandennamen durch aussagekräftigere zu ersetzen. Ein weiterer Zweck liegt darin, kleinere, oft wiederkehrende Befehlsfolgen einzufügen. Für längere Befehlsfolgen ist es sinnvoll, eine Subroutine zu verwenden.

Die **Syntax I** für Makros ohne Parameter ist

#Def *Makroname* {*Definition*}

Die **Syntax II** für Makros mit Parameter ist

#Def *Makroname*(p1 [[, p2], ..]) {*Definition*(p1 [[, p2], ..])}

Makroname ist die Bezeichnung, an der der Compiler das Makro im

Programm wiedererkennt. *Definition* ist der Befehl oder die Befehlsfolge, mit dem der Compiler den Makronamen ersetzt. p1, p2, .. pn sind optionale Parameter, die später im Quelltext mit Operanden ersetzt werden.

Beispiele

```
; Einfache Zahlersetzung
    #def MwSt {19}
    mov r0, #MwSt        ; wird zu mov r0, #19
; Befehlsbenennung
    #def Motor an {setb P1.5}
    Motor an             ; wird zu setb P1.5
    #def Displaybeleuchtung an {setb Backlight}
; Befehlssequenz
    #Bit Taktpin = P2.3
    #def 573 Takt {P0 = a: clr Taktpin: P0 = #FFh}
    573 Takt             ; wird zu P0 = a: clr P2.3: P0 = #FFh
; Makro mit zwei Parametern
    byte var1, var2
    #def vertausche(x, y) {push(x): mov(x, y): pop(y)}
    vertausche (Messwert alt, Messwert neu)
wird zu
    push Messwert alt: Messwert alt = Messwert neu: pop Messwert neu
```

Das letzte Beispiel zeigt die Verwendung von Parametern. Parameter sind immer in runde Klammern zu setzen. Die Parameter des Makronamens müssen mit den Parametern der Definition identisch sein. Bei der Ersetzung später im Programm werden diese Parameter in der richtigen Reihenfolge durch die genannten Operatoren, hier *var1* und *var2* ersetzt.

End

Das Wort *END* ist ein internes Makros und erzeugt die Struktur

```
EndLabel:
    SJMP EndLabel
```

Die Programmausführung verharrt also an dieser Stelle. Eine Unterbrechung

kann nur durch einen Interrupt erfolgen. End kann an beliebiger Stelle des Programms stehen.

Beispiel

Für das nachfolgende Programm sind acht LED (ohne Vorwiderstand) an Port 1 gegen +5V angeschlossen. Das Programm erzeugt ein Lauflicht mit unterschiedlichen Geschwindigkeiten und einer Richtungsumkehr.

```
#cpu = 8031
Byte Tempo
#def erste LED an {clr P1.0}
#def rotate left {a = P1: rl a: P1 = a}
#def rotate right {a = P1: rr a: P1 = a}
; Reset:
erste LED an: Tempo = #50h
Loop
    Loop: Delay: rotate left: dec Tempo: a = #1: Until a = Tempo
    Loop: Delay: rotate right: inc Tempo: a = #50h: Until a = Tempo
End Loop
Delay:
    for r7 = Tempo: djnz r6, hier: next
ret
```

Die Software wird durch die Verwendung von Definitionen übersichtlicher und ist daher leicht zu pflegen.

11. #CPU Anweisung

Die erste Anweisung in Ihrem Quelltext muss die CPU-Anweisung sein. Sie gehorcht folgender Syntax:

CPU = ... oder **CPU ...**
oder
#CPU = ... oder **#CPU ...**

wobei für die Punkte der Namen des verwendeten Controllers einzutragen ist.

Beispiel

```
#CPU = 89S8253
```

Haben Sie die CPU-Treiber in einem anderen Unterverzeichnis oder auf einem USB-Stick, geben Sie zusätzlich den Pfad mit an, damit der Assembler den Treiber finden kann. Weil Laufwerksangaben in der #**CPU-Anweisung** mit einem Doppelpunkt durch den Assembler in zwei Befehle zerlegt würden, müssen Sie in diesem Fall den Pfad mit CPU-Treiber zwischen Hochkommas stellen. Bei fehlender Laufwerksangabe können die Hochkamms entfallen.

Beispiel

```
#CPU = 'c:\support\80535'
```

Der Assembler holt sich den Treiber aus dem Unterverzeichnis SUPPORT. Das Suffix '.CPU' lassen Sie bitte weg.

Die CPU-Anweisung ist unabdingbar, denn ohne sie kann der Assembler nicht arbeiten. Nach dem Laden des Treibers hat der Assembler alle Informationen über den verwendeten Controller. Dazu gehören:

1. Informationen über die internen RAM/ROM-Größe
2. Die Namen und Adressen der reservierten Label
3. Die Namen und Adressen der Spezialfunktionsregister
4. Die Namen und Adressen der Bits.

Zur Erstellung eigener Treiber wandeln Sie einen bestehenden Treiber sinngemäß um.

12. #USE Anweisung

Die Verwendung von **#use acall** und **#use lcall** ist optional. Die beiden Direktiven steuern die Verwendung der CALL-Befehle für Subroutinenaufrufe im Quelltext.

Wenn keine dieser Anweisungen im Quelltext auftritt, verwendet der Assembler den ACALL-Befehl zum Subroutinenaufruf. Mit #use lcall

verwendet er nachfolgend den LCALL-Befehl. Dieses Verhalten wird mit #use acall wieder rückgängig gemacht.

#use acall und #use lcall können beliebig oft im Quelltext auftreten.

13. #Exclude-Anweisung

Syntax

#Exclude Byte *untere Grenze – obere Grenze*

#Exclude Bit *untere Grenze – obere Grenze*

Die #Exclude-Anweisung schließt einen Speicherbereich im Byte- oder Bit-Bereich für die automatische Adressvergabe von Variablen aus. Meist schützt man damit einen Bereich, der zur Laufzeit als zusammenhängender Speicherblock verwendet wird.

Der Wert von *untere Grenze* muss kleiner als der Wert von *obere Grenze* sein. Beide Zahlen müssen mit einem Minus getrennt sein. Der Wert von *obere Grenze* kann in beiden Fällen maximal 7Fh sein.

Beispiel

```
#Exclude Byte 70h – 7Fh
```

Die internen RAM-Adressen von 70h bis 7Fh stehen für die automatische Adressvergabe für Byte- und Word-Variablen nicht mehr zur Verfügung.

Beispiel

```
#Exclude Bit 28h.0 – 29h.7
#Byte Kontrolllampen 1 = 28h, Kontrolllampen 2 = 29h
```

Die 16 Bit an den Adressen von 28h.0 bis 29h.7 sind vor der automatischen Adressvergabe für Bit geschützt, weil man sie, z. B. sowohl für die Bit- als auch für die Byte-Adressierung verwenden möchte.

14. ORG Anweisung

Mit der ORG-Anweisung (Origin) können Sie dem Assembler eine absolute Adresse mitteilen.

Syntax

ORG *Adresse* (*Adresse* = Zahl zwischen 0 und FFFFh)

z.B. Org 1FAh

Trifft der Assembler auf eine ORG-Anweisung, übernimmt er den Wert und ordnet dem folgenden Befehl die genannte Adresse zu.

Achtung!

Sie können damit bereits assemblierten Code überschreiben, wenn die ORG-Anweisung in bereits assemblierten Bereich zeigt.

Gedacht ist die ORG-Anweisung für das Auslesen von Tabellen etc. Soll an die Stelle, die der ORG-Anweisung folgt, gesprungen werden, muss der Anweisung ein Label folgen. ORG-Anweisungen dürfen nicht mit Label oder Befehlen in einer Zeile stehen. Bitte verwenden Sie für Interrupt-vektoren keine ORG-Anweisung, sondern deren Namen.

Beispiel

```
Org 3000h
Copyright:
DB ‘Copyright 2019‘, 0
```

15. Daten und Tabellen

Dateneingaben müssen vorgenommen werden, wenn mit den Befehlen MOVC A, @A+DPTR oder MOVC A, @A+PC Daten aus Tabellen gelesen werden sollen, z. B. die Anzeigen-Codes für eine 7-Segment-Anzeige, oder Datum und Copyright-Vermerke Aufnahme finden sollen.

Definition der Daten

Eine Dateneingabe beginnt mit den Buchstabe DB für Define Byte.

Syntax

DB Wert1, Wert2, . . , Wert-n

Die Daten stehen in einer Zeile. Die einzelnen Datenelemente müssen

durch Kommas (,) getrennt werden. Das Ende der Daten muss nicht gekennzeichnet werden. Das Zeilenende beendet die Dateneingabe. Die Daten werden immer an der Stelle in den Maschinencode eingefügt, an der sie stehen.

Daten können alle Zahlenarten sein. Sie können also hexadezimal, dezimal oder binär erfolgen. Zusätzlich ist es möglich, einen kompletten Text (String) einzugeben. Er muss zu diesem Zweck zwischen Hochkommas (') stehen. Zwischen jedem Element muss ein Komma stehen.

Beispiel

```
DB   2, 33h, 255-3Eh, 2Eh, 0100 1110b, 33d, 'Copyright (c)'
DB   C0h, F9h, A4h, 2, 82h, Fh, BFh, 8Ch, 86h, AFh, A3h, Abh
DB   'Wind', 3, 'km/h', 9   ; für eine Zeile im Display
```

Ein jedes Zeichen, das zwischen den Hochkommas steht, wird mit seinem ANSI-Code übersetzt. D. h. der Assembler unterscheidet in diesem Fall zwischen Groß- und Kleinschreibung; auch Leerzeichen werden übersetzt. Ferner sind Berechnungen bei der Dateneingabe möglich.

Einbindung in das Programm

Die Daten werden immer an der Stelle in den Maschinencode eingefügt, an der sie stehen.

Da die Daten im Allgemeinen vom Programm verwendet werden, ist es empfehlenswert, die Daten mit einem Namen = Label zu versehen, damit z. B. der Datenpointer mit dem Tabellenanfang geladen werden kann.

Beispiel

```
dptr = #Tabellenanfang
r2 = #Tabellenende – Tabellenanfang
for r2
    clr a: movc a, @a+dptr: inc dptr: acall Anzeige
next
ret
Tabellenanfang:
    DB   C0h, F9h, A4h, B0h, 99h, 2, 82h, Fh, BFh
Tabellenende:
```

Beispiel

Das folgende Beispiel sendet eine Versionsmeldung (Text mit 0 Abschluss) über den seriellen Port an den PC. Das Bit *Sendebusy* wird im seriellen Interrupt gelöscht.

```
#Bit Sendebusy
sjmp Initialisierung
Serieller Interrupt: ; = Interruptroutine
    if bit TI then: clr TI: clr Sendebusy: end if
reti
Initialisierung:
; Timer 2 als Baudratengenerator 56000 Baud, Timer 2 Run
T2CON = #34h: RCAP2L = #E5h: RCAP2H = #FFh
SCON = #50h     ; für f = 24 MHz
; Interruptfreigabe:
setb ES: setb EA: setb PS     ; ES Priorität
Loop      ; bei Tastendruck an P2.0 gegen Masse
    if not bit P2.0 then: Kommunikation mit PC aufbauen: end if
end loop
Kommunikation mit PC aufbauen:    ; Subroutinenlabel
    r2 = #Ende - Anfang
    loop
        a = r2: movc a, @a+pc
    Anfang:
        if a = #0 then ret  ; Tabellenende
        SBUF = a: setb Sendebusy: jb Sendebusy, hier
        inc r2
    end loop
    Ende:           ; Daten mit 0 Abschluss
DB 'Version 11.18', 0        ; 13 Byte Sendedaten
```

16. Include-Dateien

Include-Dateien sind Dateien, die der Assembler während des Compilierens in den Quelltext lädt und an der aktuellen Stelle einfügt.

Syntax

#Include *Dateiname*

Beispiel

#Include Auswertung.a51

Der *Dateiname* muss vollständig genannt werden und kann Laufwerks- und Pfadangaben beinhalten. Weil Laufwerksangaben im Dateinamen mit einem Doppelpunkt durch den Assembler in zwei Befehle zerlegt würden, müssen Sie in diesem Fall den Dateinamen zwischen Hochkommas stellen. Include-Dateien können eigenständige MCS51-Programme oder Programmfragmente sein, sie dürfen aber keine CPU-Anweisung beinhalten. Das Format muss den Regeln des Assemblers entsprechen. Include-Dateien können Label beinhalten, Variablendefinition, Datenblöcke, ORG-Anweisungen etc. Es muss jedoch durch den Anwender sichergestellt sein, dass sämtliche Anweisungen eindeutig sind, d. h. derselbe Label darf nicht zweimal erscheinen. Label in der Includedatei können durch das Hauptprogramm in gleicher Weise angesprungen werden wie Label im Hauptprogramm durch Sprünge aus der Includedatei heraus.

Die #Include-Anweisung kann an jeder Stelle des Programms erfolgen. Die Include-Datei wird immer an der Stelle eingefügt, an der die #Include-Anweisung auftritt.

Sie können die Include-Datei einfach finden und einfügen durch Mausklick auf das Menü:

Datei -> Include Datei einfügen

Beispiel

#Include 'd:\as51\include\looptest.a51'

Include-Dateien können wieder #Include-Anweisungen beinhalten, d.h. sie können beliebig oft verschachtelt sein. Damit Sie sich im Listing oder im Falle einer Fehlermeldung zurecht finden, steht vor der Zeilennummer durch einen Doppelpunkt getrennt der Include-Level. 1: steht für das Startprogramm, 2: für die erste Include-Datei, 3: für die zweite etc.

Beispiele aus dem Listing

```
000A 7F 0A    1:5  mov r7, #10
; fünfte Zeile im Quellcode der Startdatei
0042 20 93 14 2:16 if not bit p1.3 then
; sechzehnte Zeile im Quellcode der ersten Includedatei
0A54 B8 3E 34 3:8  loop while r0 = #3Eh
; achte Zeile im Quellcode der zweiten Includedatei
```

16.1 #Part of

Die *#Part of* Anweisung sollte in jeder Include-Datei als erste Zeile vorhanden sein, muss aber nicht. Sie ermöglicht das Compilieren aus der Include-Datei heraus. Der Start-Dateiname ist der Projektname der Datei, die die CPU-Anweisung besitzt.

Syntax

#Part of *Start-Dateiname*

Sie können den Projektnamen einfach finden und einfügen durch Mausklick auf das Menü:

Datei -> Projektnamen einfügen

Danach können Sie das ganze Programm mit der Taste F5 compilieren.

Kontrollstrukturen

Kaum ein Programm kommt ohne die Verwendung von bedingten Sprüngen als Reaktion auf Ereignisse oder die Verwendung von Schleifen zur Wiederholung von Befehlsfolgen aus. Mühselig werden Flags abgefragt, Sprungziele konstruiert und Sprünge konstruiert. Viele Befehle müssen nachgeschlagen werden, das Programm verliert an Übersichtlichkeit, die Struktur ist nicht mehr erkennbar und das Programmieren wird zur Qual.

Der Assembler nun entbindet Sie von dieser Kleinarbeit, indem er Ihnen die Möglichkeit zur Verfügung stellt, hochsprachenähnliche Strukturen zu verwenden ohne die Vorzüge der Maschinenebene zu verlassen. Die Programme werden übersichtlicher und pflegeleichter.

Zu den Kontrollstrukturen gehören

Entscheidungen:	If-Strukturen
Zählschleifen:	For-Next-Strukturen
Schleifen:	Loop-Strukturen

Entscheidungen werden mit bedingten Sprüngen gebildet, z. B.

jump if bit is set: jb bit, rel
oder
compare and jump if not equal: cjne a,#data,rel etc.

Zählschleifen führen einen Befehsblock n-mal aus. Der zugrunde liegende Befehl ist der Befehl

decrement and jump if not equal: djnz rn,rel.

Bedingte Zählschleifen wiederholen einen Befehlsblock so lange, bis eine gewünschte Bedingung eintritt. Es liegen ihnen die bedingte Sprünge zu Grunde z.B.

jump if bit is set: jb bit,rel
oder
compare and jump if not equal: cjne a,#data,rel etc.

Beschränkte Operandenwahl

Im Gegensatz zu Hochsprachen können Sie als Argumente nicht beliebige Operanden verwenden, sondern nur diejenigen, die der zugrunde liegende Maschinenbefehl unterstützt. Doch auch hier hilft Ihnen der Assembler mit der integrierten Hilfe weiter: Bewegen Sie den Cursor auf das Schlüsselwort, drücken Sie F1 und sie bekommen die zulässige Syntax mit allen erlaubten Operanden genannt.

Das Carry Bit c

If–Strukturen mit Ausnahme von *if a = #0* und *if a > #0* können das Carry ändern, ebenso Loop-Strukturen, die eine Bedingung verwenden. Beachten Sie daher, dass das Carry nach dem Beenden der Struktur einen anderen Wert haben kann.

17. IF Strukturen

Syntax

If *Bedingung* [Then]

[*Anweisungen*]

[Elseif *Bedingung-n* [Then]

[*elseif-Anweisungen*]]

[Else

[*else-Anweisungen*]]

End If

Die Syntax für die **If ... Then ... Else**-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung

Bedingung	Erforderlich. Ein Byte-Vergleich oder Bit-Test der folgenden Art:
	a = *direct** (Direct-Adresse, Variable oder SFR)
	a = #data*
	r*n* = #data* (n = 0 – 7)
	@r*i* = #data* (i = 0 oder 1)
	a = #0
	a > #0
	bit *Bitname*
	not bit *Bitname*
	bit c

not bit c
** können das Carry c ändern.*

Anweisungen	Optionale *Anweisungen* in Form eines Blocks. Eine oder mehrere durch Doppelpunkte getrennte Befehle, die ausgeführt werden, wenn die Bedingung zutrifft.
Bedingung-n	Optional. Dieselbe Bedeutung wie *Bedingung*.
Elseif-Anweisungen	Optional. Eine oder mehrere Anweisungen, die ausgeführt werden, wenn die zugehörige Bedingung (*Bedingung-n*) zutrifft.
Else-Anweisungen	Optional. Eine oder mehrere Anweisungen, die ausgeführt werden, wenn keine der Bedingungen (*Bedingung*-Ausdruck oder *Bedingung-n*-Ausdruck) zutrifft.

Eine Anweisung für einen If-Block muss die erste Anweisung in einer Zeile sein. Der If-Block muss mit einer End If-Anweisung beendet werden.

Die Abschnitte Else und Elseif sind optional. In einem If-Block können Sie beliebig viele Elseif-Abschnitte verwenden, nach einem Else-Abschnitt sind jedoch keine Elseif-Abschnitte zulässig. If-Blöcke dürfen verschachtelt sein, also selbst wieder If-Blöcke enthalten. Der If-Block kann mit Exit If vorzeitig verlassen werden.

Bei der Ausführung eines If-Blocks wird zunächst *Bedingung* überprüft. Trifft *Bedingung* zu, so werden die Anweisungen im Anschluss an das optionale Then ausgeführt. Trifft *Bedingung* nicht zu, so werden die Elseif-Bedingungen (sofern vorhanden) der Reihe nach ausgewertet. Sobald eine dieser Bedingungen zutrifft, werden die Anweisungen im Anschluss an das optionale Then ausgeführt. Trifft keine der Elseif-Bedingungen zu (oder sind überhaupt keine Elseif-Abschnitte vorhanden), so werden die Anweisungen im Anschluss an Else ausgeführt. Sobald die Anweisungen nach einem optionalen Then- oder Else-Abschnitt ausgeführt wurden, setzt das Programm die Ausführung mit der Anweisung im Anschluss an End If fort.

Da für Then und Else relative Sprünge verwendet werden, können dazwischen maximal 127 Bytes (ca. 60 Befehle) stehen. Benötigen Sie mehr Platz, müssen Sie Teile der Befehle als Subroutine zusammenfassen und sie mit einem Call-Befehl aufrufen.

Beispiel

Im nachfolgenden Beispiel steht in der Variablen *Modus* der Betriebszustand des Programms. Die Variable wird mit drei Konstanten verglichen. Das Programm verzweigt bei Gleichheit in die zuständigen Unterprogramme. Trifft keine der drei Bedingungen zu, wird der Else-Teil mit der Fehlermeldung ausgeführt.

```
r0 = Modus
if @r0 = # Daten einlesen then
    acall Tastenabfrage bei Datenlesen
elseif @r0 = # Daten senden then
    acall Tastenabfrage bei Datensenden
elseif @r0 = # Speicher löschen then
    acall Memory erase
else
    Fehler = # 1Bh
    acall Fehlerausgabe
end if
```

Anmerkung

Der Assembler verwendet für die Bedingungen *A = direct, A = #data, Rn = #data und @Ri = #data* den CJNE-Befehl, d.h. das Carry wird bei diesen Vergleichen eventuell verändert, die Inhalte von Akku und der Register bleiben erhalten.

Für die Bedingungen *if a = #0 und if a > #0* verwendet der Assembler den JNZ- bzw. den JZ-Befehl, der das Carry nicht ändert.

Der Bittest mit *if bit Bitname* und *if not bit Bitname* erfolgt mit den Befehlen JNB bzw. JB, der Carrytest mit *if bit c* und *if not bit c* erfolgt mit JNC bzw. JC. Auch hierbei bleibt das Carry erhalten.

17.1. Then Ret

Ein besondere Syntax gestattet das bequeme Verlassen einer Subroutine. Das Schlüsselwort **Then** ist in diesem Fall erforderlich.

Syntax

If *Bedingung* Then Ret

Da diese Abfrage in Subroutinen sehr häufig auftritt, wurde ihr eine eigene Syntax eingerichtet. Sie ersetzt die Konstruktion.

```
If Bedingung then
    ret
end if
```

17.2. Exit If

Ein If-Block kann vorzeitig mit Exit If verlassen werden. Exit If kann in einem If-Block mehrmals auftreten.

Syntax 1

```
If Bedingung [Then]
    [Anweisungen]
    If Bedingung [Then]
        [Anweisungen]
        Exit If
    End If
    [Anweisungen]
End If
```

Weil diese Anweisung selbst Ergebnis einer Bedingung ist, wird mit der Syntax 1 stets der übergeordnete If-Block verlassen. Exit if außerhalb einer If-Struktur erzeugt den Fehler: *Exit If nicht möglich*

Syntax 2

```
If Bedingung [Then]
    [Anweisungen]
    If Bedingung [Then] Exit If
```

```
    [Anweisungen]
End If
```

Häufig kommt es vor, dass das Eintreten einer Bedingung sofort zum Verlassen des If-Blocks ohne Ausführung weiterer Anweisungen führen soll. Für diesen Fall gibt es die kurze, einzeilige Syntax 2.

Beispiel

Das folgende Beispiel verlässt den If-Block, wenn das Bit *ok* in der Routine Test 1 gesetzt wird.

```
#Byte Modus
#Const ist Test = 1
a = Modus
if a = #ist Test then
    acall Test 1      ; Rückgabe: Bit ok
    if Bit ok then exit if      ; ok = 1
    acall Test 2                ; ok = 0
end if
```

Hinweis: Folgende If-Konstruktion ist unschön, da der Assembler *else* mit *sjmp* übersetzt. In diesem Fall kann man statt *else* gleich den *ret* Befehl verwenden.

```
Subroutine:
if a = #0 then
    . . . Befehle
else
    . . . Befehle
end if
ret
```

Besser ist:

```
Subroutine:
if a = #0 then
    . . . Befehle
    ret
End if
```

. . . Befehle
ret

Rufen Sie auch niemals vor einem *ret* Befehl eine Subroutine auf, verwenden Sie statt dessen den *jmp* Befehl, da ja am Ende der Subroutine bereits der *ret* Befehl steht.

Schlecht:

```
acall Subroutine
ret
```

besser:

```
ljmp Subroutine
```

18. LOOP-Schleifen

Syntax

Loop [While *Bedingung*]
[*Anweisungen*]
[Exit Loop]
[*Anweisungen*]

{End Loop | Until *Bedingung*}

Die Syntax für die **Loop ... End Loop**-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung

Bedingung	Erforderlich, wenn nach **Loop** das Schlüsselwort **While** folgt bzw. wenn die Schleife mit **Until** endet. *Bedingung* ist ein Byte-Vergleich oder Bit-Test der folgenden Art:
	a = *direct** (Direct-Adresse, Variable oder SFR)
	a = #data*
	r*n* = #data* (n = 0 – 7)
	@r*i* = #data* (i = 0 oder 1)
	a = #0

	a > #0 bit *Bitname* not bit *Bitname* bit c not bit c ** können das Carry c ändern.*
Anweisungen	Optionale *Anweisungen* in Form eines Blocks. Eine oder mehrere durch Doppelpunkte getrennte Befehle, die in der Schleife ausgeführt werden.

Jeder Loop-Block muss mit Loop und einer eventuellen While-Bedingung starten. Ein Loop-Block muss mit einer End Loop- oder Until-Anweisung beendet werden.

Für die Loop-Snytax gibt es vier möglich Kombinationen:

1. Bei der Verwendung von Loop ... End Loop wird eine **Endlosschleife** ausgeführt, die nur mit Exit Loop verlassen oder mit einem Interrupt unterbrochen werden kann.
2. Loop While ... End Loop erzeigt eine Schleife mit Prüfung der **Laufbedingung** am Anfang des Blocks.
3. Loop ... Until erzeigt eine Schleife mit Prüfung der **Laufbedingung** am Ende des Blocks.
4. Loop While ... Until erzeigt eine Schleife mit Prüfung der **Laufbedingung** am Anfang und am Ende des Blocks.

18.1. Exit Loop

Innerhalb eines Loop ... End Loop Blocks kann eine beliebige Anzahl von Exit Loop Anweisungen an beliebiger Stelle als alternative Möglichkeit zum Verlassen der Schleife verwendet werden. Exit Loop wird oft in Zusammenhang mit der Auswertung einer Bedingung (zum Beispiel If ... Then) eingesetzt und hat zur Folge, dass die Ausführung mit der ersten Anweisung im Anschluss an End Loop bzw. Until fortgesetzt wird.

In verschachtelten Loop-Schleifen übergibt Exit Loop die Steuerung an die Schleife der nächsthöheren Verschachtelungsebene.

Exit Loop kann mit einer If-Bedingung in einer Zeile stehen. Gültige Aussagen und Terme für *Bedingung* finden Sie unter If-Strukturen:

If *Bedingung* [Then] Exit Loop

Beispiel

```
loop                ; wartet auf Tastendruck an Port 3
    if not bit P3.0 then
        clr TR1: acall Tastenauswertung: exit loop
    end if
end loop
```

Beispiel

```
clr F0
loop      ; äußere Schleife
    r5 = #20: a = r5
    loop while a > #0        ; innere Schleife
        inc r5
        if r5 = #40 Then
            setb F0          ; Attributwert setzen
            exit loop        ; innere Schleife verlassen
        end if
        dec a
    loop
until bit F0            ; äußere Schleife sofort verlassen
```

Auch Zählschleifen können mit **LOOP** realisiert werden.
Nachstehendes Beispiel zählt von 17 bis 188 in Einerschritten aufwärts. Die Zählvariable steht in R7 und kann in der Schleife für weitere Aktionen verwendet werden.

```
#Const Start = 17, Ende = 188
r7 = #Start
Loop
  . . .
  inc r7
Until r7 = #Ende + 1
```

19. FOR-Schleifen

Syntax

For *Zähler* [= *Anfang*]
[*Anweisungen*]
[Exit For]
[*Anweisungen*]

Next

Die Syntax für die **For...Next**-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung
Zähler	Erforderlich. Ein Register (R0 – R7), eine Direct-Adresse, ein Spezialfunktionsregister SFR oder eine Byte-Variable .
Anfang	Optional. Einer der folgenden Operanden: Wenn *Zähler* = Rn: Operand = a, direct, #data Sonst: Operand = a, Rn, direct, @Ri, #data
Anweisungen	Optionale *Anweisungen* in Form eines Blocks. Ein oder mehrere durch Doppelpunkte getrennte Befehle zwischen **For** und **Next**, die so oft wie angegeben ausgeführt werden.

Ein For-Block muss mit einer Next-Anweisung beendet werden. Ist *Anfang* angegeben, wird *Zähler* mit dem Wert von *Anfang* beschrieben. Fehlt *Anfang*, wird der aktuelle Inhalt von *Zähler* verwendet. Es können maximal 256 Schleifendurchgänge erfolgen. Da die Prüfung der Schleifenbedingung am Schluss mit dem DJNZ-Befehl stattfindet, wird die Schleife in jedem Fall mindestens einmal durchlaufen. Die Angabe von 0 als *Anfang* bewirkt 256 Schleifendurchgänge. Die For-Schleife ändert das Carry c nicht

Nachdem alle Anweisungen in der Schleife ausgeführt wurden, vermindert das Programm den Inhalt von *Zähler* um 1. Somit sind For-Schleifen immer in Einerschritten abwärts zählende Schleifen. Die Anweisungen in der Schleife werden mit *Next* entweder erneut ausgeführt oder die Schleife wird

beendet, wenn der Inhalt von *Zähler* 0 ist, und die Ausführung wird mit dem auf die Next-Anweisung folgenden Befehl fortgesetzt.

Da für die Schleife relative Sprünge verwendet werden, kann der Abstand zwischen For und Next nur 125 Byte (ca. 60 Befehle) umfassen.

Anmerkung Das Ändern des Wertes von *Zähler* innerhalb einer Schleife kann zur Folge haben, dass der Code komplizierter und schwerer zu testen wird.

Das versehentliche Ändern des Wertes von *Zähler* innerhalb einer Schleife kann zum Absturz der Software oder zu einer Endlosschleife führen.

Das Ändern des Wertes von *Zähler* innerhalb einer Schleife kann bewusst vorgenommen werden, um in n-Schritten die Schleife auszuführen.

Beispiel

```
for r2 = #60d
    Befehle
    dec r2          ; bewirkt 30 Durchgänge
next
```

Aufwärts zählende Schleifen erhält man durch zweimaliges Inkrement des Zählers.

Beispiel

```
for r2 = #60d
    Befehle
    inc r2: inc r2   ; zählt aufwärts von 60 bis 255
next
```

19.1. Exit For

Innerhalb einer Schleife kann eine beliebige Anzahl von Exit For-Anweisungen an beliebiger Stelle als alternative Möglichkeit zum Verlassen der Schleife verwendet werden. Exit For wird oft in Zusammenhang mit der Auswertung einer Bedingung (zum Beispiel If ... Then) eingesetzt und überträgt die Steuerung an die unmittelbar auf Next folgende Anweisung. In verschachtelten For-Schleifen übergibt Exit For die Steuerung an die

Schleife der nächsthöheren Verschachtelungsebene

Exit For kann mit einer If-Bedingung in einer Zeile stehen. Gültige Aussagen und Terme für *Bedingung* finden Sie unter If-Strukturen:

if *Bedingung* [then] exit for

Sie können For ... Next-Schleifen verschachteln, indem Sie eine For ... Next-Schleife innerhalb einer anderen verwenden. Das Argument *Zähler* muss für jede Schleife einen eindeutigen Variablennamen erhalten. Die folgende Konstruktion ist korrekt:

Beispiel

```
for r2 = #5
    for var1
        for var2 = r4
            ...
        next
    next
next
```

Die folgende Befehlsfolge gibt die acht Bits des Akkus seriell über einen Port-Pin aus, der hier mit Dout bezeichnet wird. Gleichzeitig wird die Taktleitung Dclk (ebenfalls ein Port-Pin) gesteuert. Die NOP-Befehle sind für ein Tastverhältnis von 50% erforderlich.

```
#Bit Dout = P1.1, Dclk = P1.2
for r7 = #8
    rlc a: Dout = c
    setb Dclk: nop: nop: clr Dclk
next
```

Für mehr als 256 Zählvorgänge müssen For-Schleifen verschachtelt werden. In diesem Fall multiplizieren sich die Inhalte von *Zähler*. Die folgende For-Kombination erzeugt 500 Schleifendurchgänge.

Beispiel

```
#Byte Zähler low, Zähler high
clr a: Zähler low = a: Zähler high = a
for r7 = #10
    for r6 = #50
        a = Zähler low: add a, #1: Zähler low = a
        a = Zähler high: addc a, #0: Zähler high = a
        ; oder kürzer
        ; if bit c then: inc Zähler high: end if
    next
next
```

Am Ende der Schleife steht in Zähler high der Wert 01h, in Zähler der Wert F4h: 1F4h = 500d.

Beispiel

Vierfachgenaue Addition zweier Longintegerwerte (4 Byte) Summand A und Summand B. Summand A wird mit dem Ergebnis überschrieben.

```
r0 = #Summand A          ; low Byte
r1 = #Summand B          ; high Byte
clr c
for r2 = #4
    a = @r0: addc a,@r1: @r0 = a
    inc r0: inc r1
next
```

Diese Zeilen werden in den folgenden Maschinencode übersetzt:

```
mov r0, #Summand A
mov r1, #Summand B
clr c
mov r2, #4
Label:
    mov a, @r0
    addc a, @r1
    mov @r0, a
```

```
    inc r0
    inc r1
djnz r2, Label
```

Compiler

20. Compiler

Der Compiler übersetzt den Inhalt des aktuellen Fensters und der Include-Dateien in ausführbaren Maschinencode in einem Schritt. Fehlermeldungen und Warnungen werden am Ende in einem separaten Fenster ausgegeben. Das Listing gibt genauere Hinweise auf mögliche Fehlerquellen. Bitte schauen Sie es sich nach dem Compilieren an.

Der Compiler kann vier Arten von Dateien erzeugen.

Hexdatei

Dateiendung: *.h51. Der Maschinencode wird im Intel-Hex-Format gespeichert. In ihm befinden sich die Adressen, der Maschinencode und eine Prüfsumme in hexadezimaler Form. Das ist die gebräuchlichste Form für Programmiergeräte.

Mit dem Assembler können Sie den Inhalt (nicht das Format) der Hex-Datei ansehen. Bereiche von mehren Byte mit Inhalt FFh können Sie zum Komprimieren des Quelltextes verwenden.

Die Form einer Hexdatei:

```
:08000000058E058FC2A78045A3
:02000B0080195A
:20002300021136D57F23  . . .  D5FC
:200043007C07757C04D2  . . .  7D27
:00000001FF
```

Binärdatei

Dateiendung: *.b51. Der Maschinencode wird binärcodiert gespeichert. Die Adresse eines jeden Byte entspricht der Platznummer in der Binärdatei, wobei der erste Platz die Adresse 0000 repräsentiert. Wenn Sie mit der Org-Anweisung hohe Adressen nennen, kann die Dateilänge auch bei kurzen Programmen sehr groß werden. Die Dateigröße ist bei kompakten Programmen stets kleiner als die der Hexdatei.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ansi
0000	05	8E	05	8F	C2	A7	80	45	00	00	00	80	19	00	00	00	·□·□Â§€E···€····
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	····
0020	00	00	00	02	11	36	D5	7F	23	75	7F	14	D2	6E	15	6B	·····6Õ□#u□·Òn·k
0030	D5	7E	07	75	7E	14	D2	7F	D2	48	D5	7D	0F	75	7D	FA	Õ~·u~·Ò□ÒHÕ}·u}ú
0040	D2	79	D5	7C	07	75	7C	04	D2	7E	C2	48	32	C2	A6	C2	ÒyÕ|·u|·Ò~ÂH2Â¦Â
0050	90	43	96	08	E4	F8	F6	D8	FD	D2	5D	75	7F	14	75	7E	□C-·äøöØýÒ]u□·u~

Bereiche von mehren Byte mit Inhalt 00 können Sie zum Komprimieren des Quelltextes verwenden.

Listingdatei

Dateiendung: *.l51. In der Listingdatei finden Sie das, was der Compiler mit Ihrem Code angefertigt hat. Widmen Sie besonders zu Anfang dieser Datei eine große Aufmerksamkeit, da Sie hier die Arbeitsweise des Compilers und Ihre Fehler erkennen können.

```
0FCE             2:858  Manuelleinstellungen aus an:
0FCE  79 00      2:859    r1 = #0: Kopfvariable holen
0FD0  12 0B 5C   2:859
0FD3  BF 01 09   2:860  if r7 = #1 then        ; an/aus einstellen
0FD6  B4 01 01   2:861    if a = #1 then ret
0FD9  22
0FDA  74 01      2:862    a = #1: ljmp EEPROM beschreiben
```

Adresse | Code maximal 3 Byte | Nummer der Include-Datei und Zeilen nummer im Quelltext | Quelltext mit Kommentaren und eventuellen Fehlermeldungen

Symboldatei

Dateiendung: *.s51. In der Symboldatei sind die Variablen und Label und deren Adressen gespeichert. Die Symboldatei kann im MCS51-Simulator als Watchvariablenliste geladen und verwendet werden. Sie gestattet ein einfacheres Debuggen Ihres Quellcodes.

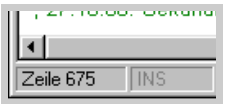
Der Inhalt einer Symboldatei gibt als erstes den Bezeichner an gefolgt vom Ort oder Art und als drittes die Adresse oder Wert:

"4 Hz Teiler","IRAM","7Dh"
"ist Minute","BIT","7Dh"
"Auswahl","CONST","04h"
"Displaybeleuchtung aus ","DEF","clr Backlight"
"Temperatur berechnen","LABEL","16C2h"

21. Fehlermeldungen

Trifft der Compiler auf eine Unstimmigkeit im Quelltext, generiert er eine Warnung oder eine Fehlermeldung. Erschrecken Sie nicht über die Fehlermenge, wenn Sie ein größeres Programm zum ersten Mal compilieren. Das ist normal. Zu den Hauptfehlern zählen Rechtschreibfehler, nicht deklarierte Variablen, Adressbereichsüberschreitung, String-Fehler, nicht abgeschlossene Kontrollstrukturen etc.

Meist ist der Assembler in der Lage, den Fehler konkret zu benennen. Die Fehlermeldung besteht aus drei Teilen. Der erste Teil ist die Nummer der Include-Datei gefolgt von der Zeilennummer. Die Projektdatei hat immer die Nummer 1. Um schnell zur fehlerhaften Zeile zu springen, klicken Sie in der Statuszeile unten links auf das Feld *Zeile* und geben Sie die Zeilennummer ein.



21.1 Häufige Fehler

Warnungen/Fehler:

Zeile 1:53 mov 20s Zähler, #20 ;>err1: ungültige Operandenkombination

Ursache. Die Variable 20s Zähler ist entweder nicht definiert oder es liegt ein Schreibfehler vor.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 2:75 sjmp Submenümarkierung setzen ;>err21: relativer Adressbereich überschritten

Ursache. Ein Sprung mit *sjmp* ist nur im Bereich von ±127 Byte möglich. Ersetzen Sie *sjmp* durch *ajmp* oder *ljmp*.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 2:775 ajmp Schattierungsmenüanzeige ; >err5: Blocküberschreitung

Ursache. Der Sprung führt über die Blockgrenze. Ersetzen Sie *ajmp* durch *ljmp*.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:140 acall Anzeige in Submenüs ;>err5: Blocküberschreitung

Ursache. Der Subroutinenaufruf führt über die Blockgrenze. Ersetzen Sie *acall* durch *lcall*.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:580 DB 2, '[x] Automatik, 12 ;>err19: Formatfehler in Tabelle

Ursache. Es fehlt das zweite Hochkomma am Ende der Zeichenkette. Ergänzen Sie es.

~ ~ ~ ~ ~ ~ ~ ~

Offene Loop-Schleife(n)

Ursache. Fehlendes End Loop. Da der Assembler nicht weiß, wo das End Loop fehlt, wird bei diesem Fehler keine Zeilennummer genannt. Benutzen Sie die Suchfunktion um nach allen Vorkommen von Loop im Quelltext zu suchen.

~ ~ ~ ~ ~ ~ ~ ~

Offene If-Schleife(n)

Zum Auffinden des Fehles schreiben am Ende des Programms temporär die Zeile *End if* und compilieren Sie neu. Nun ist die If-Struktur geschlossen, aber der relative Adressbereich überschritten. Diese Stelle wird nun mit der Zeilennummer genannt, und Sie können den Fehler korrigieren.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:80 end if ;>err22: Else/End IF ohne IF-Beginn

Ursache. Der Compiler findet eine Else oder End If Anweisung ohne einen If-Beginn. Gehen Sie zu der Zeile und ergänzen Sie ihn.

~ ~ ~ ~ ~ ~ ~ ~

Offene For-Schleife(n)

Ursache. Fehlendes Next. Da der Assembler nicht weiß, wo das Next fehlt, wird bei diesem Fehler keine Zeilennummer genannt. Benutzen Sie die Suchfunktion um nach allen Vorkommen von Next im Quelltext zu suchen.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:227 next ;>err22: Next ohne For-Beginn

Ursache. Der Compiler findet eine Next Anweisung ohne einen For-Beginn. Gehen Sie zu der Zeile und ergänzen Sie ihn.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:78 S0INT: ;>err11: Adresse liegt in eventuell beschriebenem Bereich

Ursache. Die Interruptvektoren haben feste Adressen. Vor dieser Adresse stehen zu viele Befehle. Überspringen Sie die Interruptroutine mit einem Sprungbefehl.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:88 r7 = #1234h ;>err13: #data zu groß

Ursache. Register 7 kann nur Werte von 0 bis FFh aufnehmen.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 1:89 a = #'FW' ;>err31: Wert zu groß

Ursache. Zeichenketten werden in Zahlen (ANSI-Code) umgerechnet. Der Akku a kann nur ein Zeichen aufnehmen.

~ ~ ~ ~ ~ ~ ~ ~

Zeile 2:271 #Bit Lichtauswertung ;>err54 Bit 'Lichtauswertung' ist bereits definiert

Ursache. Mehrfachdeklaration einer Variablen. Gehen Sie der Sache nach.

~ ~ ~ ~ ~ ~ ~ ~

Bei allen anderen Fehlern ohne Zeilennummern öffnen Sie die Listing-Datei *.L51 und suchen Sie nach dem Fehlercode. Alle Fehler im Quellcode werden im Listing gespeichert.

21.2 Der Statusbericht

Der Statusbericht wird nach dem Ende der Compilierung angezeigt. Er nennt alle erkannten Fehler. Sie sollten ihn stets aufmerksam zur Kenntnis nehmen. Er gibt nicht nur Warnungen und Fehler aus, sondern gestattet Ihnen Adressüberschreitung von Bit– und Byte-Variablen zu erkennen. Ferner kann ein Befehl durch einen vergessenen Doppelpunkt am Ende ungewollt zu einen Label werden. Mit dem Warnhinweis ungenutzter Routinen können Sie Speicherplatz sparen und Ihr Programm verschlanken.

```
Warnungen/Fehler:
Unbenutzte Label
  Nächst höhere Zeile: Zeile 2:253,  Adresse A99h
Unbenutzte Bit
  ist kalt: 69h
Unbenutzte Byte
  Zähler: 6Bh
1938 Zeilen übersetzt. Code Ende bei 0F4Ah.
```

Im Statusbericht finden Sie die Adressen der Bit-Variablen, der Byte-Variablen, die Namen der Konstanten mit ihren Werten, die Adressen der Label und schließlich die verwendeten Definitionen.

22. Dateiendungen

Der Compiler erzeugt sechs Arten von Dateien. Für diese können Sie Art der Endung frei wählen. Die Änderungen werden in die Datei **as51.cfg** beim Beenden des Assemblers geschrieben und beim Neustart daraus geladen. Fehlt diese Datei, startet der Assembler mit den Voreinstellungen.

Die Taste **OK** übernimmt Ihre Änderung. **Standard** wählt die Voreinstellungen. **Abbrechen** schließt das Fenster ohne Übernahme eventueller Änderungen

Programmcode

Endung für die Quelldateien. Das Assemblerprogramm wird im reinen Textformat gespeichert und kann von jedem Texteditor geladen und geändert werden. Standardendung ist **a51**.

Listing

In der Listingdatei finden Sie das, was der Compiler mit Ihrem Code angefertigt hat. Sie sehen darin die Adresse, den Maschinencode und Ihren Quelltext, den Programmcode. Die Listingdatei ist eine reine Textdatei. Standardendung ist **l51**.

Binärcode

In der Binärdatei steht der reine Maschinencode so, wie ihn der Controller im ROM vorfindet. Der Binärcode ist also ein 1:1 Abbild des ROM-Inhalts. Das erste Byte entspricht der Adresse 0000 im ROM etc. Standardendung ist **b51**.

Hexcode

In der Hexdatei befinden sich die Adressen, der Maschinencode und eine Prüfsumme in hexadezimaler Form. Eventuelle Programmlücken werden ausgespart. Die Hexdatei ist eine reine Textdatei. Standardendung ist **h51**.

Symboldatei

In der Symboldatei sind die Variablen, Konstanten und Label und ihre Adressen gespeichert. Die Symboldatei kann im MCS51-Simulator als Watchvariablenliste geladen und verwendet werden. Sie gestattet ein einfacheres Debuggen Ihres Quellcodes. Die Symboldatei ist eine reine Textdatei. Standardendung ist **s51** bzw. **wvar**.

Disassembling

Das Disassembling ist das Ergebnis des Disassemblers. Es hat eine ähnliche Form wie das Listing, d.h. in jeder Zeile finden sich Adresse, Maschinencode und die Befehlsmnemonik. Das Disassembling ist eine reine Textdatei. Standardendung ist **d51**.

23. Disassembler

Der Disassembler übersetzt compilierten Maschinencode für die MCS51-Controller in ein Programmlisting. Der Maschinencode kann im Intel-Hex-Format oder im Binärformat vorliegen. Der Disassembler erkennt automatisch das entsprechende Format, sodass auch Erzeugnisse anderer Assembler in den Quelltext gewandelt werden können.

Zur Interpretation der Spezialfunktionsregister SFR benutzt der Disassembler den aktuellen CPU-Treiber, der im Menü *Optionen* / *Controller festlegen* wählbar ist.

Der Disassembler lässt sich für seine Arbeitsweise konfigurieren. Im Menü *Optionen* / *Disassembler konfigurieren* können Sie die Auswahl treffen

1. Disassembling mit Adresse, OpCode ohne Label erzeugt ein unstrukturiertes Ergebnis vergleichbar mit dem Listung.

```
001F  75  7F  64   mov 7Fh, #64h    ; 100d = 'd'
0022  75  7E  19   mov 7Eh, #19h    ; 25d
0025  90  00  78   mov dptr, #0078h
0028  63  96  04   xrl EECON, #04h  ; 4d
002B  90  00  AD   mov dptr, #00ADh
002E  63  96  04   xrl EECON, #04h  ; 4d
0031  75  7C 01    mov 7Ch, #01h    ; 1d
0034  80  FE       sjmp 0034h       ; End
0036  D5 7F  3E    djnz 7Fh, 0077h
```

2. Disassembling ohne Adressen, ohne OpCode aber mit Label erzeugt eine Form, die wie der Quellcaode aussieht. Labelnamen bestehen aus ihren Adressen mit einem vorgestellten L.

```
L0034h:
sjmp L0034h              ; End
djnz 7Fh, L0077h
    mov 7Fh, #64h        ; 100d = 'd'
    mov a, 7Eh
cjne a, #08h, L0048h     ; 8d
```

```
    mov a, 7Ch
cjne a, #01h, L0048h        ; 1d
    clr TR2
L0048h:
djnz 7Eh, L0077h
    mov 7Eh, #19h           ; 25d
djnz 7Ch, L0077h
    setb TR2
```

3. Adressen im Hex-File sortieren. Hex-Dateien anderer Hersteller können Daten in nicht aufsteigenden Adressen beinhalten, d. .h die vorausgehende Zeile in der Datei kann eine größere Adresse als die nachfolgende haben. Sollte eine solche Datei zum Disassemblieren vorliegen, können Sie den Programmcode in aufsteigender Form generieren.

Speichern des Disassembling

Das Disassembling kann als eigenständiges Format (*.d51) oder als Listing gespeichert werden (*.l51).

Grenzen des Disassemblers

Der Disassembler kann Programmcode nicht von Tabellen unterscheiden. Der Anwender sollte chaotischen, sinnlosen Code als Tabelle interpretieren.

24. Die Assistenten

Assistenten helfen Ihnen bei der Erstellung eines Programmgerüsts. Da für die Entwicklung der Assistenten enorm viel Zeit nötig ist, sind in der derzeitigen Version des Assemblers noch nicht alle existierenden Controller berücksichtigt.

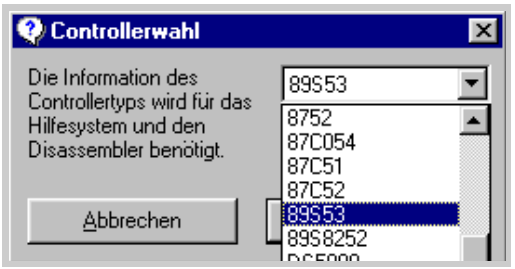
Die Bedienung des Assistenten ist einfach. Nach der Auswahl des gewünschten Controllers im Menü *Assistenten* geben Sie die Taktfrequenz der Hardware an, auf der das Programm laufen soll. Diese Angabe ist die Basis für die Berechnung der Baudraten und der Timer-Überlaufraten.

Die Wirkung jeder Änderung und jeder Einstellung sehen Sie im rechten Code-Fenster. Die Taste *Übernehmen* kopiert diesen Text an die aktuelle Cursorposition des aktiven Fensters. Sie können auch später Änderungen an Ihrem Programm mit Hilfe des Assistenten vornehmen, indem Sie den betreffenden Abschnitt in dem Fenster markieren und ihn aus dem Fenster kopieren. Dabei verwenden Sie die üblichen Tastenkombinationen Strg + C zum Kopieren und Strg + V zum Einfügen.

Die Taste *Abbrechen* verbirgt das Fenster nur, lässt aber den Inhalt unverändert, sodass Sie nach erneutem Start des Assistenten den Inhalt unverändert vorfinden. Möchten Sie den kompletten Inhalt löschen und alle Änderungen rückgängig machen, so kicken Sie auf das Kreuz ☒ rechts oben, um das Fenster zu schließen.

25. Das Hilfesystem

Wenn Sie das Menü *Optionen/Controller festlegen* anklicken, öffnet sich das Fenster *Controllerwahl*. In der Liste werden die Namen aller vorhandener CPU-Treiber genannt. Für die Spezialfunktionsregister und deren Bits der gängigsten Controller existiert eine Hilfedatei, in der Sie Näheres über die Funktionsweise und Verwendung nachschlagen können.



Dieses Hilfesystem wird ständig erweitert und ergänzt. Die Aktualisierungen finden Sie im Internet unter

`www.Controllertechnik.de`

Wenn Sie Hilfe zu einem Befehl oder einer Struktur haben möchten, bewegen Sie den Cursor auf das Wort und drücken die Taste F1.

26. Einführung in die MCS51-Programmierung

26.1 Demonstrationsprogramm

Das folgende Demonstrationsprogramm zeigt die wesentlichen Elemente des Assemblers und deren Verwendung.

Erste Anweisung muss die #cpu-Anweisung sein.

```
#cpu = 80c32          ; @ 12 MHz
```

Variablendeklarationen müssen mit #Bit, #Byte, #Word und #Const erfolgen.

```
#Bit Empfang, Sendebusy
```

Speicherbeireich reservieren, falls erforderlich:

```
#Exclude Byte 50h - 58h
```

Den Bit-, Byte und Word-Variablen können Adressen zugewiesen werden.

```
#Byte Summe = 50h, Tempo
```

Konstanten muss immer ein Wert zugewiesen werden.

```
#Const Wert = 60
```

Definitionen erfolgen mit #def.

```
#def rotate left {a = P1: rl a: P1 = a}
#def rotate right {a = P1: rr a: P1 = a}
```

Bei größeren Programmen, d.h. bei Adressen über 0400h sollte die #use lcall Anweisung folgen.

```
#use LCALL
```

; ; ; Programmstart ; ; ;

Dem ersten Befehl wird immer die Adresse 0000h zugewiesen

```
ajmp Initialisierung
```

Interrupts werden mit ihren Namen und dem Doppelpunkt festgelegt. Diese Namen besitzen feste Adressen.

```
SINT:        ; serieller Interrupt hat die Adresse 0023h
```

Sprungziele müssen im Programm mit einem Doppelpunkt gekennzeichnet sein.

```
ajmp Serieller Interrupt
```

Hier folgt der Timer 2 Interrupt:

```
Timer 2:     ; Timer 2 Interrupt hat die Adresse 002Bh
    ajmp Timer 2 Interrupt
```

Label, d.h. Sprungziele müssen immer einzeln in einer Zeile stehen und mit einem Doppelpunkt enden.

```
Initialisierung:
    mov SP, #7Fh  ; Stackbeginn
    clr a  ; Variablen und Bits löschen
    for r0 = a: @r0 = a: next
```

Statt des Mov-Befehls können Sie das Gleichheitszeichen (=) verwenden.

```
r0 = #20h      ; steht für mov r0, #20h
```

Schleifen werden mit Loop eingeleitet.

```
loop
```

Wenn Sie mehrere Befehle in eine Zeile schreiben, trennen Sie sie mit einem Doppelpunkt voneinander.

```
mov @r0, a: inc r0
```

```
until r0 = #80h
```

Initialisierung des seriellen Port.

```
SCON = #50h          ; Modus 1, asynchron, 10 Bit, Baudrate
                     ; Timer 1/2 Überlauf, Datenempfang freigeben
mov TMOD, #20h       ; Timer 1 Autoreloadmodus
mov TH1, #F3h        ; Reloadwert für 2400 Baud
setb TR1             ; Timer 1 Start
```

Initialisierung des Timer 2:

Timer 2 ist ein 16-Bit Aufwärtszähler mit automatischer Nachladung aus den Reload-Registern bei Überlauf. Die Überlauffrequenz beträgt 10000 Hz, die Periodendauer 0,1 ms.

```
RCAP2L = #9Ch: RCAP2H = #FFh: setb TR2
```

Initialisierung der Interrupts

```
setb ES              ; seriellen Interrupt freigeben
setb ET2             ; Timer 2 Interrupt freigeben
setb PT2             ; Priorität für Timer 2 Interrupt
setb EA              ; globale Interruptfreigabe

                     ; ; ; Initialisierung Ende ; ; ;
```

Hauptprogramm Beginn

```
loop                 ; Routine für seriellen Empfang
    if bit Empfang then
        clr Empfang
```

Subroutinenaufrufe erfolgen einfach mit der Nennung des Namens:

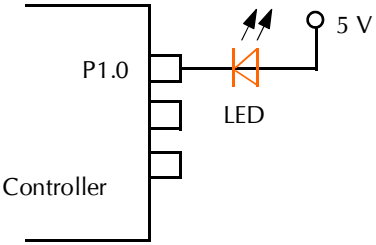
```
        Serielle Daten auswerten
    end if
```

```
        end loop
Hauptprogramm Ende

Es folgen Subroutinen.

        Serielle Daten auswerten:
            a = SBUF
            ; weitere Befehle
        ret

        Sende:   ; Sendewert in a
            jb Sendebusy, hier        ; auf Sendefreigabe warten
            SBUF = a: setb Sendebusy
        ret

        Serieller Interrupt:
            ; Herkunft ermitteln
            if bit TI then    ; Senden
                    clr TI: clr Sendebusy
            else  ; Empfangen
                    clr RI: setb Empfang
            end if
        reti

        Timer 2 Interrupt:
            ; Befehle . .
            clr TF2
        reti

        Berechnung:                    ; Vierfach genaue Addition
            r0 = #Summe: r1 = #Summe + 4
            clr c
            ; Zählschleifen werden mit For - Next gebildet.
            for r2 = # 4
                    a = @r0: addc a, @r1: @r0 = a
```

```
        inc r0: inc r1
    next
```

26.2 Das erste Programm

Alle nachfolgenden Programme können durch Kopieren und Einfügen in den Assembler compiliert und nach dem Schreiben in den Controller mit angeschlossenen LED an Port 1 getestet werden.



Die LED zeigt den logischen Pegel des Ausgangs an:
LED ist aus: Pegel ist + (1)
LED leuchtet: Pegel ist — (0)

Das einfachste Programm besteht aus drei Zeilen.

```
#cpu = 89S8253
clr P1.0
End
```

Die erste Zeile deklariert den Controller, die zweite Zeile legt den Pin 0 von Port 1 (P1) an Masse. Wenn Sie an P1.0 eine LED laut vorstehender Abbildung anschließen, sollte sie leuchten. Vergessen Sie die dritte Zeile nicht. END ist eine Endlosschleife auf der Stelle und verhindert, dass eventueller Restcode im Speicher ausgeführt wird.

26.3 Das zweite Programm

Während das erste Programm nach dem Start statisch ist, also nur die LED dauerhaft leuchten lässt, bringt das zweite Programm etwas Leben mit. Die angeschlossene LED an Port 1 Pin 0 soll abwechseln blinken. Der erste Versuch:

```
#cpu = 89S8253 ; @24 MHz
loop
    cpl P1.0
end loop
```

Mit dieser Schleife wechselt der Pin 0 des Port 1 ständig seinen Pegel. Wenn Sie das Programm aber ausprobieren, werden Sie keinen Blinkeffekt sehen. Die LED leuchtet dauerhaft - so könnte man meinen. In Wirklichkeit blinkt sie schnell, so schnell, dass man die Änderung nicht wahrnehmen kann. Dieser Vorgang muss also verlangsamt werden. Dazu probieren wir die folgende Version.

```
#cpu = 89S8253 ; @24 MHz
loop
    for r2: next
    cpl P1.0
end loop
```

Die For-Schleife zählt Register 2 (r2) 256 mal abwärts und benötigt dafür etwas Zeit, sie soll also einen Zeitgewinn vor der nächsten Pegeländerung am Port bringen. Aber auch diese Version ist unbefriedigend, weil der Zeitgewinn einfach zu klein ist. Also ist ein weiterer Versuch mit doppelter Zeitschleife nötig.

```
#cpu = 89S8253 ; @24 MHz
loop
    for r2: for r3: next: next
    cpl P1.0
end loop
```

Nun ist die Verzögerungsschleife 256 x 256 = 65536 Takte lang und man kann bei 24 MHz ein immer noch schnelles Flimmern erkennen. Wir fügen eine dritte Schleife hinzu.

```
#cpu = 89S8253 ; @24 MHz
loop
    for r2: for r3: for r4 = #4: next: next: next
    cpl P1.0
end loop
```

Mit *for r4 = #4: next* haben wir nun endlich eine Blinkfrequenz von ungefähr einem Hertz erreicht. Eine elegante Lösung ist die vorstehende Software allerdings nicht, da wir die Zeit damit nicht exakt bemessen können. Als Basis für die Uhrzeit beispielsweise taugt sie nicht.

Ein Tipp: Rücken Sie unbedingt die Befehlszeilen in der Schleife mit der Tab-Taste ein. Sie verlieren sonst bei komplexeren Programmen vollständig die Übersicht.

Wenn Ihr Controller außer Warteschleifen-Zählen auch noch andere Dinge tun soll, müssen Sie die Zeitsteuerung unbedingt einer Interrupt-Routine übergeben. Die Interruptroutine *Timer 2* des folgenden Programms wird 50 mal in der Sekunde aufgerufen. Das geschieht durch Beschreiben der Timer 2 Relaod-Register. Die niedrigste Frequenz ist 30,51758 Hz, die damit erreichbar ist. Wir wählen aber 50 Hz als Interruptfrequenz, da sich 50 besser teilen lässt. Für das Blinken der LED mit exakt 1 Hz benötigen wir die Frequenz 2 Hz: einmal fürs Einschalten, das andere Mal fürs Ausschalten. Die 2 Hz werden erreicht mit Register 0, das die 50 Hz durch 25 teilt.

```
#cpu = 89S8253  ; @24 MHz
; Timer 2 Interrupt-Frequenz setzen:
RCAP2L = #C0h: RCAP2H = #63h ; f = 50Hz
TL2 = #C0h: TH2 = #63h: setb TR2  ; Timer 2 läuft.
setb ET2: setb EA  ; Interrupt freigeben
r0 = #25
end        ; Hauptprogramm Ende

Timer 2: ; Timer 2 Interrupt, f = 50 Hz
    clr TF2: dec r0 ; Interrupt Bit löschen, 2 Hz Zähler erniedrigen
    if r0 = #0 then
          cpl P1.0: r0 = #25            ; f = 2 Hz
    end if
reti
```

Damit haben wir die Blinkfrequenz von genau einer Sekunde erreicht.

Mittlerweile Sollten Sie ein Programm lesen können. Was ändert sich am Verhalten der LED, wenn wir die Interruptroutine auf nachfolgende Weise

verändern?

```
Timer 2: ; Timer 2 Interrupt, f = 50 Hz
    clr TF2: dec r0
    if r0 = #0 then
        clr P1.0: r0 = #50; LED an 1/50 s
    else
        setb P1.0; LED aus 49/50 s
    end if
reti
```

26.4 Das dritte Programm, Lauflicht 1

Nun wollen wir etwas mehr Bewegung und mehr Schwung in die LED bringen.

Das folgende Programm setzt voraus, dass acht LED an Port 1 angeschlossen sind. Jede LED ist mit dem - Pol am Pin des Controllers verbunden und mit dem +- Pol an 5V. Das Programm erzeugt ein Lauflicht. Experimentieren Sie mit der Konstanten *Tempo*.

```
#cpu = 89S8253 ; @ 24 MHz, SW04.a51
#Const Tempo = F0h
a = #1111 1110b
loop
    P1 = a: rr a
    for r1 = #Tempo: djnz r0, hier: next
end loop
```

26.5 Lauflicht 2

Das nächste Programm arbeitet wie das vorstehende, verfügt aber mit dem Bit *Richtung* über eine Richtungsumkehr.

```
#CPU = 89S8253; 24 MHz
#Bit Richtung
#Const Tempo = 80h
clr Richtung: a = # 1111 1110b: r7 = #0
loop
    P1 = a
```

```
    if bit Richtung then: rr a: else: rl a: end if
    for r1 = #Tempo: djnz r0, hier: next
    inc r7
    if r7 = #7 then: cpl Richtung: r7 = #0: end if
end loop
```

26.6 Lauflicht 3

Das nachfolgende Programm ist ähnlich dem vorstehenden, arbeitet aber mit verschiedenen Geschwindigkeiten.

```
#cpu = 89S8253 ; @ 24 MHz
#Const Tempo = 70h
a = #0111 1111b
loop
    r7 = #0
    loop
        P1 = a
        if bit F0 then: rl a: r1 = #Tempo * 2
        else: rr a: r1 = #Tempo / 2: end if
        acall Warte
        inc r7     ; LED Nummer
    until r7 = #7
    cpl F0         ; für Richtungswechsel
end loop

Warte:
    for r1: djnz r0, hier: next
ret
```

26.7 Lauflicht 4

Eine weitere Variation des Lauflichts will Sie überraschen:

```
#cpu = 89S8253 ; @ 24 MHz
#Byte Tempo
#Const Tempomax = C0h, Tempomin = 10
Tempo = #Tempomax
```

```
a = #1111 1110b
loop
    loop
        acall Warte: P1 = a: rl a: dec Tempo: r5 = Tempo
    until r5 = #Tempomin
    loop
        acall Warte: P1 = a: rr a: inc Tempo: r5 = Tempo
    until r5 = #Tempomax
end loop

Warte:
    for r7 = Tempo: djnz r6, hier: next
ret
```

26.8 Lauflicht mit Interruptsteuerung

Nun wird es Zeit, das Timing des Controllers in den Griff zu bekommen, d. h. die Leuchtzeit jeder LED exakt zu steuern. Zu diesem Zweck verwenden wir den Interrupt des Timer 2. Auf den ersten Blick wirkt das folgende Programm etwas unübersichtlich. Wenn wir aber dessen Struktur verstehen, wird es eleganter und hat es seinen Schrecken verloren.

Das folgende Programm leistet das Gleiche wie Lauflicht 2. Es ist in zwei Teile gegliedert. Der erste Teil ist das Hauptprogramm. In ihm wird das Timing für den Interrupt des Timer 2 vorbereitet und die Variablen initiiert. Sie können zur Ermittlung der Reload-Werte und damit der Interrupt-Frequenz die im Assembler vorhandenen Assistenten benutzen.
Der zweite Teil ist die Interruptroutine, die 50mal pro Sekunde aufgerufen wird. Die Konstante *Tempo* teilt zusammen mit dem Register r6 die Interruptfrequenz durch 2. Damit wissen wir, dass jede LED genau eine fünfundzwanzigstel Sekunde lang leuchtet. Das Register r7 zählt die Anzahl der leuchtenden LED und schaltet dann die Richtung um.

```
#cpu = 89S8253 ; @ 24 MHz
#Bit Richtung
#Const Tempo = 2h
```

```
; Timer 2 Interrupt-Frequenz setzen:
RCAP2L = #C0h: RCAP2H = #63h ; f = 50Hz
TL2 = #C0h: TH2 = #63h: setb TR2  ; Timer 2 läuft.
setb ET2: setb EA  ; Interruptfreigabe
clr Richtung: r6 = #Tempo: r7 = #7: a = #1111 1110b
End        ; Hauptprogramm Ende

Timer 2: ; Interruptroutine, f = 50 Hz
    clr TF2          ; Interruptflag löschen
    djnz r6, Exit Timer 2      ; teilt durch 2
        r6 = #Tempo       ; f = 25 Hz
        P1 = a
        if bit Richtung then: rr a: else: rl a: end if
        djnz r7, Exit Timer 2
            r7 = #7: cpl Richtung   ; f = 25/7 Hz
Exit Timer 2:
reti
```

26.9 Binärlicht

Für alle, die die vorstehende Software unübersichtlich finden, hier nun eine etwas leichtere Kost. Der Befehl *dec P1* verursacht den Effekt. Nach einem Reset weist P1 den Wert FFh auf. Mit *dec P1* wird der Wert zu FEh = 1111 1110b etc und Port 1 gibt das Binärmuster aus. Das Register r7 steuert die Geschwindigkeit.

```
#cpu = 89S8253  ; @ 24 MHz, SW07.a51
; mit Timer 2 Interrupt:
RCAP2L = #C0h: RCAP2H = #63h  ; f = 50Hz
TL2 = #C0h: TH2 = #63h: setb TR2  ; Timer 2 läuft.
setb ET2: setb EA  ; Interruptfreigabe
r7 = #12
End

Timer 2: ;Interruptroutine,  f = 50 Hz
    clr TF2
    djnz r7, Exit Timer 2
```

```
        r7 = #12: dec P1
Exit Timer 2:
reti
```

26.10 Dimmer

Das folgende Programm zeigt, wie man eine oder mehrere LED dimmen kann. Dies geschieht durch Änderung des Tastverhältnisses, d. h. durch Variieren der Aus– und Anzeit. Auch hier wird der Interrupt verwendet, aber nicht zum Ansteuern der LED, sondern zum Ändern des Tastverhältnisses (englisch: Duty Cycle). Durch Verschieben der Kommentierung von der Zeile *für eine LED* auf *für acht LED* können Sie unterschiedliche Variationen der Software testen.

```
#cpu = 89S8252  ; @24 MHz
#Byte 256 Hz Teiler, Tastverhältnis
sjmp Initialisierung
Timer 0: ; Timer 0 Interrupt, f = 1 kHz
    djnz 256 Hz Teiler, Exit Timer 0  ; f = 256 Hz
        256 Hz Teiler = #39: inc Tastverhältnis
        a = Tastverhältnis: if a = #0 then: cpl F0: end if
Exit Timer 0:
reti

Initialisierung:
256 Hz Teiler = #39: Tastverhältnis = #0
orl TMOD, #2: TH0 = #38h: setb TR0 ; in TH0 ist Reloadwert
setb ET0: setb EA  ; Interrupt freigeben
loop
    ; cpl F0: c = F0: P1.0 = c          ; für eine LED
    cpl F0: if bit F0 then: P1 = #F0h: else: P1 = #0Fh: end if
    ; für acht LED
    for r3 = Tastverhältnis: next
    a = Tastverhältnis: cpl a: inc a
        ; berechnet 256 - Tastverhältnis
    ; cpl F0: c = F0: P1.0 = c          ; für eine LED
    cpl F0: if bit F0 then: P1 = #F0h: else: P1 = #0Fh: end if
```

```
    ; für acht LED
    for r3 = a: next
end loop
```

26.11 Zufallsgenerator

Bitte entschuldigen Sie, wenn ich Ihnen mit dem folgenden Programm harte Kost zumute. Die Vorgänge in der Interruptroutine sind nicht leicht zu verstehen. Durch die Zufallswerte im internen RAM nach einem Reset und die wechselnden Inhalte des Timer 1 entstehen dort unterschiedliche Zeiten und unterschiedliche Bitmuster an Port 1, die als Zufallszahlen zwischen 0 und 255 verwendet werden können. Somit dient die nachfolgende Software als endlos Zufallsgenerator mit interessanten optischen Effekten an den angeschlossenen LED.

```
#cpu = 89S8253 ; @ 24 MHz, SW08.a51
#Byte Rotate Reload
RCAP2L = #C0h: RCAP2H = #5
TL2 = #C0h: TH2 = #63h: setb TR2 ; Timer 2 läuft.
TMOD = #1: setb TR0      ; Timer 1 als 16bit Timer
setb ET2: setb EA ; Interruptfreigabe
Rotate Reload = #1
End

Timer 2: ; f = 50 Hz
    r0 = #30h      ; Zeiger auf IRAM
    loop
        dec @r0
        if @r0 = #0 then
          a = TL0: b = #C7h: div ab          ; hier kann man
                ; andere Zahlen ausprobieren.
          @r0 = b
          for b: next        ; Verzögerung
          a = r0: anl a, #07h: r2 = a: a = Rotate Reload
          for r2: rl a: next
          xrl P1, a          ; Ausgabe der Zufallszahl
```

```
            a = Rotate Reload: rl a: Rotate Reload = a
        end if
        inc r0
    until r0 = #38h
    clr TF2
reti
```

26.12 Ein Betriebssystem

Das folgende Programm ist ein Beispiel für ein Betriebssystem, das Frequenzen von 1 kHz, 50 Hz, 4 Hz und 1 Hz erzeugt. 1 kHz für Displayrefresh, Windmessung etc., 50 Hz für die Tastenabfrage, 4 Hz für Blinkfrequenz, 1 Hz für Uhrzeitaktualisierung und Temperaturmessung. Diese Aufgaben erfolgen nicht in der Interruptroutine. Diese setzt nur die Bit, die im Hauptprogramm, das als Endlosschleife (Scheduler) vorliegt, ausgewertet werden. In der Interruptroutine selbst sollten Sie keine Register und Variablen benutzen, die außerhalb des Interrupts verwendet werden, da nicht vorhersehbar ist, wann eine Subroutine durch einen Interrupt unterbrochen wird. Vermeiden Sie Befehle, die das Carry ändern. Wären danach die benötigten Variablen durch den Interrupt verändert, z. B. der Akku a oder das Carry c, würde sich vermutlich die Software aufhängen bzw. durch den Watchdog einen Dauer-Reset auslösen. In der Interruptroutine sollten Sie also nur Variablen verwenden, die außerhalb nicht benötigt werden. Dadurch kann man in der Interruptroutine auf die Sicherung von Akku, PSW, Umschalten der Registerbank und andere Rettungsmaßnahmen verzichten. Das spart viel Platz und trägt wesentlich zur Programmsicherheit bei.

Nach dem Reset ist Platz nur für wenige Befehle, da der Timer 0 Interruptvektor fest an der Adresse 000Bh steht. Daher wird die Interruptroutine mit *sjmp Initialisierung* übersprungen.

Es folgt die Initialisierung, das komplette IRAM wird gelöscht und die Variablen für den Interrupt werden vorbereitet. Der Stack behält seinen Reset-Wert und beginnt bei 08h im IRAM. Mit dieser Methode werden maximal 16 Byte für den Stack benötigt.

Anmerkung: Machen Sie nie die Initialisierung der Variablen in einer Sub-

routine, Wenn Sie darin das IRAM löschen, ist der Stack mit gelöscht und der Rücksprung in das Hauptprogramm nicht mehr möglich.

```
#cpu = 89S8253 ; @ 24 MHz
#Byte 1kHz Teiler, 50 Hz Teiler, 4 Hz Teiler
#Byte Auswertungnummer
#Bit 1 kHz, Tastenabfrage, 4 Hz, ist 1 Sekunde
#Bit schreibe Messdaten, Blink
#Bit LED = P1.0, Backlight = P1.1
#def Displaybeleuchtung aus {clr Backlight}
#def Displaybeleuchtung an {setb Backlight}
Reset:
inc AUXR: inc CLKREG ; ALE abschalten, X2-Modus wählen
sjmp Initialisierung
Timer 0:              ; Interrupt, f = 20 kHz
    djnz 1kHz Teiler, T0 Return
        1kHz Teiler = #20: setb 1 kHz          ; f = 1 kHz
        djnz 50 Hz Teiler, 50 Hz Ende
            50 Hz Teiler = #20: setb Tastenabfrage     ; f = 50 Hz
            setb LED         ; ausschalten
        50 Hz Ende:
        djnz 4 Hz Teiler, T0 Return
            4 Hz Teiler = #250: setb 4 Hz     ; f = 4 Hz
            djnz 1 Hz Teiler, T0 Return
                1 Hz Teiler = #4: setb ist 1 Sekunde ; f = 1 Hz
                clr LED   ; an
T0 Return:
reti

Initialisierung:
orl EECON, #8   ; Internes EEPROM zum Lesen verwenden
clr a: for r0 = a: @r0 = a: next ; Variablenbereich löschen
1 kHz Teiler = #20: 50 Hz Teiler = #20: 4 Hz Teiler = #250: 1 Hz
Teiler = #4 ; Vorbereitung für Timer 0 Interrupt
Displaybeleuchtung an
```

```
WDTCON = #C1h          ; Watchdog an, 256 ms
TMOD = #2       ; Timer 0 im 8-Bit Autoreload-Modus.
; Die Überlauffrequenz des Timer 0 beträgt 20 kHz
TL0 = #38h: TH0 = #38h: setb TR0
setb ET0: setb EA  ; Interrupt
Displayinitialisierung

; ~ ~ ~ ~ ~ ~ ~ Hauptprogramm ~ ~ ~ ~ ~ ~ ~
loop
    if bit 1 kHz then
        clr 1 kHz: lcall Displayrefresh: lcall Wind messen
    end if
    if bit Tastenabfrage then; 50 Hz
        clr Tastenabfrage: acall Tastenabfrage
    end if
    if bit 4 Hz then
        clr 4 Hz: cpl Blink
        orl WDTCON, #2    ; Watchdogtimer rücksetzen
        if bit schreibe Messdaten then
            lcall Messdaten ins FRAM schreiben
        end if
    end if
    if bit ist 1 Sekunde then
        clr ist 1 Sekunde: Uhrzeit aktualisieren
        Temperatur lesen
        Auswertungnummer = #FFh ; startet Gerätesteuerung
    end if
    lacall Gerätesteuerung
end loop
; ~ ~ ~ ~ ~ ~ Hauptprogramm Ende ~ ~ ~ ~ ~ ~ ~ ~
```

Das Hauptprogramm ist eine Endlosschleife. In ihr werden die Frequenz-Bit aus dem Interrupt abgefragt. Somit können alle erforderlichen Aufgaben der Software zu korrekter Zeit ausgeführt werden.

Besondere Bedeutung hat der Sprung zur Gerätesteuerung (Multitasking). Hier

werden Tastenabfragen, Messwerte und Konstanten aus dem EEPROM des Controllers ausgewertet und die Ausgänge des Controllers für die gewünschten Aktionen gesetzt.

26.13 Multitasking

Mit der Technik des vorstehenden Programms sieht es so aus, als könne der Controller zur selben Zeit die verschiedensten Aufgaben erfüllen. Man nennt dieses Verhalten **Multitasking**.

Der folgende Ausschnitt ist ein Beispiel für die Routine *Gerätesteuerung* des vorstehenden Programms. Die Variable *Auswertungnummer* entscheidet, welche Aufgabe ausgeführt wird. Sie wird im Hauptprogramm zu jeder Sekunde mit dem Wert FFh beschrieben und in der Routine *Gerätesteuerung* um Eins erhöht. Am Ende der Routine steht der Befehl *dec Auswertungnummer*, der den Zählerstand einfriert.

Die Routine *Gerätesteuerung* lässt sich auf beliebig viele Aufgaben erweitern.

```
Gerätesteuerung:  ; wird permanent aufgerufen
inc Auswertungnummer: a = Auswertungnummer
if a = #0 then
    dptr = #Befeuchter Modus: acall Befeuchter Steuerung
    c = Motor an: Befeuchter Ausgang = c: ret
end if
if a = #1 then
    dptr = #Heizung Modus: acall Heizung Steuerung
    c = Motor an: Heizung Ausgang = c: ret
end if
if a = #2 then
    dptr = #Ventilator 1 Modus: acall Ventilator Steuerung
    c = Motor an: Ventilator 1 Ausgang = c: ret
end if
if a = #3 then
    dptr = #Ventilator 2 Modus: acall Ventilator Steuerung
    c = Motor an: Ventilator 2 Ausgang = c: ret
end if
```

```
if a = #4 then: ljmp Alarmprüfung: end if
dec Auswertungnummer
ret
```

26.14 Die C-Dur Tonleiter

Das folgende Programm demonstriert die Verwendung von Tabellen und erzeugt die C-Dur Tonleiter. Ein Piezo oder ein Kleinlautsprecher mit eventuellem Treiber (74HC14 oder 40106) wird am Timer 2 Ausgang T2 = P1.0 angeschlossen.

```
#cpu = 89S53      ; @24 MHz
#Byte 100 Hz Teiler, 4 Hz Teiler, Tonhöhe, Tonlänge
ajmp Initialisierung
Timer 0: ; Timer 0 Interrupt
    ajmp Timer 0 Interrupt

Initialisierung:
orl TMOD, #2      ; Timer 0 erzeugt f = 10 kHz
TL0 = #38h: TH0 = #38h: setb TR0 ; Reloadwert für 10 kHz
; Timer 2 ist Taktgenerator. Das Signal wird an T2 ausgegeben. Er ist
konfiguriert als 16-Bit Aufwärtszähler mit automatischer Nachladung
aus den Reload-Registern.
T2MOD = #2      ; Timer 2 steht
setb ET0: setb EA
100 Hz Teiler = #100: 4 Hz Teiler = #25
dptr = #Melodie  ; Zeiger auf Melodie-Tabelle
xrl WCON, #4: dptr = #Frequenzen: xrl WCON, #4
    ; zweiter Datenpointer zeigt auf Frequenztabelle
Tonlänge = #1
end
; * * * Hauptprogramm Ende * * *

Timer 0 Interrupt: ; 10 kHz
    djnz 100 Hz Teiler, Reti
          100 Hz Teiler = #100        ; 100 Hz
       a = 4 Hz Teiler
```

```
        if a = #8 then
          a = Tonlänge: if a = #1 then: clr TR2: end if ; Ton ausschalten
        end if
        djnz 4 Hz Teiler, Reti
          4 Hz Teiler = #25          ; 4 Hz
          djnz Tonlänge, Reti
            setb TR2                 ; Timer 2 einschalten
          Tonhöhe und Tonlänge holen:
            clr a: movc a, @a+dptr; Tonhöhe lesen
            if a = #0 then
              dptr = #Melodie    ; Tabellenstart
              sjmp Tonhöhe und Tonlänge holen
            end if
            dec a: Tonhöhe = a
            inc dptr: clr a: movc a, @a+dptr
            Tonlänge = a ;Tonlänge lesen
            inc dptr
          ; Frequenz holen
            xrl WCON, #4             ; zweiten Datenpointer wählen
            a = Tonhöhe
            rl a ; mal 2, da in der Tabelle Frequenzen zwei
              ; Byte zu lesen sind.
            r7 = a ; Akku retten
            movc a, @a+dptr          ; Frequenz Highbyte lesen
            RCAP2H = a
            a = r7 ; Akku wieder herstellen
            inc a    ; nächstes Byte aus der Tabelle holen
            movc a, @a+dptr          ; Frequenz Lowbyte lesen
            RCAP2L = a
            xrl WCON, #4             ; ersten Datenpointer wählen
Reti:
reti
```

```
Melodie: ;Tonhöhe, Tondauer (1 = 1/4 Sekunde)
    DB 3,1, 4,1, 5,2, 5,2, 6,2, 2,2, 2,2, 2,1, 3,1, 4,2, 4,2, 5,2, 3,4
    DB 3,1, 4,1, 5,2, 5,2, 8,2, 7,2, 6,2, 5,1, 4,1, 3,2, 4,2, 2,2, 1,4
    DB 0
Frequenzen:        ; der C-Dur Tonleiter
    DB A7h,39h, B1h,16h, B8h,FAh, BDh,6Bh, C4h,D0h, CAh
    DB BCh, D0h,A7h, D3h, 9Ch
    ; 264, 297, 330, 352, 396, 440, 495, 528 Hz = c', d', e', f', g', a', h'. c''
```

26.15 Eine Sirene

Das folgende Programm kommt ohne Tabellen aus. Es erzeugt eine kontinuierliche Frequenzänderung. Ein Piezo oder ein Kleinlautsprecher mit eventuellem Treiber (74HC14 oder 40106) wird am Timer 2 Ausgang T2 = P1.0 angeschlossen.

```
#cpu = 89S53   ; @24 MHz
#Byte 50 Hz Teiler
#Bit Richtung
ajmp Initialisierung

Timer 0:  ; Timer 0 Interrupt
    djnz 50 Hz Teiler, Return
        50 Hz Teiler = #200
        if bit Richtung then
            inc RCAP2H
        else
            dec RCAP2H
        end if
        a = RCAP2H
        if a = #F0h then: clr Richtung ; obere Grenze
        elseif a = #C6h then: setb Richtung      ; untere Grenze
        end if
Return:
reti

Initialisierung:
```

```
orl TMOD, #2    ; Timer 0 im 8-Bit Autoreload-Modus.
TL0 = #38h: TH0 = #38h: setb TR0 ; f = 10 kHz
; Timer 2 ist Taktgenerator. Das Signal wird an T2 ausgegeben. Er ist
konfiguriert als 16-Bit Aufwärtszähler mit automatischer Nachladung
aus den Reload-Registern.
RCAP2H = #C6h: T2MOD = #2: setb TR2    ; Timer 2 läuft
setb ET0: setb EA  ; Interrupt freigeben
50 Hz Teiler = #200: setb Richtung
end
```

26.16 Echtzeituhr und Kalender

Das folgende Programm stellt eine Uhr mit Kalender dar. Die Uhrzeit stellt sich automatisch auf Sommer/Winterzeit um. Die Routine *Uhrzeit aktualisieren* wird als Multitasking-Element in jeder Sekunde einmal aufgerufen. Somit ist das nachstehende Programm nur ein Ausschnitt. Alle Byte-Variablen beinhalten BCD-Werte.

```
#Byte Sekunden, Minuten, Stunden
#Byte Wochentag, Tag, Monat, Jahr
#Bit ist Stunde, Winterzeit, Größer, Kleiner, Gleich

Uhrzeit aktualisieren: ; f = 1 Hz
    r0 = #Sekunden: acall Berechne Zeit
    if a = #0 then              ; volle Minute
        r0 = #Minuten: acall Berechne Zeit
        if a = #0 then          ; volle Stunde
            setb ist Stunde: r0 = #Stunden: acall Berechne Zeit
            if a = #0 then: acall Datum aktualisieren       ; voller Tag
            elseif a = #2 then
                sjmp Zeitumstellung     ; um 2 Uhr
            end if
        end if
    end if
ret

Berechne Zeit:
```

```
    ; Übergabe Variablenadresse in r0, Bit ist Stunde
    ; b und r6 = temporäre Register
    a = @r0: add a, #1: da a ; BCD Addition
    if bit ist Stunde then: b = #24h: clr ist Stunde
    else: b = #60h: end if
    if a = b then: clr a: end if          ; Überlauf
    @r0 = a
ret ; Rückgabe a

Zeitumstellung:   ; letzter Sonntag im März/Oktober
    jbc Winterzeit, Zeitumstellung Ende
    a = Wochentag: if a = #6 then    ; nur sonntags
        a = Tag: b = #24h: Vergleiche
        if bit Größer then      ; Tag > 24
            a = Monat
            if a = #3 then: inc Stunden: ret: end if ; März
            if a = #10h then: dec Stunden
                setb Winterzeit: end if              ; Oktober
        end if
    end if
Zeitumstellung Ende:
ret

Datum aktualisieren:
    acall Tag hochzählen
    a = Tag
    if a = #1 then      ; Monat + 1
        acall Monat hochzählen
        if a = #1 then  ; Jahr + 1
            acall Jahr hochzählen
        end if
    end if
ret

Tag hochzählen:
    acall Anzahl der Monatstage ermitteln      ; Returnwert in b
```

```
    a = Tag: add a, #1: da a: Vergleiche
    if bit Größer then: a = #1: end if
    Tag = a
    a = Wochentag: inc a   ; Wochentag aktualisieren
    if a = #7 then: clr a: end if
    Wochentag = a   ; Werte von 0 bis 6
ret

Monat hochzählen:
    a = Monat: add a, #1: da a
    if a = #13h then: a = #1: end if
    Monat = a
ret

Jahr hochzählen:
    a = Jahr: add a, #1: da a: Jahr = a
ret

Anzahl der Monatstage ermitteln:     ; Gibt Anzahl der Tage in b zurück
    b = #30h: a = Monat
    if a = #4 then ret  ; April
    if a = #6 then ret  ; Juni
    if a = #9 then ret  ; September
    if a = #11h then ret ; November
    if a = #2 then      ; Februar
       b = #29h
       a = Jahr: anl a, #0001 0011b  ; Jahr ist BCD-Zahl
       if a = #0 then ret     ; bei Schaltjahr
       if a = #12h then ret
       dec b: ret
    end if
    inc b
ret

Vergleiche:
    if a = b then
```

```
        setb Gleich: clr Größer: clr Kleiner: ret
    end if
    clr Gleich: Kleiner = c  ; falls a kleiner, Kleiner = 1
    cpl c: Größer = c        ; falls a größer, Größer = 1
ret
```

26.17 Serielle Kommunikation

Für eine Serielle Kommunikation müssen zunächst die entsprechenden Register konfiguriert und der serielle Interrupt freigegeben werden. Eingehende Daten haben das Format:

Serieller Modus, [Daten 1], [Daten 2], . . . , [Daten n]

Das erste Byte ist der serielle Modus. Er bestimmt die auszuführende Aktion (= Befehl). Dieser Befehl wird sofort in die Variable *Serieller Modus* geschrieben. Danach können weitere eingehende Daten folgen. Diese werden temporär in das interne RAM des Controllers ab Adresse D0h geschrieben, als Zeiger dient die Variable *Write Buffer*, als Zähler die Variable *Serieller Zähler*. Nach Eingang aller Daten wird *Serieller Modus* ausgewertet und die gewünschte Aktion ausgeführt.

Am Ende des Datenempfangs wird das Bit *Empfang* gesetzt, das dem Hauptprogramm den Beginn der Datenauswertung anzeigt. Das Bit *Sendebusy* wird beim Senden der Daten verwendet und im seriellen Interrupt gelöscht.

Zur Kommunikation des PC mit dem Controller können Sie das kostenlose Programm *Terminal.exe* verwenden.

```
#cpu = 89S8253 ; @24 MHz
#Byte Serieller Modus, Serieller Zähler, Write Buffer
#Byte Time out Zähler
#Bit Empfang, Sendebusy
Reset:
    inc AUXR: inc CLKREG ; ALE-Signal aus, X2-Modus
sjmp Initialisierung

SINT:  ; serieller Interrupt
```

```
    ljmp Serieller Interrupt

Initialisierung:
orl EECON, #8    ; Internes EEPROM verwenden.
clr a: for r0 = a: @r0 = a: next        ; Variablen löschen
; Timer 2 als Baudratengenerator 28800 Baud
T2CON = #34h: RCAP2L = #CCh: RCAP2H = #FFh
SCON = #50h
setb EA: setb ES   ; Interruptfreigabe

Loop            ; Hauptprogramm
    if bit Empfang then: Serielle Daten auswerten: end if
end loop      ; Hauptprogramm Ende
; ~ ~ ~ ~ ~
Serieller Interrupt:      ; = Interruptroutine
if bit TI then: clr TI: clr Sendebusy: reti: end if
if not bit RI then: reti: end if: clr RI
if bit Empfang then: reti: end if  ; zusätzliche Byte ignorieren
push acc: push PSW: Time out Zähler = #20
a = Serieller Modus   ; nur nach Abschluss des
                      ; vorausgegangenen Transfers
if a = #0 then      ; Start Empfang
; alle eingehenden Byte werden sofort ab Adresse D0h gespeichert.
    acall Seriellen Modus setzen
else            ; weitere Byte empfangen und speichern
    push 0: r0 = Write Buffer: @r0 = SBUF: pop 0
    inc Write Buffer: dec Serieller Zähler
    a = Serieller Zähler: if a = #0 then: setb Empfang: end if
    ; alle Daten wurden empfangen
end if
pop PSW: pop acc
reti
; ~ ~ ~ ~ ~
Seriellen Modus setzen:
a = SBUF: Serieller Modus = a: Write Buffer = #D0h
```

```
    ; EEPROM Page senden
if a = #3 then: Serieller Zähler = #1: ret: end if
    ; 7 Byte Uhrzeit und Datum empfangen
if a = #81h then: Serieller Zähler = #7: ret: end if
    ; EEPROM-Byte empfangen
    ; Format "85h, Adr H, Adr L, Wert"; return 'a'
if a = #85h then: Serieller Zähler = #3: ret: end if
Serieller Zähler = #0: setb Empfang ; für Ein-Byte Befehle
ret
```

Am Ende des Datenempfangs wird das Bit *Empfang* gesetzt, das dem Hauptprogramm den Beginn der folgenden Datenauswertung anzeigt.

```
Serielle Daten auswerten: ; Aufruf nach Datenempfang
clr Empfang: Time out Zähler = #0
orl WDTCON, #2 ; WDT Reset
a = Serieller Modus: Serieller Modus = #0

if a = #1 then      ; Versionsmeldung
    r2 = # kmp2 - kmp1
    loop: a = r2: inc r2: movc a, @a+pc
kmp1:
        if a = #0 then ret
        SBUF = a: setb Sendebusy: jb Sendebusy, hier
    end loop
kmp2:           ; Daten mit 0 Abschluss
    DB 'Version 8.18', 0     ; 12 Byte
end if

if a = #2 then      ; Zeit und Messdaten senden
    r2 = #amjs2 - amjs1
    loop: a = r2: inc r2: movc a, @a+pc
amjs1:
        if a = #0 then ret
        r0 = a: SBUF = @r0; Daten sofort senden
        setb Sendebusy: jb Sendebusy, hier
```

```
    end loop
amjs2:    ; Adresstabelle mit 0 Abschluss, 10 Byte
    DB Sekunden, Minuten, Stunden
    DB Tag, Monat, Wochentag, Jahr
    DB Temperatur, Licht, Wind, 0
end if

if a = #3 then      ; EEPROM-Page senden
    r0 = #D0h: DPH = @r0: DPL = #0
    loop
        movx a, @dptr: inc dptr: SBUF = a      ; senden
        setb Sendebusy: jb Sendebusy, hier     ; warten
        a = DPL: if a = #0 then ret
    end loop
end if

if a = #4 then      ; 256 Byte IRAM lesen und sofort senden
    r0 = #0
    loop: SBUF = @r0: inc r0          ; senden
        setb Sendebusy: jb Sendebusy, hier     ; warten
    until r0 = #0
    ret
end if

if a = #81h then  ; 7 Byte Uhrzeit und Datum empfangen
    r0 = #D0h: r1 = #Sekunden
    for r2 = #7: a = @r0: @r1 = a: inc r0: dec r1: next
    ret
end if

if a = #85h then  ; EEPROM-Byte empfangen, Format: "85h,
                  ; Adr H, Adr L, Wert"; return 'a'
    SBUF = #'a'   ; a ist Quittierung
    r0 = #D0h: DPH = @r0: inc r0: DPL = @r0  ; Adresse
    inc r0: a = @r0           ; Wert
```

```
    ljmp EEPROM beschreiben
end if

ret          ; Datenauswertung Ende
; ~ ~ ~ ~ ~
Seriellen Time Out prüfen: ; Aufruf mit 1 kHz
; Soll Endlosschleife bei unterbrochenen, einkommenden Byte
; verhindern
a = Serieller Modus: if a > #0 then
    a = Time out Zähler: if a > #0 then
        dec a: Time out Zähler = a
        if a = #0 then: Serieller Modus = a: end if
    end if
end if
ret
; ~ ~ ~ ~ ~
EEPROM beschreiben:       ; dptr = Adresse, a = Inhalt
    b = a
    orl EECON, #18h: movx @dptr, a: anl EECON, #EFh
    loop: movx a, @dptr: until a = b
ret
```

26.18 4 x 20 Display

Mit der nachfolgenden Software wird ein 4 x 20 Zeichen LCD Modul bedient. Der auf dem Display auszugebende Text befindet sich im IRAM des Controllers an den Adressen 80h bis CFh. Die Software muss also nicht das Display beschreiben, sondern nur den Adressbereich von 80h bis CFh. Das entlastet die Programmierarbeit enorm. Die eigentliche Arbeit - die Kopie der Zeichen in das LCD Modul - leistet die folgende Routine, die aus dem Hauptprogramm mit 1 kHz aufgerufen wird und mit dem *IRAM Pointer* die Daten aus dem IRAM ins Display kopiert. Der Wert des *IRAM Pointer* ist mit 80h zu initialisieren.

Da manchmal elektrische Störungen das Display aus dem Takt bringen können, erfolgt alle zwei Sekunden eine Initialisierung des Display.

Da das Charakter-ROM des Display keine Unterlängen unterstützt, werden die

Zeichen g und y aus kosmetischen Gründen neu definiert.

Die Bit RS (Register Select) und E (Enable) sind Hardware-Verbindungen zum Display. Der Anschluss RW (Read/Write) des Display wird nicht benutzt und ist mit Masse verbunden, da keine Daten aus dem Display gelesen werden müssen. Die 8-Bit Daten des Display sind mit dem Port 0 des Controller verbunden (mit 10k Pullup Netzwerk an 5V).

```
#Bit Daten, ist Initialisierung, Adresse gesetzt
#Bit RS = P1.0, E = P1.1
#Byte IRAM Pointer, Initialisierungszeiger

Displayrefresh:   ; wird mit 1 kHz aufgerufen
if bit ist Initialisierung then
    a = Initialisierungszeiger: inc Initialisierungszeiger
    if a = #6 then: setb Daten
    elseif a = #ddi2 - ddi1 then ; = Bytezahl
        clr ist Initialisierung  ; in Displaydata Ini
    end if
    acall Displaydata Ini
    c = Daten: RS = c: P0 = a: setb E ; Befehle/Daten senden
    ajmp Display Schluss
end if
r2 = IRAM Pointer: clr a
if r2 = #80h then: a = r2   ; Zeilenadressen ermitteln
elseif r2 = #94h then: a = #80h + 40h
elseif r2 = #A8h then: a = #80h + 20
elseif r2 = #BCh then: a = #80h + 40h + 20
elseif r2 = #D0h then
    a = #80h: IRAM Pointer = a
    inc Initialisierungszeiger: r2 = Initialisierungszeiger
    if r2 = #ddi2 - ddi1 + 20 then
        sjmp Displayinitialisierung     ; ca. alle 2 Sekunden
    end if
end if
if a > #0 then    ; neue Zeile
```

```
    jbc Adresse gesetzt, Display1
    clr RS: P0 = a: setb E    ; Befehl, Displayadresse setzen
    setb Adresse gesetzt
else
Display1:
    c = Daten: RS = c
    r0 = IRAM Pointer: a = @r0  ; Zeichen aus IRAM lesen
    if bit c then ; Daten = 0: Befehle; Daten = 1: Daten
        ; Ersetzungstabelle:
        if a = # '0' then: a = # 'O'
        elseif a = # '°' then: a = #DFh
        elseif a = # 'p' then: a = #F0h
        elseif a = # 'q' then: a = #F1h
        elseif a = # 'ä' then: a = #E1h
        elseif a = # 'ö' then: a = #EFh
        elseif a = # 'ü' then: a = #F5h
        elseif a = # 'g' then: clr a
        elseif a = # 'y' then: a = #1
        end if
    end if
    P0 = a: setb E ; Befehle/Daten an Anzeige senden
    inc IRAM Pointer
end if
Display Schluss:
    clr E
ret

Displayinitialisierung:
    setb ist Initialisierung: clr Daten: Initialisierungszeiger = #1
ret

Displaydata Ini:
    movc a, @a+pc: ret
ddi1:
    DB 38h, 2, 6, Ch, 40h ; 2 Zeilen, Home, Increment
```

```
                ; Cursor aus, Charakter Generator Adresse 0
        DB 0, 0, 1111b, 10001b, 10001b, 1111b, 1, 1110b  ; g
        DB 0, 0, 10001b, 10001b, 10001b, 1111b, 1, 1110b ; y
ddi2:
```