Mikrocontroller 8051,

Kurzbeschreibung der Hardware, das Schreiben von Software, Testen und Simulation mittels ASM51 und C51 sowie dScope (tScope)-51 und µProfi 51

Eine Lernhilfe für den FTKL-, den TINF- und den AF-Unterricht im TGM

von Walter Riemer

Fassung vom April 1998

Vorbemerkung: Dieses kleine Skriptum soll weder den Unterricht durch den Lehrer noch Hand- und Lehrbücher ersetzen, jedoch für die Schüler eine praxisorientierte Hilfe sein. Für Fehlerhinweise und Ergänzungsvorschläge ist der Autor jederzeit dankbar. Im Gegensatz zu den früheren Fassungen wurde auf den Simulator/Debugger AVSIM51 und den OnLine-Debugger FSD51 verzichtet.

Inhaltsübersicht

1. Aufbau 2
1.1 Speicherstruktur 3
1.1.1 Interner Codespeicher 5
1.1.2 Externer Codespeicher 5
1.1.3 Interner Datenspeicher 5
1.1.4 Externer Datenspeicher 6
1.2 Überblick über die Hardwarestruktur des 8051 6
1.2.1 Taktschema 6
1.2.2 Übersicht über die wichtigsten Register 6
1.2.3 Das Wichtigste über die Ports 7
1.2.4 Das Wichtigste über Zeitgeber/Zähler 8
1.2.5 Das Wichtigste über die Interruptsteuerung 9
1.2.6 RESET 10

2. Befehlssatz 11
2.0 Adressierungsarten 12
2.1 Datentransferbefehle im internen Datenspeicher 13
2.2 Datentransferbefehle für externen Datenspeicher (MOVX) 14
2.3 Datentransfer aus dem Codespeicher (MOVC) 14
2.4 Logische Befehle 16
2.5 Rotierbefehle 17
2.6 Arithmetische Befehle 17
2.7 Sprungbefehle 18
2.8 Unterprogrammbefehle 20
2.9 Stackbefehle 20

2.10 No Operation (NOP) 21
2.11 Bitoperationen 21

3. Source-Level Debugging Paket dScope51 für den Mikrocontroller 22
3.1 Aufruf und Grundstruktur des DS51 23
3.1.1 Handhabung mittels Batch-Files 25
3.2 Einfache Beispiele für das Einarbeiten in den DS51 26
3.2.1 Codeübersetzung 27
3.2.2 Registerbänke 0 und 1 mit von Null aufsteigenden Werten füllen 29
3.2.3 Neunerkomplement einer mehrstelligen BCD-Zahl bilden 32
3.3 Watch-Fenster im DS51 35
3.4 Kurze Systematik von DS51 36
3.4.1 Pulldown-Menüs 36
3.4.2 Die wichtigsten Kommandos 37
3.4.3 Testen von µProfi-Programmen mit DS51 39

4. Softwareentwicklung mittels Assemblers und C sowie Testen mittels DS51 40
4.1 ASM51 39
4.1.1 Grundlegendes 40
4.1.2 Assemblieren im Hinblick auf den µProfi 42
4.1.3 Programmbeispiele 45
4.1.3.1 Binär-BCD-Umwandlung 45
4.1.3.2 Timeranwendung 45
4.1.3.3 BCD-Addition 47
4.1.3.4 Halbbytes rotieren 48
4.1.4 Kurze Systematik von ASM51, RL51, OH51 49
4.1.4.1 ASM51 49
4.1.4.2 L51 51
4.1.4.3 OH51 und INTEL-HEX-Format 51
4.2 C51 52
4.2.1 Besonderheiten von C51 52
4.2.2 Praktische Handhabung von C51 55
4.2.2.1 Standardfall 52
4.2.2.2 Systematik beim Testen eines C-Programms (Beispiel) 60
4.2.2.3 Vom Standardfall abweichende Verarbeitung 62
4.2.2.4 Verbinden von Assembler- und C-Moduln 63
4.2.2.5 Timer- und andere Interruptroutinen 71

5. Testen auf der Zielhardware mit TS51 73

1. Aufbau

Der Mikrocontroller 8051 ist der Hauptrepräsentant einer Familie von Prozessoren. Hier wird stellvertetend für alle in erster Linie auf 8051/8031 eingegangen.

Kennzeichnend für die ganze Familie ist ein 8-Bit-Datenbus (8-Bit-Architektur) und ein 16-Bit-Adreßbus (64 kB adressierbar).

1.1 Speicherstruktur

Codespeicher (= Programmspeicher) und **Datenspeicher** sind bei Mikrocontrollern meist eindeutig getrennt, damit das Programm **während des Laufs nicht verändert** werden kann (es gibt kein Schreibsignal zum Codespeicher).

Man kann drei **hardwaremäßig (physisch)** wie auch **softwaremäßig (logisch)** getrennte Speicherbereiche (Adreßbereiche) unterscheiden. Jeder ist mit eigenen Befehlen ansprechbar:

(1) Der **Codespeicher** (Codebereich) kann mit bis zu 64 kB ausgebaut sein. Er ist beim 8031 zur Gänze extern (außerhalb des Prozessors); der 8051 enthält jedoch einen 4 kB großen internen Codespeicher (Adressen 0 bis 0FFFh), der ein **maskenprogrammierbares ROM** ist; weiterer Codespeicher kann extern vorhanden sein. Datentransfers erfolgen (nur in Richtung aus dem Codespeicher!) mittels MOVC. Der Prozessor selbst kann aus dem Codespeicher nur lesen, der Codespeicher wird daher oft als ROM bezeichnet.

Adressiert wird mittels eines 16-Bit Registers **Program Counter** (PC), für Zwecke des Datentransfers aus dem Codespeicher auch mit dem **Data Pointer** (DPTR).

(2) Der **interne Datenspeicher** (interner Datenbereich) besteht aus zwei Abschnitten, in denen der Prozessor sowohl lesen als auch schreiben kann; er wird daher oft als internes RAM bezeichnet:

(2.1) Im unteren Bereich der Adressen 0 bis 7Fh liegen direkt (durch Angabe der Adresse im Befehl) oder indirekt (durch Angabe eines die Adresse enthaltenden Registers) adressierbare 128 Bytes. Von diesen werden allerdings die ersten 48 Bytes von bis zu vier Registerbänken (im 8051/31 nur die beiden ersten vorhanden) sowie 16 bit-adressierbaren Bytes belegt; nur die restlichen 80 Bytes sind in allen Fällen verfügbarer Speicher. Jede Registerbank enthält **8 allgemeine Register**, von denen die beiden ersten (R0 und R1) auch als Zeigerregister auf den Adreßbereich des internen RAM sowie die untersten 256 Bytes des externen RAM verwendbar sind. Datentransfers erfolgen mit MOV.

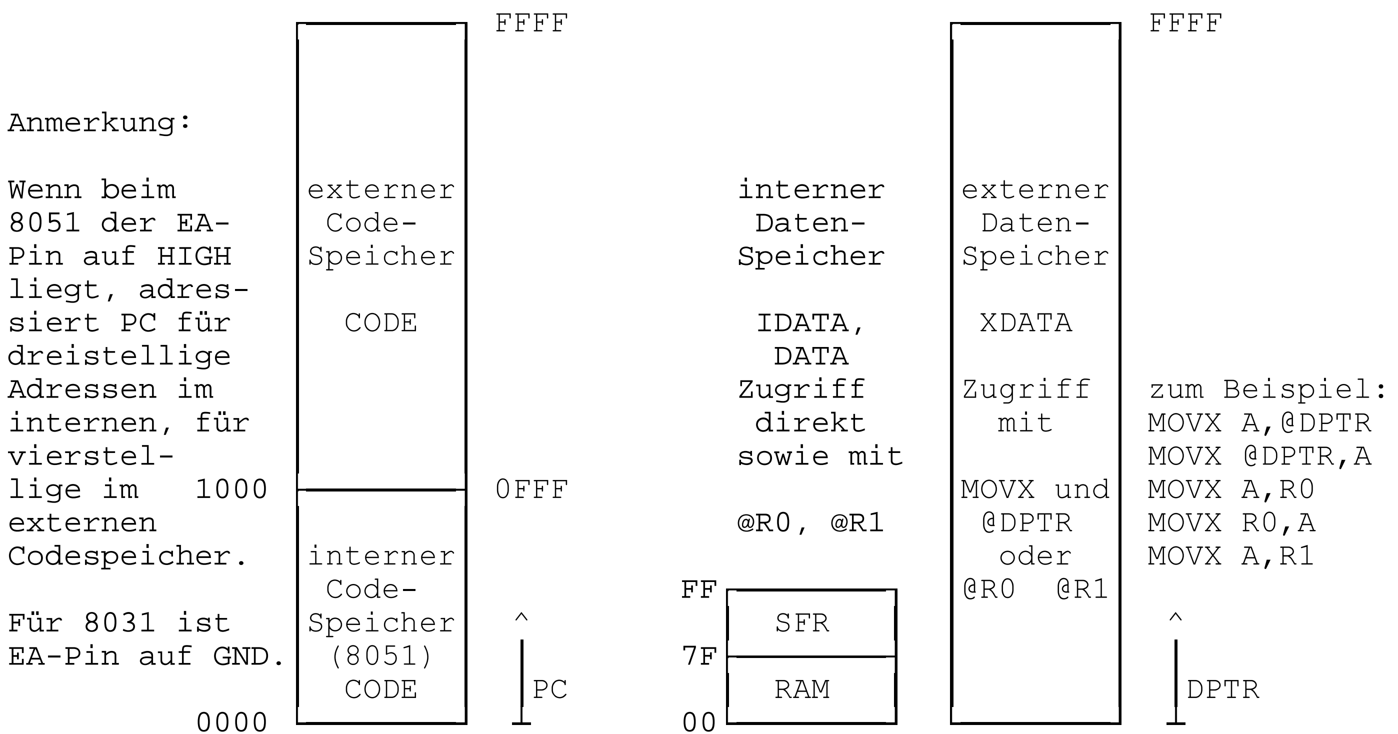
Da der **Stack-Pointer** SP nur auf das interne RAM zeigen kann, muß der Stack dort untergebracht werden. Nach dem RESET zeigt der SP auf die Adresse 7, der Stack würde daher die zweite Registerbank (Registerbank 1) überschreiben. Falls diese benötigt wird, empfiehlt es sich daher, den SP am Beginn eines Programms auf eine höhere Adresse zu setzen. Da SP vor dem Schreiben in den Stack mittels PUSH inkrementiert wird (der Stack wächst nach oben!), sollte man SP auf die um 1 verminderte Adresse des Stackbereichs setzen.

BEISPIEL: MOV SP,#2Fh ; legt Stack über dem bitadressierbaren Bereich an

Die unteren 48 Bytes können mit ihren symbolischen Namen (R0 bis R7 für die jeweils aktuelle Registerbank) adressiert werden, aber auch mit ihrer Adresse, zum Beispiel kann statt R0 auch 0 geschrieben werden. Im bitadressierbaren Bereich wird mit der Bitnummer (0 bis 127 vom niedersten Bit auf Byte 20h aufwärts) adressiert, aber ein Byte kann auch mit seiner Adresse (zum Beispiel 20h) als Ganzes angesprochen werden.

(2.2) Im oberen Bereich der Adressen 80h bis 0FFh liegen 20 **Hardware-Register** (Special Function Registers - SFRs). Die verbleibenden 108 Speicherbytes dieses Bereichs sind nicht nutzbar. Die SFRs sind direkt adressierbar (mittels ihrer Adressen oder mittels ihrer symbolischen Namen).

(3) Der **externe Datenspeicher** (Datenbereich) kann mit bis zu 64 kB ausgebaut sein. Datentransfers erfolgen mittels MOVX, wobei der auf den externen Datenspeicher Bezug habende Operand immer der **Data Pointer** (Datenzeiger) DPTR ist.



Übersicht über die Adressierung im Zusammenhang mit den obenstehenden Speicherplänen:

indirekt: nur über PC	SFR: nur direkt	nur indirekt: @DPTR
indirekt, indiziert: @A+DPTR, @A+PC	RAM: direkt, indirekt @R0, @R1	@R0, @R1 (ganz unten)
unmittelbar (immediate)	Stack im RAM: indirekt: SP	

Aus Software-Sicht werden den Speicherbereichen Segmente zugeordnet, und zwar:

CODE ... für internen oder externen Codespeicher
DATA ... für direkt adressierbaren internen Datenspeicher (128 Bytes, ermöglicht schnellsten Zugriff)
IDATA ... für indirekt adressierbaren internen Datenspeicher (256 Bytes, ermöglicht Zugriff auf vollen internen Datenbereich)
XDATA ... für den ausschließlich indirekt adressierbaren externen Datenspeicher
BIT ... für den bitadressierbaren Speicher

Es folgen nun noch einige nähere Angaben zu den Speicherbereichen.

1.1.1 Interner Codespeicher (ROM) 4 kB (nur beim 8051, nicht beim 8031 vorhanden)

Beim 8051 mit EA-Pin auf HIGH) wird mit Adressen unter 1000h automatisch der interne Codespeicher angesprochen. Der verwandte Prozessor 8031 hat keinen internen Codespeicher, der EA-Pin muß auf LOW sein; der Program Counter(PC) spricht ausschließlich den externen Codespeicher an.

1.1.2 Externer Codespeicher (RAM oder ROM) bis 64 kB

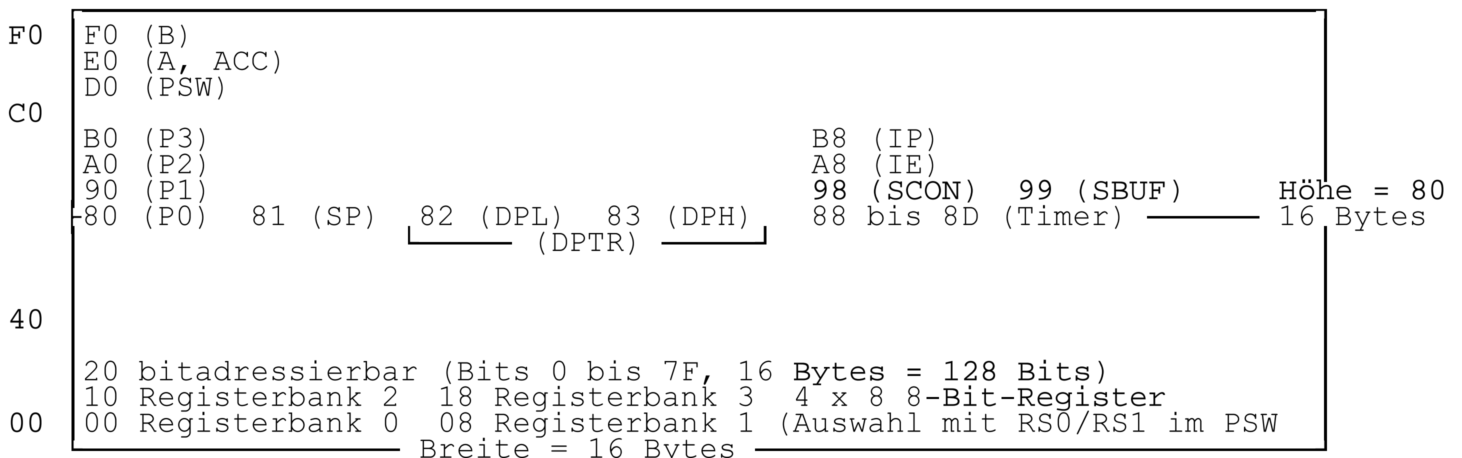
Beim 8051 (mit EA-Pin auf HIGH) wird mit Adressen ab 1000h automatisch der externe Codespeicher angesprochen.

1.1.3 Interner Datenspeicher (RAM)

Die Möglichkeit, die Register auch mit ihren Adressen anzusprechen, etwa R0 der Registerbank 1 auch mit Adresse 08, ist insofern vorteilhaft, als es Transferbefehle zwischen Registern nicht gibt, dieser Mangel aber durch die Schreibweise als Register-Speicher-Transfer umgangen werden kann (z.B. MOV R0,R1 ist nicht erlaubt, MOV R0,1 jedoch schon).

Das nachstehende Bild enthält den Aufbau des internen RAM als 16 x 16 - Matrix mit genauen Adressen aller Register samt mnemomischen Namen.

Adresse (symbolischer Name)



In den oberen 128 Bytes sind 21 Bytes als spezielle Funktionsregister (SFR) adressierbar. Von diesen sind 10 Register (jene, deren Adressen durch 8 ohne Rest teilbar sind) sowohl byte- als auch bitadressierbar. Die Byte-Adressierung erfolgt mit Hilfe der Namen, zum Beispiel PSW, die Bitadressierung entweder mit *name*.bit, zum Beispiel PSW.4, oder direkt mit dem Namen des anzusprechenden Bits selbst, in diesem Beispiel RS1 (siehe nächster Absatz). Die Bitnummern zählen in den Registern von rechts nach links entsprechen also dem Stellenwert.

In den unteren 128 Bytes liegen zunächst 4 (im 8051/31 nur 2) Registerbänke zu je 8 Registern, die durch die Register-Select-Bits RS0 und RS1 im PSW selektiert werden (00 = Bank 0, 01 = Bank 1 usw); zu beachten ist, daß RS1 das höherwertige Bit und RS0 das niederwertige Bit ist, die Reihenfolge im PSW ist also R1 R0. In jeder Registerbank können R0 und R1 außer für ihre normale Registerfunktion auch für indirektes Adressieren in einem Adreßraum von 256 Bytes verwendet werden (Schreibweise @R*i* mit *i*= 0 oder 1).

Darüber liegen 128 bitadressierbare Bits (mit Bitadressen von 0 bis 7Fh, die von unten nach oben zählen). Der ganze Bereich ist direkt oder indirekt (über die Register R0 und R1) adressierbar.

1.1.4 Externer Datenspeicher (RAM oder ROM) bis 64 kB

In ihm ist nur indirekte Adressierung mittels DPTR (in XDATA) oder R0 bzw. R1 (in IDATA) möglich. Die indirekte Adressierung wird mittels vorangestelltem @ ausgedrückt, z.B. @DPTR, @R1.

1.2 Überblick über die Hardwarestruktur des 8051

1.2.1 Taktschema

Jeder Befehlszyklus besteht aus 6 "States" S1 bis S6. In jedem State gibt es 2 Phasen P1 und P2, sodaß ein Befehlszyklus 12 Phasen besitzt, deren jede 1 Mykrosekunde dauert, wenn die Taktfrequenz 12 MHz ist.

Befehle werden in 1 bis 4 Befehlszyklen ausgeführt. Der Zeitgeber zählt mit jedem Befehlszyklus; bei einer Taktfrequenz von 12 MHZ also mit 1 MHz.

1.2.2 Übersicht über die wichtigsten Register

In der folgenden Tabelle sind die Registernamen symbolische Namen, die Adressen entsprechen; nur der Akkumulator hat auch noch den Namen A, mit dem er in den meisten Befehlen angesprochen wird. PUSH- und POP-Befehle beziehen sich jedoch grundsätzlich auf Speicheradressen, daher ist zum Beispiel PUSH A unzulässig: es muß PUSH ACC heißen! Ein weiteres Beispiel ist folgendes:

```
MOV A,#27   ist ein 2-Byte-Befehl, nämlich 74 1B
MOV ACC,#27 ist ein 3-Byte-Befehl, nämlich 75 E0 1B (E0 ist die Adresse des
                                  Akkumulators)
```

Reg	Name	Adr	Zweck
ACC	Akkumulator	E0	Datentransfer und Datenmanipulation
B	Hilfsakkumulator	F0	Multiplizieren und Dividieren, Hilfsspeicher
PSW	Programmstatuswort	D0	Carry (CY), Auxiliary Carry (AC), Overflow (OV) Parity (P), Registerbank-Auswahl (RS0, RS1) User Flag (F0, beliebig verwendbar)
SP	Stack Pointer	81	8 Bit breit, Stack ab 08h
DPTR	Data Pointer	82	16 Bit breit, auch als 8-Bit-Register (DPH, DPL)
P0	Port 0	80	Zwischenspeicher für Port (bidirektional)
P1	Port 1	90	Zwischenspeicher für Port (quasi-bidirektional)
P2	Port 2	A0	-"-, auch als Adreßbus für höhere Adressen
P3	Port 3	B0	-"-, auch mit Sonderfunktionen
TCON	Timer Control	88	Timer-Kontrollregister
TMOD	Timer Mode	89	Timer-Modusregister
T0	Timer 0		16 Bit breit, Zeitgeber/Zähler
T1	Timer 1		16 Bit breit, Zeitgeber/Zähler
PC	Program Counter		16 Bit breit, Programmzähler
IE	Interrupt Enable	A8	Interrupt-Freigabe-Register
IP	Interrupt Priority	B8	Interrupt-Prioritäts-Register

1.2.3 Das Wichtigste über die Ports

Alle vier Ports sind mit einem zugeordneten Latch gepuffert und grundsätzlich bidirektional.

Die Ports können byte- oder bitadressiert werden, zum Beispiel P0 oder P0.3. Besondere IO-Befehle (wie IN bzw. OUT) sind daher nicht erforderlich.

P0 und P2 werden zum Adressieren im externen Code- sowie Datenspeicher verwendet (der **Adreßbus** läuft also über sie). P0 wird für das niederwertige Byte verwendet, P2 für das höherwertige. Falls der externe Speicher nur über R0 oder R1 adressiert wird, ist seine Größe auf 256 Bytes beschränkt; P0 und P2 sind dann verfügbar.

In beiden Fällen wird Bit P3.7 für das RD-Signal, P3.6 für das WR-Signal verwendet. Über Port 3 laufen auch die externen Interrupts (P3.2 und P3.3), die externen Timersignale (P3.4 und P3.5) sowie RXD und TXD der seriellen Schnittstelle (P3.0 und P3.1). Port 3 ist daher nur dann frei einsetzbar, wenn von diesen Funktionen kein Gebrauch gemacht wird.

Port 1 ist frei verwendbar.

1.2.4 Das Wichtigste über die Zeitgeber/Zähler

Die Timer können als 16-Bit- oder als 8-Bit-Zähler verwendet werden (T0 ist auch als TH0, TL0 ansprechbar, T1 auch als TH1, TL1). Ein Anfangswert ist als negative Zahl einzugeben, da die Timer nach oben zählen.

Die wichtigsten, mit den Mode-Bits in TMOD (siehe unten) einstellbaren Betriebsarten sind (Betriebsart 0 ist nur zwecks 8048-Kompatibilität vorhanden und wird hier nicht besprochen):

Betriebsart 1: 16-Bit-Zählung
Betriebsart 2: 8-Bit-Zähler mit Auto-Reload, d.h. TLx arbeitet als Zählregister, THx enthält den nach Zählerüberlauf nachzuladenden (Anfangs-) Wert. Dieser wird automatisch nach TLx übertragen und das Zählen läuft weiter (Anwendung z.B. für das Erzeugen regelmäßiger Timer-Interrupts).
Betriebsart 3: T0 und T1 arbeiten verschieden. Ohne auf nähere Details hier einzugehen, ist nur festzuhalten, daß so 3 Zeitgeber/Zähler realisiert werden können.

Funktion von TCON:

TF0, TF1	Bei Eintritt eines Überlaufs (Übergang von 0FFFFh auf 0) das TFx-Flag in TCON gesetzt wird (x = 0 oder 1).
TR0, TR1	Run Control Bits TRx, welche das Laufen des Timers bewirken, wenn (vom Programm) auf 1 gesetzt, das Stoppen, wenn auf 0 gesetzt.
IE0, IE1	External interrupt edge flag, speichert externe Interrupt-Anforderung am INTx-Eingang, und zwar je nach Einstellung von ITx aufgrund eines Low-Pegels (ITx=0) bzw. einer negativen Flanke (ITx=1).
IT0, IT1	Interrupt Type control bit, legt Typ des Eingangssignals für externen Interrupt fest (Low-Pegel bei 0, negative Flanke bei 1).

Der Bitaufbau ist folgender: TF1 TR1 TF0 TR0 IE1 IT1 IE0 IT0 .

Funktion von TMOD:

Die ersten vier Bits in TMOD sind Timer 1 zugeordnet, die letzten 4 Bits Timer 0. Für jeden Aufbau ist die Bedeutung (von vorne nach hinten):

Gate	Wenn 0: kein Einfluß des Pegels an INTx; wenn 1: Zählen nur, wenn INTx=High.
C/T	Timer/Counter Selector: Wenn 0: Timer zählt internen Takt; wenn 1: Counter zählt externe Ereignisse über INTx.
M1, M0	Mode Bits: Betriebsart (siehe oben).

Ein Programm kann den Timer auf verschiedene Weise auswerten:

(1) Polling: Anfangswert in Timer laden, TRx setzen, dann in einer Schleife ständig TFx überprüfen, bis es 1 wird. Nachteil: Der Prozessor ist beschäftigt und kann nicht während des Ablaufs der eingestellten Zeit anderen Aufgaben nachkommen.

(2) Interrupt-gesteuert: Anfangswert in Timer laden, ETx und EA im IE-Register setzen, TRx setzen.

BEISPIEL einer Unterroutine zum Initialisieren von Timer0:

```
SetTmr:  MOV  TH0,#080h
         MOV  TL0,#0h ; Timer-Anfangswert laden: -32k in 16 Bits
         ANL  TMOD,#0 ; TMOD-Bits löschen
         ORL  TMOD,#00000001b ; T0 zählt internen Takt (16-Bit-Zähler)
         MOV  IE,#10000011b   ; EA, ET0 und EX0 im IE-Register setzen
                              ; (Enable All, Enable Timer 0)
         SETB  TR0     ; Timer 0 einschalten
         RET
```

Auf der dem Timer zugeordneten Interrupt-Einsprungadresse TIMER0 (0Bh) bzw. TIMER1 (1Bh) sollte ein LCALL auf die eigentliche Interrupt-Routine stehen, deren letzter Befehl ein IRET-Befehl ist.

In der Interrupt-Routine müssen alle relevanten Register gesichert werden, die Registerbänke am schnellsten durch Umschalten der Registerbank (im PSW RS0/RS1 verändern mittels SETB oder CLR). Die SFR müssen, soweit erforderlich, auf den Stack gesichert werden, insbesondere PSW, ACC.

In der Timer-Funktion erfolgt Inkrementieren einmal in jedem Befehlszyklus, bei 12 MHz Oszillatorfrequenz also mit 1 MHz.

In der Counter-Funktion inkrementiert eine fallende Flanke an einem externen Eingang den Zähler, wobei das Eingangssignal einmal je Befehlszyklus abgetastet wird. Die Flanke wird erkannt, wenn der Pegel in einem Befehlszyklus HIGH war und im darauffolgenden LOW: die Flankenerkennung erfordert also zwei Befehlszyklen, die höchste Zählrate ist daher nur halb so groß wie im Timerbetrieb, also bei 12 MHz Oszillatorfrequenz 0,5 MHz.

Bei 11 MHz Taktfrequenz (im µProfi) dauert ein Befehlszyklus 12/11 MHz = 1,090909 µs. Für einen Timer-Reloadwert von zum Beispiel 0 (entspricht -65535) ist dann das Timer-Intervall 65535 * 1,090909 µs = 71,492 ms.

1.2.5 Das Wichtigste über die Interruptsteuerung

Interrupts können von 5 Quellen ausgehen, die durch ihnen zugeordnete Bits im **Register IE** (Interrupt-Enable) wirksam oder unwirksam gemacht werden können.

Funktion von IE:

EA	Enable All, alle individuell wirksam gesetzten Interrupts gelten
ES	Enable Serial Port Interrupt
ET1	Enable Timer Interrupt 1
EX1	Enable eXternal Interrupt 1
ET0	Enable Timer Interrupt 0
EX0	Enable eXternal Interrupt 0

Der Bitaufbau ist folgender: EA r r ES ET1 EX1 ET0 EX0 (r... reserviert)

Nähere Angaben zu den Interruptmöglichkeiten stehen im Register TCON (siehe vorstehender Abschnitt 1.2.4).

Jedem Interrupt kann durch Setzen eines zugeordneten Bits im Register IP (Interrupt Priority) die höhere, durch Löschen des Bits die niedrigere von zwei Prioritätsstufen zugewiesen werden (je ein Bit ist jedem Interrupt zugeordnet). Interrupts der Priorität 1 können Interrupt-Behandlungsroutinen der Priorität 0 unterbrechen, nicht aber umgekehrt.

Der Bitaufbau ist folgender: r r r PS PT1 PX1 PT0 PX0 (r ... reserviert), entspricht also dem des IE-Registers weitgehend.

Bei Wirksamwerden eines Interrupts wird eine bestimmte Adresse (Interrupt-Einsprungadresse) im unteren Codespeicher in den PC geladen. An dieser Stelle sollte die **Interrupt-Behandlungsroutine** beginnen. Da eine solche Routine normalerweise länger ist als an Ort und Stelle Platz zur Verfügung steht, ist es üblich, an der Interrupt-Einsprungadresse einen Sprungbefehl (LJMP) auf die eigentliche Interrupt-Behandlungsroutine vorzusehen.

Die Interrupt-Einsprungadressen sind folgender Tabelle zu entnehmen:

Interrupt	Einsprungadresse
Externer Interrupt 0	3h
Timer-Interrupt 0	0Bh
Externer Interrupt 1	13h
Timer-Interrupt 1	1Bh
Interrupt der seriellen Schnittstelle	23h

Bei Auftreten eines Interrupts wird der gerade in Ausführung befindliche Befehl noch beendet, was bis zu 4 Befehlszyklen dauern kann.

1.2.6 Reset

Nach dem RESET stehen die meisten SFR auf 0, ausgenommen SP, der auf 7 steht, und die Ports, die auf 0FFh stehen. Der PC steht jedenfalls auf 0. Der Inhalt des RAM ist unbestimmt.

2. Befehlssatz

2.0 Adressierungsarten

Speicherstellen können auf folgende Arten adressiert werden; die *kursiv* geschriebenen Symbole werden in der Beschreibung der Befehle als Operandentypen verwendet:

(1) **Direkt** (*dadr* - Datenadresse, *cadr* - Codeadresse): Die Adresse steht als 8-Bit-Adresse (*dadr* für den internen Datenspeicher einschließlich SFR) oder als 16-Bit-Adresse (cadr für Sprungziele im externen Codespeicher) im Befehl selbst.

```
BEISPIELE: MOV   A,48h       ; absolute Adressen als Zahlenwert
           LJMP  1234
           MOV   Status,#5   ; symbolische Zieladresse, Direktwert
           MOV   R1,ACC      ; symbolische Quelladresse
```

(2) **Indirekt** (@): Die Adresse steht in einem Register, der Befehl enthält nur das als Zeigerregister verwendete Register selbst.

Dies ist im internen Datenspeicher (nicht aber im SFR-Bereich) mit R0 oder R1 möglich. Wenn der Inhalt von R0 und R1 größer als 7Fh ist, zeigen sie in den parallel zum SFR-Bereich liegenden, nur indirekt adressierbaren Datenspeicher. Im 8051 ist dieser Datenspeicher nicht vorhanden; dorthin geschriebene Daten gehen verloren.

```
BEISPIEL:  MOV   A,@R0
```

Im externen Datenspeicher ist nur indirektes Adressieren mittels DPTR möglich.

```
BEISPIEL:  MOVX  A,@DPTR.
```

(3) **Relativ** (*rel*): Für Sprünge um nicht mehr als 127 Bytes vorwärts oder 128 Bytes rückwärts steht ein "Short Jump" SJMP zur Verfügung, dessen Operand in Assemblersprache absolut geschrieben wird, vom Assembler (und ebenso vom DS51) jedoch in eine positive oder negative Distanz umgerechnet wird.

```
BEISPIEL:  SJMP  SprZiel
```

(4) **Indirekt, indiziert:** Im Codespeicher kann über die Summe der Inhalte von A und DPTR bzw. PC adressiert werden.

```
BEISPIELE: MOVC  A,@A+DPTR
           MOVC  A,@A+PC
           JMP   @A+DPTR
```

(5) Der **Immediate-Modus** (*imm*) kann insofern zu den Adressierungsarten gerechnet werden, als der Operand selbst (nicht seine Adresse) im Befehl steht, die Adresse des Operanden jedoch aufgrund der Befehlsstruktur des Objektcodes beim Ausführen des Befehls vom Prozessor berechnet werden kann. Der Immediate-Wert (**Direktwert**) ist mit vorangestelltem # einzugeben.

```
BEISPIELE: MOV   R7,#23         ; Zahl als Direktwert
           MOV   DPTR,#Tabelle  ; symbolische Adresse als Direktwert
```

(6) **Bitadressierung** (*badr*): Bits im bitadressierbaren internen Speicher werden mit ihrer Bitadresse (Nummer von 0 bis 127) oder einem mittels EQU-Direktive angegeben Namen adressiert.

```
BEISPIELE: SETB  27h
           SETB  Alarm1  ; identisch, sofern  Alarm1  EQU  27h gilt.
```

Bits in bitadressierbaren Registern werden mittels Bitselektors "." als Trenner zwischen Registername (oder Adresse) und der Bitnummer adressiert; sie sind von rechts nach links (entsprechend dem Stellenwert) numeriert.

```
BEISPIELE: SETB  PSW.3   ; setzt Register-Auswahlbit 0
           SETB  RS0     ; identisch, weil RS0 = PSW.3
           SETB  0D0h.3  ; identisch, weil PSW auf Adresse 0D0h liegt.
```

In den folgenden Unterabschnitten werden die zulässigen Kombinationen von Operanden samt Beispielen in Tabellenform angegeben. Wenn die Operanden durch Komma getrennt sind, ist nur die angegebene Reihenfolge erlaubt; wenn die Operanden durch "und" getrennt sind, können Datentransfers in beiden Richtungen geschrieben werden.

Die Symbole für die Operanden haben folgende Bedeutung:

A	Akkumulator
R*r*	R0 bis R7 des aktuellen Registersatzes
@R*i*	R0 oder R1 als Zeiger auf den internen Datenspeicher
dadr	8-Bit-Adresse im internen Datenspeicher
#konst	nur als zweiter Operand: Direktwert

2.1 Datentransferbefehle im internen Datenspeicher

MOV *ziel,quelle*

2.1.1 Datentransfer von Bytes

Operanden	Beispiele		
A und R*r*	MOV A,R3	MOV R4,A	
A und *dadr*	MOV A,36h	MOV HilfsByte,A	MOV P0,A *)
A und @R*i*	MOV A,@R0	MOV @R1,A	
A,*#konst*	MOV A,#27	MOV A,#1Bh	
R*r* und *dadr*	MOV R4,3	MOV 4,R3 **)	MOV Ziffer,R1
R*r*,*#konst*	MOV R0,#27	MOV R0,#Ziffer	
dadr,*dadr*	MOV 8,25	MOV Byte1,Byte2	
dadr und @R*i*	MOV @R0,36h	MOV Steuer,@R1	
dadr,*#konst*	MOV 36h,#1Bh	MOV Steuer,ESC	MOV DPTR,#Tab
@R*i*,*#konst*	MOV @R1,#41h	MOV @R1,#'A'	

*) Ports werden wie Speicherstellen behandelt; es gibt keine IN- oder OUT-Befehle: Memory-Mapped-IO.
) Ein Transfer Register - Register ist nur möglich, wenn für eines der beteiligten Register dessen **absolute Adresse angegeben wird (die beiden Beispiele bewirken daher das Gleiche; nicht erlaubt wäre aber zum Beispiel MOV 4,3). Auf die aktuelle Registerbank ist zu achten, wenn man die Registeradresse angibt.

2.1.2 Datentransfer von Halbbytes (Nibbles)

XCH A,*quelle*

vertauscht die Inhalte des Akkumulators und der internen Speicherstelle.

Operanden	Beispiele		
A,R*r*	XCH A,R1	XCH A,R7	
A,*dadr*	XCH A,1	XCH A,7	XCH A,Byte
A,@R*i*	XCH A,@R0		

SWAP A

vertauscht die beiden Bytehälften des Akkumulators.

XCHD A,@R*i*

vertauscht die niedrigwertigen Halbbytes der beiden Operanden.

2.2 Datentransferbefehle für externen Datenspeicher (MOVX)

MOVX *ziel,quelle*

Der Transfer erfolgt ausschließlich über den Akkumulator. Die Adressierung ist nur indirekt möglich (@DPTR, @R*i* mit i=0 oder 1).

Operanden	Beispiele	
A und @DPTR	MOVX A,@DPTR	MOVX @DPTR,A
A und @R*i*	MOVX A,@R1	MOVX A,@R1

Den DPTR kann man am bequemsten laden mit MOV DPTR,#*konst*.

BEISPIEL: MOV DPTR,#1245h

oder in einem Assemblerprogramm, in dem zum Beispiel eine symbolische Adresse Tabelle in XDATA vereinbart ist:

BEISPIEL: MOV DPTR,#Tabelle

Außerdem können die Halbregister des DPTR (DPH und DPL) für sich manipuliert werden; dies ist insbesondere dann zweckmäßig, wenn Adressen im XDATA dynamisch errechnet werden sollen.

```
BEISPIELE: MOV  DPL,R3
           MOV  DPH,A
```

2.3 Datentransfer aus dem Codespeicher (MOVC)

Dies kommt insbesondere in folgenden Fällen vor:

(1) **Codeumwandlungen** mittels Codetabelle, deren Adresse im DPTR steht. Das umzuwandelnde Byte steht in A; die Umwandlung erfolgt dann mit

MOVC A,@A+DPTR

Das mit DPTR + Inhalt von A adressierte Byte wird nach A geschrieben.

BEISPIEL (mit Testlauf im DS51):

```
             NAME    UmCod
CodeSeg      SEGMENT CODE
             CSEG AT 0A000h
             JMP     Anfang
Tabelle:     DB      '0123456789'
; Ein Wert zwischen 0 und 9 in ACC soll auf die entsprechende ASCII-Ziffer
; umcodiert werden, zum Beispiel aus ACC=05h wird ACC=35h
Anfang:      MOV     DPTR,#Tabelle
             MOVC    A,@A+DPTR
             SJMP    Anfang
             END  ;
```

```
A000H         LJMP      ANFANG(0A00DH)
A003H .UMCOD:TABELLE:
A003H         JNB       26H.1,0A038H  hier
A006H         RLC       A                wird
A007H         ADDC      A,#35H   die Über-
A009H         ADDC      A,@R0    setzungs-
A00AH         ADDC      A,@R1      Tabelle
A00BH         ADDC      A,R0    disassemb-
A00CH         ADDC      A,R1         liert
A00DH .UMCOD:ANFANG:
A00DH         MOV       DPTR,#TABELLE(0A003H)
A010H         MOVC      A,@A+DPTR
A011H         SJMP      ANFANG(0A00DH)
A013H         NOP
```

Beim Testen zuerst PC auf den Programmanfang stellen:

```
>$=0a000
```

Dann den Akku mit der umzucodierenden Zahl laden:

```
>acc=35h
```

Zuletzt Programm mit temporären Haltepunkt auf dem SJMP-Befehl starten:

```
>g,0a011
```

Ergebnis:

```
Register
A = 35
```

Die Übersetzungstabelle hat ständig den gewünschten Inhalt (ASCII-Ziffern):

```
>d c:0a003,0a00c
C:A003          30 31 32 33 34-35 36 37 38 39              0123456789
```

(2) **Auslesen vordefinierter Konstanten** (Meldungen, Tabellen und dergleichen) zwecks direkter Weiterverarbeitung oder Übertragung in einen Datenbereich. Eine Codeumwandlung findet nicht statt, sofern vor Ausführung A den Inhalt 0 hat:

```
MOVC A,@A+DPTR
```

BEISPIEL: Hinter dem ausführbaren Code steht im Codespeicher ein Datenfeld mit den fünf ersten Buchstaben des Alphabets. Diese Zeichen sind in den internen Datenspeicher zu übertragen, damit sie dort im Sinne einer Laufschrift rotieren können (ABCDE -> BCDEA -> CDEAB ...). Der Programmteil, der den Datentransfer aus dem Codespeicher ausführt, ist mit einer senkrechten Doppellinie hervorgehoben

```
          NAME    LaufSchrift
DatenSeg  SEGMENT DATA
          DSEG
          ORG     20h
Schrift:  DS      1              ; 1 Byte Zwischenspeicher
          DS      5              ; 5 Bytes Laufschrift
Lauf      SEGMENT CODE
          RSEG    Lauf
          MOV     DPTR,#Daten   ; Quellfeld adressieren            ||
          CLR     A             ; zum Vermeiden von Codeumwandlung ||
          MOV     R0,#Schrift   ; Adresse des Schriftfelds laden   ||
          MOV     R1,#5         ; 5 Buchstaben der Schrift         ||
```

```
Speicher: PUSH    ACC
          MOVC    A,@A+DPTR    ; Zeichen aus dem Code-Memory holen
          MOV     @R0,A        ; ins Schriftfeld abspeichern
          INC     R0           ; nächstes Byte im Schriftfeld
          POP     ACC
          INC     A            ; nächstes Byte im Code-Memory
          DJNZ    R1,Speicher  ; Schleife ausführen
Anfang:   MOV     R0,#Schrift-1  ; Adresse Zwischenspeicher laden
          MOV     R1,#5        ; 5 Buchstaben der Schrift
Schleife: INC     R0           ; nächstes Zeichen im Schriftfeld
          MOV     A,@R0        ; in Accu laden
          DEC     R0           ; davorstehendes Zeichen
          MOV     @R0,A        ; dorthin abspeichern
          INC     R0           ; nächstes Zeichen im Schriftfeld
          DJNZ    R1,Schleife  ; Schleife ausführen
          MOV     A,Schrift-1  ; Zeichen vom Zwischenspeicher holen
          MOV     Schrift+4,A  ; an letzte Stelle abspeichern
          AJMP    Anfang       ; zum Anfang
Daten:    DB      'ABCDE'
          END ;
```

Grundsätzlich kann man auch mittels PC zwecks Datentransfers aus dem Codespeicher adressieren:

```
MOVC A,@A+PC
```

Dies ist jedoch mit einigen Komplikationen verbunden, erspart allerdings das etwas zeitraubende Laden des DPTR mit der Basisadresse der Tabelle; anstatt dessen muß A mit der Adreß<u>differenz</u> zwischen dem Folgebefehl und dem zu adressierenden Byte geladen werden; der PC selbst kann nicht explizit verändert werden.

2.4 Logische Befehle (ANL, ORL, XRL, CPL, CLR)

Befehl	Bedeutung
ANL *ziel,quelle*	logisches UND
ORL *ziel,quelle*	logisches ODER
XRL *ziel,quelle*	exklusives ODER

Operanden	Beispiele		
A,R*r*	ANL A,R4	XRL A,R0	XRL A,ACC *)
A,*dadr*	ORL A,38h	ORL A,Muster	ORL A,P0
A,@R*i*	ANL A,@R1		
A,*#konst*	ORL A,#82h	ANL A,#'B'	
dadr,A	ANL 40h,A	ORL Muster,A	
dadr,*#konst*	ORL Status,#80h		

*) löscht den Akkumulator; XRL A,0E0h wäre identisch weil ACC = 0E0h

CPL A	bewirkt Komplementieren des Akkumulators (0 wird 1, 1 wird 0)
CLR A	bewirkt Nullsetzen des Akkumulators

2.5 Rotierbefehle (RL, RR, RLC, RRC)

Rotierbefehle verschieben den Inhalt des Akkumulators um eine Stelle nach rechts oder links ohne oder mit Einbeziehung des Carry-Bits im PSW. Die Befehle heißen:

Beispiele in Form von Bitmustern:

Befehl	Carry	Akkumulator	wird	Carry	Akkumulator
RL A	1	0 1 0 0 1 1 0 0		1	1 0 0 1 1 0 0 0
RR A	1	0 1 0 0 1 1 0 0		1	0 0 1 0 0 1 1 0
RLC A	1	0 1 0 0 1 1 0 0		0	1 0 0 1 1 0 0 1
RRC A	1	0 1 0 0 1 1 0 0		0	1 0 1 0 0 1 1 0

2.6 Arithmetische Befehle

Bei Operationen mit zwei Operanden ist der Zieloperand immer der Akkumulator. Alle Zahlen gelten als vorzeichenlose Ganzzahlen.

Operanden	Beispiele	
A,R*r*	A,R7	
A,*dadr*	A,Nummer	A,1234h
A,@R*i*	A,@R1	
A,*#konst*	A,#Wert	A,#27

Die Operationen beeinflussen die Flags CY (Übertrag), AC (Hilfsübertrag), OV (Überlauf) und P (Parität).

2.6.1 Addition ohne und mit Carry (ADD, ADDC)

ADD A,*quelle*	addiert *quelle* zu A, setzt CY auf 0 oder 1, je nach Überlauf
ADDC A,*quelle*	addiert (*quelle* + CY) zu A, setzt CY wie ADD-Befehl

BEISPIELE:

	Carry	Akku	Speicher	wird	Carry	Akku	Speicher
ADD	0	3Eh	D5h		1	13h	D5h
ADD	1	3Eh	D5h		1	13h	D5h
ADDC	0	3Eh	D5h		1	13h	D5h
ADDC	1	3Eh	D5h		1	14h	D5h

2.6.2 Subtraktion, nur mit Borrow (SUBB)

SUBB A,*quelle*	subtrahiert (*quelle* + CY) von A, setzt CY je nach Überlauf

BEISPIELE:

	Carry	Akku	Speicher	wird	Carry	Akku	Speicher
SUBB	0	E3h	D5h		0	0Eh	D5h
SUBB	1	E3h	D5h		0	0Dh	D5h
SUBB	0	3Eh	D5h		1	69h	D5h
SUBB	1	3Eh	D5h		1	68h	D5h

Falls man ohne Borrow subtrahieren möchte, empfiehlt es sich, sicherheitshalber vorher ein allenfalls gesetztes Carry-Bit mit CLR C zu löschen.

2.6.3 Inkrementieren und Dekrementieren (INC und DEC)

INC *ziel*	erhöht *ziel* um 1
DEC *ziel*	vermindert *ziel* um 1

Zulässige Operanden sind A, R*r*, *dadr*, @R*i* und DPTR (letzterer nur bei INC).

2.6.4 Multiplikation und Division (MUL und DIV)

Verknüpfung zweier vorzeichenloser 8-Bit-Zahlen (Verknüpfen vorzeichenbehafteter Zahlen nur softwaremäßig). Beide Befehle löschen CY.

MUL AB	stellt das Produkt aus A und Hilfsakkumulator B in das Registerpaar selbst: höherwertiges Byte in B, niederwertiges in A. OV wird gesetzt, wenn das Ergebnis größer als 0FFh wird.
DIV AB	stellt den Quotienten aus A und Hilfsakkumulator B nach A, den Rest nach B. OV wird bei versuchter Division durch 0 auf 1 gesetzt, andernfalls gelöscht.

2.6.5 Dezimalanpassung (DA)

DA A	bringt nach Binäraddition zweier BCD-Zahlen das Ergebnis wieder ins BCD-Format.

BEISPIEL: (A) = 97h (bedeutet 97d)
(R1) = 19h (bedeutet 19d)

nach ADD A,R1:
(A) = B0h , CY=0 (kein Übertrag aus dem High-Nibble)
AC=1 (Übertrag aus Low-Nibble ins High-Nibble)
Da wenigstens eines der beiden Flags gesetzt ist, wird durch DA A

06h zu A addiert:

(A) = B6h

Da das High-Nibble größer als 9 ist, wird 60h zu A addiert:

(A) = 16h, CY=1 (Übertrag aus dem High-Nibble)

CY enthält nun die Hunderterstelle, A enthält Zehner- und Einerstelle

in BCD-Format: 116h (bedeutet 116d).

2.7 Sprungbefehle

Sprungbefehle setzen die Sprungadresse *cadr* oder *rel* in den PC ein.

Im Assembler-Quellcode bzw. im DS51 wird für *rel* die absolute Sprungadresse angegeben; diese wird in die entsprechende relative Adresse umgerechnet und als solche in den Objekt-Befehlscode eingesetzt.

BEISPIEL: Auf Adresse 0A000h steht SJMP 0A009h. Der entstandene Objektcode lautet dann 80 07, also SJMP +7, weil das Sprungziel um 7 höher liegt als der Inhalt des PC zum Zeitpunkt der Ausführung des SJMP-Befehls (nämlich Adresse des Folgebefehls, also 0A002h).

2.7.1 Unbedingte Sprungbefehle (LJMP, SJMP, AJMP)

Befehl	Beschreibung
LJMP *adr64k*	bewirkt Sprung zu einer 16-Bitadresse im Codespeicher.
SJMP *rel*	bewirkt Sprung zu einem um +127 oder -128 entfernten Ziel.
AJMP *adr2k*	bewirkt Sprung zu einer Adresse innerhalb des aktuellen 2-kB-Blocks, indem nur die niederwertigsten 11 Bits des PC überschrieben werden. Man muß darauf achten, daß das Sprungziel nicht im benachbarten Block liegt, da sonst eine falsche Adresse im PC entsteht.

JMP ist ein generischer Operationscode: je nach Operand wird ein LJMP, SJMP oder ein AJMP generiert, jedenfalls so, daß das Sprungziel erreichbar ist.

2.7.2 Computed GO-TO (JMP)

Befehl	Beschreibung
JMP @A+DPTR	bewirkt Sprung zu der Adresse aus der Summe vom DPTR und A. DPTR enthält in der Regel die Basisadresse einer Sprungtabelle, in der Sprungbefehle stehen.

2.7.3 Bedingte Sprungbefehle (JZ, JNZ) und Schleifenbefehle (DJNZ, CJNE)

Alle bedingten Sprungbefehle erlauben nur relative Adressierung.

Befehl	Beschreibung
JZ *rel*	bewirkt Sprung, wenn (A)=0
JNZ *rel*	bewirkt Sprung, wenn (A)<>0.

Die beiden folgenden Befehle werden vor allem für Schleifen vom Typ for-Schleife eingesetzt:

DJNZ R*r*,*rel* DJNZ *dadr*,*rel*

bewirken Dekrementieren des Operanden, anschließend Sprung, falls der Inhalt des ersten Operanden <> 0 ist.

CJNE *op1*,*op2*,*rel*

bewirkt Vergleich der beiden Operanden und anschließend Sprung, falls die Operanden ungleich sind. Als Operanden sind folgende erlaubt:

op1: A, R*r*, @R*i* *op2*: *dadr*, *#konst*

CY wird beim Vergleich gesetzt und kann benützt werden, um festzustellen, welcher Operand im Falle "Ungleich" größer war.

2.8 Unterprogrammbefehle (LCALL, ACALL, RET, RETI)

LCALL *adr64k* ACALL *adr2k*

bewirken Abspeichern der 2-Byte-Rückkehradresse auf dem Stack und Sprung zur Zieladresse.
Erklärung der Adressen siehe 2.7.1.

CALL ist ein generischer Operationscode: je nach Operand wird ein LCALL oder ein ACALL generiert, jedenfalls so, daß das Sprungziel erreichbar ist.

RET

bewirkt Holen der Rückkehradresse vom Stack und Fortsetzen an dieser Adresse.

RETI

bewirkt das Gleiche aus einer Interrupt-Service-Routine, setzt aber zusätzlich noch IIP0 und IIP1 ("Interrupt In Progress") auf Null.

2.9 Stackbefehle (PUSH, POP)

PUSH und POP führen den Transfer von Bytes im internen direkt adressierbaren Datenspeicher (*dadr*) zum bzw. vom Stack durch. Der Stack Pointer SP wird inkrementiert; das adressierte Byte wird anschließend auf den Stack geschrieben.

PUSH *dadr* POP *dadr*

Assembler und DS51 erlauben auch **Mnemonics** wie Registernamen (der ASM51 jedoch nicht Namen der allgemeinen Register R0, R1 ... !), PSW, ACC, DPH, DPL.

```
BEISPIELE: PUSH  ACC
           PUSH  0E0h   ; ist identisch mit PUSH ACC , da ACC = 0E0h
           PUSH  PSW
```

Der Stack muß sich im internen Datenspeicher befinden und wächst nach oben. Am Anfang eines Programms sollte man SP auf die um 1 verminderte Anfangsadresse des Stacks setzen: MOV SP,#*stack*-1 . -1 deswegen, weil der Stack-Pointer vor dem Schreiben auf den Stack inkrementiert wird.

2.10 No Operation (NOP)

NOP	bewirkt nichts (kostet Zeit, belegt 1 Byte Speicherplatz)

2.11 Bitoperationen

CY wird als 1-Bit-Akkumulator benützt, das heißt, er nimmt das Ergebnis der Operation auf, und wird in den Befehlen mit C angesprochen. Bitadressen (*badr*) können nur direkt angegeben werden (als Zahlen oder als Symbole dafür); im Fall von Bits in SFRs erlaubt der Assembler jedoch auch die Angabe der Bitnummer, vom mnemonischen Namen durch einen Punkt getrennt, zum Beispiel ACC.3 (Bit 3 in ACC). Die Bitnummern entsprechen in diesem Fall dem Stellenwert, zählen also von rechts nach links im Byte.

2.11.1 Bit-Transfer (MOV)

MOV C,*badr* bzw. MOV *badr*,C

```
BEISPIELE: MOV  C,26h
           MOV  C,ACC.3
           MOV  C,0E0h.3   ; identisch, weil ACC = 0E0h
           MOV  20h.0,C
           MOV  0,C        ; identisch, weil die Bitadresse 0 das
                           ; höchstwertige Bit auf Adresse 20h ist
```

2.11.2 Logische Bitoperationen (ANL, ORL)

badr bedeutet Bitadresse, /*badr* bedeutet, daß der Bitwert des Operanden bei der Verknüpfung (nicht jedoch tatsächlich) invertiert wird. Das Ergebnis steht immer in CY.

Befehl	Funktion	Beispiel
ANL C,badr	logisches UND	zum Beispiel ANL C,ACC.7
ORL C,badr	logisches ODER	zum Beispiel ORL C,/OV

(exklusives Oder ist nicht vorgesehen).

2.11.3 Setzen und Löschen von Bits (SETB, CLR)

Befehl	Funktion
SETB C SETB *badr*	setzen ein Bit (auf 1)
CLR C CLR *badr*	löschen ein Bit(auf 0)

2.11.4 Bedingte Sprungbefehle (JC, JNC, JB, JNB, JBC)

Befehl	Beschreibung
JC *rel* JNC *rel*	Sprung in Abhängigkeit vom Inhalt von CY wenn 1 wenn 0
JB *badr,rel* JNB *badr,rel*	Sprung in Abhängigkeit von *badr* wenn 1 wenn 0
JBC *badr,rel*	Sprung in Abhängigkeit von *badr* zu der um *rel* veränderten Adresse in PC. Das abgefragte Bit wird anschließend gelöscht.

3. Source-Level Debugging-Paket dScope51 für den Mikrocontroller

Vorbemerkung: Zur leichteren Handhabbarkeit von dScope51 geschriebene Batch- und Kommandodateien basieren auf der Konfiguration, daß die dScope51-Software auf C:\8051\DSCOPE steht und die zu entwickelnden Programme (Assembler- und C-Programme) in C:\8051\ASM51\PRG bzw. C:\8051\C51\PRG stehen. Für anders gelagerte Konfigurationen müssen die Batch- und Kommandodateien entsprechend abgewandelt werden.

dScope besteht aus mehreren Komponenten:

DS51 ist ein schneller **Hardware-Simulator**, der alle 8051-Derivate simulieren kann, für die .IOF-Dateien verfügbar sind; die gewünschte muß am Beginn der Arbeit mit DS51 geladen werden.

TS51 ist eine Variante von DS51, welche zusammen mit einem passenden Monitorprogramm auf der zu testenden Hardware ein **Remote-Debugging** auf dieser ermöglicht. Remote-Debugging bedeutet, daß ein PC über die serielle Schnittstelle mit der **Zielhardware** (**Target Hardware**, deshalb TS51) verbunden ist; das Debugging-Programm kontrolliert vollständig den Ablauf auf der Zielhardware, auch hinsichtlich Einzelschritt, Haltepunkten, Einblick in Register- und Speicherinhalte und Ändern dieser Inhalte usw.

DS51 und TS51 bieten eine praktisch identische Benutzeroberfläche eines Full-Screen-Debuggers (im Gegensatz zu einem zeilenorientierten Debugger wie etwa DEBUG von MS-DOS), dem allerdings die Eigenschaft fehlt, ständig Speicherfenster darstellen zu können und sich ändernde Werte während eines Programmlaufs beobachten zu können (die Beobachtung ist nur bei Programmunterbrechung möglich). Sie sind **Source-Level-Debugger**, das heißt, das Programm wird auch in der Quellsprache (Assembler 51 oder C 51) angezeigt.

Damit nach dem Laden von DS51 bzw. TS51 gleich automatisch die gewünschten Grundeinstellungen vorgenommen werden, sollten entsprechende .INI-Dateien im aktuellen Verzeichnis vorhanden sein.

3.1 Aufruf und Grundstruktur des DS51

Der Simulator DS51 hat folgende Grundeigenschaften:

- Er ermöglicht **High-Level-Debugging**, sofern das Programm im INTEL-OMF-51-Format (das heißt, als .ABS-Datei) geladen wurde (INTEL-HEX-Format enthält keine Symbolinformationen, INTEL-M51-Dateien werden nicht gelesen).

- Er stellt C-Quellprogrammdateien dar, kann jedoch auch den Assembler-Quellcode oder beides darstellen (gemischter Modus).

- Jederzeitiges Umschalten zwischen 25 / 40 / 50 Zeilen ist möglich.

- Simulation der integrierten Peripherie (Timer, serielle Schnittstelle usw.) ist je nach 8051-Derivat durch Laden entsprechender Treiber (.IOF) möglich.

- Haltepunkte auf vorgegebenen Adressen oder aufgrund vereinbarter Bedingungen, mit Durchlaufzähler (Pass-Count) versehbar.

- Einzelschrittausführung (auf Assembler- oder C-Ebene) oder Programmlauf GO; Unterbrechen mittels CTRL-C. Unterprogramme können im Einzelschrittverfahren nach Wunsch betreten (T-Befehl) oder schnell ausgeführt werden (P-Befehl).

  Mittels Menüs Go - Animate kann ein selbständiger Ablauf mit einstellbarer Geschwindigkeit erreicht werden.

- Watch-Ausdrücke zum Anzeigen von Variableninhalten sind permanent in einem gesonderten Fenster (ganz oben) möglich und wieder entfernbar.

- Eigene Sprache (C-orientiert) zum Schreiben von Funktionen (Kommandos an DS51) oder Signalen (zum Anlegen an Eingabeleitungen).

DS51 ist menügeführt (Pulldown-Menüs, mit ALT-*kennbuchstabe* zu öffnen), kann jedoch ebenso durch die mittels der Menüs erzeugten Kommandos bedient werden, was für Geübte schneller möglich ist.

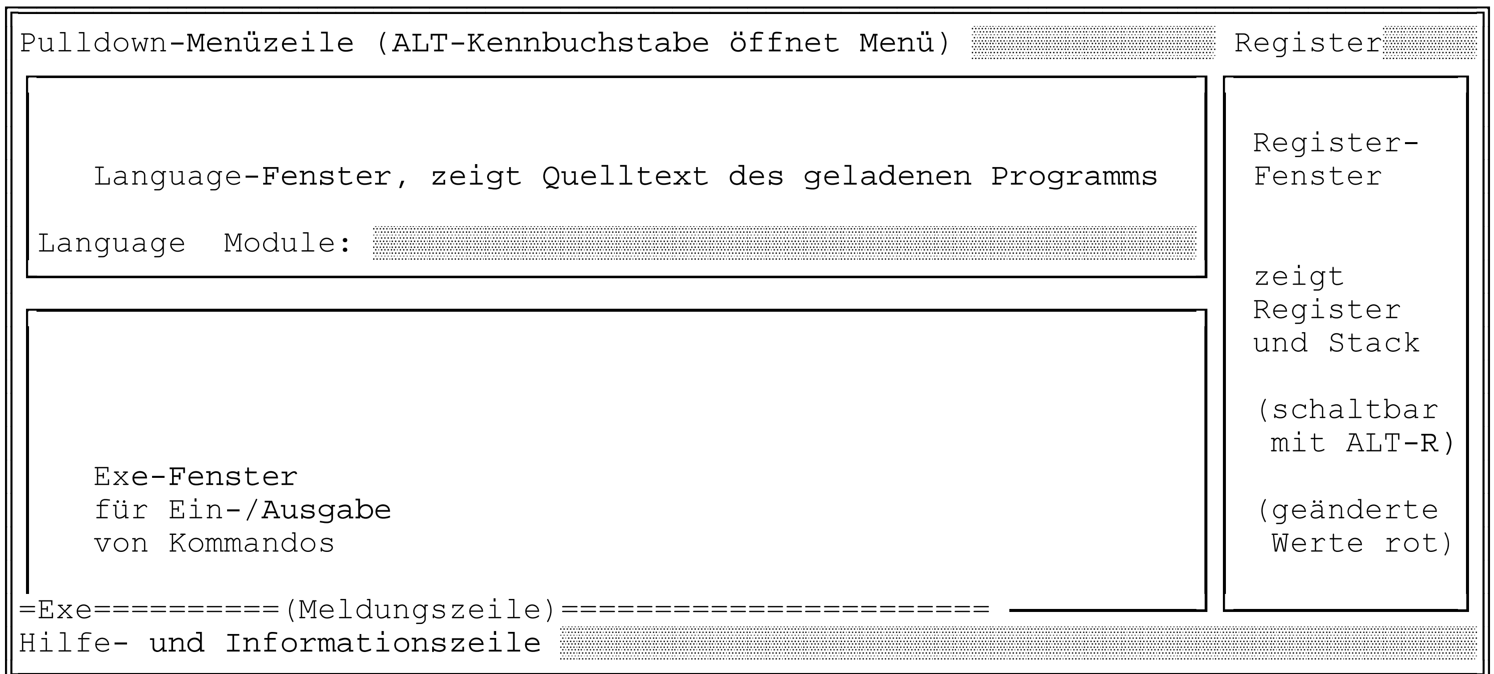
In diesem Abschnitt wird nur auf die wichtigsten Eigenschaften von DS51 eingegangen. Weiterführende Information erhält man aus dem On-Line-Help (ALT-H) sowie aus dem Handbuch.

Nach dem Laden zeigt DS51 seinen Hauptbildschirm. Er kann anschließend durch Eingeben von Kommandos verändert werden, jedoch auch von vornherein durch Kommandos, die in der Datei DS51.INI stehen, welche von DS51 gleich am Anfang aus dem aktuellen Verzeichnis geladen wird.

```
 Options  Key  View  Peripheral  Map  Help  Cpu                  Trace  Register
0000H        NOP                                                          A = 00

0001H        NOP                                                          B = 00
0002H        NOP                                                         R0 = 00
0003H        NOP                                                         R1 = 00
0004H        NOP                                                         R2 = 00
0005H        NOP                                                         R3 = 00
0006H        NOP                                                         R4 = 00
0007H        NOP                                                         R5 = 00
0008H        NOP                                                         R6 = 00
0009H        NOP                                                         R7 = 00
 Language  Module:                                                     DPTR= 0000
dScope-51+ V5.10                                                       PC $= 0000
Copyright KEIL ELEKTRONIK GmbH 1990, 1991
 Load .OBJ                                                                PSW
>/* Initialisation File for dScope-51+ */                               ---R0---
>\u\u\u\                    /* Increase Exe-Window    */               Cycles cyc
>\v\s                       /* View Serial (Off)      */                        0
>load C:\8051\C51\BIN\8051.IOF /* IOF driver for 8051    */
8051/8031 80C51/80C31 PERIPHERALS for dScope-51+  V1.1                 SP=07: 00
(C) Franklin Software Inc./KEIL ELEKTRONIK GmbH 1991                    I:06: 00
>map 0xA800,0xB000          /* XDATA memory 2 kB      */                I:05: 00
>                                                                       I:04: 00
=Exe==                                                                  I:03: 00
 !dos   ASM   ASSIGN   BreakDisable   BreakEnable   BreakKill            --more--
```

Der Bildschirm ist wie folgt eingeteilt:



Das aktuelle Fenster kann mit ALT-*kennbuchstab*e (ALT-E, ALT-L, ALT-S) ausgewählt werden. Bei Bedarf kann das aktuelle Fenster mit ALT-D ("Down") bzw. ALT-U ("Up") verkleinert bzw. vergrößert werden.

Das Registerfenster dient nur zur Anzeige; es kann mit ALT-R zu- und weggeschaltet werden, kann jedoch nicht zum aktuellen Fenster werden. Es dient daher auch nicht zum Setzen von Registerinhalten; dies ist jedoch auf andere Weise möglich. Der DS51 ist daher auch zum Testen von Assemblerprogrammen geeignet, insbesondere solchen, die man "zum Ausprobieren" schnell mit dem **inline-Assembler** schreibt.

Das Serial-Fenster wird nicht immer gebraucht: es kann mit ALT-V S aus- oder eingeblendet werden.

DS51 ladet selbsttätig die Kommandodatei DS51.INI, in der Kommandos zum Herstellen einer jedes Mal benötigten Grundeinstellung stehen können. Eine geeignete Datei DS51.INI für µProfi-Anwendungen (die auch im obenstehenden Ausdruck bereits geladen wurde) ist folgende:

```
/* Initialisation File for dScope-51+ */
\u\u\u\                          /* Increase Exe-Window    */
\v\s                             /* View Serial (Off)      */
load C:\8051\C51\BIN\8051.IOF /* IOF driver for 8051    */
map 0xA800,0xB000                /* XDATA memory 2 kB      */
```

In einer derartigen Kommandodatei ist \ für ALT (zum Beispiel \u für die Tastenkombination ALT-U) zu schreiben; alle nicht als C-Kommentare unwirksam gemachten Kommandozeilen werden ausgeführt.

3.1.1 Handhabung mittels Batch-Files

DS51 kann für Standardfälle bequem mittels eines Batch-Files DS51.BAT

DS51 [*progname*] (ohne Suffix !)

auf I: (siehe dazu Abschnitt 4!) aufgerufen werden, wenn man sich zunutze macht, daß mit dem Kommandozeilenparameter INIT(*dateiname*) eine individuelle Initialisierungsdatei angegeben werden kann. Den Namen der zu testenden .ABS-Datei kann man aber nicht ohne weiteres in die Initialisierungsdatei bringen, daher beruht die Funktionsweise auf dem Kopieren von *progname*.ABS und *progname*.C auf MODULE.* vor Aufruf des DS51.

```
@ECHO OFF
I:
IF "%1" == "" GOTO NoTest
COPY %1.C MODULE.C
COPY %1.ABS MODULE.ABS
C:\8051\DSCOPE\DS51 INIT(C:\8051\DSCOPE\DS51Test.INI)
GOTO Ende
:NoTest
C:\8051\DSCOPE\DS51 INIT(C:\8051\DSCOPE\DS51NoT.INI)
:Ende
@ECHO ON
```

BEISPIELE:

I:>DS51 ROTIER
(Aufruf mit Laden von Rotier.ABS)

I:>DS51
(Aufruf ohne Laden eines zu testenden Moduls)

Zu beachten: alle zusammengehörigen Moduln stehen nach wie vor unter ihrem ursprünglichen Namen zur Verfügung; .OBJ, .M51, .HEX und .LST nur unter diesem! Nur .C und .ABS sind mit ihren ursprünglichen Namen und als MODULE.* vorhanden.

Anmerkung: Ursprünglich wurden in DS51.BAT alle Moduls mit REN %1.* MODULE.* umbenannt; aus unerfindlichen Gründen war dann jedoch DS51 nicht imstande, nach dem Laden von MODULE.ABS den zugehörigen Source-Code MODULE.C zu finden, sodaß kein High-Level-Debugging möglich war.

Die.INI-Dateien haben folgenden Inhalt:

```
/* Initialisation File for dScope-51+ */
\u\u\u\                        /* Increase Exe-Window    */
\v\s                           /* View Serial (Off)      */
load C:\8051\C51\BIN\8051.IOF /* IOF driver for 8051    */
map 0xA800,0xB000              /* XDATA memory 2 kB      */
load MODULE.ABS                /* Load Test Module       */
```

DS51NoT.INI unterscheidet sich vom vorstehend dargestellten DS51Test.INI nur dadurch, daß die letzte Zeile "load ..." fehlt.

3.2 Einfache Beispiele für das Einarbeiten in den DS51

Diese Beispiele sind in Form eines "Tutorial" gehalten und haben den Zweck, den DS51 gut bedienen zu lernen und seine wichtigsten Möglichkeiten kennenzulernen, bevor man ihn seinem eigentlichen Zweck zuführt, Assembler- oder C-Programme zu "debuggen".

3.2.1 Codeübersetzung

Codeübersetzung kommt in der EDV ganz allgemein, insbesondere aber auch in Steuerungsaufgaben sehr häufig vor. Anhand einer Codeübersetzungstabelle sind gegebene Codes in die entsprechenden aus der Tabelle zu übersetzen, zum Beispiel:

Codetabelle	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'A'	'B'	'C'	'D'	'E'	'F'
(hexadezimal	30	31	32			...				39	41	42	43	44	45	46)

Das Codeübersetzungsprogramm besteht aus einem einzigen Befehl:

```
MOVC  A,@A+DPTR
```

Wenn der DTPR auf die Codetabelle zeigt (das heißt, auf ihren Anfang), und A enthält einen Wert von 0 bis 15, so lädt der vorstehende Befehl jenen Byteinhalt aus der Codetabelle nach A, welches auf der Adresse liegt, die im DPTR steht, jedoch erhöht um den ursprünglichen Inhalt von A. Wenn zum Beispiel der Inhalt von A gleich 5 ist und der DPTR auf den Anfang der Codetabelle zeigt (wo '0' steht), zeigt @A+DPTR und 5 Bytes weiter, also dorthin, wo '5' = 35h steht. Dieser Inhalt wird nach A geladen.

(1) Eingeben des Programms:

Im EXE-Fenster ASM eintippen oder in der Menüzeile ASM anklicken, Befehl MOVC A,@A+DPTR eingeben, mit ^C oder Eingabe eines Punktes den Vorgang beenden. Der Befehl erscheint im Language-Fenster an der gerade aktuell gewesenen Adresse (die Adresse wurde inzwischen um die Befehlslänge, hier 1 Byte, erhöht). Falls man auf eine andere Adresse assemblieren will, gibt man das Kommando ASM *adresse* ein, zum Beispiel ASM 0A100 (Assemblerkonvention 0 vor A bis F in einer hexadezimalen Konstanten beachten!).

Language-Fensters danach:

```
 Options   Key   View   Peripheral   Map   Help   Cpu
0000H        MOVC       A,@A+DPTR
```

Inhalt des EXE-Fensters:

```
>asm
0000H      movc a,@a+dptr            NOP
0001H      .                         NOP
```

(rechts steht in jeder Assembler-Zeile der frühere Inhalt auf dieser Adresse)

(2) Übersetzungstabelle im Code-Speicher ab Adresse 0001 eintragen:

ASM 1 eingeben und ab der dann aktuellen Adresse 0001 die ersten Buchstaben des Alphabets eingeben: ABCDE...., natürlich in einer den

Assemblerkonventionen (oder C-Konventionen) entsprechenden Schreibweise, also mit DB und einer Konstanten; nur die Konstante allein ändert nichts!

Inhalt des EXE-Fensters nachher:

```
>asm 1
0001H       db 'A'                  NOP
0002H       db 'B'                  NOP
0003H       db 43h                  NOP
0004H       db 0x44                 NOP
0005H       db 'E'                  NOP
0006H                               NOP
```

Language-Fensters:

```
 Options  Key  View  Peripheral  Map  Help  Cpu
0000H        MOVC      A,@A+DPTR
0001H        AJMP      0242H
0003H        ORL       44H,#45H
```

Eine andere Möglichkeit besteht darin, das ENTER-Kommando zu benützen. Um ein Zeichen einzugeben, lautet es EC (Enter Character). Dann ist der gewünschte Speichertyp samt Doppelpunkt und Adresse einzugeben und danach der Speicherinhalt:

```
>ec c:6
C:0006H = 0x00  46h
C:0007H = 0x00  'G'
C:0008H = 0x00
```

Sämtliche ENTER-Kommandos sind folgende:

EB	für Bits
EC	für Zeichen
EI	für Integers (Wörter)
EL	für Long-Integers
EF	für Float-Zahlen
EP	für C51-Pointer

Der Speichertyp ist wie folgt:

C:	für Codespeicher
D:	für DATA-Speicher
I:	für IDATA-Speicher
X:	für XDATA-Speicher
B:	für bitadressierbaren Speicher
P:	für peripheren Speicher (VTREGS)

Zur Kontrolle könnte man sich den Codespeicherinhalt ausgeben lassen. Dazu dient das DISPLAY-Kommando mit den beiden Parametern startadresse, endadresse.

```
>d c:0,10
C:0000  93 41 42 43 44 45 46 47-00 00 00           .ABCDEFG...
>
```

Anmerkung: DS51 versteht nach dem EC-Kommando nicht nur eingegebene Einzelzeichen, sondern auch eine Zeichenkettenkonstante (im Sinne von C, zum Beispiel "ABCDE..." . Dadurch kann das Eingeben der Übersetzungstabelle entscheidend vereinfacht werden.

(3) Register setzen:

Das Registerfenster kann nicht angewählt werden, es ist daher auch nicht möglich, darin durch Überschreiben Registerinhalte zu setzen. Für das Assemblerprogrammieren mittels des Inline-Assemblers des DS51 sowie für das Testen von Assemblerprogrammen ist das Setzen von Registerinhalten jedoch oft unumgänglich.

Alle Registerwerte können einfach mit C-Anweisungen gesetzt werden, im konkreten Fall zum Beispiel

```
A=5
```

(4) Programm ausführen:

Da das Programm nur einen Befehl enthält, bietet sich die Einzelschrittausführung an. Möglich ist das GO-Kommando:

G [*startadresse*][,*stopadresse*]

Damit wird der Program Counter (PC) auf *startadresse* gesetzt und das Programm bis zum temporären Haltepunkt *stopadresse* ausgeführt.

In unserem Fall soll nur der Befehl auf Adresse 0 ausgeführt werden. Er ist nur 1 Byte lang, also ist die Stopadresse 1:

```
G 0,1  oder kurz  G,1
```

Nach Erreichen eines temporären Haltepunkts wird dieser ungültig; es kann mit G fortgesetzt werden.

Falls der PC (Program Counter, = $) auf 0 steht, kann auch das TRACE-Kommando gegeben werden, mit dem ein Befehl oder mehrere ausgeführt werden:

T [*ausdruck*]

ausdruck gibt an, wieviele Schritte auszuführen sind; im Falle des Weglassens gilt 1. Ein Einzelschritt wird auch mittels ALT-T ausgeführt. In unserem Fall könnte man eingeben

```
T  oder T 1  oder ALT-T .
```

Im Registerfenster wird danach Register A mit 45 angezeigt, PC steht jetzt auf 0001 und sonst hat sich nichts geändert. Die beiden geänderten Inhalte sind für die Dauer eines Befehls oder Kommandos rot.

Um den PC auf 0 (oder eine andere gewünschte Position) zu setzen, kann man die symbolische Adresse $ benützen ($ hat immer den Wert, der der Adresse des nächsten auszuführenden Befehls entspricht; beim Assemblieren ist das der Inhalt des **Location Counters**):

```
$=0    (Dies ist also im Stil einer einfachen C-Anweisung möglich)
```

(5) Folgende Übungsvarianten sollten programmiert werden:

(5.1) In den Akku wird eine Zahl n eingegeben; die Übersetzung liefert den nten Buchstaben des Alphabets, also etwa für n=1 "A". Für n>26 soll keine Übersetzung stattfinden.

(5.2) In einer Schleife wird endlos das ganze Alphabet durchlaufen.

(5.3) Wie (5.2), jedoch bleibt das Programm nach einem Alphabetdurchlauf auf einem Haltepunkt stehen.

3.2.2 Registerbänke 0 und 1 mit von Null aufsteigenden Werten füllen

Das Programm soll die Register R0 bis R7 sowie R0' bis R7' mit den Werten 0 bis 15 füllen. Dabei macht man sich zunutze, daß die Register auch durch ihre Adressen angesprochen werden können (indirektes Adressieren mit @R0).

Das Programm kann nach Eingabe von ASM im Exe-Fenster wie folgt eingegeben werden (Kleinschreibung und abgekürzte Schreibweisen werden vom DS51 in Großbuchstaben bzw. vollständige Assemblerschreibweise übersetzt; symbolische Adressen können nicht eingegeben werden):

```
0000H        mov r0,#0                 NOP
0002H        mov @r0,0                 NOP
0004H        inc r0                    NOP
0005H        cjne r0,#16,2             NOP
0008H        mov r0,#0                 NOP
000AH        sjmp 0                    NOP
```

Im Language-Fenster sieht das Programm dann so aus:

```
 Options   Key  View   Peripheral   Map   Help
0000H         MOV        R0,#00H
0002H         MOV        @R0,00H
0004H         INC        R0
0005H         CJNE       R0,#10H,0002H
0008H         MOV        R0,#00H
000AH         SJMP       0000H
000CH         NOP
```

Für den Ablauf sind Haltepunkte zweckmäßig:

Temporäre Haltepunkte können mit dem GO-Kommando festgelegt werden; solche Haltepunkte werden nach einmaligem Erreichen gelöscht:

```
G [startadresse][,stopadresse]
```

Bleibende Haltepunkte können mit dem BS-Kommando (BreakpointSet) gesetzt, mit Bedingungen verknüpft und mit einem Zähler versehen werden, wodurch der Haltepunkt erst nach dem *n*-ten Erreichen bei zutreffenden Bedingungen wirksam wird.

```
BS stopadresse [,zähler [,kommandoString]]
```

Nach Erreichen eines bleibenden Haltepunkts kann nicht mit G fortgesetzt werden, sondern nur mit einem Trace-Befehl, wenigstens bis zum nächsten Befehl bzw. zur nächsten Anweisung.

Falls ein Kommando-String angegeben ist, wird der Programmlauf nicht angehalten, sondern nur das Kommando ausgeführt, ausgenommen die Systemvariable _BREAK_ ist auf 1 (binär!) gesetzt; das macht man so:

```
>_BREAK_ = 1        (im Stil einer gültigen C-Anweisung)
>_BREAK_            gibt den Inhalt aus
0x01
>EVAL _BREAK_       gibt den Inhalt dezimal, oktal und hexadezimal aus,
1T 1Q 1H '....'         wenn möglich auch in ASCII
>
```

Haltepunkte werden farblich hervorgehoben. Man kann auch eine Liste der Haltepunkte mit BL (Breakpoint List) ausgeben lassen, zum Beispiel

```
>bl
 0: (E C:0x0) '0', CNT=1, enabled
 1: (E C:0x5) '5', CNT=1, enabled
>bk 1
```

Das dritte Feld ist eine Wiedergabe des Adreßausdrucks, den man im BS-Befehl eingegeben hat und dient der leichteren Identifikation in der Liste.

Ferner kann man durch Angabe der am Anfang jeder Zeile stehenden Breakpoint-Nummer im BK-Kommando (Breakpoint Kill) den gewünschten Haltepunkt löschen. Überdies kann man mit BD (Breakpoint Disable) bzw. BE (Breakpoint Enable) bestehende Haltepunkte vorübergehend außer Kraft bzw. wieder in Kraft setzen, ohne sie zu löschen und dann neu eingeben zu müssen:

```
BK ┐
BD │ nummer [,nummer[,...]]
BE ┘
```

(* für *nummer* bedeute: alle Haltepunkte)

Ein Haltepunkt auf Adresse 0 bewirkt, daß das Programm nach ordnungsgemäßem Durchlauf stehen bleibt, einer auf Adresse 5 ermöglicht das Verfolgen des Prozesses nach jedem Schleifendurchlauf:

```
>bs 0
>bs 5
>bl
 0: (E C:0x0) '0', CNT=1, enabled
 1: (E C:0x5) '5', CNT=1, enabled
>
```

Um das Ergebnis in allen 16 Registern zu sehen, könnte man zum Beispiel einen Haltepunkt mit Kommandostring auf Adresse 0Ah setzen, wie etwa

```
>bs 0ah,1,"d d:0,15"
```

(DISPLAY-Kommando, DATA-Speicher von 0 bis 15)

Das Programm gibt dann am Ende folgende Ausgabe:

```
D:0000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F  ............
```

Die Register der Registerbank 1 werden angezeigt, wenn die **Register-Select-Bits** RS0 und RS1 im PSW auf 01 stehen. Dies schaltet man einfach um mit

RS0=1 (im PSW wird dann R1 angezeigt, das bedeutet "Bank 1")

Folgende Übungsvarianten sollten programmiert werden:

(1) Die Register sollen um den Inhalt von B (B zwischen 1 und 7) höhere Inhalte erhalten, jedoch modulo 16; das heißt, wenn zum Beispiel (B)=2, dann wird (R0)=2, (R1)=3 usw., jedoch (R6')=0 (und nicht 17) sowie (R7')=1 (und nicht 18).
Anmerkung: (*register*) bedeutet Inhalt des angegebenen Registers.

(2) Die Register sollen mit absteigenden Werten von 15 bis 0 gefüllt werden.

3.2.3 Neunerkomplement einer mehrstelligen BCD-Zahl bilden

Vorbemerkung:
Das Neunerkomplement einer Dezimalzahl besteht aus Ziffern, die gegenüber den ursprünglichen Ziffern durch Subtraktion von 9 entstehen; zum Beispiel ergibt sich das Neunerkomplement von 3571 zu 6428, weil 3571 + 6428 = 9999. Das

Zehnerkomplement ist jene Zahl, die durch Subtraktion der ursprünglichen Zahl von jener ganzen Zehnerpotenz entsteht, welche um eine Stelle mehr hat als die ursprüngliche Zahl, und ist daher um 1 größer als das Neunerkomplement (10000 - 3571 = 6429).

Es ist bemerkenswert, daß dementsprechend im Binärsystem negative Zahlen mittels Zweierkomplements dargestellt werden, damit unter Wegfall des Überlaufbits die Summe einer Zahl und der dem Betrag nach gleichen negativen Zahl Null ergibt, zum Beispiel bei 3 Bit langen Zahlen:

```
      3d           011b
    +(-3d)        -101b  oder: Einerkomplement bilden = 100b, zu Null addieren
                  -----        und um 1 erhöhen, ergibt 101b = -3d
  =   0d          1000b
                  └ dieses Überlaufbit wird nicht mitgezählt!
```

Also: eine Binärzahl wird negativ gemacht, indem man ihr Zweierkomplement bildet, und das geht am besten, indem man das Einerkomplement bildet (alle Ziffern invertiert) und 1 addiert.

Nun zum konkreten Beispiel:

Die BCD-Zahl steht im Datenspeicher ab einer Adresse, auf die R0 zeigt, die Anzahl Bytes, welche von der Zahl belegt sind, steht in R2.

Dieses Beispiel enthält recht viele Sprungadressen (Labels), und da zeigt sich schon die Schwäche des DS51 für die Direkteingabe eines Assemblerprogramms, da man beim Eingeben eines Vorwärtssprungs nicht weiß, auf welche Adresse das Sprungziel zu liegen kommen wird; symbolische Labels sind aber nicht erlaubt. Alle künftigen Beispiele werden daher mit dem Assembler bzw. in C geschrieben.

So könnte man das Programm eingeben:

```
0000H       clr c
0001H       clr a
0002H       subb a,@r0
0003H       jc 7
0005H       sjmp 0eh
0007H       jz 0bh
0009H       subb a,#6
000BH       subb a,#60h
000DH       setb c
000EH       mov @r0,a
000FH       inc r0
0010H       djnz r2,1
0012H       dec r0
0013H       mov a,@r0
0014H       inc a
0015H       mov @r0,a
```

Und so sieht es dann im Language-Fenster aus (die Kommentare sind nur hier im Skriptum zum besseren Verständnis der Funktion hinzugefügt, die Zeilen sind etwas zusammengeschoben, um Platz für die Kommentare zu bieten):

```
0000H  CLR   C          ; Carry-Flag löschen
0001H  CLR   A          ; Akku löschen
0002H  SUBB  A,@R0      ; 0 - 35h = CBh, das ist negativ (Carry gesetzt)
0003H  JC    0007H      ; wenn negativ: weiter bei 7h
0005H  SJMP  000EH      ; sonst: Fertig
0007H  JZ    000BH      ; wenn Akku=0: weiter bei 0Bh
0009H  SUBB  A,#06H     ; Dezimalkorrektur: C4h
000BH  SUBB  A,#60H     ; Dezimalkorrektur: 64h
000DH  SETB  C          ; Carry wieder setzen
000EH  MOV   @R0,A      ; Dieses Byte ist fertig
000FH  INC   R0         ; nächstes Byte
0010H  DJNZ  R2,0001H   ; R2 dekrementieren; wenn nicht Null: weiter 1h
0012H  DEC   R0         ; alles fertig: letztes Byte adressieren
0013H  MOV   A,@R0      ; in den Akku bringen
0014H  INC   A          ; und um 1 erhöhen
0015H  MOV   @R0,A      ; wieder abspeichern
```

Im Beispiel wird vor Ablauf des Programms auf die Adresse D:10h 3571 geschrieben; dies ist die Zahl, deren Neunerkomplement gesucht ist. R0 wird mit 10h geladen; dann kann das Programm mit einem Haltepunkt auf 16h ablaufen. Das Ergebnis ist dann 6428.

Zum Testen setzt man die Register mit den Kommandos

```
R0=10h
R2=2
```

und dann auf D:10h die Zahl 3571h am einfachsten mit

```
>ei d:10h
D:0010H = 0x0  3571h
D:0012H = 0x0  .
>
```

Zur Kontrolle könnte man noch nachschauen:

```
>d d:10h,11h
D:0010  35 71
>
```

Nun könnte man den Program Counter auf den Programmanfang stellen

```
$=0
```

und mit ALT T die Funktionsweise erforschen.

Weitere Beispiele können dem Kapitel 4 entnommen werden.

3.3 Watch-Fenster im DS51

Im Abschnitt 4 wird zunehmend vom symbolischen Debuggen Gebrauch gemacht werden, das heißt, die im Quellcode definierten Symbole scheinen auch im DS51 auf. Es sind dies Sprungziele und Variablennamen.

Die (symbolischen) Sprungziele erscheinen im Language-Fenster. Die Variablen können in einem Watch-Fenster beobachtet werden.

Das Kommando

WS *ausdruck*	(Watch-Set)

setzt eine von bis zu 16 Watch-Zeilen. ausdruck kann dabei eine Variable, aber beim C-Debuggen auch eine Struktur, ein Array sowie ein Zeiger oder das Objekt auf das er zeigt (*Zeiger) sein.

Die Watch-Zeilen sind numeriert und können mit

WK *nummer*[,*nummer*[...]] bzw. WK * (bezieht sich auf alle Watch-Zeilen)

entfernt werden (Watch-Kill).

DS51 weist von vornherein keinen externen Datenspeicher zu; man muß dies selbst tun mittels

MAP *startadresse*,*endadresse*

Man sollte nicht mehr externen Datenspeicher zuweisen als tatsächlich verwendet wird, da sonst Fehler durch falsches Adressieren leichter unbemerkt bleiben können und erst in der Zielhardware Schwierigkeiten verursachen.

MAP zeigt die Map-Zuweisungen an, RESET MAP löscht sie.

3.4 Kurze Systematik von DS51

3.4.1 Pulldown-Menüs (geltende Einstellungen sind durch » gekennzeichnet)

```
|Options| Key  View  Peripheral  Map  Help  Cpu

  Load...          Laden von Programmdateien oder .IOF-Dateien
  eXit             Beenden von DS51
  Dos shell        zwischendurch ins DOS, zurück zu DS51 mit EXIT
» Bell enable      Akustisches Zeichen von Fehleingaben Ein/Aus
» Page pause       Ausgaben im Exe-Fenster seitenweise (Leertaste drücken)
  mOnochrom        schwarz/weiß-Einstellung für LCD-Displays oder Mono-Schirme
» Normal           25 Zeilen-Darstellung
  Medium           40 Zeilen-Darstellung
  High             50 Zeilen-Darstellung
```

```
Options |Key| View  Peripheral  Map  Help  Cpu

          List          aktuelle Belegung aller Funktionstasten
          Add...        Funktionstaste mit Kommando belegen, Inhalt so:  " .. "
          Kill all      alle Funktionstasten-Belegungen löschen
```

```
Options  Key |View| Peripheral  Map  Help  Cpu

             » Highlevel       Hochsprache im Language-Fenster
               Mixed           Hochsprache und Assemblersprache
               Assembly        Assemblersprache im Language-Fenster
               Itrace          aufgezeichnete Trace-Daten werden dargestellt
               window
               Serial          Fenster für serielle Ein-/Ausgabe dargestellt
               reFresh         Bildschirm neu aufbauen (nach DOS-Fehler)
             watch
               Define...       Watch-Ausdruck eingeben
               Remove...       Watch-Zeile entfernen
               Kill all        Alle Watch-Zeilen entfernen
             » auto Update     Watches ständig aktualisieren
```

```
Options  Key  View |Peripheral| Map  Help  Cpu

    Anzeige der      i/o Port         aktuellen Port-Zustände
    Anzeige der      Interrupt        aktuellen Interrupt-Parameter
    Anzeige der      Timer            aktuellen Timer-Zustände
    Meldung bei      int Message      Auftreten eines Interrupts
    Anzeige der      a/d Converter    aktuellen a/d-Konverter-Zustände
    Anzeige der      Serial           Zustände der seriellen Schnittstelle
    Anzeige der      Other            Zustände anderer Peripheriekomponenten
    Zuweisung der    Assign...        seriellen Schnittstelle zu COM1 oder COM2
```

```
Options  Key  View  Peripheral |Map| Help  Cpu
                               +----------+
                               | Set...   |  XDATA-Größe einstellen
                               | Reset    |  XDATA-Einstellungen löschen
                               | Display  |  XDATA-Einstellungen anzeigen
                               +----------+
```

```
Options  Key  View  Peripheral  Map |Help| Cpu
                                    +-----------------+
 Allgemeines, Konventionen, Begriffe| Introduction    |
 Aufruf                             | inVocation      |
 Tastenfunktionen, Begriffe         | Keyboard        |
 Ausdrücke: Bedeutung, Schreibweise | Expression      |
 Adressen, Variablen, Funktionen    | Symbol          |
 Erklärung der Menüs                | Pull-down menu  |
 Erklärung der Maussteuerung        | Mouse           |
 Haltepunkte                        | Breakpoint      |
 C-orientierte Funktionen schreiben | Functions       |
 eingebaute Funktionen              | bUilt-in funcs  |
 Erklärung der Fenster              | Window          |
 Simulieren integrierter Peripherie | Derivative      |
                                    == commands =======  DS51-Kommandos
                                    | Help commands   |
 Symbole, Funktionen verwalten      | sYmbol/funcs    |
 Programm / Daten zeigen / verändern| mOdify/display  |  Ausführen, Trace,
 Haltepunkte usw.                   | eXecution       |
                                    +-----------------+
```

```
Options  Key  View  Peripheral  Map  Help |Cpu|
                                          +----------+
 Ausführen ab derzeitigem Stand des PC    | Go       |
 Rücksetzen (wie Hardware-Reset)          | Reset    |
 Trace-Aufzeichnung zuschalten            | Itrace   |
 Einzelschrittbetrieb (einstellbare       | Animate  |
                      Geschwindigkeit)    +----------+
```

3.4.2 Die wichtigsten Kommandos

Grundregel: Beenden jeder Funktion durch Eingabe von CTRL-C. ALT-R

Registerfenster wegschalten / zuschalten
ASM [*adresse*]   in den Assemblermodus umschalten
U   [*adresse*]   Disassemblieren (Discompilieren, wenn möglich)

D   [*startadresse*[,*endadresse*]]   Speicherbereich ausgeben, *startadresse*
mit vorgesetztem Speicherbereichscode (C:, X:, D:, I:, B:)
OBJ *ausdruck*[,*zahlenbasis*]   strukturierte Daten komplett anzeigen
E*x* *adresse*   Speicherinhalt ändern.
x = B (Bit), C (Character=Byte), I (Integer=Wort),
L (Long Integer), F (Float), P (Pointer)
MAP *startadresse*,*endadresse*   externen Speicher (XDATA) zuweisen.
MAP   MAP-Zuweisungen anzeigen
RESET MAP   MAP-Zuweisungen löschen

WS *ausdruck*[,*zahlenbasis*] Watch setzen (Set), *zahlenbasis* 16 oder 10
WK *nummer* Watch entfernen (Kill)
WK * alle Watches entfernen

G [*startadresse*] [,*stopadresse*] Ausführung starten, temporären Haltepunkt setzen. Adressen auch symbolisch möglich.
T [*ausdruck*] einen Schritt oder *ausdruck* Schritte ausführen
P [*ausdruck*] wie T, jedoch gilt Unterprogrammaufruf als ein Schritt
ALT-T Einzelschritt ausführen

BS *ausdruck* [,*count* [,"*kommando*"]] Haltepunkt setzen (Breakpoint Set):
a) auf die Adresse *ausdruck*
b) *ausdruck* ist ein Wahrheitswert, Anhalten wenn WAHR (ungleich 0)

count gibt an, beim wievielten Erreichen der Break-Bedingung ein Anhalten erfolgen soll.
kommando wird nach Anhalten ausgeführt (wie Eingabe im Exe-Fenster)

BL Liste aller Haltepunkte anzeigen

BD *nummer* [,*nummer* ...] Haltepunkte entfernen (Delete)

BK * alle Haltepunkte entfernen (Kill)

LOAD *dateiname* .ABS- oder .IOF-Datei laden
SETMOD .*modul*=*quelldatei* in modularen Programmen Quelldatei einem Modul zuordnen
DIR Symbolnamen anzeigen:
- PUBLIC alle globalen
- VTREG außen zugängliche Register
- .*modul* Symbole des angegebenen Moduls
- (und andere)

LOG >*logDateiname* Log-Datei anlegen (protokollieren aus dem Exe-Fenster)
LOG >>*logDateiname* an bestehende Log-Datei anhängen
LOG Log-Status anzeigen

LOG OFF Log-Datei schließen

INCLUDE *dateiname* Kommandodatei ausführen (enthält DS51-Kommandos)

EVAL *ausdruck* *ausdruck* berechnen und anzeigen

RESET Rücksetzen (wie Reset des Prozessors)

3.4.3 Testen von µProfi-Programmen mit DS51

Auch µProfi-Programme können im DS51 getestet werden, Vorsicht ist jedoch geboten, wenn Interrupt-Routinen verwendet werden:

Im µProfi-Programm sind alle Interrupt-Einsprungadressen um 0A000h höher als im 8051-Controller selbst (zum Beispiel aus der Timer-0-Einsprungadresse 0Bh wird 0A00Bh). Dies ist im Programm berücksichtigt (siehe zum Beispiel nachstehender Ausschnitt eines Porgrammanfangs):

```
         ORG    0A000h
         SJMP   Anfang
         ORG    0A00Bh  ; Timer-Interruptadresse
         LJMP   IntRout ; Sprung zur Interruptroutine
Anfang:  MOV    SP,#20h ; Stackanfang festlegen
```

Um ein solches Programm im DS51 zu testen, muß auf 0Bh ein LJMP IntRout gelegt werden (mittels ASM-Befehls), da sonst die Interrupt-Routine nicht erreicht wird.

Näheres zu den Besonderheiten des Assemblierens für den µProfi ist im Abschnitt 4.1.2 enthalten.

4. Softwareentwicklung mit Assembler und C sowie Testen mittels DS51

Alle in diesem Abschnitt enthaltenen Programme wurden bereits für den Assembler bzw. in C und nicht direkt im DS51 geschrieben.

Wegen der Orientierung am µProfi wird in dieser Lernhilfe nur auf die Software von INTEL (ASM51 samt Dienstprogrammen) sowie KEIL (L51) eingegangen.

Die folgenden Beschreibungen basieren auf folgender Konfiguration, welche auf dem eigenen Computer zu übernehmen empfohlen wird. Eine problemlos installierbare .ZIP-Datei mit aller erforderlichen Software steht zur Verfügung:

Auf der Festplatte C befinden sich in entsprechenden Unterverzeichnissen Software und zu entwickelnde Programme:

```
C:\8051
    ├──ASM51   enthält ASM51, RL51, OHS51, A000.COM
    │  └──PRG  zu entwickelnde .ASM-Programme, als Laufwerk I: substituiert
    ├──C51
    │  ├──BIN  enthält C51-Compiler und dScope-51 sowie TScope51
    │  ├──H    enthält Header-Dateien
    │  ├──LIB  enthält Bibliotheken
    │  └──PRG  zu entwickelnde .C-Programme, als Laufwerk I: substituiert
```

4.1 ASM51

Hier wird der Assembler ASM51 von INTEL samt zugehörigem Linker und anderen Hilfsprogrammen beschrieben.

4.1.1 Grundlegendes

Die Software steht auf C:\8051\ASM51, das zu entwickelnde Programm auf dem aktuellen Verzeichnis C:\8051\ASM51\PRG, welches als I: substituiert ist. Nach Fertigstellen des Quellprogramms (vorzugsweise mittels SideKick) und Abspeichern in C:\8051\ASM51\PRG ruft man ASMABS *progname* [*a*] auf:

```
@ECHO OFF
C:\8051\ASM51\ASM51 I:%1.ASM db ep(%1.err)       assembliert das Quellmodul
IF "%2" == "" GOTO Load0
C:\8051\C51\BIN\L51 I:%1.OBJ TO %1.ABS CODE(0A000h)   linkt das Objektmodul
GOTO Pause                                            auf A000h
:Load0
C:\8051\C51\BIN\L51 I:%1.OBJ TO %1.ABS                linkt das Objektmodul
:Pause                                                auf 0000h
C:\8051\ASM51\OH51 I:%1.ABS                 wandelt in INTEL-HEX-Format um
PAUSE                                    gibt Zeit zum Lesen der Meldungen
C:\UT\LIST I:%1.LST            bringt Assembler-Listing auf den Bildschirm,
:Ende                          wegschalten mit X-Taste
@ECHO ON
```

Diese Batch-Datei ist primär für einfache Programme gedacht, die nicht mit anderen Moduln zusammengelinkt werden und auf vorgegebene Adresse zu laden sind (daher Dateierweiterung .ABS).

Der weiteren Darstellung wird das einfache Additionsprogramm ADD.ASM zugrundegelegt:

```
Add1:       MOV A,#12      ; 12 in Akku laden
            ADD A,#58      ; Addition von 58 ohne Übertrag, ergibt 70 = 46h

Add2:       MOV R1,#30h    ; 30h in R1 laden
            ADD A,R1       ; 30h = 48 zum Akku addieren, ergibt 108 = 76h
            END ;
```

Eine Eigenheit des Assemblers ist, daß hinter dem END ein Kommentar oder wenigstens der diesen einleitende Strichpunkt ; eine absurde Fehlermeldung verhindert. Hinter dem Strichpunkt bzw. dem letzten Zeichen des Kommentars darf nichts mehr in der Quellprogrammdatei stehen, nicht einmal ein Zeilenvorschubzeichen.

Der Assembler erzeugt das .OBJ-File *progname*.OBJ, ein Programmlisting *progname*.LST, eine Fehlerdatei *progname*.err. Für ein Modul, das verschieblich sein soll und vom Linker auf die µProfi-Ladeadresse A000h gestellt werden soll, sind allerdings SEGMENT- und RSEG-Anweisung erforderlich:

```
SegName    SEGMENT  CODE        ; vereinbart ein Segment im Codespeicher
           RSEG     SegName     ; legt fest, daß die nachfolgenden Befehle
           .                    ; im Segment mit dem angegebenen Namen
           .                    ; liegen
           END ;
```

Damit wird ein **verschiebliches** Segment mit dem angegebenen Namen und vom angegebenen Typ (hier CODE, andere Möglichkeiten XDATA, IDATA, DATA, BIT) vereinbart. Erst der Linker stellt die Segmente auf ihre endgültigen Adressen.

DS51 kann zwar .HEX-Dateien laden, nicht jedoch .M51-Dateien, welche Debugging- und Symboltabelleninformationen enthalten. Nach dem Laden einer .HEX-Datei ist daher Source-Level-Debugging nur unter Verzicht auf symbolische Adressen möglich, daher im besten Fall für einfache Assemblerprogramme akzeptabel.

Der Intel-Linker RL51 ist andererseits nicht imstande, eine Ausgabedatei mit enthaltenen Symboltabelleninformationen zu erzeugen; er erzeugt nur die .M51-Datei, die wiederum vom DS51 nicht verwertet werden kann. Daher wird der zu C51 gehörende Linker L51 verwendet, der Intel-Objektformat-Dateien erzeugt, welche alle erforderlichen Debugging- und Symboltabelleninformationen enthalten.

Um ein in den µProfi ladbares Programm zu erhalten, ist es am einfachsten, das Programm **verschieblich** zu schreiben und das Verschieben auf die richtige Ladeadresse dem Linker zu überlassen:

```
CodeSeg      SEGMENT  CODE
             RSEG     CodeSeg
Add1:        MOV A,#12     ; 12 in Akku laden
             ADD A,#58     ; Addition von 58 ohne Übertrag, ergibt 70 = 46h

Add2:        MOV R1,#30h   ; 30h in R1 laden
             ADD A,R1      ; 30h = 48 zum Akku addieren, ergibt 108 = 76h
             END ;
```

Der zweite Parameter in der Batch-Datei ASMABS.BAT a darf weggelassen werden, jedoch nur für Programme (Programmteile), die nur im DS51 im internen Codespeicher "ausprobiert" werden sollen und daher auf einer sehr niederen Adresse geladen werden. Wenn der zweite Parameter fehlt, gilt als Ladeadresse 0, was für das Testen einfacher Lern- oder Ausprobierprogramme im DS51 recht praktisch ist.

Programme, die im µProfi laufen sollen, müssen auf die Ladeadresse 0A000h gelinkt werden (und schon im Quellcode entsprechende Direktiven enthalten); dazu gibt man dem zweiten Parameter a irgendeinen Wert, zum Beispiel am besten A (für Ladeadresse A000).

Der Linker erzeugt die nicht mehr verschiebliche Datei *progname*.ABS und die Linker-Map-Datei *progname*.M51 (die Erweiterung .ABS ist von uns festgelegt und kein Default des Linkers).

Am Ende erzeugt die Batchdatei ASMABS.BAT mittels OH51 auch noch die INTEL-HEX-Datei, die im konkreten Fall wie folgt aussieht (Näheres siehe auch Abschnitt 4.2.2.3).

```
00   3A 30 37 30  30 30 30 30  30 37 34 30  43 32 34 33   :07000000740C243
10   41 37 39 33  30 32 39 34  39 0D 0A 3A  30 30 30 30   A79302949··:0000
20   30 30 30 31  46 46 0D 0A                              0001FF··
```

Falls mehrere externe Moduln geschrieben und dann gelinkt werden sollen (was für größere Programme an sich empfehlenswert ist), müssen insbesondere die Unterprogramme verschieblich sein. Dies erfordert SEGMENT-Anweisungen für Code- und Datensegmente. Der Linker ist dann zum Beispiel mit

```
L51 modul1.OBJ,modul2.OBJ,...
```

aufzurufen.

4.1.2 Assemblieren im Hinblick auf den µProfi

Der µProfi ist hardwaremäßig so ausgelegt, daß Programme im externen Programmspeicher ab Adresse A000h liegen müssen, um ausführbar zu sein.

Dies erfordert entsprechende Vorkehrungen im Programm bzw. beim Linken.

Ein zweckmäßiger Assembler-Aufruf (wie in ASMABS.BAT enthalten) ist folgender:

```
ASM51 progname.ASM DB EP(progname.err)
```

DB ... DeBug-Information in .OBJ-File
EP ... Error Print
zusätzlich eventuell sinnvoll: XR ... Cross-Reference-Liste erzeugen

Der folgende Linker-Aufruf linkt den Object-Code auf die Ladeadresse A000h, sofern das Codesegment verschieblich vereinbart ist; andernfalls ist der Parameter CODE(...) unnötig.

```
L51 progname.OBJ TO progname.ABS CODE(0A000h)
```

Aus dem so entstandenen absoluten Modul *progname*.ABS kann eventuell noch ein .HEX-File erzeugt werden:

```
OH51 progname.ABS
```

Falls ein Programm (wie etwa das einfache Beispiel ADD.ASM) ohnehin nicht modular geschrieben ist und die Ladeadresse 0A000h auch festliegt, kann man das Programm auch mit einem absoluten Segment schreiben:

```
            CSEG AT 0A000h
Add1:       MOV  A,#12
            ADD  A,#58    ; Addition ohne Übertrag

Add2:       MOV  R1,#30H
            ADD  A,R1
            END  ;
```

Zu beachten ist übrigens auch, daß der externe Datenbereich des µProfi mit Sicherheit ab 6000h implementiert ist (nämlich auch dann, wenn ein 8k-RAM eingesteckt ist), während mit einem 32k-RAM schon ab 4000h adressiert werden könnte. Aus ähnlichen Gründen sollte das Programm ab 0A000h liegen, Programme ab 8000h sind nur mit einem 32k-RAM möglich.

Echte 8051-Anwendungen erfordern in der Regel eine explizite Segmentierung des Quellprogramms. Dadurch kann man insbesondere in den Datenspeicherbereichen des 8051 bequem symbolisch adressieren.

Die Speicherbereiche werden mit folgenden Symbolen bezeichnet, weiters sind die Segmenttypen angegeben:

CODE ... Segment im Codespeicher	CSEG
DATA ... Segment im direkt adressierbaren internen RAM	DSEG
IDATA ... Segment im indirekt adressierbaren internen RAM	ISEG
XDATA ... Segment im indirekt adressierbaren externen RAM	XSEG
BIT ... Segment im bitadressierbaren Bereich im internen RAM	BSEG

Das folgende Programm SegBsp.ASM ist ein Beispiel für die Segmentierung, in dem von absoluten Segmenten Gebrauch gemacht wird (*x* SEG AT ...):

```
            NAME SegBsp
DataSeg     SEGMENT DATA ; interner Datenbereich
            DSEG AT 50h  ; ab Adressen 50h im internen Daten-RAM
Status:     DS  1        ; 1 Byte für irgendeinen Status
DatISeg     SEGMENT IDATA ; indirekt adressierbarer interner Datenbereich
            DSEG AT 20h  ; ab Adresse 20h
Feld:       DS  2        ; 2 Bytes für irgendwelche Daten
DatXSeg     SEGMENT XDATA ; externer Datenbereich
            XSEG AT 06000h
Ergebnis:   DS  1        ; 1 Byte für irgendein Ergebnis

CodeSeg     SEGMENT CODE
            CSEG AT 0A000h
            MOV Status,#5
Add1:       MOV A,#12    ; 12 in Akku
            ADD A,#58    ; Addition ohne Übertrag
            MOV Feld,A   ; nach Feld abspeichern
```

```
Add2:        MOV R1,#30h  ; Adresse 30h im internen Datenspeicher laden
             MOV @R1,#161 ; 161 dorthin abspeichern
             ADD A,R1     ; 30h zum Akku addieren
             MOV Feld+1,A ; Ergebnis abspeichern (Adreßrechnung: Feld+1)
             MOV DPTR,#Ergebnis ; Adresse des Felds Ergebnis in den DPTR
             MOVX @DPTR,A ; Akkuinhalt im Feld Ergebnis abspeichern
             END ;
```

Da alle Segmente schon im Quellcode absolut bestimmt sind, genügt es, den zweiten Parameter a in ASMABS.BAT wegzulassen.

4.1.3 Programmbeispiele

4.1.3.1 Binär-BCD-Umwandlung

Das Programm BinBCD.ASM erzeugt aus einer 1-Byte-Binärzahl in A eine 2-Byte-BCD-Zahl im Registerpaar R1/R0:

```
ORG  100h
MOV  A,#0A7h    ; diese Binärzahl soll in BCD (167) umgesetzt werden
MOV  B,#100     ; Divisor 100d
DIV  AB         ; Rest jetzt in B
MOV  R1,A       ; Hunderterstelle in R1
MOV  A,#10      ; Divisor 10
XCH  A,B
DIV  AB         ; Rest jetzt in B
SWAP A          ; Zehnerstelle in High-Halbbyte
ADD  A,B        ; Zehner- und Einerstelle jetzt in A
MOV  R0,A       ; Zehner- und Einerstelle in R0
END  ;
```

Beachten: Dieses Programm hat keine Labels (Datenfeldnamen, Sprungadressen usw. Trotzdem sollte man die übliche Spalteneinteilung eines Assemblerprogramms einhalten und insbesondere nicht einfach jede Zeile ganz links beginnen. Als Muster kann das Programm unter 4.1.3.2 dienen.

Beim Testen im DS51 darf nicht vergessen werden, den PC auf 100h zu stellen.

4.1.3.2 Timeranwendung

Timer 0 wird mit einer negativen Zahl geladen und eingeschaltet, während das Programm dann eine Endlosschleife ausführt. Nach Erreichen von Null soll eine Timer-Interrupt-Routine angesprungen werden, die im konkreten Fall den Inhalt von ACC um 1 erhöht, den Timer wieder rücksetzt und das Spiel fortsetzt. Da die Timer-Interruptadresse auf 0Bh vorgegeben ist und dort die Interruptroutine eingeleitet werden muß, wird das Programm auf Adresse 800h gesetzt. Außerdem darf nicht vergessen werden, den Stack-Pointer auf eine etwas höhere Adresse zu setzen, damit der Stack nicht über die Timer-Interruptadresse vorwächst.

```
         ORG    0Bh     ; Timer-Interruptadresse
         LJMP   IntRout; Sprung zur Interruptroutine
;
         ORG    800h
         MOV    A,#0
         MOV    SP,#20h
         CALL   SetTmr  ; Timer initialisieren
Loop:    SJMP   Loop    ; Endlosschleife (simuliert beliebige Aktivität)
;
SetTmr:  CLR    TR0     ; Timer 0 stoppen
         MOV    TH0,#0FFh
         MOV    TL0,#9Ch ; Timer-Anfangswert laden: -100 in 16 Bits
         ANL    TMOD,#0  ; TMOD-Bits löschen
         ORL    TMOD,#00000001b ; T0 zählt als 16-Bit-Zähler internen Takt
         MOV    IE,#10000010b   ; EA und ET0 im IE-Register setzen
                                ; (Enable All, Enable Timer 0)
         SETB   TR0             ; Timer 0 einschalten
         RET
;
IntRout: PUSH   PSW
         INC    A
         CALL   SetTmr
         POP    PSW
         RETI
         END    ;
```

Die Simulation im DS51 kann man zum Beispiel verfolgen, indem man einen Haltepunkt auf SetTmr oder auf den INC-Befehl in der Interrupt-Routine setzt. Um während der Simulation, die sehr rasch abläuft, die Änderung des Akku-Inhalts verfolgen zu können, bietet sich als Übungsaufgabe folgende Variante an:

(1) Den Timer nicht mit -100 initialisieren, sondern mit einer größeren negativenZahl wie etwa -32786 = 8000h oder mit der größtmöglichen negativen Zahl, das ist FFFFh.

(2) Zwecks Beobachtung des Akkuinhalts einen Haltepunkt etwa auf den Befehl nach dem Inkrementierbefehl in der Interrupt-Routine setzen, welcher den Kommandostring "D ACC" aktiviert. _BREAK_ muß auf 0 gesetzt sein, andernfalls wird zwar unterbrochen, aber der Kommandostring nicht ausgeführt.

(3) Eine Verzögerungsschleife in die Interrupt-Routine einbauen, entweder als Zählschleife oder mittels Timer/Counter.

Eine professionelle Methode ist es, das Programm **animiert** laufen zu lassen: ALT-C Animate. Danach wird das Programm durch Eingeben einer Geschwindigkeitsziffer (0 bis 9) gestartet (9 ist die höchste Geschwindigkeit). Mit CTRL-C kann der Programmlauf abgebrochen werden.

Die Geschwindigkeit kann auch jederzeit neu gewählt werden.

4.1.3.3 BCD-Addition

Zwei mehrstellige BCD-Zahlen mit gleicher Stellenanzahl (die Anzahl belegter Bytes steht in R2) sind zu addieren. Die Zahlen werden mit @R0 und @R1 adressiert, das Ergebnis steht dann im ersten Operanden (@R0). Der Stellenwert nimmt bei beiden Zahlen nach links zu, wie üblich.

```
AddBCD:    CLR      C        ; Carry löschen
Schleife:  MOV      A,@R0    ; erster Summand
           ADDC     A,@R1    ; zweiten Summanden addieren
           DA       A        ; Dezimalanpassung
           MOV      @R0,A    ; Ergebnis abspeichern
           DEC      R0
           DEC      R1       ; nächste Summandenbytes
           DJNZ     R2,Schleife
           END      ;
```

Vor dem Testlauf wird im DS51 folgendes eingestellt:

```
>ec d:10h
D:0010H = 0xE6  35h
D:0011H = 0x4E  78h
D:0012H = 0x3E  62h
D:0013H = 0x00  .
>ec d:18h
D:0018H = 0x19  25h
D:0019H = 0x52  82h
D:001AH = 0x31  49h
D:001BH = 0x00  .
>r0=12h
>r1=1ah
>r2=3
>d d:10h,1ah       zur Kontrolle der Speicherinhalte
D:0010  35 78 62 00 00 00 00 00-25 82 49                 5xb.....%.I
```

hier stehen die beiden sechsstelligen BCD-Zahlen

Nach dem Testlauf sind die Daten folgende:

```
                                                R0 = 0F
                                                R1 = 17
>d d:10h,1ah                                    R2 = 00
D:0010  61 61 11 00 00 00 00 00-25 82 49                 aa......%.I
```

dies ist die BCD-Summe

dies ist der unveränderte zweite Operand

4.1.3.4 Halbbytes rotieren

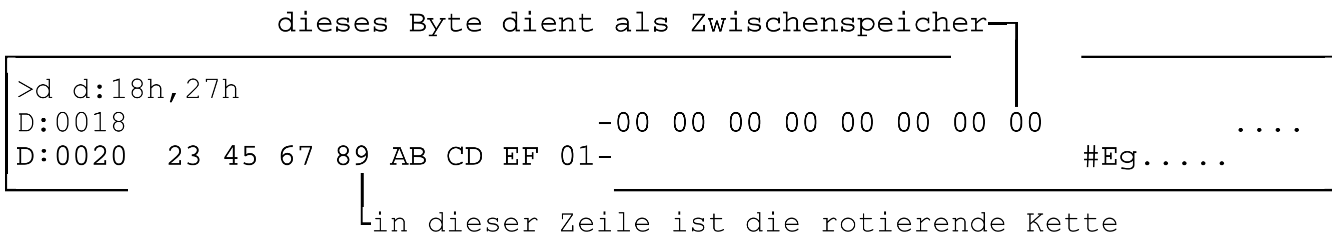
Dieses Programm ist im Gegensatz zu den vorstehenden unter voller Ausnützung der Fähigkeiten des Assemblers hinsichtlich Segmentierung und symbolischen Adressierens geschrieben.

Im internen Datenspeicher stehen einige Bytes Daten; die Anzahl der Bytes steht in R2. Die Halbbyte-Inhalte sind um eine Stelle nach links zu verschieben, das links herausfallende Halbbyte gelangt in die rechts freiwerdende Stelle. In R0 steht die Adresse der Bytekette. Zur Vereinfachung des Programms wird vorausgesetzt, daß das vor der eigentlichen Bytekette stehende linke Nachbarbyte vom Programm mitverwendet werden darf. Die eigentliche Verschiebearbeit wird in einem Unterprogramm ausgeführt.

```
DatenSeg  SEGMENT XDATA
          RSEG  DatenSeg
          ORG   20h
Kette:    DS    8
CodeSeg   SEGMENT CODE
          RSEG  CodeSeg
          MOV   SP,#67h          ; Stack ab Adresse 68h
          ACALL InitKett         ; Ketteninhalt aus Codespeicher holen
Rotieren: MOV   R2,#8            ; Kette 8 Bytes lang
          MOV   R0,#Kette        ; Adresse von Kette laden
          DEC   R0               ; linkes Nachbarbyte adressieren
          PUSH  0                ; R0 am Stack aufheben
          MOV   @R0,#0           ; linkes Nachbarbyte löschen
          INC   R0               ; wieder erstes Byte adressieren
Schleife: ACALL Schieben
          INC   R0               ; nächstes Byte
          DJNZ  R2,Schleife      ; Schleife ausführen, solange noch Bytes da
          DEC   R0               ; R0 auf letztes Byte zurücksetzen
          MOV   A,@R0
          MOV   R1,0             ; Adresse in R0 nach R1 speichern
          POP   0                ; Adresse des linken Nachbarbytes nach R0 holen
          ORL   A,@R0            ; Inhalt mit ACC kombinieren
          MOV   @R1,A            ; Byte auf letztes Byte abspeichern
          SJMP  Rotieren         ; Rotierdurchgang beendet, weiterrotieren
;
Schieben: MOV   A,@R0            ; Byte holen: 12h
          SWAP  A                ; Halbbytes vertauschen: 21h
          PUSH  ACC              ; am Stack aufheben: 21h
          ANL   A,#0Fh           ; linkes Halbbyte löschen: 01h
          DEC   R0               ; linkes Nachbarbyte adressieren: 00h
          ORL   A,@R0            ; kombinieren: 01h
          MOV   @R0,A            ; abspeichern: 01h
          INC   R0               ; wieder Byte adressieren
          POP   ACC              ; 21h
          ANL   A,#0F0h          ; rechtes Halbbyte löschen 20h
          MOV   @R0,A            ; abspeichern: 20h
          RET
;
```

```
InitKett: MOV     DPTR,#KettInit ; Quellfeld adressieren
          CLR     A              ; zum Vermeiden von Codeumwandlung
          MOV     R0,#Kette      ; Adresse des Schriftfelds laden
          MOV     R1,#8          ; 8 Zeichen kopieren
Speicher: PUSH    ACC
          MOVC    A,@A+DPTR      ; Zeichen aus dem Codespeicher holen
          MOV     @R0,A          ; ins Feld Kette abspeichern
          INC     R0             ; nächstes Byte im Feld Kette
          POP     ACC
          INC     A              ; nächstes Byte im Codespeicher
          DJNZ    R1,Speicher    ; Schleife ausführen
          RET
;
KettInit: DB      01h,23h,45h,67h,89h,0ABh,0CDh,0EFh
          END ;
```

Beim Testen wird nach ASMABS ROTIER als nächstes DS51 aufgerufen. Nach zwei Rotierdurchgängen sieht der Datenbereich so aus:



4.1.4 Kurze Systematik von ASM51, L51, OH51

4.1.4.1 ASM51

Eine Assembler-Quellzeile hat folgende Form:
[*label*:] *operation* [*operand*] [,*operand*] [,*operand*] [;*kommentar*]

Jeder mnemonische Operationscode *operation* erfordert zwischen 0 und 3 Operanden.

Im Prinzip kann man Assemblerprogramme ohne SEGMENT- und zugehörige Direktiven schreiben; man nimmt jedoch damit dem Assembler die Möglichkeit, auf formale Fehler aufmerksam zu machen.

Ein **verschiebliches Segment** (dessen absolute Adresse erst vom Linker bestimmt wird) sollte wie folgt beginnen:

```
SegmentName   SEGMENT   Typ          (Typ = CODE, XDATA, DATA, IDATA, BIT)
              RSEG      SegmentName
```

Ein **absolutes Segment** (das von vornherein auf eine bestimmte Adresse festgelegt ist) sollte wie folgt beginnen:

```
SegmentName   SEGMENT   Typ
              tSEG      AT adresse   (t = C, X, D, I, B entsprechend Typ)
```

Das Ende eines Segments ist durch die Angabe eines neuen Segmentanfangs bestimmt; das Ende des letzten Segments durch das physische Ende des Quellprogramms (END).

Adreß-Symbole (*labels*) werden durch einmaliges Vorkommen in der ersten Spalte des Quellcodes (dort mit Doppelpunkt!) definiert und dürfen beliebig oft als Operanden verwendet werden.

Namen, die nicht von vornherein bestimmten Adressen zugeordnet sind, können wie folgt vereinbart werden:

```
    name     EQU   ausdruck              zum Beispiel  ESC  EQU  27
                                                       Hier EQU  $
```

$ bedeutet den momentanen Stand des **Location Counters**.

Bitadressen können Namen zugeordnet werden mit

```
    name     BIT   ausdruck              zum Beispiel  Sw1  BIT  12
```

Speicher wird reserviert (ohne vorgegebenen Inhalt) durch

```
    name     DS    anzahlBytes
```

Speicher mit vorgegebem Inhalt wird vereinbart durch

```
    name     DB    inhalt
```

wobei der Inhalt als Liste von Bytes und/oder Zeichenketten angegeben werden

darf, wie zum Beispiel

```
    Text     DB    13,10,'Meldung 1'
```

Am Ende des Quellcodes muß END stehen:

```
             END   ;
```

Eine Eigenheit des Assemblers ist, daß hinter dem END ein Kommentar oder wenigstens der diesen einleitende Strichpunkt ; eine absurde Fehlermeldung verhindert. Hinter dem Strichpunkt bzw. dem letzten Zeichen des Kommentars darf nichts mehr in der Quellprogrammdatei stehen, nicht einmal ein Zeilenvorschubzeichen.

Der Location Counter kann an jeder beliebigen Stelle im Quellcode auf einen gewünschten Wert gesetzt werden:

```
             ORG   adresse
```

4.1.4.2 L51

Der Linker/Locater L51 ist recht leistungsfähig und vielseitig; er ermöglicht unter anderem das Einbinden externer Objektmodule und sogar **Overlay-Technik.** Da dies aber auch entsprechende Vorkehrungen in den Quellmoduln erfordert, wird hier nicht weiter darauf eingegangen. Für einfache Fälle ohne externe Unterprogramme genügt der Aufruf, wie er in ASMABS.BAT enthalten ist. Näheres kann auch dem Abschnitt 4.2.2.3 entnommen werden.

4.1.4.3 OH51 und INTEL-HEX-Format

Der Objekt-Hex-Symbolkonverter OH51 konvertiert das von L51 erzeugte Binärfile (.ABS) in das INTEL-Hex-Format. Dieses ist das gebäuchlichste Format für den Datenaustausch mit Programmiergeräten und hat den Vorteil, daß die hexadezimalen Inhalte mit jedem ASCII-Editor gelesen und bearbeitet werden können. .HEX-Dateien sind zeilenweise organisierte ASCII-Dateien; jede Zeile stellt einen Satz dar. Am Ende steht ein EOF-Satz.

Jeder Satz hat eine bestimmte Länge n (in Bytes) und ist wie folgt aufgebaut:

Byte	
0	Doppelpunkt : als Zeichen für Satzanfang
1, 2	Satzlänge = Anzahl der Datenbytes
3 bis 6	Ladeadresse des ersten Daten- oder Code-Bytes
7, 8	Satztyp: 00 = Datensatz, 01 = EOF-Satz
9 bis n-4	Daten- oder Code-Bytes
n-3 bis n-2	Prüfsumme: (Null minus Summe aller Daten- oder Code-Bytes) modulo 256
n-1 bis n	CR, LF

BEISPIEL: Das folgende Programm wird mittels ASMABS Add A in ein .HEX-File umgewandelt:

```
            CSEG AT 0A000h
Add1:       MOV  A,#12
            ADD  A,#58    ; Addition ohne Übertrag

Add2:       MOV  R1,#30H
            ADD  A,R1
            END  ;
```

```
:09A00000740C243A793077A1292F
:00000001FF
```

4.2 C51

> Vorbemerkung: In diesem Abschnitt wurde auf Vorarbeiten von Prof. Dipl.-Ing. Franz Fiala, TGM - Abt. EN, zurückgegriffen. Diese gehen teilweise weit mehr in Richtung komplexerer Aufgaben, als der Konzeption dieses Einführungsskriptums entspricht. Für weiterführende Problemstellungen ist daher zu empfehlen, die Ausarbeitungen von Prof. Fiala zu benützen.

C51 ist ein speziell für die 8051-Familie entwickelter **Cross-Compiler** (er läuft auf anderer Hardware als der mit ihm entwickelte Code). Er ist ein erweiterter ANSI-C-Compiler, welcher die Besonderheiten des Mikrocontrollers berücksichtigt; er verwendet das INTEL-OMF-Format (Object Module Format) und kann daher mit ASM51 und kompatiblen Emulatoren/Simulatoren kombiniert werden.

Beim Entwickeln eines Programms empfiehlt es sich, die leistungsfähige Entwicklungsumgebung von TURBO-C (Borland-C) zu nutzen und erst ein formalfehlerfreies und grundsätzlich funktionierendes Programm mit C51 weiterzubearbeiten.

Damit im Quellcode keine Änderung notwendig ist, wurden einige zusätzliche Include-Dateien geschrieben, welche im INCLUDE-Directory von TURBO-C stehen sollten:

REG51.H	enthält die Registerdeklarationen für 8051
TURBOC51.H	deklariert einige 8051-spezifische Datentypen sowie die Speicherbereiche
TURBO51.H	deklariert die Register- und Bit-Mnemonics extern
ABSACC51.H	deklariert Symbole für den direkten Zugriff zu XDATA

C51 ist im wesentlichen ein ANSI-C-Compiler, der jedoch mit 8051-spezifischen Erweiterungen versehen ist. Die üblichen Bibliotheksfunktionen stehen samt enstprechenden Header-Dateien zur Verfügung.

4.2.1 Besonderheiten von C51

C51 unterstützt die folgenden Datentypen:

char, unsigned char int, signed int	1 Byte 2 Bytes	long, signed long float	4 Bytes 4 Bytes	wie ANSI-C
bit (für badr) pointer	1 Bit 1 bis 3 Bytes	sbit (Zugriff sfr auf SFR) sfr16	1 Bit 1 Byte 2 Bytes	

C51 unterstützt entsprechend der Speicherstruktur des 8051 die folgenden Speicherklassen, durch deren Angabe in der Variablenvereinbarung jede Variable explizit entsprechend plaziert werden kann:

C51	entspricht ASM51	C51-spezifisch	
data idata xdata code bdata	DATA (0 bis 7Fh) IDATA (0 bis 0FFh) XDATA (bis 0FFFFh) CODE BIT	pdata	"paged" externer Datenspeicher, unterste 256 Bytes von XDATA, Zugriff mit MOVX @R*i*

Variablen sind wie folgt zu vereinbaren: *datentyp* [*speicherklasse*] *name* ;

```
BEISPIELE: char data Zeichen;
           char code Text[] = "My program";
           float idata Zahl;
           sfr P0=0x80; (Anmerkung: diesbezügliche Deklarationen in REG51.H)
           sbit OV=PSW^2;
           sbit Bit27=BitAdr^37;  (Bitadresse 37 ab BitAdr in bdata)
           unsigned long xdata Array[100];
```

Um größtmögliche Ausführungsgeschwindigkeit zu erreichen, ist es ratsam, häufig benutzte Variablen im internen Datenspeicher abzulegen und nur seltener benötigte oder sehr große Variablen, wie etwa Tabellen, im externen Datenspeicher.

Wenn die Speicherklasse weggelassen wird, ist sie durch das verwendete Speichermodell bestimmt:

	Parameter und lokale Variablen von Speicherklasse	maximale Größe
SMALL	data	128 Bytes.
COMPACT	pdata	256 Bytes
LARGE	xdata	64 kBytes

Das Speichermodell ist als Default SMALL; wird ein anderes gewünscht, gibt es zwei Möglichkeiten:

- Beim Compileraufruf wird ein zusätzlicher Kommandozeilenparameter COMPACT oder LARGE angegeben;

oder

- Das C-Quellprogramm erhält am Anfang eine (oder mehrere) #pragma-Präprozessoranweisung(en). In einer solchen können für das betreffende Modul geltende C51-Kommandozeilenparameter spezifiziert werden.

Zeiger können von zwei Arten sein:

memory specific	Speicherklasse liegt bereits zur Übersetzungszeit fest; Länge 1 oder 2 Bytes, je nach Speicherklasse
generic	Speicherklasse wird erst zur Laufzeit bestimmt; Länge 3 Bytes (1 Byte Speicherklasse, 2 Bytes Adresse)

Zeigervereinbarungen ohne Speicherklasse sind "generic", solche mit Speicherklasse "memory specific".

BEISPIELE: float *pZahl;
           char unsigned data *pZeichen;

Der Zugriff auf absolute Adressen ist mit Makros möglich; sie heißen CBYTE, DBYTE, PBYTE und XBYTE und dienen zum Adressieren im Code-, Data-, paged XData- bzw. XData-Bereich (im paged XData-Bereich wird der MOVX @R*i*-Befehl verwendet). Die zugehörige Header-Datei ist ABSACC.H.

```
Beispiele: xWert = XBYTE[0x1000] ;  schreibt den Inhalt von 1000h nach xWert
           XBYTE[0x2000] = 50    ;  schreibt 50 auf die Adresse x:2000h
           #define xPort XBYTE[0xc000] ; gibt der Adresse C000h den Namen xPort
```

Interrupt-Routinen können direkt in C geschrieben werden, wobei die zu verwendende Registerbank angegeben werden kann Näheres siehe 4.2.2.4).

Funktionen können reentrant erzeugt werden, das heißt, sie können jederzeit auch mehrfach (entweder rekursiv oder während sie aktiv sind von einer Interrupt-Routine aus) betreten werden. Dies erfordert eine aufwendige Stack-Nachbildung (da der gewöhnliche Stack nicht ausreicht); Funktionen sollten daher nur bei echtem Bedarf reentrant erzeugt werden.

Die Parameterübergabe erfolgt mittels CPU-Registern; bei mehr als 3 Parametern mittels fester Speicherbereiche. Funktionsergebnisse werden je nach Datentyp in dafür fest vorgesehenen Registern zurückgegeben. Für Fälle der Einbindung von Assemblerprogrammen werden folgende Register verwendet:

Rückgabewert	Register	Bemerkung
bit	Carry-Flag	
char, 1-Byte-Zeiger	R7	
int, 2-Byte-Zeiger	R6, R7	höchster Stellenwert R6
long, float	R4 bis R7	höchster Stellenwert R4
generic pointer	R1 bis R3	Speicherklasse in R1, Zeiger in R2/R3

BEISPIEL: Nachbildung der C-Funktion toupper, in Assemblersprache:

```
Upper    SEGMENT  CODE
PUBLIC   _toupper       ; Einsprungadresse
RSEG     Upper          ; Linker soll folgenden Code in Segment Upper stellen
toupper: MOV   A,R7     ; Zeichen aus R7 laden
         CJNE  A,#'a',ToUpp1   ; Zeichen kleiner als 'a' ?
ToUpp1:  JC    ToUppR          ;    ja: unverändert zurückgeben
         CJNE  A,#'z'+1,ToUpp2 ;    nein: Zeichen größer als 'z' ?
ToUpp2:  JNC   ToUppR          ;             ja: unverändert zurückgeben
         CLR   ACC.5           ;             nein: Bit 5 löschen
ToUppR:  MOV   R7,A            ; in Rückgaberegister speichern
         RET                   ; Unterroutine verlassen
         END ;
```

Einige Funktionen sind Intrinsic-Funktionen, das heißt, sie sind integrierender Teil der Sprache, reentrantfähig und zum Großteil in-line codiert. Falls sie verwendet werden, ist intrins.h einzuschließen:

```
memcpy, memset, memchr, memmove, memcmp, strcmp, strcpy
_crol_, _irol_, _lrol_, _cror_, _iror_, _lror_  Rotate-Funktionen, Parameter
           (byteWert,anzahl)
_nop_      NOP-Befehl
_testbit_  JCB-Befehl, zum Beispiel  if (!_testbit_(Flag)) ...
```

4.2.2 Praktische Handhabung von C51

Die C51-Software in ihrer Gesamtheit ist sehr flexibel und deswegen nicht ganz einfach zu handhaben; sie ist im Sinne "klassischer Programmentwicklungssysteme" aufgebaut ist, das heißt, daß sie, anders als etwa TURBO-C, nicht von einem zentralen Regiebildschirm aus menügesteuert ist, sondern die Steuerung durch aufeinander folgende Programmaufrufe mit im Normalfall mehreren Kommandozeilenparametern erfolgt.

4.2.2.1 Standardfall

Um die Handhabung zu erleichtern, wird die Arbeit mit Batch-Dateien durchgeführt, die aber naturgemäß nur auf Standardfälle zugeschnitten sein können. Als wichtigster Standardfall wird das Erzeugen eines in den µProfi ladbaren .HEX-Files aus einem Quellfile angesehen. Dabei wird das Default-Compilermodell SMALL verwendet, für CODE wird A100h eingesetzt (zum Vermeiden eines Konflikts mit den ab A000h stehenden Interrupt-Einsprungadressen) und für XDATA A800h.

Diesem Zweck dient die Batch-Datei CLH51.BAT (Compile - Link - Hex):

```
@ECHO OFF
IF NOT EXIST %1.C GOTO NoSource
SET :WORK:=E:\
SET :INCLUDE:=C:\8051\C51\H
SET C51LIB=C:\8051\C51\LIB
IF "%2" == "C" GOTO :AsmCode
IF "%2" == "c" GOTO :AsmCode
C:\8051\C51\BIN\C51 %1.C OE DEBUG
GOTO QueryLst
:AsmCode
C:\8051\C51\BIN\C51 %1.C CODE OE DEBUG
:QueryLst
IF "%2" == "L" GOTO :ListC
IF "%2" == "C" GOTO :ListC
IF "%2" == "c" GOTO :ListC
IF NOT "%2" == "l" GOTO :NoListC
:ListC
C:\UT\LIST %1.LST
:NoListC
IF ERRORLEVEL 0 GOTO Link
GOTO Ende
```

```
:Link
C:\8051\C51\BIN\L51 %1.OBJ TO %1.ABS CODE (0A100H) XDATA (0A800H) SB DS
IF "%2" == "L" GOTO :ListL
IF "%2" == "C" GOTO :ListL
IF "%2" == "c" GOTO :ListL
IF NOT "%2" == "l" GOTO :NoListC
IF NOT "%2" == "l" GOTO :NoListL
:ListL
C:\UT\LIST %1.M51
:NoListL
IF "%2"=="L" GOTO PrfHex1
IF "%2" == "l" GOTO PrfHex1
IF "%2"=="C" GOTO PrfHex1
IF "%2" == "c" GOTO PrfHex1
GOTO PrfHex2
:PrfHex1
IF "%3" == "" GOTO Ende
GOTO MakeHex
:PrfHex2
IF "%2" == "H" GOTO MakeHex
IF NOT "%2" == "h" GOTO Ende
:MakeHex
C:\8051\C51\BIN\OHS51 %1.ABS
C:\8051\C51\BIN\HEXPAT %1 A0 00
COPY %1.HEY MODULE.HEY
:Ende
SET :INCLUDE:=
SET C51LIB=
SET :WORK:=
GOTO Schluss
:NoSource
ECHO Quelldatei %1.C fehlt.
:Schluss
@ECHO ON
```

Mit dieser Batch-Datei werden zunächst die für die Software nötigen Environment-Variablen gesetzt (und am Ende wieder gelöscht). Sodann werden nach Ausführen des Compilierens die Parameter 2 und 3 geprüft:

- Wenn Parameter 2 ein L ist, werden nacheinander das Compiler-Listing (progname.LST) und das Linker-Listing (progname.M51) auf dem Bildschirm dargestellt. Wenn der Parameter ein C ist, erfolgt das Gleiche, jedoch wird noch der vom Compiler erzeugte Assembler-Quellcode in das Compiler-Listing aufgenommen.
- Wenn Parameter 2 ein H ist, werden keine Listings dargestellt, jedoch wird ein .HEX-File erzeugt.
- Wenn Parameter 2 ein L oder ein C und Parameter 3 ein H ist, werden Listings dargestellt und wird ein .HEX-File erzeugt.
- Parameter 1 muß der Name des C-Quellmoduls progname (ohne Suffix .C) sein.

Für das spätere Testen mit DS51 genügt ein Aufruf in folgender Form:

CLH51 *progname* , allenfalls noch mit L oder C.

Für das Testen auf der Zielhardware mit TS51 ist jedoch der Parameter H unumgänglich:

CLH51 *progname* H oder CLH51 *progname* L|C H (L oder C !)

H bewirkt sowohl Erzeugen eines .HEX-File als auch nachfolgendes Patchen dieses .HEX-Files, wobei progname.HEY entsteht. In letzterem sind alle Ladeadressen im .HEX-File, die mit 00 beginnen, auf A0 geändert, wodurch erst ein fehlerfreies Laden in den µProfi möglich wird.

In Ergänzung zum Abschnitt 5, der sich im wesentlichen auf Assemblerprogramme bezieht, ist auf eine etwas geänderte Vorgangsweise hinzuweisen:

TS51 ist mit nachfolgendem TS51.BAT aufzurufen. Darin wird das zu testende Modul eigentlich zweimal geladen: das erste Mal mit allen Symbol- und Debug-Informationen, die im .ABS-File enthalten sind, und danach entsprechend der .INI-Datei mit den gepatchten Ladeadressen:

```
I:
IF NOT EXIST %1.C GOTO NoC
COPY %1.C MODULE.C >NUL
:NoC
COPY %1.ABS MODULE.ABS >NUL
C:\8051\DSCOPE\TS51 MODULE.ABS INIT(MODULE.INI)
@ECHO ON
```

Die gegenüber Abschnitt 5 etwas abgeänderte .INI-Datei hat folgenden Inhalt:

```
/***************************************/
/* Initialisation File for tScope-51  */
/***************************************/
\o\m                          /* Options Medium (Lines) */
\v\s                          /* View Serial (Off)      */
\d\d\d\d\d\d\d\d\d\d\d\d\d    /* Decrease Exe-Window    */
LOAD C:\8051\DSCOPE\MON51.IOT  BAUDRATE (9600) CPUTYPE (8051) COM1:
LOAD I:\MODULE.HEY
$=0A100h
```

Das vorstehend geschilderte Verfahren ist ein "Work around", der wegen gewisser Unzukömmlichkeiten der von uns verwendeten Compilerversion erforderlich ist.

Als Beispiel wurde eine C51-Version von ROTIER.ASM (siehe 4.1.3.4) hergestellt:

```
#include <reg51.h>    /* Rotier.C */
const char KettLaeng= 5;
unsigned char Kett5Bytes[] = { 0x12, 0x34, 0x56, 0x78, 0x90 };
```

```
void Rotate(unsigned char *KettPtr)
{ char Zaehler;
  unsigned char ZwSpeicher;
  ZwSpeicher = *KettPtr;
  for (Zaehler=0; Zaehler < (KettLaeng-1); Zaehler++)
  { *(KettPtr+Zaehler) = *(KettPtr+Zaehler) << 4;
    *(KettPtr+Zaehler) = *(KettPtr+Zaehler) | *(KettPtr+Zaehler+1) >> 4;
  }
  *(KettPtr+KettLaeng-1) = (*(KettPtr+KettLaeng-1) << 4) | (ZwSpeicher >> 4);
}

void main()
{ for (;;)
      Rotate(Kett5Bytes);
}
```

Dieses Programm wurde in TURBO-C entwickelt und anschließend mittels

```
CLH51.BAT rotier l h
```

vom Compiler C51 mit Standardeinstellungen (Default-Einstellungen) übersetzt sowie mit Standardeinstellungen gelinkt.

Anmerkung zum folgenden: Mit anderen Versionen des Compilers bzw. Linkers können sich insbesondere Adressen etwas ändern. Die folgende Darstellung beruht auf der Verwendung von C51 V3.20 und L51 V2.8.

Aus der Linker-Mapliste Rotier.M51 erfährt man unter anderem, daß das Speichermodell (MEMORY MODEL) SMALL verwendet wurde und weiters, daß außer Rotier.OBJ noch mehrere Moduln aus der Bibliothek C51S.LIB eingebunden wurden:

```
INPUT MODULES INCLUDED:
  ROTIER.OBJ (ROTIER)
  C:\8051\C51\LIB\C51S.LIB (?C_STARTUP)
  C:\8051\C51\LIB\C51S.LIB (?C_CLDPTR)
  C:\8051\C51\LIB\C51S.LIB (?C_CLDOPTR)
  C:\8051\C51\LIB\C51S.LIB (?C_CSTPTR)
  C:\8051\C51\LIB\C51S.LIB (?C_CSTOPTR)
  C:\8051\C51\LIB\C51S.LIB (?C_INIT)
```

Es ist zu erkennen, daß die aus Rotier.C erzeugte .HEX-Datei mit 1384 Bytes wesentlich größer ist als die aus Rotier.ASM erzeugte. Der Grund liegt in dem in einer höheren Programmiersprache notwendigen "Overhead", das ist Code, der nicht unmittelbar der eigentlichen Aufgabe des Programms dient, sondern eher dem Verwalten des Systems und seiner Ressourcen. So wird zum Beispiel am Anfang mit der folgenden Routine der ganze interne Datenbereich gelöscht:

```
A0B9H        MOV       R0,#7FH
A0BBH        CLR       A
A0BCH        MOV       @R0,A
A0BDH        DJNZ      R0,0A0BCH
```

Allerdings wird dieses Verhältnis mit zunehmender Größe einer Programmieraufgabe zugunsten der höheren Programmiersprache immer günstiger.

Sehr wichtig ist die Aufstellung der Speicherbereiche für das Hauptprogramm Rotier.ABS. Hier erfährt man insbesondere, daß im Datenspeicher DATA auf Adresse 8h ein 6 Bytes langes Datensegment angelegt wurde, das offensichtlich die static-Variablen KettLaeng und Kett5Bytes[5] enthält:

```
LINK MAP OF MODULE:  ROTIER.ABS (ROTIER)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0006H     UNIT         ?DT?ROTIER
            DATA    000EH     0005H     UNIT         "DATA_GROUP"
            IDATA   0013H     0001H     UNIT         ?STACK
```

Tatsächlich werden ab DATA-Adresse 8h nach etlichen Programmschritten ("Overhead") die zu rotierenden Daten mit einer kleinen Routine zwischen A0CCh und A0D6h von der Codespeicheradresse 0A0B3h bis 0A0B7h geholt und zwischen 9h und 0Dh abgelegt (davor steht der Wert von KettLaeng):

```
A0CCH        CLR       A
A0CDH        MOVC      A,@A+DPTR
A0CEH        INC       DPTR
A0CFH        JC        0A0D4H
A0D1H        MOV       @R0,A
A0D2H        SJMP      0A0D5H
A0D4H        MOVX      @R0,A
A0D5H        INC       R0
A0D6H        DJNZ      R7,0A0CCH
```

```
>d d:8,0fh         DATA-Speicher vor Eintritt in die Routine:
D:0008                         -00 00 00 00 00 00 00 00
>d c:0a0b3,0a0b7   Code-Speicher mit initialisiertem Array:
C:A0B8                         -12 34 56 78 90

>d d:8,0fh         DATA-Speicher nach ausgeführter Routine:
D:0008                         -05 23 45 67 89 01 04 00
```

Diese Routine ist ein Teil der aus der Bibliothek stammenden, hinzugelinkten Initialisierungsroutine C_C51STARTUP, wie man aus der Link-Map des Codespeichers sehen kann:

```
* * * * * * *   C O D E   M E M O R Y   * * * * * * *
CODE    0000H    0003H    ABSOLUTE
        0003H    9FFDH                  *** GAP ***
CODE    A100H    00A2H    UNIT          ?PR?_ROTATE?ROTIER
CODE    A1A2H    000CH    UNIT          ?PR?MAIN?ROTIER
CODE    A1AEH    000BH    UNIT          ?C_INITSEG
CODE    A1B9H    008CH    UNIT          ?C_C51STARTUP
CODE    A245H    0082H    UNIT          ?C_LIB_CODE
```

Nach diesen Initialisierungsvorgängen wird ein Sprung an den Beginn des eigentlichen Hauptprogramms main() auf Adresse 0A1A2h ausgeführt. Vor und nach einem kompletten Durchlauf sieht die Bytekette samt vorangestelltem Längenbyte so aus:

```
>d d:8,0f        vor einem Rotiervorgang:
D:0008                        -05 12 34 56 78 90 00 00
>p
>d d:8,0f        nach einem Rotiervorgang:
D:0008                        -05 23 45 67 89 01 04 00
```

Im konkreten Fall stellte sich heraus, daß der von C51 erzeugte Code im Fall der folgenden Variante fehlerhaft arbeitet; diese ist dadurch gekennzeichnet daß die Bytes im Unterprogramm nicht mittels Zeigerarithetik adressiert werden, sondern mittels Index in einem Array:

```
void Rotate(unsigned char *ByteArray)
{ char Zaehler;
  unsigned char ZwSpeicher;
  ZwSpeicher = ByteArray[0];
  for (Zaehler=0; Zaehler < (KettLaeng-1); Zaehler++)
  { ByteArray[Zaehler] = ByteArray[Zaehler] << 4;
    ByteArray[Zaehler] = ByteArray[Zaehler] | (ByteArray[Zaehler+1] >> 4);
  }
  ByteArray[KettLaeng-1] = (ByteArray[KettLaeng-1] << 4) | (ZwSpeicher >> 4);
}
```

Es wird zwar in jedem Byte das rechte Halbbyte nach links gestellt, nicht jedoch das linke ins rechte Halbbyte des davor stehenden Bytes; statt dessen wird jedes rechte Halbbyte auf 2 gestellt:

```
>d d:8,0f
D:0008                        -22 42 62 78 90 05 04 00
```

(nach 3 Durchläufen der Hauptschleife)

Man erkennt, daß Software zwar in der Regel sehr verläßlich ist, in einzelnen Fällen aber doch fehlerhaft sein kann; zum Glück gibt es oft andere Möglichkeiten im Quellcode, mit denen man solche Fehler umgehen kann.

4.2.2.2 Systematik beim Testen eines C-Programms (Beispiel)

Zunächst muß das zu testende Programm geladen werden, zum Beispiel mit

```
>LOAD I:ROTIER.ABS
```

, soweit dies nicht von DS51.BAT erledigt wurde.

Da das zugehörige .OBJ-File auch den Namen der Quelldatei (im Beispiel ROTIER.C) enthält, wird diese automatisch mit in DS51 übernommen. Nach Verkleinern des Exe-Fensters sieht der Bildschirm so aus:

```
 Options  Key  View  Peripheral  Map  Help  Cpu                  Trace  Register
0000H       LJMP      00BCH                                             A = 00
0003H .ROTIER._ROTATE:                                                  B = 00
0003H #5:   MOV       KettPtr(0EH),R3                                  R0 = 00
5:    void Rotate(unsigned char *KettPtr)                              R1 = 00
6:    { char Zaehler;                                                  R2 = 00
7:      unsigned char ZwSpeicher;                                      R3 = 00
8:      ZwSpeicher = *KettPtr;                                         R4 = 00
9:      for (Zaehler=0; Zaehler < (KettLaeng-1); Zaehler++)            R5 = 00
10:     { *(KettPtr+Zaehler) = *(KettPtr+Zaehler) << 4;                R6 = 00
11:       *(KettPtr+Zaehler) = *(KettPtr+Zaehler) | *(KettPtr+Zaehle   R7 = 00
12:     }                                                             DPTR= 0000
13:     *(KettPtr+KettLaeng-1) = (*(KettPtr+KettLaeng-1) << 4) | (Zw  PC $= 0000
14:   }
15:                                                                     PSW
16:   void main()                                                     ---R0---
17:   { for (;;)                                                      Cycles cyc
18:         Rotate(Kett5Bytes);                                                0

                                                                      SP=07: 00
 Language  Module: ROTIER.C                                            I:06: 00
>load rotier.abs                                                       I:05: 00
>                                                                      I:04: 00
=Exe=================================================================  I:03: 00
                                                                       I:02: 00
```

Es kommt nun vorzugsweise in Frage, das Programm bis zum Eintritt in die Hauptfunktion main() laufen zu lassen:

```
>G,main
```

(G ... Lauf ab aktueller Adresse;
,breakpoint ... temporärer Haltepunkt)

Der blaue Balkencursor markiert die aktuelle Anweisung und steht jetzt auf der Zeile 18.

Zum Beobachten der Arbeitsweise des Programms sollte man das Watch-Fenster öffnen:

```
ALT-V(iew) - D(efine watch) - Kett5Bytes,16
```

oder kurz

```
WS Kett5Bytes,16
```

läßt das Watch-Fenster oben erscheinen und stellt die gewünschte Variable hexadezimal dar (mit ALT-V - auto Update wird automatisches Mitführen der Anzeige im Watch-Fenster einegstellt):

```
░Options░░Key░░View░░Peripheral░░Map░░Help░░Cpu░
00: Kett5Bytes,16: {0x23,0x45,0x67,0x89,0x01}
░Watch ░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░
0003H #5:    MOV       KettPtr(0EH),R3
```

Die Variable ZwSpeicher kann erst nach Eintritt in die Funktion Rotate beobachtet werden. Nach einigen T-Eingaben (Trace-Step) steht der aktuelle Cursor auf Zeile 9 und man könnte ZwSpeicher anzeigen lassen:

```
>WS ZwSpeicher
```

```
░Options░░Key░░View░░Peripheral░░Map░░Help░░Cpu░
00: Kett5Bytes,16: {0x23,0x45,0x67,0x89,0x01}
01: ZwSpeicher: 0x23
░Watch ░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░
5:    void Rotate(unsigned char *KettPtr)
```

Die Watch-Zeilen sind ganz links mit Nummern versehen, mit denen sie gezielt ansprechbar sind.

Nun bietet es sich an, auf jedem Aufruf von Rotate den Programmlauf zu unterbrechen, also auf Zeile 18 einen Haltepunkt zu setzen:

```
>BS Rotate
```

Die Wirksamkeit kann durch Ausgeben einer Breakpoint-Liste überprüft werden:

```
>BL
 0: (E C:0x3) 'Rotate', CNT=1, enabled
```

Jetzt kann man GO-Befehle riskieren, sofern man sicher ist, daß gesetzte Haltepunkte tatsächlich erreicht werden. Nach einigen GO-Befehlen sieht das Watch-Fenster so aus, wobei der Programmlauf immer auf Zeile 5 angehalten wird:

```
░Options░░Key░░View░░Peripheral░░Map░░Help░░Cpu░
00: Kett5Bytes,16: {0x45,0x67,0x89,0x01,0x23}
01: ZwSpeicher: 0x34
░Watch ░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░
5:    void Rotate(unsigned char *KettPtr)
```

4.2.2.3 Vom Standardfall abweichende Verarbeitung

Im Fall höherer Ansprüche (zum Beispiel, wenn mehrere Objektmoduln übersetzt und gelinkt werden sollen) empfiehlt es sich, die Schritte "von Hand aus" auszuführen oder sich eine eigene Batch-Datei für ein bestimmtes Projekt zu schreiben. Dazu soll die folgende Kurzanleitung dienen:

Der Aufruf des Compilers C51 ist mit

```
C51 progname.C
```

sowie insbesondere folgenden Kommandozeilenparametern (getrennt durch Blanks) möglich:

```
CODE           In die Datei progname.LST wird auch der Assemblercode eingefügt
OBJECTEXTEND   Informationen für den Debugger (abkürzbar mit OE)
SMALL          Kleinstes Speichermodell: alle Daten im internen RAM
  alternativ:  COMPACT: Parameter und lokale Variablen im "paged" XDATA LARGE:
               Parameter und lokale Variablen in XDATA
```

Auf das Setzen der Environment-Variablen (siehe CLH51.BAT) darf nicht vergessen werden!

Der Aufruf des Linkers L51 ist mit

```
L51 mainName.OBJ [,uproNam1.OBJ ...] TO mainName.ABS
```

sowie insbesondere folgenden Kommandozeilenparametern (getrennt durch Blanks) möglich:

```
CODE (adresse)  Transferadresse im Codespeicher (Segmentadresse von CODE)
XDATA (adresse) Adresse des externen Datenspeichers (Segmentadresse von XDATA)
SYMBOLS         Lokale Symbole werden in mainName.LST übernommen (kurz: SB)
DEBUGSYMBOLS    Lokale Symbole werden in mainName.ABS übernommen (kurz: DS)
```

Das Hauptprogramm main() muß im ersten der angegebenen Objektmodule stehen.

Beachten: Die Liste der Objektmodule ist durch Kommas getrennt, die Aufzählung der Parameter durch Blanks!

BEISPIEL: Das vorstehende Beispiel Rotier.C wird in zwei Module Rot_Main.C und Rot_Sub.C aufgeteilt (dabei wird auch anstelle der Globalvariablen KettLaeng ein weiterer Parameter übergeben):

```
#include <reg51.h>    /* Rotier.C */
unsigned char Kett5Bytes[] = { 0x12, 0x34, 0x56, 0x78, 0x90 };

void Rotate(unsigned char *KettPtr, char KettLaeng);

void main()
{ for (;;)
      Rotate(Kett5Bytes,5);
} /* Ende Rot_Main */
```

```
void Rotate(unsigned char *KettPtr, char KettLaeng)
{ char Zaehler;
  unsigned char ZwSpeicher;
  ZwSpeicher = *KettPtr;
  for (Zaehler=0; Zaehler < (KettLaeng-1); Zaehler++)
  { *(KettPtr+Zaehler) = *(KettPtr+Zaehler) << 4;
    *(KettPtr+Zaehler) = *(KettPtr+Zaehler) | *(KettPtr+Zaehler+1) >> 4;
  }
  *(KettPtr+KettLaeng-1) = (*(KettPtr+KettLaeng-1) << 4) | (ZwSpeicher >> 4);
} /* Ende Rot_Sub */
```

Die folgenden Schritte erzeugen funktionierende .HEX- und .ABS-Files:

```
I:\>C:\8051\C51\BIN\C51 ROT_MAIN.C SMALL
I:\>C:\8051\C51\BIN\C51 ROT_SUB.C SMALL
I:\>C:\8051\C51\BIN\L51 ROT_MAIN.OBJ,ROT_SUB.OBJ to ROT_MOD.ABS CODE (0A100h)
    DATA(0A800h) SB DS
I:\>C:\8051\C51\BIN\OHS51 ROT_MOD.ABS
I:\>
```

4.2.2.4 Verbinden von Assembler- und C-Moduln

In erster Linie bietet sich das Einbinden von Assembler- Moduln in C-Programme an; aber auch das Umgekehrte ist möglich, etwa, wenn in einem Assembler-programm float-Rechnungen auszuführen wären.

Das Einhalten der Konventionen für das Übergeben von Parametern und Funktions-ergebnissen bedingt, daß man mit nicht zu vielen Parametern auskommt; notfalls bietet sich eine verbesserte Strukturierung der Moduln an.

(1) Parameterübergabe

Für die Parameterübergabe werden die Register R2 bis R7 verwendet. Mit ihnen können übergeben werden:

- Bis zu 3 Parameter von Byte-Größe (Typ char oder 1-Byte-Pointer), wobei in der Reihenfolge der Parameter R7, R5 und R3 verwendet werden.

- Bis zu 3 Parameter von Wort-Größe (Typ int oder 2-Byte-Pointer), wobei in der Reihenfolge der Parameter die Registerpaare R6/R7, R4/R5 und R2/R3 verwendet werden.

- Nur 1 Parameter von Doppelwort-Größe (Typ long oder float), wobei R4 bis R7 verwendet werden.

Generische Pointer sind Zeiger, die auf jedes Objekt zeigen können (unabhängig von Speichermodell und Lage in irgendeinem Speicherraum). Sie belegen drei Bytes und werden von Compiler und Linker gehandhabt. Übergeben werden sie in R1 bis R3. Im Speicher enthält das vorderste Byte den Speichertyp, die beiden folgenden die 16-Bit-Adresse; die Registerbelegung ist gerade umgekehrt (von R3 abwärts).

Parameter, die nicht mehr in Registern untergebracht werden können, werden in Übergabesegmenten übergeben. Die Adresse eines solchen Segments ist im Assemblerprogramm als Externsymbol mit dem Namen ?funktionsName?BYTE zu vereinbaren und außerdem PUBLIC zu deklarieren. Falls Bit-Werte übergeben werden, tritt anstelle von BYTE BIT, also ?funktionsName?BIT (siehe auch weiter unten, (3)).

Die Übergabesegmente müssen je nach Speichermodell verschieden plaziert werden (siehe auch weiter unten, (3)):

- Im Modell SMALL im DATA-Bereich,
- im Modell COMPACT im XDATA-Bereich; wegen kurzer Adressierung mit MOVX @R0 bzw. @R1 jedoch mit dem Attribut IPAGE.
- im Modell LARGE im XDATA-Bereich; dabei wird lange Adressierung mit MOVX @DPTR verwendet.

(2) Ergebnisübergabe

Ergebnisse von Funktionen werden gemäß folgender Übersicht übergeben:

Länge	Übergabe in
1 Bit	Carry-Flag
1 Byte	R7
1 Wort	R6/R7
1 DWort	R4 bis R7
Pointer	R1 bis R3

(3) Weitere Konventionen

Alle Register der momentan aktiven Registerbank sowie A, B, DPTR und PSW dürfen in Assembler-Unterprogrammen verändert werden; ebenso muß beim Aufruf eines C-Unterprogramms damit gerechnet werden, daß diese Register verändert werden.

Die Segmente für Parameter-Übergabevariablen, lokale Variablen oder Bitvariablen haben standardisierte Namen, welche auch in Assemblerprogrammen zu verwenden sind. Die Form dieser Namen ist folgende:

?xx?FunktionsName?ModulName

In allen Speichermodellen gilt:
das CODE-Segment (für Programmcode) ist xx PR
das BIT-Segment (für lokale Bitvariablen) ist xx BI

Der Name des Segments für lokale Variablen hängt vom Speichermodell ab:

Modell	xx	
SMALL	DT	(für DATA)
COMPACT	PD	(für PDATA)
LARGE	XD	(für XDATA)

Diese Segmente erhalten vom Compiler das Attribut OVERLAYABLE.

(4) Beispiel eines Assembler-Unterprogramms, das aus einem C-Programm aufgerufen wird:

Das Unterprogramm hat die Aufgabe, zwei int-Werte zu übernehmen, zu addieren und das Ergebnis zurückzugeben.

```
NAME      Addiere
?PR?IntAdd?Addiere   SEGMENT CODE
PUBLIC    _IntAdd             ; Einsprungadresse
RSEG      ?PR?IntAdd?Addiere
_IntAdd:  MOV       A,R7      ; erster Parameter, low Byte
          ADD       A,R5      ; zweiter Parameter, low Byte
          MOV       R7,A      ; Ergebnis, low Byte
          MOV       A,R6      ; erster Parameter, high Byte
          ADDC      A,R4      ; zweiter Parameter, high Byte
          MOV       R6,A      ; Ergebnis, high Byte
          RET                 ; Ergebnis übergeben
          END ;
```

Das aufrufende C-Programm lautet wie folgt:

```
/* Modular1.C: Aufrufen eines Assembler-Unterprogramms, Parameter-
               übergabe und Ergebnisübergabe */
/* #include <reg51.h> */

int IntAdd(int Zahl1, int Zahl2);

void main()
{ int iA=4000, iB=6000, iC;
  iC=IntAdd(iA, iB);
}
```

Die Moduln werden wie folgt mit Batch-Dateien übersetzt:

ASM.BAT zum Übersetzen eines Assembler-Moduls

```
@ECHO OFF
C:\8051\ASM51\ASM51 I:%1.ASM db ep(%1.err)
PAUSE
C:\UT\LIST I:%1.LST
@ECHO ON
```

CASM.BAT zum Übersetzen eines C-Moduls samt Ausgabe des Assembler-Codes in der .LST-Datei:

```
SET :WORK:=E:\
SET :INCLUDE:=C:\8051\C51\H
SET C51LIB=C:\8051\C51\LIB
C:\8051\C51\BIN\C51 %1.C CODE OE DEBUG
LIST %1.LST
SET :INCLUDE:=
SET C51LIB=
SET :WORK:=
```

Die Ergebnisse dieser Arbeitsgänge sind folgende .LST-Dateien:

```
MCS-51 MACRO ASSEMBLER    ADDIERE                                          09/28/98   PAGE  1



DOS 20.40 (033-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN I:INTADD.OBJ
ASSEMBLER INVOKED BY:  C:\8051\ASM51\ASM51.EXE I:INTADD.ASM DB EP(INTADD.ERR)

LOC  OBJ            LINE     SOURCE

                       1     NAME     Addiere
                       2     ?PR?IntAdd?Addiere   SEGMENT CODE
                       3     PUBLIC   _IntAdd               ; Einsprungadresse
----                   4     RSEG                           ?PR?IntAdd?Addiere
0000 EF                5 _IntAdd: MOV    A,R7               ; erster Parameter, low Byte
0001 2D                6                  ADD       A,R5     ; zweiter Parameter, low Byte
0002 FF                7                  MOV       R7,A     ; Ergebnis, low Byte
0003 EE                8                  MOV       A,R6     ; erster Parameter, high Byte
0004 3C                9                  ADDC      A,R4     ; zweiter Parameter, high Byte
0005 FE               10                  MOV       R6,A     ; Ergebnis, high Byte
0006 22               11                  RET                ; Ergebnis übergeben
                      12                  END ;
MCS-51 MACRO ASSEMBLER    ADDIERE                                          09/28/98   PAGE    2


SYMBOL TABLE LISTING
------ ----- -------


N A M E                      T Y P E   V A L U E      A T T R I B U T E S

_INTADD. . . . . .  C ADDR    0000H   R PUB   SEG=?PR?INTADD?ADDIERE
?PR?INTADD?ADDIERE  C SEG     0007H           REL=UNIT
ADDIERE. . . . . .    ----    ----


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

```
C51 COMPILER V3.20,  MODULAR1                                  28/09/98  15:54:29  PAGE 1


DOS C51 COMPILER V3.20, COMPILATION OF MODULE MODULAR1
OBJECT MODULE PLACED IN MODULAR1.OBJ
COMPILER INVOKED BY: C:\8051\C51\BIN\C51.EXE MODULAR1.C CODE OE DEBUG

stmt level    source

/* Modular1.C: Aufrufen eines Assembler-Unterprogramms, Parameter-
            übergabe und Ergebnisübergabe */
 3          /* #include <reg51.h> */
 4
 5          int IntAdd(int Zahl1, int Zahl2);
 6
 7          void main()
 8          { int iA=4000, iB=6000, iC;
 9   1        iC=IntAdd(iA, iB);
10   1      }
11
C51 COMPILER V3.20,  MODULAR1                                  28/09/98  15:54:29  PAGE 2

ASSEMBLY LISTING OF GENERATED OBJECT CODE


          ; FUNCTION main (BEGIN)
                                              ; SOURCE LINE # 7
                                              ; SOURCE LINE # 8
;---- Variable 'iA' assigned to Register 'R6/R7' ----
0000 7FA0                    MOV         R7,#0A0H
0002 7E0F                    MOV         R6,#0FH
;---- Variable 'iB' assigned to Register 'R4/R5' ----
0004 7D70                    MOV         R5,#070H
0006 7C17                    MOV         R4,#017H
                                              ; SOURCE LINE # 9
0008 120000  E               LCALL       _IntAdd
000B 8E00    R               MOV         iC,R6
000D 8F00    R               MOV         iC+01H,R7
                                              ; SOURCE LINE # 10
000F 22                      RET
            ; FUNCTION main (END)

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE                   =     16    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE           =   ----    ----
   PDATA SIZE           =   ----    ----
  DATA SIZE             =   ----       2
   IDATA SIZE           =   ----    ----
   BIT SIZE             =   ----    ----
END OF MODULE INFORMATION.


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Für das Linken ist es zweckmäßig, ebenfalls eine Batch-Datei zu schreiben, zum Beispiel für die gegenwärtige Konfiguration (ein C-Hauptprogramm und ein Assembler-Unterprogramm):

LINK2.BAT

```
SET :WORK:=E:\
SET :INCLUDE:=C:\8051\C51\H
SET C51LIB=C:\8051\C51\LIB
C:\8051\C51\BIN\l51 %1.OBJ,%2.OBJ to %1.ABS CODE (0A100h) XDATA(0A800h) SB DS
SET :INCLUDE:=
SET C51LIB=
SET :WORK:=
```

Der Linker liefert folgende MAP-Datei:

```
MCS-51 LINKER / LOCATER  L51 V2.8                    DATE  28/09/98   PAGE    1


  MS-DOS MCS-51 LINKER / LOCATER  L51 V2.8, INVOKED BY:
  L51 MODULAR1.OBJ, INTADD.OBJ TO MODULAR1.ABS CODE (0A100H) XDATA (0A800H) SB DS

  MEMORY MODEL: SMALL


  INPUT MODULES INCLUDED:
    MODULAR1.OBJ (MODULAR1)
    INTADD.OBJ (ADDIERE)
    C:\8051\C51\LIB\C51S.LIB (?C_STARTUP)


  LINK MAP OF MODULE:  MODULAR1.ABS (MODULAR1)


                   TYPE        BASE          LENGTH        RELOCATION         SEGMENT NAME
                   -----------------------------------------------------

                   * * * * * * *   D A T A   M E M O R Y   * * * * * * *
                   REG         0000H         0008H         ABSOLUTE           "REG BANK 0"
                   DATA        0008H         0002H         UNIT               "DATA_GROUP"
                   IDATA       000AH         0001H         UNIT               ?STACK

                   * * * * * * *   C O D E   M E M O R Y   * * * * * * *
                   CODE        0000H         0003H         ABSOLUTE
                               0003H         A0FDH                            *** GAP ***
                   CODE        A100H         0010H         UNIT               ?PR?MAIN?MODULAR1
                   CODE        A110H         0007H         UNIT               ?PR?INTADD?ADDIERE
                   CODE        A117H         000CH         UNIT               ?C_C51STARTUP


  OVERLAY MAP OF MODULE:   MODULAR1.ABS (MODULAR1)


  SEGMENT                         DATA-GROUP
    +--> CALLED SEGMENT         START    LENGTH
  ----------------------------------------------
  ?C_C51STARTUP                 -----    -----
    +--> ?PR?MAIN?MODULAR1

  ?PR?MAIN?MODULAR1             0008H    0002H
    +--> ?PR?INTADD?ADDIERE



  SYMBOL TABLE OF MODULE:  MODULAR1.ABS (MODULAR1)

  VALUE                    TYPE              NAME
  ---------------------------------

  -------                  MODULE            MODULAR1
  C:0000H                  SYMBOL            _ICE_DUMMY_
  C:A100H                  PUBLIC            main
  -------                  PROC              MAIN
  C:A100H                  LINE#             7
  C:A100H                  LINE#             8
  -------                  DO                (NULL)
  D:0006H                  SYMBOL            iA            <--- Variablen
  D:0004H                  SYMBOL            iB            <---
  D:0008H                  SYMBOL            iC            <---
  C:A108H                  LINE#             9
  C:A10FH                  LINE#             10
  -------                  ENDDO             (NULL)
  -------                  ENDPROC           MAIN

  MCS-51 LINKER / LOCATER  L51 V2.8                       DATE  28/09/98   PAGE    2

  -------                  ENDMOD            MODULAR1

  -------                  MODULE            ADDIERE
```

```
   C:A110H            PUBLIC          _INTADD
   C:A110H            SEGMENT         ?PR?INTADD?ADDIERE
   -------            ENDMOD          ADDIERE
```

Ein Testlauf mittels DS51 läuft, verkürzt dargestellt, so ab:

Aufruf der Batchdatei DS51 Modular1 .

View auf Highlevel (Hochsprach-Darstellung)

```
░Options░░Key░░View░░Peripheral░░Map░░Help░░Cpu░░░
3:     /* #include <reg51.h> */
4:
5:     int IntAdd(int Zahl1, int Zahl2);
6:
7:     void main()
8:     { int iA=4000, iB=6000, iC;
9:       iC=IntAdd(iA, iB);
10:    }
11:

░Language░░Module:░MODULAR1.C░░░░░░░░░░░░░
```

Im unteren Bereich wird die ausgeführte .INI-Datei angezeigt:

```
dScope-51+ V5.10
Copyright KEIL ELEKTRONIK GmbH 1990, 1991
>/* Initialisation File for dScope-51+ */
>\o\m                          /* Option Medium (lines)  *\
 \o\m                          /* Option Medium (lines)  *\
>\d\d\d\d\d\d\d\d\d\d\d\d\d\d  /* Increase Exe-Window    */
>\v\s                          /* View Serial (Off)      */
>load C:\8051\C51\BIN\8051.IOF /* IOF driver for 8051    */
8051/8031 80C51/80C31 PERIPHERALS for dScope-51+  V1.1
(C) Franklin Software Inc./KEIL ELEKTRONIK GmbH 1991
>map 0xA800,0xB000             /* XDATA memory 2 kB      */
>map 0xc000,0xc001             /* XDATA µProfi IO        */
>load MODULE.ABS               /* Load Test Module       */
>$=0xA100
═Exe══░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░
 !dos   ASM   ASSIGN   BreakDisable   BreakEnable   BreakKill
```

Nach Ausführen von Zeile 8 kann überprüft werden, ob die vom C-Programm in die Variablen gesetzten Werte stimmen. iB liegt auf Adresse 4, iA auf Adresse 6, wie man auch aus der Linker-Map erkennt. iC auf Adresse 8 ist noch "leer".

```
>d iB
D:0004              17 70 0F A0-00 00 00 00 00 00 00 00  .p......
```

Durch das C-Programm geht man am besten im Modus View = Highlevel mit dem Trace-Befehl ALT-T. Sobald jedoch die Zeile mit dem Unterprogramm-Aufruf erreicht ist, bietet es sich an, auf View - Assembly umzuschalten:

```
A108H #9:    LCALL     _INTADD(0A110H)
A10BH        MOV       iC(08H),R6
A10DH        MOV       iC+1(09H),R7
A10FH #10:   RET
A110H .ADDIERE:_INTADD:
A110H        MOV       A,R7
A111H        ADD       A,R5
A112H        MOV       R7,A
A113H        MOV       A,R6
A114H        ADDC      A,R4
A115H        MOV       R6,A
A116H        RET
```

Aufruf des Unterprogramms

Hier beginnt das Assembler-Unterprogramm.

```
Register
 A = 27
 B = 00
R0 = 00
R1 = 00
R2 = 00
R3 = 00
R4 = 17
R5 = 70
R6 = 27
R7 = 10
```

Nach Erreichen (aber vor Ausführen) des RET-Befehls sind die Registerinhalte wie nebenstehend:

Zur Erklärung: 4000 = FA0h, 6000 = 1770h, das Ergebnis 10000 = 2710h .

Nach Ausführen des RET-Befehls steht dieses Ergebnis auf Adresse 8:

```
>d iC
D:0008                    -27 10 00 00 00 00 00 00          '...
```

(5) Beispiel eines C-Unterprogramms, das aus einem Assembler-Hauptprogramm aufgerufen wird:

Das Unterprogramm hat genauso wie das vorstehende die Aufgabe, zwei int-Werte zu übernehmen, zu addieren und das Ergebnis zurückzugeben.

Diese Aufgabe ist etwas einfacher. Das Assembler-Hauptprogramm Modular2.ASM lautet wie folgt:

```
NAME     Modular2
         EXTRN CODE (_IntAdd)
Modular2 SEGMENT CODE
         RSEG    Modular2
         MOV   R7,#0A0h
         MOV   R6,#0Fh          ; FA0h = 4000
         MOV   R5,#70h
         MOV   R4,#17h          ; 1770h = 6000
         LCALL _IntAdd
```

```
          END;
```

Das C-Unterprogramm IntAdd.C lautet wie folgt:

```
/* IntAdd.C: Unterprogramm zum Addieren zweier int-Zahlen */
/* #include <reg51.h> */

int IntAdd(int iA, int iB)
{ return iA+iB;
}
```

Das Linken und das Testen im DS51 erfolgen analog wie unter (4).

```
▒Options▒▒Key▒▒View▒▒Peripheral▒▒Map▒▒Help▒▒Cpu▒▒
A100H        MOV       R7,#0A0H
A102H        MOV       R6,#0FH
A104H        MOV       R5,#70H
A106H        MOV       R4,#17H
A108H        LCALL     _IntAdd(0A10BH)
A10BH .INTADD._INTADD:
A10BH #4:    MOV       A,R7
4:     int IntAdd(int iA, int iB)
5:     { int iC;
6:       iC=iA+iB;
7:       return iC;
8:     }
9:
```

```
Register
 A = 27
 B = 00
R0 = 00
R1 = 00
R2 = 00
R3 = 00
R4 = 17
R5 = 70    <-
R6 = 27    <-
R7 = 10
```

Nach Ausführung steht 2710h in R6/R7

4.2.2.5 Timer- und andere Interruptroutinen

C51 erlaubt es, Interrupt-Routinen direkt in C zu schreiben (für das Initialisieren des Timers ist es andererseits zweckmäßig, den erforderlichen Assembler-Code durch direktes Ansprechen der Register nachzuvollziehen). Dabei kann es aber auch kleine Komplikationen geben, zum Beispiel in der folgenden C51-Fassung des Programms aus 4.1.3.2:

```
#include <reg51.h>      /* Timer0.C */

void SetTmr(void);      /* Timer 0 initialisieren */
#ifdef __TURBOC__
void interrupt Tim0Int(void)
#endif
#ifdef __C51__
void Tim0Int(void) interrupt 1
```

```
#endif
{ ACC++;
  SetTmr();
}
void SetTmr(void)      /* Timer 0 initialisieren */
{ TR0=0;               /* Timer 0 stoppen */
  TH0=0xFF;
  TL0=0x00;            /* Timer-Anfangswert laden: -100 in 16 Bits */
  TMOD = TMOD & 0xF0; /* TMOD löschen */
  TMOD = TMOD | 0x01; /* Timer 0 zählt als 16-Bit-Zähler im internen Takt */
  ET0=1;               /* Timer0-Interrupt ein */
  EA=1;                /* alle maskierten Interrupts ein */
  TR0 = 1;             /* Timer 0 einschalten */
}

void main(void)
{ ACC = 0;      /* Akku auf 0 setzen */
  SetTmr();     /* Timer einschalten */
  for (;;);     /* Endlosschleife */
}
```

Dieses Programm funktioniert im DS51 "fast" einwandfrei (Haltepunkt auf Sourcecodezeile 14, indem BS 0A139 eingegeben wird), nur wird bei jeder Timer-Unterbrechung in der Interruptroutine der Akkumulator neuerlich von 0 auf 1 gestellt, statt ständig höherzuzählen. Ein Blick in den generierten Assemblercode (im Darstellungsmodus Mixed) erklärt dieses dem C-Quellcode nicht entsprechende Verhalten sofort:

(1) Das Nullsetzen von TL0 in der Routine SetTmr erfolgt anders als das Nullsetzen von TH0, nämlich mit Hilfe des Akku, der aber nicht gepusht bzw. gepopt wird. Wenn man im DS51 den entsprechenden Code "patcht", nämlich MOV TL0,#0 (zum Glück belegt dieser Befehl gerade so viele Bytes, nämlich 3, wie die beiden ursprünglichen Befehle), dann kommt man der korrekten Funktion schon näher:

```
13:    void SetTmr(void)              /* Timer 0 initialisieren */
14:    { TR0=0;                       /* Timer 0 stoppen */
A139H #13:    CLR     TR0(8CH)
15:           TH0=0xFF;
A13BH #15:    MOV     TH0(8CH),#0FFH
16:           TL0=0x00;                       /* Timer-Anfangswert laden: -100 in 16 Bits */
A13EH #16:    CLR     A
A13FH                 MOV             TL0(8AH),A
17:           TMOD = TMOD & 0xF0; /* TMOD löschen */
A141H #17:    ANL     TMOD(89H),#0F0H
18:           TMOD = TMOD | 0x01; /* Timer 0 zählt als 16-Bit-Zähler im internen Takt */
A144H #18:    ORL     TMOD(89H),#01H
19:           ET0=1;                  /* Timer0-Interrupt ein */
A147H #19:    SETB    ET0(0A9H)
20:           EA=1;                   /* alle maskierten Interrupts ein */
A149H #20:    SETB    EA(0AFH)
21:           TR0 = 1;                        /* Timer 0 einschalten */
A14BH #21:    SETB    TR0(8CH)
22:    }
23:
```

Man ändert also den Maschinencode mit ASM 0A13E und erhält dann im Language-Fenster folgendes:

```
16: TL0=0x00; /* Timer-Anfangswert laden: -100 in 16 Bits */
A13EH #16:   MOV       TL0(8AH),#00H
```

(2) In der Funktion Tim0Int werden, da die Registerbank nicht gewechselt wird, im generierten Code äußerst verdienstvoll sämtliche Register anfangs gepusht und am Ende wieder gepopt, also auch der Akku. Da die Aufgabe der Interrupt-Routine in diesem Fall unglücklicherweise darin besteht, den Akku zu inkrementieren, wird dies daher immer vor dem Verlassen der Interruptroutine unwirksam gemacht. Abhilfe besteht darin, daß man die PUSH/POP ACC-Befehle durch jeweils zwei NOPs ersetzt. Dann funktioniert das Programm genauso wie die Assembler-Version:

```
8:     void Tim0Int(void) interrupt 1
9:     #endif
A100H #8:    PUSH      ACC(0E0H)
A102H        PUSH      B(0F0H)
A104H        PUSH      DPH(83H)
   ...
A132H        POP       DPH(83H)
A134H        POP       B(0F0H)
A136H        POP       ACC(0E0H)
A138H        RETI
13:   void SetTmr(void)     /* Timer 0 initialisieren */
```

```
8:     void Tim0Int(void) interrupt 1
9:     #endif
A100H #8:    NOP                       <-- ge"patcht"
A101H        NOP                       <-- ge"patcht"
A102H        PUSH      B(0F0H)
A104H        PUSH      DPH(83H)
A106H        PUSH      DPL(82H)
   ...
A132H        POP       DPH(83H)
A134H        POP       B(0F0H)
A136H        NOP                       <-- ge"patcht"
A137H        NOP                       <-- ge"patcht"
A138H        RETI
13:   void SetTmr(void)     /* Timer 0 initialisieren */
```

Aus diesem Beispiel ist zu lernen, daß man auf der Ebene einer höheren Programmiersprache beim direkten Manipulieren von Registern vorsichtig sein sollte. Wäre die Aufgabe der Interrupt-Routine etwa das Inkrementieren einer beliebigen Variablen, etwa auch eines Ports, gäbe es das Problem nicht.

Schließlich ist aus diesem Beispiel auch zu lernen, daß man ohne Assemblerkenntnisse wahrscheinlich leicht in größere Schwierigkeiten kommen kann, ohne eine Chance zu haben, deren Ursache zu durchschauen.

Eine Funktion wird zur Interrupt-Routine, indem sie das Schlüsselwort interrupt # erhält. # bezeichnet den gewünschten Interrupt, indem vom Compiler die Interrupteinsprungadresse 8*#+3 generiert wird, also zum Beispiel für den Timer 0 im vorstehenden Beispiel 8*1+3 = 11 = 0Bh, das ist die standardmäßige Interrupteinsprungadresse für den Timer 0.

Eine mittels des Schlüsselworts interrupt deklarierte Interrupt-Funktion kann auch noch das Schlüsselwort using # enthalten, wobei für # die Nummer der in der Interrupt-Routine zu verwendenden Registerbank anzugeben ist (Default ist 0).

Interrupt-Funktionen können weder Parameter übernehmen noch Ergebnisse zurückgeben. Funktionen, die von einer Interrupt-Routine aufgerufen werden, müssen mit der selben Registerbank arbeiten wie die Interrupt-Routine selbst. Ein Interrupt-Routine sollte so kurz wie möglich sein. An ihrem Ende muß der Interrupt wieder "enabled" werden; dies geschieht bei den Interrupts 0 bis 3 selbsttätig beim Lesen der entsprechenden Register, beim Interrupt der seriellen Schnittstelle muß man jedoch explizit das Bit ES im IE-Register setzen.

Bemerkenswert ist noch die Steuerung der bedingten Compilierung: da der TURBO-C-Compiler das Schlüsselwort interrupt nicht kennt, wird die Interrupt-Rotine zwecks einfachen Entwickelns und Testens unter Zurückgreifen auf die im jeweiligen Compiler abfragbaren Makros __TURBOC__ bzw. __C51__ einfach anders definiert als für den C51-Compiler.

5. Testen auf der Zielhardware mit TS51

TS51 ist hinsichtlich Benutzeroberfläche im wesentlichen identisch mit DS51. Um Verwechslungen zu vermeiden, hat es sich bewährt, TS51 mit anderer Bildschirmauflösung (Zeilenzahl) zu betreiben als DS51. Außerdem ist anstelle des .IOF-Treibers ein .IOT-Treiber (MON51.IOT) erforderlich; dieser besorgt die Kommunikation mit dem Zielsystem über die serielle Schnittstelle.

Auf dem Zielsystem (Target-System, deshalb TS51) befindet sich ein Monitorprogramm, also ein Betriebssystem (im Monitor-Eprom) für das Kontrollieren des Testvorgangs. Selbstverständlich wird damit ein kleiner Teil der Hardware-Ressourcen des Zielsystems belegt, nämlich:

5 kByte im externen Codespeicher (Eprom) ab Adresse 0,
zusätzlich 6 Byte im Stack des Anwenderprogramms,
256 Byte RAM, das als XDATA- und als CODE-Speicher ansprechbar ist,
Serielle Schnittstelle,
zusätzlich 5 kByte im XDATA-Speicher bei Verwendung des Trace-Buffers.

Nach dem Laden von MON51 blickt man sofort im Language-Fenster auf den untersten Codespeicher und erkennt die Interrupt-Einsprungadressen, welche alle Sprünge auf die µProfi-orientierten, um A000h höher liegenden Interrupt-Adressen enthalten:

```
 Options  Key  View  Peripheral  Map
0000H        LJMP      00AEH
0003H        LJMP      0A003H
0006H        NOP
0007H        NOP
0008H        NOP
0009H        NOP
000AH        NOP
000BH        LJMP      0A00BH
000EH        NOP
000FH        NOP
0010H        NOP
0011H        NOP
0012H        NOP
0013H        LJMP      0A013H
0016H        NOP
0017H        NOP
```

Im Exe-Fenster kann man noch die bisher ausgeführten Kommandos, im wesentlichen aus TS51.INI, einsehen:

```
tScope-51+ V5.10
Copyright KEIL ELEKTRONIK GmbH 1991
>/***************************************/
>/* Initialisation File for tScope-51  */
>/***************************************/
>\o\m                          /* Options Medium (Lines) */
>\v\s                          /* View Serial (Off)      */
>\d\d\d\d\d\d\d\d\d\d\d\d\d   /* Decrease Exe-Window    */
>LOAD C:\8051\DSCOPE\MON51.IOT  BAUDRATE (9600) CPUTYPE (8051) COM1:
INSTALLED FOR PC/XT/AT (COM LINE 1) USING HARDWARE INTERRUPT SERVICE

BAUDRATE: 9600

MONITOR-51 Driver for tScope-51+  V1.2
(C) Franklin Software Inc./KEIL ELEKTRONIK GmbH 1991

*** MONITOR MODE ***
>
═Exe════════════════════════════════════════════════════════════════
```

Sollte es beim Laden von MON51 Fehlermeldungen geben, insbesondere hinsichtlich der Kommunikation, sollte man TS51 verlassen, am µProfi RESET drücken und TS51 neu laden. Eine Meldung "Breakpoint does not exist" oder dergleichen nach dem Laden eines zu testenden .ABS-Files verschwindet oft beim näch-

sten Ladeversuch. Es ist auch schon vorgekommen, daß das .ABS-File nicht richtig übertragen wurde, ohne daß eine Fehlermeldung produziert wurde; falls also das im Language-Fenster dargestellte Programm nicht mit dem ursprünglichen Quellprogramm übereinstimmt: nochmals laden!

Nach dem Laden sollte man den PC auf die Startadresse des Programms stellen (Bei Assemblerprogrammen normalerweise A000h, bei C-Programmen A100h); man kann dann im Prinzip wie mit DS51 testen. Der Testlauf ist zu beenden durch Drücken der RESET-Taste am µProfi.