Programmierung

In diesem Fachgebiet passiert das, was Sie sicherlich am meisten interessiert, nämlich der Umgang mit Programmen, das Schreiben von Programmen, kurzum ein grosser Teil dessen, was unter dem Lehrgangstitel "Assembler-Programmierung" erwartet wird.

Assembler

Unser Betriebssystem erleichert das Programmieren ganz erheblich mit einem komfortablen 2-Pass-Assembler. Das heißt, daß der Assembler seine Aufgabe in zwei Durchläufen erledigt. Die braucht er auch, um neben dem eigentlichen Assemblieren auch noch eine Labelrechnung durchzuführen und eine Symboltabelle aufzustellen. Was das alles bedeutet, wird noch ausführlich behandelt. Fangen wir von vorne an, was heißt "Assembler" eigentlich?

Das englische Wort "to assemble" bedeutet zusammensetzen, folglich wäre der Assembler ein "Zusammensetzer". Das klingt denn doch etwas komisch, man bleibt lieber, wie so oft, beim gewohnten englischen Ausdruck, geht sogar noch einen Schritt weiter und leitet noch ein paar verdeutsche Worte ab, z.B. assemblieren. Aber, was setzt der Assembler denn nun zusammen?

Wie Sie wissen, kann ein Mikroprozessor nur ein in 8-Bit-Zahlen codiertes Programm aus dem Speicher auslesen und ausführen. Mit dieser Maschinensprache (auch Opcode genannt) kann man allerdings nicht viel anfangen. Wer merkt sich schon alle Befehle eines Mikroprozessors in 8-Bit-Zahlen, besonders, wenn es so viele sind , wie beim Z80? Außerdem ist es recht mühsam, längere Programme in dieser Form einzutippen, und das auch noch fehlerfrei. Sie wissen es: ein einziges Byte vertippt und nichts geht mehr.

Der Assembler erspart nun nicht die Tipperei an sich, die Programme müssen weiterhin eingegeben werden. Aber nicht in Maschinencode, der Assembler versteht nämlich die mnemonischen Abkürzungen, wie LD, Call usw.Diese übersetzt er in den Maschinencode.Das ist natürlich einfacher und bequemer, solche Ausdrücke, die einem immerhin etwas sagen, zu gebrauchen. Der Assembler entdeckt auch Tippfehler. CSLL statt CALL zum Beispiel versteht er nicht und macht darauf aufmerksam.

Unser Assembler kann jedoch noch viel mehr. Zunächst einmal brauchen keine Adressen mehr für Speicherzellen oder

auch für Unterprogramme eingegeben zu werden. Man gibt den Speicherzellen oder Unterprogrammen einfach Namen (Labels) und kann sie dann immer unter diesem Namen ansprechen. Das hat folgenden Vorteil: wenn irgendwo im Programm einige Befehle eingefügt werden müssen, braucht man sich überhaupt nicht darum zu kümmern, daß sich alle Befehle verschieben. Der Assembler übernimmt die Adressenrechnung in eigener Regie, er rechnet alle Adressen neu aus.

C
2

Neben diesen Vorzügen kann der Assembler noch mehr. Er verfügt über eigene Anweisungen, sogenannte Pseudobefehle, deren durchdachte Anwendung unter anderem erst den Unterschied zwischen dem Eintippen von mnemonischen Codes und der Assemblerprogrammierung ausmacht.

Sie werden diese Vorzüge des Assemblers gleich in der Praxis kennenlernen, vor dem Eingeben des ersten Programms ist nur noch ein kurzer Ausflug in das Fachgebiet BETRIEBSSYSTEM notwendig, um einige Systemfunktionen zu erläutern.

Zusammenfassung

Der Assembler ist der Teil des Betriebssystems, der in mnemonischen Codes eingegebene Programme in Maschinencode umwandelt. Dabei werden Tippfehler angezeigt. Unser Assembler übernimmt außerdem noch die Adressenrechnung. Dadurch kann ungeachtet von Programmänderungen mit Labels gearbeitet werden.

Fragen

1 Was versteht man unter "mnemonischer Code"?

2. Wann wird die Adresse einer Speicherzelle für den Ablauf eines Programms wichtig?

(Die Antworten zu diesen Fragen finden Sie auf Seite G2.)

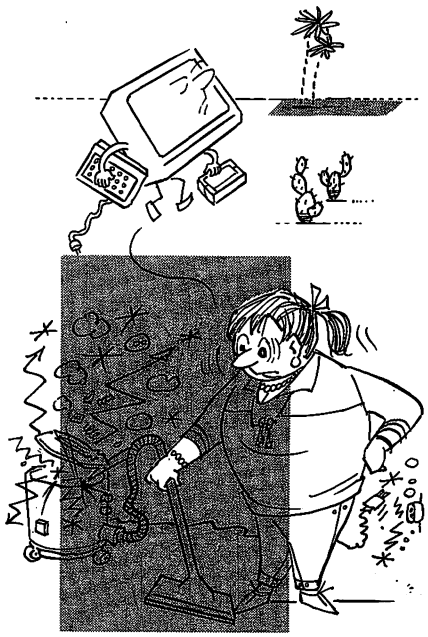
Programm 1: Zeichenausgabe

Unter der Griffmarke F sind alle in diesem Lehrgang als Praktikum gedachten Versuche aufgelistet. Das Listing für den vorliegenden Versuch finden Sie auf Seite F1. Schauen Sie sich die Art der Darstellung einmal an. Sie bemerken eine ganze Anzahl von Überschriften und Kommentaren. Bei einem so einfachen "Prográmmchen" wird man das in der Praxis gewiß nicht immer so ausführlich gestalten, für Sie soll es jedoch am Anfang möglichst ohne Mißverständnisse dastehen und als Einführung in den Umgang mit dem Assembler dienen.

Geben Sie das Programm 1 mit dem Editor in Ihren Computer ein. Die grau unterlegten Teile sind unbedingt nötig. Den Rest können Sie, wenn es schnell gehen soll, auch weglassen, worunter allerdings, wenn die Seite F1 nicht immer auf dem Tisch liegt, die Übersichtlichkeit leidet.

Jetzt können Sie anwenden, was beim Umgang mit dem Editor gezeigt wurde: für die kurzen Abstände die Tabulatorfunktion, für die grossen Abstände ESC M und ESC W usw.

Damit die Arbeit nicht umsonst war, sollten Sie das eingegebene Programm gleich anschließend auf Cassette (oder Diskette) abspeichern -ein Kurzschluß des häuslichen Staubsaugers und alles war umsonst! Wobei es selbstverständlich auch noch andere Unfälle gibt, die zu nochmaligem Eintippen zwingen, etwa irgendwelche Bedienungsfehler, die das System "in die Wüste laufen" lassen.



Nach dem Eingeben verlassen Sie den Editor und rufen den Assembler mit dem Kommando ASM auf. Danach wird folgendes Bild auf dem Bildschirm stehen:

```
>asm

Z80-Assembler
pass 1:
+++
pass 2:
+++

no fatal errors


SUM=0C30
CRC=E225

>
```

Sollten Sie etwas falsch eingegeben haben, gibt der Assembler die falschen Zeilen mit vorangestelleten Fehlermeldungen aus. Dabei bedeutet:
L: Zeile falsch, z.B. falscher Opcode.
U: Undefined, nicht definiertes Symbol.
Die übrigen Fehlermeldungen werden Ihnen zunächst nicht begegnen, Sie lernen sie später kennen. Wenn Ihr Programm in Ordnung ist (auch die bei SUM= und CRC= stehenden Werte müssen stimmen), dann starten Sie es mit RUN:


```
>run
Text

>
```

Das Programm schreibt das Wort Text auf den Bildschirm und kehrt dann ins Betriebssystem zurück. Gewiß keine große Tat, aber sehen Sie sich nun den Text dieses einfachen Beispiels einmal an.

Das Programm beginnt mit einer TITLE-Anweisung. Diese weist den Assembler an, bei eingeschalteter Druckerprotokollierung an den Anfang jeder Seite den Titel "ZEICHEN-AUSGABE - Programm 1" zusammen mit der Seitennummer zu schreiben. Wenn Sie einen Drucker angeschlossen haben, können Sie das mit dem Kommando ASM/L ausprobieren.

Es taucht die Frage auf, warum die TITLE-Anweisung eingerückt ist: sie darf nicht am Anfang der Zeile stehen. Der Assembler versteht alle Zeichenfolgen, die am Zeilenanfang stehen, als Symboldefinitionen (das wird gleich erklärt). Die Anweisungen müssen immer mit Zwischenräumen (Tabulatorfunktion) eingerückt sein.

Wie Sie sehen können, steht TITLE und einige andere, zunächst unbekannte Zeichenfolgen in gleicher Reihe mit den weiter unten folgenden vertrauten Befehlen wie z.B. LD. Trotzdem entspricht TITLE nicht einem richtigen Z80-Befehl, sondern ist eine Anweisung an den Assembler, daß er etwas bestimmtes tun soll. Man nennt solche Anweisungen im allgemeinen Pseudobefehle.

Nach dem Pseudobefehl TITLE fängt das Programm noch nicht an, es folgen noch einige Kommentarzeilen. Vor den Kommentaren muß ein Semicolon stehen, der Assembler ignoriert nämlich alle Zeichen nach einem Semicolon bis zum Zeillenende. Das bedeutet, nach einem Semicolon können Sie schreiben, was Sie wollen. Und das sollten Sie auch tun, um die Programme auch für andere verständlich zu machen. Ganz abgesehen davon, daß auch Ihnen selber solch ein gut kommentiertes Programm nach einiger Zeit wieder schneller verständlich wird, als eine Liste von Befehlen und Anweisungen ohne jede Kommentierung.

Die nächste Zeile im Programm lautet

```
System      EQU     00005H
```

Über die Adresse 0005H gehen die Systemaufrufe, das wissen Sie bereits. Mit dem Pseudobefehl EQU (Englisch equal = ist gleich) wird dem Symbol SYSTEM der Wert 0005H fest zugewiesen. Der Assembler wird also jedesmal, wenn er das Symbol SYSTEM findet, den Wert 0005H einsetzen.

Der Suffix H sagt dem Assembler, daß es sich um eine sedezimal dargestellte Zahl handelt. Ansonsten würde er von einer dezimalen Darstellung ausgehen (was im vorliegenden Fall egal ist), da seine Standard-Zahleninterpretation dezimal ist.Der Assembler erkennt aber auch Zahlen aus dem Oktalsystem richtig, wenn sie mit dem Suffix O markiert sind, außerdem binär dargestellte Zahlen mit dem Suffix B. Sie brauchen also im Bedarfsfall niemals Zahlen umzuwandeln, der Assembler interpretiert alle Zahlensysteme richtig. Alle? - tatsächlich, mit dem Pseudeobefehl

```
.RADIX Dezimalzahl
```

läßt sich die Standard-Zahleninterpretation auf beliebige Zahlensysteme umschalten. Diesen Pseudobefehl werden Sie vermutlich nicht so oft gebrauchen, hingegen ist ein Hinweis nützlich, wie Konstanten mit vielen Stellen lesbarer gemacht werden können. In diesem Fall darf das Zeichen $ als Abteilung eingefügt werden, also beispielsweise statt 1100010100111010B wird 1100$0101$0011$1010B geschrieben.

Weiter im Programm: in den folgenden Zeilen sind die abgekürzten Namen der Systemfunktionen als Symbole und dazu jeweils die entsprechenden Code-Nummern eingesetzt, sowie einige ASCII-Steuerzeichen als Symbole definiert. Beachten Sie die Null am Anfang jeder Zahlenkonstanten: nur, was mit einer Ziffer von 0 bis 9 beginnt, erkennt der Assembler als Zahlenkonstante.

Ein weiterer Pseudobefehl steht vor dem Beginn des eigentlichen Programms: mit ORG wird dem Assembler bedeutet, an welcher Stelle im Speicher das in Maschinencode übersetzt Programm stehen soll. Es ist die Adresse 0100H angegeben, also der Anfang des Anwenderprogramm-Bereichs. Diese Adresse wählt der Assembler übrigens automatisch, wenn ORG weggelassen wird.

Nach ORG folgt das Symbol START. Es markiert eine bestimmte Stelle im Programm, weshalb man auch von einem Label (Etikett) spricht. Labels werden mit einem Symbol am Anfang einer Zeile definiert. Nach einem Label kann, muß aber nicht, ein Doppelpunkt stehen. Und dann kommt endlich der erste (für Sie vertraute) Z80-Befehl: LD

E,"T" gefolgt von einem Kommentar.Es soll der Buchstabe T auf dem Bildschirm ausgegeben und dafür der entsprechende ASCII-Code ins E-Register geladen werden. Sie brauchen nicht in der ASCII-Tabelle nachschauen, der Assembler nimmt Ihnen die Arbeit ab, wenn der gewünschte Buchstabe in Anführungsstriche gesetzt ist. Mit dem Buchstaben T und anschließend mit den anderen drei Buchstaben wird das Unterprogramm ZEICHEN aufgerufen, das die jeweiligen Zeichen auf den Bildschirm schreibt.

Das folgende Unterprogramm NEUE ZEILE setzt den Cursor in eine neue Zeile.Der Strich zwischen NEUE und ZEILE ersetzt in gleicher Weise wie $ zwischen den Zahlen einen Abstand (space), der in einem Label bzw. Unterprogramm-Namen nicht gestattet ist (wenn dieser tiefgelegte Strich auf Ihrer Tastatur nicht vorhanden ist, müssen Sie das Dollarzeichen verwenden). Anschließend wird das Programm beendet, der Funktionscode für die Warmstartfunktion wird ins C-Register geladen und dann die Systemadresse aufgerufen.Die beiden Unterprogramme ZEICHEN und NEUE ZEILE sind recht einfach. Zeichen lädt den Funktionscode für die Konsolausgabe ins C-Register und ruft das Betriebssystem auf; NEUE ZEILE gibt die beiden Zeichen für Wagenrücklauf und Zeilenvorschub aus. Das Programm wird beendet durch den Pseudobefehl END. Danach läuft nichts mehr, END hat als Parameter das Symbol START und auf diese Adresse setzt der Assembler den Programmzähler, egal, ob nach END noch etwas steht.

Zusammenfassung

Anhand eines einfachen Programms wird begonnen, den Umgang mit dem Assembler zu erläutern. Dazu gehört der Umgang mit den Pseudobefehlen, mit Symbolen und Labels. Wichtig ist aber auch die ausführliche Kommentierung der Programmschritte.

Fragen:

1. Warum steht zwischen NEUE ZEILE ein Dollarzeichen oder ein Strich?

2. Wie sieht die Fehlermeldung des Assemblers aus, wenn Sie einen Eingabefehler gemacht haben? Probieren Sie es aus!

(Die Antworten zu diesen Fragen finden Sie auf Seite G2.)

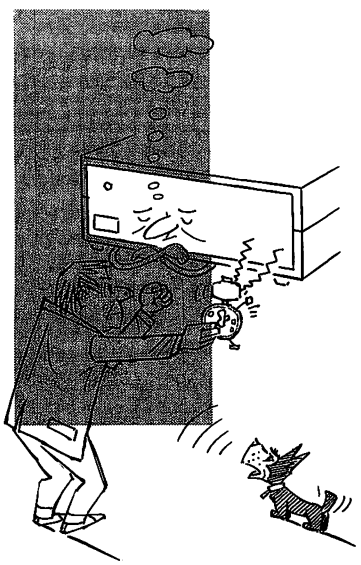
Programm 2: Stringausgabe

Nachdem Sie die Systemfunktion 9 zur Stringausgabe kennengelernt haben, sollen Sie deren Anwendung in einem Programm ausprobieren. Geben Sie das ab Seite F3 aufgelistete Programm mit dem Editor in Ihren Computer ein. Dabei gibt es Möglichkeiten, Schreibarbeit zu sparen. Entweder, das Programm 1 steht noch im Textspeicher, dann brauchen Sie nur die grau unterlegten Erweiterungen einzugeben. Oder, Sie lesen das Programm 1 von der Cassette (oder Diskette) in den Textspeicher ein und verfahren in gleicher Weise.

In den Erweiterungen zum Programm 1 werden zunächst zwei weitere Symbole für die Benutzung der Systemfunktion 9 definiert: STRAUSF für die Stringausgabe und STOP für das Stringende. Ferner wird der auszugebende String (er befindet sich direkt nach dem Hauptprogramm) mit dem Label STRING1 versehen und im Hauptprogramm mit den Z80-Befehlen LD und CALL ausgegeben.

Außerdem ist das Unterprogramm STRING hinzugekommen,das die Systemfunktion 9 in der schon bekannten Weise aufruft.

Für das Ablegen des Strings im Speicher wird ein weiterer Pseudobefehl des Assemblers benutzt: DB steht für define Byte (definiere Byte) und bewirkt, daß die nach dem Pseudobefehl stehenden und durch Kommata getrennten Zahlen byteweise im Speicher abgelegt werden. Dabei dürfen die Zahlen oder allgemeiner gesagt die Daten auch Symbole oder in Anführungszeichen gesetzte Strings sein. In letzterem Fall legt der Assembler, wie Sie wissen, die entsprechenden ASCII-Codes im Speicher ab.



Speichern Sie das Programm 2 zuerst auf Cassette ab, danach wird es mit ASM assembliert. Noch ein Wort zur anschließend erscheinenden Bildschirmausgabe. Es macht sicher Spaß, zuzuschauen, wie die beiden Reihen von Pluszeichen aufmarschieren, fast wie in einem Bildschirmspiel.

Diese Anzeige ist nun keineswegs nur dazu da, sichtbar zu machen, daß der Assembler bei besonders langen Programmen etwas tut und nicht eingeschlafen ist. Vielmehr ist jedes Pluszeichen sozusagen die Quittung für jeweils 256 Bytes assemblierten Quelltext. Gleichzeitig mit der Ausgabe der Pluszeichen wird übrigens die Tastatur abgefragt, so daß jedesmal der Assemblierungsvorgang mit einer Eingabe, in diesem Fall CTRL-C, abgebrochen werden kann. Auf die Bedeutung der Angaben SUM und CRC kommen wir noch zurück, vorerst genügt der Hinweis, daß es Prüfsummen sind.

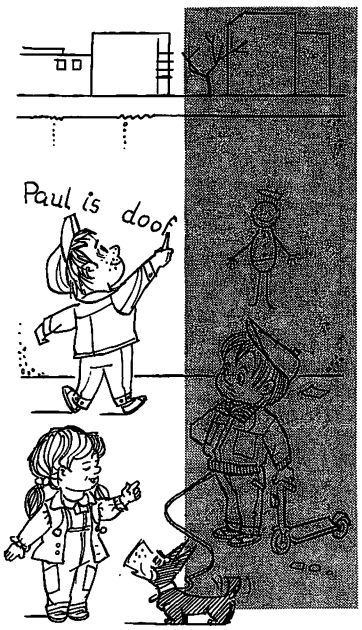
Nach der Assemblierung des eingegebenen Programms muß auf dem Bildschirm stehen:

```
>asm

Z80-Assembler
pass 1:
+++
pass 2:
+++

no fatal error(s)


SUM=17E0
CRC=70E9

>
```

Falls keine Fehler angezeigt werden und die beiden Werte für SUM und CRC stimmen, können Sie mit dem Kommando RUN das Programm starten:

```
>run
Textverarbeitung
mit ZEAT
```

Schreiben Sie einmal was anderes mit diesem Programm auf den Bildschirm, vielleicht ein Uralt-Graffiti:




```
Paule
is
doof!
```

oder auch einen längeren String seriöserem Inhaltes - wie dem auch sei, spielen Sie mit dem Programm. In unserem Beispiel müssen die Zeichen P, a, u, l, e nach dem Label START mit LD-Befehlen eingegeben werden. Die Definition des Strings nach dem Label STRING1 beginnt mit einer Leerzeile usw. - Sie werden es selber herausbekommen.

Auf Seite B18 hatten Sie die Systemfunktion 247 kennengelernt. Mit ihr kann man den Inhalt des Textspeichers auf dem Bildschirm ausgeben. Diese Möglichkeit, zusätzlich zu den bisher bekannten, wollen wir in einem weiteren Programm untersuchen, wobei Sie dann auch zum ersten mal den Tester gebrauchen.

Programm 3: Textspeicherausgabe

Geben Sie das ab Seite F5 aufgelistete Programm 3 in Ihren Computer ein und speichern Sie es auf Cassette (oder Diskette) ab, wenn die Assemblierung die folgende Bildschirmanzeige ergibt:

```
>asm

Z80-Assembler
pass 1:
++++
pass 2:
++++

no fatal error(s)

SUM=1AD7
CRC=61F8

>
```

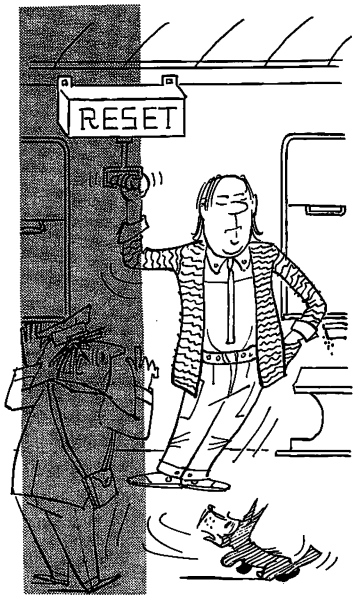
Das Programm gibt mit den bekannten Unterprogrammen STRING und NEUE ZEILE einen Anfangstext auf dem Bildschirm aus und ruft dann die Systemfunktion 247 auf, die den Textspeicheranfang ins HL-Registerpaar lädt.

Dann wird ein Zeichen aus dem Textspeicher geladen und geprüft, ob es das Zeichen für Datei-Ende (EOF) ist. In diesem Fall wird das Programm beendet mit einem Sprung zum Label ENDE. Im anderen Fall wird das Zeichen ausgegeben und zum Label SCHLEIFE zurückgesprungen, wo das nächste Zeichen bearbeitet wird.

Starten Sie das Programm jetzt mit dem Kommando RUN:

```
>run

Inhalt des Textspeichers:
   CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Das Ergebnis ist recht enttäuschend (ein richtiger Flop, wie auch Leute sagen, die nicht wissen, was ein Flipflop ist): das Programm funktioniert nicht, sondern es werden nur sinnlose Zeichen auf dem Bildschirm ausgegeben.



C
10

Noch schlimmer, es läßt sich überhaupt nicht mehr anhalten, Sie müssen die Notbremse ziehen. Anders gesagt, der Computer muß mit der Reset-Taste neu gestartet werden und jetzt ist es gut, wenn das Programm auf Cassette (oder Diskette) gespeichert wurde.

Von dort laden Sie es wieder in den Textspeicher. Auf den folgenden Seiten des Fachgebiets TESTER erfahren Sie dann, wie man mit dem ZEAT-Tester einen Programmfehler sucht und findet.

Zusammenfassung

In zwei Programm-Beispielen wird die Anwendung weiterer Systemfunktionen gezeigt. Dabei läuft das Programm mit der Systemfunktion 247 -Hole Textspeicheranfang nicht wunschgemäß, weshalb mit dem Tester auf Fehlersuche gegangen werden muß.

Fragen:

1. Beim Assemblieren zeigt der Assembler in beiden Durchläufen mit Pluszeichen an, daß er arbeitet. Beim Programm 3 stehen 4 Pluszeichen auf dem Bildschirm. Wieviel Speicherplatz braucht das Programm demnach?

2. Welche Eingaben in unseren Programmbeispielen werden zwischen Anführungsstriche gesetzt?

(Die Antworten zu diesen Fragen finden Sie auf Seite G3.)

Verbessertes Programm 3: Textspeicherausgabe

Die Fehlersuche mit dem Tester hat gezeigt, warum das Programm nicht läuft: beim Aufruf von Systemfunktionen werden Registerinhalte gelöscht, in diesem Fall derjenige des HL-Registers, also die Anfangsadresse des Textspeichers. Das kann verhindert werden: Sie kennen die PUSH- und POP-Befehle des Z80, mit denen sich der Mikroprozessor Registerinhalte in ein "Notizbuch" (Stack) notiert und dort wieder nachliest. Im Unterprogramm ZEICHEN muß der Inhalt des HL-Registers während des Aufrufs der Systemfunktion gerettet werden:


```
;************************************************************
;Unterprogramm ZEICHEN
; - schreibt das Zeichen im E-Register auf den Bildschirm
;************************************************************

ZEICHEN:

        LD      C,CONAUSF       ;Der Funktionscode wird ins
                                ;C-Register geladen,
        PUSH    HL              ;HL gerettet
        CALL    SYSTEM          ;und das Betriebssystem
                                ;aufgerufen
        POP     HL              ;wiederherstellen
        RET                     ;Rücksprung
```

Holen Sie sich das Programm in den Textspeicher und ändern Sie das Unterprogramm ZEICHEN ab. Nach der Assemblierung muß folgende Bildschirmanzeige erscheinen:

```
>asm

Z80-Assembler
pass 1:
++++
pass 2:
++++

no fatal error(s)


SUM=1CA3
CRC=C50B


>
```

Probieren Sie das geänderte Programm mit RUN oder mit dem Tester aus, jetzt muß es richtig laufen.

Wenn Sie sich das Programm noch einmal anschauen, fällt Ihnen vielleicht ein Schönheitsfehler auf, beim Unterprogramm STRING nämlich. Dort, wo dieses Unterprogramm aufgerufen wird, lädt sich die Adresse STRING1 ins DE-Registerpaar. Es kann ja sein, daß man schnell einmal wissen möchte, welcher Text mit diesem STRING1 ausgegeben wird.

Das ist beim vorliegenden Programm kein Problem, die mit STRING1 bezeichnete Stelle ist schnell gefunden. Bei längeren Programmen mit vielen unterschiedlichen Textmeldungen (und das ist schließlich die Regel -nicht unser Prögrämmchen) wird das schon recht umständlich. Besser ist es, wenn der auszugebende Text an der Stelle steht, wo das zugehörige Unterprogramm aufgerufen wird. Das ist mit zwei Programmiertricks auch möglich.

Trick 1:

Beim CALL-Befehl springt das Programm an die angegebene Adresse, legt sich jedoch vorher die Rücksprungadresse, also die auf den CALL-Befehl folgende, im Stack ab. Anders gesagt: der CALL-Befehl enthält sozusagen einen automatischen PUSH PC-Befehl. Analog verhält es sich beim RET-Befehl am Ende des Unterprogramms: es wird die Rücksprungadresse vom Stack wieder geholt und deshalb das Programm nach dem CALL-Befehl fortgesetzt. Oder, wie es eben formuliert wurde: der RET-Befehl enthält einen automatischen POP PC-Befehl.

Setzen wir zuerst, wie eingangs gewünscht, im Hauptprogramm den String direkt nach dem Aufruf des Unterprogramms STRING:

```
Hauptprogramm:
    CALL   STRING
    DB     "Textausgabe$"
    ..
    ..
```

Jetzt geht's weiter im aufgerufenen Unterprogramm STRING. Das braucht zuerst die Adresse des Stringanfangs, und die steht im Stack. Dorthin nämlich hat sie der automatische PUSH PC-Befehl von CALL gebracht, es war ja der nächste Programmschritt nach CALL. Also fängt das Unterprogramm mit dem ungewöhnlich anmutenden POP-Befehl an:

```
STRING:
    POP   DE        ;Stringadresse vom Stack holen
    PUSH  DE        ;und wieder aufs Stack zurück
```

Warum kommt die Adresse ins DE-Register? Weil am Anfang des Hauptprogramms die Anfangsadresse des Strings (Label Meldung) ins DE-Register geladen wird.

Trick 2

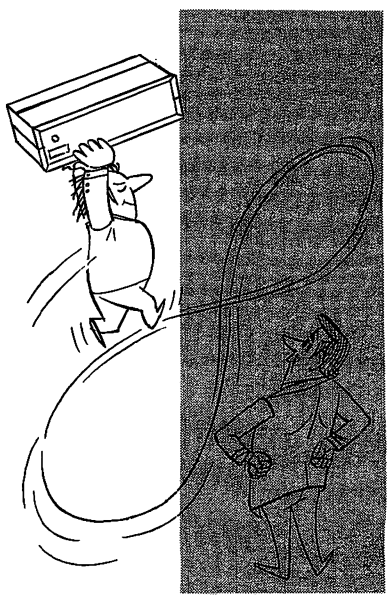
Soweit, so gut, aber der Trick 1 funktioniert so noch nicht. Das Unterprogramm String kann nicht einfach mit einem RET-Befehl enden, dann würde das Programm ja an der Stelle nach dem CALL-Befehl fortfahren. Dort aber steht kein Befehl, sondern eben der String. Das Programm muß stattdessen nach dem Stringende, also nach dem Zeichen $, weitergehen.

Dazu muß zuerst der Anfang des Strings noch einmal mit einem POP-Befehl vom Stack geholt werden und danach das Stringende gesucht werden. Dieses ist das Zeichen $, und die Suche nach einem bestimmten Byte im Speicher ist eine Aufgabe, die recht oft vorkommt, speziell bei Textverarbeitungsprogrammen.

Z80-Befehle

Grundsätzlich läßt sich die Aufgabe (Suche nach einem bestimmten Zeichen im Speicher) mit einigen alltäglichen Befehlen lösen, wir erinnern uns aber bei der Gelegenheit auch noch seltener gebrauchter Befehle. Zunächst jedoch die naheliegende Lösung:

```
        LD    HL,ADRESSE    ;HL mit der Adresse laden,
                            ;ab der gesucht wird
        LD    A,ZAHL        ;A-Register mit dem Byte la-
                            ;den, das gesucht wird
SCHLEIFE:
        CP    (HL)          ;A-Register mit der Spei-
                            ;cherzelle vergleichen, auf
                            ;die HL zeigt
        INC   HL            ;HL erhöhen
        JR    NZ,SCHLEIFE   ;weitermachen, wenn nicht
                            ;gefunden
GEFUNDEN:
        ..
        ..
```



Ahnen Sie, welche Gefahr in der Schleife dieses Programms steckt? Es könnte ja vorkommen, daß der gesuchte Wert im Speicher garnicht vorkommt. Das Programm läuft dann für alle Zeiten in der Schleife!

Mit anderen Worten, es muß noch eine Erweiterung her, damit eine eventuelle Endlos-Schleife verhindert wird. Dafür genügt ein Zähler, der von der festzulegenden Maximalzahl der Suchvorgänge bis auf Null heruntergezählt wird:


```
        LD    HL,ADRESSE    ;HL mit der Adresse laden,
                            ;ab der gesucht wird
        LD    BC,ANZAHL     ;BC-Registerpaar mit der Ma-
                            ;ximalzahl der Schleifen-
                            ;durchläufe laden
        LD    A,ZAHL        ;A-Register mit dem Wert la-
                            ;den, der gesucht wird

SCHLEIFE
        CP    (HL)          ;A-Register mit der Spei-
                            ;cherzelle vergleichen, auf
                            ;die HL zeigt
        INC   HL            ;HL erhöhen
        DEC   BC            ;BC erniedrigen
        JR    Z,GEFUNDEN    ;fertig,wenn gefunden
        LD    D,A           ;Inhalt von A in D verwahren
        LD    A,C           ;C nach A
        OR    B             ;A mit B ODER-Verknüpfen,
                            ;ist Null, wenn BC Null
        LD    A,D           ;A wiederherstellen
        JR    NZ,SCHLEIFE   ;weitermachen, wenn BC nicht
                            ;Null

NICHT_GEFUNDEN:
        ..
        ..

GEFUNDEN:
        ..
        ..
```

Jetzt könnte das Programm laufen, aber nur für sich allein, denn es wird ja der Inhalt des D-Registers zerstört, der woanders noch gebraucht wurde. Außerdem ist es um einige Befehle länger geworden, das wiederum läßt den Ehrgeiz nicht ruhen. Und der Befehlssatz des Z80 verfügt schließlich über spezielle Befehle zur Stringverarbeitung.

Da gibt es zunächt den Befehl CPI. Das ist eine Erweiterung des CP-Befehls und bedeutet CP (compare=vergleiche) und Inkrementieren. Dieser Befehl kann im obigen Programm drei Befehle ersetzen, nämlich CP (HL), INC (HL) und DEC (BC).

Es gibt im Z80-Befehlssatz einen weiteren Befehl, der im vorliegenden Fall noch mehr bringt, nämlich der CPIR-Befehl. Der Buchstabe R steht dabei für repeat= wiederhole. Dieser Befehl ersetzt die gesamte Suchschleife mit vergleichen, inkrementieren von HL, dekrementieren von BC und nachschauen, ob BC Null ist. Das alles macht CPIR ohne die Benutzung weiterer Register, sodaß auch das DE-Register unverändert bleibt. Das Suchprogramm sieht jetzt ganz einfach aus:


```
LD      HL,ADRESSE    ;HL mit der Adresse laden,
                      ;ab der gesucht wird
LD      BC,ANZAHL     ;BC mit der Maximalzahl der
                      ;Schleifendurchläufe laden
LD      A,ZAHL        ;A mit dem Wert laden, der
                      ;gesucht wird
CPIR                  ;Suchbefehl
JR      Z,GEFUNDEN
..
..
```

Im Einzelnen führt der CPIR-Befehl folgendes aus:

```
A  mit (HL) vergleichen          )wiederholen,
HL inkrementieren                )bis A=(HL)
BC dekrementieren                )oder BC=0
```

Es gibt übrigens zu diesen beiden Befehlen (CPI und CPIR) noch zwei analoge Befehle, die den Speicher rückwärts durchsuchen. In dem Fall wird das HL-Registerpaar nicht inkrementiert, sondern dekrementiert. Diese Befehle heißen CPD und CPDR. Der CPDR-Befehl führt folgendes aus:

```
A  mit (HL) vergleichen          )wiederholen,
HL dekrementieren                )bis A=(HL)
BC dekrementieren                )oder BC=0
```

Sowohl beim CPIR-Befehl als auch beim CPDR-Befehl ist das Bedingungsflag Z (Zero=Null) gesetzt, wenn das gesuchte Byte gefunden wurde. Das Bedingungsflag P (Parity) wird gsetzt, wenn der Befehl beendet, also BC zu Null wurde.

Das auf Seite F8 stehende Programm 3b gibt wie das vorangegangene Programm 3 den Inhalt des Textspeichers auf dem Bildschirm aus, es enthält jedoch ein geändertes Unterprogramm STRING.

Geben Sie das Programm in Ihren Computer ein und speichern Sie es auf Cassette ab.

Die daran anschließende Assemblierung muß folgende Bildschirmausgabe zeigen:

```
>asm

Z80-Assembler
pass 1:
++++
pass 2:
++++

no fatal error(s)


SUM=2303
CRC=22B9
```


Probieren Sie das Programm nicht nur mit RUN, schauen Sie sich auch die Funktion des CPIR-Befehls mit dem Tester im Einzelschritt an.

Zusammenfassung

Mit den Befehlen PUSH und POP wird der Fehler im Programm 3, die Überschreibung des HL-Registers, verhindert.

Die übersichtlichere Lösung, bei welcher der String gleich hinter dem Aufruf des Unterprogramms STRING folgt, läßt sich mit dem CPIR-Befehl elegant lösen.

Fragen:

1. Dürfen die Befehle PUSH HL und POP HL auch an Anfang und Ende des Unterprogramms ZEICHEN gesetzt werden, ohne daß sich am Programmablauf etwas ändert?

2. Auf welchen Umstand reagiert der Befehl JR Z,GEFUNDEN auf Seite C15?

(Die Antworten zu diesen Fragen finden Sie auf Seite G3.)

Programm 4 - Zeichenstatistik

Als letztes Programmbeispiel in diesem ersten Heft zeigen wir Ihnen ein Programm, das etwas mehr kann und entsprechend auch etwas weniger einfach ist als die vorangegangenen Beispiele.Man gibt mit der Tastatur irgendein Zeichen ein, und das Programm zählt dann, wie oft dieses Zeichen im Textspeicher vorkommt und gibt die Anzahl als in sedezimaler Darstellung aus.

Im Textspeicher steht zu dieser Zeit das Programm mit Kommentaren. Wenn Sie stattdessen einen normalen Text eingeben, dann können Sie mit dem Programm tatsächlich eine Zeichenstatistik der deutshcen Sprache aufstellen - oder einer Fremdsprache, wenn z.B. ein englischer Text im Textspeicher steht.

Geben Sie das Programm 4 in Ihren Computer ein und assemblieren Sie es. Es muß folgendes auf dem Bildschirm stehen:

```
>asm

Z80-Assembler
pass 1:
+++++
pass 2:
+++++
no fatal error(s)


SUM=4C99
CRC=5GBF


>
```

Das Programm wird mit RUN gestartet und mit der Eingabe von CTRL-C wieder verlassen.

Bevor Sie das ausprobieren, noch etwas zum Nachdenken:

Frage:

1. Es wurde vorher gesagt, daß mit einem "richtigen" Text eine Zeichenstatistik der deutschen Sprache möglich wird. Wie können Sie das Programmlisting aus dem Textspeicher heraus (und einen Text hinein) bekommen

und, wenn kein Programm mehr im Textspeicher steht, läuft es dann überhaupt noch?

(Die Antwort zu dieser Frage steht auf Seite G4. Wenn Sie etwas probieren, bevor Sie dort nachschauen -was Sie ja unbedingt sollten- dann ist es wieder wichtig, das mühsam eingegebene Programm sicher auf der Cassette zu wissen!)

Nach dem Programmstart erscheint folgende Anzeige auf dem Bildschirm:


```
>run

Zeichenstatistik: Bitte Taste drücken - K
Zeichen K (=ASCII 4BH) kommt 0008H mal vor

Zeichenstatistik: Bitte Taste drücken - E
Zeichen E (=ASCII 45H) kommt 0063H mal vor

Zeichenstatistik: Bitte Taste drücken - t
Zeichen t (=ASCII 74H) kommt 0090H mal vor

Zeichenstatistik: Bitte Taste drücken - #
Zeichen # (=ASCII 23H) kommt 0000H mal vor

Zeichenstatistik: Bitte Taste drücken - C

>
```

Wir haben einige Zeichen als Beispiel eingegeben: das E kommt recht häufig vor und das Zeichen # überhaupt nicht. Jetzt können Sie, wenn es Sie interessiert, das Programmlisting im Textspeicher durch einen anderen Text ersetzen und Statistik betreiben.

Noch wichtiger im Sinne dieses Lehrgangs ist allerdings ein Blick auf das Programm selber.Gleich am Anfang fällt eine neue Systemadresse auf: 000H wird mit WSTART definiert. Diese Adresse ist wie auch die Adresse 0005H in gleicher Weise wie im Betriebssystem CP/M definiert. Wird sie angesprungen, dann macht das System einen Warmstart. Es geschieht also praktisch das Gleiche wie beim Aufruf der Warmstartfunktion über die Adresse 0005H. Man hat die Wahl, ein Programm durch Aufruf der Warmstartfunktion oder durch einen Sprung an die Adresse 0000H zu beenden. Es gibt übrigens noch eine dritte Möglichkeit: man kann mit einem RET-Befehl ins Betriebssystem zurückkehren.Im Hauptprogramm wird, nachdem mit STRING die Aufforderung zur Zeicheneingabe auf den Bildschirm gebracht wurde, ein Zeichen von der Tastatur eingelesen und an das Unterprogramm STATISTIK übergeben. Wenn das eingebene Zeichen CTRL-C war, erfolgt ein Warmstart.

Etwas verblüffend auf den ersten Blick ist die Abfrage nach CTRL-C: es wird mit dem ASCII-Zeichen für C minus 40H verglichen. Um das zu verstehen, muß man sich die sedezimale Darstellung des ASCII-Codes etwas genauer anschauen und sogar etwas "Bitpopelei" betreiben.

Wenn Sie sich den ASCII-Code (Tafel 1) anschauen, dann sehen Sie ein Gesetzmäßigkeit in der Zuordnung zwischen den Buchstaben und ihren ASCII-Zeichen. Die Großbuchstaben haben jeweils eine um 20H größere Codezahl als die entsprechenden Kleinbuchstaben, z.B. A entspricht 41 und a entspricht 61. Wenn man sich die Bitmuster der beiden Zahlen untereinander schreibt, sieht das so aus:

```
a=  61H  = 0 1 1 0   0 0 0 1 B        (=97D)
A =  41H  = 0 1 0 0   0 0 0 1 B       (=65D)
```

Die in Klammern gesetzte Dezimaldarstellung der Zahlen zeigt, daß diese vom BASIC gewohnte Darstellungsart die angesprochenen Zusammenhänge nicht deutlich werden läßt. Bei der binären Darstellung erkennt man, daß der Unterschied zwischen kleinem und großem Buchstaben beim Bit 5 liegt: bei a hat es den Wert 1, bei A hat es den Wert 0 (probieren Sie das mit anderen Buchstaben aus). Bei der Umschaltung auf Großbuchstaben muß also nur dieses fünfte Bit zu Null gemacht werden, und genau dies tut die Shift-Taste durch eine entsprechende UND-Verknüpfung.

In gleicher Weise funktioniert die CTRL-Taste, bei ihr geht es um das Bit 6:

```
    C =  43H  = 0 1 0 0   0 0 1 1 B
CTRL-C =  03H  = 0 0 0 0   0 0 1 1 B
```

Die CTRL-Taste muß demnach den Code für C (in diesem Beispiel) mit einer solchen Zahl UND-Verknüpfen, daß das sechste Bit zu Null wird (wobei hier nur dieses Bit interessiert, die Werte der Bits 4,5 und 7 sind systembedingt festgelegt). Das sieht so aus:

```
      0 1 0 0   0 0 1 1 B
UND   0 0 0 1   1 1 1 1 B
-------------------------
      0 0 0 0   0 0 1 1 B
```

Voraussetzung ist, daß die CTRL-Zeichen sämtlich um den Betrag 40H niedriger als der jeweilige Großbuchstabe codiert sind. Das geht aus der ASCII-Steuerzeichentabelle in Tafel 2 nicht ohne weiteres hervor. Die Zuordnung der Steuerzeichen ist in jedem Betriebssystem, Textverarbeitungssystem, bei jedem Drucker usw. anders festgelegt. Beim vorliegenden Betriebssystem ZEAT ist es so, deshalb ist der Befehl CP 'C' -40H möglich. Nach dem, was bei

unserem Ausflug in die Binärdarstellung deutlich wurde, kann man auch so programmieren:

```
CP    'C' AND 0001 1111B   oder:

CP    'C' AND CTRL ,wobei dann definiert werden muß:

CTRL    EQU   0001 1111B
```


Nach diesem Ausflug in die Bit-Ebene weiter im Programm. Die verwendeten Unterprogramme sind Ihnen ja bereits bekannt. Zum Unterprogramm STATISTIK ist den Kommentaren nicht viel hinzuzufügen ausser Erläuterungen zu den verwendeten Sprungbefehlen.

Z80-Befehle

Der Z80-Mikroprozessor hat zwei Arten von Sprungbefehlen. Die JP-Befehle sind absolute Sprungbefehle und bestehen aus dem Befehlsbyte sowie einer aus zwei Byte bestehenden Adresse, die das Ziel des Sprungs angibt. Die JR-Befehle sind dagegen relative Sprungbefehle: nach dem Befehlsbyte folgt ein Byte, das den Abstand vom momentanen Befehlszählerstand zum Ziel des Sprungs angibt. Mit einem Byte lassen sich nur die Zahlen von -128 bis +127 darstellen, der Sprung kann also nicht weiter gehen.

Die relativen Sprungbefehle sind kürzer (und schneller) als die absoluten und werden deshalb bevorzugt angewendet. Wenn Sie im Zweifel über den Abstand zweier Befehle sind, versuchen Sie es einfach mit einem relativen Sprungbefehl. Wenn das nicht klappt, macht der Assembler Sie schon darauf aufmerksam.

Frage:

1. Wie muß das Hauptprogramm abgeändert werden, damit die Statistik für alle Buchstaben des Alphabets automatisch durchgeführt wird?

(Die Antwort zu dieser Frage steht auf Seite G4 bzw. als Listing eines neuen Hauptprogramms auf Seite F16. Widerstehen Sie der Versuchung, dort nachzuschauen und probieren Sie eigene Lösungsmöglichkeiten dieser ersten richtigen Programmier-Aufgabe aus!)

Conway's Life-Game

Die Überschrift weist schon darauf hin, daß hier vor den Problemen der Assembler-Programmierung erst einmal ein Ausflug in ein ganz anderes Gebiet gemacht wird. Auch das ist selbstverständlich nur ein Mittel zum Zweck: das Programm zum "Lebensspiel" gibt eine Menge "Spiel"-Material für unseren Lehrgang.

Man weiß, daß die Bevölkerungsentwicklung lebender Organismen (gleich welcher Art) komplizierten Regeln unterworfen ist. Es gibt ganz bestimmte optimale Bedingungen, unter denen sich die Organismen ausbreiten können. Bei Abweichungen von diesen Bedingungen ist dies nicht mehr möglich, z.B. bei starker Über- oder Unterbevölkerung.

Der englische Mathematiker John Horton Conway hat sich ein Spiel ausgedacht, mit dessen Hilfe eine solche Bevölkerungsentwicklung unter angenommenen Bedingungen in einem einfachen Modell simuliert werden kann. Conway's Life-Game ist also kein Spiel im eigentlichen Sinn, sondern ein spielerisches Simulationsmodell.

In diesem Heft werden Sie ein Programm mitentwickeln, das es ermöglicht, das Simulationsmodell schnell und komfortabel mit dem Computer durchzuspielen.

Zuerst aber müssen Sie Conway's Life-Game kennenlernen und sollten es zunächst von Hand spielen. Dann nämlich, mit etwas "spielerischer" Erfahrung, werden Sie die Programmteile besser verstehen und auch gleich merken, warum es sinnvoller ist, einen Computer damit zu beschäftigen.

Das Spiel wird auf einem möglichst großen Brett mit quadratischen Spielfeldern gespielt; das kann ein Schachbrett oder einfach grob kariertes Papier sein. Jedes Feld des Spielbretts kann entweder belegt (bewohnt) oder leer (unbewohnt) sein, die Verteilung ist zunächst willkürlich.

Als Markierung des Zustands legt man irgendwelche Spielsteine (oder Münzen) auf die belegten Felder. Das Spielbrett mit den Spielsteinen darauf stellt eine Bevölkerungsgeneration dar. Conway's Spielregeln legen nun fest, welche Felder von diesem Anfangsstand ausgehend in der nächsten Generation noch belegt sind. Es gibt drei Möglichkeiten:

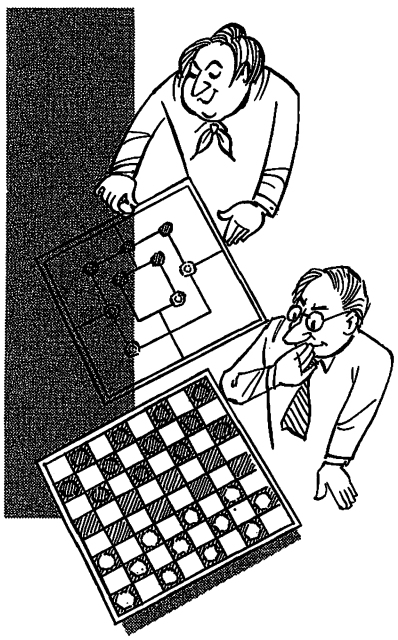
1. Überleben:

   Ein mit einem Spielstein besetztes Feld "überlebt" zur nächsten Generation, wenn zwei oder drei Felder in seiner Nachbarschaft ebenfalls besetzt sind.

2. Sterben:
   Ein mit einem Spielstein besetztes Feld "stirbt" (ist in der nächsten Generation leer), wenn von den angrenzenden Feldern weniger als zwei oder mehr als drei besetzt sind.

3. Geburt:
   Auf einem leeren Feld gibt es eine "Geburt" (das Feld ist in der nächsten Generation wieder besetzt), wenn genau drei seiner Nachbarfelder besetzt sind.

Sie sehen, daß die angenommenen einfachen Wachstumsregeln durchaus an Naturgesetze anklingen: in einer gewissen Gemeinschaft hält sich Leben und entwickelt sich sogar neu; bei zu großer Enge oder Leere geht die Population ein.



Versuchen Sie einmal, das Spiel von irgendwelchen zufälligen Konfigurationen ausgehend über eine Reihe von Generationen zu spielen. Beachten Sie, daß der Generationswechsel immer auf dem gesamten Spielfeld auf einmal stattfinden muß. Es wird also für alle Felder festgestellt, ob sie in der nächsten Generation belegt sind, danach erst dürfen die alten Spielsteine weggeräumt oder neue hinzugefügt werden.

Um das Spiel vernünftig betreiben zu können, nimmt man am besten zwei Spielbretter oder Spielsteine in zwei Farben.

Nachfolgend sehen Sie als Beispiel drei einfache Spielsituationen mit ihren Entwicklungen über drei Generationen:

	Generation 1	Generation 2	Generation 3
stirbt aus:	*** *	*	
bleibt konstant:	** *	** **	** ** *

```
                *                       *
"blinkt":       *             ***       *
                *                       *
```

Die Beispiele zeigen, daß es bei dem Spiel Konfigurationen gibt, die sehr schnell verschwinden, andere bleiben ohne Einfluß von außen immer bestehen.

Die Simulation einer Bevölkerungsentwicklung ist, wie schon gesagt, nicht sehr vollständig; es fehlt z.B. der für jedes Leben so entscheidende Zufallseffekt. Das Spiel kann aber auf einleuchtende Weise darstellen, wie mit einigen wenigen Regeln aus einem Chaos eine Ordnung entstehen kann.

Weil's so schön ist, noch zwei Figuren, mit denen Sie experimentieren können. Die erste entwickelt sich über 10 Generationen zu 4 der vorher gezeigten Blinker, die zweite reproduziert sich in 5 Generationen. Die dann entstehende Ausgangsfigur ist aber um je ein Feld nach unten und zur Seite versetzt. Diese Figur, Glider (Segler) genannt, wandert über das Spielfeld.

```
                    *
wird zu 4 Blinkern: ***


                    *
Glider:           * *
                   **
```

Es ist sehr mühselig, kompliziertere Figuren von Hand durchzurechenen oder zu probieren. Sie werden, wenn das Programm soweit ist, unter anderem eine Figur kennenlernen, die sich selbst reproduziert und dabei noch Glider erzeugt (die sogenannte Kanone, Glider-Gun).

Genug gespielt - das Life-Game dient, wie schon gesagt, schließlich nur als Programmieraufgabe und bietet dabei Gelegenheit, neue Funktionen des Betriebssystems kennenzulernen, den Gebrauch des Testers weiter zu vertiefen, kurzum, ein ganzes Stück weiter in die Assembler-Programmierung einzudringen.

Zusammenfassung

Das Life-Game des Mathematikers Conway ermöglicht die spielerische Simulation der Bevölkerungsentwicklung lebender Organismen. Diese Kombination, ein Spiel mit seriösem Hintergrund, dient als Programmieraufgabe.

C
24

Frage:

1. Welche Anforderungen würden Sie an ein Programm für das Life-Game stellen?

(Die Antwort zu dieser Frage finden Sie auf Seite G5.)

Z80-Befehle

Auf den vorangegangenenen Seiten des Fachgebiets SYSTEMBESCHREIBUNG wurde bereits erwähnt, daß es beim Z80 spezielle Befehle für den Zugriff auf die Portadressen gibt.

Diese IN- und OUT-Befehle können in drei Gruppen aufgeteilt werden:

1. IN A,(port) ;Lade den Inhalt des Portregisters port
;in das A-Register
;port = Portadresse 0...255
; A := (port)

   OUT (port),A ;Speichere den Inhalt des A-Registers
;im Portregister port
;port = Portadresse 0...255
; (port) := A

2. Der IN- und OUT-Befehl zu beliebigen Registern mit Angabe der Portadresse im C-Register:

   IN r,(C) ;Lade den Inhalt des Portregisters,
;dessen Adresse im C-Register steht, in
;das Register r:
; r := (C)

   OUT (C),r ;Speicher den Inhalt des Registers r,
;dessen Adresse im C-Register steht, in
;das Register r:
(C) := r

3. Die Befehle zur Ein- oder Ausgabe ganzer Datenblöcke:

   INI ;Lade den Inhalt des Portregisters,
;dessen Adresse im C-Register steht, in
;die Speicherstelle, auf die das HL-Re-
;gisterpaar zeigt, inkrementiere HL und
;dekrementiere B:
; (HL) := (C)
; B := B-1
; HL := HL+1

   INIR ;wiederhole den Befehl INI, bis B=0

```
IND     ;Lade den Inhalt des Portregisters,
        ;dessen  Adresse im C-Register steht,
        ;in die Speicherstelle, auf die das HL-
        ;Registerpaar zeigt, dekrementiere HL
        ;und dekrementiere B:
        ;       (HL) := (C)
        ;       B := B-1
        ;       HL := HL-1

INDR    ;wiederhole den Befehl IND, bis B=0

OUTI    ;Speichere den Inhalt der Speicherstel-
        ;le, auf die das HL-Registerpaar zeigt,
        ;in dem Portregister, dessen Adresse im
        ;C-Register steht, inkrementiere HL und
        ;dekrementiere B:
        ;       (C) := (HL)
        ;       B := B-1
        ;       HL := HL+1

OTIR    ;wiederhole den Befehl OUTI, bis B=0

OUTD    ;Speichere den Inhalt der Speicherstel-
        ;le auf die das HL-Registerpaar zeigt,
        ;in dem Portregister, dessen Adresse im
        ;C-Register steht, dekrementiere HL und
        ;dekrementiere B:
        ;       (C) := (HL)
        ;       B := B-
        ;       HL := HL-1
OTDR    ;wiederhole den Befehl OUTD, bis B=0
```

C

26

Zusammenfassung

Im Befehlssatz des Z80-Prozessors gibt es eine Anzahl spezieller Befehle, mit denen die Ein- und Ausgabe von Daten über die Ports abgewickelt werden kann. Diese Befehle greifen auf die Portadressen zu und sind die Voraussetzung für das Isolated I/O-System.

Frage:

1.Was müssen alle angeführten IN/OUT-Befehle in der Hardware des Computers bewirken?

(Die Antwort zu dieser Frage finden Sie auf Seite G5.

Life-Game, 1.Teil

Auf Seite C24 wurde die Frage gestellt, welche Anforderungen an ein Programm für das Life-Game zu stellen sind. Sicher haben Sie sich Ihre Gedanken gemacht und mit der Aufzählung auf Seite G5 verglichen. Wir wiederholen hier unsere Liste, die nach der Wichtigkeit der Anforderungen geordnet ist:

1. Eingabe einer Spielfeldsituation mit Tastatur und Bildschirm.

2. Automatisches Errechnen neuer Generationen und Ausgabe auf dem Bildschirm.

3. Eventuell verändern einer vorhandenen Spielsituation.

4. Eventuell ausdrucken einer Spielsituation.

5. Eventuell abspeichern von Spielsituationen.

Die ersten beiden Punkte sind notwendig, damit man das Programm überhaupt benutzen kann, damit das Spiel läuft. Bevor wir an die Realisierung von Punkt 1 in Gestalt eines Programms gehen, gilt es noch, einige allgemeine Überlegungen anzustellen.

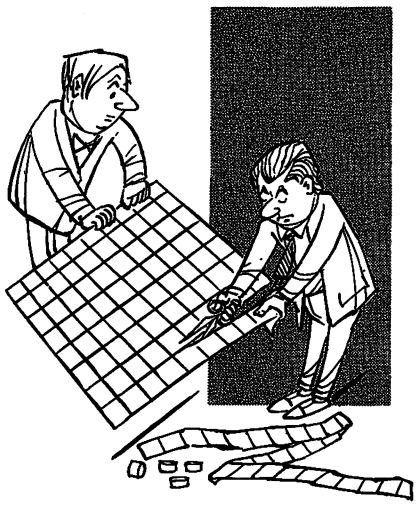
Zur Darstellung eines Spielfelds auf dem Bildschirm ist nicht unbedingt ein Grafikprogramm notwendig, denn die belegten Felder können ganz einfach durch einen Stern (*) und die leeren Felder durch ein Leerzeichen (Space) dargestellt werden. Damit ist gleichzeitig die Größe des Spielfelds vorgegeben: das Datensichtgerät hat 24 Zeilen zu je 80 Zeichen, es ergibt sich ein Spielfeld mit 24 mal 80 Feldern.

Nun muß der Computer irgendwie mit diesem Spielfeld umgehen, die jeweiligen Spielsituationen sollen ja gespeichert und verarbeitet werden. Einfach wäre die Sache, wenn es Speicherzellen in einer zweidimensionalen Anordnung gäbe: man könnte diesen Speicherzellen die Felder des Spielfelds zuordnen und eine Eins einschreiben, wenn das jeweilige Feld belegt ist , oder eine Null, wenn es leer ist.

Warum zweidimensionale Anordnung? -einen Punkt in einer Ebene (ein Feld auf dem Bildschirm) kann man nur durch zwei Koordinatenwerte bestimmen, da sich die Ebene ja in zwei Dimensionen ausdehnt. Man braucht deshalb auch zur Angabe eines Ortes auf der Erdoberfläche die Angabe von Längen- und Breitengrad oder, ein anderes Beispiel, beim

Schachspiel die Buchstaben-Ziffern-Kombination zur Angabe von Zeile und Spalte auf dem Spielbrett.

Sie wissen, daß die Speicherzellen im Mikroprozessor eindimensional angeordnet sind (die physikalische Anordnung der einzelnen Flipflops in den integrierten Schaltungen ist wohl dreidimensional, das spielt hier aber keine Rolle). Wichtig ist, daß die Speicherplätze über eine einzige Adresse ausgewählt werden, so wie man einen Punkt auf einer Linie durch die Angabe eines einzigen Wertes, nämlich dem Abstand von irgendeinem Ursprung , angeben kann.



Der Vergleich mit dem Punkt auf der Linie führt auf die Lösung der Aufgabe, die Punkte auf dem zweidimensionalen Bildschirm im eindimensionalen Speicher mit einem einzigen Wert darzustellen. Ganz einfach: man zerschneidet das Spielfeld in seine einzelnen Zeilen und läßt diese Streifen nicht untereinander, sondern legt sie nebeneinander. Die entstandene lange Reihe stellt das Spielfeld in der gewünschten eindimensionalen Anordnung dar. Das 81. Feld in dieser Anordnung entspricht dann also dem ersten Feld der zweiten Zeile in der zweidimensionalen Anordnung des Bildschirms.

Natürlich muß die Ausgabe auf dem Bildschirm wieder eine zweidimensionale Anordnung ergben. Das geschieht praktisch von allein: wenn die 24 mal 80 Feldinhalte einfach einer nach dem anderen auf den Bildschirm geschickt werden, dann kommt automatisch das 81. Feld als erstes in die zweite Zeile, weil das Sichtgerät nach 80 Zeichen von sich aus eine neue Zeile anfängt.

Die Elemente (Felder) des Spielfelds erhalten in der eindimensionalen Darstellung im Speicher z.B. für die linke obere Ecke des Bildschirms folgende Nummern:

```
  1   2   3   4   5   6   7   8   9  10  11  12  13 . . .
 81  82  83  84  85  86  87  88  89  90  91  92 . . . . .
161 162 163 164 165 166 167 168 169 170 171 . . . . .
241 242 243 244 245 246 247 248 . . . . .
321 322 323 324 325 326 . . . . .
401 402 403 404 . . . . .
481 482 483 . . . .
. . . . . .
```

Das Spielfeld für das Life-Game läßt sich also recht einfach mit einem 24 mal 80 Byte großen Speicherbereich darstellen, wobei jedes Byte den Wert Null oder Eins enthält. Dabei wird zwar immer nur eines der acht Bit jedes Bytes ausgenützt, das macht aber nichts, solange genügend Speicherraum vorhanden ist. Die Ausgabe des gespeicherten

Spielfelds auf dem Bildschirm wird ebenfalls recht einfach sein, da für jede Null ein Leerzeichen und für jede Eins ein Stern ausgegeben werden muß.

Schwieriger hingegen ist die Eingabe einer Spielsituation in den Speicher. Es soll dabei möglich sein, den Cursor über den Bildschirm zu bewegen und an beliebigen Stellen Elemente zu setzen oder zu löschen. Auch hier gibt es eine unkomplizierte Alternative: im Betriebssystem gibt es noch den Texteditor, mit dem es ganz einfach wird, ein Spielfeld einzugeben. Das Programm muß dann nur seine Anfangssituation aus dem Textspeicher holen.

Auf der Seite F17 beginnt das Listing zum ersten Programm dieses Heftes. Geben Sie das Programm in Ihr System ein, wobei wie immer die Kommentare nicht unbedingt mit abgetippt werden müssen. Korrigieren Sie Ihre Eingaben, falls sich beim assemblieren nicht die richtigen Prüfsummen ergeben:

```
>asm

Z80-Assembler
pass 1:
+++++++
pass 2:
+++++++

no fatal error(s)


SUM=396E
CRC=A7BA

>
```

Wenn Sie das Programm starten, erscheint auf dem Bildschirm eine willkürliche Felderbelegung, so wie sie im Programm aufgebaut wurde. Weiter passiert vorerst nichts, mit der Eingabe von CTRL-C geschieht ein Warmstart. Zunächst interessiert auch nicht das Life-Game, sondern das Programm selber - schauen Sie es sich an. Es beginnt mit den üblichen Konstantendefinitionen, die Sie inzwischen schon kennen. Lediglich die Definition der Spielfeldgröße ist spezifisch für dieses Programm.

Frage:

1. Im vorliegenden Programm ist lediglich die Systemfunk-

tion 6 zur direkten Ein- und Ausgabe auf der Konsole definiert (und verwendet). Was könnte das für einen Grund haben?

(Die Antwort zu dieser Frage finden Sie auf Seite G7.)

Das Programm erfüllt lediglich den Punkt 1 des "Pflichtenhefts" auf Seite C27. Es holt mit dem Unterprogramm HOLE_FELD eine Aufstellung aus dem Textspeicher und gibt sie mit dem Unterprogramm SCHREIBE_FELD auf dem Bildschirm aus. Anschließend wartet es in einer Schleife auf Tastatureingaben, wobei vorerst nur, wie schon gesagt, CTRL-C für Programmabbruch und Warmstart ausgewertet wird.

Das Unterprogramm HOLE_FELD baut aus dem Inhalt des Textspeichers ein internes Spielfeld auf. Das bedeutet, es muß für jeden Zwischenraum im Textspeicher eine Null und für jeden Stern (oder jedes andere Zeichen) eine Eins ins Spielfeld geschrieben werden. Dabei muß das Unterprogramm damit fertig werden, daß die Zeilen im Textspeicher beliebige Längen haben können, während die Zeilen des Internen Spielfelds immer genau 80 Zeichen lang sind. Es müssen also zwei Fälle berücksichtigt werden:

a) daß die Zeile im Textspeicher länger ist als 80 Zeichchen, dann müssen die überzähligen Zeichen der Zeile ignoriert werden,

b) daß im Textspeicher schon vor Erreichen von Spalte 80 (also vor dem 80. Zeichen) ein CR und LF kommt, dann müssen die restlichen Zeichen der Zeile im internen Spielfeld gelöscht werden.

Außerdem kann der "Text" im Textspeicher schon vor der letzten Zeile zu Ende sein, dann müssen alle restlichen Zeilen des internen Spielfelds gelöscht werden.

Das Unterprogramm holt zunächst zur Vorbereitung den Anfang des Textspeichers ins HL-Registerpaar, die Adresse des internen Spielfelds ins DE-Registerpaar und die Spalten- bzw. Zeilenzahl ins B- bzw. C-Register. An dieser Stelle noch einmal eine Klarstellung, um Verwechslungen zu vermeiden: mit "Feld" ist das gesamte Spielfeld gemeint, die einzelnen Felder des Spielfelds werden als "Elemente" bezeichnet.

Bei HOLE_WEITER wird jeweils ein Zeichen aus dem Textspeicher geholt. Ist es ein LF, so wird es gleich ignoriert, da es genügt, das Zeichen CR auszuwerten (im Textspeicher ist jedes Zeilenende mit CR und LF markiert). Wenn das Zeichen entweder ein Zwischenraum oder ein

druckbares Zeichen ist (z.B.*), dann wird bei SETZE_EL eine Null oder eine Eins ins Spielfeld geschrieben. Danach wird geprüft, ob die Zeile schon voll, also der Spaltenzähler Null ist (er wird von der Spaltenanzahl auf Null heruntergezählt). Ist dies der Fall, die Zeile voll, werden mit HOLE_WEITER einfach weitere Zeichen geholt, bis ein CR oder EOT kommt. Im Normalfall wird bei voller Zeile der Zeiger zum aktuellen Element des Spielfelds inkrementiert und der Spaltenzähler dekrementiert.

Punkt a) von vorher wird demnach einfach dadurch berücksichtigt, daß bei Erreichen des Zeilenendes der Spaltenzähler und der Spielfeldzeiger nicht mehr geändert, sondern festgehalten werden (das hat den kleinen Schönheitsfehler, daß bei SETZE_EL das Zeichen in Spalte 80 immer wieder überschrieben wird). Ist das aus dem Textspeicher kommende Zeichen ein CR, dann wird bei NEUE_ZEILE der Spaltenzähler wieder auf den Anfangswert gesetzt, der Rest der Zeile gelöscht (LOESCHE_ZEILE) und der Zeilenzähler dekrementiert (ZEILE_VOLL). Ist dieser Null, dann ist das Feld voll und es erfolgt ein Rücksprung, ansonsten geht es wie gehabt zu HOLE_WEITER.

Ist das Ende des Textspeichers erreicht (EOF), wird der Textzeiger zurückgesetzt auf die Textende-Markierung EOF (so daß diese bei bei HOLE_WEITER immer wieder gelesen wird), und anschließend NEUE_ZEILE ausgeführt. Es wird also nach NEUE_ZEILE bei HOLE_WEITER wieder EOF gelesen, wieder NEUE_ZEILE ausgeführt usw., bis das Spielfeld voll ist.

Das Unterprogramm SCHREIBE_FELD setzt den Cursor auf den Bildschirmanfang und gibt für jedes Element des internen Spielfelds ein Leerzeichen oder * auf dem Bildschirm aus. Der Zähler im BC-Register wird jedoch nur auf GROESSE-1 gesetzt, weil das letzte Element des Spielfelds nicht ausgegeben werden darf, denn: würde das Element ganz unten rechts auf dem Bildschirm geschrieben, "rutschte" der Cursor in die nächste Zeile und alle Zeilen würden um eine hochgeschoben - die erste Zeile wäre verschwunden!

Das Unterprogramm CURSOR setzt den Cursor mit einer Escape-Sequenz auf die Zeile L und die Spalte H. Da der Programmteil für das Addieren von 20H (ASCII-Code für Leerzeichen) zweimal benötigt wird, wird er einmal als Unterprogramm (CURSOR.1) aufgerufen, das zweite mal direkt durchlaufen.

Das Programm endet mit der Definition des Spielfelds, für das 24 mal 80 (GROESSE) Byte reserviert werden sollen. Eine solche Reservierung von Speicherplatz ist mit dem Pseudobefehl DS, Define Space (=definiere Speicherplatz) möglich. Nach DS muß immer ein Ausdruck stehen, der an-

gibt, wieviele Byte reserviert werden sollen. Wird durch ein Komma getrennt, noch ein zweiter Ausdruck angegeben, dann wird der reservierte Speicherplatz mit dem Wert dieses Ausdrucks gefüllt, der natürlich kleiner als 256 sein muß.

Bei diesem Auffüllen des reservierten Speicherbereichs (in unserem Programm mit dem Wert 0) prüft der Assembler auch gleich, ob der angegebene Speicherbereich überhaupt für den Benutzer verfügbar ist. Aus diesem Grund ist es sinnvoll, die Möglichkeit der Angabe eines zweiten Ausdrucks beim Pseusobefehl DS auszunützen.Wird nämlich lediglich der Speicher reserviert, dann ordnet der Assembler die Adressen zu, ohne zu überprüfen, ob der Platz verfügbar ist.

Zusammenfassung

Der erste Teil des Programms zum Life-Game baut ein internes Spielfeld auf und gibt es auf dem Bildschirm aus. Dabei werden die zwei Dimensionen des Spielfelds durch eine fortlaufende Adressierung im Speicher eindimensional abgelegt.

Das Programm bedient sich der auf den vorangegangenen Seiten kennengelernten Systemfunktionen zur direkten Konsolein/ausgabe und der Escape-Sequenzen für die Sichtgeräte-Steuerung.

Maschinensprache-Prinzipien

Ehe es im Programm zum Life-Game weitergeht, sind noch eine Reihe von Voraussetzungen zu klären, zuerst steht dabei ein Ausflug in die Maschinensprache.

Jeder Befehl eines Computers besteht aus zwei Teilen: einem Operationscode, der angibt, was der Befehl ausführen soll und einem Adreßteil, der die Operanden angibt, mit denen der Befehl ausgeführt wird.

Stellen Sie sich als Gedankenmodell einen einfachen Phantasiecomputer vor, der nur addieren und subtrahieren kann und über acht Register verfügt. Bei ihm könnte die Ausführung eines Maschinenbefehls so aussehen:

Operationscode	Ergebnis	Operand 1	Operand 2
X	E E E	A A A	B B B
Beispiel:			
0	1 1 1	1 0 1	0 1 1
(addiere)	(Reg.7)	(Reg.5)	(Reg. 3)

Der Operationscode besteht nur aus einem Bit X, wobei 0 Addition und 1 Subtraktion bedeuten könnte. Die drei Bits EEE geben das Register (0 bis 7) an, in welches das Ergebnis geschrieben werden sol und die beiden 3-Bit-Gruppen AAA und BBB die Register, die addiert (subtrahiert) werden sollen. Das angegebene Beispiel bedeutet also:

Addiere Register 5 zu Register 3 und bringe das Ergebnis nach Register 7. In unserem Phantasie-Computer benötigt also jeder Maschinenbefehl 10 Bit Speicherplatz.

Dieser Befehlsaufbau des Phantasie-Computers gilt im Prinzip für jede Zentraleinheit, auch für den Z80-Prozessor. Allerdings hat jeder Prozessor eine andere Codierung des Operationscodes, weshalb Programme nicht ohne weiteres zwischen unterschiedlichen Computern austauschbar sind.

Die verschiedenen Operationscodes sind aber nicht der wichtigste Unterschied zwischen den Computern, schwerwiegender sind die Unterschiede in der Art und Weise, wie die Operanden der Befehle adressiert werden.

Für einen Verknüpfungsbefehl (z.B. die Addition zweier Zahlen) braucht man zunächst die Angabe von drei Adressen: die Adressen der beiden Summanden und der Speicherzelle, in welche die Summe gebracht werden soll. Geht man davon aus, daß eine solche Maschine einen Befehlsumfang von 64 Befehlen aufweist (statt nur zwei wie im obigen Beispiel) und dazu noch einen Adreßraum von 64 KByte, dann ergäbe sich für die Codierung eines Additionsbefehls folgender Bedarf: für den Operationscode 6 Bit (die 6. Potenz von zwei ist 64), für die drei notwendigen Adressen werden 3 mal 16 Bit gebraucht, das sind 54 Bit für einen einzigen Befehl!

Damit erforderte das einfachste Programm bereits einen enormen Speicherbedarf -man muß die Sache vereinfachen und einen Prozesser entwerfen, der bei einer Addition einen der beiden Summanden löscht, er wird mit dem Ergebnis der Addition überschrieben. Man spricht in diesem

Fall von einer "Zwei-Adreß-Maschine" im Gegensatz zu der zuvor beschriebenen "Drei-Adreß-Maschine".Die Tatsache, daß mit der Vereinfachung bei dem als Beispiel genannten Additionsbefehl 16 Bit eingespart werden, weist den Weg, wie noch weiterer Programmspeicher gespart werden kann : indem für einen der beiden Summanden und das Ergebnis der Verknüpfung ein internes Arbeitsregister verwendet wird, entsteht eine "Ein-Adreß-Maschine". Das interne Arbeitsregister heißt Akkumulator, in dieses Register wird zuerst ein Summand geladen, dann der zweite addiert und danach des Ergebnis weggespeichert. Man kann auch noch einen Schritt weitergehen und auf den Adreßteil ganz verzichten, wenn man den Speicher völlig als Kellerspeicher (Stack) organisiert. In einer solchen "Null-Adreß-Maschine" beziehen sich alle Operationen auf die Spitze des Kellerspeichers.

Wenn Sie sich einmal alle internen Register der Z80-CPU außer dem A-Register (Akkumulator) wegdenken, dann stellen Sie fest, daß es sich bei dieser CPU im Prinzip um eine Ein-Adreß-Maschine handelt. Wenn man die anderen Register wieder dazu nimmt, so stellt man fest, daß diese praktisch nur eine Erweiterung um einige Speicherstellen sind, auf welche die CPU sehr schnell und einfach zugreifen kann. Bei fast allen Z80-Befehlen befindet sich der eine Operand im A-Register, der andere in einem der zusätzlichen Register der CPU oder in einer Speicherstele im Hauptspeicher. Das Ergebnis wird dann ins A-Register geschrieben. Es gibt allerdings einige Befehle, die das A-Register nicht benötigen und dennoch zwei Operanden haben, das sind also Zwei-Adreß-Befehle (Beispiel: LD B,E ;Lade das B-Register mit dem Inhalt des E-Registers). Solche Zwei-Adreß-Befehle gibt es (mit einer Ausnahme) allerdings nur für die internen Register der CPU, denn diese können mit weniger Aufwand adressiert werden als Speicherstellen im Hauptspeicher. Die Z80-CPU ist also keine reine Ein-Adreß-Maschine.

Zusammenfassung

Die Zusammensetzung der Maschinenbefehle aus Operationscode und Adreßteil ist im Prinzip bei allen Computern gleich, lediglich die Codierung ist unterschiedlich. Um Adreßraum zu sparen, wird das Prinzip der Ein-Adreß-Maschine bzw. der Zwei-Adreß-Maschine angewendet, wobei die internen Register der CPU als schnell und einfach erreichbare Speichererweiterung dienen. Die Z80-CPU ist eine Ein-Adreß-Maschine mit einigen Zwei-Adreß-Befehlen.

Befehlscodierung

Im Fachgebiet PROGRAMMIERUNG haben Sie gesehen, daß der Assembler verschiedene Pseudo-Befehle und den Befehlssatz des Z80-Prozessors erkennt. Nach dem Ausflug in die allgemeinen Prinzipien von Maschinenbefehlen auf den Seiten C32 bis C34 verstehen Sie etwas besser, was der Assembler "zusammensetzt": er setzt aus dem Operationsteil und dem Adreßteil komplette Maschinenbefehle zusammen.

Mit dieser Erkenntnis kann man darangehen, etwas tiefer hinter die Kulissen des Assemblers zu schauen. Geben Sie hierfür die in der unten angegebenen Tabelle links stehenden Befehle mit dem Editor in Ihren Computer ein. Schauen Sie sich vorher nocheinmal irgend ein Programm-Listing auf den Seiten mit der Griffmarke F an, damit Ihnen der nachfolgende Assemblerlauf nicht lauter Fehler bescheinigt. Die erste Spalte jeder Zeile muß freibleiben (einmal Tabulator), an dieser Stelle erwartet der Assembler einen Label und keinen Befehl. Außerdem muß jede Zeile mit Ret (CR) abgeschlossen sein, das gilt bei der Befehlsliste auch für die letzte Zeile.

Rufen Sie nach dem Assemblieren den Tester auf. Sehen Sie sich mit dem Kommando L die assemblierten Maschinenbefehle an, sie stehen in sedezimaler Schreibweise da. Tragen Sie diese in die rechts neben den Befehlen stehende Spalte der untenstehenden Tabelle ein. Anschließend werden die Sedezimaldarstellungen in die binäre Darstellung umgewandelt. Dabei soll Ihnen die nebenstehende Tabelle eine Hilfe sein, welche die Umwandlung für jeweils 4 Bit (1 Sedezimalstelle) auf einmal ermöglicht.

Tabelle C 35

0 = 0000	8 = 1000
1 = 0001	9 = 1001
2 = 0010	A = 1010
3 = 0011	B = 1011
4 = 0100	C = 1100
5 = 0101	D = 1101
6 = 0110	E = 1110
7 = 0111	F = 1111

Assembler-Anweisung	Maschinenbefehl sedezimal	dual	
LD A,A	7F	0111.1111	;Beispiel
LD A,B	__	____.____	
LD A,C	__	____.____	
LD A,D	__	____.____	
LD A,E	__	____.____	
LD A,H	__	____.____	
LD A,L	__	____.____	
LD A,(HL)	__	____.____	
LD B,A	__	____.____	
LD B,B	__	____.____	
	__	____.____	
	__	____.____	
	__	____.____	

Erweitern Sie die Tabelle noch mit einigen weiteren Ladebehlen und markieren Sie dann in der dualen Darstellung diejenigen Bits, die bei allen Ladebefehlen gleich sind. Es ist zu erkennen, daß es die beiden höchsten Bits (7 und 8) sind, die immer die Werte 0 und 1 haben.

Daraus folgt: alle Ladebefehle fangen in der dualen Darstellung des Machinenbefehls mit 01 an. Die Befehle der ersten Gruppe sind bei den drei nachfolgenden Bits alle gleich, nämlich 111. Danach folgen die letzten drei Bits, die anscheinend das Register angeben, dessen Inhalt in den Akkumulator (das A-Register) geladen werden soll.

Prüfen Sie diese Zusammenhänge nach, indem Sie die untenstehende Tabelle vervollständigen, die den Zusammenhang zwischen dem Ursprungsregister des Ladebefehls und den letzten drei Bits des Maschinenbefehls angibt. Prüfen Sie ferner nach, ob diese Codierung auch für das Zielregister des Ladebefehls gilt.

Register	Bitmuster
A	111
B	000
C	___
D	___
E	___
H	___
L	100
(HL)	___

Die Ergebnisse Ihrer Untersuchungen lassen sich zusammenfassen:

Die Register-Ladebefehle bestehen aus dem Bitmuster 01, gefolgt von zwei Bitmustern von je drei Bits, die das Ursprungs- und Zielregister des Ladebefehls codiert angeben.

Registersätze

Aus dem vorangegangenem folgt, daß sich mit drei Bits in der CPU acht verschiedene Register adressieren lassen. Es sind dies die internen Register A, B, C, D, E, H, L und die Speicherstelle im Hauptspeicher, die durch den Inhalt

des Registerpaars HL adressiert wird :(HL). Die durch HL adressierte Speicherstelle im Hauptspeicher kann also wie ein internes Register verwendet werden.

Über die aufgezählten acht Register verfügen auch die Prozessoren 8080 und 8085, von denen die Z80-CPU abgeleitet ist. Bei der Entwicklung der Z80-CPU wollte man zwar mehr Register unterbringen, aber trotzdem den Befehlssatz der alten 8080-CPU verwenden können.

Das Problem war, ein Verfahren zu finden, das mit nur drei Adreßbits mehr als acht Register adressieren kann. Hierzu kann man die Register doppelt aufbauen und einmal die einen und danach die anderen acht Register adressieren.

In der Z80-CPU sind die Register A, B, C, D, E, H, L sowie das Flagregister tatsächlich doppelt vorhanden und man verfügt über doppelt soviele Register wie bei der 8080-CPU. Zugreifen kann man aber immer nur auf den einen Registersatz, der gerade "eingeschaltet" ist. Dieses Einschalten geschieht mit zwei Befehlen: der Befehl

```
EXX
```

schaltet die Adressierung der Register B, C, D, E, H und L um, der Befehl

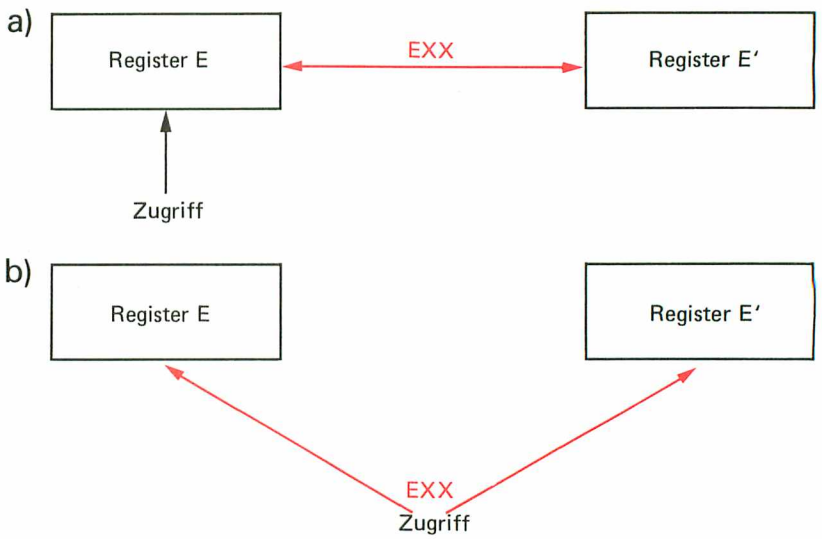
```
EX  AF,AF'
```

tut dies für das A-Register und das Flag-Register. Die Mnemonics EX bzw. EXX stehen für EXCHANGE : tausche aus. Genau das bewirkt eine Umschaltung der Adressierung, nämlich den Austausch der Registerinhalte. Es wird immer zuerst auf den Haupt-Registersatz zugegriffen, während der zweite Registersatz sozusagen im Hintergrund wartet.

Wenn man mit dem zweiten Registersatz arbeiten möchte, wird ein EXX-Befehl ins Programm eingefügt. Durch diesen Befehl wird der zweite mit dem ersten Registersatz ausgetauscht. Danach stehen die vorherigen Inhalte des zweiten Registersatzes im ersten und umgekehrt.

Nach dem vorhergesagten wissen Sie, daß in Wirklichkeit nicht die Inhalte aller Register ausgetauscht werden (das wäre viel zu zeitaufwendig), sondern die Adressierung vom ersten auf den zweiten Registersatz umgeschaltet wird. Ein weiterer EXX-Befehl tauscht die Registersätze wieder zurück.

Bild C 38.1 auf der nächsten Seite zeigt es noch einmal in grafischer Darstellung am Beispiel eines einzelnen Registers, um was es bei den EXX-Befehlen geht : für das



C
38

Bild C 38.1

Die Wirkung der EXX-Befehle: a) vom Programm aus gesehen, b) tatsächlich.

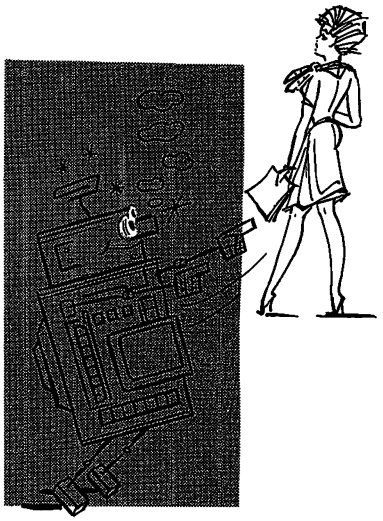
Programm (und damit natürlich für den Programmierer) sieht es so aus, als ob mit einem EXX-Befehl die Inhalte der beiden Register im ersten und zweiten Registersatz ausgetauscht werden ; es greift immer auf das Register im ersten Registersatz zu (Teilbild a).

Teilbild b zeigt, daß in Wirklichkeit nicht die Inhalte der beiden Register umgetauscht werden. Vielmehr greift das Programm einmal auf den ersten, ein andermal auf den zweiten Registersatz zu.

Zusammenfassung

Bei den Register-Ladebefehlen werden durch zwei Bitmuster von jeweils drei Bits die Ziel- und Ursprungsregister angegeben. Mit drei Bits lassen sich acht Register adressieren. Die bei der Z80-CPU erfolgte Erweiterung des Registersatzes geschieht durch einen zweiten internen Registersatz. Mit den EXX-Befehlen kann zwischen den beiden Registersätzen hin- und hergeschaltet werden.

Maschinensprache-Prinzipien



Die (zwei mal) drei Registerpaare der Z80-CPU BC, DE und HL sind nicht gleichwertig. Sie haben gesehen, daß man mit der Speicherstelle, die durch das Registerpaar HL adressiert wird, genauso arbeiten kann wie mit einem internen Register der CPU. Das Registerpaar HL kann als "Pointer" oder "Zeiger" verwendet werden. In beschränktem Umfang gilt dies auch für die beiden Registerpaare BC und DE; es gibt für diese beiden die folgenden Befehle:

```
LD      A,(BC)      ;Lade A mit (BC)
LD      A,(DE)      ;Lade A mit (DE)
LD      (BC),A      ;Speichere A in (BC) (=Lade
                    ;(BC) mit A)
LD      (DE),A      ;Speichere A in (DE) (=Lade
                    ;(DE) mit A)
```

Es sind also Ladebefehle zwischen dem Register A und der durch die Registerpaare BC oder DE adressierten Speicherstelle möglich. Mit dem Registerpaar HL als Pointer funktionieren jedoch alle Lade- und Verknüpfungsbefehle!

Beispiel:

```
        ADD     A,(HL)      ;Addiere (HL) zu A
        CP      (HL)        ;Vergleiche A mit (HL)
        INC     (HL)        ;Inkrementiere (HL)
```

aber

```
L       ADD     A,(BC)      ;geht nicht !!!!!!
```

Mit dem Registerpaar HL läßt sich jedoch noch mehr anfangen, was mit den Registerpaaren BC und DE nicht möglich ist:

- Der Inhalt des Registerpaars HL kann direkt aus dem Speicher geladen und im Speicher abgelegt werden:

```
        LD      HL,(adr)    ;Lade HL mit dem Inhalt der
                            ;Speicherstellen adr und
                            ;adr+1
```

```
LD      (adr),HL    ;Speichere den Inhalt von HL
                    ;in den Speicherstellen adr
                    ;adr+1
                    ;Speicherstellen adr und
```

- Der Inhalt des Registerpaars HL kann als Ziel eines Sprungsbefehls verwendet werden:

```
JP      (HL)        ;Springe an Adresse (HL)
```

- Das Registerpaar HL ist Akkumulator für 16-Bit Additionen:

```
ADD     HL,BC       ;Addiere BC zu HL
ADD     HL,DE       ;Addiere DE zu HL
ADD     HL,HL       ;Addiere HL zu HL (=verdopp-
                    le HL)
ADD     HL,SP       ;Addiere SP zu HL
```

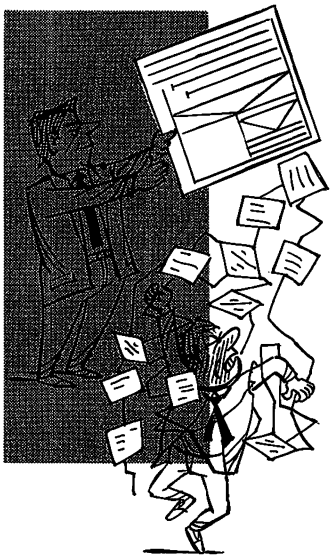
Das Registerpaar HL hat in den 8080-, 8085- und Z80-Prozessoren also eine sehr große Bedeutung (alle oben aufgeführten Befehle für das Registerpaar HL sind keine speziellen Z80-Befehle, sondern auch auf den einfacheren Prozessoren 8080 und 8085 verfügbar). Daher haben die Entwickler der Z80-CPU das Registerpaar HL nicht nur wie die beiden anderen Registerpaare verdoppelt, sondern ihm noch zwei weitere Pointer-Registerpaare an die Seite gestellt, die IX und IY genannt wurden. Diese sind Ihnen bei der Registerausgabe im Tester sicher schon aufgefallen.

Auf den Seiten C 37 und C 38 wurde beschrieben, wie die Anzahl der internen Register erhöht wird durch die Verdopplung des Registersatzes mit den Register-Austauschbefehlen. Diese elegante Lösung macht es möglich, daß Programme für die 8080/8085-Prozessoren ohne Änderungen auch auf dem Z80 laufen.

Es gibt allerdings auch einen Nachteil: beim Programmieren muß darauf geachtet werden, welcher Registersatz gerade aktiv und welcher im Hintergrund ist. Das kann bei Programmen mit vielen Verzweigungen recht kompliziert werden. Das Programm geht unter Umständen viele Wege, um an einer bestimmten Stelle anzukommen und es muß dafür gesorgt werden, daß am Ende eines jeden möglichen Weges auch der richtige Registersatz aktiv ist. Das wiederum

setzt voraus, daß man sich vorher genau überlegt, wie das Programm mit allen seinen Variationen abläuft.

Man kann sich diese Aufgabe erleichtern, wenn man viel Wert darauf legt, das Programm übersichtlich anzulegen, es immer aus einzelnen, überschaubaren Unterprogrammen aufzubauen (die ihrerseits womöglich wiederum aus Unterprogrammen zusammengesetzt sind- schauen Sie sich daraufhin noch einmal das Programm 1 ab Seite F 17 an). Wenn ein Programm dann nur aus Unterprogrammen besteht, die samt und sonders beim Eintritt (Aufruf) und auch beim Austritt (Rücksprung) den ersten Registersatz aktiv haben, dann ist der Überblick auch bei einigen Verzweigungen leicht zu behalten. Dieses Verfahren führt zur sogenannten strukturierten Programmierung, mit der das schreiben von zuverlässigen und vor allem übersichtlichen Programmen wesentlich einfacher ist. Auf dieses Thema kommen wir später noch zurück, zunächst einmal zurück zu den Registern der Z80-CPU.



Der Überblick über die beiden Registersätze ist also bei einer übersichtlichen Programmierung nicht schwer. Nun kommen aber noch zwei Registerpaare, die auch als Pointer verwendbar sind, hinzu : IX und IY. Könnte man auf diese Registerpaare ebenfalls mit Austauschbefehlen wie auf den zweiten Registersatz zugreifen - das gäbe ein schönes Durcheinander. Stellen Sie sich nur vor, Sie tauschen IX mit HL, dann die beiden Registersätze, dann IY mit HL - wo ist dann welcher Registerinhalt?

Nun, zum Glück werden die Pointer IX und IY anders behandelt; sie haben zum eigenen Namen auch eigene Befehle bekommen. Das heißt, daß die Registerpaare IX und IY genau wie das Registerpaar HL verwendet werden können. Es gibt also zu (fast) jedem Befehl bezüglich des Pointers HL entsprechende Befehle für die Pointer IX und IY.

Beispiele:

```
LD      IX,(adr)    ;Lade IX mit dem Inhalt der
                    ;Speicherstellen adr und
                    adr+1
LD      (adr),IY    ;Speicher den Inhalt von IY
                    in den Speicherstellen adr
                    und adr+1
JP      (IX)        ;Springe an Adresse (IX)

ADD     IX,BC       ;Addiere BC zu IX
ADD     IX,DE       ;Addiere DE zu IX
ADD     IY,IY       ;AddiereIY zu IY (=ver-
                    ;dopple IY)
ADD     IX,SP       ;Addiere SP zu IX
```

An dieser Stelle sollen Sie wieder einmal selber auf einen Dreh kommen: geben Sie einige der vorgenannten Befehle jeweils für das Registerpaar (oder den Pointer) HL, IX und IY mit dem Editor in Ihr System ein und assemblieren Sie diese Befehle. Prüfen Sie die erzeugten Operationscodes mit Hilfe des Testers (Kommando LIST). Stellen Sie fest, ehe Sie weiterlesen, wie sich diese Operationscodes für die den einzelnen Pointern zugeordneten Befehle unterscheiden.

Die Auflistung zeigt es: dem Operationscode der auf HL bezogenen Befehle ist das Byte 0DDH vorangestellt, wenn der Befehl sich auf den IX-Pointer bezieht. Beim IY-Pointer steht das Byte 0FDH vor dem entsprechenden HL-Befehl. Demnach stimmt es also doch nicht so ganz, daß die Pointer IX und IY eigene Befehle bekommen hätten. Es handelt sich nach wie vor um die alten Befehle der 8080-CPU, es wird lediglich vom HL-Pointer durch einen Präfix 0DDH auf den IX-Pointer und durch einen Präfix 0FDH auf den IY-Pointer umgeschaltet. Wichtig ist: diese Umschaltung ist im Operationscode des jeweiligen Befehls enthalten, sie gilt immer nur für diesen einen Befehl und ist nicht dauerhaft wie die Registersatz-Umschaltung mit EXX.

Eigentlich ist diese Umschaltung ein Maschinen-Geheimnis, von dem der Benutzer nichts zu wissen braucht. Der Assembler kennt ja die Befehle und übersetzt sie immer richtig. Es ist dennoch recht nützlich, von diesen Tricks der internen Verarbeitung eine Ahnung zu haben, denn auf diese Weise läßt sich zum Beispiel eine Merkwürdigkeit bei den Pointer-Befehlen erklären. Man kann sich fragen, warum es den Befehl

```
ADD     IY,IY       ;Addiere IY zu IY
```

gibt, die Befehle

```
ADD     IY,HL       ;Addiere HL zu IY
ADD     IY,IX       ;Addiere IX zu IY
```

jedoch nicht existieren? Wenn Sie sich den Aufbau der Befehle vor Augen führen, ist das völlig klar. Wird in einem HL-Befehl mit dem Präfix 0FDH auf IY umgeschaltet, dann wird im gesamten Befehl HL durch IY ersetzt. Woher sollte die CPU auch wissen, daß sie einmal als Summand IX oder HL, für die Summe aber IY nehmen soll?

Sie haben bislang nur einen Teil der Befehle für die beiden zusätzlichen Registerpaare untersucht. Wir nannten sie meist Pointer, und jetzt fehlen noch alle 8-Bit-Befehle, die IX und IY tatsächlich als Pointer (Zeiger) benutzen. Geben Sie noch einige solche Befehle mit Hilfe des Editors in Ihr System ein. Zum Beispiel die folgenden

Befehle, wobei Sie aber nicht vergessen dürfen, zum Vergleich auch die entsprechenden Befehle für den Pointer HL einzugeben:

```
LD      E,(IY)      ;Lade Register E mit (IY)
ADD     A,(IX)      ;Addiere (IX) zu A
CP      (IY)        ;Vergleiche A mit (IY)
INC     (IX)        ;Inkrementiere (IX)
```

Assemblieren Sie die eingegebenen Befehle und untersuchen Sie den erzeugten Operationscode mit Hilfe des Testers. Sicher fällt Ihnen auf, daß bei den Befehlen für IY und IY außer dem schon bekannten Präfix noch ein Byte 00 am Ende der Befehle dazugekommen ist.

Die Befehle, bei denen das IX- oder IY-16-Bit-Register als Pointer verwendet wird, haben also nicht nur den Präfix 0DDH bzw. 0FDH, sondern auch noch einen Suffix 00H. Dahinter verbirgt sich eine angenehme zusätzliche Möglichkeit: man kann mit diesen Befehlen nicht nur auf die Speicherstelle zugreifen, auf die der Pointer zeigt, sondern auf alle Speicherstellen bis 128 Stellen vor und 127 Stellen nach der Adresse, auf die das Pointer-Register zeigt.

Dieser Abstand, man spricht von einem Offset, zum Inhalt des Pointers steht im letzten Byte der eben von Ihnen untersuchten Befehle. Da zunächst in keinem der Befehle ein Abstand angegeben war, hat der Assembler an dieser Stelle immer 00H eingetragen.

Sie können die zuletzt eingegebenen Befehle so abändern, daß Sie einen Offset mit angeben, z.B. IY durch IY+3 ersetzen. Für einen Offset von 1 ergibt sich das Byte 01H, für -1 ergibt sich 0FFH, für +127 ergibt sich 7FH und für -128 ergibt sich 80H. Der Offset ist demnach als vorzeichenbehaftete 8-Bit-Dualzahl im Zweierkomplement aufzufassen. Nachstehend ein paar Beispiele für Offsetangaben:

```
LD      E,(IY+3)    ;Lade Register E mit (IY+3)
ADD     A,(IX-128)  ;Addiere (IX-128) zu A
CP      (IY+127)    ;Vergleiche A mit (IY+127)
INC     (IX-1)      ;Inkrementiere (IX-1)
```

Zusammenfassung

Bei der Z80-CPU gibt es außer dem HL-Register noch zwei weitere Pointer-Register IX und IY. Bei ihrer Verwendung kann ein Offset zwischen +127 und -128 angegeben werden.

Life-Game, 2.Teil

Im ersten Teil war es mit dem fertiggestellten Programmteil möglich, eine Spielfiguration in das interne Spielfeld einzugeben und auf dem Bildschirm auszugeben. Jetzt steht noch die eigentliche Aufgabe des Programms an, nämlich neue Generationen auszurechnen. Dazu muß für jedes Element geprüft werden, ob es selbst belegt ist und wieviele seiner Nachbarfelder belegt sind. Entsprechend den Spielregeln ist dann der Wert einzusetzen, den das Element in der nächsten Generation hat.

Dabei darf aber, wie beim Spiel von Hand , das alte Spielfeld nicht verändert werden, solange das neue nicht fertig ausgerechnet ist. Es darf beispielsweise nicht ein Element in der ersten Zeile sofort auf seinen neuen Wert gesetzt werden. Bei der Untersuchung der zweiten Zeile würden bei der Abfrage, ob die Nachbarn "oben" schon belegt sind, bereits die Werte der nächsten Generation verwendet werden. Das bedeutet, in der zweiten Zeile entstünden fehlerhafte Ergebnisse.

Wir hatten festgestellt, daß es beim Spiel von Hand zwei Möglichkeiten gibt, dieses Problem zu umgehen: entweder muß man zwei Spielfelder verwenden, oder Steine in zwei Farben nehmen. Diese beiden Varianten lassen sich auch im Computer realisieren. Man kann statt verschiedenfarbiger Steine verschiedene Bits der Elemente des Feldspeichers für verschieden Generationen verwenden. Wenn z.B. jeweils das zweite Bit gesetzt wird, um die Belegung eines Elementes in der nächsten Generation festzuhalten, dann kann unabhängig davon das erste Bit für das ganze Spielfeld richtig ausgewertet werden.Diese Methode erfordert mit Sicherheit mehr Programmieraufwand und Rechenzeit als der einfachere Weg, zwei Spielfelder für zwei verschiedene Generationen zu verwenden. Genügend Speicher ist ohnehin vorhanden und außerdem verfügt der Prozessor, wie Sie auf den vorhergehenden Seiten gesehen haben, über einige Pointer-Registerpaare, um bequem in zwei Spielfeldern gleichzeitig zu operieren.

Gehen wir also von zwei Spielfeldern aus, einem Ausgangsfeld und einem Zielfeld (Ergebnisfeld). Der Pointer IX wird z.B. über das Ausgangsfeld und der Pointer IY über das Zielfeld bewegt. Um spätere Generationen zu berechnen, müssen dann nur die Anfangswerte der Pointer vertauscht werden :IX läuft dann wieder über das Ausgangsfeld und IY über das Ergebnisfeld, aber: dieses Ausgangsfeld war ja das Ergebnisfeld der vorhergegangenen Generation und wird in der nächsten Generation wieder Ergebnisfeld sein und so geht das weiter. Wie sich die Pointer über die beiden Spielfelder bewegen, läßt sich besser er-

klären, wenn (wie auf Seite C28) man einen Teil der Numerierung der beiden Spielfelder vor Augen hat:

```
  1   2   3   4   5   6   7   8   9  10  11  12  13 . . .
 81  82  83  84  85  86  87  88  89  90  91  92 . . . . .
161 162 163 164 165 166 167 168 169 170 171 . . . . .
241 242 243 244 245 246 247 248 . . . . .
. . . . . .
```

Angenommen, beide Pointer zeigen auf das dritte Element der zweiten Zeile des jeweiligen internen Spielfelds. Ob dieses Element belegt ist, läßt sich in dieser Situation leicht feststellen mit dem Befehl:

```
LD    A,(IX)
```

Wie sieht es nun mit den Nachbarn aus? Kein Problem gibt es mit dem linken und rechten Nachbarn. Das als Beispiel genommene Feld hat die Nummer 83, der linke Nachbar demnach die Nummer 82 und der rechte die Nummer 84. Da der Pointer auf Nummer 83 steht, lassen sich beide Nachbarn auch ganz einfach ins A-Register laden bzw. gleich aufaddieren, da ja nur die Anzahl der belegten Nachbarn interessiert:

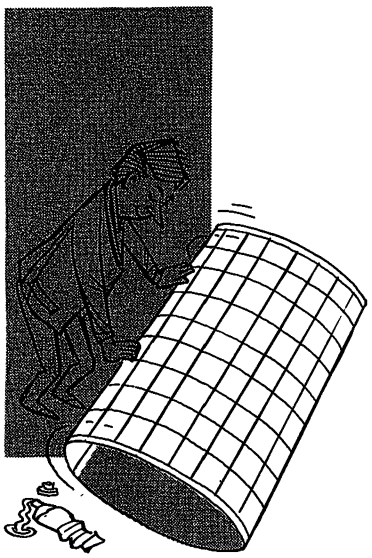
```
LD    A,(IX-1)
ADD   A,(IX+1)
```

Genau so einfach verläuft die Abfrage der anderen Nachbarn: derjenige direkt über dem Element Nummer 83 hat die Nummer 3, der darunter die Nummer 163. das aber ist 83-80 bzw. 83+80, also:

```
ADD   A,(IX-80)
ADD   A,(IX+80)
```

Jetzt fehlen noch die seitlich diagonalen Nachbarn, aber auch diese sind mit einer Offset-Angabe zu erreichen. Die Größe der Abstände ist eine einfache Rechenaufgabe, überlegen Sie kurz, ehe Sie weiterlesen!

Man kann von den oberen und unteren Nachbarn ausgehen und jeweils eine Nunmmer dazuzählen oder abziehen, man kann auch denkfaul einfach nachzählen: -81 für links oben, - 79 für rechts oben, +79 für links unten und +81 für rechts unten.Mit den beiden Pointer-Registern und der Offsetangabe lassen sich also die Nachbarn eines Elementes auf einfache Weise untersuchen, ein Problem bleibt aber: die Ränder des Spielfelds. Betrachten Sie als Beispiel das Element mit der Nummer 81. Es ist das erste Element der zweiten Zeile, deshalb wird mit dem Offset -1 nicht sein linker Nachbar, sondern das letzte Element der ersten Zeile (Nummer 80) angesprochen (welches in der



"eindimensionalen" Art der Adressierung jedenfalls sein linker Nachbar ist).Im ganzen gesehen führt das praktisch zu einem Spielfeld, bei dem die linke Seite mit der rechten verbunden ist; als ob das Spielfeld zusammengerollt und hinten zusammengeklebt wäre. Das ist aber nicht so schlimm, solange die uns interessierenden Dinge sich nicht gerade am äussersten Rand abspielen, was in in der Praxis eigentlich nicht vorkommt.Das größere Problem ist der obere und untere Rand. Befindet sich der Pointer z.B. noch in der ersten Zeile, dann wird mit dem Offset -80 eine Speicherstelle angesprochen, die garnicht zum Spielfeld gehört, sondern einem Element aus dem Spielfeld der vorangegangenen Generation zugeordnet ist.Auch hier gibt es eine einfache Lösung: es werden mindestens 81 Speicherstellen vor und nach dem Spielfeld auf Null gesetzt. Damit ist der Bereich oberhalb und unterhalb des Spielfelds grundsätzlich "tot".

Man kann die Belegung des gerade untersuchten Elements zusammen mit der Anzahl seiner belegten Nachbarn mit einigen bedingten Sprungbefehlen auswerten, um die Belegung des entsprechenden Elements in der nächsten Generation auszurechnen. Eine andere Möglichkeit besteht darin, daß man sich für alle Kombinationen eine Regeltabelle aufstellt und in dieser nachschaut, wie das Element in der nächsten Generation auszusehen hat. Diese Methode haben wir vorgezogen, weil sich damit die Regeln im Bedarfsfall leicht ändern lassen -es wird dann nur die Tabelle geändert.Geben Sie jetzt erst einmal das Programm (Seite F21) ein und speichern es ab. Auch hier gilt wieder, daß die Kommentare selbsverständlich nicht mit eingegeben werden müssen.Nach den Assemblerläufen steht auf dem Bildschirm die gewohnte Darstellung:

```
>asm


Z80-Assembler
pass 1:
++++++++++++
pass 2:
++++++++++++
no fatal error(s)

SUM=6E3A
CRC=54D4
```

Wenn die Prüfsummen stimmen, können Sie zur Abwechslung erst einmal mit dem Life-Game spielen. Nach dem Programmstart erscheint eine Konfiguration auf dem Bildschirm, die sich nach den Spielregeln des Life-Game mit jeder Betätigung der CR- (RET-) Taste verändert. Sie

werden erleben, wie die auf den Seiten C22 und C23 gezeigten Konfigurationen entstehen.

Frage:

1. Nach dem Programmstart erscheint eine Konfiguration auf dem Bildschirm - was für eine ? Anders gefragt: woher nimmt das Programm die Daten für dieses erste interne Feld?

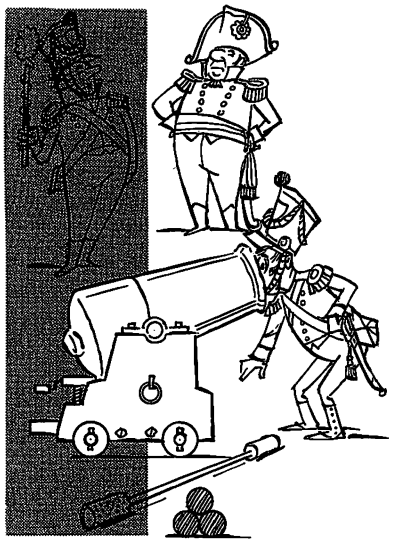
(Die Antwort zu dieser Frage finden Sie auf Seite G7.)

Schauen Sie sich das Programm an, wenn Sie genug gespielt haben. Es enthält die aus dem ersten Programm bekannten Unterprogramme HOLE_FELD und SCHREIBE_FELD, in denen lediglich zu Beginn wegen der jetzt vorhandenen zwei Spielfelder das Laden der Spielfeldadresse anders abläuft. Es wird nicht mehr die Spielfeldadresse direkt geladen, sondern die bei VON_FELD stehende Adresse des aktuellen Ausgangsfelds. Die Labels VON_FELD und NACH_FELD sind mit Pseudobefehlen DW definiert. DW steht für Definiere Wort (Define Word). Mit diesem Pseudobefehl wird also ein Wort (das sind 2 Byte) definiert, wie mit dem Pseudobefehl DB ein einzelnes Byte. Im vorliegenden Fall werden vom Assembler bei den Labels VON_FELD und NACH_FELD die Adressen der beiden Spielfelder eingetragen, die am Ende des Programms definiert und wegen der zuvor erwähnten Randprobleme jeweils mit einem Vor- und Nachspann von Nullen versehen sind.

Neu hinzugekommen ist das Unterprogramm GENERATION, das eine neue Generation berechnet. Am Anfang dieses Unterprogramms werden die Pointer-Register geladen, die Werte der Variablen VON_FELD und NACH_FELD vertauscht für die nachfolgende Generation und der Zähler mit der Feldgröße gesetzt. Auch hier gilt wieder die Einschränkung, daß das allerletzte Element nicht bearbeitet werden kann, deshalb wird derZähler auf GROESSE-1 gesetzt. Anschließend läuft das Unterprogramm in einer Schleife, in der Element für Element mit Hilfe von drei weiteren Unterprogrammen geprüft wird und sein Wert für die nächste Generation berechnetundfestgehaltenwird.

Im Unterprogramm PRUEFE_ELEMENT wird das aktuelle Element geladen, mit 16 multipliziert (genau genommen viermal zu sich selbst addiert: 1+1=2, 2+2=4, 4+4=8, 8+8=16) und die Werte aller Nachbarn addiert. Das ergibt z.B. bei drei besetzten Nachbarn zum Wert 3, wenn das aktuelle Element leer war, zum Wert 16+3=19, wenn es belegt war. Dieser Wert wird ins DE-Registerpaar geladen, ins HL-Register-

paar die Adresse der Regeltabelle und danach werden beide addiert. Das HL-Registerpaar zeigt dann in den beiden Beispielfällen auf das 3. bzw. 19. Byte der Regeltabelle. An dieser Stelle steht nun eine Null oder eine 1, je nachdem, ob das Element (laut Regeln) in der nächsen Generation belegt sein soll oder nicht.Die Unterprogramme SETZE_ELEMENT und SCHREIBE_ELEMENT geben den Wert ins Ergebnis-Feld bzw. als Leerzeichen oder Stern auf den Bildschirm.



Zum Schluß noch die auf Seite C23 versprochene "Kanone", die nach jeweils dreißig Generationen wieder ihre ursprüngliche Gestalt annimmt und außerdem einen "Glider" produziert. Um die Eingabe leichter zu machen, sind die nicht belegten Elemente mit einem Punkt statt eins Leerzeichens markiert. Sie sollten nämlich die Kanone genau gleich aufbauen und vor allem an gleicher Stelle plazieren, und dabei lassen sich Punkte besser abzählen als leerer Raum. Bezugspunkt ist der oberste Stern, er steht auf Zeile 14, Spalte 23:

```
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
.....................*........................
.....................**.......................
................**....**.............*.........
............**..**....***............*.*.......
............**..**....**.............*.*.......
.....................**.............*..*...**...
.....................*...............*.*....**...
.....................................*.*..........
.....................................*............
..................................................
..................................................
```

Frage:

1. Wie stellen Sie es an, daß die oben gezeigte Kanone als erste Konfiguration in das interne Feld kommt?

(Die Antwort zu dieser Frage steht auf Seite G8.)

Life-Game, Verbesserungen und Änderungen

Am Life-Game-Programm läßt sich manches verbessern, beispielsweise könnte der Wunsch auftauchen, daß man sich das Spielfeld ausdrucken läßt. In diesem Fall wird das Programm erweitert um ein Unterprogramm DRUCKE_FELD, dessen Aufbau sich weitgehend an dem Unterprogramm SCHREIBE_FELD orientiert. Man muß nur daran denken, daß zusätzlich nach jeder Zeile ein CR und ein LF an den Drucker geschickt wird. Die gesamte Programmergänzung sieht dann so aus:

bei Definition von Systemfunktionen einfügen:

```
DRUCKF  EQU     5
```

im Hauptprogramm nach "JP Z,WSTART" einfügen:

```
        CP      'P' AND CONTROL         ;Control-P = Drucken
        CALL    Z,DRUCKE_FELD
```

Unterprogramm:

```
DRUCKE_FELD:
        LD      HL,(VON_FELD)
        LD      B,ZEILEN
DRUCKE_CRLF:
        CALL    CR_LF
        LD      C,SPALTEN
DRUCKE_NORMAL:
        LD      E,'*'
        LD      A,(HL)
        INC     HL
        OR      A
        JR      NZ,D_NICHT_LEER
        LD      E,' '
D_NICHT_LEER:
        CALL    DRUCKE
        DEC     C
        JR      NZ,DRUCKE_NORMAL
        DEC     B
        JR      NZ,DRUCKE_CRLF
        CALL    CR_LF
        LD      A,0                     ;damit in A nicht CR!
        RET
CR_LF:
        LD      E,CR
        CALL    DRUCKE
        LD      E,LF
```

```
DRUCKE:
        PUSH    BC
        PUSH    HL
        LD      C,DRUCKF
        CALL    SYSTEM
        POP     HL
        POP     BC
        RET
```

C

50

Frage:

1. Wie muß das Unterprogramm DRUCKE_FELD abgeändert werden, damit die Zeichen nicht an die Druckersystemfunktion übergeben werden, sondern in den Textspeicher geschrieben und von dort ausgegeben werden?

(Die Lösung dieser kleinen Programmieraufgabe sollten Sie auf Seite G8 wirklich erst dann sich anschauen, wenn Sie Ihre eigenen Ideen zu Papier gebracht haben !)

Mit dieser Programmvariante lassen sich Spielfelder wieder in den Textspeicher zurückschreiben und auf Cassette speichern oder auch ausdrucken oder nur einfach ändern und weiterspielen.

Außerdem ist es möglich, dieses Unterprogramm automatisch vor jedem Warmstart aufzurufen. Dann muß aber darauf geachtet werden, daß genügend Textspeicherraum resserviert wird, weil sonst unter Umständen das Betriebsprogramm überschrieben werden kann. Wenn das Programm in dieser Hinsicht perfekt sein soll, dann kann man mit der Systemfunktion 249 die höchste Adresse abfragen, die für den Textspeicher zur Verfügung steht und daraufhin das Einschreiben in den Textspeicher abbrechen, falls diese Adresse erreicht wird. Die Systemfunktion sieht so aus:

Funktion 249 - Hole Textspeicherende

Parameter beim Aufruf: Register C: 0F9H

Parameter beim Rücksprung: Registerpaar HL: maximaler Textspeicher

Die Systemfunktion 249 lädt die größte für den Textspeicher zur Verfügung stehende Adresse ins Registerpaar HL.

Zurück zum Life-Game-Programm. Hier gibt es noch vielfältige Verbesserungsmöglichkeiten. Ein Hauptnachteil des Programms ist die recht langsame Bildschirmausgabe.

Frage:

1. Mit welchen Programmänderungen läßt sich die Bildschirmausgabe beschleunigen?

Die Antwort auf diese Frage steht nicht Auf Seite G .., sondern diesmal im folgenden Heft 3 des Lehrgangs, damit Sie tatsächlich mal ranmüssen. Wir lassen Sie aber nicht ganz im Regen stehen, daher noch einige Tips für die Arbeit mit dem Programm:

Auf dem Bildschirm müßte nicht immer das ganze Feld ausgegeben werden. Das Unterprogramm SCHREIBE_ELEMENT könnte Elemente, die sich nicht geändert haben, einfach auf dem Bildschirm stehen lassen und nur die tatsächlich geänderten Elemente ausgeben. In diesem Fall müssen aber die Spalten und Zeilen mitgezählt werden, damit der Cursor vor einer Ausgabe mit CURSOR wieder auf die richtige Position gesetzt werden kann, wenn Ausgaben ausgelassen wurden. Ob sich ein Element geändert hat, kann leicht durch einem Vergleich von (IX) mit (IY) festgestellt werden.

Es wird auch viel Zeit verbraucht beim Aufaddieren der Anzahl der belegten Nachbarelemente. Mit einer Änderung des Programmkerns läßt sich ein großer Teil dieser Zeit sparen. Man kann beim Aufbau eines (vorher gelöschten) Spielfelds zu den Elementen, die belegt sein sollen, jeweils 16 (10H) addieren und die jeweiligen Nachbarn inkrementieren. Das Unterprogramm SETZE_ELEMENT sieht dann so aus:

```
SETZE_ELEMENT:
        OR      A
        RET     Z               ;A=0: schon fertig!
        LD      A,10H
        ADD     A,(IY)
        LD      (IY),A
        INC     (IY-81)
        INC     (IY-80)
        INC     (IY-79)
        INC     (IY-1)
        INC     (IY+1)
```

```
        INC     (IY+79)
        INC     (IY+80)
        INC     (IY+81)
        RET
```

Jedes Element enthält am Schluß automatisch 10H, falls es belegt ist, plus der Anzahl seiner Nachbarn. Dabei werden die zeitaufwendigen Operationen nur noch bei den Feldern durchgeführt, die tatsächlich belegt sind. Das Unterprogramm PRUEFE_ELEMENT schrumpft stark:


```
PRUEFE_ELEMENT:
        LD      E,(IX)
        LD      D,0
        LD      HL,REGELN
        ADD     HL,DE
        LD      A,(HL)
        RET
```

Für den Umbau des Programms sind aber noch mehr Änderungen notwendig. In den Unterprogrammen SCHREIBE_ELEMENT, SCHREIBE_FELD und DRUCKE_FELD darf der Inhalt eines Elements nicht mehr auf Null abgefragt werden, um festzustellen, ob es belegt ist. Vielmehr muß es mit 10H UND-verknüpft werden. Es ist also dreimal "OR A" durch "AND 10H" zu ersetzten.

Im Unterprogramm GENERATION muß nach "CALL SCHREIBE_ELEMENT" eingefügt werden "LD (IX),0",. Damit wird das alte Ausgangsfeld gelöscht, weil es ja beim nächsten Durchgang zum Ergebnisfeld wird, das gelöscht sein muß.

Das Unterprogramm HOLE_FELD muß so abgeändert werden, daß es den IY-Pointer statt des Registerpaars DE zum Feldaufbau verwendet, das Unterprogramm SETZE_ELEMENT zum besetzten der belegten Elemente benutzt und die nicht belegten Elemente nicht mehr löscht, damit auch das erste Ausgangsfeld korrekt aufgebaut ist.

Die angedeuteten Änderungen wirken sich erst aus, wenn die als erste erwähnte schnellere Bildschirmausgabe funktioniert.

Wir wünschen Ihnen guten Erfolg und viel Spaß beim Ergänzen und Verbessern des Programms!

Verbessertes Life-Game-Programm

Wir hoffen, daß Sie sich mit Erfolg an die Verbesserung des Life-Game-Programms gemacht haben, wobei das Ergebnis die Erhöhung der Geschwindigkeit oder des Bedienungskomforts sein kann. Wie am Schluß des letzten Hefts erwähnt, kann die Geschwindigkeit des Spielablaufs erheblich gesteigert werden, wenn der Algorithmus zur Berechnung des neuen Spielfelds verbessert und gleichzeitig die Ausgabe dieses neuen Spielfelds in der Art optimiert wird, daß diejenigen Elemente, die sich nicht geändert haben, auch nicht auf dem Bildschirm ausgegeben werden.

Wir haben in unserer verbesserten Version nur diese Erhöhung der Geschwindigkeit angestrebt. Wenn Sie sich das ab Seite F27 aufgelistete Programm ansehen und es mit dem Life-Game-Programm ab Seite F21 im Heft 2 vergleichen, dann werden Sie eine Reihe von Veränderungen feststellen.

Da sind zunächst die Unterprogramme SETZE_ELEMENT und PRUEFE_ELEMENT, die bereits im Heft 2 bei der Beschreibung eines verbesserten Algorithmus zur Berechnung des neuen Spielfelds aufgelistet wurden. Es wird viel Rechenzeit gespart, wenn bereits beim Belegen der Elemente die Nachbarelemente inkrementiert werden -sozusagen als Nachricht, daß sie einen neuen Nachbarn bekommen haben.

Damit wird nur bei den Elementen, die tatsächlich belegt werden, Rechenzeit gebraucht. In der ursprünglichen Version hingegen mußten alle Nachbarn eines jeden Elements aufaddiert werden. In der neuen Version enthält jedes Element gleich zwei Informationen: ob es belegt ist und wie viele seiner Nachbarn belegt sind. Das entsprechende Ergebnis muß dann nur noch in der Regeltabelle nachgeschlagen werden.

Beim Vergleich der beiden Versionen werden Sie auch die bereits auf Seite C 52 angeführten Änderungen finden, nämlich das erheblich kürzer gewordene Unterprogramm PRUEFE_ELEMENT sowie die Änderungen in den Unterprogrammen GENERATION, HOLE_FELD, SCHREIBE_ELEMENT und SCHREIBE_FELD.

Doch nun zu Ihrer eigentlichen Aufgabe, der Optimierung der Ausgabe. Wichtig ist dabei, daß das Programm in einem Zähler verfolgt, welche Spalte und Zeile des Spielfelds gerade bearbeitet wird; damit kann der Cursor dann vor einer Ausgabe wieder richtig gesetzt werden, falls einige Elemente sich nicht geändert hatten und daher keine Ausgabe erfolgte. Letzteres muß sich das Programm ebenfalls

merken, wenn es also Elemente bei der Ausgabe ausgelassen hat. Denn, es muß ja wissen, wann es ein Element einfach ausgeben kann und wann es zunächst den Cursor richtig setzen muß.

Im verbesserten Programm findet sich ein Zähler (im DE'-Registerpaar), der bei jeder Ausgabe auf Null gesetzt und bei jeder ausgelassenen Ausgabe inkrementiert wird. Steht dieser Zähler dann bei einer Ausgabe auf Null, so ist das vorhergehende Element auch ausgegeben worden, der Cursor steht also richtig und die Ausgabe kann sofort erfolgen. Steht der Zähler auf Eins, dann ist nur ein Element ausgelassen worden und der Cursor wird korrigiert, indem das ASCII-Steuerzeichen für Cursor nach rechts (09H) ausgegeben wird. Ist der Zählerstand größer, wird der Cursor mit der im letzten Heft (Seite B 25) beschriebenen Escape-Sequenz auf die richtige Stelle gesetzt.

Zweiter Registersatz

Im Unterprogramm GENERATION sehen Sie, daß für die Zähler für Zeile und Spalte (HL'-Registerpaar) und für ausgelassene Zeichen Register aus dem zweiten Registersatz benutzt werden. Um die Übersicht etwas besser wahren zu können, sind im Assembler-Listing alle die Befehle etwas eingerückt, die sich des zweiten Registersatzes bedienen.

Es wurde bereits im zweiten Heft (Seiten C 40 und C 41) gesagt, daß die Verwendung des zweiten Registersatzes nicht ungefährlich ist und es wurden auch Hinweise gegeben, was dabei schief gehen kann: wenn man z.B. die Übersicht über die Belegung der einzelnen Register verliert oder nicht mehr durchblickt, welcher Registersatz gerade aktiviert ist.

Wie gefährlich solche Programmierfehler sind, soll Ihnen das vorliegende, verbesserte Life-Game-Programm zeigen. Es enthält nämlich einen schlimmen Programmierfehler, der zu allem Überfluß noch nicht einmal vordergründig in Erscheinung tritt! Das Programm funktioniert scheinbar einwandfrei, obwohl es an einer Stelle überhaupt nicht das tut, was es tun sollte.

Frage:

1. Im Heft 2 wurde ein Tip gegeben, wie man die Übersicht bei der Verwendung des zweiten Registersatzes besser

wahren kann. Gegen diese selbstgemachte Regel verstößt das Programm an einer Stelle. Versuchen Sie, den Fehler zu finden und überlegen Sie, was an dieser Stelle im Programmablauf geschieht. Erforschen Sie dies mit Hilfe des Testers im Einzelschritt oder mit Programmunterbrechungen.

Die Antwort zu dieser Frage steht nicht auf Seite G.., sondern im anschließenden Text. Seien Sie hart gegenüber der Versuchung, weiterzulesen und nehmen Sie sich das Programmlisting vor!

Im Unterprogramm SCHREIBE_ELEMENT wird bei entsprechendem Stand des Zählers für ausgelassene Zeichen der Cursor neu gesetzt, betrachten Sie beim Label NEU_CURSOR, was geschieht: mit EXX wird der zweite Registersatz aktiviert und das Unterprogramm CURSOR aufgerufen, das den Cursor auf Spalte H und Zeile L setzt.

Wir haben hier gegen die auf Seite C 41 (Heft 2) genannte Regel verstoßen, die besagt, daß Unterprogramme immer mit dem ersten Registersatz begonnen und verlassen werden: das Unterprogramm CURSOR wird im vorliegenden Programm mit dem zweiten Registersatz aufgerufen und durchlaufen. Dabei wird der Inhalt des Zählers für ausgelassene Zeichen (DE'-Registerpaar) zerstört.

Wie könnte dieser Fehler behoben werden? -es bieten sich zwei Varianten an: ohne viel Überlegung kann man den Inhalt des DE'-Registerpaars vor Aufruf des Unterprogramms auf den Stack retten und danach wieder vom Stack holen. Damit wird zwar die Zerstörung des Zählers für ausgelassene Zeichen verhindert, aber diese Lösung ist doch nicht sehr schön, vor allem auch deshalb, weil sie an dem Grundübel nichts ändert: das Unterprogramm CURSOR wird nach wie vor im zweiten Registersatz durchlaufen.

Mit etwas Nachdenken findet sich der elegantere Weg: der Inhalt des HL'-Registerpaars kann mit dem Umweg über den Stack vor dem Aufruf des Unterprogramms ins HL-Registerpaar gebracht werden. Dann wird das Unterprogramm CURSOR mit dem richtigen (ersten) Registersatz durchlaufen:

```
NEU_CURSOR:

   EXX
     PUSH  HL
   EXX
   POP     HL
   CALL    CURSOR
```

Zusammenfassung

Das verbesserte Life-Game-Programm bringt einen schnelleren Spielablauf. Dies wird erreicht mit einem durchdachteren Algorithmus zur Berechnung des neuen Spielfelds und eine optimierte Ausgabe. Dabei wird der zweite Registersatz mit benützt, wobei die in den vorangehenden Seiten aufgestellten Regeln beachtet werden müssen, damit keine Programmierfehler entstehen.

Frage:

1. Was hat den nun der "eingebaute" Programmierfehler eigentlich im Programmablauf bewirkt, und vor allem, warum hat man ihn nicht bemerkt?

(Die Antwort zu dieser Frage finden Sie auf Seite G 9.)

Rechengeschwindigkeit

Das Life-Game hat Ihnen sicherlich Spaß gemacht, einerseits als Bildschirmspiel mit seriösem Hintergrund , andererseits und vor allem als Programmieraufgabe. Der Spaß kann weitergehen - die jetzt vorliegende Version ist in Hinblick auf die Geschwindigkeit schon recht gut, es bleiben aber in anderer Hinsicht noch Wünsche offen. Das läßt sich durch einen Vergleich mit dem ursprünglichen Pflichtenheft leicht feststellen. Es gäbe also an dem Programm noch manche Verbesserung und Erweiterung anzubringen. Dabei können Sie Ihre Programmierkenntnisse anwenden und Erfahrung gewinnen.

Vielleicht ist Ihnen aufgefallen, daß bei der Realisierung des Programms zum Life-Game die Assemblersprache geradezu ideal war. Die benötigten Datenstrukturen (die Spielfelder) ließen sich auf Maschinenebene leicht darstellen und für die durchzuführenden Operationen genügten wenige Maschinenbefehle. In Assemblersprache lassen sich deshalb sehr schnelle Video-Spiele programmieren, mit höheren Programmiersprachen ist das nicht möglich.

Das gilt aber auch auf anderen Gebieten, unter denen wir uns nach dem Ausflug zu den Bildschirmspielen ein neues Thema als Programmierbeispiel aussuchen. Es geht um die Mathematik auf dem Mikroprozessor. Sie brauchen aber nicht gleich den Schreck derjenigen zu bekommen, für die noch von der Schule her der Begriff Mathematik ein wahres Schreckgespenst ist. Für das, was hier betrieben werden soll, ist das Wort Mathematik etwas hoch gegriffen, denn im Grund geht es um das Lösen von schlichten Rechenaufgaben, das mit Hilfe von Assemblerprogrammen geschehen wird.

Von seiner Anlage her ist der Mikroprozessor eigentlich nicht so gut geeignet für diese Aufgabe. Das fängt mit der Registerbreite an: die Breite eines Datenwortes beträgt im Z80-Mikroprozessor 8 Bit. Damit lassen sich nur die Zahlen von 0 bis 255 (oder bei anderer Zuordnung von -128 bis +127) darstellen. Mit Hilfe der Registerpaare können wohl auch 16-Bit-Datenworte verarbeitet werden, aber auch das bringt nicht viel: der mit 16 Bit darstellbare Zahlenbereich von 0 bis 65535 reicht für die meisten mathematischen Anwendungen nicht aus.

Wird ein größerer Zahlenbereich gewünscht, dann müssen die durchzuführenden Rechenoperationen auf einzelne Operationen mit 8-Bit-Worten oder 16-Bit-Worten zurückgeführt werden. Da dies meist recht aufwendig ist, werden

rein mathematische Programme in der Regel nicht in Assembler, sondern in einer höheren Programmiersprache geschrieben. Dort sind (z.B. in BASIC) die entsprechenden Maschinensprache-Unterprogramme in einer sogenannten Bibliothek vorhanden und werden automatisch aufgerufen. Es gibt aber viele Anwendungen, bei denen dieser Weg aus verschiedenen Gründen nicht möglich ist. Drei dieser Gründe fallen in's Gewicht: der benötigte umfangreiche Speicherplatz, die längere Rechenzeit und die mangelnde Flexibilität.

BASIC-Programme (um beim Beispiel dieser Programmiersprache zu bleiben) benötigen relativ viel Speicherplatz, da ja neben dem eigentlichen Programmtext in der höheren Programmiersprache immer der BASIC-Interpreter im Speicher stehen muß. Er arbeitet das Programm ab und muß dabei alle Maschinenprogramme zur Ausführung der einzelnen BASIC-Befehle von der Ein- und Ausgabe bis hin zur Berechnung mathematischer Funktionen enthalten - auch wenn das gerade vorliegende Programm nur einen Teil dieser Befehle benötigt.

Solange die Speichergröße des Computers ausreicht, mag der Interpreter und das Programm den notwendigen Platz haben. Wenn es ein längeres Programm ist, kann auch anstelle des Interpreters ein Compiler verwendet werden. Solch ein Übersetzer nimmt sich das in der höheren Sprache geschriebenen Programm vor der Verarbeitung vor, übersetzt es in Maschinensprache, wobei nur die Maschinenprogramme beigegeben werden, die tatsächlich notwendig sind. Gegen den Gebrauch solcher Compiler spricht, daß dafür größere Entwicklungssysteme mit Diskettenlaufwerken benötigt werden, während auf der anderen Seite für viele Anwendungen ein kleines System oder sogar ein Einchip-Prozessor ausreichen.

Der zweite und schwerwiegendere Nachteil bei der Verwendung höherer Programmiersprachen ist die längere Rechenzeit. Sie wird vor allem durch den Interpreter verursacht: er muß sich ja das Programm ansehen und die den Befehlen entsprechenden Maschinensprache-Programme heraussuchen. Man kann die Rechenzeit verkürzen, indem die BASIC-Programme in einer vorcodierten Form eingegeben werden. Außerdem wird die Rechengeschwindigkeit durch die Anwendung eines Compilers gesteigert.

In vielen zeitkritischen Anwendungen ist aber auch ein durch Compiler vorübersetztes Programm noch viel zu langsam. Das in einer höheren Sprache geschriebene Programm kann bei weitem nicht so gut optimiert werden wie ein von vornherein in Maschinensprache programmiertes. Das bei Kraftfahrzeug-Bremsen eingesetzte ABS-System (Anti-Blokkier-System) ist neben anderen Anwendungen (Textverarbei-

tungsprogramme, digitale Filter, "Life-Game") ein gutes Beispiel für die Forderung nach großer Rechengeschwindigkeit. Stellen Sie sich vor, daß in der sehr schnellen Regelschleife ein Meßwert von einem A/D-Wandler eingelesen und mit anderen Werten verrechnet werden muß, wobei in der Berechnungsformel beispielsweise eine Division durch 17 mit einer geforderten Rechengenauigkeit von 1 Prozent vorkommt.

Ist das Programm in einer höheren Programmiersprache geschrieben, dann wird es an dieser Stelle immer durch 17 mit der vollen Rechengenauigkeit dividieren. In Maschinensprache geht es in einem Bruchteil der Rechenzeit: man multipliziert mit 15 (vier Additionen und drei Schiebebefehle) und schiebt das Ergebnis um 8 Bit nach rechts, was einer Division durch 256 entspricht. Insgesamt wurde damit durch 256/15 = 17,067 dividiert und dabei kein (umfangreicher) Divisionsalgorithmus benötigt. Manche zeitkritischen Aufgaben lassen sich mit den heutigen Mikroprozessoren nur unter Verwendung solcher und ähnlicher Tricks lösen.

Neben Speicherplatzbedarf und Rechengeschwindigkeit ist der dritte Vorteil von Assemblerprogrammen die größere Flexibilität in der Anpassung an festliegende Bedürfnisse. Sie werden das an unserem Programmbeispiel zur Mathematik sehen. Dort kann das Assemblerprogramm ohne Mühe mit 70 und mehr Stellen Genauigkeit rechnen, was bei der Verwendung einer höheren Programmiersprache nicht möglich wäre.

Vor der Programmierung von Rechenaufgaben ist noch einige Vorarbeit zu erledigen. Zuerst erkunden wir die Möglichkeiten der Zahlendarstellung im Mikroprozessor und sehen uns dabei an, wie im Assembler und Tester gerechnet wird. Danach lernen Sie noch eine Systemfunktion sowie einige Pseudobefehle kennen. Anschließend wird dann Schritt für Schritt ein mathematisches Porgramm entwickelt.

Zusammenfassung

Mathematische Programme werden im allgemeinen in einer höheren Programmiersprache geschrieben, weil durch den begrenzten Zahlenbereich des Mikroprozessors recht aufwendige Assemblerprogramme nötig werden. Bei bestimmten Anwendungen verbietet sich jedoch der Gebrauch einer höheren Programmiersprache wegen des großen Bedarfs an Speicherplatz, der mangelnden Anpassungsfähigkeit oder der geringen Rechengeschwindigkeit. Besonders bei zeitkritischen Aufgaben werden auch mathematische Programme in Assembler geschrieben.

Zahlendarstellung im Mikroprozessor

Zunächst eine kurze Erinnerung an Grundlagenwissen, in diesem Fall geht es um die Zahlensysteme. Im Dezimalsystem werden Zahlen zur Basis 10 dargestellt. Man gibt hier also die Koeffizienten einer Summe von Zehnerpotenzen an:

$$1739_{10} = 1*1000 + 7*100 + 3*10 + 9*1$$

$$= 1*10^3 + 7*10^2 + 3*10^1 + 9*10^0$$

allgemein:

$$x_{10} = b_n*10^n + b_{n-1}*10^{10-1} + \ldots + b_1*10^1 + b_0*10^0$$

(Das Zeichen * steht für eine Multiplikation)

Während es in dieser dezimalen Darstellung um 10 verschiedene Ziffern geht, stehen in einem binären Rechenwerk zunächst nur 2 Ziffern zur Verfügung: die Null und die Eins. Man arbeitet daher mit dem Dualsystem, bei dem die einzelnen Ziffern Koeffizienten einer Summe von Zweierpotenzen sind:

$$01011001_2 = 1*64 + 0*32 + 1*16 + 1*8 + 0*4 + 0*2 + 1*1$$

$$= 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$$

allgemein:

$$x_2 = b_n*2^n + b_{n-1}*2^{n-1} + \ldots + b_1*2^1 + b_0*2^0$$

Dies zur Erinnerung. Wie Sie auch wissen, kann man ausgehend vom Dualsystem jeweils 4 dual dargestellte Zahlen zusammenfassen und gelangt so zum Sedezimalsystem mit den 16 Ziffern 0 bis 9 und A bis F:

0 = 0000	4 = 0100	8 = 1000	C = 1100
1 = 0001	5 = 0101	9 = 1001	D = 1101
2 = 0010	6 = 0110	A = 1010	E = 1110
3 = 0011	7 = 0111	B = 1011	F = 1111

Eine aus 8 Bit bestehende dual dargestellte Zahl läßt sich sedezimal zweistellig dargestellt viel leichter lesen, weshalb auch alle Zahlenausgaben des Betriebssystems in sedezimaler Darstellung erfolgen. Wenn man den Mikroprozessor zum Lösen normaler Rechenaufgaben verwendet, dann sollten die Ergebnisse natürlich in dezimaler Darstellung ausgegeben werden.

Wenn Sie die Tabelle auf der vorhergehenden Seite anschauen, können Sie sich sicher denken, wie Dezimalzahlen im Mikroprozessor dargestellt werden: man faßt jeweils 4 Bit zu einer Dezimalstelle zusammen, benützt hier natürlich nur die Bitkombinationen für die Ziffern 0 bis 9. Diese Mischung aus dezimaler und binärer Zahlendarstellung wird BCD-Code (Binary Coded Decimal) genannt. Wenn in einem solchen BCD-System gerechnet werden soll, muß dafür gesorgt werden, daß das Rechenwerk entsprechend funktioniert. Beispiel: 9 (1001B) plus 1 (0001B) darf nicht A (1010B) ergeben, sondern 0 und einen Überlauf auf die nächste Stelle. Sie werden später noch entsprechende Befehle des Z80-Prozessors kennenlernen. Man hat also die Wahl, im Dualsystem zu rechnen und das Ergebnis am Schluß ins Dezimalsystem zu wandeln, oder von Anfang an den BCD-Code zu verwenden. So weit, so gut - es gibt aber leider nicht nur positive und nicht nur ganze Zahlen, sondern auch negative und gebrochene Zahlen.

Zunächst zu den Bruchzahlen, die im Rahmen dieses Lehrgangs nicht ausführlich behandelt werden können. Für den Umgang mit gebrochenen Zahlen gibt es zwei Möglichkeiten: die Rechnung mit dem Komma an einer festen Stelle (Festkomma) oder die Rechnung mit dem Komma an einer beliebigen Stelle der Zahl (Fließkomma, Gleitkomma).

Die Festkomma-Rechnung unterscheidet sich praktisch überhaupt nicht von der Rechnung mit ganzen Zahlen. Der Rechenvorgang bei einer Addition von 12,34 + 5,86 ist identisch mit demjenigen bei der Addition von 1234 + 586. Lediglich bei den Ein- und Ausgabeoperationen muß das Komma berücksichtigt werden.

Bei der Gleitkomma-Rechnung wird die Kommastelle in einem getrennten Register, dem Exponentenregister, mitgeführt. Außerdem werden die internen Zahlen normalisiert, anders gesagt, die Mantisse wird so verschoben, daß in ihr keine führenden Nullen auftreten. Hier ein Beispiel:

Dezimalzahl	Mantisse	Exponent	
1923,45	(0),192345	4	(d.h. $0{,}192345*10^{4}$)
0,00719	(0),719000	-2	(d.h. $0{,}719*10^{-2}$)

(Tatsächlich wird übrigens statt des hier angegebenen Exponenten normalerweise eine sogenannte Charakteristik mitgeführt. Sie unterscheidet sich vom Exponenten nur durch die Addition einer Konstanten, damit sie nur positive Werte annimmt.)

Sollen zwei in dieser Art mit Gleitkomma dargestellte Zahlen addiert oder subtrahiert werden, dann wird zuerst die Mantisse der kleineren Zahl solange nach rechts ge-

schoben und dabei ihr Exponent (bzw. die Charakteristik) erhöht, bis die beiden Exponenten gleich sind. Dann stehen sozusagen die richtigen Stellen der Mantissen untereinander und können wie gewohnt addiert (subtrahiert) werden.

Bei der Multiplikation und Division wird so vorgegangen: die Mantissen werden multipliziert (dividiert) und die Exponenten addiert (subtrahiert). Anschließend muß das Ergebnis wieder normalisiert werden. Dabei wird die Mantisse solange nach links geschoben und der Exponent dabei dekrementiert, bis keine führenden Nullen mehr in der Mantisse auftreten.

Die Darstellung gebrochener Zahlen im Mikroprozessor fordert also einigen Aufwand, sie macht aber keine unüberwindlichen Schwierigkeiten. Wie sieht es nun bei der Darstellung negativer Zahlen aus? Sie lassen sich mit Komplementen darstellen. Das hört sich komplizierter an ,als es ist. Die folgende Komplement-Tabelle zeigt für die dezimal dargestellten Zahlen von 0 bis 9 die Komplemente:

Ziffer	0	1	2	3	4	5	6	7	8	9
Komplement	9	8	7	6	5	4	3	2	1	0

Das Komplement ist also die Zahl, die auf Neun ergänzt, oder anders gesagt: die Summe einer Ziffer und ihres Komplements ist immer 9, die höchste darstellbare Ziffer. Das gilt natürlich nur im Dezimalsystem, weshalb man auch vom Zehnerkomplement spricht.

Bildet man das Komplement der einzelnen Ziffern einer (mehrstelligen) Zahl, dann erhält man das sogenannte unechte Komplement dieser Zahl. Die Zahl -19 beispielweise erhält durch Komplementbildung der einzelnen Ziffern die Darstellung 80 im unechten Komplement.

Auf den ersten Blick ist nicht einzusehen, daß diese Darstellung im Komplement vorteilhafter sein soll als die gewohnte Darstellung aus Vorzeichen und Betrag. Beim Versuch, mit dem Komplement zu rechnen, wird die Sache schon klarer. Zum Beispiel die Substraktion 25 - 19. Da eine Subtraktion auch die Addition negativer Zahlen ist, muß 25 + (-19) das gleiche sein (wobei für -19 das Komplement eingesetzt wird:

```
  25          25
 -19         +80
 ---         ---
   6         105
               1
             ---
               6
```

Das Beispiel zeigt: bei der Verwendung von Betrag und Vorzeichen führt kein Weg an der Subtraktion vorbei. Bei der Verwendung des Komplements hingegen wird die Subtraktion durch die Addition des Komplements ersetzt. Beim unechten Komplement muß dann noch der aufgetretene Überlauf an der niedersten Stelle zur Korrektur addiert werden.

Die Bezeichnung "unechtes" Komplement läßt vermuten, daß es auch ein echtes Komplement gibt. Es wird gebildet, indem nach der Komplementierung der einzelnen Ziffern zur entstandenen Zahl eine Eins addiert wird. Dafür entfällt die Korrektur nach der Addition:

Betrag und Vorzeichen	unechtes Komplement	echtes Komplement
25	25	25
-19	+80	+81
---	---	---
6	105	106
	1	

	6	
2	2	2
-27	+72	+73
---	---	---
-25	74	75

Bei Verwendung des echten Komplements können Subtraktionen also einfach auf Additionen negativer Zahlen zurückgeführt werden, wobei ein negatives Ergebnis richtig im echten Komplement erscheint.

Im Dualsystem ist das Rechnen mit Komplement-Darstellungen besonders einfach: das unechte Komplement läßt sich durch Invertierung (Zweierkomplement) aller Bits bilden. Dieses unechte Komplement wird inkrementiert zum echten Komplement.

Beim Z80-Prozessor geht man davon aus, daß negative Zahlen im echten Komplement dargestellt werden. Dementsprechend bilden die Additions- und Subtraktionsbefehle intern das echte Komplement und addieren dann. Negative Ergebnisse erscheinen natürlich auch als echtes Komplement.

Die nebenstehende Tabelle deutet an, daß sich in der Zahlendarstellung mit echtem Komplement mit 8 Bit die Null und die positiven Zahlen von 1 bis 127 (höchstes Bit 0) sowie die negativen Zahlen von -1 bis -128 darstellen lassen.

0 1 1 1 1 1 1 1	127
.	.
.	.
.	.
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 0 0	0
1 1 1 1 1 1 1 1	-1
1 1 1 1 1 1 1 0	-2
.	.
.	.
.	.
1 0 0 0 0 0 0 0	-128

Die Verwendung des echten Komplements hat übrigens den schönen Nebeneffekt, daß Inkrementier- und Dekrementier-

befehle auch mit negativen Zahlen richtig funktionieren: -1 dekrementiert ergibt zum Beispiel -2.

Bei der ganzen Rechnerei muß man allerdings Bereichsüberschreitungen beachten:

```
  0 1 1 1 1 1 1 1        127
+ 0 0 0 0 0 0 1 0      +   2
-----------------      -----
  1 0 0 0 0 0 0 1        129
```


Die Addition von 127 und 2 ergibt 129, eine Zahl, die sich als echtes Komplement mit 8 Bit nicht mehr darstellen läßt. Während das links stehende Ergebnis der binären Rechnung völlig richtig ist, wenn man es als 8-Bit-Dualzahl ohne Vorzeichen interpretiert, ist es als 8-Bit-Dualzahl mit Vorzeichen in echter Komplementdarstellung falsch, es entspricht dann nämlich -127! Allerdings, der Mikroprozessor zeigt einen solchen Überlauf an, indem er das Overflow-Flag im Status-Register setzt.

Zusammenfassung

Bei normalen Rechenaufgaben gibt es die Möglichkeit, nach den Operationen mit dual dargestellten Zahlen die Ergebnisse dezimal darzustellen oder von Anfang an mit BCD-Code zu arbeiten. Bei Rechnungen mit gebrochenen Zahlen besteht die Wahl zwischen Festkomma- und Gleitkomma-Rechnung. Bei Rechnungen mit negativen Zahlen wird intern mit der Darstellung im echten Komplement gearbeitet, wobei sich mit 8 Bit nur der Zahlenbereich von +127 bis -128 darstellen läßt und Bereichsüberschreitungen beachtet werden müssen.

Fragen:

1. Warum erfolgen die Ausgaben des Betriebssystems in sedezimaler Darstellung?

2. Wie unterscheiden sich Rechnungen mit Festkomma und Fließkomma?

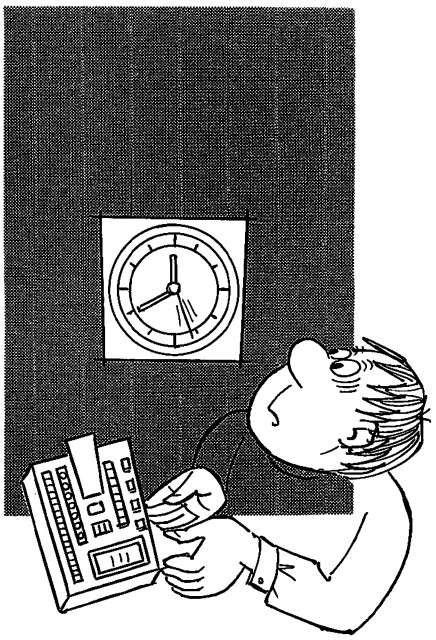
3. Auf welche Weise wird im Prozessor eine negative Zahl als echtes Komplement dargestellt?

(Die Antwort zu diesen Fragen finden Sie auf Seite G 9.)

Hauptprogramm "Fakultät"

Nach den verschiedenen vorbereitenden Kapiteln folgen jetzt die ersten Schritte zum Entwurf des angekündigten Programms, das Fakultäten berechnen soll. Wie schon gesagt, es handelt sich hier nicht um höhere Mathematik, sondern um eine ganz einfache Rechenaufgabe. Eine Fakultät ist das Produkt einer Folge von natürlichen Zahlen. In der Mathematik kürzt man Fakultät mit einem Ausrufezeichen ab. Fünf Fakultät wird also 5! geschrieben und bedeutet 1*2*3*4*5.

Wenn Sie Fakultät bisher kaum kannten, spätestens bei der Erwähnung des Ausrufezeichens kam die Erinnerung: das war doch die einzige Taste auf dem neuen Taschenrechner, bei deren Gebrauch nicht sofort ein Ergebnis in der Anzeige steht. Vielmehr dauert es eine sichtbare Weile, bis das Ergebnis ausgerechnet ist. Fakultäten haben nämlich die unangenehme Eigenschaft, daß das Ergebnis sehr schnell anwächst: während 5! mit 120 noch eine mäßig große Zahl ergibt, ist 10! schon über 3 Millionen groß. In der Mathematik haben die Fakultäten eine große Bedeutung in der Statistik, aber auch bei der Entwicklung transzendenter Funktionen in Potenzreihen.



Wegen der schnell anwachsenden Stellenzahl ist bei den erwähnten Taschenrechnern meist bei 69! Schluß, das Ergebnis 1,7112245 mal 10 hoch 98 läßt sich gerade noch darstellen. Wir haben in unserem Programm die Stellenzahl der auszugebenden Dezimalzahl auf 78 begrenzt. Damit paßt das Ergebnis gerade noch in eine Bildschirmzeile.

Die größte Fakultät, die sich mit 78 Dezimalstellen darstellen läßt, ist 57! = 4,053 * 10 hoch 76, also eine 4 mit 76 folgenden Stellen. Diese Zahl benötigt in binärer Darstellung 256 Stellen, da 2 hoch 256 gerade 1* 10 hoch 77 ist. Es muß demnach für das Ergebnis in Dezimaldarstellung ein BCD-Register mit 78 Stellen im Speicher reserviert werden, für das Ergebnis in Binärdarstellung ein binäres Register mit 256 Bit Breite.

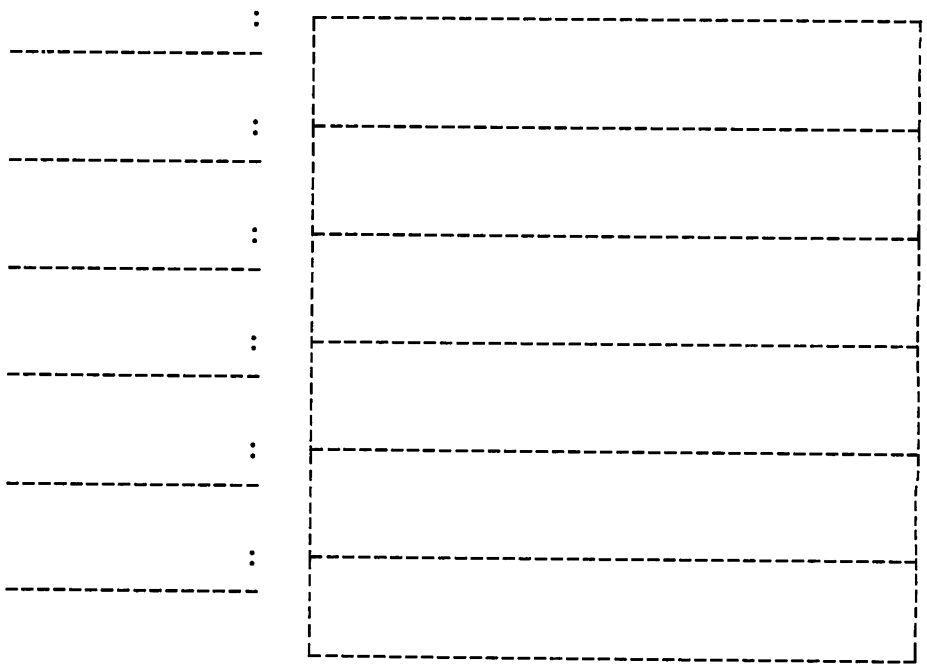
Damit ist auch schon entschieden, im Programm intern im Dualsystem zu rechnen und nicht mit BCD-codierten Zahlen. Bei deren Verwendung entfiele zwar die dezimal-dual- und die dual-dezimal-Wandlung zu Beginn und Ende des Programms, dafür wäre die eigentliche Rechnung etwas komplizierter. Wir haben uns aus didaktischen Gründen für die rein binäre Rechnung entschieden, weil man die dabei entstehenden Programme zur Zahlenumwandlung auch an anderen Stellen weiterverwenden kann.

Die Vorüberlegungen über den Speicherplatzbedarf finden sich zwar bei den Konstantendefinitionen im Listing zum Programm 2 auf den F-Seiten, dort sollten Sie aber vorläufig nicht nachschauen. Daher hier ein Auszug:

```
MAX_FAKT        EQU     57      ;größte mögliche Eingabe
                                ;bei
DEZ_STELL       EQU     78      ;maximal 78 Dezimalstel-
                                ;len
                                ;(57! =4*10 hoch 76) bzw.
BIN_STELL       EQU     256     ;256-Bit Binärzahl
                                ;(2 hoch 256 = 1*10 hoch
                                ;77)

BIN_BYTE        EQU     BIN_STELL/8
                                ;Anzahl Bytes d. Binär-
                                ;zahlen
```

Es wird deutlich, daß sich die Gedanken zum Speicherplatzbedarf sehr leicht in Assemblersprache formulieren lassen. Die nächste Überlegung geht dahin, was ein Programm zur Berechnung von Fakultäten überhaupt tun soll. Wir möchten Ihnen einige Hinweise geben, wie Sie Ihre Gedanken dazu systematisch in Assemblersprache formulieren und ein entsprechendes Programm schreiben können. Denn das ist ja der Zweck des Lehrgangs, daß Sie eigene Programme schreiben lernen!

Haben Sie sich überlegt, was das Programm alles tun soll? Dann tragen Sie die einzelnen Aufgaben in kurzen Stichworten in die nachstehenden Kästchen ein (wobei Sie sich nicht unbedingt an die Anzahl der Kästchen halten müssen):



Sie sollten in die Kästchen wirklich **Aufgaben** des Programms eintragen: z.B. "Fakultät berechnen" oder "dezimal wandeln". Keinesfalls aber Maschinenbefehle wie z.B. "LD C,FUNKTION". Anders gesagt: die Kästchen sollten so ausgefüllt sein, daß auch jemand, der mit Programmierung gar nichts am Hut hat, versteht, um was es geht.

Wenn Sie die Kästchen ausgefüllt haben, denken Sie sich für jede der anfallenden Aufgaben einen kurzen Namen aus und tragen diesen jeweils links vom Kästchen ein.

Wenn das erledigt ist, kommt gleich der nächste Schritt: Schreiben Sie vor jeden Namen, den Sie vor die Kästchen gesetzt haben, einfach einen CALL-Befehl. Wenn Sie nun nach dem letzten CALL-Befehl einen Sprung zurück zum Anfang schreiben und einen Teil der Kästcheninhalte mit Semicolon versehen als Kommentare hinter die CALL-Befehle setzen, dann haben Sie bereits ein wunderschönes, gut kommentiertes Hauptprogramm vor sich!

Selbstverständlich ist das nur der Anfang der Problemlösung. Die bislang fiktiven Unterprogramme, die dieses Hauptprogramm aufruft, müssen ja erst noch geschrieben werden - aber nach dem gleichen Verfahren.

Vergleichen Sie jetzt (aber wirklich erst jetzt!) Ihr Hauptprogramm mit dem Hauptprogramm im Listing des Programms 2 auf Seite F 35.

CSEG, DSEG und bedingte Assemblierung

Diese Überschrift sagt Ihnen im Moment nicht viel, es wird aber gleich deutlich werden, daß vor dem Fortgang der Programmierung noch einige Spezialitäten des Assemblers untersucht werden müssen.

Bei der Suche nach dem Hauptprogramm im Listing haben Sie sich vielleicht über den "Datenbereich" im Programm und über den Vorspann des Hauptprogramms gewundert, vielleicht auch über die Definition von "wahr" und "falsch" bei den Konstantendefinitionen.

Der Assembler besitzt einen internen Programmzähler, der mitzählt, an welche Adresse der gerade assemblierte Befehl geschrieben wird. Nachdem der Befehl assembliert ist, wird dieser Zähler erhöht, damit er auf die nächste freie Speicherstelle zeigt.

Mit dem Pseudobefehl ORG, den Sie bereits kennen, kann der Programmzähler auf einen bestimmten Wert gesetzt werden:

ORG <ausdruck>

setzt den internen Zähler auf den Wert von <ausdruck>. Mit der Anweisung ORG 100H (<ausdruck>= 100H) wurde zu Beginn der bisherigen Programme der Progammzähler auf die Speicherstelle 100H, den Anfang des Benutzerspeichers, gesetzt. Das wäre übrigens nicht unbedingt nötig gewesen, da der Assembler bei fehlender ORG-Anweisung automatisch bei 100H mit der Assemblierung beginnt.

Auf Seite D 14 steht, daß im Tester mit dem Zeichen $ der momentane Stand des Programmzählers in die Berechnung eines Ausdruck einbezogen werden kann. Bei einem Rechenausdruck im Assembler wird für das Zeichen $ natürlich nicht der Programmzählerstand der Z80-CPU oder des Testers eingesetzt, sondern der interne Programmzähler des Assemblers. Man könnte also mit

```
SYMBOL   EQU   $
```

dem Symbol SYMBOL den Wert des momentanen Programmzählerstands zuweisen.

Frage:

1. Die gleiche Zuweisung wird meistens anders erreicht, wie?

(Die Antwort zu dieser Frage finden Sie auf Seite G 10.)

Viele Mikroprozessor-Programme werden für spezielle Anwendungen geschrieben, die nicht auf einem großen Entwicklungssystem ablaufen sollen. Vielmehr wird oft ein kleines System bevorzugt, das nur aus CPU, RAM, EPROM und einigen Ein- und Ausgabebausteinen besteht. So etwas läßt sich preisgünstig auf einer einzigen Leiterplatte aufbauen und kann dann mit entsprechenden Programmen die verschiedensten Steuer- und Regelaufgaben übernehmen.

In solchen Systemen gibt es keinen durchgehenden Hauptspeicher wie in einem Entwicklungssystem, sondern einen getrennten Speicherbereich mit einem EPROM, in dem das Programm abgelegt ist und RAM's, in denen Daten gespeichert werden können. Es besteht also eine strikte Tren-

nung zwischen Programm- und Datenbereich, denn im EPROM können keine variablen Daten gespeichert werden, während ein Programm im RAM bei jedem Abschalten der Betriebsspannung (und bei Stromausfall) verloren geht.

Dieser Trennung von Programm- und Datenbereich tragen komfortable Assembler und natürlich auch der Assembler Ihres Systems Rechnung: es sind für beide Bereiche verschiedene interne Programmzähler vorgesehen.

Für den Programm-Bereich ist die englische Bezeichnung Code-Segment üblich, entsprechend für den Daten-Bereich Data-Segment. Mit den Pseudo-Befehlen

```
CSEG    ;Programmzähler auf Code-Segment
DSEG    ;Programmzähler auf Data-Segment
```

kann zwischen den beiden Bereichen bzw. zwischen den beiden Zählern hin und her geschaltet werden.

Normalerweise ist der Programmzähler des Code-Segments aktiv. Am Anfang des Datenbereichs im Fakultäts-Programm wird mit DSEG der Zähler des Data-Segments aktiviert. Natürlich muß dieser Zähler erst einmal mit einer ORG-Anweisung auf einen Anfangsstand gesetzt werden. Er würde sonst den Datenbereich ab Adresse 100H beginnen lassen, wo aber eigentlich der Programmbereich anfängt.

Wir haben für den Anfang des Datenbereichs die Adresse 1000H gewählt. Diese liegt so hoch, daß es sicher nicht zu Überschneidungen mit dem Programmbereich kommt.

Nach der Definition des Datenbereichs wird mit CSEG in den Programmbereich zurückgeschaltet und dort der Beginn des Programmbereichs definiert. Nachdem so beide Programmzähler gesetzt sind, kann zwischen ihnen beliebig hin und her geschaltet werden.

Man kann also z.B. einige Zeilen Programm schreiben, und dann zwischendurch zur Definition eines Variablenspeichers mit DSEG in den Datenbereich schalten. Die neue Variable wird dort definiert, wo der Datenbereich zuletzt verlassen wurde.

Im Beispiel auf der nächsten Seite sind die Programm- und Datenabschnitte einfach mit großen Buchstaben dargestellt.

Es wird deutlich, wie der Assembler durch die Verwendung von CSEG und DSEG aus den im Programmtext gemischt vorkommenden Programm- und Datenabschnitten getrennte Programm- und Datenbereiche erzeugt.

Programmtext	ergibt	Programmbereich	Datenbereich
CSEG		0000H: A	2000H: B
ORG 0		C	E
A		D	F
		G	H
DSEG			
ORG 2000H			
B			
CSEG			
C			
D			
DSEG			
E			
F			
CSEG			
G			
DSEG			
H			
END			

Im vorliegenden Fakultät-Programm ist die Verwendung von CSEG und DSEG natürlich nicht zwingend notwendig. Da dieses Programm ohnehin im RAM-Bereich des Entwicklungssystems abläuft, könnte es komplett wie bisher gewohnt mit dem Programmzähler des Code-Segments assembliert werden.

Im Datensegment sind die Speicherbereiche für die verschiedenen Register mit dem schon bekannten Pseudobefehl DS (Define Space) reserviert. Mit DS wird (siehe Seite C 31, C 32) ein Speicherbereich nur zugewiesen. Ob dieser auch für Benutzerprogramme frei ist, wird dabei nicht geprüft. Wird dagegen mit dem Pseudobefehl DB ein Byte in den Speicher geschrieben, dann prüft der Assembler, ob die Adresse möglicherweise für das Betriebssystem reserviert ist und zeigt dies gegebenenfalls mit einer I-Fehlermeldung (Illegale Adresse) an. Am Ende des Datenbereichs haben wir deshalb eine Anweisung

```
DB  0
```

angefügt, die eine nutzlose Null in den Speicher schreibt. Der Assembler prüft dabei aber, ob der bei 1000H beginnende Datenbereich bis in die reservierten Systemadressen oder den Textspeicher reicht. Je nachdem, wieviel Platz Sie für den Textspeicher reserviert haben, kann dies durchaus der Fall sein.

Der Pseudobefehl DB (Define Byte) steht aber nicht allein in der Landschaft, er ist vielmehr von zwei anderen Pseudobefehlen eingerahmt:

```
IF    CHECK
DB    0
ENDIF
```

Es handelt sich hier um eine sogenannte bedingte Assemblierung. "If" ist das englische Wort für "wenn", es steht also sozusagen da:

```
WENN  CHECK
DB    0
ENDE-WENN
```

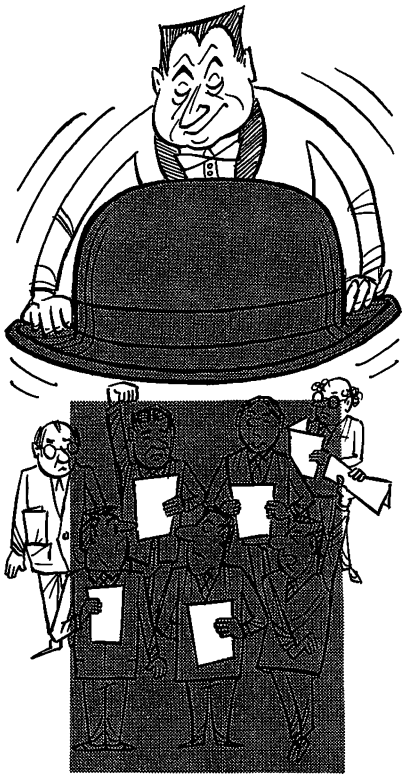
Wenn das Symbol CHECK ungleich Null ist, werden die folgenden Zeilen (im vorliegenden Fall ist es nur eine Zeile) assembliert, bis zum Ende der bedingten Assemblierung, die mit ENDIF markiert ist. Hat CHECK dagegen den Wert Null, dann werden die Zeilen bis ENDIF einfach ausgelassen. Man kann also am Anfang des Programms das Symbol CHECK auf Null oder einen Wert ungleich Null setzen und damit entscheiden, ob der Pseudobefehl DB assembliert wird oder nicht. Auf diese Weise läßt sich ein Programm variabel gestalten und für unterschiedliche Versionen assemblieren. Dabei müssen lediglich am Anfang des Programms die entsprechenden Symbole gesetzt werden.

Im vorliegenden Programm sind für diesen Zweck die beiden Symbole TRUE (=wahr) und FALSE (=falsch) definiert. FALSE ist auf Null gesetzt, denn es soll bedeuten: nicht assemblieren! TRUE könnte man z.B. auf 1 setzen, der Assembler bietet aber eine elegantere Möglichkeit : NOT FALSE (das ist beinahe Umgangssprache). In diesem Fall wird der Wert von TRUE auf OFFFFH gesetzt, was Sie mit dem Rechenprogramm des Testers leicht nachprüfen können.

Diese Art der Definition von TRUE und FALSE ist nicht nur recht elegant, sondern hat noch weitere Vorteile. Es wird nämlich NOT TRUE automatisch wieder zu Null, also FALSE, was ja auch beabsichtigt ist.

Im vorliegenden Beispiel wird DB 0 assembliert, wenn der Wert von CHECK auf TRUE gesetzt ist. Ist er auf FALSE gesetzt, dann wird der zwischen IF und ENDIF stehende Programmtext ausgelassen.

Auch bei diesen Feinheiten des Assemblers taucht sofort die naheliegende Frage auf, wozu das gut ist: angenommen, es soll ein Programm geschrieben werden, das für verschiedene Drucker mit unterschiedlichen Steuercodes leicht anzupassen sein soll, das in einer Mini-Version



über einen Teil seiner Funktionen nicht verfügt und ausserdem Fehlertexte je nach Einsatzgebiet in Deutsch oder Englisch ausgeben soll. Sie möchten dafür sicher nicht lauter verschiedene Programme schreiben und in der Tat: mit bedingter Assemblierung lassen sich alle diese Anforderungen unter einen (Programm-)Hut bringen.

Zunächst einmal werden die Steuercodes für die Drucker bedingt assembliert:

```
IF   DRUCKER1
DB   ...
ENDIF

IF   DRUCKER2
DB   ...
ENDIF

usw.
```

Zu Beginn des Programms muß dann eines der Symbole DRUCKERx auf TRUE, die anderen auf FALSE gesetzt werden, um das Programm für diesen Drucker angepaßt zu assemblieren. Genauso wird das Symbol MINI auf TRUE gesetzt, wenn die MINI-Version des Programms gewünscht ist. Mit

```
IF   NOT MINI
LD   HL,...
..
..
..
ENDIF
```

kann man auf elegante Weise erreichen, daß bestimmte Programmteile in der MINI-Version nicht assembliert werden. Für die Ausgabe der Fehlertexte in verschiedenen Sprachen schließlich wird ein Symbol DEUTSCH eingeführt:

```
IF   DEUTSCH
DB   'FEHLER'
ELSE
DB   'ERROR'
ENDIF
```

Hier ist noch der Pseudobefehl ELSE hinzugekommen ("Else ist das englische Wort für "sonst"). Mit der Konstruktion aus IF-ELSE-ENDIF kann eine von zwei Programmvarianten ausgewählt werden, es wird dabei zu Beginn des Programms ein Symbol (hier DEUTSCH) entsprechend gesetzt.

Natürlich kann statt des einzelnen Symbols auch wieder ein ganzer Rechenausdruck stehen, womit verschiedene Bedingungen verknüpft werden können, beispielsweise:

```
IF   DEUTSCH AND NOT MINI
DB   'Dies ist eine Fehlermeldung'
ENDIF
```

Zusammenfassung

Der Assembler unseres Betriebssystems arbeitet mit getrennten Programmzählern für Code-Segment und Data-Segment. Normalerweise wird der Zähler des Code-Segments benutzt. Mit den Pseudobefehlen CSEG und DSEG kann zwischen beiden Programmzählern hin und her geschaltet werden.

Mit den Pseudobefehlen IF, ELSE und ENDIF ist eine bedingte Assemblierung möglich. Dabei wird durch entsprechende Labeldefinitionen am Anfang des Programms festgelegt, welche Programmteile assembliert werden und welche nicht.

Fragen:

1. Welchen Sinn hat die Verwendung von getrennten Programmzählern bei Ihrem System, wo Data-Segment und Code-Segment im Hauptspeicher stehen?

2. Auf Seite C 71 wird gesagt, daß bei der Definition TRUE EQU NOT FALSE der Wert von TRUE auf OFFFHH gesetzt wird und dies mit dem Tester nachprüfbar ist. Wie machen Sie das ?

(Die Antworten zu diesen Fragen finden Sie auf den Seiten G 10 und G 11.)

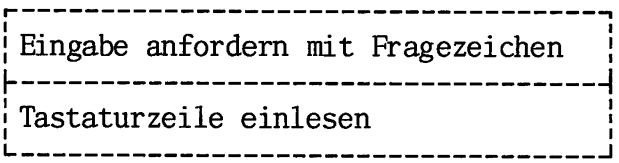
Weitere Teile des Programms Fakultät

Auf den Seiten C66 und C 67 haben Sie ein Hauptprogramm zur Berechnung von Fakultäten entwickelt. Für die verschiedenen Teilaufgaben werden dabei Unterprogramme aufgerufen. Damit der Asssembler diese Unterprogramme findet, können diese zunächst einmal definiert und mit einem RET-Befehl abgeschlossen werden. An den Kopf jedes Unterprogramms kommt eine kurze Beschreibung, was es tun soll. So wird das Programm Stück für Stück weiterentwickelt.

Im Programm 2 (Seite F 34) sind die Unterprogramme für die Eingabe (EINGABE) und die Wandlung der eingegebenen Zahl in die Binär-Darstellung (HOLE_ZAHL) schon ausgeführt. Die Unterprogramme FAKT und WANDLE_DEZ stehen noch ganz leer da. Im Unterprogramm AUSGABE wird vorerst provisorisch die erweiterte Systemfunktion für die Ausgabe einer vierstelligen Sedezimalzahl (Seite B 20) aufgerufen, um das Funktionieren der beiden schon fertigen Programmteile überprüfen zu können.

Nach der Aufstellung des Hauptprogramms muß als nächstes das Unterprogramm EINGABE geschrieben werden. Es soll eine Zahl von der Tastatur einlesen. Die Eingabe sollte natürlich vom Computer im Dialog angefordert werden, z.B. durch die Ausgabe eines Fragezeichens auf dem Bildschirm.

Grundsätzlich lassen sich die Unterprogramme ganz ähnlich wie das Hauptprogramm entwickeln, das wurde bereits auf Seite C 67 gesagt. Die Aufgaben des Unterprogramms EINGABE lassen sich in zwei Kästchen beschreiben:



Diese beiden Aufgaben können nun wieder an zwei kleinere Unterprogramme weiterverwiesen werden. Da sich aber beide mit Systemfunktionen in wenigen Schritten lösen lassen, ist dies nicht nötig.

Nachdem Sie sich überlegt haben, wie die Lösung aussieht, vergleichen Sie Ihre Gedanken mit unserem EINGABE-Programm. Wahrscheinlich wird es keine großen Unterschiede geben. Selbstverständlich können Sie einen anderen Anforderungstext als nur ein Fragezeichen ausgeben lassen, oder eine andere Puffergröße (MAX_ LÄNGE) für die Texteingabe vorsehen.

Frage:

1. Wie könnte gegebenenfalls das aus Hauptprogramm und Unterprogramm EINGABE bestehende Programm verlassen werden?

(Die Antwort zu dieser Frage finden Sie auf Seite G 12.)

Um das Unterprogramm HOLE-ZAHL zu verwirklichen, sind wieder einge Vorüberlegungen notwendig. Da ist die Frage, wie man eine als ASCII-String vorgegebene Dezimalzahl in eine Dualzahl umwandeln kann. Führen Sie sich noch einmal vor Augen, was die einzelnen Ziffern b_1 bis b_n einer dezimal dargestellten Zahl allgemein bedeuten:

$$x_{10} = b_n*10^n + b_{n-1}*10^{n-1} + \ldots + b_1*10^1 + b_0*10^0$$

Es sind die Koeffizienten der zu den einzelnen Stellen gehörenden Zehnerpotenzen. Man kann diese Schreibweise umformen, indem man die Zehner ausklammert:

$$x_{10} = ((..(b_n*10 + b^{n-1})*10 + b^{n-2})*10 + \ldots)*10 + b^1$$

Aus dieser Schreibweise läßt sich ableiten, daß der Wert einer dezimal dargestellten Zahl folgendermaßen ermittelt werden kann: man nimmt die höchste Stelle (b^n), multipliziert sie mit 10, addiert die nächste Stelle, multipliziert wieder mit 10 usw., bis alle Stellen verrechnet sind. Führt man diese Rechnung im Dezimalsystem aus, dann ist das ganze natürlich witzlos. Rechnet man aber im Dualsystem, dann erhält man zum Schluß den Wert der ursprünglich dezimal dargestellten Zahl in binärer Darstellung.

Anders gesagt: man nimmt von der dezimal dargestellten Zahl die höchste Stelle. Ist dies schon die Einerstelle, dann kann ihr Wert bleiben, die Rechnung ist fertig. Ist es jedoch noch nicht die Einerstelle, dann muß sie mit Zehn multipliziert werden. Dann wird der Wert der nächsten Stelle addiert. War dies die Einerstelle, so ist man fertig. Andernfalls muß wieder mit zehn multipliziert und danach die nächste Stelle addiert werden. Durch die Multiplikation mit Zehn wird automatisch die bisherige Einer- zur Zehnerstelle, die bisherige Zehner- zur Hunderterstelle usw. Sind alle Stellen verarbeitet, dann liegt die betreffende Zahl in der Darstellung des Zahlensystems vor, in dem gerechnet wurde - bei unserer Rechnung wird es das Dualsystem sein.

Als Beispiel für die Umwandlung dient die dezimal dargestellte Zahl 173:

$10 = 0001^2$

es handelt sich nicht um die letzte Stelle, also wird mit $10^{10} = 1010^2$ multipliziert.

$0001^2 * 1010^2 = 1010^2$

und die nächste Stelle $7^{10} = 0111^2$ addiert:

$1010^2 + 0111^2 = 10001^2$

es ist nicht die letzte Stelle, also multiplizieren:

$10001^2 * 1010^2 = 10101010^2$

und nächste Stelle $3^{10} = 0011^2$ addieren:

$10101010^2 + 0011^2 = 10101101^2$

Da dies die letzte Stelle war, ist 173^{10} gleich 10101101^2. Prüfen Sie's nach, es stimmt!

Wir gehen nicht auf die Ableitung dieses Umwandlungsalgorithmus ein, es gibt auch noch andere Möglichkeiten - er steht hier, weil er sich für die Umsetzung in ein Computerprogramm bestens eignet.

Struktogramme

Die zuvor erwähnte Umsetzung des Umwandlungsalgorithmus in ein Programm beginnt damit, seinen Ablauf wie die bisher behandelten Programmteile in Kästchen zu fassen. Dabei muß jetzt aber überlegt werden, wie in den Kästchen Programmschleifen dargestellt werden. In den Struktogrammen wird dies so gemacht: an der rechten Seite eines grösseren Kästchens, das die ganze Schleife darstellt, wird ein kleineres Kästchen gesetzt, das den Inhalt der Schleife enthält. Bild C 76.1 zeigt, wie das gemeint ist. Man sieht, daß mit den eingerückten Kästchen auch die Wiederholungen in einem Ablauf recht übersichtlich darge-

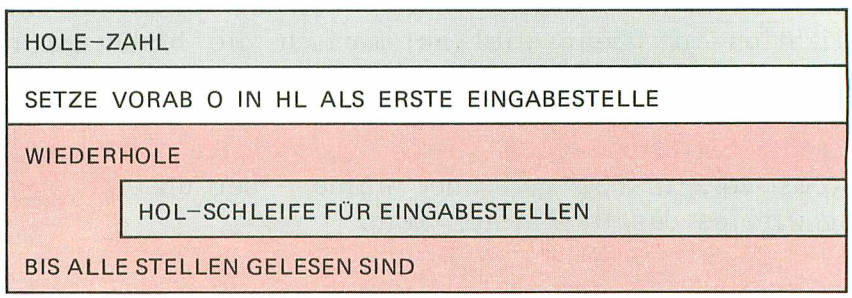


Bild C 76.1

Der Name des im Struktogramm dargestellten Programmteils ist grau unterlegt. Die Schleife ist in unserer Darstellung farbig betont. Die graue Markierung des Schleifeninhalts weist daraufhin, daß dieser Programmteil noch gesondert erläutert wird, was in Bild C 78.1 der Fall ist.

HOL–SCHLEIFE FÜR EINGABESTELLEN
HOLE NÄCHSTE EINGABESTELLE NACH A
ASCII IN A WIRD BINÄR
ABBRUCH, WENN EINGABESTELLE KEINE ZAHL
BRINGE AKTUELLE EINGABESTELLE AUS HL NACH DE
SETZE HL:=0
SETZE ZÄHLER B:= 10
WIEDERHOLE
ADDIERE HL ZU DE, ERGEBNIS IN HL
ZÄHLER: = ZÄHLER–1
BIS ZÄHLER = 0
ADDIERE ERGEBNIS ZU NÄCHSTER EINGABESTELLE IN A

Bild C 77.1

Die Hol-Schleife im Struktogramm Bild C 77.1 enthält eine weitere Schleife. Was dort geschieht, ist im Text auf Seite C 79 dargestellt.

stellt werden können. Nun ist allerdings die Umsetzung in Assemblersprache nach einem wegen der Übersichtlichkeit sehr generellen Struktogramm nicht einfach. Deshalb wird man für Kästchen mit umfangreichem Inhalt noch einmal eine Art "Unterstruktogramm" anlegen, so wie das Programm selber in Unterprogramme aufgeteilt wird. Im vorliegenden Fall sehen Sie in Bild C 77.1, daß in der Schleife "Hol-Schleife für Eingabestellen" neben einer Reihe von Kästchen noch einmal eine Schleife enthalten ist. Stünde das alles im Struktogramm in Bild C 76.1, dann wäre dieses recht unübersichtlich.

Im Unterprogramm HOLE_ZAHL kommen also zwei Schleifen vor, wobei ebenfalls zweimal ein Maschinenbefehl verwendet wird, der für die Programmierung solcher Schleifen sehr praktisch ist und deswegen noch oft auftauchen wird. Es ist der Befehl DJNZ, der zwei Befehle kombiniert:

```
DJNZ Zieladresse
```

bewirkt das gleiche wie

```
DEC  B
JR   NZ,Zieladresse
```

Es wird das B-Register dekrementiert und ein Sprung ausgeführt, wenn dieses nicht Null ist. Vor der Schleife wird das B-Register auf die Anzahl der gewünschten Schleifendurchläufe gesetzt. Es wird dann solange zum Schleifenbeginn (Zieladresse) zurückgesprungen, bis im B-Register auf Null heruntergezählt ist, dann geht es weiter im Programm.

Im Programm-Listing (Seite F 36) sehen Sie, wie zu Beginn des Unterprogramms HOLE_ZAHL das B-Register mit der Anzahl der von der Eingabefunktion zurückgegebenen Zeichen geladen wird.

Nachdem eine Null als erste Eingabestelle in das HL-Registerpaar geschrieben ist, beginnt die Hol-Schleife für Eingabestellen (Bild C 77.1) mit dem Hereinholen der nächsten Eingabestelle und ihrer Wandlung von ASCII in die binäre Darstellung. Diese Wandlung ist recht einfach. Da die Ziffern von 0 bis 9 die ASCII-Codes von 30H bis 39H belegen, muß lediglich vom ASCII-Code der untersuchten Ziffer der Wert 30H (der Ascii-Code für 0) abgezogen werden und es bleibt der Wert der Ziffer übrig. Wird dabei das Carry-Flag gesetzt, dann ist es ein Zeichen für ein negatives Ergebnis: der untersuchte ASCII-Code war kleiner als 30H und daher keine Zahl zwischen 0 und 9. In diesem Fall erfolgt ein Rücksprung ins Hauptprogramm, wo das gesetzte Carry-Flag als Fehlerindikator ausgewertet und eine neue Eingabe angefordert wird. Mit dem Befehl

```
CP   10
```

wird noch geprüft, ob die untersuchte Ziffer größer als 9 (also größer oder gleich 10) war. Wenn ja, dann handelt es sich wiederum um eine nicht gültige Eingabe. Der CP-Befehl ist im Prinzip ein SUB-Befehl, der Inhalt des A-Registers wird nicht verändert. Bei dieser Abfrage bedeutet im Gegensatz zur vorhergehenden ein positives Ergebnis einen Fehler. Da bei einem positiven Ergebnis das Carry-Flag nicht gesetzt ist, wird es mit CCF invertiert und dann als Fehlerindikator erkannt. In der anschließenden Schleife wird die im DE-Registerpaar stehende Eingabestelle zehnmal (B=10) mit sich selber addiert (mit 10 multipliziert wie auf Seite C 75 beschrieben). Die nachstehende Tabelle zeigt, wie das durch die fortlaufende Addition der Registerpaare HL und DE geschieht:

	akt. Stelle					Schleifendurchlauf
	HL	+	DE	->	HL	
Anfang:	0	+	2	=	2	1
	2	+	2	=	4	2
	4	+	2	=	6	3
	.		.		.	.
	.		.		.	.
	16	+	2	=	18	9
	18	+	2	=	20	10

Als Eingabebeispiel wurde die Zahl 23 und davon die 2 als aktuelle Stelle genommen.

Nach der Multiplikation wird die im A-Register stehende nächste Eingabestelle addiert.

Wenn alle Eingabestellen gewandelt sind, wird der gefundenen Wert noch mit dem maximal erlaubten Wert durch Subtraktion (HL-DE) verglichen. Auch dieser Vergleich liefert ein gesetztes Carry-Flag als Fehlerindikator, wenn ein negatives Ergebnis anzeigt, daß DE (die eingegebene Zahl) größer als HL (der Maximalwert) ist.

Geben Sie nun das Programm 2 (ab Seite F 34) in Ihr System ein und assemblieren Sie es. Auf Seite F 38 steht, was nach dem Assemblieren auf dem Bildschirm angezeigt werden muß. Sie können das Programm mit RUN starten und Dezimalzahlen eingeben. Ist die eingegebene Zahl kleiner als 58, wird sie in sedezimaler Darstellung vom provisorischen Unterprogramm AUSGABE wieder ausgegeben.

Sie können den Rücksprung bei Fehler im Hauptprogramm auch versuchsweise entfernen und dann mit größeren Zahlen als 57 spielen. Vor allem aber sollten Sie mit dem Tester spielen. Verfolgen Sie, was in den Schleifen geschieht !

Zusammenfassung

Das Unterprogramm HOLE_ZAHL gibt Gelegenheit, den Umgang mit Struktogrammen und deren Darstellungsarten kennenzulernen. Wie bei den Programmen selber wird auch bei den Struktogrammen aus Gründen der Übersichtlichkeit eine Darstellung von umfangreicheren Teilen in getrennten Struktogrammen vorgezogen.

Frage:

1. Wie sehen Sie sich die Vorgänge in der Schleife im Bild C 78.1 (Multiplikation mit 10) im Tester an?

(Die Antwort zu dieser Frage finden Sie auf Seite G 12.)

Weitere Unterprogramme

Nachdem die Eingabe funktioniert, wird als nächstes der Kern des Programms realisiert, nämlich das Unterprogramm zur Berechnung der Fakultät der eingegebenen Zahl. Gehen Sie dabei vor, wie wir es bisher gemacht haben: zuerst wird als Text formuliert, was das Programm tun soll. Danach werden die einzelnen Schritte in die Kästchen eines Struktogramms geschrieben und schließlich in Assemblersprache umgesetzt.

Schritt 1:

Das Unterprogramm FAKT soll beginnend mit 1 alle natürlichen Zahlen bis einschließlich der eingegebenen miteinander multiplizieren. Mit Rücksicht auf die Befehle des Z80-Prozessors wird der Satz umgestellt: das Unterprogramm FAKT soll beginnend von der eingegebenen Zahl abwärts bis 1 alle Zahlen miteinander multiplizieren.

Schritt 2:

Bild C 80.1 zeigt, wie das Struktogramm für das Unterprogramm FAKT aussieht. Damit ist der Entwurf fertig. Bei der Umsetzung in Assemblersprache wird der Aufruf weiterer Unterprogramme notwendig sein, denn sowohl das Löschen eines ganzen Registers (mit 256 Bit) als auch der Registertransport und die Multiplikation erfordern umfangreiche Programmstücke und eigene Schleifen.

Schritt 3:

```
FAKT:
        LD    HL,BIN_REG1      ;BIN_REG1 := 1
        CALL  CLEAR            ;BIN_REG1 wird gelöscht
        CALL  INKREMENT        ;und auf 1 dekrementiert
```

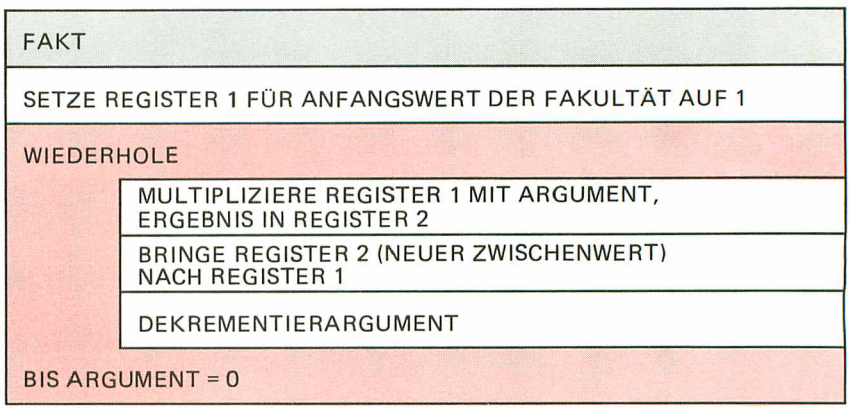


Bild C 80.1

Struktogrammm des Unterprogamms FAKT.

```
        LD    B,E              ;Argument ins B-Register
FAKT_SCHLEIFE:                 ;wiederhole
        CALL  MULT             ;BIN_REG2 :=BIN_REG1 * B
        CALL  MOV12            ;BIN_REG1 :=BIN_REG2
        DJNZ  FAKT_SCHLEIFE    ;bis Zähler=0
```

Als nächstes geht es um die Formulierung der verschiedenen Unterprogramme, angefangen mit CLEAR:

Schritt 1

Das Unterprogramm CLEAR soll das Binärregister löschen, auf welches das HL-Registerpaar zeigt.

Schritt 2

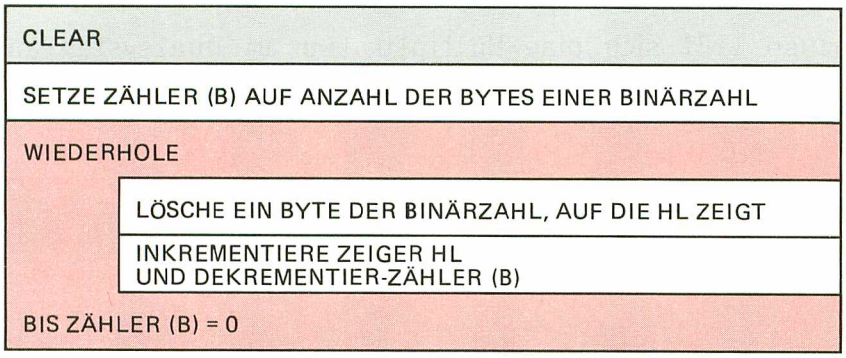


Bild C 81.1

Auch im Unterprogramm CLEAR ist eine Schleife enthalten.

Schritt 3

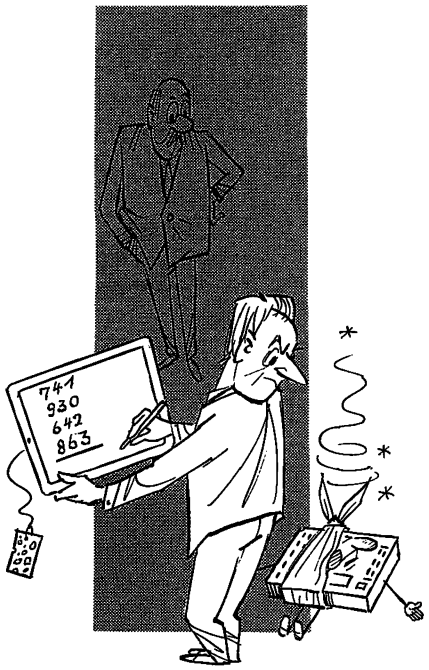
```
CLEAR:
        PUSH  BC
        LD    B,BIN_BYTE       ;Zähler auf Anzahl Byte
CLEAR1                         ;wiederhole
        LD    (HL),0           ;ein Byte löschen
        INC   HL               ;Zeiger inkrementieren
        DJNZ  CLEAR1           ;Zähler (B) dekrementieren
        POP   BC               ;bis Zähler =0
        RET
```

Als Nebeneffekt ergibt sich, daß der Zeiger HL nach dem Unterprogramm CLEAR auf die Stelle mit dem niederwertigsten Byte des gelöschten Registers zeigt. Dadurch wird das Unterprogramm INKREMENT so einfach, daß man es weglassen und seine Funktion ins Unterprogramm FAKT übernehmen kann.

Frage:

1. Wie kann bei den genannten Bedingungen das Register 1 sehr einfach auf 1 gesetzt werden?

(Die Antwort zu dieser Frage finden Sie auf Seite G 12.)



C

82

Das Unterprogramm MULT soll BIN_REG1 mit dem Inhalt von B multiplizieren und das Ergebnis nach BIN_REG2 schreiben. Sie können es sich schon denken, daß man sich hier wieder einige Gedanken machen muß. Es soll nämlich an dieser Stelle keine solch primitive Multiplikation wie im Unterprogramm HOLE_ZAHL verwendet werden - dort wurde statt einer Multiplikation mit 10 einfach zehnmal addiert.

Beherrschen Sie trotz Taschenrechner noch die schriftliche Multiplikation von dezimal dargestellten Zahlen? Das war doch recht einfach: die beiden beteiligten Zahlen wurden hingeschrieben und die erste (Multiplikand) mit den Stellen der zweiten Zahl (Multiplikator) einzeln multipliziert. Die Ergebnisse wurden schräg untereinander geschrieben, um die Stellenwerte zu berücksichtigen, bevor die einzelnen Produkte aufaddiert wurden.

Genauso läßt sich eine Multiplikation im Dualsystem ausführen, hier ein Beispiel:

```
1 0 0 1 0  *  1 0 1 1
--------------------
      1 0 0 1 0
      0 0 0 0 0
        1 0 0 1 0
          1 0 0 1 0
---------------------
1 1 0 0 0 1 1 0
```

Im Dualsystem ist die Multiplikation mit den einzelnen Ziffern besonders einfach, es muß ja nur mit 1 oder 0 multipliziert werden. Das Beispiel zeigt ganz eindeutig, wie die Multiplikation ablaufen muß: die eine Zahl wird mit den einzelnen Stellen der anderen multipliziert, die einzelnen Produkte schräg untereinander geschrieben und aufaddiert. Das Multiplizieren ist hier, wie gesagt, sehr einfach, das Aufaddieren der Produkte kann auch nicht schwierig sein. Aber "schräg untereinander schreiben", wie soll das gehen? Diese Anweisung muß erst einmal in der Umgangssprache anders formuliert werden, um den Schritt zum Struktogramm und damit zum Programm möglich zu machen. Um zu einer geeigneteren Formulierung zu kommen, spielen wir zunächst noch einmal mit dem Rechenbeispiel und ändern die Reihenfolge der Einzelprodukte:

```
1 0 0 1 0  *  1 0 1 1
--------------------
          1 0 0 1 0
        1 0 0 1 0
      0 0 0 0 0
    1 0 0 1 0
---------------------
1 1 0 0 0 1 1 0
```

Von dieser Darstellung ausgehend läßt sich eine bessere Beschreibung des Verfahrens finden:

Schritt 1

Der Multiplikand wird bitweise nach links geschoben (d.h. mit 2 multipliziert), mit den einzelnen Binärstellen des Multiplikators multipliziert und die Einzelprodukte aufaddiert.

Die einzelnen Binärstellen des Multiplikators wiederum lassen sich gewinnen, wenn man sie bitweise nach rechts schiebt (d.h. durch 2 dividiert). Man kann z.B. das unterste Bit ins Carry-Flag schieben und dann abhängig von diesem eine Addition durchführen. Mit dem vorher gesagten ist eine Formulierung gefunden, die sich in ein Struktogramm umsetzen läßt.

Schritt 2

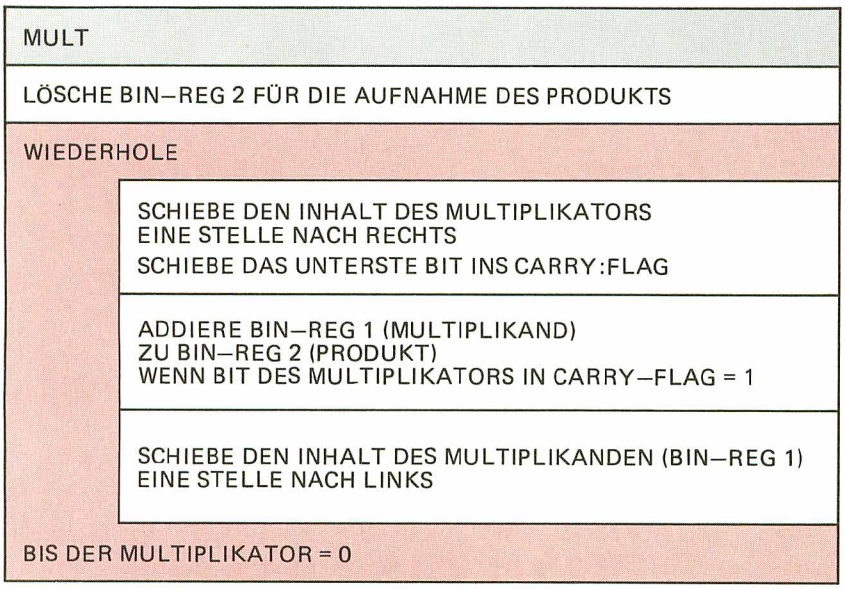


Bild C 83.1

Struktogramm zum Unterprogramm MULT.

Schritt 3

```
MULT:
        LD      HL,BIN_REG2
        CALL    CLEAR           ;BIN_REG2:=0

        LD      C,B             ;Faktor ins C-Register

MULT_SCHLEIFE                   ;wiederhole
        SRL     C               ;C := C/2, CY := Rest
        CALL    C,ADDIERE       ;falls Rest (Carry)=1 :
                                ;BIN_REG2 := BIN_REG2
                                ;          +BIN_REG1
        CALL    SCHIEBE         ;BIN_REG1:= BIN_REG1 *2
```

```
LD    A,C
OR    A                ;bis C=0
JR    NZ,MULT_SCHLEIFE
RET
```

Es wird hier wieder deutlich, daß bei der Lösung einer Programmieraufgabe nicht so sehr das Programmieren selbst ein Problem ist, sondern vielmehr das exakte Vorformulieren der Aufgabe in Umgangssprache. Dabei können Struktogramme eine große Hilfe sein.

Die restlichen Unterprogramme sollen hier nicht alle einzeln mit ihrem kompletten Werdegang beschrieben werden. Es wird lediglich mit einer groben Formulierung angegeben, was geschieht.

Sie sollten diese Formulierungen genauer ausführen und in Struktogramme mit nachfolgenden eigenen Programmen umsetzen. Erst dann sollten Sie diese mit unserer Lösung vergleichen!

ADDIERE:
Addiert BIN_REG1 zu BIN_REG2 für MULT.

SCHIEBE:
Schiebt BIN_REG1 nach links für MULT.

MOV12:
Kopiert BIN_REG2 nach BIN_REG1 für MULT.

WANDLE_DEZ:
Wandelt BIN_REG1 in eine dezimal dargestellte Zahl nach dem gleichen Verfahren wie HOLE_ZAHL. Die Rechnung muß in diesem Fall im BCD-Code erfolgen. Statt mit 10 wird entsprechend der Wertigkeit der Stellen jeweils mit 2 multipliziert (zu sich selbst addiert) und eine Binärstelle zuaddiert.

AUSGABE:
Gibt die in Dezimaldarstellung gewandelte Zahl aus und merkt sich dabei in einem Flag, ob schon Ziffern außer Null vorkamen, um führende Nullen zu unterdrücken.

Z80-Befehle

Das Unterprogramm WANDLE_DEZ soll eine binär dargestellte Zahl in ihre Dezimaldarstellung wandeln. Dazu müssen die einzelnen Stellen der Binärzahl nach dem bei HOLE_ZAHL beschriebenen Verfahren zur Dezimalzahl addiert und diese dazwischen jeweils mit 2 multipliziert werden. Das Ergebnis dieser Rechnungen soll im BCD-Code zur Verfügung stehen. Der Z80-Prozessor verfügt über einen Befehl


```
DAA
```

mit dem das Ergebnis einer Addition oder Subtraktion zweier BCD-Zahlen zu einer BCD-Zahl korrigiert werden kann. Will man also BCD-Zahlen addieren oder subtrahieren, dann wird die Rechnung mit den normalen Befehlen für die binären Operationen ausgeführt und das Ergebnis anschließend mit DAA korrigiert.

Das Unterprogramm AUSGABE verwendet einige Z80-Befehle, die Ihnen vielleicht noch nicht so bekannt sind. Mit

```
SET   Bitnummer, Register
```

bzw.

```
RES   Bitnummer, Register
```

lassen sich einzelne Bits von Registern setzen bzw. löschen. Mit

```
BIT   Bitnummer, Register
```

kann der Zustand eines einzelnen Bits des angegebenen Registers abgefragt werden. Diese Befehle werden vor allem dazu benutzt, um Flags zu setzen, zu löschen und abzufragen.

Im Unterprogramm AUSGABE wird das Bit 0 des C-Registers als Flag verwendet, um sich zu merken, ob schon Ziffern außer Null ausgegeben wurden oder nicht.

Die Befehle

```
RLD
```

bzw.

```
RRD
```

ermöglichen es, ein BCD-Digit von der durch HL adressierten Speicherstelle ins A-Register zu rotieren. Bei RLD

wird das A-Register und (HL) vier Stellen nach links rotiert. Die oberen vier Bits des A-Registers wandern dabei in die vier unteren Bit von (HL). Die vier oberen Bits von (HL) wandern in die vier unteren Bits des A-Registers.

Bei RRD wird das A-Register und (HL) vier Stellen nach rechts rotiert. Die unteren vier Bits des A-Registers wandern dabei in die vier oberen Bits von (HL). Die vier unteren Bits von (HL) wandern in die vier oberen Bits des A-Registers. Einige Beispiele zeigen besser als die Beschreibung, was geschieht:

	vorher		nachher	
	A	(HL)	A	(HL)
RLD	41H	38H	13H	84H
RRD	41H	38H	84H	13H

Nachdem alle Unterprogramme fertig sind, können Sie das komplette Programm (ab Seite F 38) eingeben und feststellen, daß die Berechnung der Fakultäten tatsächlich schneller geschieht als bei einem auf Rechenprogramme spezialisierten Taschenrechner.

Zusammenfassung

Die einzelnen Unterprogramme des Programms zur Fakultätenberechnung werden in drei Schritten realisiert: Formulierung der Aufgabe in Umgangssprache, Aufstellung eines Struktogramms und Umsetzung in Assemblersprache. Dabei ist das exakte Vorformulieren in Umgangssprache oft das größere Problem.

Innerhalb der einzelnen Unterprogramme werden eine Reihe von nicht alltäglichen Z80-Befehlen angewendet: der DAA-Befehl zur BCD-Korrektur, die Befehle SET, RES, BIT zur Flag-Manipulation und die Rotationsbefehle RLD und RRD.

Primzahlen

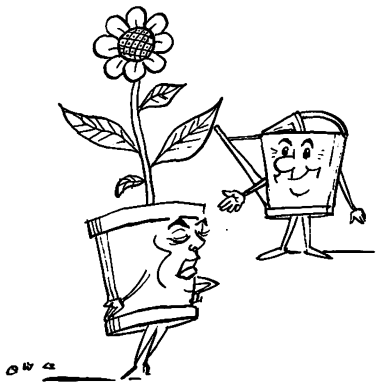
Im Heft 3 wurde am Beispiel der Berechnung von Fakultäten gezeigt, wie man eine Multiplikation programmieren kann. Es liegt nahe, in einem weiteren Beispiel nun auch die Programmierung einer Division anzugehen. Dann stünden mit den nebenbei anfallenden Grundrechenarten Addition und Subtraktion alle vier Grundrechenarten als Unterprogramme zur Verfügung. Mit diesen Unterprogrammen lassen sich bei Bedarf auch komplexere Probleme lösen.

Als Aufhänger oder Programmierbeispiel für die Division haben wir das Berechnen von (oder besser: die Suche nach) Primzahlen gewählt. Wissen Sie noch, was eine Primzahl ist? Ganz einfach - Primzahlen sind alle natürlichen Zahlen, die sich nur durch Eins und sich selbst ganzzahlig teilen lassen. Das sind also

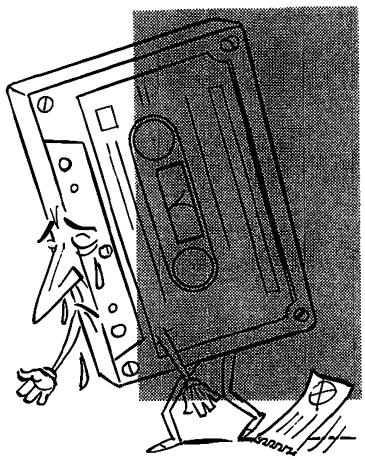
2, 3, 5, 7, 11, 13, 17 ...

aber beispielsweise nicht 9, denn 9 ist durch 3 teilbar, oder nicht 15, denn diese Zahl ist durch 3 und durch 5 teilbar.

Es gibt nun keine Rechenvorschrift, nach der sich die Primzahlen eine nach der anderen berechnen lassen, etwa in der Art, wie das bei den Fakultäten so schön (und leicht zu programmieren) der Fall war. Primzahlen lassen sich nur finden, indem man für jede einzelne infrage kommende Zahl nachprüft, ob diese tatsächlich keine Teiler hat.



Es ist abzusehen, daß diese Aufgabe sehr viel Rechnerei erfordert, eine ideale Sache für den Computer. So ist es auch kein Wunder, daß in den letzten Jahren auf den grossen Rechenanlagen immer wieder eine noch größere Primzahl gefunden wurde, es ist ein richtiger Wettbewerb des besten Rechners und Programms im Gange. Unser Programm kann da sicher keinen Blumentopf gewinnen, da wir auf einem kleinen Computer ein recht primitives Verfahren programmieren. Unser Programm wird aber jedes in einer Hochsprache wie BASIC oder sogar PASCAL geschriebene Primzahlen-Programm das Fürchten lehren können!



Wie die Suche nach den Primzahlen angegangen werden muß, ist im Prinzip klar: es muß jede zu untersuchende Zahl durch alle überhaupt möglichen Teiler dividiert werden. Geht dabei eine Division ohne einen Rest auf, dann ist die Zahl keine Primzahl. Bleibt immer ein Rest, dann ist es eine Primzahl.

Algorithmus

Um zum ersten Schritt bei der Programmierung, der Formulierung der Aufgabe, zu kommen, muß erst einmal untersucht werden, wie die Aufgabe im einzelnen aussieht. Da ist zu klären, was unter "jede zu untersuchende Zahl" und unter "alle möglichen Teiler" genau verstanden werden muß.

Bei den Primzahlen-Beispielen auf der vorangegangenen Seite fällt auf, daß die 2 die einzige gerade Primzahl ist. Ganz klar, denn alle anderen geraden Zahlen sind ja eben deswegen gerade Zahlen, weil sie durch 2 teilbar sind. Damit sind sie aber auch keine Primzahlen. Das Programm läßt sich also dadurch etwas beschleunigen, daß von vornherein nur ungerade Zahlen ab der 3 untersucht und die geraden Zahlen weggelassen werden. Die Primzahl 2 wird dann nicht ausgegeben. Die kann mann sich aber vielleicht noch merken - um festzustellen, daß 2 eine Primzahl ist, braucht man keinen Computer.

Das Ergebnis der angestellten Überlegung: "jede zu untersuchende Zahl" läßt sich genauer beschreiben mit "jede ungerade Zahl ab der Zahl 3".

Für die Teiler sind ähnliche Gedankengänge erforderlich. Ungerade Zahlen können keine geraden Zahlen als Teiler haben (klar - sonst enthielte dieser Teiler und damit auch die geteilte Zahl den Faktor 2 und wäre nicht ungerade). Es müssen also nur alle ungeraden Zahlen ab 3 als Teiler durchprobiert werden. Alle? - natürlich nicht, sonst fände das Programm ja überhaupt kein Ende!

Ein erstes Nachdenken ergibt, daß der Teiler höchstens halb so groß wie die untersuchte Zahl sein kann, sonst funktioniert ja keine Division mehr. Wenn die untersuchte Zahl z.B. 17 ist, genügt es, als Teiler nacheinander 3, 5, 7 zu probieren - 17 durch 9 gibt weniger als 2.

Es geht aber noch mit weniger Versuchen. Nehmen Sie als Beispiel die 27 als untersuchte Zahl: 27 läßt sich durch 3 und auch durch 9 teilen . Gleich beim ersten probieren mit dem Teiler 3 steht fest, daß es keine Primzahl ist. Ein anderes Beispiel ist 35: bevor man feststellt, daß 35 durch 7 teilbar ist, ergibt bereits der Versuch mit dem Teiler 5 das Ergebnis "teilbar, keine Primzahl".

Man kann noch mehr Beispiele anführen, aus denen sich eine Gesetzmäßigkeit ableiten läßt: wenn das Ergebnis der Teilung (der Quotient) kleiner ist als der Teiler, kann die Versuchsreihe abgebrochen werden, einen größeren Teiler gibt es nicht mehr - probieren Sie's aus.

Genau genommen könnte man die Anzahl der Divisionen noch weiter einschränken, es brauchen nicht alle ungeraden Zahlen als Teiler probiert werden. Es würde nämlich auch genügen, alle Primzahlen durchzuprobieren. Wäre eine Nicht-Primzahl Teiler, so wären die in dieser Zahl enthaltenen Primzahlen ebenfalls Teiler. Es kann deshalb kein vorhandener Teiler übersehen werden, wenn man nur alle (schon gefundenen) Primzahlen als Teiler probiert. Dieses Verfahren spart viel Rechenzeit, erfordert aber sehr viel Speicherplatz, weshalb wir uns beim vorliegenden Programm auf das primitivere Verfahren beschränken, bei dem einfach alle ungeraden Zahlen als Teiler probiert werden (falls Sie hier einen Wink mit dem Zaunpfahl für eigene Weiterentwicklung sehen, liegen Sie ganz richtig!).

Das, was im Heft 2 als "Schritt 1" bezeichnet wurde, nämlich die so wichtige Formulierung der Aufgabe, sieht dann so aus:

1. Es müssen alle ungeraden Zahlen ab 3 auf Primeigenschaft geprüft werden.

2. Diese Prüfung erfolgt dadurch, daß durch alle ungeraden Zahlen ab 3 aufwärts dividiert wird, bis eine Division aufgeht oder das Ergebnis (der Quotient) kleiner ist als der untersuchte Teiler. Geht eine Division ohne Rest auf, dann ist die Zahl keine Primzahl, ansonsten ist es eine.

Diese Aufgabe läßt sich (Schritt 2) in die nun schon vertraute Struktogramm-Form bringen, wobei das Prüfen gleich an ein Unterprogramm weiterverwiesen wird.

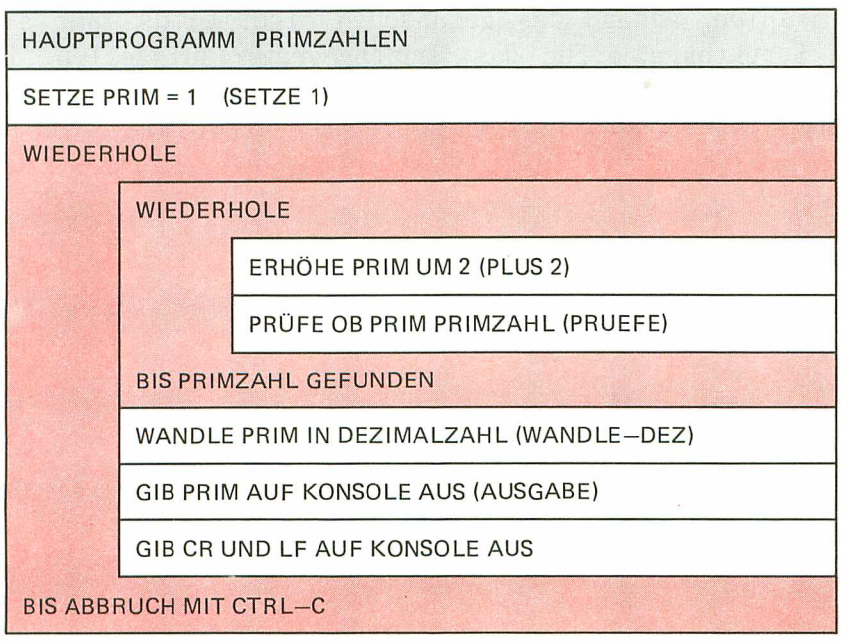


Bild C 89.1

In diesem Struktogramm sind zwei Schleifen ineinander verschachtelt. Dies ist durch die unterschiedliche Intensität der farbigen Unterlegung verdeutlicht.

Für das Wandeln der gefundenen Primzahl in die dezimale Darstellung und die Ausgabe werden die entsprechenden Unterprogramme aus dem Programm zur Berechnung von Fakultäten genommen, von dem übrigens auch noch einige andere Unterprogramme in leicht modifizierter Form zu verwenden sind. Das Unterprogramm PRUEFE soll die möglichen Teiler durchprobieren:

C

90

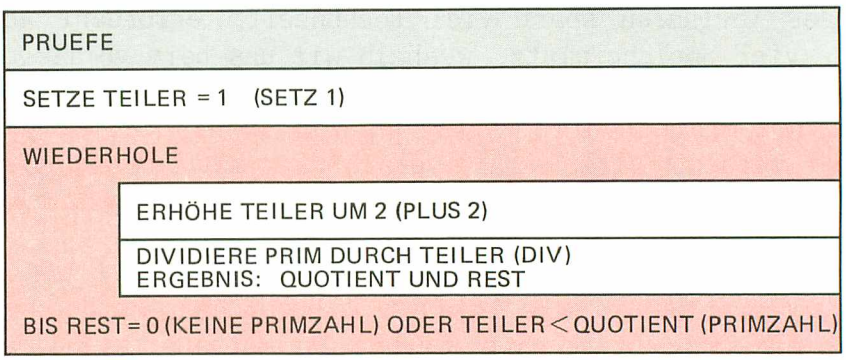


Bild C 90.1

Das Struktogramm zum Unterprogramm PRUEFE, das der eigentliche Kern des Primzahlenprogramms ist.

In diesem Unterprogramm wird nun das Unterprogramm zur Division benötigt. Hier geht es wie beim Unterprogramm zur Multiplikation: bevor man ein Struktogramm und danach ein Assemblerprogramm hinschreiben kann, bedarf es einiger Überlegungen, wie überhaupt dividiert wird und wie dieser Vorgang formuliert werden kann. Bevor diese Aufgabe in Angriff genommen wird, folgt als Abwechslung ein anderes Thema aus den Fachgebieten MODEM und TESTER.

Zusammenfassung

Als weiteres, besonders für die Assemblerprogrammierung geeignetes Programmierbeispiel steht die Berechnung von Primzahlen. Anhand des gefundenen Algorithmus läßt sich ein Struktogramm für das Hauptprogramm aufstellen. Bei den Unterprogrammen ergibt sich die Wiederverwendung für einige Unterprogramme des Fakultäten-Programms, für ein Unterprogramm zur Division sind noch Vorüberlegungen notwendig.

Unterprogramm Division

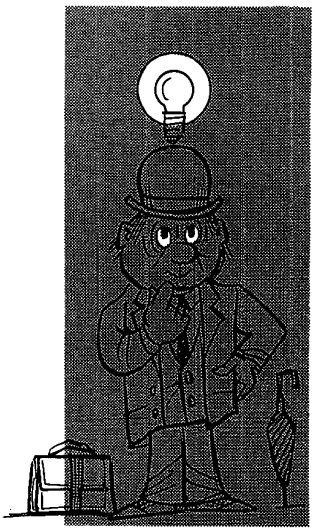
Der Weg, auf dem der Algorithmus für die Division gefunden werden kann, ist der gleiche wie im Heft 2 bei der Multiplikation: man erinnert sich daran, wie eine Division "von Hand" vor sich geht. Dieses Verfahren wird dann anschließend etwas genauer unter die Lupe genommen, um es so zu formulieren, daß sich danach ein Struktogramm aufstellen läßt.

Zuerst einmal soll ein Beispiel zeigen, wie eine Division vor sich geht, etwa 1293 durch 7:

```
1 2 9 3  :  7  =  184
  7               ===
---
  5 9
  5 6
  ---
    3 3
    2 8
    ---
      5
      =
```

Sie werden sicher zustimmen: der Rechengang und das Ergebnis bedürfen keiner Erläuterung. Wie man eine solche Division durchführt, ist völlig klar - jeder kennt die Rechenvorschrift (noch, trotz Taschenrechner).

Allerdings bereitet es erhebliche Kopfschmerzen, diese Rechenvorschrift so auszudrücken, daß sie sich in ein Programm übersetzen läßt.



Das Beispiel allein hat uns noch nicht auf die Sprünge geholfen, deshalb wird es notwendig, noch einen Schritt zurück zu gehen. Was soll denn überhaupt mit der Division erreicht werden? Wir wollen herausfinden, wie oft 7 in 1293 enthalten ist. Anders gesagt (jetzt kommt's!): wie oft man die 7 von 1293 abziehen kann. Geht Ihnen an dieser Stelle ein Licht auf?

Aus der Division wird eine Subtraktion und so gesehen, läßt sich die Aufgabe leicht formulieren:

> Ziehe 7 so oft von 1293 ab, bis das Ergebnis der Subtraktion kleiner als 7 ist. Die Anzahl der Subtraktionen ist das Ergebnis der Division (der Quotient). Das Ergebnis der letzten Subtraktion ist der Rest.

Ein Programm, das von dieser Formulierung ausgeht, müßte ein Unterprogramm zur Subtraktion ("Ziehe ab") und eines zum Vergleich ("Bis kleiner als 7") zur Verfügung haben. Im Hinblick darauf wird ein vorausschauender Programmierer die Formulierung noch ein wenig ändern:

> Ziehe 7 so oft von 1293 ab, bis das Ergebnis negativ wird und mache dann die letzte Subtraktion rückgängig.

"Mache rückgängig" - das klingt eigentlich auch nicht gut, denn das hiesse ja eigentlich "addieren". Also noch eine andere Idee:

> Ziehe 7 so oft von 1293 ab, bis das Ergebnis negativ wird. Speichere das Ergebnis der Subtraktion jeweils zuerst in einem Zwischenregister. Übertrage es dann in ein Hauptregister, wenn feststeht, daß es positiv ist.

Bei den angesprochenen Registern handelt es sich natürlich nicht um CPU-Register, sondern um als Register eingerichtete Speicherbereiche, wie z.B. das 256-Bit-Binärregister auf Seite C 65.

Für einen Programmierer, der den Z80 kennt, läßt sich hinter der letzten Formulierung ein einfacheres Programm vermuten: das Übertragen von einem Zwischenregister in ein Hauptregister wird einfach und schnell mit dem LDIR-Befehl erledigt.

Ein Struktogramm nach dieser Vorschrift sieht dann so aus:

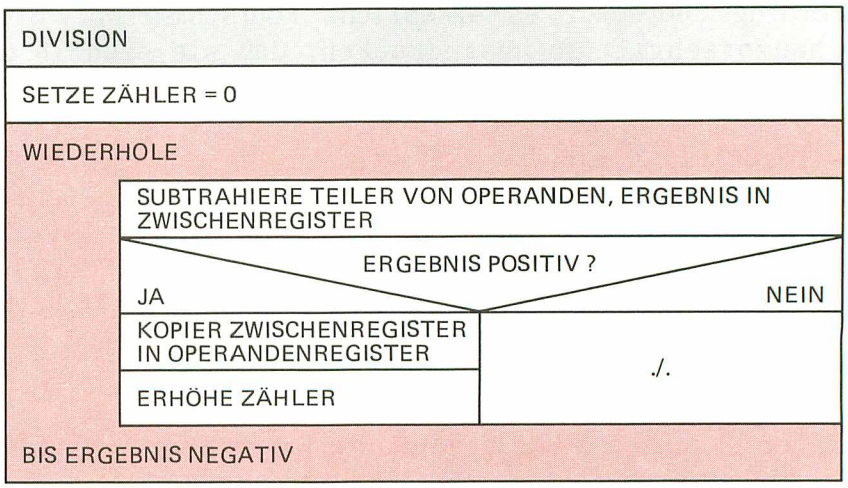


Bild C 92.1

In diesem Struktogramm taucht zum erstenmal die Darstellung einer Verzweigung auf.

Das Ergebnis der Division steht demnach im ZÄHLER, der Rest im Operandenregister.

So schön, so gut - dieses einfache Verfahren hat aber einen gravierenden Nachteil: es ist schrecklich langsam.

Wäre bei der Beispiel-Rechnung 184 mal die 7 von der 1293 subtrahiert worden, dann hätte der Computer eine Weile zu tun gehabt. Vielleicht war der bei der Rechnung "von Hand" verwendete Algorithmus doch besser.

Wir sehen uns die Rechnung nach der eben gemachten Vorarbeit unter dem Aspekt der Subtraktion des Teilers noch einmal genauer an. Dabei steht die Handrechnung auf der linken Seite und rechts davon eine etwas ausführlichere Notierung:

```
1 2 9 3  :  7  =  184          1 2 9 3  :  7  =  1 0 0
  7                ===        - 7 0 0
---                             -------
  5 9                           5 9 3
  5 6                           - 7 0
  ---                           - 7 0
    3 3                         - 7 0
    2 8                         - 7 0
    ---                         - 7 0
      5                         - 7 0
      =                         - 7 0
                                - 7 0              8 0
                                  ---
                                  3 3
                                  - 7
                                  - 7
                                  - 7
                                  - 7                4
                                   --            -----
                                    5            1 8 4
                                    =
```

Auch im rechts gezeigten Verfahren wird im Prinzip subtrahiert, aber nicht ganz so unüberlegt wie im ersten Programmentwurf. Es wird nicht zehnmal die 7 subtrahiert, sondern gleich probiert, ob sich 70 subtrahieren läßt. 70 kann nämlich durch Verschiebung leicht aus 7 gewonnen werden. Konsequenterweise darf man natürlich auch nicht mit 70 anfangen, sondern nimmt im vorliegenden Beispiel als erstes die 700, da diese Zahl noch in 1293 enthalten ist, die 7000 jedoch nicht mehr.

Die Ergebnisse der einzelnen Schritte mit 700, 70 und 7 müssen ihren Stellenwerten entsprechend aufaddiert werden: für jede subtrahierte Zahl 7 in der Hunderterstelle muß 100, für jede 7 in der Zehnerstelle muß 10 und für jede 7 in der Einerstelle muß 1 zum Ergebnis addiert werden.

Erinnern Sie sich noch einmal an die vorangegangene Formulierung der Aufgabe:

Ziehe 7 so oft von 1293 ab, bis das Ergebnis negativ wird. Speichere das Ergebnis der Subtraktion jeweils zuerst in einem Zwischenregister und übertrage es erst in ein Hauptregister, wenn feststeht, daß es positiv ist.

Diese Formulierung muß nach den neuen Erkenntnissen mit der erweiterten Rechnung etwas ergänzt werden:

1.)Schiebe die 7 so oft nach links, bis die Zahl größer als 1293 wird. Merke die Anzahl der Schiebeschritte.

2.)Ziehe die nach links geschobene 7 so oft von 1293 ab, bis das Ergebnis negativ wird. Speichere das Ergebnis der jeweiligen Subtraktion zuerst in einem Zwischenregister. Übertrage es erst dann in ein Hauptregister, wenn feststeht, daß es positiv ist. Schiebe bei negativem Ergebnis den Teiler (die 7) wieder eine Stelle nach rechts. Wiederhole 2.) so oft, bis alle Schiebeschritte von 1.) rückgängig gemacht sind.

So sieht das schon ganz ordentlich aus, es fehlt aber noch etwas: das Ergebnis der Division. Es müssen noch die einzelnen Ergebnisse, die sich jeweils bei 2.) ergeben, ihren Stellenwerten entsprechend aufaddiert werden. Das ist nicht schwer: die Einzelergebnisse werden einfach von rechts in ein Register geschoben, dann stehen sie am Schluß automatisch richtig. Sie können das am Beispiel nachprüfen:

```
    1    <-  1

  1 8    <-  8

1 8 4    <-  4
```

Wo aber stehen die Einzelergebnisse von 2.) denn? Bei einer Rechnung im Dezimalsystem müßten sie auch noch in irgendeinem Zähler festgehalten werden. Jedes Einzelergebnis kann ja von 1 bis 9 lauten - im Dezimalsystem.

Wenn man das ganze Verfahren in das Dualsystem überträgt, vereinfacht sich die Sache: das Einzelergebnis jeder Subtraktion nach 2.) lautet entweder 0 oder 1. Anders gesagt: jede Subtraktion von 2.) kann nur einmal erfolgen und jedes Teilergebnis besteht nur aus einem Bit (0 oder 1), das jeweils von rechts in das Ergebnisregister geschoben werden muß.

Frage:

1. Warum gibt es beim Rechnen im Dualsystem nur eine einzige Subtraktion unter 2.)? Machen Sie sich das mit einem Beispiel klar!

(Die Antwort zu dieser Frage finden Sie auf Seite G 13.)

Im Übrigen läßt sich schon jetzt absehen, welche Unterprogramme das Divisionsprogramm benötigen wird: es sind Unterprogramme zur Subtraktion, zum Übertragen, zum Links- und zum Rechtsschieben von Zahlen notwendig. Mit diesen Unterprogrammen sieht das Struktogramm für das Divisionsprogramm dann so aus:

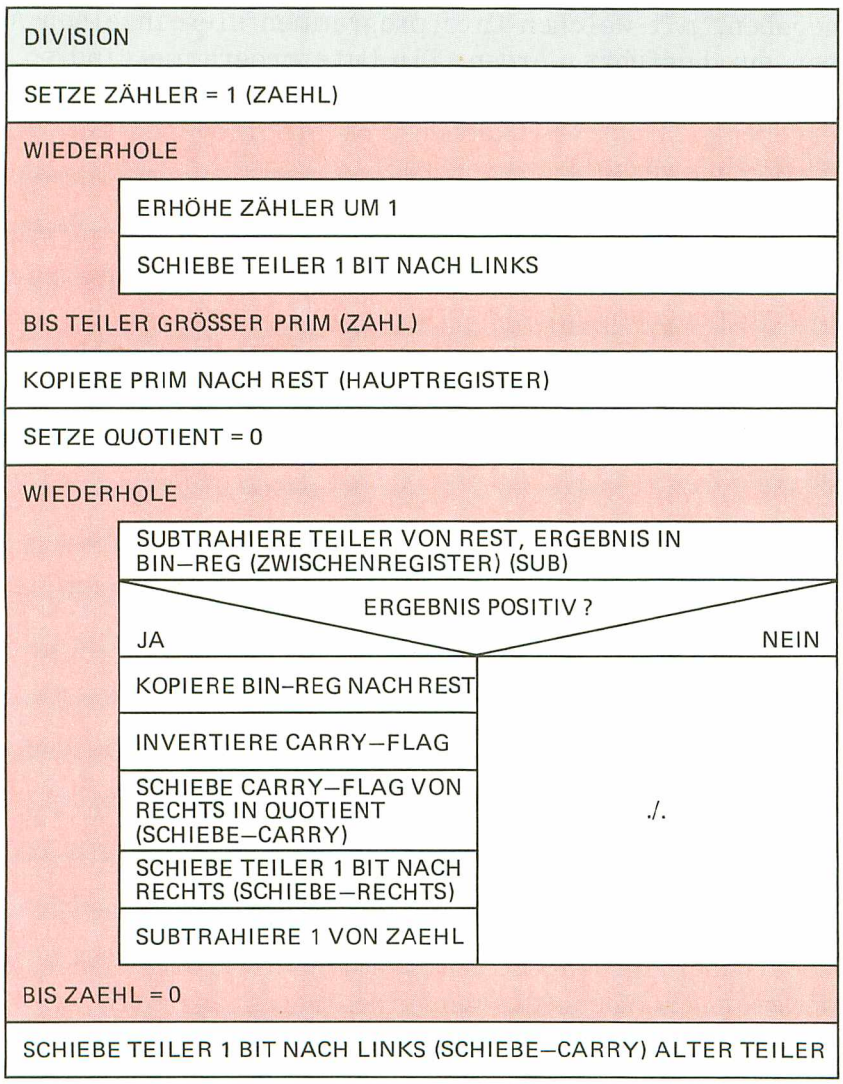


Bild. C 95.1

Die in Klammern stehenden Ausdrücke deuten entweder auf die Unterprogramm-Namen im Listing oder auf Bezeichnungen im Lehrbrief-Text hin.

Als erstes fällt im Struktogramm auf, daß der Zähler, mit dem die Anzahl der Verschiebeschritte festgehalten wird, zu Beginn des Ablaufs nicht auf Null, sondern auf Eins gesetzt ist. Damit wird erreicht, daß die zweite Schleife des Programms auch dann noch einmal durchlaufen wird,

wenn der Teiler schon wieder an seiner ursprünglichen Stelle steht. Im vorher erwähnten Beispiel aus der Dezimalrechnung hieße das: die Rechnung ist ja nicht zu Ende, wenn die 7 wieder an ihrer alten Stelle steht, sondern erst dann, wenn auch für die 7 geprüft wurde, wie oft sie enthalten ist.

Nach diesem letzten Durchgang steht der Teiler allerdings eine Stelle zu weit rechts, da er einmal mehr nach rechts als nach links geschoben wurde. Das wird einfach korrigiert: er wird wieder eine Stelle nach links geschoben - die eigentlich verlorene niederste Stelle des Teilers steht ja noch im Carry-Flag (von SCHIEBE_RECHTS dorthin geschoben). Ein Unterprogramm zum Einschieben des Carry-Flags steht innerhalb der Division ohnehin zur Verfügung.

Im Struktogramm (Bild C 95.1) wird jeweils in Klammern angegeben, mit welchen Unterprogrammen die einzelnen Aufgaben durchgeführt werden. Die Unterprogramme sind so gestaltet, daß ihnen jeweils die Adressen der Operanden übergeben werden, mit denen sie arbeiten sollen. Damit sind sie universell verwendbar.

Aus Platzgründen sind die Unterprogramme nicht auch noch als Struktogramme ausgeführt.Schauen Sie sich die fertigen Unterprogramme im Listing (TITEL Primzahlen, Programm 1, ab Seite F 45) genau an: wenn Ihnen dabei alles klar ist, brauchen Sie nicht unbedingt ein Struktogramm. Gibt es jedoch irgendwo eine Unklarheit bei einem der Unterprogramme, dann gehen Sie den bewährten Weg, wie er Ihnen gezeigt wurde:

1. Aufgabe in Worten klar formulieren!

2. Struktogramm entwerfen!

3. Programm schreiben!

Grundsätzlich ist bei den Unterprogrammen eines zu beachten: Additionen, Subtraktionen und Linksschieben müssen mit dem niederwertigsten Byte beginnen. Dieses ist bei den im Speicher eingerichteten Binärregistern immer das letzte Byte. Auf dieses letzte Byte (bzw. das Byte danach) müssen daher in den Unterprogrammen die Zeiger erst einmal gesetzt werden: die Länge des Registers wird zur Zeigerposition addiert. Außerdem muß in den erwähnten Unterprogrammen der Zeiger vor jedem Schritt dekrementiert werden. Er steht dann vor dem ersten Schritt tatsächlich auf der niedersten Stelle des Registers. In den Unterprogrammen, die mit der Verarbeitung des höchsten Byte beginnen sollen, kann die Verschiebung des Zeigers am Anfang natürlich entfallen und der Zeiger muß nach jedem Schritt inkrementiert werden.

Genug des Überlegens und des scharfen Nachdenkens (...wie soll denn das Laufen, warum steht da ein xx-Befehl ...?): machen Sie eine Pause zum Spielen, tippen Sie das Programm ein (ab Seite F 45) und lassen Sie es laufen. Sie werden feststellen, daß es erstaunlich schnell ist. Das hatten wir am Anfang des Heftes ja schon gesagt, und wir haben auch Vergleiche angestellt: es ist etwa viermal schneller als ein entsprechendes PASCAL-Programm auf einem 16-Bit-Computer!

Die Anzahl der Stellen wurde im vorliegenden Programm zunächst so gewählt, daß die Genauigkeit ungefähr der eines BASIC-Interpreters entspricht:

```
DEZ_STELL   EQU   10    ;maximal 10 Dezimalstellen
                        ;2 hoch 32 = 4*10 hoch 9
BIN_STELL   EQU   32    ;32-Bit-Binärzahl

BIN_BYTE    EQU   BIN_STELL/8
                        ;Anzahl Byte der Binärzahlen
```

Was in einem BASIC-Programm garnicht geht, macht in diesem Programm überhaupt keine Schwierigkeiten: Sie können die Stellenzahl reduzieren und erhalten dann eine höhere Rechengeschwindigkeit. Sie können aber auch die Stellenzahl erheblich grösser wählen und damit Primzahlen in Regionen suchen, die einem BASIC-Programm überhaupt nicht mehr zugänglich sind. Probieren Sie das einmal aus: ändern Sie die Stellenzahl BIN_STELL (auf Vielfache von 8) und beobachten Sie, wie sich die Rechengeschwindigkeit verändert. Wählen Sie DEZ_STELL jeweils so, daß die größte Binärzahl noch im Dezimalregister Platz hat. DEZ_STELL kann auch beliebig grösser gewählt werden, dann allerdings dauert die Binär-Dezimal-Umwandlung länger.

Fragen:

Die Ausgabe der Primzahlen wäre leichter zu verfolgen, wenn die Darstellung auf dem Bildschirm nicht so schnell "weglaufen" würde, wenn z.B. mehr als eine Primzahl pro Zeile ausgegeben würde.

1. Wie muß zunächst das Unterprogramm zur Ausgabe geändert werden, damit führende Nullen bei der Ausgabe nicht unterdrückt, sondern als Zwischenräume ausgegeben werden ? (Die Ausgabe der Primzahlen erfolgt dann rechtsbündig.)

2. Wie muß anschließend das Hauptprogramm geändert werden, so daß erst nach einer gewissen Anzahl von ausgegebenen Zahlen eine neue Zeile angefangen wird ? Definieren Sie hierzu einen entsprechenden Zähler. Er kann sich in einem Register befinden, das dann während der Primzahlensuche auf einem Stack gerettet wird. Er kann sich auch in einer Speicherstelle im RAM befinden, die bei der Primzahlensuche nicht zerstört wird.

Die Antworten zu diesen Fragen finden Sie im Listing zum Programm 2 auf Seite F 53. Wie immer an dieser Stelle der mahnend erhobene Zeigefinger: tippen Sie nicht gleich unseren Entwurf ein, sondern versuchen es zuerst selbst!

Übrigens, falls Sie es nicht schon gemerkt haben, unser Programm hat auch noch einen richtigen Fehler: es läßt nicht nur die Primzahl 2, sondern auch die 3 aus. Dies liegt ganz einfach daran, daß für die 3 als erster Teiler 3 ausprobiert wird. Diese Division geht auf und das Programm folgert daraus haarscharf und falsch, daß die 3 keine Primzahl ist. Diesen Fehler des Unterprogramms PRUEFE zu beheben, ist nicht ganz einfach. Sie können es aber trotzdem probieren!

Zusammenfassung

Das Unterprogramm für die Division wird auf dem gleichen Weg gefunden wie dasjenige zur Multiplikation im Heft 3. Man macht sich klar, was bei einer Division überhaupt geschieht, formuliert dies klar und entwirft danach ein Struktogramm. Wesentlich ist, daß die Division auf Subtraktionen zurückgeführt wird.

Für das Unterprogramm zur Division und für das gesamte Programm zur Primzahlensuche sind eine Reihe von weiteren Unterprogrammen notwendig, die zum Teil bereits aus den vorangegangenen Programmen fertig vorliegen. Das entstandene lauffähige Programm ist wesentlich schneller als ein entsprechendes PASCAL- oder BASIC-Programm.

Frage:

1. Wie sieht eine Programmänderung aus, die bewirkt, daß es nicht bei 3, sondern bei einer beliebig eingegebenen Zahl mit der Primzahlensuche beginnt?

(Die Antwort zu dieser Frage finden Sie auf Seite G 15.)

Primzahlen- Ausgabe mit Drucker

Dieses Kapitel setzt, wie schon erwähnt, voraus, daß ein Drucker vorhanden ist. Wenn nicht, befassen Sie sich trotzdem mit den Programm-Änderungen (und außerdem folgt auch noch eine Programm-Variante für Multi-Tasking ohne Drucker-Ausgabe).

Nehmen Sie sich das Programm 2 (Seite F 52) vor und ändern Sie das Ausgabeprogramm und das Hauptprogramm so ab, daß alle Ausgaben auf dem Drucker erfolgen. Beachten Sie dabei, daß CR und LF (neue Zeile) nicht mehr mit der Systemfunktion 9 ausgegeben werden können. Vergleichen Sie danach (danach!) Ihre Änderungen mit dem Programm 3:

Im Unterprogramm AUSGABE:

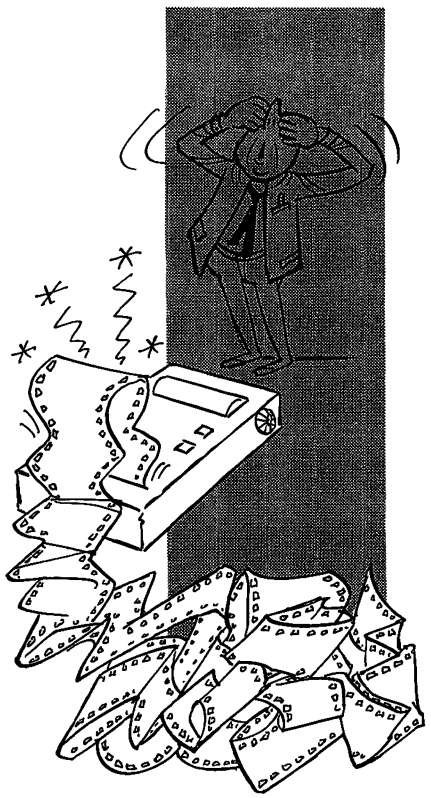
```
Auf Seite F 69    LD  C,DRUCKF
```

Bei den Definitionen:

```
Auf Seite F 61    DRUCKF   EQU  5
```

Im Hauptprogramm PRIMZAHLEN

```
Auf Seite F 63    LD   E,CR
                  LD   C,DRUCKF
```



Mit diesen Änderungen läuft das Primzahlen-Programm wie zuvor, nur werden die Primzahlen auf einem angeschlossenen Drucker ausgegeben. Wenn Sie das probieren, werden Sie das Programm irgendwann, bevor das Druckerpapier ausgeht, mit CTRL-C anhalten. Dies weist auf ein weiteres Problem hin. Bei einem gleichzeitigen Betrieb von Primzahlenprogramm und Schreiben im Editor funktioniert die Abfrage auf CTRL-C zum Abbruch des Programms nicht mehr ohne weiteres. Es wäre ja nie klar, ob ein CTRL-C ein Scroll-Befehl für den Editor oder ein Abbruch-Befehl für das Primzahlen-Programm sein soll.

Es bleibt demnach zunächst nichts anderes übrig, als die Abfrage auf CTRL-C aus dem Programm zu entfernen. Es wird dadurch vorläufig zu einem Endlos-Programm, das erst durch einen System-Reset wieder angehalten werden kann.

Ihre nächste Aufgabe ist also, die Abfrage auf CTRL-C aus dem Primzahlen-Programm zu entfernen. Es darf danach keine Konsolfunktionen mehr enthalten, weshalb Sie auch deren Definitionen am Anfang des Programms entfernen können.

Multi-Tasking

Nach diesen Vorbereitungen sind die Fronten klar: den Systemprogrammen sind Bildschirm und Tastatur zugeordnet, dem Primzahlen-Programm der Drucker. Damit wird es grundsätzlich möglich, daß Systemprogramme und das Primzahlenprogramm gleichzeitig ablaufen.

C
100

Die Aufteilung der Rechenzeit kann nach einem einfachen Schema erfolgen. Immer dann, wenn das Betriebssystem

entweder die Tastatur auf eine gedrückte Taste abfragt (also das BIOS-Unterprogramm Konsolstatus aufruft),

oder eine Taste einlesen will (also das BIOS-Unterprogramm Konsoleingabe aufruft),

muß ein noch zu schreibendes Programm (mit Hilfe des BIOS-Unterprogramms Konsolstatus) nachsehen, ob tatsächlich eine Tasteneingabe vorliegt. Wenn ja, wird das dem rufenden Programm mitgeteilt. Wenn nein, dann können einige Schritte des Primzahlen-Programms ausgeführt werden.

Das BIOS-UNTERPROGRAMM Konsolstatus wird also durch ein Programm ersetzt, das so abläuft:

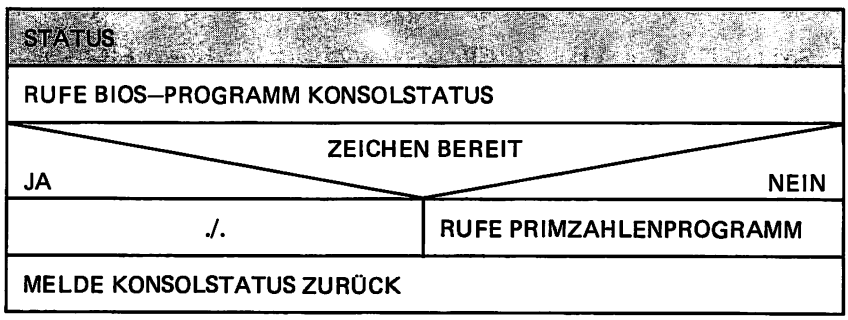


Bild C 100.1

Das Struktogramm zum Unterprogramm STATUS.

Das sieht ganz einfach aus. Aber, das Kästchen "Rufe Primzahlen-Programm" kann natürlich nicht einfach durch einen CALL-Befehl realisiert werden. Vielmehr müssen vorher alle Register der CPU gerettet werden, die innerhalb des Primzahlen-Programms möglicherweise geändert werden. Anschließend muß auch der zu den Systemprogrammen gehörende Stackpointer gespeichert und sein zum Primzahlen-Programm gehörender Kollege geladen werden. Schließlich müssen alle zum Primzahlen-Programm gehörenden Register-Inhalte geladen werden. Mit alledem ist eine komplette Umschaltung auf das Primzahlenprogramm erfolgt und es kann dort fortgesetzt werden, wo es zuletzt abgebrochen wurde.

Als Programm sieht diese Umschaltung so aus:

```
STATUS:
        CALL    BIOS-ST         ;Konsolstatus abfragen
        OR      A               ;Rücksprung, wenn Zeichen be-
        RET                     ;reit
TRANSFER_PRIM                   ;sonst Primzahlen-Programm:
        PUSH    BC              ;Register retten
        PUSH    DE
        PUSH    HL
        PUSH    IX
        LD      (SP-SYS)        ;Stackpointer (System) retten

        LD      SP,(SP_PRIM)    ;Stackpointer (Prim) holen
        POP     HL              ;4 Registerpaare laden
        POP     DE
        POP     BC
        POP     AF
        RET                     ;und Rücksprung an letzte
                                ;Adresse
```

(Im Programm-Listing finden diesen Teil auf Seite F 62)

Das Primzahlenprogrammm darf aber nicht lange fortgesetzt werden. Es muß bald wieder ins System zurückkehren, damit dieses nichts von der Schummelei merkt. Dazu wird ein Programm aufgerufen, das die Register alle miteinander wieder zurücktauscht:

```
TRANSFER_SYS:
        PUSH    AF              ;4 Registerpaare für Prim
        PUSH    BC              ;retten
        PUSH    DE
        PUSH    HL
        LD      (SP_PRIM),SP    ;Stackpointer (Prim) retten

        LD      SP,(SP_SYS)     ;Stackpointer (System) holen
        POP     IX              ;4 Registerpaare laden
        POP     HL
        POP     DE
        POP     BC
        XOR     A               ;=keine Taste bereit
        RET                     ;zurück ins System
```

Das Systemprogramm wird fortgesetzt, als ob nichts gewesen wäre.

Das vorher beschriebene Programm TRANSFER_SYS ersetzt das Unterprogramm für Konsolstatus. Wenn keine Taste bereit ist, schaltet es auf das Primzahlen-Programm um. Damit dies auch beim Aufruf des Konsoleingabeprogramms des BIOS funktioniert, wird auch an dessen Stelle ein anderes Unterprogramm eingefügt:

```
INPUT:                          ;wiederhole:
        CALL    STATUS          ;Status abfragen (mit PRIM)
        JR      Z,INPUT         ;bis Zeichen bereit
        JP      BIOS_IN         ;Zeichen holen
```

Dieses Unterprogramm ruft solange das Konsolstatus-Programm auf (und damit eingeschlossen auch das Primzahlen-Programm), bis eine Taste betätigt wird. Dann springt es in das BIOS-Unterprogramm für Konsoleingabe.

Das Ganze läuft erst, wenn die beiden Eintragungen für die Unterprogramme Konsolstatus und Konsoleingabe in der BIOS-Sprungtabelle durch Sprünge zu den beiden eigenen Programmen (STATUS und INPUT) ersetzt werden. Es müssen also die beiden folgenden Sprungbefehle an der richtigen Stelle der BIOS-Sprungtabelle stehen:

```
        JP      STATUS
        JP      INPUT
```

Gleichzeitig benötigen unsere eigenen Programme die alten Sprungadressen aus der BIOS-Sprungtabelle. Dazu werden einfach im Hauptprogramm die beiden Sprungbefehle der BIOS-Sprungtabelle mit den beiden mit Label versehenen Befehlen unserer eigenen Sprungtabelle getauscht:

```
BIOS_ST:
        JP      STATUS          ;werden von Haupt gegen die
BIOS_IN:                        ;beiden der BIOS-Sprung-
        JP      INPUT           ;tabelle getauscht
```

Nach dem Austausch stehen dann in der BIOS-Sprungtabelle wie gewünscht die beiden Befehle JP STATUS und JP INPUT, in unserer eigenen Sprungtabelle an der Stelle BIOS_ST der von der BIOS-Sprungtabelle geholte Sprung in das BIOS-Unterprogramm für Konsolstatus und bei BIOS_IN derjenige für das BIOS-Unterprogramm Konsoleingabe.

Das hat übrigens einen recht praktischen Nebeneffekt: das Programm wird, nachdem es durch RUN gestartet wurde, die beiden Tabellen vertauschen und damit den Hintergrundbetrieb des Primzahlen-Programms auslösen. Wird das Programm dann mit RUN noch einmal gestartet, dann wird es die beiden Tabellen wieder zurücktauschen und so den Hintergrundbetrieb anhalten.

Das Hauptprogramm, das den eigentlichen Tausch durchführt, findet zuerst einmal die Adresse des Konsolstatus-Unterprogramms in der BIOS-Sprungtabelle durch Erhöhen der Warmstartadresse um 3. Die Erhöhung um 3 wird hier nicht als drei einzelne INC HL - Befehle geschrieben. Es steht vielmehr ein einziger INC HL - Befehl zwischen den Pseudobefehlen REPT 3 und ENDM, damit wird eine dreifache

Assemblierung des Befehls INC HL erreicht (diese Pseudobefehle werden anschließend noch vorgestellt).

```
HAUPT:
        LD      HL,(WSTART+1) ;Adresse Warmstart in BIOS-
        REPT    3             ;Tabelle + 3 = Adresse Kon-
        INC     HL            ;solstatus
        ENDM
```

Anschließend werden die beiden Tabellen getauscht:

```
        LD      DE,BIOS_ST    ;Zeiger DE auf interne
                              ;Sprungtabelle
        LD      B,6           ;Zähler für 6 Byte setzen
HAUPT_SCHLEIFE:
        LD      C,(HL)        ;Byte aus BIOS-Tabelle mer-
        LD      A,(DE)        ;ken, ersetzen durch Byte aus
        LD      (HL),A        ;interner Tabelle und in in-
        LD      A,C           ;terner Tabelle speichern
        LD      (DE),A
        INC     DEC           ;Zeiger für interne und BIOS-
        INC     HL            ;Tabelle erhöhen
        DJNZ    HAUPT_SCHLEIFE;Zähler (B) dekrementieren,
                              ;bis Zähler = 0
```

Schließlich muß noch der Start des Primzahlenprogramms vorbereitet werden:

```
        LD      SP,STACK      ;Stack und Stackpointer für
                              ;Primzahlen-Programm vorbe-
                              ;reiten:
        LD      HL,START      ;Startadresse auf Stack
        PUSH    HL            ;(RET-Sprung nach START!)
        PUSH    AF            ;4 Registerpaare auf Stack
        PUSH    BC
        PUSH    DE
        PUSH    HL
        LD      (SP_PRIM),SP  ;Stackpointer ablegen
        JP      WSTART        ;Warmstart
```

Assembler-Anweisungen

So ganz nebenbei wurde zuvor im Programm ein bisher noch unbekannter Pseudobefehl verwendet, nämlich REPT/ENDM. Dies ist nicht der einzige Pseudobefehl des Assemblers, der bisher nicht verwendet wurde. Wir stellen Ihnen hier nun die restlichen bisher nicht verwendeten Pseudobefehle vor.

```
        RPT     <data>
        .
        .
        ENDM
```

Die zwischen REPT und ENDM stehenden Befehle und Anweisungen werden <data>-mal assembliert. <data> kann ein Ausdruck zwischen 0 und 255 sein. Im vorangegangenen Programm war es 3: der Befehl INC HL wurde dreimal assembliert.

<label> ASET <ausdruck>

Dem Symbol <label> wird der Wert <ausdruck> zugewiesen. Dieser Pseudobefehl entspricht der EQU-Anweisung mit einem Unterschied: mit ASET kann der Wert des Symbols <label> auch wieder geändert werden, was bei EQU zu einer Fehlermeldung führen würde. Ein mit ASET definiertes Symbol kann also an verschiedenen Stellen des Assemblerprogramms mit unterschiedlichen Werten belegt werden.

Ein praktisches Beispiel soll zeigen, wie zusammen mit dem REPT-Pseudobefehl eine einfache Möglichkeit entsteht, um etwa alle Buchstaben des Alphabets im Speicher abzulegen:

```
BUCHST ASET   'A'          ;Beginn mit 'A'
       REPT   1+'Z'-'A'
       DB     BUCHST      ;Buchstabe in Speicher able-
BUCHST ASET   BUCHST+1     ;gen und nächsten Buchstaben
       ENDM                ;erzeugen
```

Statt ASET kann auch der gleichlautende Pseudobefehl DEFL verwendet werden. Der Assembler unseres Betriebssystems verfügt bei einigen Pseudobefehlen über mehrere Synonyme, um mit verschiedenen anderen Assemblern gleichzeitig kompatibel zu sein. Man kann also im eben gezeigten Beispiel auch schreiben:

```
BUCHST DEFL  'A'
usw.
```

Vielleicht läßt sich DEFL leichter merken, es kommt von DEFiniere Local.

Weitere Synomyme sind:

```
DEFB für DB
DEFW für DW
DEFS für DS
```

Zu diesem Trio DB, DW und DS gibt es auch noch eine Ergänzung:

```
        DC    'Zeichenkette'
```

Dieser Pseudobefehl legt die in Anführungszeichen stehenden Zeichen wie der Pseudobefehl DB als ASCII-Zeichen im Speicher ab. Das letzte Zeichen der Kette erhält jedoch als Ende-Markierung das höchste Bit gesetzt:

```
DC      'ABCD'
```

ergibt die Bitfolge 41H, 42H, 43H, 0C4H im Speicher.

Vielleicht benutzen Sie Ihren Computer als Entwicklungssystem und schreiben darauf ein Programm, das später selbständig im EPROM eines kleinen Mikroprozessorsystems laufen soll. Dieses Programm muß dort bei der Adresse 0 starten. Es ist aber nicht möglich, es im vorliegenden System an der Adresse 0 zu assemblieren, da die untersten Adressen Systemparameter enthalten und nicht überschrieben werden dürfen. Hier schafft ein weiterer Pseudobefehl Abhilfe:


```
.PHASE <ausdruck>
   .
   .
.DEPHASE
```

Die zwische .PHASE und .DEPHASE stehenden Zeilen werden so assembliert, daß sie an der Adresse <ausdruck> lauffähig sind und nicht an der Adresse, an der sie assembliert wurden. Anders gesagt: der Assembler legt das Programm zwar an der mit ORG bestimmten Adresse ab. Er legt die Symboltabelle aber so an, als ob er das Programm an der Stelle <ausdruck> assemblieren würde. Das Programm für das vorher erwähnt EPROM müßte also mit .PHASE 0 assembliert werden.

```
      PAGE
```
oder
```
      PAGE   <data>
```

weisen den Assembler an, bei der Druckausgabe eine neue Seite zu beginnen, z.B. am Anfang eines wichtigen Unterprogramms. Wird ein Wert <data> mitangegeben, dann beginnt der Assembler nach jeweils <data> Zeilen wieder eine neue Seite.

```
      END
```
oder
```
      END    <ausdruck>
```

geben das Ende eines Programms an, danach stehende Befehle oder Anweisungen werden nicht mehr assembliert. Wenn mit der END-Anweisung <ausdruck> mit angegeben ist, wird der Programmzähler automatisch auf die Adresse <ausdruck> gesetzt, sonst steht er auf dem Standardwert 0100H.

Multi-Tasking

Nach dem Abstecher zu den Pseudobefehlen des Assemblers, ausgelöst durch die Verwendung von REPT im Programm, wieder zu letzerem zurück. Geben Sie das Programm 3 (Seite F 60), das sämtliche zuvor besprochenen Ergänzungen enthält, in Ihren Computer ein und lassen Sie es assemblieren. Sie wissen, daß im Programm einige riskante Manöver vollführt werden: es wird in Systemadressen eingegriffen. Achten Sie deshalb darauf, daß die Prüfsummen bei der Assemblierung stimmen.

Der Profi-Status braucht übrigens nicht eingeschaltet zu werden, wenn auch das Programm verbotenerweise direkt in Systemadressen eingreift. Das Betriebssystem hat über das Programm keine Kontrolle und wird daher auch keinen Fehler melden.

Starten Sie das Programm mit RUN, wenn ein Drucker angeschlossen ist. Während auf dem Drucker die Primzahlen laufend ausgegeben werden, können Sie mit Ihrem System weiterarbeiten, den Editor aufrufen oder den Tester sowie das Modemprogramm benutzen. Sie dürfen nur keine Programme assemblieren, damit würde das im Hintergrund laufende Programm überschrieben werden. Natürlich dürfen Sie das laufende Programm auch nicht im Tester auseinander nehmen und müssen auch mit PACK-Kommandos vorsichtig sein. Mit RUN läßt sich das im Hintergrund laufende Programm jederzeit anhalten und wieder starten, es fängt dann wieder von vorn an.

Der Befehl für den Rücksprung in das rufende Systemprogramm, CALL TRANSFER_SYS, ist zwar nur an einer einzigen Stelle des Primzahlenprogramms vorgesehen. Dafür aber an einer, die sehr oft durchlaufen wird: in einem Schiebe-Unterprogramm, das seinerseits von verschiedenen anderen Programmen aufgerufen wird. Damit ist sichergestellt, daß der Prozessor nie sehr lange im Primzahlen-Programm bleibt, wodurch seine "Nebenbeschäftigung" z.B. beim Editieren kaum auffällt (es muß schon recht schnell geschrieben werden, um die Pausen zu merken).

Wenn Sie aber wirklich so schnell schreiben oder sonstwie das Gefühl haben, der Computer würde infolge der Hintergrundarbeit deutlich langsamer arbeiten, dann können Sie den Rücksprungbefehl noch an anderen Stellen zusätzlich in das Programm einfügen.

Umgekehrt kann er aber auch an eine Stelle verlegt werden, wo er weniger oft zu einem Rücksprung in das Systemprogramm führt, etwa im Unterprogramm PRUEFE. Probieren

Sie das einmal aus: zunächst wird im Unterprogramm SCHIEBE_CARRY der Rücksprung ins System entfernt (Seite F66):

```
SCHIEBE_SCHL:                   ;wiederhole:
        DEC     HL              ;  Zeiger dekrement.,
        RL      (HL)            ;  byteweise schieben
        DJNZ    SCHIEBE_SCHL    ;  Zähler (B) dekrem.
        POP     BC              ;bis Zähler=0
-->                 <--
        RET
```

Danach müssen Sie am Ende des Unterprogramms PRUEFE (Seite F 64) den Rücksprung einfügen:

```
PRUEFE_SCHL:                    ;wiederhole:
        LD      HL,TEILER
        CALL    PLUS2           ;  erhöhe TEILER um 2
        CALL    DIV             ;  Dividiere PRIM/TEILER
                                ;  Ergebnis: QUOTIENT, REST
        LD      HL,REST         ;bis REST gleich Null
        CALL    NULL            ;  --> keine Primzahl
        RET     Z               ;  --> Carry=0
        LD      IX,BIN_REG      ;oder TEILER >= QUOTIENT
        LD      DE,TEILER       ;  TEILER - QUOTIENT
        LD      HL,QUOTIENT     ;  Carry=Vorzeichen Ergebnis
        CALL    SUB             ;  Carry=1: TEILER<QUOTIENT
                                ;  Carry=0: TEILER>=QUOTIENT
        JR      C,PRUEFE_SCHL   ;  --> Primzahl
        SCF                     ;  --> Carry:=1

-->     CALL    TRANSFER_SYS    <--

        RET
```

Nach dieser Veränderung ist deutlich zu merken, wie das System auf einmal viel langsamer auf die Tastatureingaben reagiert.

Wenn Sie keinen Drucker anschließen können, läßt sich, wie schon gesagt, das Multi-Tasking nicht so schön demonstrieren. Um Ihnen trotzdem einen Begriff davon zu geben, folgt eine Programmerweiterung, bei der das im Hintergrund arbeitende Programm seine Primzahlen in einen Speicherbereich schreibt. Dort können Sie dann hin und wieder nachschauen, wie weit es gekommen ist, während im Vordergrund anderes geschieht.

Sie sollten aber, auch wenn Sie das Programm 3 mit dem angeschlossenen Drucker schon ausprobiert haben, trotzdem die nachfolgende Ergänzung mitmachen. In diesem Programmierbeispiel wird nämlich wieder einmal die elegante bedingte Assemblierung angewendet und im einem der beiden Fälle läuft die Sache mit dem Drucker ebenso wie zuvor.

Die eigentliche Druckerausgabe wird in der neuen Version nicht mehr über die entsprechende Systemfunktion abgewikkelt, sondern über ein Unterprogramm DRUCKE_ZEICHEN. Dieses Unterprogramm kann mit bedingter Assemblierung auf zwei Weisen assembliert werden: wenn ein Drucker angeschlossen ist, gehen die Ausgaben (die gefundenen Primzahlen) an den Drucker, wie im Programm 3 auch.

Wenn kein Drucker vorhanden ist, werden die Ausgaben zyklisch in einen 256 Byte großen Ringpuffer eingeschrieben. Ein Zähler läuft von 0 bis 255 und der Zählerstand wird jeweils zur Anfangsadresse des Puffers addiert, um die nächste Speicherstelle zu erhalten. Nach der Adresse DRUCK_PUFFER+255 beginnt das Einschreiben wieder von vorn und die alten Werte werden überschrieben.

Damit man nun auch nachschauen kann, wie das Primzahlen-Programm im Hintergrund arbeitet, kann im Tester der aktuelle Inhalt des Druckpuffers mit dem Dump-Kommando angesehen werden.

Folgende Ergänzungen bzw. Änderungen müssen Sie im Programm 3 vornehmen:

Unter den Definitionen von True und False ist einzufügen:

```
DRUCKER  EQU      FALSE   (oder TRUE, wenn Druckerausgabe
                           möglich ist)
```

Im Unterprogramm AUSGABE muß auf Seite F 69 nach EINE_STELLE und SCHREIBE jeweils der Aufruf der Systemfunktion

```
LD C,DRUCKF
CALL SYSTEM
```

ersetzt werden durch den Aufruf des Unterprogramms DRUCKE_ZEICHEN:

```
CALL DRUCKE_ZEICHEN
```

Der gleiche Austausch muß auf Seite F 63 im Hauptprogramm PRIMZAHLEN, Label SCHLEIFE (7., 6., 4., 3. Zeile von unten) vorgenommen werden.

Das Unterprogramm DRUCKE_ZEICHEN wird nach dem Unterprogramm AUSGABE vor dem Datenbereich eingefügt und sieht so aus:

```
DRUCKE_ZEICHEN:
        IF      DRUCKER
        LD      C,DRUCKF        ;ausgeben
        CALL    SYSTEM          ;mit Systemfunktion
        ELSE                    ; oder
        LD      A,(ZEIGER)      ;Zeiger holen,
        INC     A               ;erhöhen
        LD      (ZEIGER),A      ;und speichern,
        LD      L,A             ;dann Zeiger nach HL
        LD      H,0
        LD      BC,DRUCK_PUFFER ;und zu Pufferadresse
        ADD     HL,BC           ;addieren,
        LD      (HL),E          ;ASCII Zeichen in Puffer
        ENDIF                   ;ablegen

        RET
```

Im Datenbereich müssen noch (auf Seite F 70) zwischen

```
REST:
        DS BIN_BYTEund
STACK:
        DS 100
```

die Speicherbereiche für Zeiger und Druckpuffer angegeben werden:

```
        IF      NOT DRUCKER
ZEIGER: DS      1               ;Zeiger und
DRUCK_PUFFER:                   ;Puffer für Ausgabe falls
        DS      256             ;kein Drucker
        ENDIF
```

Nach der Assemblierung ergibt sich:

```
SUM:C715
CRC:18D7
```

Starten Sie das Programm mit RUN und beschäftigen Sie den Rechner mit einer Vordergrundaufgabe, z.B. Briefschreiben im Editor. Zwischendurch kommt dann immer mal wieder ein Blick auf die Ergebnisse der Hintergrundarbeit:

```
Test
*D DRUCK_PUFFER
```

Dabei erscheint die gewohnte Weise der Inhalt eines Speicherbereichs in sedezimaler Darstellung und am rech-

ten Rand die entspechenden ASCII-Zeichen, in diesem Fall die bis dahin gefundenen und gerade im Pufferspeicher abgelegten Primzahlen.

Mit CTRL-C können Sie den Tester wieder verlassen und zur Vordergrundbeschäftigung zurückkehren. Wenn Sie später wieder einmal nachschauen, sehen Sie, wie weit das Primzahlenprogramm inzwischen gekommen ist.

Zusammenfassung

Als Beispiel für Multi-Tasking läuft das Primzahlen-Programm im Hintergrund jeweils in den Pausen zwischen den Tastatureingaben. Je nach der Plazierung des Rücksprungs zum System innerhalb des Primzahlen-Programms kann dessen Geschwindigkeit beeinflußt werden und umgekehrt die Vordergrundarbeit entsprechend schneller oder langsamer vonstatten gehen. Falls kein Drucker angeschlossen ist, kann durch eine Programmänderung erreicht werden, daß die gefundenen Primzahlen in einen Pufferspeicher eingeschrieben werden, wo sie zwischendurch mit einem Dump-Kommando im Tester auf den Bildschirm gebracht werden.

Eine Anzahl bisher noch nicht benutzter Assembler-Anweisungen erlaubt einen profimäßigen Umgang mit dem System, z.B. als Entwicklungssystem für Programme, die auf Mini-Konfigurationen laufen sollen.

Fragen:

1. Was muß bei der Hin- und Herschaltung zwischen dem Primzahlenprogramm und den Systemprogrammen unbedingt beachtet werden?

2. Warum ist es nicht möglich, beim Multi-Tasking-Betrieb im Editor sich einen beliebig großen Textspeicherbereich mit dem PACK-Kommando reservieren zu lassen?

3. Sicher haben Sie es schon oft als lästig empfunden, wenn ein fertiges und abgespeichertes Programm jedesmal assembliert werden muß, bevor es laufen kann. Welche Möglichkeit gibt es, ein assembliertes Programm (also in Maschinencode) abzuspeichern und wieder aufzurufen? Denken Sie daran, was auf Seite D 17 zur Modemübertragung solcher Programme gesagt wurde!

(Die Antworten zu diesen Fragen finden Sie auf der Seite G 17.)

Programm Buchstabenwandlung

Der Umgang mit dem Disketten-Betriebssystem auf der Kommandoebene ist Ihnen inzwischen schon vertraut geworden. Sie können so das Programm 1 (ab Seite F 71) eingeben und auf Diskette abspeichern. Lassen Sie sich einen sinnvollen Namen nach den CP/M-Konventionen dafür einfallen, etwa WANDLUNG.ASM oder ähnlich. Geben Sie diesmal den gesamten Text ein, auch die Kommentare (das Programm ist ja nicht so lang).

Geben Sie ruhig den gesamten Text in Kleinbuchstaben ein; dann läßt sich das Programm gleich an sich selber ausprobieren. Wenn es nämlich nach der fehlerfreien Eingabe assembliert und mit RUN gestartet wird, wandelt es den noch im Textspeicher stehenden Programmtext in Großbuchstaben um. Dabei bleiben aber die in Anführungszeichen stehenden Strings und die mit Semikolon beginnenden Kommentare unverändert. Anders gesagt: der Text sieht so aus wie im Listing ab Seite F 71, obwohl er von Ihnen der Einfachheit halber in Kleinbuchstaben eingegeben wurde.

Speichern Sie sich unbedingt den eingetippten Programmtext (mit Kleinbuchstaben) noch einmal als gesonderte Datei (z.B. unter KLEIN.ASM) ab. Sie werden diese Datei noch einige Male als Versuchsobjekt benötigen.

Im Hauptprogramm wird zunächst ein Unterprogramm VORBEREITUNG aufgerufen, in dem die Zeiger IX und IY auf den Anfang des Textspeichers gesetzt werden. Das folgende Unterprogramm HOLE_ZEICHEN holt ein Zeichen nach dem anderen aus dem Textspeicher und wandelt es unter Umständen in Großbuchstaben. Mit dem Unterprogramm SCHREIBE_ZEICHEN werden die Zeichen wieder in den Textspeicher zurückgebracht.

Nachdem der ganze Inhalt des Textspeichers bearbeitet ist, wird das Unterprogramm ABSCHLUSS aufgerufen. In ihm geschieht im vorliegenden Programm noch garnichts (in den folgenden Programmen werden Sie das gleiche Hauptprogramm jedoch mit anderen Unterprogrammen vorfinden).

Mit einiger Programmier-Erfahrung läßt sich das Hauptprogramm vielleicht auf einen Blick verstehen. Die davor stehenden Definitionen von Sytemadressen, Systemfunktionen und ASCII-Konstanten sind Ihnen inzwischen vertraut, in diesem Programm kommt noch die Definition der Flags hinzu.

Falls Ihre Erfahrung noch nicht ganz ausreicht, um auf Anhieb durchzusteigen, können Sie dieses Programm dazu benützen, um Ihre Erfahrungen zu erweitern. Überlegen Sie sich, wie Sie die gestellte Aufgabe gelöst hätten. Dazu gehört die einfache Methode, die wir Ihnen in den vier Heften des Assembler-Lehrgangs immer wieder empfohlen und vorgeführt haben :

C
112

Aufgabe formulieren,

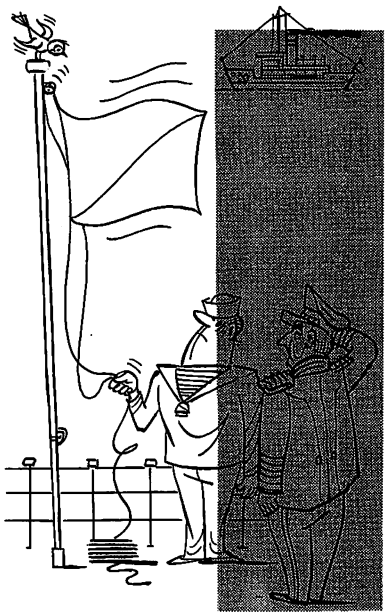
Struktogramm entwerfen,

Struktogramm in Assemblersprache übersetzen!

Erst wenn Sie sich selbst einige Gedanken gemacht haben, sollten Sie unsere ausführliche Aufgabenformulierung ansehen und mit Ihren Ideen vergleichen.

Zunächst kann man die Formulierung mehr oder weniger dem obigen Text entnehmen: Zeichen für Zeichen holen, in Großbuchstaben wandeln außer es handelt sich um einen Kommentar oder String, und wieder ablegen bis zum Ende des Programms.

Der Haken bei dieser Formulierung ist ganz offensichtlich der Nebensatz "außer es handelt sich um einen Kommentar oder um einen String". Dieser Tatbestand läßt sich dem Mikroprozessor nicht so ohne weiteres beibringen. Er müsste ja schon wissen, wann es sich um einen Kommentar oder String handelt. Wir machen es uns leicht und definieren einfach für jeden der beiden Fälle (Kommentar oder String) je einen Merker. Das kann ein ganzes Register oder auch ein einzelnes Bit sein, in denen die Information über "Kommentar oder String" enthalten ist.



Jetzt sieht die Formulierung schon besser aus, der Nebensatz heißt nun " außer der Merker (das Merk-Bit) für "Kommentar" oder der Merker für "String" sind gesetzt. Das hört sich schon fast wie Maschinensprache an, denn einzelne Bits eines Registers lassen sich beim Z80-Prozessor mit den Befehlen SET und RES setzen und löschen und mit dem Befehl BIT abfragen. Übrigens, um beim gebräuchlichen englischen Umgangston zu bleiben: dort nennt man solche Merk-Bits "Flag" - sozusagen eine gehobene oder gesenkte Flagge als Zeichen.

Bei der Formulierung fehlt jetzt noch die Angabe, wann diese beiden Flags gesetzt werden sollen: "Setze das Kommentar-Flag, wenn das gelesene Zeichen ein Semikolon (;) ist" und "Setze das String-Flag, wenn das gelesene Zeichen ein Apostroph (') ist".

Schließlich muß noch gesagt werden, wann die Flags gelöscht werden sollen, falls sie gesetzt sind: "Lösche das Kommentar-Flag am Zeilenende (gelesenes Zeichen Wagenrücklauf =CR)" und "Lösche das String-Flag am Zeilenende oder wenn das gelesene Zeichen ein Apostroph (') ist". Der Übersicht halber noch einmal die gesamte Formulierung:

Zeichen für Zeichen holen. Kommentar-Flag bzw. String-Flag setzen, wenn das Zeichen ein Semikolon bzw. Apostroph ist. Kommentar-Flag am Zeilende löschen, String-Flag am Zeilenende oder bei Apostroph löschen. Die eingelesenen Zeichen in Großbuchstaben wandeln und wieder in den Textspeicher ablegen bis zum Ende des Textes.

Diese Formulierungen als Ausgangspunkte müssen eigentlich genügen, um ein Struktogramm zu erstellen. Nehmen Sie dazu aber ein größeres Blatt, denn die vielen "wenn" in den Formulierungen ergeben ein etwas größeres Struktogramm.

Wenn Sie versuchen, dasjenige Struktogramm zu finden, das exakt zum Programm 1 (ab Seite F 71) paßt, dann müssen wir Ihnen noch einen Tip geben:

Lassen Sie das Programm, wenn es ein Semikolon bzw. einen Apostroph gefunden hat (wenn also ein Kommentar oder ein String beginnt) nicht nach WANDLE_NICHT springen, sondern nach WANDLE. Dann entspricht das Programm ganz genau dem Struktogramm, während es mit dem zunächst unmotiviert erscheinenden Sprung nach WANDLE_NICHT einige Mikrosekunden schneller abläuft.

Vielleicht ist Ihnen eine Besonderheit des ZEAT-Assemblers aufgefallen: in den Bit-Manipulationsbefehlen (BIT, SET und RES) des Z80-Prozessors kann die Bit-Nummer auch als symbolischer Ausdruck angegeben werden. Das gilt übrigens ebenso für die Restart-Befehle (RST).

Wie gesagt, Sie sollten das Programm gründlich durcharbeiten, weil es Ihnen in anderen Variationen wieder begegnen wird. Da Sie inzwischen über genügend Erfahrung verfügen, überlassen wir es Ihnen, ein solches Programm selbstständig zu analysieren.

Frage:

1) Wie kann mit dem vorliegenden Programm ein beliebiges Programm-Listing in Großbuchstaben gewandelt werden?

(Die Antwort zu dieser Frage finden Sie auf Seite G 20, überlegen Sie aber erst einmal, ehe Sie dort nachschauen, und außerdem, lesen Sie nicht weiter, denn mit dem nachfolgenden Text wird die Antwort leicht gemacht.)

Sie werden uns sicher zustimmen: es ist nicht besonders komfortabel, wenn man jedes Programm-Listing, das umgewandelt werden soll, erst in den Textspeicher laden muß. Danach wird das Umsetzungsprogramm gestartet und schließlich das gewandelte Programm-Listing aus dem Textspeicher wieder auf die Diskette abgelegt.

Viel einfacher wäre es, wenn das Umsetzungsprogramm sich das Programm-Listing selber direkt von der Diskette holen und nach der Umwandlung wieder dorthin zurückbringen könnte. Genau dies werden die beiden nächsten Programm-Beispiele mit Hilfe der ensprechenden Systemfunktionen auch tun. Davor ist es aber notwendig, sich mit eben diesen CP/M-Systemfunktionen ausführlich zu befassen.

Zusammenfassung

Das erste Programm-Beispiel zeigt, wie innerhalb eines Programm-Listings alle Buchstaben, soweit es keine Kommentare oder Strings sind, in Großbuchstaben umgewandelt werden. Dadurch gewinnt das Listing an Übersichtlichkeit und brauchte doch bloß in fortlaufendem Text ohne fleißigen Gebrauch der Shift-Taste eingetippt zu werden. Der strukturierte Aufbau des Programms ist anhand der Erfahrungen mit den Programm-Beispielen der vorangegangenen Hefte leicht zu durchschauen.

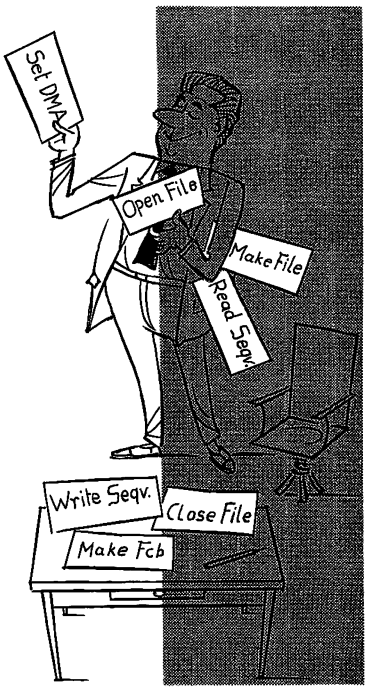
Programm Buchstabenwandlung, 2.Version

Der vorangegangene doch recht lange Exkurs zu den CP/M-Systemfunktionen war unumgänglich, damit Sie sich jetzt mit der zweiten Version des Programms zur Buchstabenwandlung (Seite F 74) befassen.

Der Gebrauch jener Systemfunktionen macht eben den Unterschied zur ersten Version des Programms aus: es wird zwar im Prinzip das gleiche gemacht, nur holt sich das Programm jetzt die Programmdatei, in der es die Kleinbuchstaben in Großbuchstaben wandeln soll, selbst von der Diskette in den Arbeitsspeicher und legt sie nach getaner Arbeit wieder ab.

Der erste Unterschied zum Programm 1 zeigt sich dementsprechend bei der Definition der Systemfunktionen. Die Ihnen bekannten Diskettenfunktionen (sofern benötigt) werden gleich am Anfang des Programms definiert:

```
WSTARTF EQU     0       ;Warmstart
DIRCONF EQU     6       ;Direkte Konsol Ein/Ausgabe
STRAUSF EQU     9       ;Ausgabe von Strings auf CON:
STREINF EQU     10      ;Eingabe von Strings mit CON:
OPENF   EQU     15      ;Datei oeffnen
CLOSEF  EQU     16      ;Datei schliessen
DELETEF EQU     19      ;Datei löschen
READF   EQU     20      ;aus Datei lesen
WRITEF  EQU     21      ;in Datei schreiben
MAKEF   EQU     22      ;Datei anlegen
SETDMAF EQU     26      ;DMA-Adresse setzen
MAKEFCB EQU     250     ;FCB aus String erzeugen
```



Zunächst unbekannt ist für Sie die Funktion zum Löschen einer Datei (Funktion 19), die Sie zusammen mit der Funktion zum Umbenennen einer Datei noch genauer Kennenlernen werden. Vorläufig genügt es, wenn Sie wissen, daß die Funktion durch die Übergabe eines FCB genauso bedient wird wie die anderen Diskettenfunktionen.

Über den Ablauf des Hauptprogramms brauchen Sie sich keine Gedanken zu machen, es ist identisch mit dem im ersten Programm. Neu hingegen sind die Unterprogramme VORBEREITUNG und ABSCHLUSS. Das erste hat die Aufgabe, einen Programmtext von der Diskette in den freien Bereich der TPA zu laden; das zweite muß den gleichen, nach der Bearbeitung leicht geänderten Text wieder auf die Diskette ablegen.

Das Unterprogramm VORBEREITUNG fordert mit der Systemfunktion 9 (STRAUSF) zur Eingabe des Namens der Datei auf, die gewandelt werden soll. Anschließend wird dieser Namen mit der Systemfunktion 10 (STREINF) in einen kleinen Textpuffer eingelesen. Mit der Systemfunktion 250 (MAKEFCB) wird aus dem Namen im Textpuffer ein FCB erzeugt. Danach wird mit der Systemfunktion 15 (OPENF) versucht, diese Datei zu öffnen.

Falls sie nicht vorhanden ist, wird das Programm nach Ausgabe der entsprechenden Fehlermeldung abgebrochen. Im anderen Fall wird begonnen, die Datei Datensatz für Datensatz in den Speicher zu übertragen. Die erste Transferadresse ist die Speicheradresse gleich nach dem Ende des Programms (PROGRAMM_ENDE).

Die Transferadresse muß dabei, wie Sie wissen, vor jedem Aufruf der Lesefunktion um 128 erhöht (erster Befehl von LESE_SCHLEIFE1 auf Seite F 78) und dahingehend überprüft werden, ob an der neuen Transferadresse noch genügend freier Speicherplatz zum Einlesen eines Datensatzes von 128 Bytes vorhanden ist. Da die an den Adressen 6 und 7 stehende Sprungadresse auf die erste vom Betriebssystem belegte Adresse zeigt, erfolgt diese Prüfung einfach dadurch, daß die Transferadresse mit der um 256 reduzierten (DEC H) bei System+1 stehenden Adresse verglichen wird (LESE_SCHLEIFE).

Im Fehlerfall wird der bei VORB_FEHLER2 stehende Text "Speicher voll" mit der Stringausgabe-Funktion ausgegeben und mit dem Carry-Flag =1 ins Hauptprogramm zurückgekehrt.

Im Normalfall wird festgestellt, daß genügend Platz vorhanden ist für den einzulesenden Datensatz und den von dieser höchsten freien Adresse "nach unten" wachsenden Stack - der Stackpointer wurde zu Beginn des Hauptprogramms ebenfalls auf diese Adrese gesetzt (START1 , Seite F 75).

Um dem Unterprogramm ABSCHLUSS die Arbeit etwas zu erleichtern, wird beim Einlesen gleich mitgezählt, wieviele Datensätze die Datei enthält, denn soviele Datensätze müssen ja in diesem Unterprogramm dann auch wieder abgespeichert werden. Der Zähler wird bei VORBEREITUNG1 im BC-Registerpaar gesetzt, nach LESE-SCHLEIFE1 inkrementiert und gerettet, später restauriert und beim Label SATZ_ZAH Labgelegt.

Das Unterprogramm ABSCHLUSS baut sich zunächst einen FCB für die Datei zum Abspeichern des gewandelten Programmtextes auf. Danach wird geprüft, ob diese Datei schon existiert, und zwar wird versucht, die Datei zu öffnen

(LD C,OPENF, Seite F 79). Während beim Einlesen in den Speicher die gewählte Datei existieren mußte, soll sie hier nicht vorhanden sein.

Wird von der Funktion zum Öffnen einer Datei doch schon ein entsprechender Eintrag im Inhaltsverzeichnis gefunden, dann ist dies ein Fehler. Mit LD DE,ABSCH_FEHLER1 wird der am Anfang des Unterprogramms definierte Text der Fehlermeldung ("Datei existiert") ausgegeben und das Programm abgebrochen.

Gibt es den gewählten Dateinamen noch nicht, dann wird eine entsprechende Datei angelegt und die zuvor von der Eingangsdatei eingelesene Anzahl Datensätze (LD BC,(SATZ_ZAHL) nach ABSCHLUSS2) in diese Datei geschrieben.

Dabei muß nicht wie beim Einlesen in den Speicher darauf geachtet werden, ob dort noch genügend Platz ist. Vielmehr ist jetzt wichtig, ob die Diskette nicht unter Umständen voll ist. Dies wird von der Funktion zum Schreiben eines Datensatzes gemeldet. Für den Fehlerfall ist vorgesehen, daß die neu angelegte Datei gleich gelöscht wird (LD C,DELETEF, Seite F 80), da ein halber Programmtext ja nichts nützt. Außerdem wird der Fehlertext bei ABSCH_FEHLER3 ("Diskette voll") ausgegeben und das Programm abgebrochen.

Im Normalfall werden nach SCHREIB_SCHLEIFE die jeweiligen Transferadressen für den nächsten Durchlauf ermittelt, der Zähler dekrementiert usw. bis alle Datensätze auf der Diskette abgelegt sind. Schließlich wird bei SCHREIB_ENDE mit der entsprechenden Funktion die Datei geschlossen und zum Hauptprogramm zurückgesprungen. Damit hat das Programm seine Aufgabe erledigt.

Es kann vorkommen, daß die TPA Ihres Systems zum Einlesen eines längeren Programms zu klein ist. In diesem Fall kann die TPA unter Umständen mit dem PACK-Kommando etwas vergrößert werden.

Sie können aber auch die (nur im ZEAT vorhandene erweiterte) Systemfunktion MAKEFCB durch ein eigenes Unterprogramm ersetzen. Dann kann das Programm auch ohne ZEAT in einem reinen CP/M-System laufen. Da es in diesem Fall keinen Textspeicher mehr benötigt, wird die TPA noch wesentlich größer.

Sie können aber einen noch einfacheren Weg gehen: nämlich mit dem Umsetzen von längeren Programmen warten, bis wir Ihnen auch noch eine dritte Variante des Umwandlungsprogramms vorführen - diese wird dann mit beliebig langen Programmen fertig.

Zusammenfassung

Das Hauptprogramm ist das gleiche wie bei der ersten Version des Programms zur Buchstabenwandlung. In der zweiten Version wird mit den Systemfunktionen für den Diskettenbetrieb gearbeitet, das Programm holt sich die zu wandelnde Datei selber von der Diskette und bringt sie dort nach der Wandlung wieder hin. Bei sehr langen Programmdateien ist es möglich, das die TPA zu klein wird. Wenn die Vergrößerung mit PACK nicht ausreicht, kann in einem reinen CP/M-System gearbeitet werden, dort wird durch den Wegfall des Textspeichers die TPA noch größer. Allerdings muß die ZEAT-spezifische Systemfunktion "MAKEFCB" durch ein entsprechendes Unterprogramm ersetzt werden.

Frage:

1) Beim reinen CP/M-Betrieb muß "MAKEFCB" durch ein entsprechendes Unterprogramm ersetzt werden. Wie sieht dieses aus?

(Die Antwort zu dieser Frage finden Sie auf Seite G 21.)

Überlegungen zum Programm Buchstabenwandlung

Das neu gewonnene Wissen zum Thema Parameter-Übergabe läßt sich gut zur Verbesserung der zweiten Version des Programms Buchstabenwandlung einsetzen. Und zwar geht es darum, das Unterprogramm VORBEREITUNG so abzuändern, daß es nicht mehr nach Dateinamen fragt, sondern einfach den bei der Adresse 005CH stehenden und vom Betriebssystem zur Verfügung gestellten Dateinamen in den vorgesehenen FCB kopiert.

Die Adresse 005CH könnte aber auch gleich als FCB verwendet werden. Dann muß aber der zweite Dateiname vorher an eine andere Stelle kopiert werden. Die Funktionsaufrufe für Textausgabe, Texteingabe und "FCB erzeugen" müssen demnach durch ein kleines Schiebeprogramm ersetzt werden:

```
STANDARD_FCB1   EQU     005CH

                LD      HL, STANDARD_FCB1
                LD      DE, EINGABE_FCB
                LD      BC, 12          ;12 Byte schieben
                LDIR
```

Die entsprechende Änderung für das Unterprogramm ABSCHLUSS sieht fast gleich aus:

```
STANDARD_FCB2   EQU     006CH

                LD      HL, STANDARD_FCB2
                LD      DE, AUSGABE_FCB
                LD      BC, 12          ;12 Byte schieben
                LDIR
```

Das so geänderte Programm kann nach dem Assemblieren nicht mehr mit RUN gestartet werden, genauer gesagt: starten können Sie es schon, das Betriebssystem bereitet die Adressen 005CH, 006CH und 0080H aber nur beim Aufruf externer Programme auf.

Richtig funktioniert es so: nach dem Assemblieren muß der Tester aufgerufen und das Programm als .COM-Datei auf der Diskette abgelegt werden:

```
A>test
*SF start,programm_ende
FILENAME? gross.com
```

Dann kann das Programm als externes Kommando gestartet werden:

```
A>gross versuch.1 versuch.2
Klein-Grossbuchstabenwandlung fuer Programmtext
.
.
.
```

C

120

Führen Sie diese Änderungen an der Version 2 selbst durch, es wird Ihnen sicher gelingen. Das so geänderte Programm läuft übrigens auch in jedem CP/M-System ohne ZEAT, weil es ja keine erweiterten Systemfunktionen mehr verwendet.

An unserem Programm sollten aber noch zwei weitere Änderungen durchgeführt werden. Zum einen soll die Bedienung recht einfach werden, es soll beim Aufruf lediglich der Name der zu wandelnden Datei angegeben werden. Sie soll gewandelt und unter ihrem gleichen Namen wieder abgespeichert werden.

Dafür müßte eigentlich die Eingangsdatei zuerst gelöscht werden. Das wollen wir aber nicht tun. Statt dessen wird die Eingangsdatei umbenannt in den Typ ".BAK". Damit ist sie nicht verloren, allerdings muß eine eventuell vorhandene .BAK-Datei mit dem gleichen Namen gelöscht werden.

Der Dateityp ".BAK" ist eine Sicherungsdatei, der auch bei Textverarbeitungs-Systemen oft verwendet wird: eine solche .BAK-Datei wird automatisch angelegt, um z.B. nach einer Änderung mit darauffolgendem Abspeichern auch den ursprünglichen Zustand noch parat zu haben.

Die zweite Änderung betrifft den benötigten Speicherplatz. Es soll nicht mehr die ganze Datei auf einmal in den Speicher eingelesen werden. Vielmehr geschieht das Datensatz für Datensatz, so wie diese für die Wandlung benötigt werden. Und genauso werden die gewandelten Datensätze auch wieder auf die Diskette abgespeichert.

Damit lassen sich mit dem Programm mit einem viel geringeren Bedarf an Speicherplatz beliebig lange Dateien umwandeln. Dieses Verfahren ist an sich unproblematisch, da das Betriebssystem mit mehreren FCB auch mehrere Dateien gleichzeitig bearbeiten kann.

Damit das Programm nun ganz vorbildlich ist, soll es die ursprünglich Datei erst umbenennen und die möglicherweise vorhandene .BAK-Datei löschen, wenn feststeht, daß die neue Datei auch tatsächlich Platz auf der Diskette hat. Sonst könnte es nämlich passieren, daß die alte .BAK-Da-

tei gelöscht wird, obwohl es gar keine neue .BAK-Datei gibt, und das sollte auf keinen Fall geschehen!

Das Verfahren, bei dem die neue, zunächst vorläufige Datei den Typ ".$$$" erhält, sieht dann am Beispiel der Datei "PROG.TLC" so aus:

PROG.TLC	Öffnen zum Einlesen
PROG.$$$	Ev. vorhandene vorläufige Datei löschen
PROG.$$$	Vorläufige Datei anlegen
PROG.TLC	Lesen und
PROG.$$$	Beschreiben

Nach erfolgreichem Programmdurchlauf (Diskette nicht voll) können die Dateien umbenannt und die alte ".BAK" Datei gelöscht werden. Bei einer vollen Diskette müßte lediglich die halbfertige Datei "PROG.$$$" gelöscht werden:

PROG.BAK	Löschen
PROG.TLC	umbenennen in
PROG.BAK	
PROG.$$$	umbenennen in
PROG.TLC	

Dieses etwas aufwendige Verfahren stellt sicher, daß keine Daten unnötig verloren gehen. Es wird übrigens auch vom ZEAT-Editor bzw. Tester verwendet, wenn eine Datei auf Diskette geschrieben wird. Die von uns verwendeten Dateitypen entsprechen einigen bei CP/M-Systemen üblichen Festlegungen:

$$$	vorläufige Datei
BAK	Sicherungsdatei
ASM	Assemblerprogramm
HEX	HEX-Listing
PRN	Drucker-Listing
COM	Kommando, lauffähiges Programm
BAS	BASIC-Programm
PAS	PASCAL-Programm
TXT	Textdatei

Selbstverständlich gibt es neben diesen oft benutzten Bezeichnungen von Dateitypen noch andere. Außerdem bleibt es jedem Benutzer überlassen, seine eigenen "Kreationen" zu verwenden, solange sie nur den für CP/M gültigen Festlegungen entsprechen.

Zusammenfassung

Die zweite Version des Programms zur Buchstabenwandlung wird so geändert, daß es nicht mehr nach einem Dateinamen fragt, sondern als .COM-Datei auf der Diskette steht. Bei seinem Aufruf wird dann der Name der zu wandelnden Datei in der Kommandozeile mit angegeben.

Eine weitere Änderung bewirkt, daß nicht die komplette zu wandelnde Datei in den Speicher geladen wird, die Verarbeitung geschieht vielmehr Datensatz für Datensatz. Dadurch ist es möglich, beliebig lange Dateien ohne Rücksicht auf den verfügbaren Speicherbedarf zu bearbeiten.

Frage:

1) Was muß man beachten, wenn man nach den für CP/M gültigen Festlegungen einen Dateinamen auswählt?

(Die Antwort zu dieser Frage finden Sie auf Seite G 22.)

Programm Buchstabenwandlung, 3.Version

In dieser Version wird, entsprechend den Vorüberlegungen auf den Seiten C 119 bis C 122, die zu bearbeitende Datei fortlaufend eingelesen und wieder abgespeichert. Daher muß das Unterprogramm VORBEREITUNG keine Daten mehr einlesen, sondern nur noch die notwendigen FCB vorbereiten und die Dateien jeweils öffnen oder anlegen.

Das Vorbereiten der FCB ist sehr einfach, wenn die Hilfestellung des Betriebssystems richtig ausgenützt wird. Der an der Adresse 005CH vorbereitete FCB muß an die entsprechenden Stellen im Programm kopiert werden und schon ist diese Sache erledigt. Bei der Ausgabedatei, die den Typ ":$$$" und der Sicherungsdatei (Backup), die den Typ ".BAK" erhalten soll, darf der Dateityp natürlich nicht mitkopiert werden.

Vor dem Anlegen der Ausgabedatei auf der Diskette muß die vorläufige Datei sicherheitshalber gelöscht werden. Das kann ohne weiteres geschehen, da eine vorläufige Datei vom Typ ".$$$" in jedem Fall gelöscht werden darf. Die Auswertung des Fehlercodes, welchen das Betriebssystem bei der Löschfunktion zurückgibt, kann unterbleiben. Wurde die Datei nämlich erfolgreich gelöscht, dann ist die Sache in Ordnung; trat dagegen ein Fehler auf, dann heißt das, daß die Datei nicht gefunden wurde. Das aber ist auch in Ordnung, in beiden Fällen ist die störende Datei nicht vorhanden.

Wenn die Ausgabedatei aber nicht angelegt werden kann, weil das Inhaltsverzeichnis voll ist, dann muß dieser Fehler mit einem gesetzten Carry-Bit an das Hauptprogramm zurückgemeldet werden.

Beim Unterprogramm ABSCHLUSS gibt es analog zum Unterprogramm VORBEREITUNG ähnliche Änderungen. Hier müssen jetzt keine Daten mehr abgespeichert werden, außer dem letzten noch nicht auf Diskette geschriebenen Inhalt des Ausgabepuffers.

Die Aufgabe des Unterprogramms ABSCHLUSS sieht jetzt so aus: nach einem erfolgreichen Programmdurchlauf muss die alte ".BAK"-Datei gelöscht werden, die alte Programmdatei (Eingabedatei) muß in eine ".BAK"-Datei umbenannt werden und schließlich erhält die neue Programmdatei, die noch den Typ ".$$$" hat, den Namen und Typ der ursprünglichen Eingabedatei.

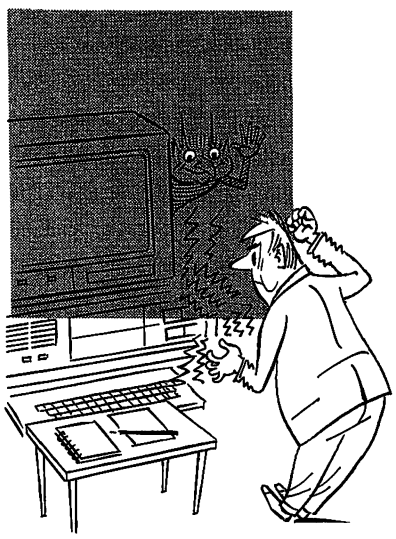
Eine Auswertung der Fehlermeldungen haben wir uns auch in diesem Unterprogramm an einigen Stellen sparen können. Man kann ja davon ausgehen, daß eine vom Unterprogramm VORBEREITUNG ohne Fehler geöffnete oder angelegte Datei auch tatsächlich vorhanden ist. Selbstverständlich könnte es nicht schaden, auch diesen Fehlerfall zu erfassen, damit würden grobe System- oder Programmfehler angezeigt.

Das Unterprogramm HOLE_ZEICHEN holt mit Hilfe eines Zeigers jeweils ein Zeichen aus einem Eingabepuffer. Dieser wird zunächst mit einem Datensatz von der Diskette geladen, wenn der Zeiger auf den Anfang des Puffers zeigt.

Nachdem das letzte Zeichen aus dem Puffer (Nummer 127 oder 7FH) ausgelesen ist, wird der Zeiger mit einem einfachen UND-Verknüpfungsbefehl wieder auf den Anfang zurückgesetzt: eigentlich würde er auf 128 gleich 80H vorrücken, nach der UND-Verknüpfung mit 7FH wird daraus aber wieder Null.

Das Unterprogramm SCHREIBE_ZEICHEN funktioniert im Prinzip gleich wie HOLE_ZEICHEN, nur werden hier die Zeichen nicht aus dem Puffer heraus gelesen sondern in den Puffer hinein geschrieben. Außerdem wird der Inhalt des Puffers auf die Diskette geschrieben, wenn der Zeiger nach dem Einschreiben eines Zeichens auf den Pufferanfang zeigt.

Von den beiden Unterprogrammen zur zeichenweisen Verarbeitung werden Fehlermeldungen mit einem gesetzten Carry-Bit angezeigt und verursachen dadurch einen Programmabbruch im Hauptprogramm. Es wird lediglich vor dem Warmstart noch die halbfertige ".$$$"-Datei gelöscht. Damit hinterläßt das Programm im Fehlerfall die Diskette wie vor dem Programmstart. Nur nach einem vollständigen und fehlerfreien Ablauf werden die Änderungen durch das Unterprogramm ABSCHLUSS endgültig auf der Diskette durchgeführt.



Vergessen Sie nicht, das Programm nach dem Assemblieren als .COM-Datei auf der Diskette abzulegen (wie auf Seite C 119 beschrieben). Es kann nicht mit RUN gestartet werden.

Wenn Sie das Programm ausprobieren, fällt Ihnen sicher auf, daß der Schreib/Lesekopf des Diskettenlaufwerks schwer beschäftigt ist, außerdem ist das Programm recht langsam. Der Grund für beides sind die winzigen Datenpuffer, die jeweils nur einen Datensatz aufnehmen. Das Betriebssystem muß für jeden einzelnen Datensatz zwischen den beiden beteiligten Dateien hin- und herfahren.

Noch weiter verschlechtert wird die Leistung durch den Umstand, daß auf der Diskette mehr als ein Datensatz in

einem Sektor gespeichert ist. Deswegen muß für jeden Datensatz immer ein viel größerer Sektor eingelesen werden. Dieser kann aber nicht weiter benutzt werden, weil in der Zwischenzeit wieder auf die andere Datei zugegriffen werden muß und das System nur einen internen Puffer hat.

Man kann die Leistung dieses Programms um ein Vielfaches steigern, wenn es auf eine Puffergröße von z.B. 1024 Bytes (das sind 8 Datensätze) umgebaut wird. Das Füllen und Leeren der Puffer muß dann mit ähnlichen Schleifen geschehen, wie sie im Programm 2 eingesetzt sind. Die Zeiger in den Puffern müssen in diesem Fall mit 16-Bit-Zählern realisiert werden.

Versuchen Sie es, das ist eine anspruchsvolle und zudem nützliche Aufgabe. Ein Programm mit der vorliegenden Funktion ist nämlich universell einsetzbar. Man nennt das ein Filter: es kopiert eine Datei in eine andere und ändert sie dabei ein wenig. Solch ein Filter läßt sich leicht für viele Aufgaben anpassen: z.B. kann man in einem Text alle Umlaute durch die entsprechenden Kombinationen aus 2 Buchstaben ersetzen.

Zusammenfassung

In der dritten Version des Programms zur Buchstabenumwandlung erhalten die Unterprogramme VORBEREITUNG und ABSCHLUSS andere Aufgaben, die beiden Unterprogramme zur zeichenweisen Verarbeitung (HOLE_ZEICHEN und SCHREIBE_ZEICHEN) werden modifiziert und das Hauptprogramm bleibt praktisch unverändert.

Das Programm entspricht in dieser Form einem "Filter" genannten Programmtyp: eine Datei wird duch dieses Filter geschickt und entsprechend verändert auf die Diskette abgelegt. Mit einem solchen Filter lassen sich viele Aufgaben in der Praxis lösen.

Wir hoffen, daß wir Ihnen, ähnlich wie in den vier Heften des Lehrgangs Assembler-Programmierung, auch mit diesem zusätzlichen Heft einige Anregungen für eigene Arbeiten geben konnten. Die verwendeten Programmbeispiele sollten nicht nur dazu dienen, die Möglichkeiten des Z80-Prozessors, des ZEAT-Betriebssystems und der Assembler-Programmierung allgemein vorzuführen. Vielmehr enthalten sie ja auch Unterprogramme, die als Programm-Bausteine bei anderen Programmieraufgaben immer wieder gebraucht werden können.

C
126

Das vorliegende Heft beschäftigte sich mit dem Diskettenbetrieb, wobei das Standard-Betriebssystem CP/M nur soweit behandelt wurde, wie es als DOS zusammen mit ZEAT notwendig ist. Wir können Ihnen nur noch einmal empfehlen, sich mit den übrigen Möglichkeiten von CP/M vertraut zu machen. Dazu genügt nach dem in unserem Lehrgang erworbenen Wissen ein Handbuch.