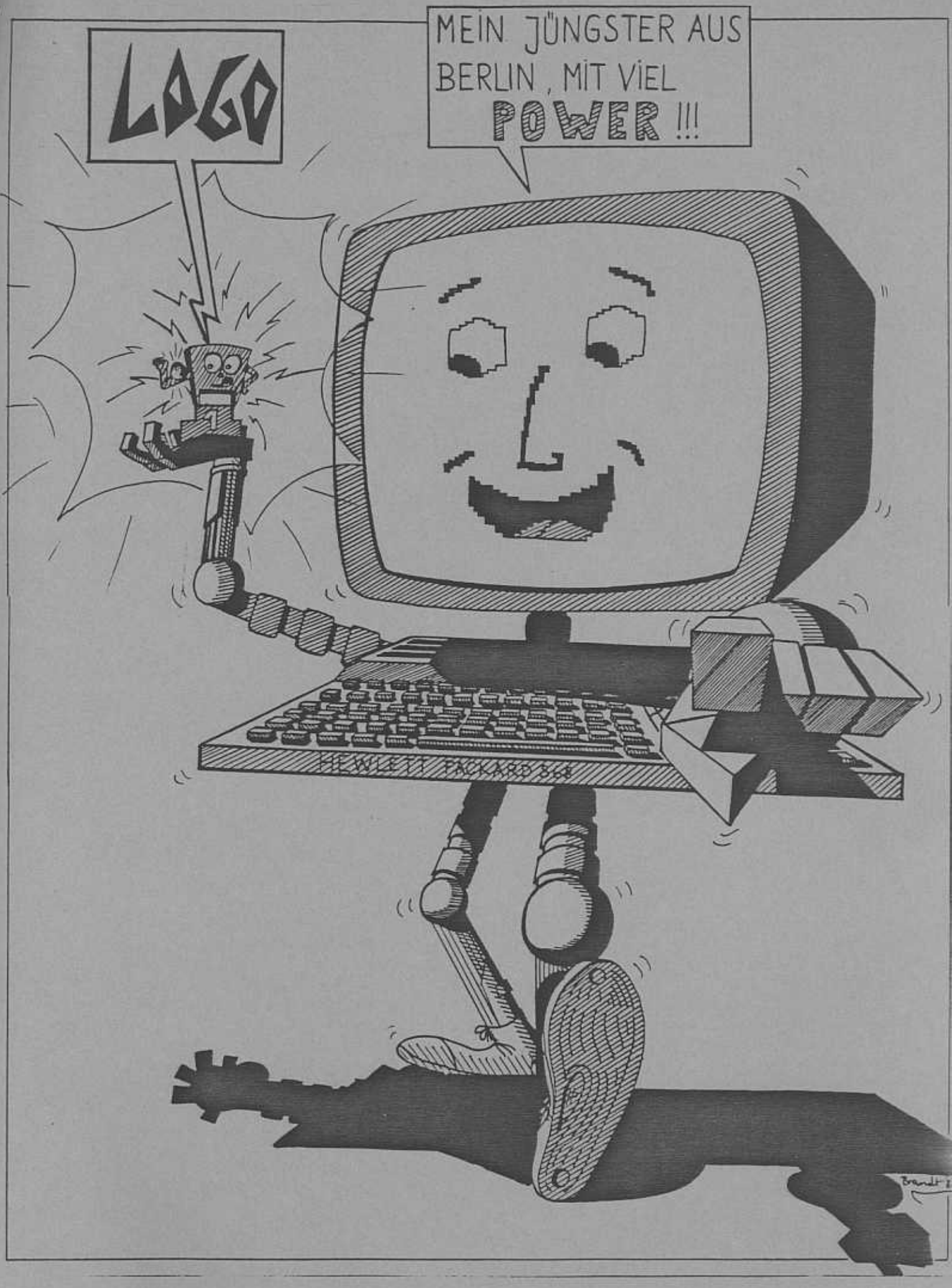
HP86/87

Diskettenorganisation
Advanced Programming

Bit/Stringsmanipulationen
Maschinensprachesteuerung

SYSEXT

Betriebssystemerweiterung


LOGO
MEIN JÜNGSTER AUS BERLIN, MIT VIEL POWER !!!

André Koppel Software

Handbuch Nummer

inklusive

Diskette | ROM | EPROM | Softwarestecker

Seriennummer

Es wird dem Käufer gestattet, von den gelieferten Produkten jeweils eine Sicherheitskopie für Archivzwecke anzufertigen, sofern dies möglich ist. Defekte Disketten werden beim Einsenden der Diskette zum Selbstkostenpreis ersetzt. Für den abteilungsinternen Gebrauch dürfen sofern möglich bis zu drei Kopien angefertigt werden. Bei höherem Bedarf ist eine schriftliche Genehmigung bei uns einzuholen. Es besteht die Möglichkeit, daß die von uns vertriebene Software in von Ihnen erstellte und vertriebene Software eingebaut wird. Für diesen Fall können Sie von uns speziell modifizierte Software erhalten. In jedem Fall bedarf dieser Vorgang eine vertragliche Vereinbarung mit uns. Die zugehörigen Unterlagen können Sie bei uns kostenfrei anfordern. In jedem Fall hat der Käufer sicherzustellen, daß André Koppel Softwareprodukte ohne unsere schriftliche Zustimmung Dritten nicht zugänglich gemacht werden.

André Koppel Software

Sehr geehrter Anwender,

das SYSEXT-ROM basiert auf einer langen Erfahrung in der Serie-80 Maschinenspracheprogrammierung. Alle neuen Befehle und Funktionen des ROMs sollen Sie in möglichst vielen Bereichen bei der Programmierung unterstützen. Mit seinen über 90 neuen Schlüsselworten setzt das ROM einen Meilenstein in der fortgeschrittenen Programmierung und ich hoffe, daß es Ihnen viel Freude bereitet oder zumindest erheblich Speicherplatz und Programmierzeit sparen hilft.

Mit freundlichen Grüßen

Andri Koppel

Bemerke

Andre Koppel Software übernimmt keine Haftung für Schäden, die durch der Nutzung des ROMs oder Handbuches entstehen

Andre Koppel Software - D-1000 Berlin 65 - Lüderitzstr. 16
Tel.: 030 / 451 62 96

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

1 Wie bekommt man die Befehle in den Computer

Nun, diese Frage läßt sich sehr leicht beantworten. Bei Verwendung des SYSEXT-Binärprogrammes wird einfach LOADBIN "SYSEXT" eingegeben. Haben Sie jedoch die EPROM-Version vom SYSEXT erstanden, so muß das EPROM im HP 28929A-Programmable-ROM-Module installiert werden. Gehen Sie dafür wie folgt vor:

1) Nehmen Sie sich den EPROM-Halter und schrauben Sie ihn auseinander. Der EPROM-Halter wird von sechs Kreuzschrauben zusammengehalten. Legen Sie die Platine so vor sich, daß die Bauteileseite nach oben und die Anschlußleiste von Ihnen weg zeigt.

2) In der rechten unteren Ecke des EPROM-Halters finden Sie eine mit W1 markierte Brücke, den Jumper. Ziehen Sie diese Brücke aus der Platine heraus und stecken Sie sie in die etwas weiter rechts gelegene zweite Position. Hiermit wurde der EPROM-Halter auf 8 kB EPROMs eingestellt.

3) Das EPROM stecken Sie nun in einen der beiden Sockel. Es mag merkwürdig klingen, aber als Sockel werden die beiden Doppelkunstoffnippelreihen bezeichnet. Stecken Sie das EPROM einfach in die Gummis. Keine Sorge, daß funktioniert wirklich, bei den Nippeln handelt es sich um leitenden Elektrogummi. U.U. müssen Sie die Beinchen des EPROMs etwas nach innen biegen, damit das EPROM in die Sockel paßt. Achten Sie darauf, daß das EPROM in der richtigen Richtung in den Sockel gesteckt wird. Wenn Sie den Sockel U18 benutzen, so muß die Einbuchtung am EPROM in Ihre Richtung zeigen. Benutzen Sie den Sockel U19, so muß die Einbuchtung nach rechts zeigen.

4) Stellen Sie nun die zu der EPROM-Fassung gehörenden DIP-Schalter in die richtige Position; DIP-Switch S1 gehört zum Sockel U18. Das EPROM hat die Nummer 56, also ist folgende Einstellung vorzunehmen:

1 = on	oder Position	0
2 = on	" "	0
3 = on	" "	0
4 = off	" "	1
5 = off	" "	1
6 = off	" "	1
7 = on	" "	0
8 = on	" "	0

5) Legen Sie die Platine in das Gehäuse und schrauben Sie dieses wieder zusammen. Achten Sie darauf, daß Sie nicht vergessen, die kleine Metallplatte mit einzusetzen.

6) Stecken Sie den EPROM-Halter in Ihren HP86/87 und schalten Sie diesen an. Wenn der Computer überhaupt nicht aufwacht oder gleich am Anfang eine Fehlermeldung generiert, so haben Sie etwas Grundlegendes falsch gemacht. Schalten Sie den Rechner wieder aus, und überprüfen Sie nochmals alles.

7) So, Sie sind bis zu diesem Punkt gekommen, also haben Sie zumindest nichts grundliegendes falsch gemacht. Nun wollen wir überprüfen, ob der Computer das ROM auch anerkennt. Geben Sie das Schlüsselwort SYSEXT ein. Wenn das Revisionsdatum auf dem Bildschirm erscheint, so ist alles in Ordnung, und Sie können mit

dem nächsten Kapitel weitermachen. Erscheint die Fehlermeldung BAD STATEMENT auf dem Bildschirm, so hat der Computer das ROM nicht gefunden, das bedeutet, daß die DIP-Schalter auf der EPROM-Karte falsch eingestellt sind. Korrigieren Sie den Fehler.

1.1 Vom ROM benötigter Speicherplatz

Das ROM belegt mit seinem Programmkode zwar nicht das RAM, aber es benötigt trotzdem einen gewissen Platz vom RAM als Speicher; daher reserviert das SYSEXT-ROM 376 Bytes für eigene Nutzung, sowie der Computer angeschaltet wird.

2 Arbeiten mit den Befehlen des ROMs

In den folgenden Kapiteln werden alle Befehle nach Anwendungsgruppen alphabetisch sortiert aufgezeigt. Sofern notwendig, wird zu einigen Befehlen der BASIC-Hintergrund erklärt, d.h. es wird erklärt, warum einige Anweisungen die strikte Einhaltung von bestimmten Konventionen voraussetzen.

Ein zu jedem Befehl gehörendes Beispiel soll die Anwendung der neuen Anweisung, u.U. in Verbindung mit anderen Anweisungen erklären.

2.1 Massenspeicherbefehle

DCAT$ (Nummer des Eintrages)

Hierbei handelt es sich um eine Stringfunktion mit einem numerischen Parameter. Als Resultat wird der mit der Zahl angegebene Katalogeintrag in der Form

NNNNNNNNNNTTTTLLLLLAAAAA

zurückgegeben. Dabei ist N der Name des Workfiles, T der Typ (DATA, BPGM, ASSM etc.), L die Länge eines Records und A die Anzahl der Records in diesem File.

Ist die Zahl größer als die Anzahl der im Katalog eingetragenen Files, so gibt die Funktion einen Nullstring (Länge ist null) als Resultat zurück.

DCATNEXT$

Diese Funktion kann nur nach Benutzen der Funktion DCAT$ angewendet werden. DCATNEXT$ gibt den nächsten Katalogeintrag in der oben beschriebenen Stringform zurück. Die Funktion DCATNEXT$ holt sich ihre Daten aus einem, von dem Funktionsaufruf DCAT$ existierenden Buffer. Erst wenn der Buffer abgearbeitet wurde, füllt DCATNEXT$ ihn wieder auf. Daher kommt es, daß beim Aufruf der Funktion DCATNEXT$ nicht immer auf den Massenspeicher zugegriffen wird.

Wenn die Funktion DCATNEXT$ den letzten Katalogeintrag erreicht hat, so werden bei erneutem Funktionsaufruf nur noch Nullstrings zurückgegeben.

Beispiel: Ein Programm soll von einer Autostartroutine beim Einschalten des Computers geladen werden. Folgende Forderungen soll die Autostartroutine erfüllen:

* Sie soll als erstes Programm nach dem Einschalten des Computers geladen werden.

* Sie soll einige Systemeinstellungen vornehmen, Binärprogramme laden, und dann zum Hauptprogramm übergehen.

* Da das Hauptprogramm des öfteren geändert wird, wobei sich die Namen jeweils durch eine höhere Nummer unterscheiden, kann der CHAIN-Befehl nicht einfach mit einer Namenskonstante durchgeführt werden, sondern die Autostartroutine muß sich den neuesten Programmnamen heraussuchen (Beispiele für die Namen des zu ladenden Hauptprogrammes sind: 2DINPUT112, 2DINPUT113, 2DINPUT114 usw.)

Beispielprogramm:

```
10 ! Autostartprogramm zum automatischen Laden der neuesten Version
20 ! des Hauptprogrammes sowie für diverse Systemeinstellungen
30 DIM C$[25]
40 LOADBIN "SYSEXT" ! Bei Verwendung des SYSEXT-ROMs kann die Zeile
50                  ! entfallen
60 PRINTER IS 704 @ PLOTTER IS 705 ! Systemeinstellungen
70 !
80 !
90 ! Platz für weitere Anweisungen
100 !
110 !
120 TEST$="2DINPUT" @ NR=0
130 C$=DCAT$ (1)
140 LOOP @ IF C$="" THEN GOTO ENDE
150 IF C$[1,7]="2DINPUT" THEN NR=MAX (NR,VAL (C$[8]))
160 C$=DCATNEXT$ @ END LOOP
170 ENDE: CLEAR @ AWRITE 0,0,"2DINPUT"&VAL$ (NR)&" wird geladen"
180 CHAIN "2DINPUT"&VAL$ (NR)
190 END
```

Das hier gezeigte Programm wird unter dem Namen Autost auf der Diskette gespeichert.

FLOCATE (Filename)

Die Funktion sucht den mit dem Filenamen angegebenen File auf dem momentan selektierten Massenspeicher. Wenn der File im Katalog verzeichnet ist, so gibt die Funktion die Position des ersten Records mit dem der File anfängt zurück. Auf diesen Record kann dann mit RSECTOR und WSECTOR sehr einfach zugegriffen werden. Ist der File auf der Diskette nicht vertreten, so gibt die Funktion eine 0 als Resultat zurück. Ein File kann abhängig von der Größe des Kataloges an einem der Records im Bereich zwischen 3 und 65535 anfangen. Wenn ein Massenspeicher "HP8290X" verwendet wird, so kann das Ergebnis eine Zahl im Bereich zwischen 3 und 1059 sein, da nur die großen Winchesterlaufwerke, die der HP86/87 nur mit dem SS-ROM ansprechen kann bis 17 MB (65535 Records) beschrieben werden können.

In dem String, der den Filenamen angibt ist auch die Spezifizierung eines Volumelabels oder Massenspeicherspezifikators erlaubt.

Die Anweisung FLOCATE sollte nicht in Verbindung mit PRINT benutzt werden, da zwei gleichzeitig ablaufende I/O-Vorgänge leicht zu Fehlermeldungen führen können.

```
RSECTOR String, Record#, Msus$
```

Mit dieser Anweisung kann ein beliebiger Record vom, mit dem Massenspeicherspezifikator angegebenen Laufwerk, in die Stringvariable gelesen werden. Die Stringvariable muß vor dem Aufruf dieser Anweisung auf eine Länge von mindestens 256 Zeichen vordimensioniert werden, sonst wird der Error 56 STRING OVF erzeugt.

Der Massenspeicherspezifikator (MSUS$) darf kein Volumelabel sein, d.h. er muß die Form ":Dxxx" haben, da sonst Fehler auftreten können.

Abhängig vom angeschlossenen Laufwerk kann eine unterschiedlich große Anzahl von Records angesprochen werden. Bei den "HP8290X"-Laufwerken können die Records 0 bis 1119 und bei den neueren, mit dem SS-ROM getriebenen Laufwerken die Records 0 bis 65535 gelesen werden. Wird versucht, einen Record zu lesen, der nicht auf der Diskette existiert, so generiert die Anweisung eine Fehlermeldung.

```
WSECTOR String, Record#, MSUS$
```

Die Anweisung schreibt einen 256 Byte langen String ab dem angegebenen Record auf den mit dem MSUS$ deklarierten Massenspeicher. Der String muß eine Länge von 256 Bytes haben. Ansonsten gelten die gleichen Konventionen wie bei der RSECTOR-Anweisung.

2.2 Mathematische Funktionen

Die auf den folgenden Seiten vorgestellten Funktionen dienen zur Bit-, String-, und Zahlenmanipulation sowie zur Umrechnung von Strings in Zahlen und umgekehrt. In ihrer Gesamtheit ergeben die Befehle eine wertvolle Erweiterung zur Beschleunigung von I/O-Prozessen, Erleichterung von Bitmanipulationen sowie zur Dateiverwaltung.

ADR$ (Zahl)

Die Funktion wandelt eine Zahl im Bereich zwischen 0 und 16777215 in einen drei Byte langen String um. Um eine Kompatibilität zu der im Computer üblichen Notation zu erhalten, erzeugt die Funktion einen "umgedrehten" Kode, d.h. das Zeichen mit der kleinsten Wertigkeit steht vorn und jenes mit der größten Wertigkeit hinten. Der erzeugte String hat immer eine Länge von drei Bytes; somit erzeugt die Funktion Strings, wie sie der Computer in seinem Speicher für die Adresshandhabung benutzt.

ADR (String)

Die Funktion wandelt einen bis zu drei Byte langen String in eine Zahl im Bereich zwischen 0 und 16777215 um. Auch diese Funktion benutzt zur Umrechnung einen "umgedrehten" Kode, das soll von den folgenden Beispielen illustriert werden:

```
ADR ("■■■") ist 16777215
ADR ("■■")  ist    32639
ADR ("■")   ist      255
ADR ("123") ist  3355185
```

Die Funktion kann in Verbindung mit PEEK$ und EMC PEEK$ hervorragend zur Berechnung von Systemadressen herangezogen werden. Um z.B. den Anfang des BASIC-Programmes auszurechnen, muß man nur an einer Adresse Namens FWCURR (100006 oktal) nachschauen, und diese drei Bytes in eine Zahl umwandeln lassen. Um dies zu praktizieren ist folgende Eingabe vorzunehmen:

```
ADR (PEEK$ (100006,0,3))
```

Es lassen sich noch hunderte weiterer solcher Beispiele anführen, das hier aufgezeigte soll jedoch genügen. Wenn die im eben aufgeführten Beispiel entstandene Adresse im oktalen Format benötigt wird, so kann die Dezimalzahl mit der Funktion D_O in eine Oktalzahl umgewandelt werden.

Die beiden eben aufgeführten Funktionen sind noch hervorragend zum Aufbau von Karteiverwaltungsprogrammes o.ä. geeignet, da besonders bei derartigen Programmen oftmals Zahlen in Strings und Strings in Zahlen umgerechnet werden müssen.

Wenn die umgedrehte Notation mißfallen sollte, so kann diese bequem mit der Funktion REV$ wieder in das herkömmliche Format gebracht werden.

AND$ (String# 1, String# 2)

Die Funktion bildet das logische UND der beiden Stringausdrücke. Das logische UND erzeugt folgende Resultate in der Wahrheitstabelle:

	wahr	unwahr
wahr	wahr	unwahr
unwahr	unwahr	unwahr

Als wahr respektive unwahr wird jeweils der Zustand der einzelnen Bits in einem Zeichen bezeichnet. Ist ein Bit gesetzt oder 1 , so ist sein Zustand wahr, ist es gelöscht oder 0, so ist sein Zustand unwahr.

Mit der AND$-Funktion können nun einzelne Bits der Zeichen in einem String gelöscht werden; ist das zu löschende Bit bereits 0, so verbleibt es in diesem Zustand.

Die sich aus der Anwendung der logischen Stringfunktionen ergebenden Möglichkeiten sind mannigfaltig. So können z.B. invers-Video-Strings (das achte Bit ist gesetzt) wieder in die Normalform gebracht werden, indem mit der Funktion AND$ in jedem Zeichen das achte Bit gelöscht wird. Um das achte Bit eines Zeichens zu löschen, ist das Zeichen unter Benutzung der AND$-Funktion mit einem Zeichen zu verknüpfen, bei dem das achte Bit unwahr (0) und die vorderen sieben Bits wahr (1) sind. Dadurch ergibt sich, daß alle Bits im Zeichen, die eine Position kleiner acht innehaben stehenbleiben (wenn sie im Zustand wahr sind), da ihr "Gegenüber" wahr ist, das achte Bit jedoch wird gelöscht, wenn es den Zustand 1 hat bzw. bleibt gelöscht, wenn sein Zustand 0 ist, da das achte Komplementbit 0 ist.

Beispiel: Das achte Bit eines Zeichens soll gelöscht werden:

Das Zeichen ist hier das inversvideo-"A", mit dem ASCII-Kode 193 und der Bit-Kombination (11000001). Das für die AND$Funktion zugehörige Gegenstück ist das Zeichen 127 (01111111). Der Computer macht folgendes:

```
       inversvideo-A = ASCII 193 = Bitkombination 11000001
AND    Zeichen "■"   = ASCII 127 = Bitkombination 01111111
ergibt Zeichen "A"   = ASCII  56 = Bitkombination 01000001
```

Nicht so abstrakt und etwas praxisnäher ist das folgende Beispiel:

Beispiel: Ein String A$ enthält Inversvideo- und Normalzeichen, dieser String soll in Normalzeichenform ausgedruckt werden.

```
PRINT AND$ (A$,RPT$(CHR$ (127),LEN (A$)))
```

Ein großes Anwendungsgebiet für AND$ und weitere boolsche Funktionen ergibt sich noch bei der Dateiverwaltung, da Strings z.B. hervorragende Informationsträger für das Vorhandensein sowie Nichtvorhandensein von Dateien, Artikeln etc. sind, denn jedes Bit in einem String kann mit seinem Zustand wahr (1) oder unwahr (0) eine Aussage darüber ma-

chen, ob die Datei bzw. der Artikel usw. vorhanden ist oder nicht. Das Löschen eines Artikels kann hier mit AND$ vorgenommen werden

Bei der Anwendung der AND$-Funktion muß darauf geachtet werden, daß beide in der Funktion angegebenen Stringargumente die gleiche Länge haben, da die Funktion sonst einen Error generiert. An Stelle einen Error zu generieren einfach den kürzeren der beiden Strings zu verwenden, wäre gemäß der boolschen Mathematik inkorrekt und würde auch beim Programmieren zu einer unsauberen Arbeitsweise verleiten.

BLANK$ (Zahl)

Die Funktion erzeugt einen aus Leerzeichen (ASCII 32) bestehenden String. Die Funktion kann dazu herangezogen werden, um Stringvariablen zu initialisieren, oder bei der Generierung von Bildschirmmasken Teile des Bildschirmes selektiv zu löschen.

Die Alternative zur BLANK$-Funktion ist die Funktion RPT$. Jedoch arbeitet BLANK$ ca. 2.5 mal schneller als RPT$ und belegt zudem noch weniger Bytes im Computerspeicher als RPT$.

BSET? (String, Zahl)

Die Funktion dient zum Abfragen eines Bitmusters in einem String. Mit der Zahl wird ein Bitmuster definiert, nach welchem die Funktion den String durchforstet. Als Resultat liefert die Funktion die Position des Zeichens im String, bei dem mindestens eines, der mit der Zahl angegebenen Bits gesetzt ist. Ist keines der Bits im String vertreten, so gibt die Funktion eine 0 zurück.

Die Kodierung der Zahl ist denkbar einfach. Es werden gemäß unten stehender Tabelle die Zahlen addiert, deren zugehöriges Bit abgefragt werden soll.

abzufragende(s) Bit(s)	Zahl
1	2^0 = 1
2	2^1 = 2
3	2^2 = 4
4	2^3 = 8
5	2^4 = 16
6	2^5 = 32
7	2^6 = 64
8	2^7 = 128

Beispiel: Es soll überprüft werden, ob in einem String das erste, fünfte oder achte Bit gesetzt ist.

```
Zahl=2^0+2^4+2^7          oder
Zahl=2^7+2^4+1            das ist in Rechnernotation:
Zahl=2^7+2^4+1            Ergebnis:
Zahl=145
```

Beispiel: Überprüfung diverser Strings mit der 145'er Bitkodierung

```
BSET? ("abs",145) = 1     BSET? ("nN&*",145) = 0
BSET? ("HP",145)  = 2     BSET? ("ö",145)    = 1
```

Inversvideo-Zeichen würden in diesem Fall immer ein Resultat größer 0 liefern, da ihr achtes Bit gesetzt ist.

CBIT$ (String, Zahl)

Mit dieser Funktion können ebenfalls Bits in einem String gelöscht werden. Die Negation des der Zahl entsprechenden ASCII-Zeichens wird mit allen Zeichen im String durch ein AND verknüpft. Für die Zahl kommt ein Wert im Bereich zwischen 0 und 256 in Frage. Aus der Position des zu löschenden Bits ergibt sich die zu verwendende Zahl. Folgende Tabelle soll bei der Ermittlung der Zahl helfen:

zu löschende(s) Bit(s)	1	2	3	4	5	6	7	8
Zahl	2^0 1	2^1 2	2^2 4	2^3 8	2^4 16	2^5 32	2^6 64	2^7 128

Wenn mehrere Bits in einem Zeichen gelöscht werden sollen, so sind die entsprechenden Zahlen zu addieren.

Beispiel: Das kleine inversvideo-"k" (ASCII 235) soll in ein großes normal-"K" (ASCII 75) umgewandelt werden.

Um die Umwandlung vorzunehmen ist das achte und das sechste Bit zu löschen, nach obiger Tabelle lautet die zu verwendende Zahl also 160. Vorausgesetzt, der Ausgangsbuchstabe steht im A$, lautet die Eingabe:

```
CBIT$ (A$,160)
```

Es ist darauf zu achten, das sich die CBIT$-Funktion auf den gesamten String bezieht, d.h. alle Zeichen des Strings werden mit der Negation der Zahl durch AND verknüpft. Somit kann die im umseitig aufgezeigten zweiten Beispiel gestellte Aufgabe auch anders gelöst werden.

Beispiel: Der String A$ enthält Inversvideo- und Normalzeichen und soll in Normalzeichenform ausgedruckt werden.

```
PRINT CBIT$ (A$,128)
```

CHKSUM (String)

Die Funktion berechnen die Checksumme des Strings. Als Checksumme wird die Summe aller numerischen Äquivalente des Strings bezeichnet. Der String kann eine Länge bis zu 8192 Bytes haben.

Die Anwendungsmöglichkeiten der CHKSUM-Funktion sind mannigfaltig. Im grunde genommen dient die Checksumme eines Strings jedoch immer zur Fehlerprüfung hauptsächlich bei Datenübertragungsprozessen. Hierbei wird die Checksumme immer in den letzten Bytes eines Datenübertragungsprozesses gesendet. Der Empfänger ermittelt dann ebenfalls die Checksumme und vergleicht sein Resultat mit der empfangenen Checksumme.

Oftmals wird nicht die komplette Checksumme sondern nur ihre letzten beiden Bytes übertragen. Das soll im folgenden Beispiel erläutert werden.

Beispiel: Berechnen der Checksumme eines Strings und übertragen der letzten beiden Bytes der Checksumme hinter dem String

```
1000 ! Der zu sendende String ist der A$
1010 ! Der Empfänger steht in der Variablen ISC
1020 M$=REV$ (ADR$ (CHKSUM (M$) MOD 65535))
1030 OUTPUT ISC;A$&M$[2,3]
1040 RETURN
```

Das Berechnen des Modulowertes der Checksumme mit 65535 bewirkt, daß als Resultat nur die beiden Bytes mit der niedrigsten Wertigkeit übrigbleiben. Und da die Funktion ADR$ einen drei-Byte-String, mit für diesen Anwendungsfall falscher Reihenfolge erzeugt, muß der String noch mit REV$ umgedreht und beim Senden durch die Verwendung der Subscriptklammern gekürzt werden.

D_O (Zahl)

Die Funktion wandelt eine dezimale Zahl in eine Oktalzahl um, wobei sie eine drei-Byte-Umwandlung ohne Vorzeichen vornimmt. D.h. die dezimale Zahl kann einen Wert im Bereich zwischen 0 und 16777216 haben.

Eine besondere Bedeutung kommt dieser Funktion insbesondere in Verbindung mit den PEEK- und POKE-Anweisungen zu, denn alle diese Anweisungen arbeiten mit Oktalzahlen in der Adressangabe. Um nun die mit den herkömmlich dezimal arbeitenden Funktionen und Statements ermittelten Werte für das "PEEKen" und "POKEn" verwendbar zu machen, benötigen Sie eine dezimal in oktal-Umwandlung, die Ihnen hiermit zur Verfügung steht.

Die Umkehrung der Funktion D_O ist die Funktion O_D.

FACT (Zahl)

Die Funktion berechnet die Fakultät einer Zahl. Als Fakultät einer Zahl n wird die multiplikation aller Zahlen 1 bis n miteinander bezeichnet. Einer besonderen Bedeutung kommt die Fakultätsfunktion in der Statistik zu, z.B. um Permutationen oder Kombinationen zu berechnen.

Um die Anzahl der Kombinationsmöglichkeiten bei jeweils N Zügen aus X Elementen zu bestimmen, ist folgende Formel anzuwenden

$$K = \frac{X!}{(X-N)!*N!}$$

Beispiel: Wie viele Kombinationsmöglichkeiten gibt es beim Lottospiel 6 aus 49?

```
FACT (49)/(FACT (49-6)*FACT (6))
```

Ergebnis: 13.983.816 Kombinationsmöglichkeiten

Wenn Sie die gleiche Rechnung für das Spiel 7 aus 38 vornehmen, werden Sie sehen, daß sich als Resultat annähernd die gleiche Anzahl von Kombinationsmöglichkeiten ergibt.

Beispiel: Wie viele Akkorde von vier Tönen können einem Klavier mit 88 Tasten entlockt werden, wenn jeweils vier Tasten gleichzeitig angeschlagen werden?

```
FACT (88)/(FACT (88-4)*FACT (4))
```

Ergebnis: 2.331.890 Akkorde

HEX$ (String)

Die Funktion wandelt einen String mit einer Länge von maximal 16384 Zeichen in die hexadezimale Notation um. Die Nützlichkeit des hexadezimalen Formats braucht wohl nicht näher beschrieben werden.

Anwendungsbeispiel sind: Anzeigen und drucken von Speicherinhalten, Konvertierung von Zeichenketten bei der Datenübertragung usw.

Beispiel: Umwandlung einer Dezimalzahl in eine Hexadezimalzahl

```
HEX$ (ADR$ (Zahl))
```

HEX_ASC$ (String)

Die Funktion wandelt einen hexadezimal kodierten String in einen ASCII-String um. Diese Funktion ist die Umkehrung der Funktion HEX$.

Wenn die Stringlänge ungerade, d.h. das High-Nibble undefiniert ist, so wird das High-Nibble von der Funktion automatisch auf Null gesetzt.

Beispiel: Umwandlung einer Hexadezimalzahl in eine Dezimalzahl

```
ADR (HEX_ASC$ (String))
```

NUMBER? (String)

Die Funktion überprüft, ob an Anfang des Strings eine Zahl steht, und wenn dies der Fall sein sollte, ob es sich um eine Integer- oder um eine Real-Zahl handelt. Die Funktion soll zur Abwendung eines Fehlers beim Verwenden der VAL-Funktion dienen. Ist am Anfang des Strings keine Zahl enthalten, so gibt die Funktion als Resultat eine 0 zurück, eine Integerzahl generiert eine 2 und eine Realzahl eine 1.

Die Funktion überprüft nur die Zeichen des Strings, nicht jedoch die Wertigkeit der Zahl bei zusammengesetzten Ziffernzeichen, d.h. der String "1234567890123" ergibt seinen Ziffern entsprechend eine Integerzahl, und die Funktion gibt eine 2 als Resultat zurück, der Computer allerdings interpretiert diese Zahl bei Verwendung der VAL-Funktion als Realzahl, da es sich um einen Integeroverflow handelt. Das Gleiche trifft natürlich bei der Überprüfung einer zu großen Realzahl zu, z.B. ergibt NUMBER? ("12345.12E600") eine 1, da sich in dem String ausschließlich Realzahl-Zeichen befinden, VAL ("12345.12E600") allerdings ergibt eine Overflow Warning-Meldung.

O_D (Oktalzahl)

Die Funktion wandelt eine Oktalzahl im Bereich zwischen 0 bis 77777777 in eine Dezimalzahl um. Besondere Bedeutung kommt dieser Anweisung bei der Verwendung der PEEK- und POKE-Anweisungen zugute. Auch beim Arbeiten mit dem Assemblerhandbuch können die dort verwendeten Oktalzahlen leicht in Dezimalzahlen umgerechnet werden.

ODD (Zahl)

Die Funktion gibt eine 0 zurück, wenn die Zahl gerade ist, ansonsten eine 1. Besondere Bedeutung bekommt die Funktion überall dort, wo anhand der Parität einer Zahl eine Enscheidung getroffen werden soll. Ein Beispiel der Anwendung von ODD ist im Integrationsprogramm im Kapitel 2.3.6 zu finden.

Die Funktion ODD ersetzt folgenden BASIC-Ausdruck:

```
NOT (Zahl/2=Zahl DIV 2)
```

OR$ (String# 1, String# 2)

Die Funktion bildet das logische ODER der beiden Stringausdrücke. Das logische ODER erzeugt folgende Resultate in der Wahrheitstabelle:

	wahr	unwahr
wahr	wahr	wahr
unwahr	wahr	unwahr

Als wahr oder unwahr wird der jeweilige Zustand der einzelnen Bits in einem Zeichen bezeichnet. Ein gesetztes Bit (1) wird als wahr und ein gelöschtes Bit (Ø) als unwahr bezeichnet.

Mit der OR$-Funktion können einzelne Bits in einem String gesetzt werden. Um in einem String ein Bit zu setzen, muß ein entsprechender Maskenstring erstellt werden, bei dem das in dem Originalstring zu setzende Bit bereits eins ist. Wird nun der Maskenstring und der Originalstring durch ein logisches ODER verknüpft, so ist das Resultat gemäß oben stehender Wahrheitstabelle ein String, bei dem alle die Bits gesetzt sind, die entweder im Originalstring, im Maskenstring oder in beiden Strings gesetzt sind.

Beispiel: Bei allen Zeichen eines Strings soll das achte Bit gesetzt werden, d.h. der String soll inversvideo dargestellt werden (HGL$).

```
OR$ (A$, RPT$ (CHR$ (128),LEN (A$)))
```

REV$ (String)

Die Funktion gibt als Resultat den String in umgekehrter Reihenfolge zurück.

ROUND (Zahl, Stelle)

Die Funktion rundet die Zahl an der angegebenen Stelle. Als Stelle wird hier die absolute Position der Ziffer in der Mantisse relativ zur ersten Ziffer bezeichnet. Das heißt, es wird keine Nachkommarundung vorgenommen, da das Dezimalkomma (bzw. der Dezimalpunkt) ohnehin bei seiner Position von der Größe des Exponenten abhängt. Zum Beispiel steht das Dezimalkomma bei Zahlen kleiner als 10^{13} immer als Trennungszeichen zwischen den Zifferngruppen ≥ 1 und <1, dagegen steht es bei Zahlen, die größer als 10^{12} sind, immer hinter der ersten Ziffer. Es hätte also keinen Sinn, als Rundungsstelle die Anzahl der Positionen hinter dem Komma anzugeben.

Beispiel: Diverse mit Round gerundete Zahlen

```
ROUND (1234567890,4)   = 1235000000
ROUND (1.234567890,4)  = 1.235
ROUND (0.012345678,4)  = .01235
ROUND (123,0)          = 0
ROUND (123,1)          = 100
```

RPT$ (String, Anzahl)

Die Funktion gibt als Resultat einen String zurück, in dem der in der Funktion angegebene String entsprechend der Anzahl n-mal wiederholt wurde. Auf diesem Wege können leicht große, bereits vordimensionierte Strings auf Standartparameter gesetzt werden. Die Funktion kann in Verbindung mit den CRT-Manipulationsstatements auch zur schnellen Erzeugung von Bildschirmmasken herangezogen werden.

Die Funktion sollte allerdings nicht zur Erzeugung eines Leerzeichenstrings herangezogen werden, da es hierfür die spezielle BLANK$-Funktion gibt.

Ein mit RPT$ erzeugter String kann die Länge von 32767 Zeichen nicht überschreiten, da sonst ein Fehler erzeugt wird.

Beispiel: Bildschirm invers-video löschen

```
RPT$ ("■",1920)
```

SBIT$ (String, Zahl)

Die Funktion liefert als Resultat einen String, bei dem alle mit der Zahl angegebenen Bits gesetzt sind, d.h. das Stringäquivalent der Zahl wird mit jedem Zeichen des Strings durch ein logisches ODER verknüpft.

Die Kodierung der Zahl gestaltet sich wie bereits auf Seite 10 und Seite 11 bei den Befehlen BSET? und CBIT$ beschrieben.

Beispiel: Ein String A$ soll in der inversvideo-From angezeigt werden.

```
DISP SBIT$ (A$,128)
```

TRIM$ (String)

Die Funktion gibt als Resultat einen String zurück, bei dem alle führenden und anhängenden Leerzeichen gelöscht sind. Dabei ist zu beachten, daß hier als Leerzeichen nur das ASCII-Zeichen 32 bezeichnet wird. Das Zeichen Nr. 13 wird von der TRIM$-Funktion nicht berücksichtigt, obwohl es auf dem Bildschirm ebenfalls einen Leerraum erzeugt.

XOR$ (String#1,String#2)

Die Funktion bildet als Resultat das logische EXKLUSIV ODER von String#1 und String#2. Die Wahrheitstabelle dieser logischen Operation hat folgendes Aussehen:

	wahr	unwahr
wahr	unwahr	wahr
unwahr	wahr	unwahr

Die Funktion kann also dazu herangezogen werden, ein Bit welches bereits gesetzt ist, zu löschen und und jenes welches gelöscht ist, zu setzen. Auf diesem Wege steht dem Programmierer ein einfach zu handhabender binärer Schalter zur Verfügung. Die Bits eines Strings, die z.B. als Flaggen zur Programmsteuerung verwendet werden, können so auf einfachem Wege ein- oder ausgeschaltet werden.

Beispiel: Die inversvideo-Zeichen eines Strings sollen in Normalzeichen und die Normalzeichen in inversvideo-Zeichen umgewandelt werden.

```
XOR$ (A$, RPT$ (CHR$ (128),LEN (A$)))
```

2.3 Advanced BASIC-Funktionen und -Statements

Die im folgenden aufgeführten Befehle werden nicht in alphabetischer Reihenfolge gelistet, da einige Befehle nur in Kombination mit anderen Befehlen einen Sinn ergeben. Jeder Befehlsgruppe ist daher ein eigenes Kapitel gewidmet.

2.3.1 Abfragen von ROMs und Binärprogrammen

BPGM? (Zahl)

Der Systemspeicher wird nach dem mit der Zahl angegebenen Binärprogramm durchsucht. Dem Programmierer wird hiermit eine einfache Möglichkeit in die Hand gegeben, ohne ON ERROR-Absicherung das Vorhandensein eines Binärprogrammes zu überprüfen. Gerade bei Programmen, die mit Binärprogrammen arbeiten und des öfteren neu gestartet werden, kann mit diesem Befehl eine sehr elegante Programmiertechnik aufgebaut werden.

Beispiel: Ein Programm benötigt die Binärprogramme TDGRAPH und BINCALC1 (die beiden Binärprogramme haben die Nummern 41 und 48)

```
10 IF NOT BPGM? (41) THEN LOADBIN "TDGRAPH"
20 IF NOT BPGM? (48) THEN LOADBIN "BINCALC1"
```

Für den Fall, daß Ihnen die Nummer eines Binärprogrammes nicht bekannt ist, hilft Ihnen folgende Routine die Nummer herauszusuchen:

```
10 FOR I=0 TO 255
20 IF BPGM? (I) THEN DISP "Das Binary hat die Nummer:";I
30 NEXT I
40 END
```

Sie brauchen das Binärprogramm nur noch zu laden und die Routine zu starten. Wenn Sie übrigens nicht das SYSEXT-ROM sondern das Binärprogramm einsetzen, so gibt Ihnen die Routine natürlich zwei Identifikationsnummern aus, das SYSEXT-Binary hat die Nummer 56.

Bei dieser Gelegenheit möchte ich Sie gleich darauf hinweisen, daß der Computer es nicht zuläßt, daß mehere Binärprogramme mit der gleichen Nummer geladen werden, da er die Nummer zu Idendifikation und Unterscheidung der Binärprogramme heranzieht. Zwei Binärprogramme können also, selbst wenn es sich um völlig unterschiedliche Binaries handelt nicht zur gleichen Zeit im Rechner existieren; beim Versuch das zweite Binärprogramm zu laden, erzeugt der Computer einen BAD BIN LOAD ERROR.

ROM? (Zahl)

Die Funktion gibt eine 1 zurück, wenn das mit der Zahl angegebene ROM vorhanden ist, ansonsten eine 0. Zwei offensichtliche Anwendungsgebiete dieser Funktion liegen auf der Hand: Erstens kann der Benutzer leicht überprüfen, welche ROMs in den Rechner eingesteckt sind, und zweitens kann ein Programm, welches mit der Funktion erkannt hat, daß ein ROM fehlt, entweder andere Unterprogrammebenen aufrufen oder mit einer Fehlermeldung auf dem Bildschirm den Anwender über das Fehlen des ROMs informieren. So wird vermieden, daß ein Programm, nachdem es schon mehrere Minuten oder Stunden läuft, plötzlich auf Grund eines fehlenden ROMs aussteigt.

Beispiel: Programm zum Anzeigen der eingesteckten ROMs

```
10 DIM A$[30]
20 FOR I=0 TO 255
30 A$=""
40 IF NOT ROM? (I) THEN GOTO 90
50 ON ERROR GOTO 80
60 RESTOREX I+200
70 READ A$
80 DISP "ROM# ";I;" ";A$
90 NEXT I
100 PAUSE
200 DATA System-Main
201 DATA System-Graphics
214 DATA MIKSAM
224 DATA Language
240 DATA Assembler
256 DATA Sysext
376 DATA Matrix 1
377 DATA Matrix 2
392 DATA I/O
407 DATA Extendet-Massstorage
408 DATA System-Massstorage
409 DATA Electronic-Disc
431 DATA Advanced-Programming 2
432 DATA Advanced-Programming 1
440 DATA Plotter-Driver
```

Geben Sie dieses Programm ein und starten Sie es, auf dem Bildschirm erscheinen alle derzeitig im Computer eingesteckten ROMs.

Eine Möglichkeit, Programmerrors mit dem Befehl abzufangen ist die folgende.

Beispiel: Abfangen eines Errors bei nicht vorhandenem Plotterausgabe-ROM

```
5000 PLOTTINGROUTINE:
5010 IF ROM? (240) THEN PLOTTER IS 704 @ GOTO 5030
5020 DISP "Kein Plotterausgabe-ROM vorhanden" @ PLOTTER IS 1
5030 !
5040 ! Hier beginnt Ihre Routine
5050 !
```

2.3.2 Handhabung mehrerer ON ERROR-Statements

Die Möglichkeit der Serie 80 mit ON ERROR-Statements bei Auftreten von Programmfehlern einen Programmstop zu vermeiden, ist an sich eine feine Sache. Mitunter macht es sich jedoch störend bemerkbar, daß nicht mehrere ON ERROR-Statements sozusagen aufeinander gestapelt aktiv sein können, besonders beim Aufrufen von Funktionen oder Unterprogrammen wäre eben genanntes oftmals wünschenswert. Mit zwei vom Sysext-ROM zur Verfügung gestellten Befehlen können alle, die eben genannten Schwierigkeiten umgangen werden. Bevor jedoch auf die beiden neuen Befehle eingegangen wird, wird im Folgenden kurz erklärt, was im Computer vor sich geht, wenn das Programm ein ON ERROR-Statement abarbeitet und danach, irgenwann im Programmlauf, auf eine Fehlerbedingung trifft.

Im Programmspeicher gibt es eine Speicherstelle namens ERGOTO. Bei Programmstart wird diese Speicherstelle auf 0 gesetzt. Tritt nun während des Programmlaufes ein Fehler auf, so überprüft das System die genannte Speicherstelle. Ist in der Zwischenzeit kein ON ERROR-Statement aktiviert worden, so steht bei ERGOTO nach wie vor eine 0, und das System gibt auf Grund dessen eine Fehlermeldung auf dem Bildschirm aus und hält das Programm an.

In dem Augenblick jedoch, in dem das System ein ON ERROR-Statement abarbeitet geschieht folgendes: Die Routine des ON ERROR-Statements lädt den BASIC-PC (Programmcounter) und speichert ihn nach ERGOTO. Erkennt das Betriebssystem im Laufe des weiteren Programmes nun eine Fehlerbedingung, so bemerkt es, daß die Speicherstelle ERGOTO ≠ 0 ist, also lädt das System drei Bytes von ERGOTO und speichert sie zum BASIC-PC. Die Wirkung ist die Gleiche, als ob beim Auftreten eines Fehlers mit einem GOTO direkt hinter die auf das ON ERROR-Statement folgende Anweisung gesprungen wird, d.h. in Folge des Fehlers wird die, auf das ON ERROR-Statement folgende Anweisung abgearbeitet. Das Betriebssystem sorgt noch dafür, daß nur eine Anweisung (GOTO oder GOSUB) abgearbeitet wird, die hinter dem @-Zeichen folgenden Anweisungen werden beim Auftreten eines Fehlers nicht berücksichtigt.

ERRBR?

Diese parameterlose numerische Funktion gibt eine Realzahl zurück, über deren Bedeutung sich der Anwender an sich keine Gedanken machen braucht. Es handelt sich bei der zurückgegebenen Zahl, um den Wert, der an der Speicherstelle ERGOTO steht. Ist die Zahl 0, so ist zur Zeit kein ON ERROR-Statement im Programm aktiv. Ist die Zahl ≠ 0, so handelt es sich bei der Zahl um einen Zeiger, der auf die Anweisung zeigt, die hinter dem ON ERROR-Statement steht.

Die von der Funktion zurückgegebene Zahl sollte vom Programm zwischengespeichert werden, da sie später noch Verwendung findet. Nachdem die Zahl gespeichert wurde, kann ein neues ON ERROR-Statement abgearbeitet werden. Das alte ON ERROR-Statement kann mit der Zahl jederzeit wieder reaktiviert werden.

SET ERRBR Zahl

Mit diesem Statement und der durch ERRBR? gewonnenen Zahl kann ein altes ON ERROR-Statement jederzeit wieder reaktiviert werden. Besonders sinnvoll ist der Einsatz von ERRBR? und SET ERRBR am Anfang und am Ende von Unterprogrammen. Mit ERRBR? wird am Anfang des Unterprogrammes das alte ON ERROR-Statement zwischengespeichert. Das Unterprogramm kann nun seine eigenen ON ERROR Statements aktivieren, um Fehler, die während der Abarbeitung des Unterprogrammes auftreten, abzufangen. Bevor jedoch mit RETURN zum aufrufenden Hauptprogramm zurückgesprungen wird, reaktiviert das Unterprogramm mit SET ERRBR die alte ON ERROR-Anweisung.

Beispiel: Zwischenspeichern eines ON ERROR-Statements und seine Reaktivierung.

```
1000 Unterprogramm: Branch=ERRBR?
1010 ON ERROR GOTO Error_handling
1020 !
1030 ! Hier steht Ihr Programm
1040 !
1050 SET ERRBR Branch @ RETURN
1060 Error_handling:
1070 !
1080 ! Hier steht die Fehlerhandhabungsroutine
1090 ! des Unterprogrammes
1100 !
1110 GOTO 1050
```

Bei Beschreitung dieses Weges und Verwendung verschiedenen Variablen zum Zwischenspeichern der ERRBR?-Resultate, können sie an sich beliebig viele ON ERROR-Statements aktivieren und reaktivieren.

Die hier aufgezeigte Vorgehensweise ist natürlich auch bei benutzerdefinierten Funktionen anwendbar.

2.3.3 Erweiterte Bildschirmsteuerung

Das man mit den "normalen" DISP-Anweisungen keine Maskeneingabe formulieren kann, dürfte wohl jedem klar sein. Auch arbeiten die DISP-Anweisungen relativ langsam. Mit den von Sysext zur Verfügung gestellten neuen Bildschirmsteuerungsanweisungen gehören diese Probleme der Vergangenheit an. Die drei im Folgenden beschriebenen CRT-Anweisungen ermöglichen eine absolute Kontrolle des Bildschirmes.

AWRITE Reihe, Spalte [,String]

Dieses Statement setzt den Kursor auf die angegebene Reihen-Spalten-Position und schaltet ihn aus. Die kleinste Reihen-Spaltenposition ist 0,0. Die größte Reihenposition ist abhängig vom eingestellten PAGESIZE-Format. Bei PAGESIZE 16 ist sie 15, bei PAGESIZE 24 ist sie 23. Wird für die Spaltenposition eine Zahl größer als 79 angegeben, so zählt das Statement auf der nächsten Zeile weiter. Es ist also gleichgültig, ob AWRITE 0,260 oder AWRITE 2,20 eingegeben wird.

Die Reihen-Spaltenangabe des AWRITE-Statements bezieht sich immer relativ auf die momentan angezeigte Seite. Zum definieren des Bildschirmfensters kann die Anweisung also nicht herangezogen werden (siehe dazu START CRT AT).

Wenn für die Reihenangabe Parameter herangezogen werden, die außerhalb des Bildschirmfensters liegen, so wird der Kursor auch in diesem Fall korrekt positioniert, wenn auch außerhalb der Bildschirmseite.

Durch den optionalen Stringparameter können an der angegebenen Reihen-Spaltenposition Texte ausgegeben werden. Der Text wird ab der neu erreichten Kursorposition sehr schnell auf den Bildschirm geschrieben. Nach der Stringausgabe wird anders als bei DISP kein Zeilenvorschub durchgeführt, da die Anweisung die CRLF-Ausgabe unterdrückt. Vielmehr verbleibt der Kursor sogar an seiner durch die beiden Parameter angegebenen Position.

Wird AWRITE im GRAPH ALL-Modus durchgeführt, so wird die Anweisung nicht abgearbeitet. Ist der Computer dagegen im GRAPH NORMAL-Modus, so schaltet die Anweisung den ALPHA-Modus ein.

Beispiel: Löschen aller Bildschirmseiten bei ALPHA ALL und PAGESIZE 24

```
10 FOR I=1 TO 9
20 AWRITE 24,0
30 CLEAR
40 NEXT I
50 END
```

Das hier gezeigte Programm nutzt eine Wirkung des CLEAR-Statements aus. AWRITE 24,0 setzt den Kursor auf die erste Position der nächsten, auf die aktuelle folgende Bildschirmseite. CLEAR sorgt nun dafür, daß diese Seite gelöscht und zur aktuellen Bildschirmseite wird.

Beispiel: Effektvoller Aufbau einer Bildschirmmaske

```
10 DIM EFFEKT$[80]
20 PAGESIZE 24
30 EFFEKT$="Erste Zeile der Maske" @ R=0
40 GOSUB EFFEKT
50 EFFEKT$="Zweite Zeile der Maske" @ R=1
60 GOSUB EFFEKT
70 EFFEKT$="Dritte Zeile der Maske" @ R=2
80 GOSUB EFFEKT
90 !
100 ! Hier können weitere Unterprogrammaufrufe stehen
110 !
990 PAUSE
1000 EFFEKT:
1010 FOR I=23 TO R STEP -1
1020 AWRITE I,0,BLANK$ (80)
1030 AWRITE I,0,EFFEKT$
1040 BEEP I,I @ WAIT 70
1050 AWRITE I,0,BLANK$ (80)
1060 NEXT I
1070 AWRITE R,0,EFFEKT$
1080 RETURN
```

Geben Sie das Programm ein, und lassen Sie sich von der Wirkung überraschen.

Wenn mit dem AWRITE-Statement numerische Werte ausgegeben werden sollen, so müssen sie natürlich unter Benutzung von VAL$ umgewandelt werden.

AREAD String

Das AREAD-Statement füllt den angegebenen String oder Substring mit den Zeichen, die ab der momentanen Kursorposition auf dem Bildschirm stehen. Es ist darauf zu achten, daß der Kursor vor der Durchführung von AREAD ausgeschaltet wird, da er sonst mit in die Stringvariable eingelesen wird. Am einfachsten kann der Kursor dadurch ausgeschaltet werden, daß er mit dem AWRITE-Statement auf die Position gesetzt wird, ab der gelesen werden soll.

Die Anzahl der Zeichen, die von AREAD in die Stringvariable gelsen werden, hängt von der Lände der Stringvariablen ab. Ist z.B. M$ auf 30 Zeichen Länge dimensioniert, so bewirkt AREAD M$, daß 30 Zeichen in den M$ gelesen werden. AREAD A$[10,40] bewirkt, daß der A$ ab seiner 10 Position mit 31 Zeichen gefüllt wird.

Anders als bei der Verwendung von INPUT ließt AREAD auch Kommas, Anführungszeichen, CRs sowie führende oder anhängende Leerzeichen.

Beispiel: Tippen Sie folgendes Programm ein, und lassen Sie sich überraschen

```
10 DIM A$[100]
20 CLEAR @ M=0 @ A$=""
30 M,H=0 @ FOR K=1 TO 17 @ RESTORE
40 IF H THEN H=0 ELSE H=128
50 FOR I=1 TO 110
60 READ B
70 AWRITE 0,M,SBIT$ (XOR$ (CHR$ (B),"0"),H)
80 M=M+1
90 NEXT I
100 NEXT K
110 PAUSE
120 DATA 119,66,81,69,28,16,68,85,69,66,85,66,16
130 DATA 118,66,85,69,94,84,28,16,89,67,68,16,81
140 DATA 92,92,85,16,100,88,85,95,66,89,85,16,69
150 DATA 94,84,16,87,66,42,94,16,84,85,67,16,124
160 DATA 85,82,85,94,67,16,87,95,92,84,85,94,85
170 DATA 66,16,114,81,69,93,30,16,107,125,85,64
180 DATA 88,89,67,68,95,64,88,85,92,85,67,28,16
190 DATA 81,69,67,16,119,95,85,68,88,85,23,67,16
200 DATA 118,81,69,67,68,109,16
```

Sie werden bestimmt zugeben müssen, das Resultat hätten Sie nicht erwartet.

START CRT AT Zahl

Mit dieser Anweisung können Sie den Bildschirm an der mit der Zahl angegebenen Reihe starten. Das Statement zählt die Bildschirmreihen anfangend mit der 1. Abhängig davon, ob sich der Computer im ALPHA ALL- oder im ALPHA NORMAL-Modus befindet, kann als größte Reihe entweder 204 oder 54 eingegeben werden. Das Vorzeichen der Zahl spielt keine Rolle.

Nach der Abarbeitung eines START CRT AT-Statements hat der Kursor immer noch seine alte Position inne. Um ihn auf die neue Bildschirmseite zu bringen, kann man sich des CLEAR- oder AWRITE-Statements bedienen.

Beispiel: Ein sehr simples Textprogramm

```
10 DIM A$[80]
20 START CRT AT 1 @ PRINTER IS 704 @ PAUSE
30 FOR I=1 TO 62
40 START CRT AT 1
50 AWRITE 0,0 @ AREAD A$
60 PRINT A$
70 NEXT I
80 PRINT CHR$ (12)
90 END
```

Das hier gelistete Programm ist so einfach, daß man es bei Bedarf schnell auswendig eintippen und dann anwenden kann.

Nach der Eingabe wird das Programm wie üblich durch [RUN] gestartet. Die beiden Initialisierungszeilen bis zur PAUSE werden derart schnell abgearbeitet, daß man die RUN-LED kaum

blinken sieht. Nachdem das Programm abgestoppt ist, können Sie Ihren Text eingeben.

Bedenken Sie, daß die erste Zeile einer Seite die oberste Zeile der Bildschirmseite ist, mit dem das Programm stoppt.

Nachdem Sie Ihren Text eingegeben haben, drücken Sie die Taste [CONT]. Der Text wird in der Form ausgedruckt, in der er auf dem Bildschirm steht.

2.3.4 Erweiterte Kontrolle des Tastenfeldes

Bei den hier beschriebenen neuen Befehlen handelt es sich um solche, die sinnvoll im RUN-Modus, und um solche, die sinnvoll im Rechenmodus Anwendung finden. Als erstes werden die Befehle vorgestellt, die Ihnen die Programmier- und Editierarbeit im Rechenmodus erleichtern.

MASK

Mit diesem Statement wird die Tastatur maskiert und erhält dabei auf zwei noch unbelegten Tasten neue Funktionen.

* Der Taste [SHIFT] [RUN] wird der Befehl LIST zugewiesen. Und zwar nicht in der Form, wie es mit der [LIST]-Taste durch Sofortausführung zur Verfügung steht, sondern in einer der KEY-Tasten im Rechenmodus üblichen Form. Das Drücken von [SHIFT] [RUN] bewirkt, daß der Befehl LIST auf den Bildschirm geschrieben wird. Danach können die beim List üblichen Zeilenparameter eingegeben werden.

  Diese neue Tastenbelegung erspart dem Programmierer die beim Editieren unerläßliche erneute Eingabe die vier Buchstaben LIST.

* Die Taste [SHIFT] [END LINE] bekommt eine Bedeutung von besonderer Art. Sie dient nunmehr als Umschalttaste zwischen dem Normal- und dem Inversvideoschreibmodus. Durch einmaliges Drücken der Taste [SHIFT] [END LINE] wird der Inversvideoschreibmodus eingeschaltet. Alle nun gedrückten Tasten werden inversvideo auf dem Bildschirm ausgegeben. Das erleichtert die Eingabe von hervorgehobenen DISP- oder AWRITE-Programmzeilen erheblich. Auch die Abfrage der durch KEY$ (↑) erzeugten Zeichenkodes ist nun mit erheblich geringerem Programmieraufwand zu verwirklichen. Ein erneutes Drücken der Taste [SHIFT] [END LINE] schaltet den Inversvideoschreibmodus wieder aus.

  Die Tastenfeldsteuertasten (↑, ↓, →, ←, etc.) werden vom Inversvideoschreibmodus nicht betroffen, d.h. sie behalten ihre alten Funktionen.

UNMASK

Diese Anweisung schaltet die oben beschriebene Maskierung des Tastenfeldes wieder aus.

An sich kann sich der Benutzer die Eingabe von MASK und UNMASK ersparen, da das SYSEXT-ROM automatisch erkennt, wann die Tastatur maskiert werden darf, und wann sie nicht maskiert werden sollte.

Bei den folgenden Computeraktionen löst das SYSEXT-ROM ein automatisches MASK bzw. UNMASK aus:

Aktion	Resultat
Einschalten	MASK
SCRATCH	MASK
RESET	UNMASK
RUN	UNMASK
Programmhalt wegen Error	MASK

Dei weiteren in diesem Kapitel vorgestellten Anweisungen dienen zum Überwachen und steuern des Tastenfeldes im Programmmodus.

TAKE KEYBOARD

Nach der Abarbeitung dieser Taste ist die Tastatur gesperrt. Alle im folgenden gedrückten Tasten werden nicht mehr auf dem Bildschirm ausgegeben und halten während eines laufenden Programmes das Programm an, sondern sie werden in einem 80 Byte langen Speicher gepuffert.

Im Anhang stehen die Zeichenkodes, die beim Drücken der Steuertasten im Tastenfeldpuffer abgelegt werden. Die Kodes für die Tasten [SHIFT] [RUN] und [SHIFT] [END LINE] werden nicht vom Betriebssystem sondern vom SYSEXT-ROM speziell generiert. Dadurch wird erreicht, daß dem Programmierer zwei zusätzliche Steuertasten zur Verfügung stehen.

Die einzigen Tasten, die eine direkte Aktion bewirken, und nicht in den Tastenfeldbuffer geschrieben werden, sind die Steuertasten [k1]-[k14] und die Taste [RESET].

Wenn während eines laufenden Programmes bei verriegelter Tastatur ein Programmerror auftritt, der das Programm zum stehen bringt, so entriegelt das SYSEXT-ROM, wie auch aus oben stehenden Tabelle ersichtlich, automatisch die Tatatur.

RELEASE KEYBOARD

Die Tastatur wird wieder entriegelt. Alle im Folgenden gedrückten Tastn, ob im RUN- oder im Rechenmodus werden wieder vom Betriebssystem gehandhabt.

KEY$

Die KEY$-Funktion gibt das nächste Zeichen aus dem durch TAKE KEYBOARD aktivierten Buffer aus. Ist der Buffer leer, so wird ein Nullstring ausgegeben. Um bei Verwendung der NUM-Funktion den Nullstring vom ASCII-Zeichen 0 zu unterscheiden, ist wie folgt vorzugehen:

```
1000 K$=KEY$ @ IF NOT LEN (K$) THEN 1000 ELSE K=NUM (K$)
```

NOT BLOCKED KEYS ARE String

Mit dieser Anweisung kann definiert werden, welche Tasten beim TAKE KEYBOARD ungesperrt bleiben. In dem String müssen die Zeichen der Tastenkodes der ungesperrten Tasten stehen. In der Grundeinstellung des SYSEXT-ROMs, d.h. direkt nach dem Einschalten sind die Tasten [k1]-[k14] sowie [RESET] ungesperrt. Der String darf eine maximal Länge von 20 Zeichen haben. Weitere Tasten, bei denen eine Entrieglung bei TAKE KEYBOARD u.U. sinnvoll sein könnte, sind [CLEAR], [A/G], [ROLL ▽] und [ROLL Δ].

Vergessen Sie bei der Eingabe neuer ungesperrter Tasten nicht, die Tastenkodes der bereits ungesperrten Tasten erneut einzugeben, da durch das Setzten des neuen Strings der alte String aufgehoben wird.

Beispiel: Ein kleines Spiel unter Benutzung der ALPHA und KEYBOARD-Befehle

```
10 DIM A$[30]
20 TAKE KEYBOARD @ CLEAR
30 SCHRANKEN=3 @ ZAHL,COUNTER=0
40 ZEIT=1500
50 A$="" @ ZEIT=ZEIT/1.5 @ IF ZEIT<10 THEN ENDE
60 START:
70 GEDRUECKT=0 @ FOR I=1 TO 60 @ IF GEDRUECKT>75 THEN ENDE
80 B$=CHR$ (INT (RND *11)+48) @ IF B$=":" THEN B$="■"
90 A$=A$&B$
100 IF 20-LEN (A$)>SCHRANKEN THEN 130
110 IF 21-LEN (A$)>0 THEN SCHRANKEN=SCHRANKEN-1 @ A$="" @ GOTO 80
120 GOTO ENDE
130 FOR M=1 TO 10 @ GOSUB 140 @ WAIT ZEIT/8 @ NEXT M @ NEXT I @ GO
TO 50
140 AWRITE 10,31,RPT$ (":",SCHRANKEN)&BLANK$ (25)
150 IF ZAHL>9 THEN AWRITE 10,30,"■" ELSE AWRITE 10,30,VAL$ (ZAHL)
160 AWRITE 10,50-LEN (A$),A$
170 K=NUM (KEY$ ) @ IF K=44 THEN ZAHL=(ZAHL+1) MOD 11 @ GOTO 230
180 IF K#43 THEN GOTO 230 ELSE GEDRUECKT=GEDRUECKT+1
190 IF ZAHL<10 AND NUM (A$)=ZAHL+48 THEN BEEP 10,10 @ COUNTER=COUN
TER+1 @ GOTO 220
200 IF ZAHL#10 OR NUM (A$)#127 THEN BEEP 1,.1 @ GOTO 230
210 COUNTER=COUNTER+10 @ FOR O=20 TO 1 STEP -1 @ BEEP O,O @ NEXT O
220 IF LEN (A$)>1 THEN A$=A$[2] ELSE A$=""
230 AWRITE 12,30,"Stand: "&VAL$ (COUNTER) @ RETURN
240 ENDE: RELEASE KEYBOARD @ END
```

Die Regeln dieses kleinen Reaktionsspieles sind sehr einfach. Ihr Stützpunkt ist eine Ziffer, die von drei Doppelpunkten beschützt wird. Der Computer generiert nun Ziffern, die eine Kette bilden und langsam auf Ihren Stützpunkt zuwandern. Sie müssen nun versuchen, durch wiederholtes Drücken der Kommataste Ihre eigene Ziffer auf den gleichen Wert zu bringen, den die, die Zahlenkette anführende Ziffer hat. Hat Ihre Ziffer den gleichen Wert, so können Sie die [+]-Taste drücken, dadurch verschwindet die erste Zahl der Ziffernkette. Wenn Sie zu langsam sind, so wandert die Ziffernkette auf Sie zu, zerstört Ihre ":"-Barrikaden, und wenn die Kette Sie erreicht, so haben Sie verloren. Das Auslöschen der Ziffern 0-9 bringt einen Punkt, und das Zeichen "■" bringt 10 Punkte. Aber Vorsich, Sie dürfen für jeweils 60 Ziffern nur maximal 75 mal auf die [+]-Taste drücken.

2.3.5 Löschen von Unterprogrammebenen

Die Computer der Serie 80 können beim abarbeiten von Programmen bis zu 256 Unterprogrammebenen anspringen. Bei einer rationellen Programmiertechnik wird es zwar kaum geschehen, daß derartig viele Unterprogrammebenen angesprungen werden, aber ein ganz anderes Problem taucht bei der Verwendung von Unterprogrammen oftmals auf, und zwar daß die Möglichkeit fehlt, ein Unterprogramm ohne Verwendung von RETURN durch ein einfaches GOTO zu verlassen. Mit den folgenden Befehlen ist auch dieses Problem gelöst.

POP RETURN

Mit dieser Anweisung kann eine Unterprogrammebene vom Unterprogrammrücksprungstapel gelöscht werden. Ist zur Zeit keine Unterprogrammebene aktiv, so wird die Anweisung ignoriert. Nachdem mit POP RETURN die Unterprogrammebene gelöscht wurde, kann das Unterprogramm durch ein GOTO verlassen werden.

Beispiel: Vorgehensweise bei der Anwendung von POP RETURN

```
500 !
510 ! irgend ein Programmkode
520 !
530 GOSUB SPEICHERE_TEXT
540 GOTO 100
550 SPEICHERE_TEXT:
560 GOSUB TEST_MSUS
570 ASSIGN# 1 TO "TEXT"
580 PRINT# 1;TEXT
590 ASSIGN# 1 TO *
600 RETURN
610 TEST_MSUS:
620 ON ERROR GOTO 640
630 A$=CAT$ (1) @ RETURN
640 BEEP 100,200 @ BEEP 200,100
650 POP RETURN @ RETURN
```

Dadurch, daß die Testroutine im Falle eines Fehlers eine Unterprogrammebene löscht, wird beim Auftreten eines Fehlers direkt zum Hauptprogramm zurückverzweigt, ohne daß mit der Massenspeicherroutine fortgefahren wird.

C_RETURNS

Diese Anweisung löscht alle Unterprogrammrücksprungebenen. Sollten keine Unterprogramme aktiv sein, so wird die Anweisung ignoriert. Dem Programmierer wird mit dieser Anweisung die Möglichkeit gegeben, selbst tiefe Programmverschachtelungen, mit vielen Unterprogrammebenen programmgesteuert, z.B. beim Auftreten einer besonderen Bedingung, aufzuheben.

Es ist auch von Vorteil, wenn die Hauptroutine eines Programmes die Anweisung ab und zu auf gut Glück einsetzt, um von eventuell im Laufe des Programmes angesprungenen Unterprogrammen verbliebene Rücksprungadressen zu löschen.

2.3.6 Abarbeiten von in Strings enthaltenen BASIC-Ausdrücken

Normalerweise gibt es keine Möglichkeit, eine Formel bzw. überhaupt irgend einen BASIC-Ausdruck, der in einem String enthalten ist, abzuarbeiten. Bei Programmen, die mit variablen, vom Benutzer einzugebenden Formeln arbeiten, müssen die Formeln normalerweise vor dem Programmstart an einer bestimmten Stelle im Programm, meistens als Funktionen, einprogrammiert werden. Mit den im folgenden beschriebenen drei Schlüsselworten kann nun ein programmtechnischer Weg eingeschlagen werden, bei dem Formeln durch INPUT-Anweisungen oder Maskeneingabefunktionen in das Programm übernommen werden. Die Programme können dann die Strings, in denen die Formeln enthalten sind, abarbeiten lassen.

EXECUTE String

Hierbei handelt es sich um eine Anweisung, die ausschließlich im Programmmodus abgearbeitet werden kann. Das Abarbeiten von EXECUTE im Rechenmodus erzeugt einen BAD STATEMENT Error.

In dem String kann ein BASIC-Ausdruck mit einer Länge von bis zu 152 Zeichen enthalten sein. Dieser BASIC-Ausdruck wird vom EXECUTE-Statement syntaktisch überprüft und abgearbeitet. Alle im String enthaltenen Variablen müssen dem Programm bereits bekannt sein, d.h. sie müssen allocated sein, da das EXECUTE-Statement keine neuen Variablen deklariert.

Ein weiterer Punkt, der bei der Verwendung der Anweisung beachtet werden sollte, die EXECUTE-Anweisung arbeitet alle BASIC-Ausdrücke Befehle und Funktionen ab, egal ob diese programmierbar sind oder nicht. Folgende Anweisungen, die bei der Verwendung mit EXECUTE zum Absturz des Systems führen können, sollten daher nicht verwendet werden:

SCRATCH
LOAD
STORE
DELETE
SAVE
GET
IF ... THEN

Der Verzicht auf diese Anweisungen wirkt sich allerdings kaum schmerzlich aus, da das EXECUTE-Statement ohnehin nicht für derartige Anwendungen gedacht ist.

Das große Anwendungsgebiet der EXECUTE Anweisung ist überall dort zu suchen, wo während des Programmlaufes Formeln eingegeben werden sollen, die das Programm abarbeitet oder wo neben dem Programmlauf noch Zwischenrechnungen eingegeben werden sollen. Um nur einige Beispiele zu nennen:

* Programme zur numerischen Integration
* " " Nullstellensuche
* Datenmanipulationsprogramme
* Textverarbeitungsprogramme

Beispiel: Kleines Programm zur Demonstration der EXECUTE-Anweisung

```
10 DIM A$[50]
20 X=0
30 A$="FOR X=1 TO 20 @ BEEP X,X @ NEXT X"
40 EXECUTE A$
50 A$="SIN (15)*TAN (60)+LOG (10)*45"
60 EXECUTE "X="&A$
70 DISP X
80 END
```

Die Notwendigkeit der Zeile 20 ergibt sich aus dem Zustand, daß die EXECUTE-Anweisung keine neuen Variablen anlegt, sie kann nur auf bereits vorhandene Variablen, wie hier das vorher durch die Zeile 20 allocatete X zurückgreifen.

TOKEN$ (String)

Diese Funktion überprüft den in dem String enthaltenen BASIC-Ausdruck syntaktisch und übersetzt ihn in die maschineninterne Form. Als Resultat wird ein String ausgegeben, in dem der BASIC-Ausdruck in einer Form enthalten ist, die der Computer direkt abarbeiten kann.

Die Funktion hat ihr Anwendungsgebiet in Verbindung mit dem Statement TOKEN EXECUTE. Die TOKEN$-Funktion erzeugt Kodes, die von der TOKEN EXECUTE-Anweisung abgearbeitet werden können. Somit übernimmt die Funktion den ersten Arbeitsgang des EXECUTE-Statements, das Parsen.

Für Anweisungen, die nicht in dem String enthalten sein sollten, gelten die gleichen Regeln wie bei der EXECUTE-Anweisung.

TOKEN EXECUTE String

Die Anweisung kann einen von der TOKEN$-Funktion geparsten BASIC-Ausdruck abarbeiten. Gegenüber der EXECUTE-Anweisung führt TOKEN EXECUTE keine syntaktische Prüfung des BASIC-Kodes durch, das Statement übersetzt den BASIC-Ausdruck auch nicht in die maschineninterne Form, sondern es geht davon aus, daß der in dem String enthaltene Ausdruck in korrekter Form durch TOKEN$ erzeugt, vorliegt.

Wenn der in dem String enthaltene BASIC-Ausdruck nicht von der TOKEN$-Funktion erzeugt wurde und die TOKEN EXECUTE-Anweisung versucht, diesen String abzuarbeiten, so "hängt" sich das System mit großer Wahrscheinlichkeit auf.

Das Vorhandensein der beiden getrennten Anweisungen TOKEN$ und TOKEN EXECUTE bekommt ihre Berechtigung beim Betrachten der Abarbeitungsgeschwindigkeit:

Wenn ein EXECUTE-Statement in einer Programmschleife steht, und somit immer wiederkehrend abgearbeitet werden muß, so fällt jedes mal bei der Abarbeitung die Zeit an, die das Statement zur syntaktischen Überprüfung und Übersetzung in Tokenform benötigt.

Diese Zeit kann nun dadurch gewonnen werden, daß der BASIC-Ausdruck vor der Programmschleife mit TOKEN$ überprüft und umgewandelt und in

der Programmschleife nur noch mit TOKEN EXECUTE abgearbeitet wird. Im folgenden Beispiel findet diese Technik Anwendung.

Beispiel: Programm zur numerischen Integration nach der Simpsonschen Regel

```
10 DIM FORMEL$[200]
20 CLEAR
30 AWRITE 0,0 @ DISP "Geben Sie die Formel für das Integral F(x)=dx
ein"
40 DISP "X ist die Rechenvariable für Ihre Formel";
50 ON ERROR GOTO 30 @ INPUT FORMEL$
60 ON ERROR GOTO 90
70 X,:,F=0
80 FORMEL$=TOKEN$ ("F="&FORMEL$) @ GOTO 100
90 DISP "Sie haben einen Eingabefehler gemacht" @ BEEP @ GOTO 30
100 AWRITE 4,0,BLANK$ (50) @ DISP "Geben Sie die untere Grenze ein";
110 ON ERROR GOTO 100 @ INPUT A
120 AWRITE 6,0 @ DISP "Geben Sie die obere Grenze ein";
130 ON ERROR GOTO 120 @ INPUT B
140 AWRITE 8,0 @ DISP "Geben Sie die Anzahl der Unterteilungen ein";
150 ON ERROR GOTO 140 @ INPUT M
160 S=0 @ H=(B-A)/M @ X=A
170 GOSUB FUNKTION
180 S=F @ I=0
190 FOR X=A+H TO B-H STEP H @ I=I+1
200 GOSUB FUNKTION @ S=S+2*F
210 IF ODD (I) THEN S=S+2*F
220 NEXT X
230 GOSUB FUNKTION
240 S=S+F
250 S=H/3*S
260 DISP "F(X)=";S
270 GOTO 30
280 FUNKTION: x=X @ ON ERROR GOTO 300
290 TOKEN EXECUTE FORMEL$ @ RETURN
300 IF ERRN <>8 THEN RETURN
300 POP RETURN @ DISP "Mathematischer Fehler" @ GOTO 30
```

Zum Testen können Sie das bestimmte Integral zur Lösung von π nehmen:

$$\int_0^1 \sqrt{(1-x^2)}\,dx = \frac{\pi}{4}$$

Geben Sie nach dem Programmstart die Formel ein:

```
SQR (1-X*X)
```

Hiernach werden die Grenzen eingegeben:

```
untegre Grenze 0
obere Grenze 1
```

Geben Sie nun 100 für die Anzahl der Unterteilungen des Integrals ein.

Nach etwa 15 Sekunden zeigt Ihnen das Programm die Lösung mit einer Genauigkeit von fünf Stellen an:

```
F(X)=.785375
```

Notizen

2.3.7 Arbeiten mit variablen BASIC-Programmzeilen

In diesem Abschbitt werden Befehle beschrieben, die sich mit Variablen auf die Zeilenstruktur des Programmes beziehen.

LINE?

Mit dieser Funktion kann die Zeilennummer der Programmzeile abgefragt werden, die gerade abgearbeitet wird. Im Programmodus findet die Funktion ihre Bedeutung im Zusammenhang mit GOTOX und RESTOREX. Und im Rechenmodus nach einem Programmstopp erleichtert die Funktion die Programmkorrektur, da mit ihr leicht die aktuelle vom Programm abgearbeitete Zeile zu finden ist.

Befindet sich der Computer nicht im RUN- oder unterbrochenen RUN-Modus, so gibt die Funktion als Resultat eine 0 zurück.

GOTOX Zahl

Die Anweisung bewirkt, daß die Programmausführung an einer durch die Zahl angegebenen Zeile fortgesetzt wird. Die Zahl kann eine Konstante oder Variable sein, und hierinliegt auch der entscheidende Unterschied zwischen GOTO und GOTOX, die GOTO-Anweisung kann nur mit einer Konstanten als Zeilenangabe programmiert werden.

Beispiel: Gegenüberstellung von ON Zahl GOTO und GOTOX Zahl

Zuerst die herkömmliche Programmtechnik mit ON Zahl GOTO:

```
1000 ON A GOTO 1010,1040,1070
1010 ! Erste Routine
1020 !
1030 ! ************************************
1040 ! Zweite Routine
1050 !
1060 ! ************************************
1070 ! Dritte Routine
1080 !
1090 ! ************************************
```

Nun die etwas ausgefallenere Programmtechnik mit GOTOX Zahl:

```
1000 GOTOX LINE?+A*30-20
1010 ! Erste Routine
1020 !
1030 ! ************************************
1040 ! Zweite Routine
1050 !
1060 ! ************************************
1070 ! Dritte Routine
1080 !
1090 ! ************************************
```

Anhand dieses einfachen Beispieles ist leicht zu sehen, welche Programmierarbeit bei umfangreichen variablen GOTO-Befehlen gespart werden kann. Nicht zuletzt ist auch die Speicherplatzersparnis immens,

da für jede Zeilennummer im ON ... GOTO-Statement vier Bytes belegt werden.

Bei Verwendung von GOTOX ist darauf zu achten, daß der betroffene Programmbereich nicht mit REN oder RENUM in einer Form umnumeriert wird, die die GOTOX-Anweisung falsch abarbeitet. Bei dem umseitig aufgeführten GOTOX-Beispiel können Sie den Programmbereich beliebig umnumerieren, solange der Zeilenabstand von 10 und der Routinenabstand von jeweils 30 Zeilen beibehalten wird.

Wenn die mit der Variablen angegebene Zeile nicht existiert, so sorgt GOTOX dafür, daß zur der Programmzeile verzweigt wird, die auf die nichtexistente Zeile folgt.

RESTOREX Zahl

Die Anweisung setzt den DATA-Pointer auf die mit der Zahl angegebene Programmzeile. Gegenüber dem Standard-BASIC-Ausdruck RESTORE kann beim RESTOREX eine Variable als Zeilennummer angegeben werden.

In Verbindung mit der LINE?-Funktion können mit dieser Anweisung große DATA-Felder sehr leicht selektiv ausgelesen werden. Dies ist mit der Standard-RESTORE-Anweisung nicht möglich, da mit dieser Anweisung der DATA-Pointer nur auf eine konstante Zeile gesetzt werden kann, und das Programm sich von dort an die zu selektierende Zeile "heranarbeiten" muß.

Beispiel: Selektives Auslesen von DATA-Feldern

Herkömmliche Methode mit der RESTORE-Anweisung:

```
500 ! Die Variable A enthält die relative Position der zu
510 ! selektierenden DATA-Zeile
520 RESTORE 600
530 FOR I=2 TO A
540 READ A$
550 NEXT I
560 ! Die DATA-Zeile ist nun selektiert, und das Daten-
570 ! element kann mit der nächsten READ-Anweisung einge-
580 ! lesen werden.
600 DATA ELEMENT#1
610 DATA ELEMENT#2
620 DATA ELEMENT#3
630 DATA ELEMENT#4
```

Neue Methode mit der RESTOREX-Anweisung:

```
500 ! Die Variable A enthält die relative Position der zu
510 ! selektierenden DATA-Zeile
520 RESTOREX 600+(A-1)*10
560 ! Die DATA-Zeile ist nun selektiert, und das Daten-
570 ! element kann mit READ ausgelesen werden
600 DATA ELEMENT#1
610 DATA ELEMENT#2
620 DATA ELEMENT#3
630 DATA ELEMENT#4
```

Sie sehen die Handhabung der RESTOREX-Anweisung ist sehr einfach. Die Anweisung bewährt sich besonders dort, wo unterschiedlich lange komplette Datensätze in DATA-Feldern definiert sind. Um ein Beispiel zu nennen, graphische Symbole sind in DATA-Feldern untergebracht, dabei belegt jedes Symbol maximal 2 Programmzeilen, es ist jedoch nicht definiert, wie viele Einträge in den beiden DATA-Zeilen vorhanden sind. Dieses Problem kann mit der RESTOREX-Anweisung sehr einfach gelöst werden. Das Einzige was benötigt wird, ist ein Kennungszeichen, welches die einzelnen Datensätze voneinander trennt. Wenn z.B. davon ausgegangen wird, daß der erste Datensatz in der Zeile 600 anfängt, und der Zeilenabstand 20 beträgt, so kann der DATA-Pointer mit folgender BASIC-Anweisung auf die richtige Zeile gesetzt werden:

```
RESTOREX (EINTRAG*20+580)
```

EINTRAG ist hierbei eine Variable, die mit 1 anfangend angibt, welcher Datensatz selektiert werden soll. Nachdem das RESTOREX durchgeführt wurde, kann das Pogramm mit READ die einzelnen Elemente einlesen, bis es am Markierungszeichen das Ende des Datensatzes erkennt. Ein sinnvolles Markierungszeichen ist zum Beispiel eine Zahl oder ein Zeichen, welches in dem Datensatz nicht vorkommt. Wenn der Datensatz nur Zahlen enthält, so können Sie als Markierungszeichen auch einen Buchstaben verwenden, denn in diesem Fall erzeugt das System beim Einlesen des Buchstabens in eine numerische Variable einen Error, und diesen Error können Sie leicht mit einer ON ERROR-Anweisung abfangen.

Beispiel: Selektieren und einlesen eines Datensatzes mit ON ERROR

```
10 DIM A(100)
20 EINTRAG=2
30 GOSUB LESE_DATEN
40 !
50 ! Hier folgt Ihre weitere Routine
60 !
500 LESE_DATEN: RESTOREX (EINTRAG*20+580)
510 ON ERROR GOTO 550
520 FOR I=0 TO INF
530 READ A(I)
540 NEXT I
550 ANZAHL_ELEMENTE=I-1
560 RETURN
570 ! Nach dem Rücksprung enthält das Datenfeld A die Ele-
580 ! mente und die Variable ANZAHL_ELEMENTE enthält die
590 ! Anzahl der Elemente
595 ! Casino Baden-Baden 24.06.1962
600 DATA 35,29,6,20,26,15,9,25,21,28,7,21,3,0,36,1,6,35,5
610 DATA 23,21,21,15,17,27,25,34,7,2,18,36,26,11,17,E
615 ! 25.06.1962
620 DATA 29,27,1,20,15,19,24,6,23,26,18,8,1,4,31,18,21,0
630 DATA 23,34,25,3,6,21,3,1,13,7,16,25,18,1,12,23,13,E
635 ! 26.06.1962
640 DATA 34,35,14,12,27,17,2,24,7,21,26,3,26,7,27,13,8,19
650 DATA 19,15,15,12,14,8,8,15,10,18,22,20,4,3,19,10,E
```

Ein bei RESTOREX zu beachtender Punkt ist das Setzten des POINTERS auf nicht existente DATA-Zeilen. Wird bei RESTOREX eine nicht existierende Programmzeile angegeben, wobei die folgende Zeile jedoch eine DATA-Zeile ist, so wird der DATA-Pointer auf diese Folgezeile gesetzt. Enthält die Folgezeile keine DATA-Anweisung, so erzeugt RESTOREX eine NO DATA Fehlermeldung.

2.3.8 Finden von Zeilenmarken in BASIC-Programmen

Die Advanced-BASIC-Möglichkeiten des HP86/87 lassen die Definierung von Zeilenmarken zu, zu denen das Programm mit GOTO und GOSUB in der gleichen Weise verzweigen kann, wie zu Zeilennummern. Die Anwendung von Zeilenmarken ist auch äußerst sinnvoll, da die Selbstdokumentation dadurch erheblich steigt. Zum Beispiel ist die Programmzeile

```
1000 GOSUB Druckroutine
```

erheblich verständlicher als die Programmzeile:

```
1000 GOSUB 5760
```

Aber so sinnvoll der Einsatz von Zeilenmarken auch ist, ein Manko weist ihre Anwendung schon auf: Einmal eingegebene Zeilenmarken lassen sich in großen Programmen nur mit hohem Zeitaufwand wiederfinden. Selbst bei der Verwendung der Suchbefehle SCAN (AP-ROM) oder FREFS (ASSM-ROM) können u.U. bis zu 2 Minuten vergehen, bis die Zeile gefunden wird, in der die Zeilenmarke steht. Die beiden folgenden Befehle verkürzen die Suchzeit erheblich.

BFLABEL String

Die Anweisung sucht die mit dem String angegebene Zeilenmarke. Bei der Angabe der Zeilenmarke in dem String ist der Doppelpunkt wegzulassen, d.h. die Eingabe ist z.B. in folgender Form vorzunehmen:

```
BFLABEL "Druckroutine"
```

Wenn die angegebene Zeilenmarke im Programm vorhanden ist, so wird die Zeile, in der die Marke steht, auf den Bildschirm gelistet, und der LIST-Pointer wird auf die Zeile gesetzt. Nachdem BFLABEL seine Arbeit beendet hat, wird die Meldung READY auf dem Bildschirm ausgegeben. Wenn keine Programmzeile sondern nur READY auf dem Bildschirm ausgegeben wird, so ist die Zeilenmarke entweder nicht vorhanden oder der Bildschirm ist nicht als CRT IS-Gerät deklariert.

Wurde die Zeilenmarke gefunden und nach der Ausgabe der READY-Meldung die Taste [LIST] gedrückt, so wird das Programm ab der Zeilenmarke auf den Bildschirm gelistet.

Selbst bei sehr großen Programmen findet BFLABEL eine Marke in weniger als einer Sekunde.

LIST LABELS

Die Anweisung listet alle Programmzeilen, an deren Anfang eine Zeilenmarke steht, auf den Bildschirm bzw. auf das mit CRT IS deklarierte Device.

Nachdem LIST LABELS das Programm durchsucht und alle Zeilenmarken ausgegeben hat, wird READY auf dem Bildschirm ausgegeben, und der LIST-Pointer steht auf der letzten gefundenen Zeilenmarke.

Das Listen der Zeilenmarken kann durch Drücken einer beliebigen Taste

abgebrochen werden, auch in diesem Fall steht der LIST-Zeiger auf der zuletzt angezeigten Programmzeile, so daß das Drücken der [LIST]-Taste das Programm ab dieser Zeile auf den Bildschirm listet. Auf diesem Wege steht Ihnen eine elegante Möglichkeit zur Verfügung eine Zeilenmarke zu finden, deren exakte Schreibweise Sie nicht (mehr) wissen. Sie warten, bis LIST LABELS die gesuchte Zeile anzeigt und unterbrechen LIST LABELS dann durch Drücken der [LIST]-Taste. Das Resultat ist, daß das Programm ab der gesuchten Zeile auf dem Bildschirm gelistet wird.

Die Anweisung LIST LABELS arbeitet mit solch hoher Geschwindigkeit, daß sie kaum vom herkömmlichen LIST zu unterscheiden ist.

2.3.9 Druckergerechtes Konvertieren von Zeichenketten

Besonders bei HP-artfremden Druckern gibt es beim Drucken immer wieder Schwierigkeiten möglichst schnell die HP-Sonderzeichen (z.B. Umlaute) gegen die entsprechenden Sonderzeichen der Druckerhersteller zu tauschen. Die im Folgenden aufgeführten Befehle übernehmen diese Arbeit.

Das Anwendungsgebiet der Befehle beschränkt sich jedoch nicht nur auf die Konvertierung von Druckerzeichen. Die Befehle können überall dort eingesetzt werden, wo Zeichen in einem String gegen andere Zeichen ausgetauscht werden sollen.

```
SET REPLACE$S String#1,String#2
```

Mit dieser Anweisung kann eingestellt werden, welche Zeichen die Anweisung RPL$ (↑) tauscht.

Im ersten String müssen die Zeichen stehen, die von RPL$ gegen die Zeichen im zweiten String getauscht werden sollen. Beide Strings müssen die gleiche Länge haben, sonst generiert die Anweisung einen Er-Error. Die mit SET REPLACE$S gesetzten Zeichen bleiben so lange erhalten, bis entweder eine neue SET REPLACE$S-Anweisung abgearbeitet oder der Computer ausgeschaltet wird.

Die beiden Tauschstrings dürfen eine Länge von jeweils maximal 20 Zeichen haben.

Beim Einschalten des Computers sorgt das ROM dafür, daß bereits Standarttauschzeichen gesetzt werden. Bei diesen Standartzeichen handelt es sich um die Konvertierung von HP-Umlauten in EPSON-Umlaute entsprechend folgender Tabelle:

HP-Zeichen		EPSON-Zeichen
5	ß	126
21	Ä	91
22	ä	123
23	ö	92
24	ö	124
25	Ü	93
26	ü	125

```
REPLACE$S? String#1,String#2
```

Das Statement dient zum Abfragen der mit SET REPLACE$S gesetzten Zeichenketten. Die beiden Strings müssen auf eine Länge von jeweils mindestens 20 Zeichen vordimensioniert sein. Wenn im Laufe eines Programmes die RPL$-Funktion für mehrere verschiedene Tauschvorgänge benötigt wird, so können vor einer Abarbeitung des SET REPLACE$S-Statements mit REPLACE$S? die beiden Tauschstrings abgefragt und nach der Abarbeitung von SET REPLACE$S und RPL$ wieder gesetzt werden.

RPL$ (String)

Die Funktion tauscht die mit SET REPLACE$S gesetzten Zeichen. Wenn die Anweisung bei jedem PRINT-Statement "eingebaut" wird, so lassen sich damit alle Zeichenkonvertierungsprobleme lösen:

```
PRINT RPL$ ("String mit oder ohne Umlaute wie äöüßÄÖÜ")
```

2.3.10 Sortieren von Zeichenketten

```
SORT String,Teilstringlänge,Von,Bis
```

Das Statement sortiert den Inhalt des Strings entsprechend den Steuerparametern. Die einzelnen Steuerparameter haben folgende Bedeutung:

Teilstringlänge: Hierbei handelt es sich um die Länge der zu sortierenden Teilstrings, d.h. die Länge eines Eintrages unabhängig davon, ob der gesammte Eintrag als Sortierkriterium herangezogen wird oder nicht.

Von: Mit dieser numerischen Variablen wird festgelegt, ab welcher relativen Zeichenposition die Zeichen der Teilstrings als Sortierkriterium herangezogen werden. Wird z.B. eine 1 angegeben, so wird jeweils ab dem ersten Zeichen eines Teilstrings sortiert.

Bis: Diese Variable legt das relative Ende des Teilstringbereiches fest, der als Sortierkriterium herangezogen wird. Dieser Parameter darf aus wohl verständlichen Gründen nicht kleiner als der Bis-Parameter und nicht größer als der Teilstringlänge-Parameter sein.

Beispiel: Sortieren einer Namensdatei nach verschiedenen Kriterien

Als erstes wird ein String mit Daten geladen:

```
DIM A$[100]
A$[1,25]="#001#Erwin       Klabuster"
A$[26,50]="#002#Snoopy      Peanut    "
A$[51,75]="#003#Ernst       Riebesehl"
A$[76,100]="#004#Paul        Gathmann "
```

Der A$ enthält nun fünf Namen mit den zugehörigen Ordnungsnummern. Jeder Teilstring hat eine Länge von 25 Zeichen.

Der A$ soll nun nach Vornamen sortiert ausgegeben werden:

```
SORT A$,25,6,16
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Anzeige:

```
#003#Ernst      Riebesehl
#001#Erwin      Klabuster
#004#Paul       Gathmann
#002#Snoopy     Peanut
```

Nun soll der A$ nach den Nachnamen sortiert ausgegeben werden:

```
SORT A$,25,17,25
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Anzeige:

```
#004#Paul       Gathmann
#001#Erwin      Klabuster
#002#Snoopy     Peanut
#003#Ernst      Riebesehl
```

Der String soll nun wieder in seinen Urzustand, nach Zahlen sortiert, versetzt werden:

```
SORT A$,25,6,16
FOR I=1 TO 75 STEP 25 @ DISP A$[I,I+24] @ NEXT I
```

Anzeige:

```
#001#Erwin      Klabuster
#002#Snoopy     Peanut
#003#Ernst      Riebesehl
#004#Paul       Gathmann
```

Die Leistungsfähigkeit und Anwendung des SORT-Statements dürfte hiermit wohl ausreichend erklärt sein. Sie sehen, daß auch Zahlen sortiert werden können, da ihre logische Ziffernfolge der ASCII-Zeichenfolge entspricht.

```
UPSORT String,Teilstringlänge,Von,Bis
```

Kleine und große Buchstaben haben eine unterschiedliche ASCII-Kodierung, das kleine Alphabet beginnt mit dem Zeichen 97 und das Große Alphabet mit dem Zeichen 65. Aus dieser Gegebenheit resultiert ein Mißzustand, und zwar folgt bei der Sortierung mit SORT das "a" auf das "Z" oder anders ausgedrückt "adaptieren" steht nicht hinter Adapter sondern hinter "Zylinderstift". Daraus folgt, daß bei einer Sortierung von Klein- und Großbuchstaben sehr eigenartige Ergebnisse zustande kommen können. UPSORT umgeht diesen Zustand, da die Groß- und die Kleinbuchstaben gleichberechtigt behandelt werden. Die Parameterbedeutung bei UPSORT ist die gleiche wie die bei SORT.

2.4 Advanced-BASIC-Programmstrukturen

Die Programmstrukturen die sich mit dem HP-BASIC realisieren lassen, sind mit GOTO, GOSUB, ON GOTO, ON GOSUB, ON ERROR, FOR ... NEXT usw. recht umfangreich. Mitunter ist jedoch eine Situation gegeben, bei der insbesondere bei Schleifenkonstruktionen die Anwendung von FOR ... NEXT nicht besonders elegant ist. Die folgenden Programmstrukturanweisungen des SYSEXT-ROMs ermöglichen es hier, neue Wege zu gehen.

2.4.1 Programmschleifen

```
WHILE logischer Ausdruck DO
WHILE numerischer Ausdruck DO
```

Mit dieser Anweisung wird der Start einer Programmschleife definiert, die vom Programm so oft durchlaufen wird, bis der logische oder numerische Ausdruck unwahr ist.

Die WHILE ... DO Anweisung muß als erstes Statement in einer Programmzeile stehen, und hinter dem Statement darf abgesehen von einem durch "!" oder "REM" gekennzeichneten Kommentar keine weitere Anweisung folgen. Diese Einschränkungen sollen bewirken, daß die durch WHILE ... DO ermöglichte strukturierte Programmierung nicht durch unübersichtliche Programmgestaltung zerstört wird.

Eine WHILE ... DO Schleife wird durch ein

```
END WHILE
```

an erster Stelle in einer der folgenden Programmzeilen abgeschlossen.

Der Computer erkennt automatisch, welches END WHILE zu welchem WHILE ... DO gehört.

Es dürfen maximal 15 WHILE ... DO ... END WHILE Schleifen ineinander verschachtelt werden. Der Computer "merkt" sich dabei automatisch, welche END WHILE Anweisung zu welchem WHILE ... DO Statement gehört.

Eine WHILE ... DO Schleife sollte nicht einfach mit GOTO verlassen werden, da die Schleife trotzdem aktiv bleibt. In diesem Fall stapeln sich dann ähnlich wie beim durch GOTO Verlassenen Unterprogramm die Verschachtelungsebenen auf einem Stapel, bis dieser überläuft und das SYSEXT-ROM einen "WHILE Nesting"-Error erzeugt.

Wenn eine WHILE ... DO Schleife verlassen werden soll, so ist das am besten mit den dafür vorgesehenen Anweisungen durchzuführen.

```
EXIT WHILE
```

Diese Anweisung dient zum vorzeitigen Verlassen einer WHILE ... DO Schleife. Die Anweisung löscht die momentan aktive Schleife und verzweigt zu der, auf das, zu dieser WHILE ... DO Ebene gehörende END WHILE folgende Statement.

POP WHILE

Mit dieser Anweisung kann die momentan aktive WHILE ... DO Ebene gelöscht werden. Die Programmschleife kann dann durch ein GOTO verlassen werden.

Die Anweisung ermöglicht es dem Programmierer insbesondere nach einer IF Abfrage eine logische Progammstruktur zu verlassen, die an sich ähnlich einem Unterprogramm immer vom Anfang bis zum Ende abgearbeitet werden sollte.

REPEAT

Die REPEAT Anweisung stellt den Anfang einer nicht abweisenden Programmschleife dar. Nicht abweisend deshalb, weil eine REPEAT Schleife mindestens einmal durchlaufen wird, bevor ein Test durchgeführt wird, von dessen Ausgang es abhängt, ob die Schleife ein weiteres Mal durchgeführt wird oder nicht.

Die REPEAT Schleife wird mit einer Testanweisung abgeschlossen, die folgenden Aufbau hat:

```
UNTIL logischer Ausdruck
UNTIL numerischer Ausdruck
```

Die REPEAT ... UNTIL Schleife wird so oft wiederholt, bis der logische oder numerische Ausdruck wahr ist. Damit ergibt sich im Vergleich mit der WHILE ... DO Schleife eine leichte logische und strukturelle Veränderung.

Es können bis zu 15 REPEAT ... UNTIL Ebenen ineinander verschachtelt werden. Wird versucht, eine sechzehnte EBENE anzusprechen, so erzeugt das ROM einen "REPEAT NESTING"-Error.

Beispiel: Zerlegung einer Zahl in Ziffern

```
10 DISP "Wie lautet die Zahl";
20 INPUT ZAHL
30 REPEAT
40 DISP ZAHL MOD 10;
50 ZAHL=ZAHL DIV 10
60 UNTIL NOT ZAHL
70 DISP
80 END
```

POP UNTIL

Die Anweisung dient zum Löschen der momentan aktiven REPEAT ... UNTIL Schleife. Die Schleife kann dann, ohne daß die logische UNTIL Bedingung erfüllt wird, mit einem GOTO verlassen werden.

Beispiel: Berechnen des kleinsten gemeinsamen vielfachen zweier Zahlen unter Benutzung von REPEAT ... UNTIL und WHILE ... DO

```
10 DISP "Wie lauten die beiden Zahlen";
20 INPUT X,Y
30 A=X @ B=Y
40 REPEAT
50 WHILE X>Y DO
60 X=X-Y
70 END WHILE
80 WHILE Y>X DO
90 Y=Y-X
100 END WHILE
110 UNTIL X=Y
120 DISP "kgV=";A DIV X*B DIV X*X
130 END
```

LOOP

Das Statement dient zur Definierung einer Endlosschleife, die mit

END LOOP

abgeschlossen wird. Die zwischen LOOP und END LOOP stehenden Anweisungen werden so oft wiederholt, bis die LOOP mit einem GOTO verlassen wird.

Es besteht nicht die Möglichkeit, mehrere LOOPs ineinander zu verschachteln. Jedes erneute LOOP Statement überschreibt das Vorhergehende.

Das Anwendungsgebiet der LOOPs liegt überall dort, wo schnell arbeitende Endlosschleifen benötigt werden. Eine Endlosschleife ohne LOOP hat zum Beispiel folgendes Aussehen:

```
100 FOR I=1 TO INF
110 !
120 ! Programmaktion
130 !
140 NEXT I
```

Bei Verwendung von LOOP sieht das Programm dann so aus:

```
100 LOOP
110 !
120 ! Programmaktion
130 !
140 END LOOP
```

Der entscheidende Vorteil von LOOP liegt nun darin, daß die Verzweigung von END LOOP zu LOOP etwa doppelt so schnell abläuft, wie die Verzweigung von NEXT zu FOR. Diese Zeitersparnis ist besonders dann, wenn die LOOP sehr oft durchlaufen wird, sehr von Vorteil.

2.4.2 Vergleichsstrukturen

Bei der Anwendung der IF ... THEN ... ELSE Anweisung tritt mitunter das Problem auf, daß der auf IF oder auf ELSE folgende Ausdruck nicht in eine Zeile paßt. Mit der folgenden Bedingugsanweisung wird dieses Manko umgangen.

```
BLIF logischer Ausdruck BLTHEN
BLIF numerischer Ausdruck BLTHEN
```

BLIF definiert den Anfang eines bedingt abzuarbeitenden Programmblokkes. BLIF muß am Anfang einer Programmzeile stehen und BLTHEN darf kein weiteres Statement folgen.

Die auf BLTHEN folgenden Programmzeilen werden dann abgearbeitet, wenn der logische oder numerische Ausdruck wahr oder ungleich null ist.

Ein BLELSE oder END BLIF definiert das Ende der Programmzeilen, die bei einer wahren Bedingung abgearbeitet werden.

```
BLELSE
```

BLELSE ist in der BLIF Struktur eine optionale Anweisung. Ist sie in der Struktur vorhanden, so muß sie am Anfang einer Zeile stehen und hinter BLELSE dürfen keine weiteren Statements folgen.

Die auf BLELSE folgenden Programmzeilen werden dann abgearbeitet, wenn der zwischen BLIF und BLTHEN stehende Ausdruck unwahr oder null ist.

```
END BLIF
```

Das END BLIF muß am Anfang einer Programmzeile stehen. Es definiert das Ende eines BLIF-Blockes.

Es ist nicht erlaubt, mehrere BLIF ... BLTHEN Strukturen ineinander zu verschachteln. Folgende Bedingungs- und Strukturanweisungen dürfen in einer BLIF ... BLTHEN Struktur enthalten sein:

```
FOR ... NEXT
WHILE ... DO ... END WHILE
REPEAT ... UNTIL ...
IF ... THEN ... ELSE
LOOP ... END LOOP
```

```
EXIT BLIF
```

Die Anweisung dient zum vorzeitigen Verlassen einer BLIF ... BLTHEN Struktur. Bei der Durchführung von EXIT BLIF verzweigt das Programm zu der auf END BLIF folgenden Anweisung. Wird EXIT BLIF außerhalb einer BLIF ... BLTHEN Struktur durchgeführt, so wird ein Error erzeugt.

Beispiel: Die Anwendung der BLIF ... BLTHEN Struktur

```
100 BLIF Erster_Durchgang BLTHEN
110 Flagge=0
120 DISP "Bitte Kommando eingeben";
130 INPUT A$
140 IF UPC$ (A$)="KEIN KOMMANDO" THEN EXIT BLIF
150 IF A$="" THEN DISP "Eingabe ist notwendig" @ GOTO 120
160 GOSUB Syntaxcheck
170 BLELSE
180 IF NOT Flagge THEN EXIT BLIF
190 GOSUB Haupttest
200 END BLIF
```

Notizen

2.5 Assembler und Maschinensprache unterstützende Anweisungen

Die in den folgenden Kapiteln aufgeführten Statements und Funktionen sind insbesondere für den Anwender gedacht, der mit der Maschinensprache der Serie 80 vertraut sind. Es ist unmöglich, daß das Handbuch detailiert auf alle Möglichkeiten eingeht, die die folgenden Statements und Funktionen bieten. Es sei daher auf die mitgelieferte Übersetzung des Assemblerhandbuches verwiesen die dem Benutzer in vielen Kapiteln ausführlich in die Maschinensprache einweist und damit auch die Möglichkeiten der Verwendung des SYSEXT-ROMs aufzeigt.

Es sei hier noch gesagt, daß die Fantasie, das Können und die Ausdauer des Anwenders sehr viel mehr als jedes Handbuch wert ist. Das soll heißen, die folgenden Kapitel sowie das Assemblerhandbuch sollen nur als Anregung dienen. Fleißiges Experimentieren mit den maschinennahen Statements und Funktionen zeigt mit Sicherheit früher oder später einen Erfolg. Diverse, mit Sicherheit vorkommende Systemabstürze sollten nicht dazu führen, daß Sie das Handtuch werfen.

2.5.1 Abfragen von Speicherinhalten und Adressen

```
PEEK (Adresse)
```

Die Funktion gibt den Inhalt der, mit der Adresse angegebenen Speicherstelle zurück. Die Adresse muß oktal angegeben und im Bereich zwischen 0 und 177777 liegen. Das von der Funktion zurückgegebene Ergebnis ist eine Zahl im Bereich zwischen 0 und 377.

Das besondere Anwendungsgebiet der Funktion liegt in der Abfrage von ein-Byte-Zeigern.

Beispiel: Überprüfen, ob sich der Bildschirm im Alpha- oder im Graphikmodus befindet. Dazu braucht nur das an der Adresse CRTSTS stehende Byte mit einer Maske abgefragt werden.

```
NUM (AND$ (CHR$ (O_D (PEEK (177702))),CHR$ (128)))
```

Ist das Resultat 0, so ist der Alpha-Bildschirm angeschaltet, ist es 128, so ist der Graphikbildschirm an.

Beispiel: Überprüfen, ob Pagesize 24 oder Pagesize 16 vorliegt. Auch diese Information steht an der Adresse CRTSTS.

```
NUM (AND$ (CHR$ (O_D (PEEK (177702))),CHR$ (8)))
```

Ist das Resultat eine 0, so liegt Pagesize 16 vor, ist es eine 8, so liegt Pagesize 24 vor.

Die Pagesize-Abfrage ist übrigens eine elegante Methode, um von einer Autostartroutine feststellen zu lassen, ob es sich bei dem Computer um einen 86A/87/87XM oder einen 86B handelt, den der HP86B wacht im Gegensatz zu den anderen Computern im Pagesize-24-Format auf.

Beispiel: Überprüfen, in welchem Winkelmodus sich der Computer befindet

```
PEEK (100160)
```

Im RAD-Modus ist das Ergebnis 0, im DEG-Modus 220 und im GRAD-Modus 231.

Weiter Anwendungsmöglichekeiten der PEEK-Funktion finden Sie im Kapitel 8 des Assemblerhandbuches.

PEEK$ (Adressen,ROM#,Anzahl)

Die Funktion dient zum Auslesen des Speichers ab der angegebenen Adresse. Die Funktion gibt einen String mit der Länge Anzahl zurück. Die Adresse ist oktal anzugeben. Die ROM# ist nur im Adressbereich zwischen 60000 und 77777 von Belang.

Beispiel: Länge des BASIC-Programmes berechnen. Dazu braucht nur der Inhalt der Adresse BOVAR (Ende des Programmes) vom Inhalt der Adresse FWCURR (Anfang des Programmes) subtrahiert werden

```
ADR (PEEK$ (100006,0,3))-ADR (PEEK$ (100014,0,3))
```

Beispiel: Adresse des momentan eingestellten Massenspeichers finden

```
MS$=PEEK$ (103500,0,3) ! Der String braucht jetzt nur noch
                       ! dekodiert werden
MS$=":D"&VAL$ (NUM (MS$)+3)&VAL$ (NUM (MS$[2]))
MS$=MS$&VAL$ (NUM (MS$[3]))
```

Beispiel: Name des momentan aktive Volume-Labels finden

```
PEEK$ (103527,0,6)
```

EMC PEEK$ (Adresse,Anzahl)

Diese Funktion erlaubt das Auslesen von Speicherbereichen, die eine drei-Byte-Adresse haben, also über 177777 oktal liegen. In diesen Speicherbereichen liegen die BASIC-Programme und die Variablen. Die Adressangabe in der Funktion hat oktal zu erfolgen. Die Variable Anzahl gibt an, wie viele Bytes die Funktion zurückgeben soll.

SADR (String)

Die Funktion gibt die dezimale Startadresse des Strings zurück. Der Anwender kann, nachdem er mit dieser Funktion die Startadresse ermitermittelt hat, direkt auf die Speicherstellen zugreifen, und dort Manipulationen vornehmen.

2.5.2 Manipulieren und neu Setzen von Adressen und Speicherinhalten

```
POKE Adresse,Inhalt
```

An der oktal angegebenen Adresse wird das oktal angegebene Byte Inhalt abgelegt. Bei der Adresse kann es sich um eine zwei-Byte-Adresse im Bereich zwischen 100000 und 177777 handeln. Als Inhalt kann ein Byte mit der Wertigkeit 0 bis 377 angegeben werden.

Beispiel: Umschalten des Bildschirmmodus von Inversvideo auf Normalvideo und umgekehrt

```
POKE 177702,D_0 (NUM (XOR$ (PEEK$ (177702,0,1)," ")))
```

Beispiel: Blinken der RUN-LED ein/ausschalten

```
POKE 177704,1                    ! LED blinkt
POKE 177704,0                    ! LED leuchtet
```

Beispiel: Einstellen der Dauer, bevor nach einem Tastendruck die Taste wiederholt wird

```
POKE 100154,N                    ! N ist der Warteparameter
                                 ! Grundeinstellung ist 34
```

Beispiel: Einstellen der Tastenwiederholungsgeschwindigkeit

```
POKE 100155,N                    ! N ist der Speedparameter
                                 ! Grundeinstellung ist 2
```

```
EMC POKE$ Adresse,String
```

Der Inhalt des Strings wird in aufsteigender Reihenfolge an der oktal angegebenen Adresse abgelegt. Als Adressbereich kommen alle zwei- und drei-Byte-Adressen zwischen 100000 und 2000000 in Frage

Die Anwendungsmöglichkeiten der Anweisung erstrecken sich über viele Gebiete. Zum Beispiel können Manipulationen in BASIC-Programmen programmgesteuert dadurch vorgenommen werden, daß direkt in den Programmspeicher hineingepoked wird, d.h. man kann mit gewissen Arbeitsaufwand Programme schreiben, die sich selbst programmieren. Des weiteren kann direkt auf die von der electronic disc erzeugten Programm oder Datenfiles zugegriffen werden. Und zuletzt sei hier noch auf die Möglichkeit hingewiesen, die POKE-Anweisung in Verbindung mit der Funktion SADR dazu zu benutzen, Strings direkt, unter Umgehung des Zuweisungszeichens "=", zu manipulieren. Bei der direkten Stringmanipulation unter Benutzung von POKE können bis zu 50% der Zeit eingespart werden.

SETPTR2 Adresse,Wert

Die Anweisung dient zum Setzen der zwei-Byte-Zeiger im Computerspeicher. Die Adresse sowie der Wert müssen oktal angegeben werden, wobei die Adresse im Bereich zwischen 100000 und 177777 und der Wert im Bereich zwischen 0 und 177777 liegen müssen.

Ein typischer zwei-Byte-Zeiger ist CRTBAD, ein CRT-Kontrollerregister.

Beispiel: Setzen des Kursors auf die 10'te Reihe und 11'te Spalte

```
SETPTR2 177701,D_O (10*80-80+11-1) ! 177701 = CRTBAD
```

Wenn hiernach z.B. AREAD durchgeführt wird, so wird der Bildschirm ab der neuen Kursorposition ausgelesen.

Wenn direkt nach dem Setzen des Kursors mit SETPTR2 eine beliebige Taste gedrückt wird, so scheint es, als ob sich der Kursor überhaupt nicht bewegt hat, da die Reaktion des Bildschirmes auf den Tastendruck genau an der Stelle geschieht, an der vorher der Kursor stand. Das liegt jedoch daran, daß das Betriebssystem eine Kopie der Kursoradresse im Speicher hat, auf die es zurückgreift. Das Setzen des Kursors mit SETPTR2 ist also nur im RUN-Modus mit darauf folgender Aktion wie AREAD oder dergleichen sinnvoll.

SETPTR3 Adresse,Wert

Mit dieser Anweisung können die drei-Byte-Zeiger im Computerspeicher gesetzt werden. Die Adresse sowie der Wert werden oktal angegeben. Auch hier liegt die Adresse im Bereich zwischen 100000 und 177777. Der Wert jedoch kann eine Breite von drei Byte haben, d.h. er liegt zwischen 0 und 77777777.

Typische drei-Byte-Zeiger sind die extendet Memory-Kontroller PTR1 und PTR2, wobei hier ein besonderes Augenmerk auf den Zeiger PTR1 gerichtet werden sollte. Eine Speicherung zu PTR1 bewirkt, daß der BASIC-Programmzeiger zu der angegebenen Adresse verzweigt. Soll das Programm z.B. zu der Adresse 300000 verzweigen, so sieht das Statement folgendermaßen aus:

```
SETPTR3 177710,300000
```

Dem Anwender wird also die Möglichkeit gegeben, überall in das Programm, selbst mitten in eine Multistatementzeile hinein, zu verzweigen. Es können jedoch auch, wie schon beim EMC POKE$ erwähnt, Programme geschrieben werden, die sich selbst programmieren, d.h. Tokenfolgen in Strings ablegen, die dann mit SETPTR2 angesprungen werden. Besondere Achtung verdienen dabei die Tokens, die normalerweise nicht einzeln im Programm auftauchen, also nur in Verbindung mit anderen, vorangestellten Tokens anzutreffen sind. Um hier nur einige Beispiele zu nennen, das zweite Token des INPUT-Statements, diverse Massenspeichertokens des PRINT#- und READ#-Statements und die unsichtbaren, sekundären Tokens der verschiedenen ROMs insbesondere des IO-, des Matrix- und des Assembler-ROMs. Das Austesten der diversen Möglichkeiten ist ein großes Aufgabengebiet.

2.5.3 Die Kodierung der Zahlen

Die Computer der Serie 80 können bekannterweise mit drei verschiedenen dezimalen Zahlendarstellungen arbeiten. Sie können Integer-, Short- und Realzahlen handhaben. Die Vorteile der, sich von Real unterscheidenden Systeme liegen einzig im Speicherbedarf und fallen besonders bei großen Datenfeldern ins Gewicht.

In der Maschinensprache müssen oftmals Zahlen, im Real- oder im Integerformat als Konstanten geladen werden, um von Maschinenebene aus, diverse Systemroutinen aufzurufen. Jeder, der schon in Maschinensprache programmiert hat, wird die Probleme kennen, die sich ergeben, wenn komplizierte Zahlen in Kodeform in die CPU-Register geladen werden müssen. Man fängt an, die Zahl zu kodieren, beachtet Ziffernfolgen, Vorzeichen, Komplementformen und Exponenten und wenn man dann nach einigen Minuten das Maschinenspracheprogramm austestet, so stellt man fest, daß die kodierten Zahlen doch nicht denen entsprechen, die man eigentlich haben wollte.

Die folgenden Anweisungen lösen dieses Problem ein für alle Mal.

REGREP$$ (Zahl)

Diese Funktion gibt als Resultat einen String zurück, in dem in oktaler Form die Zahl gemäß den Vereinbarungen des Betriebssystemes kodiert ist. In dem, von der Funktion erzeugten String sind immer acht Zahlen enthalten, dabei stellt die erste Zahl das Highbyte und die achte Zahl das Lowbyte dar.

Beispiel: Welche Kodierung hat die Zahl INF (9.99999999999E499)?

```
REGREP$$ (INF)
231 231 231 231 231 231 100 231
 ↑                      ↑   ↑
erste/zweite            Vorzeichen und Exponent
  Ziffer
```

Beispiel: Laden der Zahl INF in die CPU-Register R40-R47

```
LDM R40,=231,100,231,231,231,231,231,231
```

Der von REGREP erzeugte String ist also in umgedrehter Reihenfolge in die CPU-Register zu laden.

Mitunter schreiben die Systemroutinen vor, welchem Zahlensystem die Parameter entsprechen müssen, die auf den Operationsstapel oder in den CPU-Registern übergeben werden müssen. Die REGREP$$-Funktion erkennt selbstständig, welchem Zahlensystem ein Parameter angehört und kodiert entsprechend eine Integer oder Realzahl. Eine Integerzahl unterscheidet sich in CPU-Schreibweise von einer Realzahl hauptsächlich dadurch, daß nur vier der acht Bytes definiert sind, die restlichen vier Bytes beeinflussen die Zahl nicht und können daher beliebig sein.

Beispiel: Laden der Integerzahl 1000 in die CPU-Register R50-R57 und laden der Realzahl 1000 in die CPU-Register R60-R67

```
REGREP$$ (1000)                    ! Integerzahl, erkennbar am
000 020 000 377 000 000 316 016    ! Byte 377. Die Bytes 0,0,
LDM R54,=377,0,20,0                ! 316,16 sind undefiniert
REGREP$$ (1000.)                   ! Der "." macht eine Real-
                                   ! zahlaus der Integerzahl.
020 000 000 000 000 000 000 003
LDM R60,=3,0,0,0,0,0,0,20
```

Dieses Beispiel zeigt deutlich, daß die REGREP$$-Funktion durch setzen eines Punktes dazu gebracht werden kann, eine Integer- in eine Realzahl zu konvertieren. Wenn eine Integerzahl in die CPU-Register geladen werden soll, so brauchen nur die Bytes 1-4 berücksichtigt werden.

Beispiel: Berechnen der Formel SIN (45)*15.89 in Maschinensprache

```
REGREP$$ (45)                      ! Darstellung von 45?
000 000 105 377 000 000 177 246
1000 LDM R44,=377,105,0,0          ! nach R44-R47, da Integer
1010 PUMD R40,+R12                 ! auf OP-Stapel
1020 JSB =SIN10                    ! Sinus berechnen lassen
REGREP$$ (15.89)                   ! Darstellung von 15.89?
025 211 000 000 000 000 000 001
1030 LDM R40,=1,0,0,0,0,0,211,25   ! nach R40-R47 laden
1040 PUMD R40,+R12                 ! auf OP-Stapel
1050 JSB =MPYROI                   ! multiplizieren lassen
1060 POMD R40,-R12                 ! Ergebnis steht in
                                   ! R40-R47
```

REGREP$ (Zahl)

Die Funktion kodiert eine Zahl in einen acht Byte langen String um und gibt diesen als Resultat zurück. REGREP$ findet dort Anwendung, wo mit SCALL (↑) Strings aufgerufen werden, in denen Maschinenkode enthalten ist. REGREP$ kann dann dazu benutzt werden, um in dem Kode enthaltene Zahlen, zum Aufruf von Systemroutinen, zu kodieren.

Beispiel: Ablegen des Maschinenkodes LDM R40,=Dezimalzahl 1.256E7

```
A$=CHR$ (96)                       ! setzen des DRP
A$=A$&CHR$ (169)                   ! LDM
A$=A$&REV$ (REGREP$ (1.256E7))     ! Zahl laden
```

REGREP (String)

Diese Funktion konvertiert einen kodierten, acht Byte langen String in eine Zahl um. Die Funktion kann z.B. in Verbindung mit PEEK$ überall dort eingesetzt werden, wo Maschinenkode wieder zurückübersetzt werden soll.

Beispiel: Disassemblieren der Zahl PI

```
REGREP (REV$ (PEEK$ (54376,0,8)))
```

2.5.4 Aufrufen von Systemroutinen und Abarbeitet von BPGM-Strings

Das Betriebssystem der HP86/87-Computer enthält sehr viele Routinen, die sich als nützlich für den Programmierer erweisen würden. Leider hat man normalerweise jedoch keinen Zugriff auf diese Routinen, da die Routinen keine zugehörigen Schlüsselwörter aufweisen, mit denen man sie sie aufrufen kann. Als Beispiel seien hier die Kursorsteuertasten [→], [←], [↓], [↑], [ROLL ▽] und [ROLL △] aufgeführt. Mit dem in diesem Kapitel erklärten Statement können alle Systemroutinen aufgerufen werden.

```
SCALL Adresse [,ROM#]
SCALL String
```

Mit SCALL Adresse [,ROM#] kann jede Maschinenspracheroutine in einem der optionalen ROMs, sowie in den fest eingebauten ROMs im Bereich zwischen 0 und 77777 aufgerufen werden. Wird die ROM# nicht angegeben, so ist wird Augenblick des Routinenaufrufs das ROM 0 selektiert. Im Folgenden sind einige wichtige Systemroutinen aufgeführt.

SCALL	ROM#	Wirkung
11606	0	[A/G]
11520	0	[BACK SPACE]
11565	0	[SHIFT] [BACK SPACE]
13671	0	[ROLL ▽]
13736	0	[ROLL △]
13651	0	[→]
13623	0	[←]
13607	0	[↓]
13562	0	[↑]
13661	0	[HOME]
13447	0	[←LINE]
14225	0	[CLEAR]
14165	0	löschen der kompletten Zeile
5407	0	[RESET]
1241	0	[INIT]
0	0	PWO, Computer aus und wieder an
12246	0	alle Bildschirmseiten löschen
14030	0	Kursor einschalten
13467	0	Kursor ausschalten
12360	0	Bildschirm einschalten
12374	0	Bildschirm ausschalten
5601	0	SCRATCH
1074	0	[RUN] Programmneustart

Im Kapitel acht des Assemblerhandbuches stehen noch weitaus mehr Routinen, die für Sie von Interesse sein könnten.

Sie können auch Maschinenkode mit EMC POKE$ an der Adresse 101145 ablegen (dort stehen die Bezeichnungen der Keylabels) und diese Adresse dann mit SCALL aufrufen.

Wenn bei dem SCALL Statement an Stelle der Adresse und der optionalen ROM# ein String angegeben wird, so setzt die Anweisung des Maschinensprache-PC auf den Anfang des Strings und der String wird abgearbeitet. Hierbei ist jedoch darauf zu achten, daß die Strings in umgedrehter Reihenfolge im Speicher stehen, d.h. das Maschinenprogramm muß am Ende des Strings beginnen und am Anfang enden. Ein weiterer Punkt, der beachtet werden muß, ist die Position des Strings. Der String darf in keinem Fall im Adressbereich > 177777 liegen, da der Prozessor nicht in der Lage ist, Programmkode abzuarbeiten, der im drei-Byte-Adressbereich liegt. Wenn der String im drei-Byte-Adressbereich liegt, so generiert das SCALL-Statement eine Error-Meldung.

Die Parameterübergabe vom BASIC-Betriebssystem zum Maschinenspracheprogramm nimmt man am besten folgendermaßen vor:

Stringparameterübergabe: Mit der SADR-Funktion wird die Position des Strings berechnet. Diese Adresse wird mit ADR$ in einen drei Byte langen String umgewandelt und der String mit EMC POKE$ an die Adresse 101145 gespeichert. Danach wird mit LEN die Länge des Strings ermittelt, diese Länge wird dann ebenfalls, in einen String umgewandelt mit EMC POKE$ zur Adresse 101150 geschrieben. Danach kann das Maschinenspracheprogramm aufgerufen werden und dieses kann sich die Stringadresse und Länge zur Handhabung von den genannten Speicherstellen laden. Die Parameterübergabe ab den Speicherstellen 101145 ist natürlich nur ein Beispiel. Die Speicherstellen scheinen jedoch deshalb recht günstig, weil dort die Bezeichnungen der Keylabels des Programmodus stehen, diese Speicherstellen sind somit für den Computer also nicht lebensnotwendig.

Übergabe numerischer Parameter: Der Parameter wird mit REGREP$ in einen String umgewandelt, und dieser acht Byte lange String wird mit EMC POKE$ ab der Adresse 101145 in den Speicher geschrieben. Hiernach wird das Maschinenspracheprogramm mit SCALL aufgerufen, welches sich daraufhin den numerischen Parameter von der Adresse LEGEND laden kann. Nach der Bearbeitung kann das Binärprogramm den neuen Parameter wieder an genannte Speicherstelle schreiben und mit RTN in den BASIC-Modus zurückspringen. Das BASIC-Programm kann sich den Parameter mit PEEK$ von der Adresse 101145 laden und mit REGREP in einen numerischen Wert umwandeln.

2.5.5 Katalogfunktionen für Binärprogramme und ROMs

Die Vielzahl der Anweisungen, Funktionen und Statements, die die Computer der Serie 80 zu bieten haben, ist wohl für jeden beeindruckend. Leider verliert man jedoch selbst als versierter Benutzer leicht den Überblick und formuliert u.U. einen Programmteil etwas umständlich, weil man im Moment nicht an die eine oder andere Funktion dachte, die das Betriebssystem zur Verfügung stellt. Fehlen gar die Kurzanleitungen des Rechners und der diversen ROMs, so hat man mitunter starke Probleme, sich zurechtzufinden. Jeder kennt wohl auch das Problem, daß man genau weiß, daß der Computer die gerade beötigte Anweisung kennt, man kann sich jedoch nicht an den Namen oder die Parameterversorgung erinnern. All diese Probleme werden mit den folgenden beiden Statements zwar nicht aus der Welt geschafft, jedoch stark vereinfacht.

```
RCAT ROM#
```

Das Statements listet den Katalog des mit der ROM# angegebenen ROMs oder EPROMs auf den Bildschirm, d.h. das Statement listet die Startadressen, die Einträge sowie die Positionen der zu den Einträgen gehörenden Parse- und Runroutinen und die Attribute in Form einer Tabelle. Alle vom RCAT-Statement angegebenen Adressen und Parameter werden im oktalen Zahlensystem ausgegeben.

RCAT 208 (Massenspeicher-ROM) bringt auszugsweise folgende Anzeige auf den Bildschirm:

```
ROM:     000320
Runtim:  060012
ASCIIS:  060215
Parse:   060130
Init:    073631
Ermsg:   073440
Tok# Runtim  Parse   Name              Attributes
-------------------------------------------------
001  065466  061237  ASSIGN#                 241
002  061414  060542  CAT                     241
003  072474  060600  CHECK READ OFF#         241
004  072433  060600  CHECK READ#             241
```

Der Kopf des Kataloges gibt oktal die ROM-Nummer, den Start der Tabelle, die die Schlüsselworte enthält, den Start der Tabelle, die die Routinenstartadressen angibt, den Start der Tabelle, die die Parseroutinen definiert, den Start der Initialisierroutine, sowie den Start der Fehlermeldungstabelle wieder.

Im Katalog werden die einzelnen Schlüsselworte mit den zugehörigen Startadressen und Attributfolgen nach aufsteigender Tokenfolge ausgegeben.

Wie die Attribute eines Schlüsselwortes zu interpretieren sind, wird im Kapitel sieben des Assemblerhandbuches genau erklärt. Hier sei nur kurz darauf hingewiesen, daß ein dreistelliges einzelnes Attribut immer darauf hinweist, daß es sich bei dem Schlüsselwort um ein Statement und nicht um eine Funktion handelt. Ist die erste der drei Ziffern eine 3, so kann das Schlüsselwort nicht hinter einem THEN eingegeben werden, ist sie eine 1, so kann das Statement nicht programmiert werden. Ist das erste zweistellige Funktionsattribut eine 55, so

handelt es sich bei dem Schlüsselwort um eine Funktion, die als Resultat einen numerischen Parameter zurückgibt. Ist es eine 56, so gibt die Funktion einen String als Resultat zurück.

Eine kurze Zusammenfassung der Attribute und ihrer Bedeutungen ist im Anhang dieses Handbuches zu finden.

Beim Betrachten der verschiedenen ROM-Kataloge werden Sie feststellen, daß in den Katalogen Schlüsselworte stehen, die in den Handbüchern gar nicht aufgeführt sind. In diesem Fall müssen Sie sich die Arbeit machen, selbst herauszufinden, welche Bedeutungen die entdeckten Schlüsselworte haben. Bei dieser Gelegenheit möchte ich Sie gleich auf das Token Nummer 36 im SYSEXT-ROM hinweisen. Das dort stehende Schlüsselwort ist in diesem Handbuch nicht aufgeführt, das ROM benötigt es jedoch in Verbindung mit den EXECUTE-Anweisungen. Sie werden in den verschiedenen ROM-Katalogen auch Dummy-Schlüsselworte finden, die nur aus dem Null-Zeichen bestehen, diese Schlüsselworte benutzen die ROMs als unsichtbare Tokens in Verbindung mit anderen Schlüsselworten zum Aufbau von Multitoken-Statements.

Falls Ihnen die zu den ROMs gehörenden Nummern nicht geläufig sein sollten, im Anhang ist eine Tabelle zu finden.

```
BCAT BPGM#
BCAT Basisadresse des BPGM
```

Bei BCAT handelt es sich um eine Kataloganweisung für die momentan im Computer enthaltenen Binärprogramme. Um den Katalog aufzurufen, kann entweder die Binärprogrammnummer oder die dezimale Basisadresse angegeben werden. Sollte die Binärprogrammnummer unbekannt sein, so können sie diese mit der kurzen Routine im Kapitel 2.3.1 suchen lassen. Um die Basisadresse des Binärprogrammes anzugeben, brauchen Sie das Assembler-ROM mit seiner REL-Funktion. Ist dieses vorhanden, so gehen Sie folgendermaßen vor:

```
LOADBIN "Binärprogramm"
REL (0)
122470
BCAT 0_D (122470)
```

Die Anzeige, die das BCAT Statement auf dem Bildschirm erzeugt, ist mit der RCAT-Anzeige identisch, nur die Adressangaben liegen im Bereich > 100000.

Notizen

Anhang A
Ein Disketteneditor

Der, auf den folgenden Seiten gelistete Diskettenеditor ermöglicht es dem Benutzer, einzelne Records der Diskette oder Winchester zu betrachten und beliebig zu ändern. Damit hat der Benutzer auch die Möglichkeit, defekte Disketten oder gelöschte Fileeinträge zu rekonstruieren.

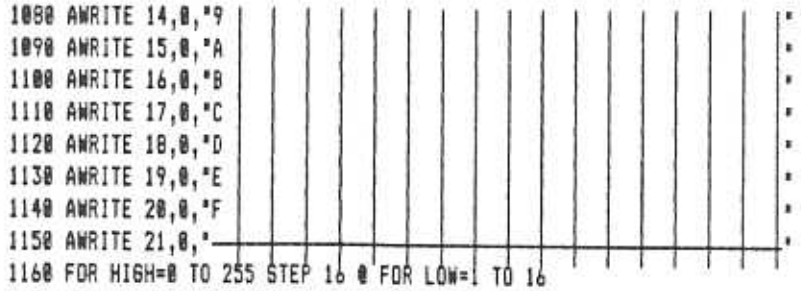
Das gesamte Programm arbeitet mit Maskensteuerung, d.h. die Tastatur ist immer gesperrt. Die Programmsteuertasten werden in den untersten beiden Zeilen erklärt. Folgende Steuertasten sind definiert:

[↑]	= Record n+1
[SHIFT] [↑]	= Record n+10
[SHIFT] [ROLLΔ]	= Record n+100
[↓]	= Record n-1
[SHIFT] [↓]	= Record n-10
[ROLL▽]	= Record n-100
[N]	= n'ter Record, fünfstellige Recordnummer eingeben
[S]	= angezeigten (editierten) Record auf Diskette schreiben
[E]	= Record editieren, Zeichenposition eingeben, neues Zeichen eingeben
[F]	= n eingeben, File n wird gesucht, und der erste Record des Files wird angezeigt

Die Kodierung der Anzeige ist hexadezimal. Zudem wird jede Spalte nochmals zur Übersicht als String angezeigt.

```
10 DIM WORK$[256],A$[256],B$[256],CRT$[512],BRANCH$[50]
20 TAKE KEYBOARD
30 BRANCH$=CHR$ (163)&CHR$ (152)&CHR$ (145)&CHR$ (164)&CHR$ (168)&CHR$ (169)&"NE
SF" ! Diese Zeile kann mit [SHIFT] [END LINE] auch direkt eingegeben werden
40 ON KEY$ 14 GOTO PROGRAMMER
50 CLEAR @ RECORD=0 @ GOSUB MS_UNIT
60 GOSUB READ_BUILD_UP
70 K=POS (BRANCH$,KEY$ ) @ IF NOT K THEN 70
80 ! Positionen entsprechen den Steuertasten: 1 = [UP], 2 = [SHIFT] [UP]
90 ! 3 = [SHIFT] [ROLL], 4 = [DOWN], 5 = [SHIFT] [DOWN], 6 = [ROLL]
100 ! 7 = N, 8 = E, 9 = S, 10 = F
110 ON K GOTO UP ,SHIFT_UP ,ROLL_UP ,DOWN ,SHIFT_DOWN ,ROLL ,ANY_RECORD ,EDIT ,S
TORE_RECORD ,FETCH_FILE
120 UP: RECORD=RECORD+1 @ GOSUB READ_BUILD_UP  @ GOTO 70
130 SHIFT_UP: RECORD=RECORD+10 @ GOSUB READ_BUILD_UP  @ GOTO 70
140 ROLL_UP: RECORD=RECORD+100 @ GOSUB READ_BUILD_UP  @ GOTO 70
150 DOWN: RECORD=RECORD-1 @ GOSUB READ_BUILD_UP  @ GOTO 70
160 SHIFT_DOWN: RECORD=RECORD-10 @ GOSUB READ_BUILD_UP  @ GOTO 70
170 ROLL: RECORD=RECORD-100 @ GOSUB READ_BUILD_UP  @ GOTO 70
180 ANY_RECORD: AWRITE 22,0,BLANK$ (160)
190 AWRITE 22,10,"Bitte die Recordnummer eingeben"
200 AWRITE 23,10,"5 Ziffern, oder n Zahlen und [END LINE]:"
210 GOSUB 270 @ AWRITE 23,50,CHR$ (K) @ RECORD=K-48
220 GOSUB 270 @ AWRITE 23,51,CHR$ (K) @ RECORD=RECORD*10+K-48
230 GOSUB 270 @ AWRITE 23,52,CHR$ (K) @ RECORD=RECORD*10+K-48
240 GOSUB 270 @ AWRITE 23,53,CHR$ (K) @ RECORD=RECORD*10+K-48
250 GOSUB 270 @ RECORD=RECORD*10+K-48
260 AWRITE 22,0,BLANK$ (160) @ GOSUB READ_BUILD_UP  @ GOTO 70
270 K=NUM (KEY$ ) @ IF NOT K THEN 270
280 IF K<48 OR K>57 AND K#154 THEN GOSUB ERRBP  @ GOTO 270
290 IF K=154 THEN POP RETURN @ GOTO 260
300 BEEP 20,20 @ WAIT 300 @ RETURN
310 ERRBP: BEEP 100,50 @ BEEP 200,25 @ RETURN
320 EDIT: AWRITE 22,0,BLANK$ (160)
330 AWRITE 22,0,"Bitte Zeilen-Spaltenkode des zu editierenden Elementes eingeben
:"
340 AWRITE 23,20,"[  ]"
350 A$=""
360 GOSUB 540 @ AWRITE 23,21,A$
370 GOSUB 540 @ AWRITE 23,21,A$[1,2]
380 K=NUM (HEX_ASC$ (A$)) @ E=K @ HIGH=K DIV 16+5 @ LOW=K MOD 16*3+3
390 AWRITE HIGH,LOW @ AREAD B$[1,2]
400 AWRITE HIGH,LOW,"  " @ AWRITE HIGH,E MOD 16+55,"  " @ GOSUB 480
410 AWRITE 22,0,"Bitte geben Sie den neuen Kode für dieses Feldelement ein:
 "
420 AWRITE 23,20,"    "
430 A$=""
440 GOSUB 540 @ AWRITE HIGH,LOW,A$
450 GOSUB 540 @ AWRITE HIGH,LOW,A$[1,2]
460 WORK$[E+1,E+1]=HEX_ASC$ (A$)
470 B$=A$ @ GOSUB 480 @ GOSUB DESCRIPTION  @ GOTO 70
480 FOR I=1 TO 10
490 AWRITE HIGH,LOW,"  " @ AWRITE HIGH,E MOD 16+55,"  "
500 WAIT 100
510 AWRITE HIGH,LOW,B$[1,2] @ AWRITE HIGH,E MOD 16+55,HEX_ASC$ (B$[1,2])
520 NEXT I
530 RETURN
540 K=NUM (KEY$ ) @ IF NOT K THEN 540 ELSE BEEP 20,20 @ WAIT 200
```

```
550 IF NOT LEN (A$) THEN A$=HEX$ (HEX_ASC$ (CHR$ (K))) @ A$=A$[2] ELSE A$[2]=HEX
$ (HEX_ASC$ (CHR$ (K))) @ A$[2]=A$[3]
560 RETURN
570 STORE_RECORD: AWRITE 22,0,BLANK$ (160)
580 ON ERROR GOTO DISC_ERROR1
590 WSECTOR WORK$,RECORD,MS$
600 GOSUB DESCRIPTION  @ BEEP 20,20 @ GOTO 70
610 DISC_ERROR1:
620 AWRITE 22,25,"Massenspeichererror: "&VAL$ (ERRN ) @ AWRITE 23,25,"Bitte Fehl
er beheben" @ GOSUB ERRBP @ GOTO 70
630 GOTO 70
640 FETCH_FILE: AWRITE 22,0,BLANK$ (160)
650 ON ERROR GOTO DISC_ERROR1
660 RSECTOR A$,0,MS$ @ KATALOG=NUM (A$[20]) @ AWRITE 0,49,VAL$ (KATALOG)&" Recor
ds    "
670 IF KATALOG>= RECORD-1 THEN GOTO 690
680 AWRITE 23,0,"Sie befinden sich nicht innerhalb des Kataloges" @ GOSUB ERRBP
690 AWRITE 22,0,"Geben Sie die Nummer des zu findenden Katalogeintrages ein 0 <
n < 9:"
700 K=NUM (KEY$ ) @ IF NOT K THEN 700
710 K=K-48 @ IF K<1 OR K>8 THEN GOSUB ERRBP  @ GOTO 700
720 AWRITE 22,80,VAL$ (K)
730 A$=WORK$[K*32-31] @ RECORD=ADR (REV$ (A$[15,16]))
740 AWRITE 1,0,"Filelänge: "&VAL$ (ADR (REV$ (A$[19,20])))
750 K=NUM (A$[12]) @ IF K=0 THEN A$="NULL" @ GOTO 850
760 IF K=8 OR K=9 OR K=10 THEN A$="BPGM" @ GOTO 850
770 IF K=12 OR K=13 OR K=14 THEN A$="GRAPH" @ GOTO 850
780 IF K=16 OR K=17 OR K=18 THEN A$="DATA" @ GOTO 850
790 IF K=20 OR K=21 OR K=22 THEN A$="ASSM" @ GOTO 850
800 IF K=32 OR K=33 OR K=32 THEN A$="PRGM" @ GOTO 850
810 !
820 ! Platz für weitere Typenprüfungen
830 !
840 A$="????"
850 AWRITE 1,20,"Filetyp: "&A$
860 GOSUB READ_BUILD_UP  @ GOTO 70
870 READ_BUILD_UP: ! Routine zum Lesen eines Records und aufbauen des
880                ! Bildschirmes
890 ON ERROR GOTO DISC_ERROR
900 RSECTOR WORK$,RECORD,MS$
910 CRT$=HEX$ (WORK$) @ IF RECORD=0 THEN KATALOG=NUM (WORK$[20])
920 AWRITE 0,0,"Massenspeicher: "&MS$&" Record: "&VAL$ (RECORD)&"     "
930 AWRITE 0,35,"Kataloglänge: "&VAL$ (KATALOG)&" Records     "
940 AWRITE 2,0,"HEX-Codierung: Reihen = High, Spalten = low           ASCII-Dar
stellung"
950 A$="  |0 | 1| 2| 3| 4| 5| 6| 7| 8| 9| A| B| C| D| E| F|"
960 AWRITE 3,0 @ AREAD B$[1,51] @ IF A$=B$ THEN GOTO 1160
970 AWRITE 3,0,A$
980 AWRITE 4,0,"--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  ----------
--------"
990 AWRITE 5,0,"0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1000 AWRITE 6,0,"1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1010 AWRITE 7,0,"2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1020 AWRITE 8,0,"3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1030 AWRITE 9,0,"4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1040 AWRITE 10,0,"5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1050 AWRITE 11,0,"6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1060 AWRITE 12,0,"7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
1070 AWRITE 13,0,"8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |"
```



```
1080 AWRITE 14,0,"9
1090 AWRITE 15,0,"A
1100 AWRITE 16,0,"B
1110 AWRITE 17,0,"C
1120 AWRITE 18,0,"D
1130 AWRITE 19,0,"E
1140 AWRITE 20,0,"F
1150 AWRITE 21,0,"
1160 FOR HIGH=0 TO 255 STEP 16 @ FOR LOW=1 TO 16
1170 AWRITE HIGH/16+5,LOW*3,CRT$[HIGH*2+LOW*2-1,HIGH*2+LOW*2]
1180 NEXT LOW
1190 AWRITE HIGH/16+5,55,WORK$[HIGH+1,HIGH+16]
1200 K=POS (BRANCH$,KEY$ ) @ IF K THEN C_RETURNS @ GOSUB DESCRIPTION  @ GOTO 110
1210 NEXT HIGH
1220 DESCRIPTION:
1230 AWRITE 22,0,"UP = Rec.+1, [SHIFT] UP = Rec.+10, [SHIFT] ROLL = Rec.+1
00, dito DOWN'
1240 AWRITE 23,0,"N = N'ter Rec., E = edit, S = speichern, F = File such
en
1250 RETURN
1260 DISC_ERROR: FOR I=2 TO 21 @ AWRITE 1,0,BLANK$ (80) @ NEXT I
1270 IF ERRN #130 THEN AWRITE 10,25,"Massenspeichererror: "&VAL$ (ERRN ) @ AWRIT
E 12,25,"Bitte Fehler beheben" @ GOSUB ERRBP @ C_RETURNS @ GOTO 70
1280 IF RECORD<0 THEN RECORD=0 @ GOSUB ERRBP  @ GOTO READ_BUILD_UP
1290 AWRITE 10,25,"Recordnummer zu groß" @ GOSUB ERRBP  @ RETURN
1300 MS_UNIT: MS$=PEEK$ (103500,0,3) @ MS$=":D"&VAL$ (NUM (MS$)+3)&VAL$ (NUM (MS
$[2]))&VAL$ (NUM (MS$[3])) @ RETURN
1310 PROGRAMMER: RELEASE KEYBOARD @ PAUSE
```

Die unterstrichenen Buchstaben müssen inversvideo eingegeben werden (siehe Kapitel 2.3.4).

Notizen

Anhang B

HP Serie 80 Referenztabelle

DEC	HEX	OCT	Binär	Mnemonik	Zeichen	Tastenkode
0	0	0	00000000	NUL		[CTRL] [@]
1	1	1	00000001	SOH		[CTRL] [A]
2	2	2	00000010	STX		[CTRL] [B]
3	3	3	00000011	ETX		[CTRL] [C]
4	4	4	00000100	EOT		[CTRL] [D]
5	5	5	00000101	ENQ		[CTRL] [E]
6	6	6	00000110	ACK		[CTRL] [F]
7	7	7	00000111	BEL		[CTRL] [G]
8	8	10	00001000	BS		[CTRL] [H]
9	9	11	00001001	HT		[CTRL] [I]
10	A	12	00001010	LF		[CTRL] [J]
11	B	13	00001011	UT		[CTRL] [K]
12	C	14	00001100	FF		[CTRL] [L]
13	D	15	00001101	CR		[CTRL] [M]
14	E	16	00001110	SO		[CTRL] [N]
15	F	17	00001111	SI		[CTRL] [O]
16	10	20	00010000	DLE		[CTRL] [P]
17	11	21	00010001	DC1		[CTRL] [Q]
18	12	22	00010010	DC2		[CTRL] [R]
19	13	23	00010011	DC3		[CTRL] [S]
20	14	24	00010100	DC4		[CTRL] [T]
21	15	25	00010101	NAK		[CTRL] [U]
22	16	26	00010110	SYN		[CTRL] [V]
23	17	27	00010111	ETB		[CTRL] [W]
24	18	30	00011000	CAN		[CTRL] [X]
25	19	31	00011001	EM		[CTRL] [Y]
26	1A	32	00011010	SUB		[CTRL] [Z]
27	1B	33	00011011	ESC		[CTRL] [[]
28	1C	34	00011100	FS		[CTRL] [\]
29	1D	35	00011101	GS		[CTRL] []]
30	1E	36	00011110	RS		[CTRL] [^]
31	1F	37	00011111	US		[CTRL] [_]

DEC	HEX	OCT	Binär	Zeichen	Tastenkode
32	20	40	00100000		Leertaste
33	21	41	00100001	!	[!]
34	22	42	00100010	"	["]
35	23	43	00100011	#	[#]
36	24	44	00100100	$	[$]
37	25	45	00100101	%	[%]
38	26	46	00100110	&	[&]
39	27	47	00100111	'	[']
40	28	50	00101000	(	[(]
41	29	51	00101001	)	[)]
42	2A	52	00101010	*	[*]
43	2B	53	00101011	+	[+]
44	2C	54	00101100	,	[,]

DEC	HEX	OCT	Binär	Zeichen	Tastenkode
45	2D	55	00101101	-	[-]
46	2E	56	00101110	.	[.]
47	2F	57	00101111	/	[/]
48	30	60	00110000	0	[0]
49	31	61	00110001	1	[1]
50	32	62	00110010	2	[2]
51	33	63	00110011	3	[3]
52	34	64	00110100	4	[4]
53	35	65	00110101	5	[5]
54	36	66	00110110	6	[6]
55	37	67	00110111	7	[7]
56	38	70	00111000	8	[8]
57	39	71	00111001	9	[9]
58	3A	72	00111010	:	[:]
59	3B	73	00111011	;	[;]
60	3C	74	00111100	<	[<]
61	3D	75	00111101	=	[=]
62	3E	76	00111110	>	[>]
63	3F	77	00111111	?	[?]
64	40	100	01000000	@	[@]
65	41	101	01000001	A	[A]
66	42	102	01000010	B	[B]
67	43	103	01000011	C	[C]
68	44	104	01000100	D	[D]
69	45	105	01000101	E	[E]
70	46	106	01000110	F	[F]
71	47	107	01000111	G	[G]
72	48	110	01001000	H	[H]
73	49	111	01001001	I	[I]
74	4A	112	01001010	J	[J]
75	4B	113	01001011	K	[K]
76	4C	114	01001100	L	[L]
77	4D	115	01001101	M	[M]
78	4E	116	01001110	N	[N]
79	4F	117	01001111	O	[O]
80	50	120	01010000	P	[P]
81	51	121	01010001	Q	[Q]
82	52	122	01010010	R	[R]
83	53	123	01010011	S	[S]
84	54	124	01010100	T	[T]
85	55	125	01010101	U	[U]
86	56	126	01010110	V	[V]
87	57	127	01010111	W	[W]
88	58	130	01011000	X	[X]
89	59	131	01011001	Y	[Y]
90	5A	132	01011010	Z	[Z]
91	5B	133	01011011	[	[[]
92	5C	134	01011100	\	[\]
93	5D	135	01011101	]	[]]
94	5E	136	01011110	^	[^]
95	5F	137	01011111	_	[_]
96	60	140	01100000	`	[SHIFT KEY LABEL]

DEC	HEX	OCT	Binär	Zeichen	Tastenkode
97	61	141	01100001	a	[SHIFT] [A]
98	62	142	01100010	b	[SHIFT] [B]
99	63	143	01100011	c	[SHIFT] [C]
100	64	144	01100100	d	[SHIFT] [D]
101	65	145	01100101	e	[SHIFT] [E]
102	66	146	01100110	f	[SHIFT] [F]
103	67	147	01100111	g	[SHIFT] [G]
104	68	150	01101000	h	[SHIFT] [H]
105	69	151	01101001	i	[SHIFT] [I]
106	6A	152	01101010	j	[SHIFT] [J]
107	6B	153	01101011	k	[SHIFT] [K]
108	6C	154	01101100	l	[SHIFT] [L]
109	6D	155	01101101	m	[SHIFT] [M]
110	6E	156	01101110	n	[SHIFT] [N]
111	6F	157	01101111	o	[SHIFT] [O]
112	70	160	01110000	p	[SHIFT] [P]
113	71	161	01110001	q	[SHIFT] [Q]
114	72	162	01110010	r	[SHIFT] [R]
115	73	163	01110011	s	[SHIFT] [S]
116	74	164	01110100	t	[SHIFT] [T]
117	75	165	01110101	u	[SHIFT] [U]
118	76	166	01110110	v	[SHIFT] [V]
119	77	167	01110111	w	[SHIFT] [W]
120	78	170	01111000	x	[SHIFT] [X]
121	79	171	01111001	y	[SHIFT] [Y]
122	7A	172	01111010	z	[SHIFT] [Z]
123	7B	173	01111011	{	[SHIFT] [/]
124	7C	174	01111100	|	[|]
125	7D	175	01111101	}	[SHIFT] [-]
126	7E	176	01111110	~	[SHIFT] [*]
127	7F	177	01111111		[SHIFT] [+]
128	80	200	10000000		[k1]
129	81	201	10000001		[k2]
130	82	202	10000010		[k3]
131	83	203	10000011		[k4]
132	84	204	10000100		[k8]
133	85	205	10000101		[k9]
134	86	206	10000110		[k10]
135	87	207	10000111		[k11]
136	88	210	10001000		[←-CHAR]
137	89	211	10001001		[CLEAR]
138	8A	212	10001010		
139	8B	213	10001011		[RESET]
140	8C	214	10001100		[INIT]
141	8D	215	10001101		[RUN]
142	8E	216	10001110		[PAUSE]
143	8F	217	10001111		[CONT]
144	90	220	10010000		[STEP]
145	91	221	10010001		[ROLL△]
146	92	222	10010010		[TEST]
147	93	223	10010011		[k14]
148	94	224	10010100		[LIST]
149	95	225	10010101		[PLIST]
150	96	226	10010110		[KEY LABEL]
151	97	227	10010111		[SHIFT END LINE]
152	98	230	10011000		[HOME]

DEC	HEX	OCT	Binär	Zeichen	Tastenkode
153	99	231	10011001		[BACK SPACE]
154	9A	232	10011010		[END LINE]
155	9B	233	10011011		[SHIFT BACK SPACE]
156	9C	234	10011100		[k7]
157	9D	235	10011101		[-LINE]
158	9E	236	10011110		[I/R]
159	9F	237	10011111		[←]
160	A0	240	10100000		[E]
161	A1	241	10100001		[k5]
162	A2	242	10100010		[k6]
163	A3	243	10100011		[↑]
164	A4	244	10100100		[↓]
165	A5	245	10100101		[k12]
166	A6	246	10100110		[RESLT]
167	A7	247	10100111		
168	A8	250	10101000		[A/G]
169	A9	251	10101001		[ROLL▽]
170	AA	252	10101010		[→]
171	AB	253	10101011		[SHIFT RUN]
172	AC	254	10101100		[k13]
173	AD	255	10101101		[TR/NORM]
174	AE	256	10101110		
175	AF	257	10101111		
176	B0	260	10110000		
177	B1	261	10110001		
178	B2	262	10110010		
179	B3	263	10110011		
180	B4	264	10110100		
181	B5	265	10110101		
182	B6	266	10110110		
183	B7	267	10110111		
184	B8	270	10111000		
185	B9	271	10111001		
186	BA	272	10111010		
187	BB	273	10111011		
188	BC	274	10111100		
189	BD	275	10111101		
190	BE	276	10111110		
191	BF	277	10111111		
192	C0	300	11000000		
193	C1	301	11000001		
194	C2	302	11000010		
195	C3	303	11000011		
196	C4	304	11000100		
197	C5	305	11000101		
198	C6	306	11000110		
199	C7	307	11000111		
200	C8	310	11001000		
201	C9	311	11001001		
202	CA	312	11001010		
203	CB	313	11001011		
204	CC	314	11001100		
205	CD	315	11001101		
206	CE	316	11001110		
207	CF	317	11001111		

DEC	HEX	OCT	Binär	Zeichen	Tastenkode
208	D0	320	11010000		
209	D1	321	11010001		
210	D2	322	11010010		
211	D3	323	11010011		
212	D4	324	11010100		
213	D5	325	11010101		
214	D6	326	11010110		
215	D7	327	11010111		
216	D8	330	11011000		
217	D9	331	11011001		
218	DA	332	11011010		
219	DB	333	11011011		
220	DC	334	11011100		
221	DD	335	11011101		
222	DE	336	11011110		
223	DF	337	11011111		
224	E0	340	11100000		
225	E1	341	11100001		
226	E2	342	11100010		
227	E3	343	11100011		
228	E4	344	11100100		
229	E5	345	11100101		
230	E6	346	11100110		
231	E7	347	11100111		
232	E8	350	11101000		
233	E9	351	11101001		
234	EA	352	11101010		
235	EB	353	11101011		
236	EC	354	11101100		
237	ED	355	11101101		
238	EE	356	11101110		
239	EF	357	11101111		
240	F0	360	11110000		
241	F1	361	11110001		
242	F2	362	11110010		
243	F3	363	11110011		
244	F4	364	11110100		
245	F5	365	11110101		
246	F6	366	11110110		
247	F7	367	11110111		
248	F8	370	11111000		
249	F9	371	11111001		
250	FA	372	11111010		
251	FB	373	11111011		
252	FC	374	11111100		
253	FD	375	11111101		
254	FE	376	11111110		
255	FF	377	11111111		

Anhang C
Serie 80 CPU-Befehle

low / high	0000 M0 0	0001 M1 1	0010 M2 2	0011 M3 3	0100 M4 4	0101 M5 5	0110 M6 6	0111 M7 7	1000 M8 8	1001 M9 9	1010 M10 A	1011 M11 B	1100 E0 C	1101 MS D	1110 E1 E	1111 E2 F
0000 0 0	ARP 00 § c@ 0	ARP * ç cA 1	ARP 02 ' cB 2	ARP 03 â cC 3	ARP 04 à cD 4	ARP 05 ß cE 5	ARP 06 ì cF 6	ARP 07 ò cG 7	ARP 10 ù cH 10	ARP 11 ˆ cI 11	ARP 12 ¨ cJ 12	ARP 13 ` cK 13	ARP 14 ° cL 14	ARP 15 cM 15	ARP 16 â cN 16	ARP 17 ê cO 17
0001 16 1	ARP 20 ì cP 20	ARP 21 d c@ 21	ARP 22 Ô cR 22	ARP 23 ê cS 23	ARP 24 ì cT 24	ARP 25 Ä cU 25	ARP 26 ê cV 26	ARP 27 Ô cW 27	ARP 30 ô cX 30	ARP 31 Ö cY 31	ARP 32 ö cZ 32	ARP 33 é c[33	ARP 34 [c\ 34	ARP 35 - c] 35	ARP 36 £ c^ 36	ARP 37 + c_ 37
0010 32 2	ARP 40 40	ARP 41 ! 41	ARP 42 " 42	ARP 43 # 43	ARP 44 $ 44	ARP 45 % 45	ARP 46 & 46	ARP 47 ' 47	ARP 50 (50	ARP 51) 51	ARP 52 * 52	ARP 53 + 53	ARP 54 , 54	ARP 55 - 55	ARP 56 . 56	ARP 57 / 57
0011 48 3	ARP 60 0 60	ARP 61 1 61	ARP 62 2 62	ARP 63 3 63	ARP 64 4 64	ARP 65 5 65	ARP 66 6 66	ARP 67 7 67	ARP 70 8 70	ARP 71 9 71	ARP 72 : 72	ARP 73 ; 73	ARP 74 < 74	ARP 75 = 75	ARP 76 > 76	ARP 77 ? 77
0100 64 4	DRP 00 @ 100	DRP 1 A 101	DRP 02 B 102	DRP 03 C 103	DRP 04 D 104	DRP 05 E 105	DRP 06 F 106	DRP 07 G 107	DRP 10 H 110	DRP 11 I 111	DRP 12 J 112	DRP 13 K 113	DRP 14 L 114	DRP 15 M 115	DRP 16 N 116	DRP 17 O 117
0101 80 5	DRP 20 P 120	DRP 21 Q 121	DRP 22 R 122	DRP 23 S 123	DRP 24 T 124	DRP 25 U 125	DRP 26 V 126	DRP 27 W 127	DRP 30 X 130	DRP 31 Y 131	DRP 32 Z 132	DRP 33 [133	DRP 34 \ 134	DRP 35] 135	DRP 36 ^ 136	DRP 37 _ 137
0110 96 6	DRP 40 ` 140	DRP 41 a 141	DRP 42 b 142	DRP 43 c 143	DRP 44 d 144	DRP 45 e 145	DRP 46 f 146	DRP 47 g 147	DRP 50 h 150	DRP 51 i 151	DRP 52 j 152	DRP 53 k 153	DRP 54 l 154	DRP 55 m 155	DRP 56 n 156	DRP 57 o 157
0111 112 7	DRP 60 p 160	DRP 61 q 161	DRP 62 r 162	DRP 63 s 163	DRP 64 t 164	DRP 65 u 165	DRP 66 v 166	DRP 67 w 167	DRP 70 x 170	DRP 71 y 171	DRP 72 z 172	DRP 73 { 173	DRP 74 | 174	DRP 75 } 175	DRP 76 ~ 176	DRP 77 □ 177
1000 128 8	ELB 0R § c@ 200	ELM 0R ç cA 201	ERB 0R ' cB 202	ERM 0R â cC 203	LLB 0R è cD 204	LLM 0R ß cE 205	LRB 0R ì cF 206	LRM 0R ò cG 207	ICB 0R ù cH 210	ICM 0R ˆ cI 211	DCB 0R ¨ cJ 212	DCM 0R ` cK 213	TCB 0R ° cL 214	TCM 0R cM 215	NCB 0R â cN 216	NCM 0R ê cO 217
1001 144 9	TSB 0R ì cP 220	TSM 0R d c@ 221	LLB 0R Ô cR 222	LLM 0R ê cS 223	DRB 0R, AR ì cT 224	DRM 0R, AR Ä cU 225	XRB 0R, AR ê cV 226	XRM 0R, AR Ô cW 227	BIN ô cX 230	BCD Ö cY 231	SAD ö cZ 232	DCE é c[233	ICE [c\ 234	DLE - c] 235	RTN £ c^ 236	PAD + c_ 237
1010 160 A	LOB 0R, AR 240	LOM 0R, AR ! 241	STB 0R, AR " 242	STM 0R, AR # 243	LOBD 0R, AR $ 244	LOMD 0R, AR % 245	STBD 0R, AR & 246	STMD 0R, AR ' 247	LOB 0R, =literal (250	LOM 0R, =literal) 251	STB 0R, =literal * 252	STM 0R, =literal + 253	LOBI 0R, AR , 254	LOMI 0R, AR - 255	STBI 0R, AR . 256	STMI 0R, AR / 257
1011 176 B	LOBD 0R, =label 0 260	LOMD 0R, =label 1 261	STBD 0R, =label 2 262	STMD 0R, =label 3 263	LOBD 0R, XAR, label 4 264	LOMD 0R, XAR, label 5 265	STBD 0R, XAR, label 6 266	STMD 0R, XAR, label 7 267	LOBI 0R, =label 8 270	LOMI 0R, =label 9 271	STBI 0R, =label : 272	STMI 0R, =label ; 273	LOBI 0R, XAR, label < 274	LOMI 0R, XAR, label = 275	STBI 0R, XAR, label > 276	STMI 0R, XAR, label ? 277
1100 192 C	CMB 0R, AR @ 300	CMM 0R, AR A 301	AOB 0R, AR B 302	AOM 0R, AR C 303	SBB 0R, AR D 304	SBM 0R, AR E 305	JSB XR, label F 306	ANM 0R, AR G 307	CMB 0R, =literal H 310	CMM 0R, =literal I 311	AOB 0R, =literal J 312	AOM 0R, =literal K 313	SBB 0R, =literal L 314	SBM 0R, =literal M 315	JSB =label N 316	ANM 0R, =literal O 317
1101 208 D	CMBD 0R, =label P 320	CMMD 0R, =label Q 321	AOBD 0R, =label R 322	AOMD 0R, =label S 323	SBBD 0R, =label T 324	SBMD 0R, =label U 325	V 326	ANMD 0R, =label W 327	CMBD 0R, AR X 330	CMMD 0R, AR Y 331	AOBD 0R, AR Z 332	AOMD 0R, AR [333	SBBD 0R, AR \ 334	SBMD 0R, AR] 335	^ 336	ANMD 0R, AR _ 337
1110 224 E	PUBD 0R, *AR ` 340	PUMD 0R, *AR a 341	PUBD 0R, -AR b 342	PUMD 0R, -AR c 343	PUBD 0R, *AR d 344	PUMD 0R, *AR e 345	PUBD 0R, -AR f 346	PUMD 0R, -AR g 347	PUBI 0R, *AR h 350	PUMI 0R, *AR i 351	PUBI 0R, -AR j 352	PUMI 0R, -AR k 353	PUBI 0R, *AR l 354	PUMI 0R, *AR m 355	PUBI 0R, -AR n 356	PUMI 0R, -AR o 357
1111 240 F	JNP label p 360	JNO label q 361	JOD label r 362	JEV label s 363	JNS label t 364	JPS label u 365	JNZ label v 366	JZR label w 367	JEN label x 370	JEZ label y 371	JNC label z 372	JCY label { 373	JLZ label | 374	JLN label } 375	JMZ label ~ 376	JAN label □ 377

Anhang D
ROM- und BPGM-Nummern

ROM-Nummern	
ROM#	Name
0	System
1	System Graphics
14	MIKSAM
24	LANG
40	ASSEMBLER
56	SYSEXT
176	MATRIX# 1
177	MATRIX# 2
192	I/O
207	SS (EXTEND.-MASS.)
208	System MASSSTORAGE
209	ED (ELECTRONIC DISC)
231	AP# 2 (ADVANCED PROG.)
232	AP# 1
240	PLOTTER OUTPUT

BPGM-Nummern	
BPGM#	Name
12	SPKB87
13	BIN15
14	STEVIE
16	FORM
18	IPBINg
19	KEYONBg
20	BIN24
22	UTIL/1
23	LINCURg
26	REDZERg
33	GETSAVEg
34	ASKIOB
37	TRACKBg
39	FILE/80BIN
41	TDGRAPH
44	GCURSBg
45	STRNGBg
46	MATHBIg
47	LIFg
48	BINCALC
56	SYSEXT
72	GDUMP
73	SORTB2g
74	FORMSBg
129	TEXTBIN

Anhang E
Stichwortverzeichnis

Syntax-Richtlinien

Punkt Matrix	In Punkt Matrix angegebene Begriffe müssen in exakt dieser Schreibweise, in Groß- oder Kleinschreibung eingegeben werden.
( )	Klammern schließen die Argumente der SYSEXT-ROM Funktionen ein.
[ ]	Eckige Klammern geben optionale Parameter an
gestapelte Parameter	Wenn zwei verschiedene Parameter übereinander stehen, so wird damit angegeben, daß beide Parameterformen gültig sind
F	Besagt, daß das Schlüsselwort eine Funktion ist
S	Besagt, daß das Schlüsselwort ein Statement ist

Die SYSEXT-ROM Funktionen und Statements

SBIT$ (String,Zahl) F Seite 20
Setzt in allen Zeichen des Strings oder Substrings das mit der Zahl angegebene Bitmuster

SCALL Adresse [,ROM#] S Seite 59
Ruft den Maschinenkode an der oktal angegebenen Adresse im definierten ROM auf
Bei nicht angegebener ROM# gilt ROM# 0

SCALL String S Seite 59
Ruft den Maschinenkode im String auf

SET ERRBR Adresse S Seite 25
Reaktiviert eine alte ON ERROR Bedingung

SET REPLACE$S String#1,String#2 S Seite 43
Setzt die mit RPL$ zu tauschenden Zeichen

SETPTR2 Adresse,Wert S Seite 56
Setzt den mit der Adresse oktal angegebenen 2-Byte-Zeiger auf den oktal angegebenen Wert

SETPTR3 Adresse,Wert S Seite 56
Setzt den mit der Adresse oktal angegebenen 3-Byte-Zeiger auf den oktal angegebenen Wert

SORT String,Teilstringlänge,Von,Bis S Seite 45
Sortiert den String entsprechend den Parametern

START CRT AT Zahl S Seite 28
Startet das Bildschirmfenster an der mit der Zahl angegebenen Reihe

TAKE KEYBOARD S Seite 31
Sperrt die Tastatur

TOKEN EXECUTE String S Seite 35
Führt den in dem String in Tokenform enthaltenen BASIC-Ausdruck aus

TOKEN$ (String) F Seite 35
Parsed und wandelt den in dem String enthaltenen BASIC-Ausdruck in Tokens um

TRIM$ (String) F Seite 20
Löscht alle Leerzeichen am Anfang und am Ende des Strings

UNMASK S Seite 30
Demaskiert die Tastatur

UNTIL Bedingung S Seite 48
Defiert das Ende eines REPEAT ... UNTIL Blockes

UPSORT String,Teilstringlänge,Von,Bis S Seite 46
Sortiert einen String entsprechend den Parametern, wobei Klein- wie Großbuchstaben gleichberechtigt sind

WHILE Bedingung DO S Seite 47
Definiert den Anfang eines WHILE Blockes

WSECTOR String,Record,MSUS$ S Seite 11
Schreibt den String an den angegebenen Record auf den definierten Massenspeicher

XOR$ (String#1,String#2) F Seite 21
Verknüpft die beiden Strings mit einem logischen XOR

Anhang F
Error-Meldungen

Neben den Standartfehlermeldungen mit den Nummern 1 bis 131 erzeugt das SYSEXT-ROM eine Anzahl eigener Fehlermeldungen. Die Funktion ERROM gibt, nachdem das ROM einen Error erzeugt hat, die ROM-Kennung 56 zurück. Die Error-Nummer kann mit ERRN abgefragt werden.

Nummer	Meldung	Erklärung
109	No Numeral	Die Funktion REGREP kann den angegebenen String nicht in eine Zahl umwandeln
110	Address > 16 Bit	Es wurde versucht, mit SCALL einen String aufzurufen, der in einem zu hohen Adressbereich liegt
111		Nicht belegt
112	WHILE Nesting	Es wurde versucht, mehr als 15 Ebenen zu verschachteln
113	Missing END WHILE	Eine WHILE Ebene wurde nicht mit END WHILE abgeschlossen
114	Missing WHILE DO	Ein END WHILE ist zuviel
115	No active WHILE	EXIT oder POP WHILE wurde ohne aktive Ebene durchgeführt
116	No active LOOP	END LOOP wurde ohne LOOP durchgeführt
117	incorrect Syntax	TOKEN$ oder EXECUTE wurde mit einem fehlerhaften BASIC-Ausdruck durchgeführt
118	No active REPEAT	POP UNTIL oder REPEAT ... wurde ohne aktive Ebene durchgeführt
119	REPEAT Nesting	Es wurde versucht, mehr als 15 Ebenen zu verschachteln
120	String len1 # String len2	SET REPLACES, AND, OR$ oder XOR$ wurde mit Strings von verschiedener Länge durchgeführt
121		Nicht belegt

Nummer	Meldung	Erklärung
122	Missing END BLIF	Zum durchgeführten BLIF Statement existiert kein END BLIF Statement
123	Missing BLIF	BLESE oder EXIT BLIF wurde ohne aktiven BLIF durchgeführt
124	BLIF Nesting	Es wurde versucht, mehrere BLIF Blöcke zu verschachteln
125	Element len = 0	Die beim SORT oder UPSORT angegebene Teilstringlänge ist 0
126	Element len > String len	Die beim SORT oder UPSORT angegebene Teilstringlänge ist größer als die Gesamtstringlänge
127	Last col > Element size	Die beim SORT oder UPSORT angegebene Bis-Variable ist größer als die Teilstringlänge
128	First col > last col	Die bei SORT oder UPSORT angegebene Von-Variable ist größer als die Bis-Variable
129	First col = 0	Die beim SORT oder UPSORT angegebene Von-Variable ist 0