Memory Map mit Wandervorschlägen (9)

Die Adressen 80 bis 143, die heute behandelt werden, benutzt der Basic-Interpreter und das Betriebssystem für Stringoperationen und zur Auswertung arithmetischer Ausdrücke. Auch die interessante Charget-Routine ist dabei.

Aufmerksamen Lesern wird es nicht entgangen sein, daß wir uns bei der Wanderung durch die Speicherlandschaft des C 64 beziehungsweise VC 20 selbst überholt haben. Versehentlich wurden die Adressen 80 bis 143, die heute behandelt werden, unterschlagen. Das nächste Mal werden wir dann wieder in der richtigen Reihenfolge fortfahren.

Adresse 80 — 82 ($50 — $52)

Zeiger auf einen provisorischen Speicherplatz einer Zeichenkette, die gerade bearbeitet wird

Die Teilprogramme (von Programmierern »Routinen« genannt) des Basic-Übersetzers im ROM des Computers, welche Zeichenketten (Strings) behandeln, verwenden die ersten beiden Bytes dieser drei Speicherzellen, nämlich 80 und 81, um in Low/High-Byte-Darstellung diejenige Speicheradresse anzugeben, ab der die Zeichenkette im Programmspeicher zu finden ist.

Das dritte Byte (82) enthält die Länge der Zeichenkette. Wegen der provisorischen Natur dieses Zeigers ist er für Basic-Programme nicht geeignet.

Adresse 83 ($53)

Flagge für die Garbage Collection

In dieser Speicherzelle steht während der sogenannten Garbage Collection (Müllabfuhr) eine Zahl, die angibt, ob die Variable der zur Überprüfung anstehenden Zeichenkette eine Länge von 3 oder 7 Byte hat.

Der Vorgang der Garbage Collection ist von B. Schneider, 64'er-Ausgabe 1/85, ausführlich beschrieben worden. Angaben über die Bedeutung der Variablen einer Zeichenkette finden Sie in Teil 4 und 5 des Memory Map-Kurses (64'er-Ausgaben 2/85 und 3/85).

Adresse 84 — 86 ($54 — $56)

Sprungbefehl auf die Adressen der Basic-Funktionen

Jede Basic-Funktion, wie zum Beispiel SGN, INT, ABS, USR und so weiter, wird durch ein spezielles Teilprogramm (Routine) des Basic-Übersetzers ausgeführt. Die Anfangsadresse jeder dieser Routinen sind in einer Tabelle im ROM fest eingespeichert. Im VC 20 steht diese Tabelle von 49234 bis 49279 ($C052 bis $C07F), im C 64 von 41042 bis 41087 ($A052 bis $A07F).

In der Speicherzelle 84 steht der Sprungbefehl JMP in Maschinencode, dargestellt durch die Zahl 75 ($4C). In den beiden anderen Zellen 85/86 steht dann in Low/High-Byte-Darstellung die jeweilige Adresse in der Tabelle, welche der vom Programm gerade gebrauchten Basic-Funktion entspricht. Dieser gesamte Befehl JMP plus Adresse entspricht in Basic der GOSUB-Zeilennummer.

Ein Beispiel soll das verdeutlichen. Geben Sie direkt ein:

```
PRINT PEEK(84);PEEK(85);
PEEK(86)
```

Wir erhalten
beim C 64: 76 13 184
beim VC 20: 76 13 216

Die erste Zahl ist genauso wie oben beschrieben. Die beiden anderen Zahlen ergeben zusammen die Adresse 47117 ($B80D) beziehungsweise 55309 ($D80D). Wenn Sie ein Buch mit ROM-Listing haben, werden Sie unter dieser Adresse die Routine für die Funktion »PEEK« finden. Das ist natürlich nicht erstaunlich, haben wir doch gerade vorher als letzten Befehl genau diese Funktion eingegeben.

Leider ist das auch die einzige Funktion, die ich Ihnen vorführen kann, denn zum Vorführen muß ich eben immer PEEKen, so daß beim besten Willen immer nur die oben angegebenen Zahlen erscheinen können.

Adresse 87 — 96 ($57 — $60)

Arbeitsspeicher für diverse Arithmetik-Routinen des Basic-Übersetzers

Diese zehn Speicherplätze werden von verschiedenen Teilprogrammen (Routinen), besonders bei arithmetischen Operationen, als Zwischenspeicher verschiedener Werte, Flaggen und Zeiger benützt.

Adresse 97 — 102 ($61 — $66)

Gleitkomma-Akkumulator Nr. 1

»Akkumulator« heißt seit der Zeit der mechanischen Rechenmaschinen eine Speicherzelle, welche bei Rechenoperationen dadurch im Mittelpunkt steht, daß laufend Daten in sie hineingeschrieben beziehungsweise aus ihr herausgelesen werden.

Normalerweise trägt diesen Namen das zentrale Rechenregister des Mikroprozessors. Leser des Assembler-Kurses kennen diesen Akkumulator inzwischen zur Genüge.

Die Speicherzellen 97 bis 102 werden deswegen ebenfalls Akkumulator genannt, weil sie bei der Verarbeitung von Gleitkommazahlen eine ähnliche zentrale Rolle spielen.

Am Anfang dieses Kurses in Ausgabe 11/84, bei der Beschreibung der Adressen 0 bis 2 des VC 20 (784 bis 786 beim C 64), und dann noch einmal im 64'er, Ausgabe 12/84, bei den Adressen 3/4 und 5/6 habe ich Ihnen versprochen, im Detail auf die Darstellung von Gleitkommazahlen und auf die Verwendung des damals schon genannten Gleitkomma-Akkumulators einzugehen. Heute wäre es nun soweit.

Inzwischen hat Herr Ponnath im Assemblerkurs mir die Arbeit aber bereits abgenommen. Im Teil 8 des Kurses (64'er, Ausgabe 4/85) finden Sie alle Details zu diesem Thema.

Hier soll uns eine kurze, zusammenfassende Bemerkung genügen.

Zelle 97 enthält den Exponenten. Die Zellen 98 bis 101 enthalten die Mantisse.

Zelle 102 enthält das Vorzeichen der Gleitkommazahl. Eine 0 bedeutet ein positives, die Zahl 255 ein negatives Vorzeichen.

Mit dem Gleitkomma-Akkumulator Nr. 1 sind zwei weitere Speicherzellen eng verbunden, nämlich 104 ($68) und 112 ($70).

Ganz zum Schluß ist noch erwähnenswert, daß nach der Umwandlung einer Gleitkommazahl in eine ganze Zahl diese als Low/High-Byte in den beiden Speicherzellen 98 und 99 steht, was für Maschinenprogramme vielleicht recht nützlich sein kann.

Adresse 103 ($67)

Zwischenspeicher beziehungsweise Zählregister

Diese Adresse wird von zwei Routinen verwendet. Der Basic-Übersetzer benützt sie als Vorzeichenspeicher bei der Umwandlung von Zahlen aus dem ASCII-Format in Gleitkommazahlen. Das Betriebssystem verwendet diese Adresse als Zähler der Abarbeitungsschritte bei der Berechnung eines Polynoms der Form

y = a0 + a1*x + a2*x↑2 + a3*x↑3 + ...

Adresse 104 ($68)

Überlauf-Speicher des Gleitkomma-Akkumulators Nr. 1

Wenn eine Zahl so groß wird, daß sie mit den zur Verfügung stehenden Stellen nicht mehr dargestellt werden kann, sprechen wir von einem »Überlauf«.

Bei Gleitkommazahlen liegt diese Überlaufgrenze bei $1{,}70141183 * 10^{38}$.

Während einer mathematischen Berechnung kann es intern im Computer vorkommen, daß ein Überlauf eintritt, der aber am Ende der Operation wieder verschwinden würde. Der Akkumulator Nr. 1 benützt in einem derartigen Fall die Speicherzelle 104, um die verfügbare Stellenzahl um 8 Bit zu vergrößern. Für endgültige Resultate steht diese Erweiterung natürlich nicht zur Verfügung.

Dieser Vorgang tritt besonders häufig bei der Umwandlung von ganzen Zahlen oder Zeichenketten in Gleitkommazahlen auf.

Adresse 105 — 110 ($69 — $6E)

Gleitkomma-Akkumulator Nr. 2

Spätestens jetzt verstehen Sie, warum der Akkumulator der Speicherzellen 97 bis 102 die Nr. 1 hat. Es gibt hier noch einen zweiten Gleitkomma-Akkumulator, der ein identischer Zwilling ist. Zwei Akkumulatoren sind immer dann notwendig, wenn mathematische Operationen ablaufen, welche mehr als einen Operanden verarbeiten, wie zum Beispiel Multiplikation, Division und so weiter.

Aufgrund der Identität der beiden Akkumulatoren kann ich mir eine weitere Beschreibung ersparen.

Adresse 111 ($6F)

Flagge für Vorzeichenvergleich der Gleitkomma-Akkumulatoren Nr. 1 und Nr. 2

Wenn die Zahl in beiden Akkumulatoren gleiche Vorzeichen haben, steht in Speicherzelle 111 eine 0, bei verschiedenen Vorzeichen eine 255.

Adresse 112 ($70)

Rundungsspeicher des Gleitkomma-Akkumulators Nr. 1

Es kann vorkommen, daß die Mantisse einer Gleitkommazahl mehr Stellen hat, als mit den vier Mantissen-Bytes des Akkumulators Nr. 1 (Zellen 90 bis 101) dargestellt werden können. In diesem Fall werden die hintersten, das heißt die unwichtigsten Stellen hinter dem Komma in der Zelle 112 abgelegt. Von dort werden sie geholt, um die Genauigkeit von mathematischen Operationen zu erhöhen und auch, um Endresultate abrunden zu können.

Adresse 113 — 114 ($71 — $72)

Zwischenspeicher für verschiedene Routinen

Diese Speicherzellen werden von sehr vielen Routinen des Übersetzers und des Betriebssytems, wie zum Beispiel Zeichenkettenverarbeitung, interne Uhr (TI$), Bestimmung der Größe von Feldern (Arrays) und etlichen anderen verwendet.

Adresse 115 — 138 ($73 — $8A)

Teilprogramm »Nächstes Zeichen eines Basic-Textes holen« (CHRGET-Routine)

Die Problematik der Übersetzung von Basic-Befehlen und Anweisungen besteht darin, daß die Übersetzungsschritte durch entsprechende Programmteile des Basic-Übersetzers im Computer fest vorprogrammiert sein müssen, was bedeutet, daß diese Programme natürlich im — nicht veränderbaren — ROM stehen.

Auf der anderen Seite verlangt aber der Übersetzungsvorgang, daß gewisse Teile dieser Programme sich laufend verändern. Als Beispiel soll der Zeiger herhalten, der angibt, in welcher Speicherzelle das nächste zu bearbeitende Zeichen steht. Dieser Zeiger und die zusammengehörigen Programmschritte dürfen natürlich nicht im ROM stehen, denn da sind sie ja nicht änderbar.

Dieser Konflikt wird dadurch gelöst, daß dieses »variable« Teilprogramm des Übersetzers zwar im ROM steht (im C 64 ab 58274 oder $E3A2, im VC 20 ab 58247 oder $E387), von wo es aber direkt nach dem Einschalten des Computers in das RAM, und zwar in die Speicherzellen 115 bis 138, umgeladen wird.

Dieses Teilprogramm, welches die Zeichen zur Übersetzung herbeiholt und deswegen »Character-Get« oder kurz CHARGET-Routine genannt wird, ist wegen seiner Veränderbarkeit natürlich ein beliebtes Objekt aller möglichen Manipulationen. Es ist deshalb im Assembler-Kurs, Teil 5, im 64'er, Ausgabe 1/85, im Detail beschrieben worden, allerdings mit Schwerpunkt auf Assembler/Maschinensprache.

Für Basic-Programmierer möchte ich hier deshalb eine kurze Beschreibung der CHRGET-Routine einfügen.

Die Routine beginnt mit einem Sprung auf den oben schon erwähnten Zeiger in Adresse 122/123, welcher seinerseits auf die Adresse zeigt, in welcher das nächste zu übersetzende Zeichen steht. Das Zeichen wird entsprechend dem Hinweis des Zeigers geholt, in den Akkumulator des Mikroprozessors geladen und dort verschiedenen Prüfungen unterzogen. Ist das Zeichen ein Gänsefuß, erkennt das Programm, wie es das nächste Zeichen interpretieren und behandeln muß. Ein Doppelpunkt leitet einen neuen Befehl ein, eine Leerstelle wird unterdrückt und so weiter.

Mit dem Befehl

```
PRINT PEEK(122)+256*
PEEK(123)
```

können wir innerhalb eines Programms ausdrucken, wohin der Zeiger nach dem letzten Basic-Zeichen deutet. Eine Überprüfung mit den Methoden, die ich bei der Besprechung der Speicherzellen 43 bis 56 genannt habe, zeigt Ihnen den Zusammenhang.

Normalerweise wird der Zeiger in 122/123 nach jedem Zeichen um 1 erhöht, da ja die Zeichen einer Basic-Zeile hintereinander im Speicher stehen. Ein GOTO- oder GOSUB-Befehl kann diese Folge natürlich unterbrechen, ebenso wie eine willkürliche Änderung durch einen Eingriff von außen.

Ein derartiger Eingriff, auch »wedge« (Keil) genannt, öffnet natürlich Tür und Tor für Programmiertricks, insbesondere für Einbau von neuen, selbsterfundenen Befehlen. Man kann entweder den allerersten Sprungbefehl auf den Zeiger so umlenken, daß er auf ein eigenes Maschinenprogramm springt, oder man kann den Zeiger selbst »verbiegen«, so daß er auf eine andere Adresse und damit auf ein anderes Zeichen zeigt. Es gibt dafür viele Möglichkeiten, die aber alle nur in Maschinencode funktionieren. Theoretisch können wir natürlich den Inhalt des Zeigers in 122/123 durch POKE verändern. Aber was dann? Jeder nachfolgende Basic-Befehl löst natürlich wieder die normale Übersetzungsroutine aus und unser schöner POKE ist für die Katz.

Wie so ein Wedge in Maschinensprache gemacht wird, hat Christoph Sauer im VC 20-Kurs — 64'er, Ausgabe 9/84 — beschrieben. Allerdings ist das Beispiel für Anfänger nicht verständlich, was mich zu der Überzeugung bringt, daß die CHRGET-Routine und ihre Anwendung einen eigenen Aufsatz wert wäre.

Adresse 139 — 143 ($8B — $8F)

Wert der RND-Funktion als Gleitkommazahl

Mit dem Befehl RND(X) kann bekanntlich eine Zufallszahl erzeugt werden. Was das bedeutet und wie »zufällig« diese Zahlen sind, können Sie dem nebenstehenden Texteinschub »Wie zufällig sind Zufallszahlen« entnehmen.

Beim Einschalten des Computers werden die Zahlen 128, 79, 199, 82 und 88 in diese Speicherzellen geschrieben. Mit der folgenden Zeile können Sie das gleich nach dem Einschalten des Computers leicht überprüfen.

```
FOR X=139 TO 143:PRINT
PEEK(X):NEXT
```

Nach den Manipulationen des RND-Befehls wird das Resultat wieder in die Zellen 139 bis 143 als neuer Ausgangswert (seed) für den nächsten RND-Befehl gebracht.

Diese fünf Zahlen stellen eine Gleitkommazahl dar. Ihre Form entspricht dabei der Aufteilung, wie sie auch im Gleitkomma-Akkumulator (97 bis 101) verwendet wird.

Eine Abfrage dieser Zahlen aus den Zellen 139 bis 143 ist natürlich möglich, aber nicht ergiebig, weil das Resultat von RND(X) direkt als Zahl verfügbar ist, während die 5 Bytes erst ein eine brauchbare Zahl umgerechnet werden müßten. Eine Änderung durch POKEn neuer Werte in diese Speicherzellen geht leider nicht.

Adresse 144 ($90)

Ein-/Ausgabe-Status ST

Ab dieser Speicherzelle wurde der Kurs irrtümlicherweise, wie schon erwähnt, im 64'er, Ausgabe 6/85, weitergeführt. Hiermit haben wir die Lücke geschlossen und den Anschluß erreicht.

Das nächste Mal werden wir dann wieder in der richtigen Reihenfolge fortfahren.

(Dr. H. Hauck/ah)

Wie zufällig sind Zufallszahlen?

Der Befehl RND(X) ergibt eine Zufallszahl zwischen 0 und 1 — so steht es im Commodore-Handbuch.

Eine Zufallszahl ist definitionsgemäß rein dem Zufall überlassen, ihr Wert kann nicht vorhergesehen werden. Wie kann aber ein Computer, in dem alle Vorgehensweisen und Arbeitsschritte fest vorprogrammiert sind, eine zufällige Zahl erzeugen? Die Commodore-Computer machen das so:

Der Befehl RND nimmt eine bestimmte Ausgangszahl (auf die ich noch näher eingehen werde), auf englisch »seed« = Samen genannt, multipliziert sie mit 11879546.4 und zählt 3.92767778 * 10^8 dazu. Die 5 Bytes der resultierenden Gleitkommazahl werden miteinander vertauscht und in einen positiven Bruch umgewandelt. Diese Manipulation ergibt die »Zufallszahl«, die als neuer »Samen« in den Speicherzellen 139 — 143 gespeichert wird.

Es ist sicher einzusehen, daß die Zufälligkeit nicht sehr hoch sein kann, es sei denn, die oben genannte und noch nicht erklärte Ausgangszahl ist zufällig.

Die erste Ausgangszahl hängt vom »Argument« des RND(X)-Befehls ab, das heißt vom Wert X, der in der Klammer dahinter steht. Es gibt drei Möglichkeiten für das Argument:
— eine positive Zahl (egal, welcher Wert)
— eine negative Zahl
— die Zahl 0
Eine positive Zahl
zum Beispiel RND (1) oder RND(56) nimmt als Samen die Zahl 0.811635157, die beim Einschalten des Computers als 5-Byte-Gleitkommazahl in die Speicherzellen 139 bis 143 geschrieben worden ist. In den fünf Zellen stehen die Zahlen 128, 79, 199, 82, 88.

Daraus folgt aber, daß nach dem Einschalten des Computers mit RND(1) immer dieselbe Sequenz von Zufallszahlen erzeugt wird. Schalten Sie bitte den Computer aus und ein und geben Sie ein:

```
10 PRINT RND(1):GOTO 10
```

Notieren Sie die ersten paar Zahlen und wiederholen Sie mit Aus-/Einschalten die Prozedur. Sie werden immer dieselben Zahlen sehen.

Zum Austesten von Programmen mit bekannten Zahlensequenzen ist diese Methode sicher wichtig, aber echte Zufallszahlen sind das nicht!

Eine **negative Zahl**
zum Beispiel RND(—1) oder RND(—95) bringt als erstes das Argument selbst (in meinem Beispiel —1 oder —95) als Gleitkommazahl in die Speicherzellen 139 bis 143, von wo sie als Samen den schon beschriebenen Manipulationen unterworfen wird. Nur — mit einem bestimmten negativen Argument erhalten Sie immer dieselbe Zufallszahl. Probieren Sie es aus:
PRINT RND(—2) ergibt immer dieselbe Zahl.

Es mag Fälle geben, wo die Vorgabe des allerersten Samens wünschenswert ist. Ich will aber von zufälligen Zahlen sprechen. Wir können diese Methode des negativen Arguments dadurch verbessern, daß wir als Argument selbst eine Zufallszahl nehmen.

Als derartige Zahl bietet sich der Wert der inneren Uhr TI an, die beim Einschalten des Computers losläuft und 60mal in der Sekunde weitergestellt wird. Da kein Mensch wissen kann, welchen Wert die Uhr TI gerade hat, kommt der Befehl RND(—TI) dem absoluten Zufall schon sehr nahe.

Das **Argument (0)**
verwendet eine andere Methode. Als Samen nimmt er eine sich ständig ändernde Zahl, die beim VC 20 aus vier Registerinhalten des VIC-Interface-Bausteins genommen werden. Beim C 64 wird es ähnlich gemacht, nur ist der VIC-Baustein ein anderer Typ.

Mit derselben Methode nach dem Einschalten wie im ersten Fall oben, können Sie das leicht überprüfen.

Ich habe eingangs zitiert, daß RND(X) eine Zahl zwischen 0 und 1 erzeugt; das gilt aber nur für ein positives Argument. Wenn Sie hingegen eine Zufallszahl innerhalb eines ganz bestimmten Bereiches brauchen, müssen Sie anders vorgehen.

Kochrezept Nr. 1

Mit folgender Formel ist der Zahlenbereich beliebig vorgebbar:
X=(RND(1)*A)+B

Die Zahl A gibt einen Bereich von 0 bis A vor.

Die Zahl B legt den untersten Wert des Bereiches fest.

Beispiele:
10 PRINT (RND(1)*6)+1:GOTO 10 erzeugt Zahlen von 1 bis 6
10 PRINT (RND(1)*52)+1:GOTO 10 erzeugt Zahlen von 1 bis 52
10 PRINT (RND(1)*6)+10:GOTO 10 erzeugt Zahlen von 10 bis 16

Mit dem Vorschalten der Funktion INT vor den Befehl RND werden die Zufallszahlen auf ganze Zahlen beschränkt.
10 PRINT INT (RND1)*6)+10:GOTO 10

Noch einmal: Zufallszahlen innerhalb bestimmter Zahlenbereiche sind gekoppelt mit einem positiven Argument. Wir haben aber gesehen, daß gerade so keine echten Zufallszahlen erzeugt werden. Deshalb brauchen wir noch ein zweites Kochrezept.

Kochrezept Nr. 2

Wenn Sie in einem Programm nach dem Einschalten des Computers immer neue Zufallszahlen brauchen, ist es empfehlenswert, für die allererste Zufallszahl RND(—TI) oder RND(0) zu verwenden, dann aber mit RND(1) fortzufahren.

Dasselbe gilt, wenn ein Programm wegen INPUT oder WAIT eine Pause hat. Nach der Pause sollte zuerst ein neuer Ausgangswert genommen werden.

Als letztes will ich noch beschreiben, wie man Zufallszahlen innerhalb von Maschinenprogrammen erzeugen kann.

Im Betriebssystem steht natürlich eine Routine für den Befehl RND. Im C 64 beginnt sie ab 57495 ($E097), im VC 20 ab 57492 ($E094).

Der Ausgangswert (Samen) wird dabei aus dem Gleitkomma-Akkumulator Nr. 1 geholt, dessen Vorzeichen oder Wert 0 das weitere Vorgehen der Routine bestimmt.

Sie müssen also den Samen in den Akkumulator Nr. 1 laden und dann mit JSR auf die RND-Routine springen. Als Resultat können Sie einen oder mehrere Werte der Zellen 140 bis 143 verwenden und nach Belieben weiterverarbeiten.

Fortsetzung von Seite 121

mit denen allgemein Schleifen wesentlich flexibler und übersichtlicher aufgebaut werden können. Dabei können 64 solcher Schleifen ineinander verschachtelt werden.

Weiterhin können Fehler im Programm mit dem Befehl TRAP abgefangen und mit den dazu vorhandenen Variablen EL, ER und ERR$ lokalisiert werden. Anschließend kann mit RESUME zu der Programmstelle zurückgesprungen werden, an der der Fehler auftrat.

Einige Kommandos zur Stringverarbeitung wurden ebenfalls eingebaut, wie etwa INSTR zur Stringsuche. Interessant ist hier die Erweiterung des Befehls MID$, der jetzt bei Zuweisungen auch links vom Gleichheitszeichen stehen kann und somit ein kontrolliertes Einsetzen von Zeichenketten in andere ermöglicht.

Die gesamte Speicherverwaltung wurde, wie am Anfang erwähnt, umgekrempelt. Deshalb wurden Befehle wie POKE und PEEK mit geändert und greifen jetzt nur noch auf den 64-KByte-RAM-Speicher zu. Um trotzdem alle Ein-/Ausgabebausteine zu erreichen, wurden einige fest installierte Variablenfelder eingerichtet. Man kann zum Beispiel mit den Variablen VIC(x) auf alle Register des Grafikbausteins direkt zugreifen, ohne einen einzigen POKE-Befehl verwenden zu müssen. Ähnliches gilt für den Soundchip und die beiden CIAs. Ferner läßt sich der Bildschirm- und Farbspeicher mit den Variablen VID(x) und COL(x) beeinflussen, wobei x von 0 bis 999 reicht. Daneben gibt es die Variablen BORDER, PAPER und INK, mit denen man die Bildschirmfarben direkt beeinflussen kann. Der Nachteil dieses Konzeptes ist es, daß Programme, die in Basic V2.0 geschrieben wurden und viele POKEs verwenden, oft nicht funktionieren.

Abgerundet wird die Palette der neuen Befehle durch Kommandos wie zum Beispiel UPPER und LOWER zum Festlegen des Groß- oder Kleinschriftmodus, CLS zum Löschen des Bildschirmes, RESET um den Einschaltzustand zu erreichen und MONITOR zum direkten Aufruf eines vorher eingeladenen Monitors, wenn dieser über einen BREAK-Einsprungspunkt verfügt. Auch der OLD-Befehl ist vorhanden, mit dem man ein NEW oder das oben erwähnte RESET wieder aufheben und damit ein Programm im Speicher wieder restaurieren kann. Ferner läßt sich mit dem Befehl STOP ON/OFF die Stop-Taste blockieren oder freigeben.

Das Handbuch erläutert all diese Befehle in verständlicher Art mit vielen Programmbeispielen.

Bemerkenswert ist in diesem Zusammenhang, daß Erweiterungen wie TurboAccess oder Hypra-Load (ROM-Version) weiterhin funktionieren.

Business Basic ist sowohl wegen der Fähigkeit mehr als 61 KByte direkt in Basic zur Verfügung zu stellen, als auch wegen seiner Qualitäten als Basic-Erweiterung sehr interessant. Nur einige Befehle zur Steuerung der hochauflösenden Grafik, die ja durchaus im Sinne dieser Erweiterung liegen würden, fehlen etwas. Der Name dieser Erweiterung ist trotzdem durchaus berechtigt, denn Business Basic erlaubt professionelles Programmieren zum Heimanwender-Preis (198 Mark).

(K. Hinsch/A. Wängler/rg)

Info: Kingsoft, Schnakebusch 4, 5106 Roetgen, Tel. 02408/8319, Preis: 198 Mark