Wer viel mit Strings arbeitet, wurde wohl schon öfter mit einem Problem konfrontiert, der Garbage Collection. Um was es dabei geht, und warum es sich lohnt, sich etwas mehr mit Strings zu beschäftigen, erfahren Sie in folgendem Artikel. Dies soll zugleich der Auftakt zu einer Serie sein, in der wir uns mit speziellen Problemen und Programmiertechniken beschäftigen werden.

Müllabfuhr im Comp
Die Garbage Co

Kennen Sie das? Sie haben ein Basic-Programm geschrieben, starten es, und eine Zeit lang läuft es perfekt. Und plötzlich, wenn Sie nur mal zwei Strings vertauschen oder die FRE-Funktion aufrufen, spielt Ihr Computer nicht mehr mit, ja, er reagiert nicht einmal mehr auf die STOP-Taste. Warten Sie dann einige Sekunden oder Minuten, ist der Spuk vorbei. Ihr Computer tut so, als sei nichts geschehen. In solchen Augenblicken hat der Basic-Interpreter mal wieder Müllabfuhr gespielt, er hat die »Garbage Collection« durchgeführt. Dieses Wort ist der Alptraum mancher Programmierer, die schnelle Programme schreiben möchten, welche Strings in großen Mengen benutzen.

Hier sind schon einige Stichworte gefallen: Bei der Garbage Collection handelt es sich wohl um etwas, das mit Strings zu tun hat und wohl auch mit Speicherplatz.

Um allerdings genau zu verstehen, was es mit dieser Routine auf sich hat, und welche programmtechnischen Wege man gehen muß, um sie zu meiden, müssen wir uns erst einmal mit dem computerinternen Aufbau von Strings und Stringarrays beschäftigen.

Was ist ein String?

Ein String ist eine Zeichenkette, also eine Aneinanderreihung von Zeichen aus dem Zeichensatz eines Computers. Unserem Computer zeigen wir an, daß er die nächste Eingabe als String betrachten soll, indem wir selbigem ein Gänsefüßchen voranstellen. Mit einem solchen geben wir ihm auch das Ende eines Strings bekannt.

Ein String hat immer eine bestimmte Länge, sie entspricht der Anzahl der in ihm enthaltenen Zeichen. Die Länge darf zwischen 0 und 255 betragen, weil der Computer nur über Umwege mehr als 255 Zeichen auf einmal überblicken könnte. Bei einer Länge von Null spricht man übrigens auch von einem Leerstring, weil keine Zeichen in ihm enthalten sind, nicht zu verwechseln mit einem String, der Leerzeichen (Spaces, Blanks) enthält.

Da es unpraktisch wäre, jedesmal, wenn ich einen bestimmten String bearbeiten will, diesen vollständig einzugeben, gibt es die Stringvariablen. Eine Stringvariable ist nichts anderes als ein Platzhalter für einen String. Wenn ich beispielsweise der Stringvariablen A$ den String »BORIS SCHNEIDER« zuteilen möchte, so schreibe ich:

```
A$="BORIS SCHNEIDER"
```

Das Dollarzeichen hinter dem Variablennamen zeigt dem Computer an, daß es sich hier um eine Stringvariable handelt. Nun kann ich im weiteren Verlauf jedesmal, wenn ich »BORIS SCHNEIDER« eingeben müßte, A$ dafür schreiben. Allerdings können Basic-Befehle so nicht abgekürzt werden. Wenn ich eingebe:

```
A$="PRINT":A$ 3*5
```

so erhalte ich nicht das gewünschte Ergebnis, 15, sondern schlicht und einfach SYNTAX ERROR. Das liegt daran, daß Basic-Befehle intern anders codiert sind als Strings.

Aufbau einer Stringvariablen

Wie sieht nun eine Stringvariable aus? Dazu betrachten wir ein Speichermodell des Computers. Die folgenden Erläuterungen beziehen sich auf Bild 1.

Beim Ablauf eines Programms läßt sich der Speicher in folgende fünf Bereiche aufteilen, deren Grenzen allerdings flexibel sind und sich laufend ändern:

— Basic-Programm

Das gerade im Speicher befindliche Programm liegt ganz unten, angefangen bei den kleinsten erreichbaren Adressen.

— Variable

In diesem Bereich befinden sich alle Variablen, Zahlvariablen wie auch Stringvariablen. In den Stringvariablen ist allerdings nicht der String selbst enthalten, sondern nur ein Zeiger auf die Adresse, wo sich der eigentliche String befindet.

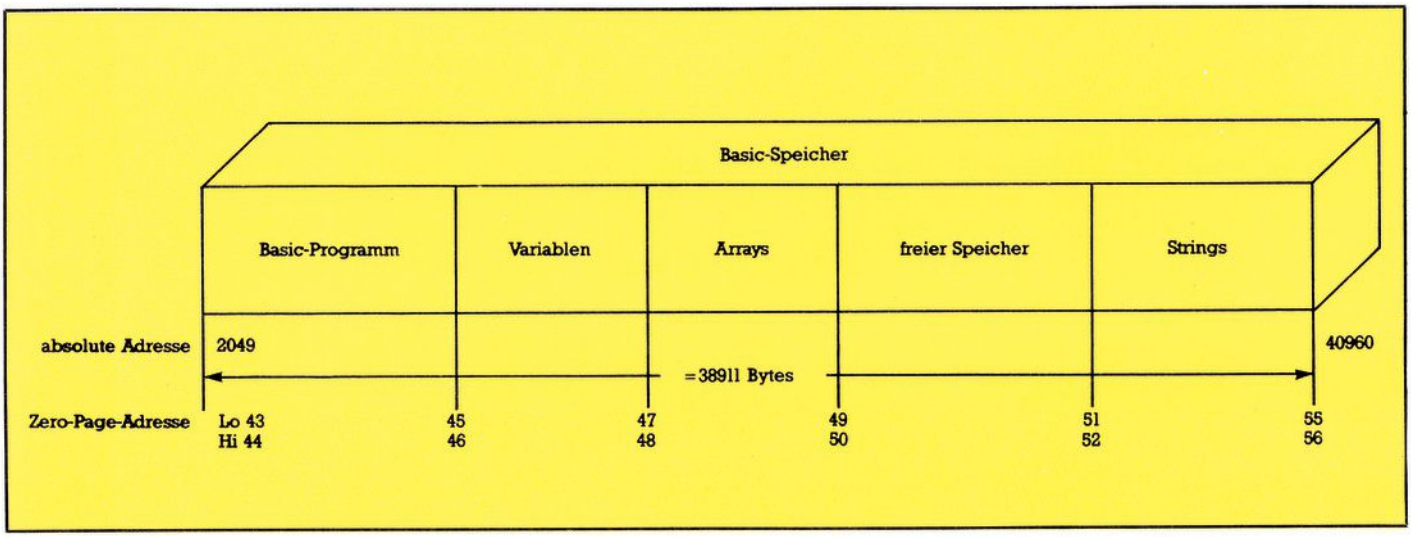

Bild 1. So ist der Basic-Speicher im Prinzip aufgeteilt. Bekannt sind normalerweise nur die Anfangs- und Endadresse des Speichers (2049/40960). Mit den Zeigern in der Zero-Page können alle Adressen abgefragt werden. Die Basic-Anfangsadresse errechnet sich zum Beispiel aus PRINT PEEK(43)+PEEE K(44)x256, der Variablenanfang aus PRINT PEEK(45)+PEEK(46)x256 und so weiter.

uter: llection

einer Ausnahme, aber davon später).

Nun kommen wir aber wieder auf die Stringvariablen selbst zurück.

Wie schon erwähnt, steht, speichertechnisch gesehen, in einer Stringvariablen gar kein String, sondern nur die Adresse, wo wir den Inhalt der Stringvariablen finden können. Das bemerken wir allerdings normalerweise nicht, da der Basic-Interpreter für uns automatisch den String an der entsprechenden Adresse abholt.

Der genaue Aufbau einer Stringvariablen steht in Tabelle 1. Hier einige Erläuterungen dazu:

Jede Variable ist durch eine Zweizeichen-Kombination gekennzeichnet. Das erste Zeichen muß ein Buchstabe, das zweite Zeichen darf auch eine Ziffer sein. Durch das Anhängen eines »$« definieren Sie die vorstehende Variable als Stringvariable. Intern speichert der Computer nun das Dollarzeichen nicht mit, sondern kennzeichnet eine Stringvariable, indem er zum Code des zweiten Buchstabens 128 addiert. So werden nur zwei Speicherstellen benötigt, um den Variablennamen zu speichern. Bei Zahlvariablen wird nichts addiert, bei Integervariablen (Ganzzahl) zu beiden Codes jeweils 128.

Das dritte Byte gibt die Länge des Strings an. Dies ist notwendig, da die eigentlichen Strings in ihrem Speicherbereich ohne Trennzeichen einfach aneinandergehängt sind. Wüßte der Computer nicht, wie lang der gefragte String ist, so könnte er auch nicht sein Ende feststellen. Da kein Trennzeichen verwendet wird, darf in einem String auch jedes Zeichen, das der Computer kennt, vorkommen. Außerdem wird so der Speicherplatz am besten ausgenutzt.

Die Bytes 4 und 5 enthalten die Adresse des Strings im Low/High-Byte-Format. Um diese beiden Bytes in die entsprechende Zahl umzuwandeln, müßte folgende Rechnung durchgeführt werden:
Byte 4 + (256 x Byte 5)

Die Bytes 6 und 7 enthalten beide den Wert Null. Wenn Sie nach dem Sinn und Zweck fragen, so ist die Antwort, daß es für den Computer einfacher ist, wenn alle Variablentypen, Zahlen wie Strings, die gleiche Länge haben. Eine Zahlvariable belegt nun genau 7 Bytes, deswegen werden Stringvariable auf diese Länge aufgefüllt.

Der Teil der Stringvariablen, der Länge und Adresse angibt, wird als Descriptor bezeichnet.

```
0 REM +++ STRINGVERTAUSCHER +++
1 REM  GESCHRIEBEN AM 07.10.84
2 REM    VON BORIS SCHNEIDER
3 REM
4 REM SYNTAX:
5 REM SYS STARTADRESSE(STRING1,STRING2)
6 REM
7 REM BELIEBIG IM SPEICHER VERSCHIEBBAR
8 :
10 DATA  32,250,174, 32,158,173, 32,143
20 DATA 173,165,100,133,247,165,101,133
30 DATA 248, 32,253,174, 32,158,173, 32
40 DATA 143,173,160,  0,177,247,133,249
50 DATA 177,100,145,247,165,249,145,100
60 DATA 200,192,  3,208,239, 32,247,174
70 DATA  96,  0,  0,  0,  0,  0,  0,  0
80 :
100 INPUT"STARTADRESSE";SA
110 FOR I = SA TO SA+48
120 :READ X:POKEI,X:CS=CS+X
130 NEXTI
140 IF CS<>7314THEN PRINT"FEHLER!!"
150 END
READY.
```

Listing 1. Mit diesem Programm können Sie Strings vertauschen, ohne daß dabei Stringmüll entsteht.

— Arrays

Hier befinden sich die Arrays, auch Felder genannt. Wenn Sie mehr als elf Elemente enthalten sollen, müssen Sie vor ihrer Benutzung erst durch den DIM-Befehl definiert werden, damit der Basic-Interpreter genügend Speicherplatz in diesem Bereich bereitstellt.

— Freier Speicher

Freier Speicher existiert interessanterweise nicht am oberen oder unteren Ende des Speichers, sondern in der Mitte, zwischen Arrays und Strings. Das hat aber enorme Vorteile, wie wir noch sehen werden.

— Strings

Hier sind sie endlich, am oberen Ende des Speichers, die von uns gesuchten Strings. In diesem Bereich befinden sich, dicht aneinander gepackt, die Inhalte der Stringvariablen (mit

Stringarrays

Nun gibt es aber neben den normalen Stringvariablen auch Stringarrays. Dies sind ein- oder mehrdimensionale Platzhalter. Am besten läßt sich das mit einem Beispiel erklären. Mit dem Befehl DIM A$(100,5) wird ein zweidimensionales Stringarray definiert. In diesem Array lassen sich bestimmte Strings mit Hilfe von zwei Koordinaten ablegen, zurückholen oder bearbeiten. Das soeben definierte Array läßt sich mit genau 606 Strings auffüllen. Man darf die erste Koordinate, oder besser gesagt den ersten Index, von 0 bis 100 und den zweiten Index von 0 bis 5 angeben. Dies sind 101 x 6 = 606 Platzhalter für Strings. Der Vorteil von Stringarrays ist, daß mit ihnen berechnete Zugriffe auf Strings möglich werden. So lassen sich zum Beispiel Strings in eine Reihenfolge bringen, diese Reihenfolge aber auch beliebig ändern, indem zwei Strings miteinander vertauscht werden. Anders gesagt sind zum Beispiel Sortierprogramme erst mit Stringarrays möglich geworden. Denn es gibt wohl keinen Weg, ohne große Tricks und viel Speicheraufwand die Inhalte der Stringvariablen A$, B$ .... Z$ zu sortieren. Mit einem Array ist dies sehr einfach möglich. Doch nun zu den Stringarrays selber.

Auch hierzu habe ich wieder eine Tabelle zusammengestellt (Tabelle 2).

Wie Sie schon auf einen Blick sehen, ist der Aufbau von Stringarrays ungleich komplizierter als der von normalen Stringvariablen.

Hier bezeichnet man den gesamten Block der einzelnen Stringdescriptoren als Arraydescriptor (description = Beschreibung, Darstellung).

In den ersten beiden Bytes steht, wie bei normalen Stringvariablen, der Name im Interpretercode. Auch hier wird zum zweiten Byte 128 addiert. Als nächstes wird, wieder im Low/High-Format, die Gesamtlänge des Arrays angegeben (Byte 3 und 4). Diese Gesamtlänge umfaßt sowohl den Arraydescriptor, wie alle nachfolgenden Stringdescriptoren.

Im Byte 5 wird die Anzahl der Dimensionen angegeben. Im Beispiel A$(100,5) wäre dies 2, da ja zwei voneinander abhängige Indizes vorhanden sind. Man spricht auch von einem zweidimensionalen Array oder Feld. Jeder Index definiert also eine Dimension. Da ein Byte maximal den Wert 255 aufnehmen kann, wäre dies auch die höchstmögliche Dimensionierung.

Nun steht hintereinander die Anzahl der Elemente jeder einzelnen Dimension in absteigender Reihenfolge. Diese Angaben sind wieder im Low/High-Format. Um auf unser Beispiel zurückzukommen: Zuerst würde hier eine 5 stehen und dann eine 100. (Beide im Low/High-Format!)

Damit wäre dann der Arraydescriptor abgeschlossen; an ihn schließen sich die Descriptoren der einzelnen Strings an. Jeder Stringdescriptor belegt 3 Bytes: Das erste gibt die Stringlänge, die beiden anderen die Speicheradresse des Stringinhalts an. Diese Stringdescriptoren stehen in folgender Reihenfolge hinter dem Arraydescriptor: Zuerst wird der Index der höchsten Dimension hochgezählt, dann der der zweithöchsten und so weiter, bis zur Dimension 1. Auch hier greife ich kurz auf unser Beispiel zurück: Der erste angegebene Stringdescriptor bezieht sich auf den String A$(0,0), der zweite auf A$(0,1), der sechste auf A$(0,5) der siebte auf A$(1,0), ... der 606te auf A$(100,5). Damit wäre dann auch die Liste der Stringdescriptoren zu Ende.

Diesen kleinen Ausflug in die Speichertechnik möchte ich mit einer Formel beenden, mit der Sie die Größe eines Arraydescriptors und seiner Stringdescriptoren berechnen können. Sie lautet: 5 + (Anzahl der Dimensionen) x 2 + (Anzahl der Elemente Dimension 1) x (Anzahl Elemente Dimension 2) x ... (Anzahl Elemente Dimension n) x 3.

Wichtig wäre hierbei, daß Sie bei der Angabe der Anzahl der Elemente beachten, daß die Zählung immer bei Null beginnt. A$(100,5) belegt also 5 + 2 x 2 + 101 x 6 x 3 = 1827 Bytes, die In-

halte der einzelnen Variablen natürlich ausgeschlossen.

Ein Blick in den Speicher

Nun wissen wir also, wie Stringvariablen und -arrays aussehen. Was passiert nun aber, wenn ich einen String anlege oder ihn bearbeite?

Um uns die Stringverarbeitung des Computers genauer anzusehen, bedienen wir uns ein paar speichertechnischer Tricks.

Wie schon erwähnt, stimmen die Adressenangaben, die ich bei unserer Beispielspeicherbelegung gemacht habe, nicht. Der Beginn eines Basic-Programms liegt normalerweise an der Adresse 2048, das Ende des Speichers bei 40959.

Die Grenzen zwischen den einzelnen Bereichen sind, wie gesagt, nicht exakt festlegbar, da sie sich laufend ändern. Sollten Sie sich für die gerade bestehenden Grenzen interessieren, so können Sie sie mittels PEEK erfragen. In den in Tabelle 3 angegebenen Speicherstellen stehen die jeweiligen Grenzen im schon erwähnten Low/High-Format. Um also beipielsweise den Beginn der Strings auszudrucken, müssen Sie folgendes eingeben:

PRINT PEEK(51)+256xPEEK(52)

Wir beschreiten nun einen sehr ungewöhnlichen, aber wirkungsvollen Weg. Wir verkleinern unseren Speicher so, daß er auf den Bildschirm paßt, und legen den Bildschirmspeicher an diese Stelle. Wir können dann zwar nicht mehr sehen, was wir eintippen, haben aber einen vollen Überblick über den Speicher.

Geben Sie bitte folgendes ein:

FORI=2060TO3072:POKEI,32:NEXTI, POKE 56,11:POKE 53272, 37:CLR.

Sie haben im Augenblick zirka 750 Bytes freien Speicherplatz, der Rest wurde vorhin für Basic gesperrt.

Tippen Sie nun mal (blind) A$="ihr name". Sie sehen an zwei Stellen auf dem Schirm ein paar Grafikzeichen. Drücken Sie gleichzeitig die SHIFT- und die COMMODORE-Taste, um auf Kleinschrift umzuschalten. Nun erkennen Sie Ihren Namen, der im String A$ gespeichert wurde und an der obersten Speichergrenze steht.

Die Stringvariable selbst am oberen Bildschirmrand können Sie auch jetzt nicht entziffern, da sie im Interpretercode gespeichert wird. Zumindest können Sie aber die Stringinhalte lesen.

Jetzt wird es interessant: Tippen Sie mal A$="BORIS SCHNEIDER".

Sie werden bemerken, daß ihr Name stehengeblieben ist, während mein Name einfach vorne angehängt wurde.

Dies ist der Haken der Stringverarbeitung, denn Ihr Name ist jetzt, bitte nehmen Sie es nicht persönlich, zu Stringmüll geworden. Er steht immer noch im Speicher herum, obwohl kein Stringdescriptor mehr auf ihn zeigt.

Dies macht einen großen Teil der gegenüber anderen Computern hohen Geschwindigkeit der Stringverarbeitung aus, denn neue Strings werden einfach vor die anderen gesetzt, ohne daß diese überprüft werden, ob sie noch gültig sind. Hier findet sich dann auch die Erklärung, warum freier Speicher gerade in der Mitte anzutreffen ist, zwischen Arrays und Strings. Denn während die Arrays und anderen Variablen mit zunehmender Zahl nach oben wachsen, wachsen die Strings nach unten. Der dazwischenliegende Speicher wird immer kleiner.

Byte	Bedeutung
1	Buchstabe 1 des Stringnamens im Interpretercode
2	Buchstabe 2 des Stringsnamens im Interpretercode + 128
3	Länge des Strings
4	LSB der Adresse des Strings
5	MSB der Adresse des Strings
6 & 7	Auffüllbytes, immer =0

Tabelle 1. Der Aufbau einer normalen Stringvariablen.

Byte	Bedeutung
1	Buchstabe 1 des Arraynamens im Interpretercode
2	Buchstabe 2 des Arraynamens im Interpretercode + 128
3	LSB der Länge des Arrays
4	MSB der Länge des Arrays
5	Anzahl der Dimensionen
6 & 7	LSB & MSB der Anzahl der Elemente in der höchsten Dimension =n
8 & 9	LSB & MSB der Anzahl der Elemente der Dimension n-1
...	u.s.w. bis:
? & ?	LSB & MSB der Anzahl der Elemente der Dimension 1
Ende	Jeweils drei Byte Stingdescriptoren, sortiert in aufsteigender Reihenfolge, wobei der letzte Index als erster durchläuft.

Tabelle 2. So sehen Stringarrays im Computerspeicher aus. Nähere Erläuterungen im Text.

Diese Sequenz vertauscht die Inhalte der zwei Stringvariablen A$ und B$. So zu tauschen ist der einfachste Weg, er hat aber auch zwei große Nachteile:

Adresse	Bedeutung
43,44	Start des Basic-Programms
45,46	Ende des Basic-Programms, Start der Variablen
47,48	Ende der Variablen, Start der Arrays
49,50	Ende der Arrays, Start freier Speicher
51,52	Ende freier Speicher, Start der Strings
55,56	Ende der Strings, Ende des für Basic verfügbaren Speichers.

Tabelle 3 enthält die Adressen, unter denen Sie die aktuellen Speichergrenzen erfahren können.

Und was passiert nun, wenn Strings und Arrays »zusammenstoßen«?

Hier gibt es zwei Möglichkeiten: Da ja kein Speicherplatz mehr frei ist, wird OUT OF MEMORY angezeigt, oder ...

— Es wird eine zusätzliche Hilfsvariable, hier H$ benötigt.
— Es entsteht eine Menge Stringmüll, nämlich 150 Prozent!

Schauen wir uns mal genau an, was bei der oben genannten Sequenz im Speicher passiert.

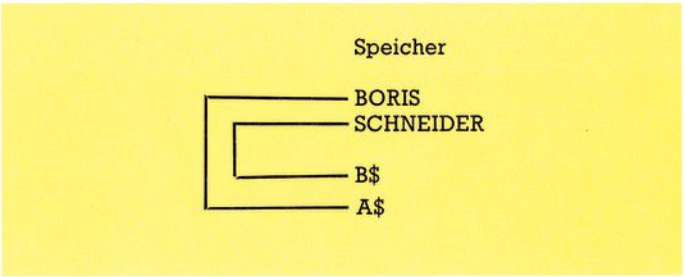

Bild 2. So sieht der Speicher nach dem Befehl A$="BORIS":B$="SCHNEIDER" aus. Oben stehen die Strings, unten die dazugehörigen Descriptoren.

Wie Sie sich vielleicht schon gedacht haben, können sich unter den Strings eine Vielzahl von Müllstrings befinden, die einfach nicht mehr gebraucht werden, sondern nur Speicherplatz schlucken. Könnte man diese entfernen, und nur noch die tatsächlich verwendeten Strings übriglassen, so wäre wieder eine Menge Speicher für weitere Anwendungen frei.

Nun sind wir also endlich beim Stichwort: Garbage Collection. Dies ist die vorhin angesprochene Möglichkeit der Müllbeseitigung.

Problemkind: Das Vertauschen

Bei der Garbage Collection wird sämtlicher vorhandener Stringmüll entfernt; so wird der größtmögliche Speicherplatz frei.

Wie arbeitet nun die Garbage Collection? Auch dies zeigt man am besten an einem Beispiel. Am praktischsten finde ich das Stringvertauschen. Sie wissen ja wahrscheinlich, wie man normalerweise zwei Strings vertauscht:

H$=A$: A$=B$: B$=H$

Wenn Sie es selbst am Bildschirm verfolgen wollen, geben Sie bitte nochmals die oben genannten POKE-Befehle ein. Ansonsten betrachten Sie bitte Bild 2 bis 6.

Zuerst definieren wir die zwei Strings A$ und B$:

A$="BORIS" : B$="SCHNEIDER" (Sie können selbstverständlich jeden anderen Stringinhalt wählen.) Sie sehen, daß das Betriebssystem zwei Strings im Speicher abgelegt hat; die Descriptoren der entsprechenden Variablen zeigen auf diese (Bild 2).

Nun folgt der erste Vertauschungsschritt: H$=A$. Sie sehen, daß nun zweimal der String "BORIS" im Speicher steht, einmal als Inhalt von A$, einmal von H$ (Bild 3).

Der nächste Schritt wäre: A$=B$.

Schon wieder wird ein schon vorhandener Inhalt nochmals angehängt, in diesem Falle das »SCHNEIDER«. Das erste »BORIS« ist jetzt schon Stringmüll, da kein Descriptor auf ihn zeigt (Bild 4).

Als letztes käme noch: B$=H$. Nochmals wird der String »BORIS« in den Speicher geschrieben. Damit ist auch der vorheri-

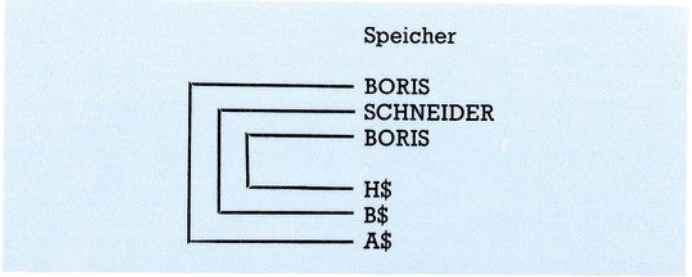

Bild 3. HS=AS. Bitte beachten Sie, daß in Wirklichkeit die Strings nicht unter- sondern in »Wirklichkeit« nebeneinander stehen.

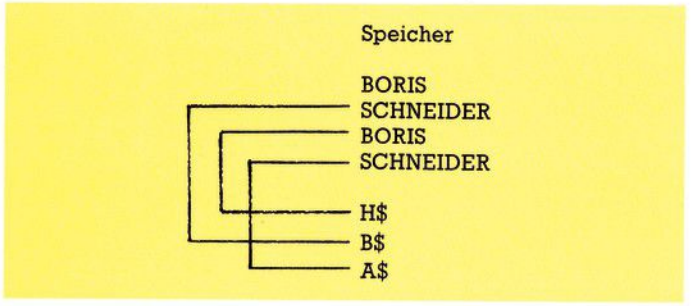

Bild 4. Nach A$=B$ ist der erste Müllstring entstanden

ge Inhalt von B$, der erste »SCHNEIDER«, zu Stringmüll geworden (Bild 5).

Zum Schluß wäre zu beachten, daß wir ja nun die Variable H$ nicht mehr benötigen, da sie nur Zwischenspeicher bei der Vertauschung war. Wenn wir also schreiben: H$="", wird auch das zweite "BORIS" zu Stringmüll (Bild 6).

Nun rechnen wir mal: Fünf Strings stehen im Stringspeicher, aber nur zwei werden benötigt. Eine traurige Bilanz, wenn man bedenkt, daß wir ja nur zwei Stringvariablen vertauscht haben. Denn bei jeder Stringfunktion entsteht Stringmüll. Probieren Sie doch mal A$=A$, und Sie werden sehen, wie schnell sich Stringmüll entwickeln kann.

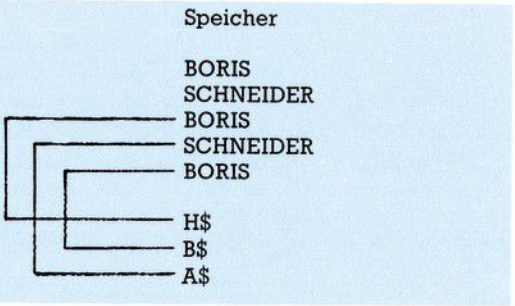

Bild 5. Und so sieht's nach B$=H$ aus

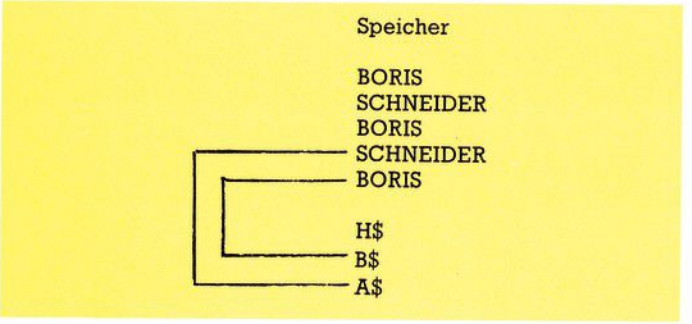

Bild 6. Das ganze entwirrt sich bei H$=""

Allerdings gibt es eine löbliche Ausnahme, die man immer ausnutzen sollte. Wenn Sie Stringvariablen in einem Programm definieren, zum Beispiel: 10 A$="HALLIHALLO", so werden Sie am oberen Speicherende vergeblich nach "HALLIHALLO" suchen! Denn im Programmtext ist dieser String ja schon enthalten, wieso sollte der entsprechende Descriptor nicht einfach auf die entsprechende Stelle im Programmtext zeigen? Das tut er auch; zu beachten wäre aber, daß bei jeder weiteren Manipulation an dieser Stringvariable ein String im Stringspeicher abgelegt wird; es können also beim Vertauschen mehrerer im Programm definierter Stringvariablen ohne weiteres wieder Müllstrings entstehen.

Ich würde Ihnen empfehlen, einige andere Stringoperationen selbst auszuführen und sich die Effekte mit Hilfe des oben genannten Tricks selbst anzusehen.

Doch nun endlich zurück zur Garbage Collection und unserem Beispiel. Wir können den entstandenen Stringmüll durch Aufrufen der FRE-Funktion beseitigen, zum Beispiel: PRINT FRE(0).

Garbage Collection

Wenn Sie dies bei unserem Beispiel machen, dann sehen Sie, wie blitzschnell der Speicher umorganisiert wird; leider zu schnell, denn wir bekommen gar nicht mit, was passiert. Deswegen gehen wir also die Verfahrensweise der Garbage Collection an unserem Beispiel durch (Bild 7 und 8).

Die Garbage Collection durchsucht den gesamten Variablenspeicher nach dem String, der den am höchsten liegenden Pointer hat. Dabei werden alle Strings, die die Länge Null haben, übergangen, da sie ja keinen Inhalt haben können.

In unserem Fall ist das ziemlich einfach, da wir ja nur zwei richtige Strings haben, A$ und B$. H$ hat ja die Länge 0, wird also nicht beachtet.

Der String, der den höchsten Descriptor hat, ist bei uns A$, denn der Inhalt von A$ steht an einer höheren Stelle als B$. Daraus folgt aber auch sofort, daß alles, was über dem Inhalt von A$ steht, Stringmüll sein muß, da ja kein Descriptor mehr darauf weist. Also kann der Inhalt von A$ an die obere Speichergrenze geschoben, und der Descriptor von A$ entsprechend geändert werden (Bild 7).

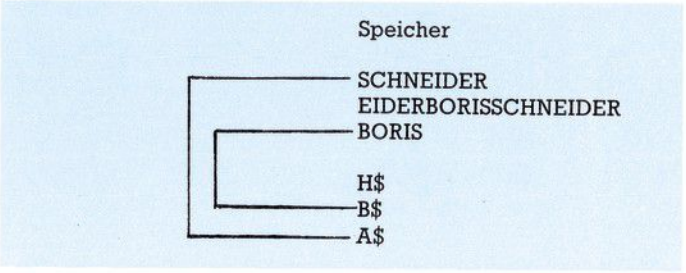

Bild 7. Der erste Schritt der Garbage Collection ist gelaufen, A$ ist aufgeräumt. In der Mitte übriggebliebener Stringmüll.

Die Garbage Collection merkt sich nun noch die Adresse, bei der der Inhalt der zuletzt bearbeiteten Stringvariablen (hier A$) endet, denn alle Stringvariablen, deren Inhalte über dieser Adresse stehen, müssen ja schon von der Garbage Collection behandelt worden sein.

Nun haben wir also A$ schon aufgeräumt, es fehlt uns noch B$. Er wird an die vorhin gemerkte Adresse (Ende von A$) geschoben, und sein Ende wird wieder gemerkt (Bild 8).

Da jetzt alle vorhandenen Stringdescriptoren über diese Grenze hinaus weisen, sind alle Stringvariablen aufgeräumt worden, und die Garbage Collection ist beendet.

Sie werden jetzt vielleicht fragen, wo denn der Haken bei der Garbage Collection liegt, denn sie war ja unwahrscheinlich schnell.

Nun, wir hatten aber auch nur zwei Strings im Speicher, die aufgeräumt werden mußten. Wenn Sie Ihren Computer mal beschäftigen wollen, so probieren Sie folgendes:

```
DIM A$(9000)
FOR I = 0 TO 9000:A$(I)="A":
NEXT I
TI$="000000":?FRE(0):?TI$
```

Bitte tippen Sie das nur, wenn die normale Speicherbelegung eingeschaltet ist, sonst erhalten Sie sofort einen OUT OF MEMORY ERROR.

Und jetzt können Sie erst mal in Ruhe diesen Artikel weiterlesen, denn Ihr Computer kann über eine Stunde lang mit der Garbage Collection beschäftigt sein! Wenn er fertig ist, sagt er Ihnen auch, wie lange er gebraucht hat.

Die Zeit, die die Garbage Collection benötigt, ist proportional zum Quadrat der vorhandenen Stringvariablen. Für die Nicht-Mathematiker unter uns: Die Garbage Collection muß so viele Durchgänge machen, wie Stringvariablen vorhanden sind, da in jedem Durchgang nur eine aufgeräumt wird. In jedem Durchgang muß aber auch jede Stringvariable angeschaut werden, ob sie:

— schon aufgeräumt ist; dann liegt ihr Descriptor über der gemerkten Grenze.
— jetzt aufgeräumt werden soll; dann hat sie den höchsten Descriptor unter der Grenze.
— erst später aufgeräumt wird, dann trifft keiner der beiden obigen Fälle zu.

Also werden soviele Vergleiche benötigt, wie die Anzahl der vorhandenen Stingvariablen im Quadrat, von der Anzahl der Speicherverschiebungen mal abgesehen, denn sie entspricht der Anzahl der Stringvariablen.

Auf einen Nenner gebracht bedeutet das, daß die FRE(0) Routine aus dem DIMA$(9000) Beispiel 9001x9001 = 81 Millionen Vergleiche benötigt!

Diese Zahl stimmt nicht ganz mit der Wirklichkeit überein, da auch andere Faktoren eine Rolle spielen, die Größenordnungen sind aber ziemlich richtig.

Die Zeit der Garbage Collection hängt aber nicht von der Anzahl der Müllstrings ab, da diese ja gar nicht überprüft werden, sondern nur alle Descriptoren.

Es gibt mehrere Möglichkeiten, die Garbage Collection kurz zu halten:

— Benutzen Sie so wenig Strings wie möglich, dann braucht die Garbage Collection weniger Zeit, außerdem muß sie nicht so oft aufgerufen werden, da ja mehr Platz für Müllstrings ist.

— Strings wenn möglich im Programm definieren, und dann nie mehr verändern, dann entsteht weniger Stringmüll

— Weitere Stringmüll verursachende Befehlssequenzen meiden, wie zum Beispiel A$=A$ und ähnliches.

— Sollen Strings vertauscht werden, SWAP-Routine verwenden.

Halt, werden Sie jetzt sagen, was ist denn eine SWAP-Routine?

Um einen der größten Stringmüll-Verursacher, das Vertauschen von Strings, zu entschärfen, habe ich diese Routine geschrieben. Sie vertauscht zwei Strings, ohne daß Stringmüll entsteht.

Sie werden sich wahrscheinlich schon gefragt haben, warum eigentlich beim Vertauschen von Strings nicht einfach die Descriptoren der beiden Stringvariablen ausgetauscht werden. Dann könnte kein Stringmüll entstehen.

Genau das tut die SWAP-Routine. Eine ähnliche ist vielleicht schon in einer Basic-Erweiterung enthalten, die Sie benützen. Wenn nicht, so tippen Sie einfach Listing 1 ab. SWAP ist im Speicher frei verschiebbar, Sie können es also an jede beliebige Adresse legen. Sinnvoll wäre wohl der Kassettenpuffer (Adresse 828) oder der Bereich ab $C000. Das hängt ganz von weiteren Programmen oder Erweiterungen ab, die sich gleichzeitig im Speicher befinden.

Wie Sie die SWAP-Routine anwenden, steht in den REM-Zeilen des Listings, die Arbeitsweise verdeutlichen die Bilder 9 und 10.

Wenn Sie das Thema Stringverarbeitung und neue Stringfunktionen in Maschinensprache mehr interessiert: Darauf werde ich in der nächsten Ausgabe genauer eingehen, ebenso wie auf die genaue Funktionsweise der SWAP-Routine.

(Boris Schneider/gk)

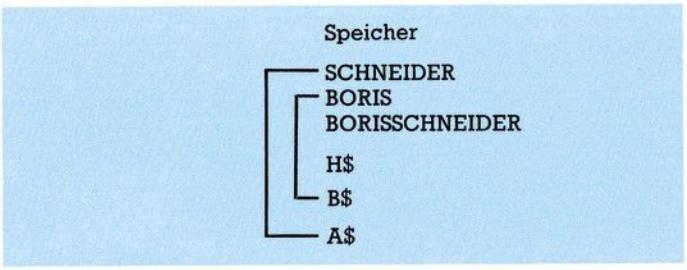

Bild 8. Die Garbage Collection ist beendet, der Restmüll am Ende des Stringspeichers kann einfach überschrieben werden.

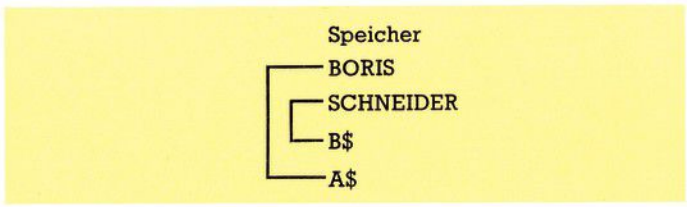

Bild 9. Und noch einmal vertauschen. Vor der Ausführung der SWAP-Routine

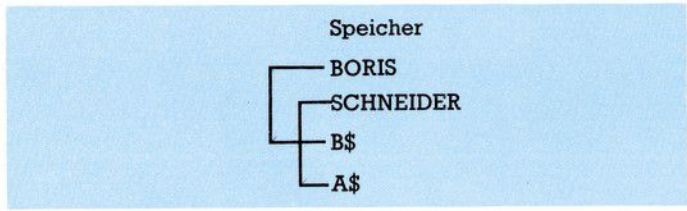

Bild 10. Nach Ausführung der SWAP-Routine.

MEMO MAP

Die Landkarte, der wir auf unserer Reise durch den Speicher folgen, weist diesmal — neben vielen interessanten Plätzen — auch zwei Orte auf, die nützlich sind.

Zum einen ist es die Adresse 19, zum anderen gelangen wir in den Bereich ab Speicherzelle 43, der für Speicher-Manipulationen so eminent und wichtig ist.

Adresse 18 ($12)

1. Flagge für Vorzeichen bei SIN, COS und TAN

2. Flagge bei Vergleich

Zuerst kommt das Vorzeichen der trigonometrischen Funktionen an die Reihe.

Die Routinen des Basic-Übersetzers (Interpreter), welche die drei trigonometrischen Funktionen SIN, COS und TAN berechnen, verwenden die Speicherzelle 18 zur Bestimmung des Vorzeichens.

Zur Erinnerung: Die trigonometrischen Funktionen haben in den vier »Quadranten« des Kreises (0-90, 90-180, 180-270, 270-360 Grad) nicht unbedingt dieselben Vorzeichen. Die Vorzeichen ändern sich allerdings nur an den Grenzen der Quadranten, wie in Bild 1 zu sehen ist. Die Flagge in Zelle 18 gibt das Vorzeichen nicht direkt an, sondern auf Umwegen. Die Darstellung ist in der folgenden Tabelle 1 zusammengefaßt.

WINKEL	0-90	90-180	180-270	270-360
SIN	gleich	Wechsel	Wechsel	gleich
COS	255	255	0	0
TAN	0	255	255	0

Dabei bedeutet »gleich«: 0-0-0-0 oder 255-255-255
»Wechsel«: 0-255-0-255

Da die Erklärung mit »gleich« beziehungsweise »Wechsel« nicht gerade einleuchtend ist, schlage ich vor, daß Sie sich das Ganze mit dem folgenden kleinen Programm selbst anschauen, welches für viele Werte des Winkels im Bogenmaß — und in kleinen Schritten — den Wert der Flagge, daneben den Winkel I und den Wert der Funktion mit Vorzeichen ausdruckt.

```
10 FOR I=0 TO 10 STEP 0.01
20 PRINT PEEK(18);INT(I*100)/
100;SIN(I):NEXT
```

Diese etwas umständliche Art, den Wert von I auszudrucken, vermeidet Rundungsfehler und begrenzt den Ausdruck auf zwei Dezimalstellen. Wenn Sie die Winkelwerte von I in Graden ausgedruckt haben wollen, können Sie eine andere Zeile 20 verwenden, welche die Umrechnungsformel vom Bogenmaß in Grade verwendet:

```
Winkel in Grad = Winkel im Bo-
genmaß * 180/π
20 Print PEEK(18);INT(I*180/π)
;SIN(I):NEXT
```

Statt SIN können Sie genauso gut COS und TAN einsetzen.

In Bild 1 sind nicht nur die Kurven und die Bereiche der Vorzeichen, sondern auch die Winkelbereiche sowohl im Bogenmaß, als auch in Graden dargestellt.

Die Speicherzelle 18 wird auch noch von anderen Routinen des Basic-Interpreters beansprucht und zwar von allen, die einen Vergleich wie <−>,)= und so weiter durchführen. Entsprechend der Art des Vergleichs steht dann in der Zelle 18 eine Ziffer von 0 bis 6.

Das folgende Programm macht das deutlich.

```
10 A=2
20 FOR I=1 TO 3
30 IF I= A THEN PRINT I;
PEEK(18);"="
40 IF I< >A THEN PRINT I;
PEEK(18);"> <"
50 IF I> A THEN PRINT I;
PEEK(18);">"
60 IF I< A THEN PRINT I;
PEEK(18);"<"
70 IF I> =A THEN PRINT I;
PEEK(18);"> ="
80 IF I< =A THEN PRINT I;
PEEK(18);"< ="
```