Stringprogrammierung in Maschinensprache Teil 2

Hier ist ein weiterer Beitrag über effektives Programmieren. Auch er beschäftigt sich mit Strings und enthält einige neue Tips zur Garbage Collection.

Diesmal geht es darum, wie man Stringfunktionen selbst programmieren kann. Hierbei werden wir streng darauf achten, die Müllstrings im Zaum zu halten, um unserem »Erzfeind«, der Garbage Collection, wenig Arbeit zu lassen.

Noch einmal: Garbage Collection

Ich möchte aber vorher noch etwas richtigstellen, was ich in der letzten Ausgabe wohl nicht klar genug ausgedrückt habe. Von vielen Leuten hört man den Vorschlag, ab und zu per FRE(0) eine Garbage Collection auszulösen, um ein Ansammeln von Müllstrings und eine lange Garbage Collection zu vermeiden.

Wer allerdings den Beitrag aus der letzten Ausgabe aufmerksam gelesen hat, wird mir zustimmen: Dies ist absolut falsch! Denn die Dauer der Garbage Collection richtet sich ja hauptsächlich nach der Anzahl der definierten Strings und nicht nach der der Müllstrings. Die von Hand ausgelöste Garbage Collection ist also nur unwesentlich kürzer als die automatisch durchgeführte. Jede von Hand ausgelöste Müllabfuhr ist damit unnötig und kostet nur Zeit! Also: Lieber sehr viel Müll ansammeln lassen und dafür mit möglichst wenig Stringvariablen arbeiten. Eine Ausnahme ist natürlich klar. Sollten Sie zeitkritische Teile in einem Programm haben, in denen Sie sich gar keine Garbage Collection erlauben können, so lohnt es sich, kurz vorher PRINT FRE(0) einzugeben.

```
JSR $AEFA     Klammer auf?
JSR $AD9E     Auswerten eines beliebigen Ausdrucks
JSR $B6A3     Weitere Auswertung für Strings
STX $FB       Stringadresse LO-Byte
STY $FC       Stringadresse HI-Byte
STA $FD       Länge des Strings
JSR $AEFD     Komma?
JSR $B79E     Hole Bytewert in X
TXA
JSR $B47D     Reserviere Speicherplatz für Endstring Adresse
              in $62/$63, Länge in $61
LDY #$00
LBL1
-----------------------------------------------------------
CPY $61       Länge des Endstrings erreicht?
BEQ LBL3
CPY $FD       Startstring komplett kopiert?
BEQ LBL2
LDA($FB),Y    kopieren des Startstrings
STA($62),Y    in den Endstring
INY
BNE LBL1      unbedingter Sprung
LBL2
-----------------------------------------------------------
LDA #$20      Leerzeichen zum Auffüllen
STA($62),Y    Auffüllen
CPY $61       Endstring voll?
BEQ LBL3
INY
BNE LBL2      Unbedingter Sprung
LBL3
-----------------------------------------------------------
JSR $AEFD     Komma?
JSR $B08B     Variable suchen/einrichten
LDX $0D
BEQ LBL5      Wenn kein String dann TYPE MISMATCH
STA $FB       Variablenadresse LO
STY $FC       Variablenadresse HI
LDX #$02
LDY #$02      drei Werte sind zu übertragen
LBL4
-----------------------------------------------------------
LDA $61,X     kopieren des Descriptors von Endstring
STA($FB),Y    in die Stringvariable
DEX
DEY
BPL LBL4
JMP $AEF7     Klammer zu? Rücksprung zu Basic
LBL5
-----------------------------------------------------------
JMP $AD99     »TYPE MISMATCH ERROR«
```

Listing 1. Die FORMAT-Routine (nähere Erklärung im Text).

Problem: Strings auffüllen

Vielleicht haben auch Sie in einigen Programmen diese oder eine ähnliche Zeile entdeckt:

```
170 IF LEN(A$)<40 THEN A$=A$+" ": GOTO 170
```

Die Bedeutung ist klar: Hier soll der String A$ auf 40 Zeichen Länge aufgefüllt werden. Aber schon jetzt müßte es Ihnen eiskalt den Rücken herunterlaufen. Angenommen, A$ hätte zu Anfang 20 Zeichen, dann werden mindestens 20 Müllstrings durch die ständige Zuweisung erzeugt. Sollte diese Auffüllung öfter durchgeführt werden, darf man sich nicht wundern, wenn bald der ganze Speicher voll ist.

Eine weitere Lösung sieht meist so aus:

```
10 H$="40 * space"
170A$=A$+LEFT$(H$,40—LEN(A$))
```

Schon sehr viel besser! Es entstehen zwei Müllstrings (der alte Wert von A$ und einer, der durch die LEFT$-Funktion entsteht und nur Leerzeichen enthält). Es wird außerdem eine Variable H$ benötigt.

Kurz ein paar Worte zur Entstehung des zweiten Müllstrings. Stringfunktionen werden über einen String-Stack abgewickelt. Auf diesem werden die Stringdescriptoren der Zwischenergebnisse bei längeren Stringoperationen abgelegt. In unserem Fall wird zuerst der A$-Descriptor auf diesen String-Stack gelegt, dann der LEFT$-String erstellt und dessen Descriptor abgelegt. Danach erst wird die »+«-Verknüpfung durchgeführt, die einen neuen String erstellt. Die beiden Descriptoren werden vom String-Stack entfernt, und die zugehörigen Strings finden sich als Müll im Speicher wieder. Das Prinzip des String-Stacks ermöglicht eine hierarchische Abarbeitung von Stringfunktionen. Es gilt dabei die Regel:

Erst LEFT$, RIGHT$, MID$, dann + ähnlich dem »Punkt vor Strich« aus der Mathematik.

Deswegen dürfen auch bei Stringoperationen Klammern gesetzt werden.

Wir werden uns im folgenden nicht mehr mit dem String-Stack beschäftigen, weil Sie ihn bei selbstprogrammierten Funktionen wohl nie benötigen werden. Außerdem ist der Umgang mit ihm nicht ganz so einfach, wie es im ersten Augenblick klingt.

Nun aber zu unserem Beispiel. Will man auf der Basic-Ebene ohne PEEK und POKE arbeiten, ist die zweite Lösung die effektivste. Aber ich gebe mich, unersättlich wie ich bin, immer noch nicht zufrieden, denn es geht

1. noch etwas schneller und
2. mit nur einem einzigen Müllstring.

Dazu müssen wir aber auf die Maschinensprachen-Ebene herunter. Werfen Sie mal einen Blick auf Listing 1.

Diese FORMAT-Routine simuliert einen neuen Basic-Befehl. FORMAT füllt einen String mit Leerzeichen, bis eine definierte Länge erreicht ist. Wenn ein String länger ist, wird er abgeschnitten. Der Aufruf muß allerdings über einen SYS-Befehl erfolgen. Steht FORMAT beispielsweise im Kassettenpuffer, so führt

```
CLR: SYS 826 (A$,250,A$)
```

dazu, daß A$ 250 Leerzeichen enthält.

```
A$="HALLO": SYS 826 (A$,10,B$)
```

läßt A$ wie es war, B$ enthält aber »HALLO« und 5 angehängte Leerzeichen, hat also die Länge 10.

```
A$="TEST":N=1:SYS 826 (A$,N,B$(0))
```

hinterläßt in B$(0) ein einsames T, der Rest wird abgeschnitten.

Zusammengefaßt läßt sich also sagen, daß bei der Parameterübergabe:

(String1, N, String2)

in String2 genau N Zeichen aus String1 stehen, und daß gegebenenfalls String2 mit Leerzeichen aufgefüllt wird.

Nur wenn »String1« den gleichen Namen hat wie »String2«, entsteht ein Müllstring, nämlich der alte Inhalt der Stringvariablen. Für die, die es nun gar nicht mehr erwarten können, dieses Programm auszuprobieren, gibt es in Listing 2 einen Basic-Lader. Listing 1 kann direkt mit einem Assembler oder auch mit dem SMON eingegeben werden. Anhand dieses Programms, das alle wichtigen ROM-Routinen, die mit Strings zu tun haben, aufruft, wollen wir nun die Programmierung solcher Routinen erarbeiten.

FORMAT analysiert

Nehmen wir uns erst einmal die grundsätzliche Funktionsweise von FORMAT vor. Ein einfaches Flußdiagramm ist in Bild 1 dargestellt. Dies zeigt aber nur die Verfahrensweise von FORMAT. Die eingebauten Sicherheitsüberprüfungen sind hier nicht enthalten. Bekannt sind am Start der String1, seine Länge 1

```
10 REM    *** FORMAT-ROUTINE ***          <177>
20 REM BRINGT STRINGS AUF DEFINIERTE      <078>
30 REM LAENGEN, FUELLT GGF. AUF.          <171>
40 REM                                    <183>
50 REM                                    <193>
60 REM SYNTAX:                            <236>
70 REM SYS ADRESSE(STARTSTR,N,ENDSTR)     <041>
80 :                                      <138>
90 :                                      <148>
100 REM DIESE ROUTINE IST FREI IM         <047>
110 REM SPEICHER VERSCHIEBLICH !!!        <110>
120 :                                     <178>
130 DATA 032,250,174,032,158,173,032,163  <254>
140 DATA 182,134,251,132,252,133,253,032  <003>
150 DATA 253,174,032,158,183,138,032,125  <027>
160 DATA 180,160,000,196,097,240,022,196  <033>
170 DATA 253,240,007,177,251,145,098,200  <044>
180 DATA 208,241,169,032,145,098,196,097  <076>
190 DATA 240,003,200,208,247,032,253,174  <050>
200 DATA 032,139,176,166,013,240,019,133  <071>
210 DATA 251,132,252,162,002,160,002,181  <062>
220 DATA 097,145,251,202,136,016,248,076  <101>
230 DATA 247,174,076,153,173,000,000,000  <087>
240 :                                     <042>
250 :                                     <052>
260 INPUT "STARTADRESSE";SA               <050>
270 FOR I=SA TO SA+85                     <110>
280 READ A:POKE I,A                       <104>
290 NEXT I                                <238>
300 END                                   <173>
```

Listing 2. Basic-Lader der FORMAT-Routine

und die gewünschte Länge des String2.

Im ersten Schritt werden N Bytes für den String2 reserviert. Sodann werden solange Zeichen vom String1 in den String2 kopiert, bis entweder der String1 komplett kopiert wurde, oder der String2 schon voll ist. Im ersten Fall wird dann in einer zweiten Schleife der String2 mit Leerzeichen aufgefüllt. Ganz zum Schluß wird der Descriptor der zweiten Stringvariablen auf den String2 gerichtet. Das klingt alles ganz einfach, die Realisierung nach diesem Schema ist jedoch etwas umfangreicher.

Parameterübergaben

Sehen wir uns nun die ersten Zeilen des Listing 1 an. Der erste Befehl ist ein Sprung nach $AEFA. Dort steht eine ROM-Routine, die überprüft, ob als nächstes Zeichen ein »(« folgt. Dies ist an sich nicht notwendig, trägt aber erheblich zur Übersichtlichkeit solcher Routinen bei. Fehlt das »(«, so wird SYNTAX ERROR ausgegeben.

Die nächsten zwei Sprungbefehle gehören zusammen. $AD9E wertet einen beliebigen Term, Zahlenrechnung oder String aus und hinterläßt wichtige Parameter für $B79E. Diese Routine prüft, ob der vorherige Term ein String war, und stellt dann im X-Register die LO- und im Y-Register die HI-Adresse des resultierenden Strings, sowie im Akku die Länge des Strings bereit. Diese beiden Sprungbefehle werten auch Ausdrücke wie LEFT$(A$+B$,8) oder ähnliche aus, so daß auch solche Ausdrücke an selbstentwickelte Routinen weitergegeben werden können. Auch können Array-Werte wie A$(14) übergeben werden, ohne daß eine Spezialbehandlung nötig wäre. TYPE MISMATCH und ähnliche Fehlermeldungen werden vollautomatisch ausgegeben.

Die soeben gewonnenen Parameter des ersten Strings werden nun zwischengespeichert. $AEFD prüft auf ein Komma, und verhält sich ansonsten genauso wie $AEFA.

Mit $B79E wird ein Ein-Byte-Wert, das heißt eine Zahl zwischen 0 und 255, in das X-Register geholt. Bei größeren Zahlen wird ILLEGAL QUANTITY angezeigt. Auch hier dürfen wieder Berechnungen oder Variablen stehen.

Damit wäre die Parameterübergabe vorläufig beendet, die weiteren Parameter für String2 besorgen wir uns erst, wenn wir sie tatsächlich brauchen.

Erstellen von String2

Erinnern wir uns noch einmal an den Artikel aus der letzten Ausgabe. Dort wurde gesagt, daß Strings im Speicher von oben nach unten wachsen, während es beim Programm und bei den Variablen genau umgekehrt ist.

An sich müßten wir nun an der unteren Grenze der Strings genügend Bytes für String2 durch Verändern der entsprechenden Pointer herstellen. Es geht aber auch einfacher. $B47D reserviert so viele Bytes an der entsprechenden Stelle, wie ihr im Akku übergeben werden. Es wird sogar, wenn nötig, eine Garbage Collection durchgeführt, und schlimmstenfalls OUT OF MEMORY angezeigt, falls der Speicherplatz nicht reicht. Nach dem Aufruf der Routine $B47D stehen in den Bytes $62/$63 die Startadressen für den neuen String und in $61 dessen Länge, die wir ja festgelegt haben. Wenn Sie genauer hinschauen, bemerken Sie, daß

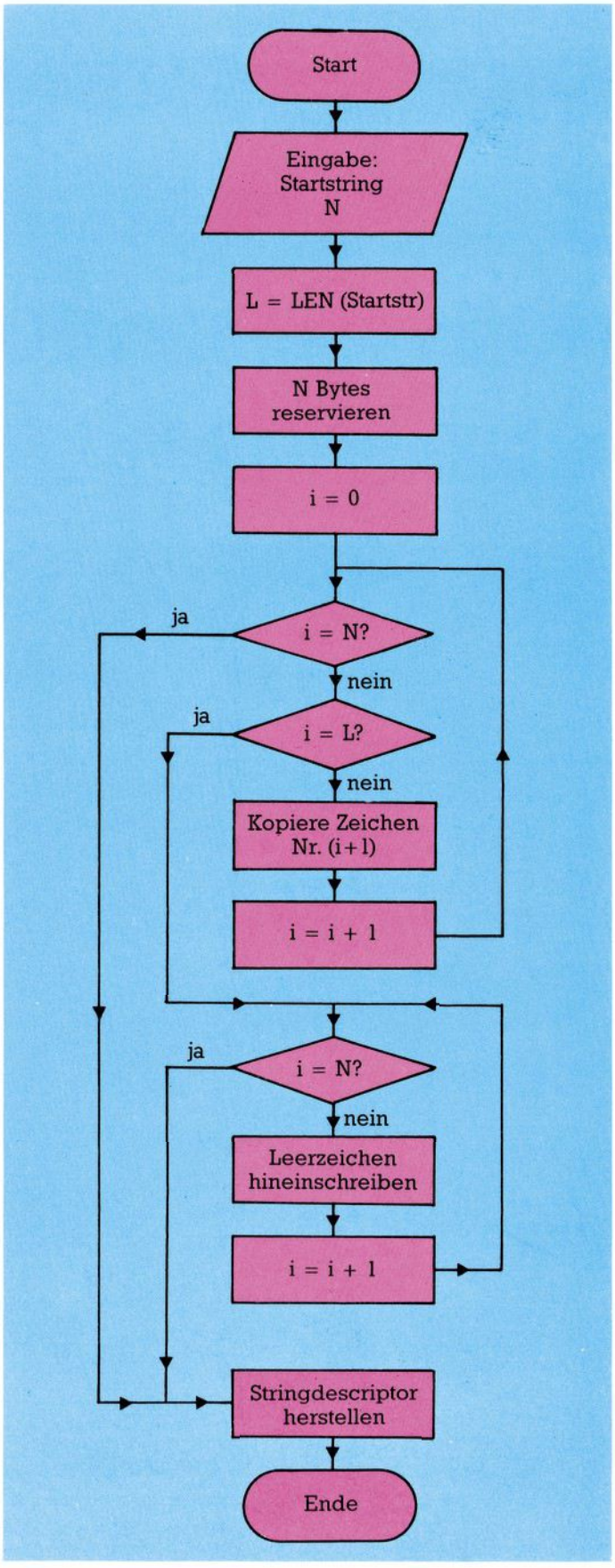

Bild 1. Flußdiagramm der FORMAT-Routine

dies schon der komplette Descriptor ist, der später nur noch in die Stringvariable, die String2 enthalten soll, einkopiert werden muß. Somit ist es recht einfach, String2 zu erstellen, da wir ja jetzt seinen Bestimmungsort kennen und die Länge für spätere Vergleiche zwischengespeichert wurde. Diese Routine ist wohl der Kernpunkt eines jeden Maschinenprogramms, das Strings verwalten soll.

Nun folgen zwei Schleifen, die genauso aufgebaut sind, wie die entsprechenden Teile im Flußdiagramm. Die erste kopiert Zeichen von String1 in String2, die zweite füllt String2 mit Leerzeichen auf.

Sollten Sie die etwas unlogische Struktur oder die beiden unbedingten Sprünge BNE nach den Inkrementanweisungen stören: Dieses Programm durfte, damit es frei verschiebbar ist, keine JMP-Befehle enthalten. Außerdem werden ohne Spezialabfrage sämtliche Sonderfälle (Länge eines oder beider Strings gleich Null) berücksichtigt.

Parameterübergabe II

Nachdem String2 fertiggestellt wurde, folgt der abschließende Teil, der den Descriptor von String2 in die entsprechende Stringvariable hineinkopieren soll.

Doch zuerst wieder der Test auf ein Komma. Die Routine $B08B nimmt die Schlüsselposition in der Übergabe von Strings an den Basic-Text ein. Sie versucht eine Variable im Speicher und gibt deren Adresse im Akku (LO-Byte) und Y-Register (HI-Byte) zurück. Leider funktioniert diese Routine auch mit numerischen Variablen. Die nächsten beiden Befehle fangen diesen Fall ab. In der Zelle $0D steht (handelt es sich um eine Zahl), $00, bei einem String $FF. Haben wir keine Stringvariable vor uns, so müssen wir selber in die Routine springen, die TYPE MISMATCH ERROR ausgibt ($AD 99). Falls eine gesuchte Variable noch nicht existiert, wird sie in der Routine $B08B angelegt.

Anschließend speichern wir in den zwei nun überflüssig gewordenen Speicherstellen $FB/$FC die Adresse der Variablen ab. Dies ist an sich nicht die genaue Adresse der Variablen, sondern, zu unserem Glück, die Adresse, an der der Stringdescriptor stehen muß. Im vorletzten Schritt kopieren wir also den schon lange vorhandenen Descriptor in die entsprechenden Speicherstellen. Somit ist die Arbeit faktisch beendet, und wir prüfen noch, weil's ordentlicher aussieht, ob auch ein »)« vorhanden ist. Dadurch, daß wir mit JMP springen, ersparen wir uns auch noch ein RTS, das ja am Ende der Klammerzu-Betriebssystem-Routine steht.

Ich glaube, daß ich mit diesem Beispiel das angenehme mit dem nützlichen verbunden habe, und hoffe, daß Sie jetzt sowohl ein wenig schlauer, als auch um ein nützliches Programm reicher sind.

So nicht !

Bevor noch einmal alle wichtigen ROM-Routinen zusammengefaßt werden, möchte ich Ihnen ein Beispiel geben, wie man sehr leicht Fehler in eine Stringfunktion einbauen kann, die nicht sofort erkennbar sind. Auch hier ist ein Blick auf die Garbage Collection angebracht.

Wie oben schon erklärt wurde, erzeugt die Zuweisung
B$ = LEFT$(A$,3)
einen zweiten String, der nur die drei ersten Zeichen von A$ enthält und auf den Descriptor von B$ zeigt. Manche Leser mögen jetzt folgende Ideen haben:

Theoretisch ist es ja nicht notwendig, einen zweiten String mit den drei Zeichen zu erstellen. Vielmehr könnte man ja den Descriptor von B$ auf dieselbe Position wie den von A$ zeigen lassen. Im B$-Descriptor wird dann die Stringlänge 3 angegeben, während die in A$ gleich bleibt.

Sodann wäre der Teilstring B$ in A$ enthalten und müßte nicht noch einmal im Speicher stehen. Ähnliches ließe sich auch mit den Routinen erreichen, die dann RIGHT$ und MID$ ersetzen.

Doch vielleicht ahnt der eine oder andere unter Ihnen schon den Haken an dieser Geschichte. Es ist wieder einmal (was sonst) die Garbage Collection, die uns einen Strich durch die Rechnung macht.

Findet, aus welchem Grund auch immer, eine Garbage Collection statt, so passiert folgendes: A$ und B$ zeigen auf denselben Stringdescriptor. Je nachdem welcher String als erster im Programm definiert wurde, wird einer der beiden zuerst aufgeräumt, nehmen wir mal an, es sei B$. Wenn B$ nun aufgeräumt ist, wird sein Inhalt an die aktuelle »Aufräumgrenze« hochkopiert. Diese aktuelle Aufräumgrenze kann allerdings so ungünstig liegen, daß A$ von B$ teilweise überschrieben wird. Im nächsten Aufräumschritt würde A$ nicht an die aktuelle Aufräumgrenze hoch, sondern unter sie herunter kopiert und somit weitere Strings zerstört... Dieses Spielchen würde sich dann bis zum Ende der Garbage Collection hinziehen. (Oder bis einmal tatsächlich Müllstrings an den kritischen Stellen stehen, die ja überschrieben werden dürfen).

Zugegeben, ich habe mir gerade den Worst Case, den allerschlimmsten Fall, herausgesucht. Er zeigt aber ganz deutlich folgende Grundregel, die beim Programmieren von Stringfunktionen beachtet werden soll:
Zwei Stringdescriptoren dürfen niemals auf denselben String zeigen, sonst kann die Garbage Collection Stringinhalte zerstören.

Sie können sich ja spaßeshalber vor Augen führen, was man so alles an Stringsalat produzieren kann, indem Sie mal solch eine Routine programmieren. Mit den oben gegebenen Informationen dürfte das nicht schwerfallen.

Noch fehleranfälliger als die neue LEFT$-Routine wären solche, die RIGHT$ und MID$ ersetzen sollen, weil da nämlich sowohl Länge als auch Adresse im Stringedescriptor unterschiedlich sind. Dann kann eine Garbage Collection zum völligen Chaos führen, es entsteht im wahrsten Sinne des Wortes Stringmüll.

In solchen Fällen muß man, wie in unserer FORMAT-Routine, einfach den Teil des Strings kopieren, der weiterbearbeitet werden soll. Dies machen ja schließlich auch die originalen LEFT$-, RIGHT$- und MID$-Routinen.

Bild 2 enthält noch einmal eine Kurzbeschreibung aller verwendeten Routinen, damit Sie bei der Programmierung von Funktionen nicht immer in der Beschreibung des Beispiels nachsehen müssen.

Mit diesen Routinen läßt sich meiner Ansicht nach jede nur erdenkliche Stringmanipulation durchführen, auch wenn der Aufwand teilweise durch weitere ROM-Routinen eingeschränkt werden kann. Aber das Arbeiten über die Descriptoren der zu bearbeitenden Strings selbst, wie wir es in unseren Beispielen immer gemacht haben, reicht völlig aus. Auch benötigen wir nicht den schon erwähnten String-Stack. Wer trotzdem mehr über die interne Stringverarbeitung des C 64 wissen möchte, der sollte sich ein ROM-Listing zur Hand nehmen und versuchen, die einzelnen Routinen nachzuverfolgen.

So, ich glaube für heute reicht's wirklich. Aber ich entlasse Sie nicht, ohne Ihnen einen Vorgeschmack auf die nächste Ausgabe zu geben. Dort werden wir uns etwas genauer mit Arrays beschäftigen, allerdings nicht gar so ausführlich wie mit den Strings. Im Mittelpunkt steht nämlich das Suchprogramm INTELLISEARCH. Dieses Suchprogramm bietet, neben einem sehr schnellen Suchalgorithmus, viele Besonderheiten, die jede INSTR- oder ähnliche Funktion vor Neid erblassen läßt. Das Ganze natürlich ausführlich dokumentiert.(Boris Schneider/gk)

$AEFA	Prüft aktuelles Zeichen auf »Klammer auf«, sonst SYNTAX ERROR
$AEF7	Prüft auf »Klammer zu«
$AEFD	Prüft auf »Komma«
$AD9A	Vorauswertung eines beliebigen Ausdrucks (auch verschachtelt und verklammert), weitere Auswertung bei Strings immer mit $B6A3
$B6A3	Weitere Auswertung eines Stringausdrucks, nachher: Akku: Länge des Strings, X-Reg: Adresse LO-Byte, Y-Reg: Adresse HI-Byte
$B79E	Holt Byte-Wert, das heißt Zahl zwischen 0 und 255 ins X-Reg, kann auch Ergebnis einer Rechnung oder Funktion sein.
$B47D	Reserviert Speicher für einen String am Ende des Stringspeichers, Anzahl der Zeichen muß vorher im Akku stehen. Nachher stehen in $61: Länge des reservierten Bereichs, $62: Adresse LO-Byte, $63: Adresse HI-Byte des reservierten Bereichs.
$B08B	Holt sich Variablennamen aus Basic-Text, sucht diese Variable im Speicher. Ist sie nicht vorhanden, wird sie automatisch angelegt. Es stehen dann in: $0D: Das Typflag dieser Variablen. (Strings = $FF, Zahl = $00), Akku: LO-Byte, Y-Reg: HI-Byte der Adresse, an der, war es ein String, der Stringdescriptor beginnt.
$AD99	Gibt Fehlermeldung TYPE MISMATCH aus.

Bild 2. Alle im Listing 1 verwendeten Betriebssystem-Routinen