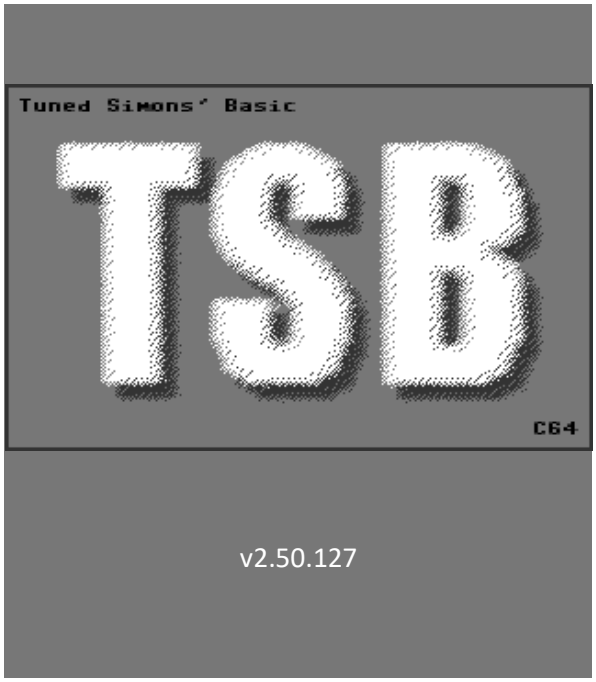


v2.50.127

Handbuch

Arndt Dettke

Inhalt

TSB

Beschreibung

„**TSB**“ ist ein Akronym zu „**T**uned **S**imons' **B**asic“. Es handelt sich dabei um eine „Erweiterung einer Erweiterung“, d.h. die bekannte Basic-Erweiterung „Simons' Basic“ wird durch TSB noch einmal verbessert und erweitert.

Durch eine Eigenheit des Simons’ Basic können die Schlüsselwörter, die für die Basic-**Funktionen** reserviert sind, ein zweites Mal als normale **Befehlswörter** aktiviert werden (und umgekehrt), wovon TSB reichlich Gebrauch macht. Auf diese Weise war es möglich, TSB gegenüber dem Original rund 40 zusätzliche, komplett neue Befehle zu spendieren. Über 30 weitere Befehle/Funktionen wurden mit zusätzlichen, neuen Eigenschaften ausgestattet oder so überarbeitet, dass die in ihnen enthaltenen Fehler ausgemerzt sind.

Bild 1 Ein TSB-Screen mit geändertem Zeichensatz.

Schließlich wurden auch alle wichtigen **Systemroutinen** einer gründlichen Revision und Anpassung unterzogen, so dass die Interpreterschleife, die Eingabewarteschleife, die Routine zur Umwandlung von Schlüsselwörtern in Tokens, die Routine zur Auswertung eines arithmetischen Ausdrucks, die LIST-Routine (siehe auch DELAY), die Befehle LOAD und SAVE und der BASIC-Warmstart (Fehlerkontrolle) nicht mehr mit den entsprechenden Routinen in Simons’ Basic vergleichbar sind.

So ist z.B. das Anhängen einer **Laufwerksnummer** bei den entsprechenden Befehlen in TSB überflüssig. Und: Die Befehle, die einen eigenen **Stack** verwalten (EXEC, LOOP und REPEAT) leisten in TSB wirklich das, was in Simons’ Basic nur versprochen wurde, nämlich jeweils 10 Verschachtelungen zu verkraften, ohne (mit einer Fehlermeldung) in die Knie zu gehen.

Alle TSB-Befehle können bei der Eingabe abgekürzt werden (z.B. *re Shift+N* für RENUMBER), allerdings (und das ist ein kleiner Wermutstropfen) haben TSB-Befehle dabei Vorrang vor den BASIC-V2-Befehlen. *L Shift+O* kürzt in TSB den Befehl LOOP ab (und nicht mehr wie sonst LOAD). Anders als in Simons' Basic können mehrere Aufrufe von Prozeduren in einer einzigen BASIC-Zeile stehen. Außerdem kann das Schlüsselwort EXEC (ähnlich wie der BASIC-V2-Zuweisungsbefehl LET) beim Eingeben des Prozeduraufrufs weggelassen werden, so dass BASIC-Zeilen wie diese möglich sind:

```
100 vorne: mitte: hinten: END
```
oder auch
```
100 IF NOT hindernis THEN vorwaerts
```

(Die Prozedurnamen dürfen BASIC-Schlüsselwörter enthalten wie hier OR (in VORNE und VORWAERTS) und INT (in HINTEN), allerdings nicht als ersten Buchstaben.) Das heißt, dass bei entsprechender Prozedurnamensvergabe fast natürliche Sprache zum Programmieren verwendet werden kann (siehe dazu das zweite Beispiel zu DUP oder das zweite Beispiel zu MOB SET).

Zu den Features von TSB gehört es auch, dass es per BASIC-Befehl zwei weitere BASIC-Erweiterungen einbinden kann. Diese sind das DOS Wedge 5.1 (das danach die JiffyDOS-Abkürzungen zur Verfügung stellt) und die Hochgeschwindigkeits-Grafikerweiterung HSG aus dem 64'er-Sonderheft 6/86) mit neuen, geschwindigkeitsoptimierten Grafikbefehlen, vor allem für das Zeichnen von Kreisen (siehe TSB-Befehl GRAPHICS).

Im Unterschied zu allen anderen C64-Basics startet TSB im Großschrift-/Kleinschrift-Textmodus.

Übersicht über die gegenüber Simons‘ Basic zusätzlichen bzw. geänderten Befehle

In der folgenden Tabelle sind Befehlsworte, die fett gedruckt sind, komplett neu geschriebene Befehle, die andere Funktionen ausführen als die Original-Simons'-Basic-Befehle. Die übrigen wurden um Parameter erweitert oder von ihren internen Fehlern befreit, so dass sie nun erst ihre Funktion wirklich erfüllen.

%%	**$$**	**AT**	CALL	CENTER	**CHECK**	**CLS**	COLD
COLOR	COPY	CSET	**D!**	**D!PEEK**	**D!POKE**	DIR	DISK
DISPLAY	DIV	**DO .. DONE**	**DO NULL**	**DRAW TO**	DUMP	**DUP**	ELSE
ERROR	EXEC	**FETCH**	**GRAPHICS**	**INSERT**	**INST**	JOY	KEY
KEYGET	**LIN**	**MAP**	**MEMCLR**	**MEMCONT**	**MEMDEF**	**MEMLEN**	**MEMLOAD**
MEMOR	**MEMPEEK**	**MEMPOS**	**MEMREAD**	**MEMRES-TORE**	**MEMSAVE**	MERGE	**MOBCOL**
MOB ON/OFF	**MOD**	MOVE	MULTI	NO ERROR	NRM	ON ERROR	ON KEY
OUT	PAGE	**PAINT**	PAUSE	**PLACE**	RENUMBER	RESUME	**RETRACE**
SCRLD	**SCRLD|SV** DEF	**SCRLD|SV** RESTORE	**SCRSV**	**SCRLD|SV** DEF	**SCRLD|SV** RESTORE	**SOUND**	TEXT
TRACE	**USE**	WAVE	**X!**				

Außerdem: Der BASIC-V2-LOAD-Befehl wurde in TSB so erweitert, dass er auch an vorgegebene absolute Adressen laden kann (wie etwa BLOAD in BASIC 7.0). Dazu muss, anders als in BASIC V2, die Sekundäradresse 0 angegeben werden. Ein Aufruf könnte so aussehen:

```
LOAD "dateiname",USE,0,adresse
```

Da die Geräteadresse unter TSB unerheblich ist, kann sie sogar 0 lauten (statt mit USE abgerufen zu werden), womit sich die ungewöhnliche Sekundäradresse 0 vielleicht leichter merken lässt.

Alle Befehle von Simons' Basic gelten auch unter TSB. Unterschiede zwischen einem Simons'-Basic-Befehl und seinem TSB-Pendant werden beschrieben.

Weblinks

TSB Download: http://www.godot64.de/german/downloads.htm
Das *D64-Image* ist rund 99 KByte groß und enthält die BASIC-Datei „README.RUN“, die genauere Informationen zu den obigen Befehlen enthält. Außerdem befindet sich eine ganze Reihe von Beispielprogrammen darauf. TSB steht dort auch als *Magic-Desk-Cartridge-Datei* zur Verfügung (65 KByte).

TSB-Sourcecodes: https://github.com/godot64/TSB
Dies ist so etwas wie ein ROM-Listing von TSB, an vielen Stellen ausführlich kommentiert.

TSB-Tipps: https://www.godot64.de/TSB

Befehle gruppiert nach Anwendungsgebieten

Abfangen von Laufzeitfehlern

ERRLN – ERRN – ERROR – NO ERROR – ON ERROR – OUT – RESUME

Ansprechen von Peripheriegeräten

COPY – DIR – DISK – HRDCPY – JOY – MEMDEF – MEMCONT – MEMLEN – MEMLOAD – MEMOR – MEMPOS – MEMREAD – MEMRESTORE – MEMSAVE – PENX – PENY – POT – SCRLD – SCRSV

Bearbeiten des Textbildschirms

AT – BCKGNDS – BFLASH – CENTER – CLS – COLOR – CSET – DISPLAY – DIV – DOWN – FCHR – FCOL – FILL – FLASH – INSERT – INV – LEFT – LIN – MAP – MOVE – MULTI – NRM – OFF – RIGHT – UP

Bearbeiten und Darstellen von Zahlen

$ – $$ – % – %% – D!PEEK – D!POKE – DIV – EXOR – FRAC – MEMPEEK – MOD – NRM – USE

Bearbeiten von Sprites

@ – CHECK – CMOB – DESIGN – DETECT – MMOB – MOBCOL– MOB ON/OFF – MOB SET – RLOCMOB

Ein- und Ausgabe von Daten

COPY – DIR – DISABLE – DISK – DISPLAY – FETCH – GRAPHICS – HRDCPY – INKEY – INST – JOY – KEY – KEYGET – LIN – MEMCLR – MEMDEF – MEMCONT – MEMLEN – MEMLOAD – MEMOR – MEMPEEK – MEMPOS – MEMREAD – MEMRESTORE – MEMSAVE – MERGE – ON KEY – PENX – POT – RESET – RESUME – SCRLD – SCRSV – SCRLD|SV DEF – SCRLD|SV RESTORE – USE

Geräusche, Töne und Klänge erzeugen

ENVELOPE – MUSIC – PLAY – SOUND (System-Variable) – SOUND (Anweisung) – VOL – WAVE

Grafik-Befehle

ANGL – ARC – BLOCK – CHAR – CIRCLE – CSET – DRAW – DRAW TO – DUP – HI COL – HIRES – LINE – LOW COL – MOD – MULTI – NRM – PAINT – PLOT – REC – ROT – TEST – TEXT

Hilfen zur Programmstrukturierung

CALL – CGOTO – CHECK – DISABLE – DO .. DONE – DO NULL – ELSE – END LOOP – END PROC – EXEC – EXIT – INKEY – KEY – LOOP – ON KEY– PAUSE – PROC – RCOMP – REPEAT – RESUME – UNTIL

Programmierhilfen

AUTO – COLD – D! – D!PEEK – D!POKE – DELAY – DISAPA (SB) – DISPLAY (Variable) – DISPLAY (Befehl) – DUMP – FIND – GLOBAL – GRAPHICS – INKEY – INST – KEY– LOCAL – MEMPEEK – MERGE – OLD – OPTION – OUT – PAGE – PLACE – RENUMBER – RETRACE (TSB) – RETRACE (SB) – SECURE – TRACE

Stringfunktionen

$$ – %% – AT – DUP – INSERT – INST – PLACE

Zeichen ändern

@ – CSET – DESIGN – MEM

→ Inhalt
→ Alle Befehle
→ Alle TSB-Befehle in Übersicht
→ Alle von TSB beeinflussten Adressen
→ Übersicht über die gegenüber Simons‘ Basic zusätzlichen bzw. geänderten Befehle
→ TSB (Einleitung)

Befehl:	$	
Syntax:	**a = $C000** **PRINT $C000**	
Zweck:	Darstellung einer Zahl im Hexadezimalsystem, Ziffern reichen von 0 bis 9 und a bis f, auch zweistellig	Bearbeiten und Darstellen von Zahlen
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Zahlen-Präfix)	

Die Umrechnungsfunktion für Konstanten in Hexadezimaldarstellung zu entsprechenden Dezimalwerten ist in Simons' Basic und TSB als Präfix ausgeführt und kann in jedem numerischen Ausdruck verwendet werden. Der Ausdruck **$** (Dollar) gefolgt von genau **vier oder zwei Zeichen**, die Hexadezimalziffern von **0 bis 9** und **a bis f** enthalten, werden in die entsprechende Dezimalzahl umgerechnet (0 bis 65535).

Sobald eines der vier betrachteten Zeichen nach dem Dollarzeichen (Leerzeichen werden dabei ignoriert) nicht 0..9 und a..f entspricht oder weniger als vier Ziffern vorhanden sind, führt dies zur Fehlermeldung **HEX CHAR ERROR**.

Beispiel:

```
PRINT "DIE ZAHL $1001 ENTSPRICHT DEZIMAL" $1001
```

zeigt:

```
DIE ZAHL $1001 ENTSPRICHT DEZIMAL 4097
```

```
PRINT $ F F F F
```

zeigt trotz der Leerzeichen zwischen den Ziffern das dezimale Ergebnis:

```
 65535
```

In Simons‘ Basic dürfen nur **vierstellige** Hexadezimalzahlen verwendet werden.

Befehl:	$$	
Syntax:	**a$ = $$49152** **PRINT $$49152**	
Zweck:	Umrechnung einer Zahl ins Hexadezimalsystem, liefert einen **String** zurück	Bearbeiten und Darstellen von Zahlen Stringfunktionen
Kürzel	-	
Status:	TSB (Funktion)	

Die Umrechnungsfunktion für Dezimalzahlenausdrücke in Hex-Strings ist in TSB wie ein Präfix ausgeführt. Die Zeichen **$$** (Dollar-Dollar) vor einer vorzeichenlosen Zahl oder einem entsprechenden Ausdruck zwischen 0 und 65535 wandeln diese Zahl/das Ergebnis des Ausdrucks in den entsprechenden Hex-String um.

Ergebnisse im Bytebereich (0..255) sind zweistellig, alle anderen vierstellig. Es wird bei der Umwandlung kein Präfix erzeugt.

Beispiel:

```
10 PRINT "545 + 1328 sind in hex $" $$(545 + 1328)
```

Ergebnis:

```
545 + 1328 sind in hex $0751
```

Hinweis: Mit **a$=$$$c000** wandelt man eine Hexzahl in einen **String** (ohne Präfix) um.

Befehl:	%	
Syntax:	**a = %10000000** **PRINT %10000000**	
Zweck:	Darstellung einer Zahl im Binärsystem, Ziffern reichen von 0 bis 1, auch 16-stellig	Bearbeiten und Darstellen von Zahlen
Kürzel	-	
Status:	Simons‘ Basic (Zahlen-Präfix)	

Die Umrechnungsfunktion für Konstanten in Binärdarstellung zu entsprechenden Dezimalwerten ist in Simons' Basic als Präfix ausgeführt und kann in jedem numerischen Ausdruck verwendet werden. Der Ausdruck **%** (Prozent) gefolgt von **acht Zeichen**, die aus den Binärziffern **0 und 1** bestehen, werden in die entsprechende Dezimalzahl umgerechnet (0 bis 255). Eine 16-bittige Zahl wird nicht unterstützt.

Sobald eines der acht betrachteten Zeichen nach dem Prozentzeichen (an das sich noch beliebige Leerzeichen anschließen können) weder 0 noch 1 entspricht oder weniger als acht Ziffern vorhanden sind, führt dies zur Fehlermeldung **BIN CHAR ERROR**.

Beispiele

```
PRINT "DIE ZAHL %10010101 ENTSPRICHT DEZIMAL" %10010101
```

zeigt

```
DIE ZAHL %10010101 ENTSPRICHT DEZIMAL 149
```

```
PRINT %    1111 0000
```

zeigt trotz der Leerzeichen nach dem Präfix das dezimale Ergebnis

```
 240
```

Hinweis: Mit a$=%%%10000000 wandelt man eine Binärzahl in einen String (ohne Präfix) um.

Befehl:	%%	
Syntax:	**a$ = %%49152** **PRINT %%49152**	
Zweck:	Umrechnung einer Zahl ins Binärsystem, liefert einen **String** zurück	Bearbeiten und Darstellen von Zahlen Stringfunktionen
Kürzel	-	
Status:	TSB (Funktion)	

Die Umrechnungsfunktion für Dezimalzahlenausdrücke in Binärstrings ist in TSB wie ein Präfix ausgeführt. Die Zeichen **%%** (Prozent-Prozent) vor einer vorzeichenlosen Zahl oder einem entsprechenden Ausdruck zwischen 0 und 65535 wandeln diese Zahl/das Ergebnis dieses Ausdrucks um in den entsprechenden Binärstring.

Beispiel:

```
10 PRINT "545 + 1328 sind binaer %" %%(545 + 1328)
```

Ergebnis:

```
545 + 1328 sind binaer %0000111010100001
```

Ergebnisse im Bytebereich sind achtstellige **Strings**, alle anderen 16-stellig. Es wird kein Präfix erzeugt.

Befehl:	@	
Syntax:	@..... (für Sprites) @..... (für Zeichen)	
Zweck:	Mithilfe von @ werden Sprites oder Zeichen definiert	Bearbeiten von Sprites Zeichen ändern
Kürzel	-	
Status:	Erweitertes Simons' Basic (Definitionsdaten-Präfix)	

Der Befehl **@** ist untrennbar mit dem Befehl DESIGN verbunden. Er legt den Beginn einer Definitionszeile des Bitmusters von entweder einem **Sprite** (DESIGN 0, 1 oder 5) oder einem **Zeichensatzzeichen** (DESIGN 2, 3 oder 7) fest, wobei ein ungerader Parameter für die Multicolor-Variante steht, ein gerader für die hochauflösende. Der jeweilige Kontext bestimmt auch, wie viele Definitionszeichen auf das einleitende „@" folgen dürfen (wenn man Leerzeichen verwendet: jedes ist signifikant, außer am Zeilenende, dort ignoriert der Basic-Editor Leerzeichen). Die folgende Tabelle zeigt den Zusammenhang.

	DESIGN 0 (Hires)	**DESIGN 1** (Multi)	**DESIGN 5** (TSB-Multi)	**DESIGN 2** (Hires)	**DESIGN 3** (Multi)	**DESIGN 7** (TSB-Multi)
Breite	24	12	24	8	4	8
Höhe	21			8		

Zulässige Definitionszeichen sind: „A", „B", „C" und „D", sowie die beiden Zeichen „." (Punkt) und Leerzeichen, die beide stellvertretend für „A" gesetzt werden können. Andere Definitionszeichen oder zu wenige/zu viele Zeichen in einer Zeile bzw. fehlende Zeilen lösen die Fehlermeldung **BAD CHAR ERROR** aus. Überflüssige Zeichen in einer Zeile werden schlicht ignoriert, jedoch führt eine Definitionszeile zu viel zum Fehler **SYNTAX ERROR**.

Dabei bedeuten diese Zeichen (bei der **Sprite**-Definition):

Zeichen	bewirkt	aktiviert mit dem Befehl (bei Hires)	aktiviert mit dem Befehl (bei Multicolor)
A . (Punkt) Leerzeichen	Punkt in Hintergrundfarbe (MULTI: Farbquelle 0)	- (Sprite hier transparent, $D021, %00)	- (Sprite hier transparent, $D021, %00)
B	Punkt in Vordergrundfarbe (Multi: Farbquelle 1)	MOB SET ($D027..$D02E) oder MOBCOL	CMOB ($D025, %01)
C	Punkt in Multifarbe 1 (Multi: Farbquelle 2)	-	MOB SET ($D027..$D02E, %10) oder MOBCOL
D	Punkt in Multifarbe 2 (Multi: Farbquelle 3)	-	CMOB ($D026, %11)

Beispiel für DESIGN, MOB SET, MMOB und @:

```
1700 PROC SPRITE
1710   DESIGN 0, 15*64
1720   @BBBBBBBBBBBB............
1721   @BBBBBBBBBBBB............
1722   @BB........BB............
1723   @BB........BB............
1724   @BB........BB............
1725   @BB........BB............
1726   @BB........BB............
1727   @BB........BB............
1728   @BB........BB............
1729   @BB........BB............
1730   @BBBBBBBBBBBB............
1731   @BBBBBBBBBBBB............
1732   @........................
1733   @........................
1734   @........................
1735   @........................
1736   @........................
1737   @........................
1738   @........................
1739   @........................
1740   @........................
1750   MOB SET 1,15,1,0,0    : REM SPRITE DEFINIEREN
1760   S8=38: Z8=80          : REM SPRITE POSITION
1770   MMOB 1,S8,Z8,S8,Z8,0,0: REM SPRITE ANZEIGEN
1780 END PROC
```

(Definiert ein weißes Kästchen als Hires-Sprite mit einem zwei Pixel dicken Rand, in der Mitte bleiben 8x8 Bit frei. Bild siehe Beispiel zu MMOB.)

TSB lässt im Multicolor-Definitionsmodus auch Doppelpixel zu (**DESIGN 5** für Sprites und **DESIGN 7** für Zeichensätze).

Befehl:	ANGL	
Syntax:	**ANGL <x>,<y>,<w>,<rx>,<ry>,<fq>**	
Zweck:	Zeichnen des Radius von Ellipsen (u. Kreisen)	Grafik-Befehle
Kürzel	**aN**	
Status:	Simons' Basic (Anweisung)	

Wenn CIRCLE für Ellipsen und Kreise zuständig ist, freut man sich über den Befehl **ANGL** zum Zeichnen von Radiuslinien in diesen Figuren.

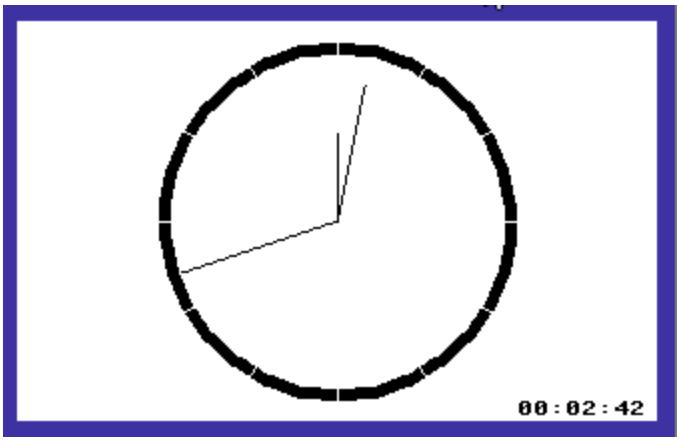


Bild 2 Uhrzeiger mit ANGL

Sie gehen aus vom Mittelpunkt (**<x>,<y>**) einer Ellipse (bzw. eines Kreises), verlaufen in der Richtung **<w>** (mit Winkelangaben in Grad) und reichen bis zur (imaginären) Ellipsen- bzw. Kreislinie mit den Radien **<rx>** und **<ry>** in der Farbe, die durch **<fq>** bestimmt wird (s. dazu HIRES). In x-Richtung dürfen Ellipsen 255 Punkte Radius aufweisen, in y-Richtung ebenfalls.

Punkte, die über den Bildschirmrand hinausgehen würden, werden auf diese höchstmöglichen Koordinaten begrenzt, der Interpreter malt aber weiter. Die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Beachten: Zu große Angaben bei <w> werden nicht abgewiesen, aber dennoch berechnet, allerdings falsch. Wenn für <w> Werte angegeben werden, die über 360 Grad hinausgehen, ergibt sich intern Folgendes: w = 360 + (360 - w), für ein <w> von 370 errechnet der Interpreter also den Wert 350.

```
Beispiel (bitte zuerst die Systemvariable TIME$ bzw. TI$ stellen):

100 HIRES 0,1: mx=160: my=100: h=0: m=0: s=0
110 CIRCLE mx,my,90,90,1: CIRCLE mx,my,85,85,1: PAINT mx+87,my,1
115 FOR w=0 TO 359 STEP 30: ANGL mx,my,w,90,90,0: NEXT
120 REPEAT: GET x$
130 t$=ti$: hh=VAL(LEFT$(t$,2))*30: mm=VAL(MID$(t$,3,2))*6:
    ss=VAL(RIGHT$(t$,2))*6
131 IF hh>=360 THEN hh=hh-360
133 mh=DIV(mm,12): hh=hh+mh: hf=hh<>h: mf=mm<>m: sf=ss<>s
140 zz$=LEFT$(t$,2)+":"+MID$(t$,3,2)+":"+RIGHT$(t$,2)
145 IF t$=ti$ THEN TEXT 250,190,zz$,1,1,8
150 IF sf THEN ANGL mx,my,s,83,83,0: ANGL mx,my,ss,83,83,1: s=ss:
    mf=(s-6)=m
155 RCOMP hf=((s-12)<=h) AND (s>=h)
160 IF mf THEN ANGL mx,my,m,70,70,0: ANGL mx,my,mm,70,70,1: m=mm
165 RCOMP hf=((m-12)<=h) AND (m>=h)
170 IF hf THEN ANGL mx,my,h,45,45,0: ANGL mx,my,hh,45,45,1: h=hh
220 IF t$<>ti$ THEN BLOCK 250,190,250+8*8,190+8,0
230 UNTIL x$>""
```

Kommentar zum ANGL-Beispielprogramm: In Zeile 115 werden die 5-Minutenmarkierungen auf das Zifferblatt aufgetragen. HF, MF und SF (Zeile 133) sind Flags, die dafür sorgen, dass ein Uhrzeiger, der gerade überholt wird, nicht für die nächste Runde verschwindet (da der überholende Zeiger ihn löscht). Probleme macht dabei der Stundenzeiger (Zeile 170), der sich ja im Laufe der Stunde auch bewegt, allerdings anders als der Minuten- (Zeile 160) und Sekundenzeiger (Zeile 150). Daher die Bereichsabfragen hinter RCOMP. Man hätte den Stundenzeiger auch alle 12 Minuten vorrücken können, aber dies hier ist „echter“. Zweimal am Tag verschwindet der Stundenzeiger dennoch: um 2 Minuten vor 12 für 3 Minuten, wegen der Zeilen 155 und 165.

Befehl:	ARC	
Syntax:	**ARC <mx>,<my>,<ws>,<we>,<ww>,<rx>,<ry>,<fq>**	
Zweck:	Zeichnen von Bögen (bzw. ganze Ellipsen oder Kreise)	Grafik-Befehle
Kürzel	**aR**	
Status:	Simons' Basic (Anweisung)	

ARC ist sozusagen die Mutter des CIRCLE-Befehls. Man kann mit ARC auch Ellipsen und Kreise zeichnen, dabei sind aber **alle** erforderlichen Parameter variabel, nicht nur Mittelpunkt (**<mx>,<my>**) und Radien (**<rx>,<ry>**), sondern auch der Startwinkel **<ws>** (bei CIRCLE immer 0), der Endwinkel **<we>** (bei CIRCLE immer 360) und sogar die Winkelschrittweite **<ww>** beim Malen der Figur (bei CIRCLE immer 12, was eigentlich ein regelmäßiges Zwölfeck erzeugt, das auf einem C64-Bildschirm aber hinreichend kreisähnlich aussieht). In x-Richtung dürfen ARC-Bögen 255 Punkte Radius aufweisen, in y-Richtung ebenfalls.

Die Farbe der Punkte hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Das Feature, die Winkelschrittweite für das Zeichnen eines Bogens anzugeben, eröffnet ein paar positive Nebeneffekte, die in Beispiel 2 demonstriert werden: ARC kann damit regelmäßige Vielecke zeichnen, allerdings mit eingeschränkten Gestaltungsmöglichkeiten. So ist bei allen Vielecken der Anfang der Figur oben, senkrecht über dem Mittelpunkt. Dieser Punkt kann nicht verlegt werden. Und ab dem regelmäßigen Zehneck sieht alles wie ein Kreis aus. Das regelmäßige 360-Eck ist jedoch eine schöne (wenn auch sehr langsam gezeichnete) Kreisfigur (siehe Beispiel 2).

Beachten: Zu große Angaben bei den Winkeln werden nicht abgewiesen, aber dennoch falsch berechnet. Wenn für sie Werte angegeben werden, die über 360 Grad hinausgehen, ergibt sich intern Folgendes: w = 360 + (360 - w), für einen Winkel von 370 errechnet der Interpreter also den Wert 350.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **ILLEGAL QUANTITY ERROR**.

Beispiel 1:

```
100 HIRES 0,1
110 FOR i=1 TO 18: w=360/18*i
120 w1=w-35: IF w1<0 THEN w1=w1+360
130 w2=w+195: IF w2>360 THEN w2=w2-360
140 x=160+70*SIN(w*{PI}/180): y=100-
70*COS(w*{PI}/180)
150 ARC x,y,w1,w2,10,30,30,1: NEXT
160 FOR i=1 TO 9: w=2*{PI}*i/9
170 PAINT 160+70*SIN(w),100-
70*COS(w),1: NEXT
200 DO NULL
```

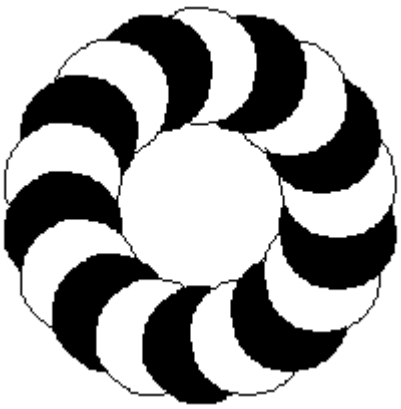


Bild 3 "Überlappende" Kreise

(erzeugt die abgebildete Grafik, der Ausdruck {PI} muss vom PETSCII-Zeichen für PI ersetzt werden)

Beispiel 2:

```
100 HIRES 1,0: mx=160: my=100: rx=25: ry=25
110 ARC 50,my,0,360,120,rx,ry,1
112 ARC 50,my,0,360,120,rx-8,ry-8,1
115 PAINT 50,my,1
120 ARC 105,my,0,360,90,rx,ry,1
122 ARC 105,my,0,360,90,rx-5,ry-5,1
125 PAINT 105,my,1
130 ARC mx,my,0,360,72,rx,ry,1
132 ARC mx,my,0,360,72,rx-5,ry-5,1
135 PAINT mx,my,1
140 ARC 215,my,0,360,60,rx,ry,1
142 ARC 215,my,0,360,60,rx-5,ry-5,1
145 PAINT 215,my,1
150 ARC 270,my,0,360,45,rx,ry,1
152 ARC 270,my,0,360,45,rx-5,ry-5,1
155 PAINT 270,my,1
160 ARC mx,my,0,360,1,150,50,1
162 ARC mx,my,0,360,1,145,45,1
165 PAINT 0,0,1: PAINT mx,my-40,1
900 DO NULL
```

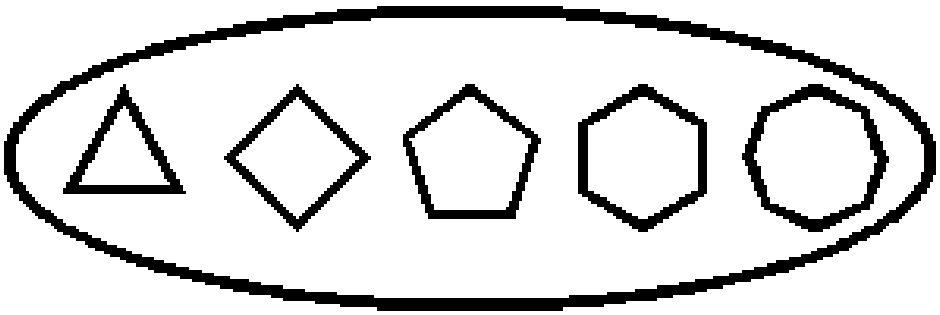


Bild 4 Ellipse, Dreieck, Viereck, Fünfeck, Sechseck, Achteck

Befehl:	AT	1
Syntax:	**AT(string1, string2)**	
Zweck:	Austausch des Inhalts zweier String-Variablen	Stringfunktionen
Kürzel	**aT** (im Kürzel enthalten ist die öffnende Klammer)	
Status:	Neuer TSB-Befehl	

Mit diesem Befehl tauscht man zwei Strings miteinander aus, ohne dass dabei Stringmüll entsteht, der irgendwann zur Garbage Collection führt. **AT** arbeitet sehr schnell selbst bei langen und vielen Strings, da intern keinerlei Kopiertätigkeit bei den String-Inhalten anfällt. Es werden nur die String-Adressen vertauscht.

Beispiel ohne AT:

```
10 a$="Samuel": b$="Beckett": c$=""
20 PRINT a$, b$
30 c$=a$: a$=b$: b$=c$: c$=""
40 PRINT a$, b$
```

Beispiel mit AT:

```
10 a$="Samuel": b$="Beckett"
20 PRINT a$, b$
30 AT(a$,b$)
40 PRINT a$, b$
```

(Ergebnis in beiden Fällen: "Samuel Beckett", "Beckett Samuel")

Beachten: Die sich öffnende Klammer hinter dem eigentlichen Schlüsselwort ist intern noch Teil des Schlüsselworts, so dass eine Auseinanderschreibung an dieser Stelle zu einem **BAD SUBSCRIPT ERROR** führt (der Interpreter vermutet dann die Verwendung einer Feld-Variablen namens AT).

Befehl:	AT	2
Syntax:	**PRINT AT(<zl>,<sp>) <Ausdruck> [;]**	
Zweck:	gezieltes Setzen des Cursors	Bearbeiten des Textbildschirms
Kürzel	**aT** (im Kürzel enthalten ist die öffnende Klammer)	
Status:	Erweitertes Simons‘ Basic (Anweisung), geändert seit v2.40.131	

AT ist eine vom BASIC-V2-Befehl PRINT abhängige Anweisung, die den Cursor an eine definierte Stelle (Zeile **<zl>** und Spalte **<sp>**) auf dem Bildschirm positioniert. Der Parameter **<Ausdruck>** darf nicht weggelassen werden (ggf. Leerstring setzen). Ein **Semikolon** bewirkt das Gleiche wie bei PRINT allein.

Auch den Befehlen CENTER, FETCH und USE kann **AT** mitgegeben werden.

Beachten: Die sich öffnende Klammer hinter dem eigentlichen Schlüsselwort ist intern noch Teil des Schlüsselworts, so dass eine Auseinanderschreibung an dieser Stelle zu einem **BAD SUBSCRIPT ERROR** führt (der Interpreter vermutet dann die Verwendung einer Feld-Variablen namens AT).

Beachten bei Simons' Basic: Die Bildschirmgrenzen werden bei AT von Simons' Basic nicht überwacht. Das kann vor allem bei zu hoher Zeilenangabe (größer als 24) zur Zerstörung des Basic-Programms im Speicher führen, da der Programmspeicher direkt an den Bildschirmspeicher anschließt. Die Parameter sind in SB andersherum angeordnet (erst SP, dann ZL). In TSB sind die Fehler behoben.

Befehl:	AUTO	
Syntax:	**AUTO [<start>,<step>]**	
Zweck:	Automatische Vorgabe von Zeilennummern beim Programmieren	Programmierhilfen
Kürzel	**aU**	
Status:	Erweitertes Simons' Basic (Kommando)	

Mit **AUTO** kann man sich das mitunter lästige Tippen von Zeilennummern beim Programmieren ersparen, der Interpreter übernimmt diese Aufgabe. Will man AUTO beenden, muss man einfach direkt hinter einer Vorgabezeilennummer <RETURN> drücken.

Bei Überschreiten der höchstmöglichen BASIC-Zeilennummer (63999) erscheint ein **SYNTAX ERROR**. Der Interpreter zählt zwar weiter, aber übernimmt keine weitere Zeile mehr. Man sollte die Eingabe abbrechen (und evtl. ein RENUMBER durchführen). Hat man die Startzeile versehentlich falsch angegeben (und es droht der Verlust bereits vorhandener Zeilen) drückt man <Shift +RETURN> hinter einer Zeilennummer, was den AUTO-Modus ebenfalls beendet.

Während des Editierens kann man auf andere Zeilen wechseln, nach <RETURN> dort übernimmt der Interpreter die dann aktuelle Zeilennummer auch für AUTO.

In TSB kann man die beiden Parameter weglassen. Es werden dann **100** für die Startzeile und **10** für die Schrittweite vorgegeben.

Befehl:	BCKGNDS	
Syntax:	**BCKGNDS [128 +] <b1>, <b2>, <b3>, <b4>**	
Zweck:	Färben eines ECM-Textbildschirms	Bearbeiten des Textbildschirms
Kürzel	**bC**	
Status:	Erweitertes Simons' Basic (Anweisung)	

BCKGNDS aktiviert den so genannten *Extended-Background-Color-Modus* des C64 und setzt die dafür erforderlichen Farbinformationen in die VIC-Register $D021 (**<b1>**), $D022 (**<b2>**) $D023 (**<b3>**) und $D024 (**<b4>**) ein. Jedes Zeichen auf dem Bildschirm kann nun eine aus diesen vier verschiedenen Hintergrundfarben erhalten. Mit diesem Befehl und einem vielleicht extra für diese Zwecke erstellten Zeichensatz (siehe Bild 5 und MEM) kann man wunderschöne Benutzeroberflächen gestalten, da sich auch die Farbe als Informationsträger einsetzen lässt.

Nachteil des Extended-Background-Color-Modus ist, dass die Anzahl verschieden aussehender Zeichen auf ein Viertel reduziert ist, da die oberen zwei Bits der Zeichencodierung für die Zuordnung zu den vier Hintergrundfarben Verwendung finden. Damit ist Groß-/Kleinschreibung ausgeschlossen.

Da das generelle Hintergrundfarbregister $D021 bei **BCKGNDS** auch betroffen ist, hat der Befehl COLOR nach **BCKGNDS** konkurrierende Wirkung (er beeinflusst ebenfalls $D021). Man kann z.B. COLOR einsetzen, um die globale Hintergrundfarbe zu ändern, ohne nochmals **BCKGNDS** bemühen zu müssen (COLOR 16,<b1>).

NRM schaltet den Extended-Background-Color-Modus wieder aus.

Wird ein Parameter bei der Eingabe weggelassen, erscheint die Fehlermeldung **SYNTAX ERROR**, ein falscher Parameterwert bleibt unbeachtet und wird behandelt als wäre ein *Wert AND 15* vorgenommen worden.

Hinweis:

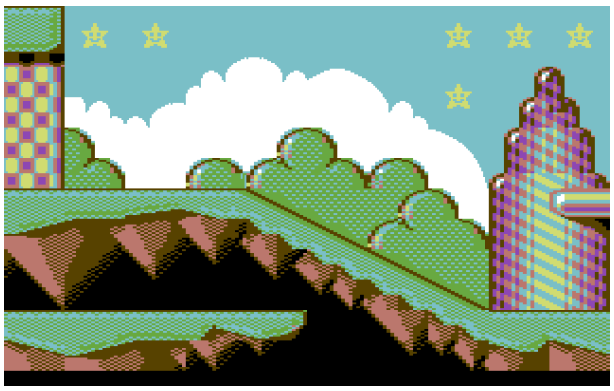


Bild 5 Multicolor-Text-Modus, Bild von TSB-Programm erzeugt

In TSB kann **BCKGNDS** darüber hinaus dazu verwendet werden, die **Farben im MULTI** zu setzen, ohne zusätzlich auch den ECM einzuschalten. Dazu muss zum ersten Parameter <b1> lediglich die Zahl 128 (oder: $80) addiert werden. Der vierte Parameter spielt in diesem Fall keine Rolle und kann beliebig beschickt (aber nicht weggelassen!) werden:
`BCKGNDS 128+3,10,9,x` für Zeichen in hellrot (**10**) und braun (**9**) auf einem Cyan-Hintergrund (**3**). Damit sind (bei Veränderung des Zeichensatzes mit MEM) Bildschirme möglich wie in Bild 5.

Befehl:	BFLASH	
Syntax:	**BFLASH <sp>, <f1>, <f2>** **BFLASH 0 | ON | OFF**	
Zweck:	Blinken des Bildschirmrahmens	Bearbeiten des Textbildschirms
Kürzel	**bF**	
Status:	Erweitertes Simons' Basic (Anweisung)	

BFLASH dient dazu, den Rahmen des Bildschirms in einer gewünschten Geschwindigkeit blinken zu lassen. Der erste Parameter (**<sp>**) legt die Geschwindigkeit fest, wobei die eingegebene Zahl die Anzahl von sechzigstel Sekunden (Jiffys) für jede Phase angibt. Bei einem Wert von 30 für <sp> wechselt der Rahmen also zwei Mal pro Sekunde die Farbe. Die beiden weiteren Parameter definieren genau die zwei Farben, zwischen denen der Interpreter bei Ausführung des Befehls hin- und herwechselt.

BFLASH 0 beendet in Simons' Basic das Blinken. In TSB kann man auch **BFLASH OFF** schreiben. TSB stellt nach BFLASH 0/OFF die ursprüngliche Rahmenfarbe wieder her.

Beachten bei Simons' Basic: Die Ausführung des Befehls findet im Interrupt statt, das Programm läuft in dieser Zeit weiter. Der Programmierer hat währenddessen keinen Einfluss mehr auf BFLASH. Das Ende des Befehlslaufs ist in Simons' Basic nicht synchronisiert, die zuletzt angezeigte Rahmenfarbe hängt dort daher vom Moment des Ausführens von BFLASH 0 ab.

Wenn ein Programm vorzeitig abbricht (<RUN/STOP>-Taste gedrückt oder Laufzeitfehler), muss das Blinken von Hand mit BFLASH 0/OFF ausgeschaltet werden, da der Interpreter es im Direktmodus weiterlaufen lässt.

Nach BFLASH 0/OFF darf kein weiterer Parameter folgen, sonst meldet der Interpreter einen **SYNTAX ERROR** und bricht das Programm ab. Der Wert 0 als Zeitangabe ist wegen der besonderen Bedeutung von BFLASH 0 von Basic aus (anders als beim Befehl FLASH) unzulässig. Farbangaben größer als 15 werden akzeptiert, aber intern wie *Wert AND 15* behandelt.

Hinweis: TSB akzeptiert auch **BFLASH ON** (und verwendet dabei die zuletzt vorgenommenen Einstellungen; wurden vorher keine definiert, verwendet es für alle drei Parameter den Vorgabewert 0, färbt also den Rahmen schwarz und wechselt etwa alle vier Sekunden (256 Jiffys) auf ebenfalls schwarz).

Tipp: Um Zeitspannen zu bemessen, die kürzer sind als eine Sekunde, verwendet man die Speicherstelle **$C516**. Sie wird von (T)SB 60-mal in der Sekunde erhöht, zählt also 60tel Sekunden. Will man z.B. zur Aufmerksamkeitssteigerung den Bildschirmrahmen kurz flackern lassen, geht man so vor:

Beispiel:

```
1000 BFLASH 5,2,11
1010 POKE $C516,0: REPEAT: UNTIL PEEK($C516)>15: BFLASH OFF
```

Lässt den Rahmen für etwa eine Viertelsekunde mit schneller Frequenz zwischen Rot und Dunkelgrau blinken.

Befehl:	BLOCK	
Syntax:	**BLOCK <x1>,<y1>,<x2>,<y2>,<fq>**	
Zweck:	Zeichnen eines ausgefüllten Rechtecks	Grafik-Befehle
Kürzel	**bL**	
Status:	Simons' Basic (Anweisung)	

BLOCK zeichnet ein ausgefülltes Rechteck. Der Ort der linken oberen Ecke wird durch die beiden ersten Parameter **<x1>** und **<y1>** bestimmt, die Koordinaten der rechten unteren Ecke des Rechtecks legen Parameter drei und vier fest (**<x2>** und **<y2>**, wobei x2=x1+breite und y2=y1+hoehe, wichtig bei REC, s. Beispiel). Die Farbe des Rechtecks wird durch den letzten Parameter (**<fq>**, Farbquelle) bestimmt. LOW COL hat keine Auswirkungen auf BLOCK.

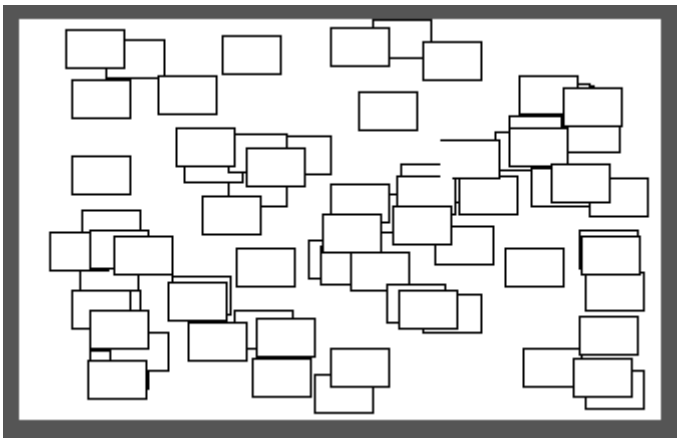


Bild 6 "Papierblätter" - BLOCK (Fläche) und REC (Rand) gemeinsam

Zulässige Werte sind 0..319 für <x1> und <x2> (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für <y1> bzw. <y2> sind in beiden Fällen Werte von 0 bis 199 erlaubt. Auch die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Beachten: Leider stimmen die Parametertypen von REC und BLOCK nicht überein, was das Programmieren ein wenig umständlicher macht (s. Programmbeispiel).

Beispiel:

```
100 HIRES 0,1
105 REPEAT: GET x$
110 x=290*RND(1): y=180*RND(1)
120 BLOCK x,y,x+29,y+19,0
130 REC x,y,29,19,1
140 UNTIL x$>"": PAINT 0,0,1
150 DO NULL
```

(erzeugt die abgebildete Animation Bild 6)

(Beispiel übernommen und angepasst aus dem Buch „Spiele mit Computergrafik“.)

Befehl:	CALL	
Syntax:	**CALL <label>**	
Zweck:	Aufrufen einer Prozedur	Struktur
Kürzel	**cA**	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

In Simons' Basic können einzelne Zeilen und Unterprogramme mit einem Namen versehen werden (siehe PROC). Sie werden dadurch unabhängig von ihrer Lage im Programm und der Programmierer kann leichter den Überblick bewahren (Namen lassen sich leichter einem Zweck zuordnen als Zeilennummern). Der Befehl **CALL** springt zu einem solchen Label. Er entspricht damit weitgehend dem BASIC-V2-Befehl GOTO.

In Simons' Basic dürfen in der Zeile, in der CALL verwendet wird, keine weiteren Basic-Befehle eingegeben werden (sie werden dann als zum Label gehörig betrachtet). Ein Leerzeichen am Anfang eines Labels (die meistens gesetzte Lücke zwischen PROC und dem Labelnamen wie im Beispiel unten) darf in Simons‘ Basic bei CALL nicht weggelassen werden.

In TSB wurden beide Mängel behoben, nach dem Befehl `CALL label` kann nach einem Doppelpunkt ähnlich wie bei REM beliebiger Kommentar folgen (am besten – wie auch bei REM – nach einem führenden Anführungszeichen, damit nicht etwa Großbuchstaben bei LIST als Basic-Token fehlinterpretiert werden), und das Leerzeichen vor dem Label wird nicht beachtet.

Wird das Label hinter CALL im Programm nicht gefunden, so erscheint die Fehlermeldung **NO PROC ERROR**. Wenn der Interpreter nach CALL im Hauptprogramm auf ein Prozedurende (END PROC) trifft, erscheint die Meldung **END PROC W/O EXEC ERROR**. Ausnahme: Eine Prozedur wurde mit EXEC aufgerufen und CALL wechselt aus der aktuellen Prozedur zu einer anderen.

Befehl:	CENTER	
Syntax:	**CENTER [AT(zl,sp)] <string> | <strvar> [, <breite>]**	
Zweck:	zentrierte Ausgabe eines Strings	Bearbeiten des Textbildschirms
Kürzel	**cE**	
Status:	Erweitertes Simons' Basic (Anweisung)	

CENTER (bis v2.40.425: CENTRE) gibt Strings oder den Inhalt von Stringvariablen zentriert auf dem Bildschirm aus. **Der Cursor bleibt hinter der Ausgabe stehen** (wie bei PRINT mit Semikolon).

Wenn nicht durch das **optionale AT(zl,sp)** anders bestimmt, erfolgt die Ausgabe auf der Bildschirmzeile, auf der der Cursor sich gerade befindet, und bezieht sich ohne den Parameter <breite> auf die volle Zeilenbreite von 40 Zeichen.

Hinweis: Wenn der Cursor nicht an Spalte 0, sondern irgendwo innerhalb der Bildschirmzeile positioniert ist, wird die Restbreite (40 minus POS(0)) als Zentrierungsgrundlage verwendet.

Der optionale Parameter **<breite>** legt fest, welcher Wert als Grundlage für die Zentrierung gelten soll. Die Einrückung wird also bezüglich dieser Angabe vorgenommen (Formel: (<breite> minus LEN(<string>)) durch 2).

Ist das Ergebnis der internen Einrückungsberechnung 0, negativ oder kleiner als LEN(<string>) plus 2, wird nicht eingerückt. Der Wert <breite> kann größer sein als 40 (maximal 255), wobei ab einem Berechnungsergebnis von LEN(<string>) plus 128 (ergibt intern einen negativen Bytewert) keine Einrückung mehr erfolgt. Ist der erste Parameter kein Ausdruck vom Typ String, kommt es zur Fehlermeldung **TYPE MISMATCH ERROR**. Fehlt der erste Parameter oder sind mehr als zwei angegeben, führt das zum Abbruch mit **SYNTAX ERROR**. Ist der zweite Parameter kleiner als 0 oder größer als 255, gibt es einen **ILLEGAL QUANTITY ERROR**.

Beispiel:

```
10 CLS: CENTER "demo"
20 PRINT AT(2,0) DUP("{shift-*}",40)
30 PRINT AT(4,0) "Stop by C=" AT(5,0) "Slow by CTRL"
   AT(6,0) "Break by STOP"
40 PRINT AT(8,0) DUP("{shift-*}", 40)
50 FOR i=1 to 5: PRINT AT(10+I,4)"{shift--}" AT(10+I,14)"{shift--}"
   AT(10+I,24)"{shift--}" AT(10+I,34)"{shift--}"
60 NEXT
70 FOR i=0 to 2
80 CENTER AT(11,5+i*10) "Nr."+str$(i+1),9
90 NEXT: PRINT AT(20,0)"";
```

Gibt eine zentrierte Gesamtüberschrift (Zeile 10), ein paar Infozeilen (Zeilen 20 bis 40) und eine (leere) Tabelle (Zeilen 50 und 60) mit zentrierten Überschriften aus (speziell in Zeile 80)

Befehl:	CGOTO	
Syntax:	CGOTO <ausdruck>	
Zweck:	Berechnetes GOTO	Struktur
Kürzel	cG	
Status:	Simons' Basic (Anweisung)	

CGOTO ist ein Befehl, der lange Sprungtabellen hinter dem BASIC-Befehl ON ersetzt. Statt die Sprungziele wie bei ON vorzugeben und dann durch Angabe der Platznummer des gewünschten Ziels aufzurufen, kann hinter CGOTO ohne große Tipperei sofort die gewünschte Zielzeile durch eine Berechnung angepeilt werden. Die Verwendung von CGOTO setzt voraus, dass beim Programmieren die Zeilen so angeordnet sind, dass für den Algorithmus **<ausdruck>** hinter CGOTO diese Zeilen auch erreichbar bleiben.

Es gibt kein entsprechendes CGOSUB.

Da Simons' Basic ohnehin darauf ausgerichtet ist, ohne Zeilennummern auszukommen (s. EXEC und PROC), stellt sich der Nutzen von CGOTO als eher zweifelhaft dar.

Beachten: Nach einem RENUMBER muss der Algorithmus hinter CGOTO angepasst werden, sonst funktioniert das Programm nicht mehr (wenn der Algorithmus im CGOTO-Ausdruck sich überhaupt anpassen lässt!)

Beispiel:

```
(ohne CGOTO) ...
50 i4=1: GOSUB 1500: IF ri=0 THEN 50
60 IF ri=1 THEN GOSUB 1300: i7=i7+1: i8=1: i4=2: IF i7>mn THEN i7=1
70 IF ri=2 THEN GOSUB 1300: i7=i7-1: i8=1: i4=2: IF i7=0 THEN i7=mn
80 IF ri=3 THEN i8=i8+1: IF i8>ho THEN i8=1
90 IF ri=4 THEN i8=i8-1: IF i8=0 THEN i8=ho
100 IF ri=5 THEN i6=br: GOSUB 1300: GOSUB 1700: GOSUB 2000: zl=1:
br=i6: GOTO 30
110 ON i4 GOSUB 1610, 1600: GOTO 50
999 END
...
```

(Dies ist eine **Menü-Navigationskontrolle**. In Zeile 50 werden die Pfeiltasten abgefragt, die Routine liefert in ri die angesteuerte Richtung zurück. In Zeilen 60 bis 90 wird darauf reagiert. Die Variable i4 steuert dabei das Öffnen eines von zwei Menükästen in den Zeilen 1600 und 1610 (in Zeile 110 nach ON). Zeile 100 ist die Reaktion auf die <RETURN>-Taste. Im Folgenden eine CGOTO-Version von Zeile 110:)

```
(mit CGOTO)...
110 CGOTO (i4-1)*10+1600
...
1620 GOTO 50
```

(Diese Änderung hätte die gleiche Wirkung, ist aber kürzer. Eine echte CGOTO-Version hätte in i4 bereits die Zeilennummernschrittweite vorgesehen und wäre dadurch noch kürzer (CGOTO 1600+i4). Die Unterroutinen enden hier mit GOTO!)

Befehl:	CHAR	
Syntax:	**CHAR <x>,<y>,<bc>,<fq>,<zm>**	
Zweck:	Schreiben einzelner Zeichen im Grafikmodus	Grafik-Befehle
Kürzel	**cH**	
Status:	Simons' Basic (Anweisung)	

CHAR schreibt ein einzelnes Bildschirmcode-Zeichen **<bc>** in der Farbe der angegebenen Farbquelle **<fq>** (s. dazu HIRES) in den Grafikbildschirm. Zulässige Werte für **<x>** sind 0..319 (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für **<y>** sind in beiden Fällen Werte von 0 bis 199 erlaubt. Der Punkt 0,0 ist in der linken oberen Ecke.

Leider wirkt sich der Zoom (**<zm>**) nur auf die Höhe der Zeichen aus, nicht jedoch auf deren Breite. Zooms größer als 5 wirken daher bereits unansehnlich (vgl. dazu DUP).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

```
Beispiel: 100 HIRES 1,0: bb=-1: vv=.8
          105 a$="forum64.ist.super!":
              s=LEN(a$): PRINT "{clr/home}"a$:
              sc=DISPLAY
          110 CIRCLE 160,100,100,100,1:
              PAINT 160,100,1
          113 LOOP: LOOP
          115     bb=bb+.008/COS(bb):
                  ll=ll+.2/COS(bb)
          116     BLOCK 0,0,8,8,0
          120     ch=PEEK(sc+a): a=a+1:
                  IF a=s THEN a=0
          130     CHAR 0,0,ch,1,1
          140     FOR i=0 TO 8 STEP .5: FOR j=0 TO 8 STEP .5
          145         IF TEST(I,8-j)=1 THEN DO
          150           b=bb+.02*j: l=ll+.02/COS(bb)*i
          160           x=SIN(l)*COS(b)
          170           y=COS(l)*COS(b)
          180           z=SIN(b)
          190           u=y*COS(vv)+z*SIN(vv)
          200           v=-y*SIN(vv)+z*COS(vv)
          205           EXIT IF u<0 OR bb>1.4
          210           PLOT 160+100*x,100-100*v,0
          215         DONE
          220       NEXT : NEXT
          225   END LOOP : EXIT IF bb>1.35
          226 END LOOP
          230 BLOCK 0,0,8,8,0
          235 CIRCLE 160,100,100,100,1
          250 DO NULL
```

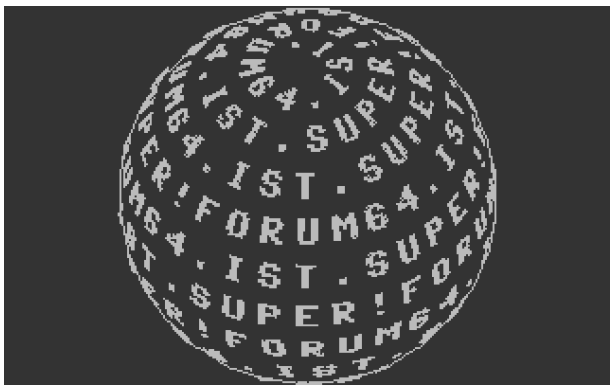


Bild 7 CHAR im Einsatz

Zur Verwendung eines anderen als dem vorgegebenen ROM-Zeichensatz s. den Hinweis bei TEXT.

Befehl:	CHECK	1
Syntax:	**CHECK**	
Zweck:	Beschleunigen von Prozeduraufrufen	Struktur
Kürzel	**chE**	
Status:	Neuer TSB-Befehl	

In TSB können genau wie in Simons' Basic Unterprogramme mit einem Namen (Label) versehen werden (siehe PROC). Sie werden dadurch unabhängig von ihrer Lage im Programm und der Programmierer kann leichter den Überblick bewahren (Namen lassen sich leichter einem Zweck zuordnen als Zeilennummern).

Intern durchsucht der Interpreter bei jedem PROC-Aufruf mittels EXEC oder CALL den ganzen Programm-Code nach dem Label hinter dem aufrufenden Befehl und wechselt bei erfolgreicher Suche dorthin. Diese Suche dauert länger als bei den vergleichbaren Befehlen GOTO und GOSUB, da nicht nur die kurzen (zweibytigen) Zeilennummern überprüft werden, sondern alle Zeichen eines Labels.

TSB beschleunigt diesen Vorgang dadurch, dass es sich einmal aufgerufene Prozeduren in einer Liste (an Adresse $C400) merkt und bei nachfolgenden Aufrufen zuerst diese Liste durchsucht. Im Lauf des Programms wird dessen Arbeitsgeschwindigkeit also immer größer.

Der Befehl **CHECK** legt diese Liste nun sofort und vollständig (im Speicherbereich $C400 bis $C4FF) an, sodass alle folgenden Prozeduraufrufe stark beschleunigt werden. CHECK sollte man deshalb gleich am Programmanfang oder in einer Programm-Initialisierungsroutine einsetzen. Die interne CHECK-Liste kann maximal 128 Einträge aufnehmen (zwei Bytes pro Prozedur). Weist ein Programm mehr als 128 PROC-Anweisungen auf, werden die überzähligen ohne Fehlermeldung ignoriert und auf herkömmliche Weise (langsam) gesucht.

Hinweis: Der Bereich $C440 bis $CFFF (die Spriteblöcke 17 bis 19) ist für Sprite-Definitionen (vgl. Sprites) dennoch geeignet, wenn die Anzahl der verwendeten PROCs das zulässt (32 PROCs je Spriteblock).

Befehl:	CHECK	2
Syntax:	x = CHECK(<n>) x = CHECK(<n1>,<n2>)	
Zweck:	Sprite kontrollieren	Bearbeiten von Sprites
Kürzel	chE	
Status:	Simons' Basic (boolesche Funktion)	

CHECK überprüft Sprite-Kollisionen, die mit DETECT vorbereitet wurden. Die Funktion liefert dabei entweder den Wert 1 (keine Kollision, FALSE) oder 0 zurück (gesuchte Kollision hat sich ereignet, TRUE).

CHECK(**<n>**) (nur ein Argument: die Nummer des Sprites, 0..7) ist dabei für die Kollisionsabfrage Sprite-Hintergrund zuständig, CHECK (**<n1>**, **<n2>**) (mit zwei Argumenten) für diejenige zwischen zwei Sprites.

Will man herausfinden, ob irgendeine Sprite-Sprite-Kollision mit einem bestimmten Sprite stattgefunden hat (egal, mit welchem Sprite), schreibt man das gleiche Argument (die gesuchte Sprite-Nummer) zweimal in die Klammer, z.B.: `k = CHECK(1,1)`, bedeutet: Hatte Sprite 1 eine Kollision?

Achtung: In Simons' Basic werden die Nummern der Sprites nicht auf Plausibilität geprüft, so dass Werte über 7 (bis 255) akzeptiert werden, die aber zu völlig falschen Ergebnissen führen (in TSB behoben).

Je nach Wahl des Parameters von DETECT liefert die Funktion CHECK einen **SYNTAX ERROR**, wenn die Anzahl der Argumente nicht stimmt (DETECT 0 erwartet CHECK(<n1>,<n2>), DETECT 1 dagegen CHECK(<n>).)

Beachten: Vor jeder Kollisionsabfrage sollte eine Leerabfrage mit DETECT erfolgen, damit die Register und Variablen initialisiert sind (die VIC-Kollisionsregister werden beim Lesezugriff gelöscht).

Beispiele:

```
100 DETECT 0: k1 = CHECK(1,4)
```
Haben sich Sprites 1 und 4 berührt?

```
110 DETECT 0: k2 = CHECK(1,1)
```
Wurde Sprite 1 von irgendeinem anderen berührt?

```
120 DETECT 1: k3 = CHECK(2)
```
Hat Sprite 2 den Hintergrund berührt?

Damit eine Kollision sicher erkannt wird, sollte DETECT mit dem Rasterstrahl synchronisiert werden (in Basic wegen Bit 9 des Rasterstrahls in $D011 zu langsam!):

```
1100 proc bump
1110 repeat:x=peek($d011)and$80:until:repeat:x=peek($d012):until>55:detect0
1120 k4=check(s1,s2)
1130 end proc
```

Befehl:	CIRCLE	
Syntax:	**CIRCLE <x>,<y>,<rx>,<ry>,<fq>**	
Zweck:	Zeichnen von Ellipsen (u. Kreisen)	Grafik-Befehle
Kürzel	**cI**	
Status:	Simons' Basic (Anweisung)	

Mit **CIRCLE** zeichnet man Ellipsen bzw. Kreise (Sonderfall der Ellipse) um einen Mittelpunkt (**<x>**,**<y>**) mit den Radien **<rx>** und **<ry>** in der Farbe, die durch **<fq>** bestimmt wird (s. dazu HIRES).

In x-Richtung darf eine Ellipse 255 Punkte Radius aufweisen, in y-Richtung ebenfalls. Punkte, die über den Bildschirmrand hinausgehen würden, werden auf diese höchstmöglichen Koordinaten begrenzt, der Interpreter malt aber weiter. Die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **ILLEGAL QUANTITY ERROR**.

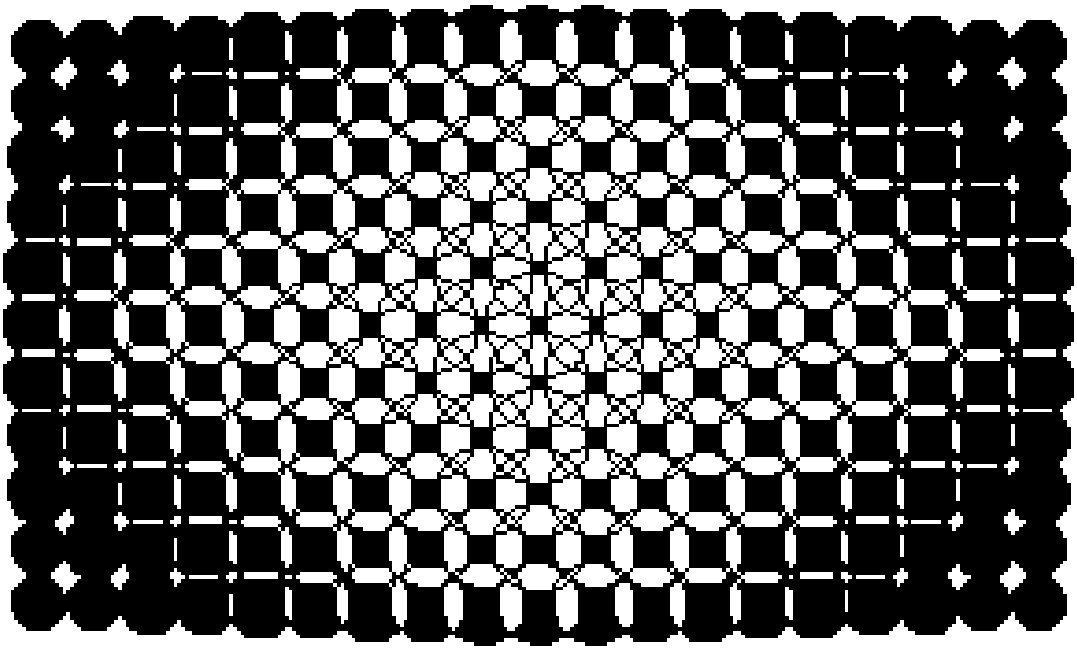


Bild 8 Kreise, teilweise gefüllt.

Befehl:	CLS	
Syntax:	CLS	
Zweck:	löscht den Textbildschirm	Bearbeiten des Textbildschirms
Kürzel	cL	
Status:	Neuer TSB-Befehl (Kommando)	

CLS löscht den Textbildschirm. Der Cursor steht danach in der linken oberen Ecke an Position 0,0. CLS ist eine Kurzform für die Anweisung `PRINT CHR$(147);` (mit Semikolon).

Eine andere Schreibweise für CLS ist (die in TSB veraltete Form) DIV.

Befehl:	CMOB	
Syntax:	**CMOB <f1>, <f2>**	
Zweck:	Färben eines Multicolorsprites	Bearbeiten von Sprites
Kürzel	**cM**	
Status:	Simons‘ Basic (Anweisung)	

Mit **CMOB** setzt man die beiden für alle Multicolor-Sprites einheitlichen Farben in VIC-Register $D025 (**<f1>**, Bitmuster %01) und $D026 (**<f2>**, Bitmuster %11). Die dritte, für jedes Sprite individuelle Farbe erhält es mit dem Befehl MOB SET (Register $D027 und folgende, Bitmuster %10) oder auch MOBCOL. Die vierte Farbe schließlich ist die Bildschirmhintergrundfarbe, die mit COLOR bzw. FCOL gesetzt wird (Bitmuster %00).

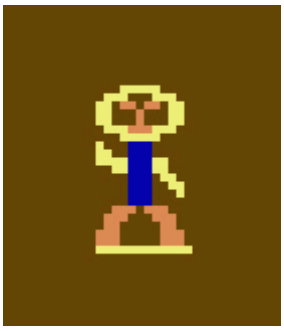


Bild 9 Sprites färben mit CMOB

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, ein falscher Wert (größer als 15) wird behandelt, als ob er mit 15 geANDet wurde (wie in Simons‘ Basic bei Farben üblich).

Die bei CMOB festgelegten Farben werden bei der Sprite-Definition (mit Sprites und @) den Definitionsbuchstaben **B** und **D** zugewiesen.

Der Buchstabe C erhält seine Farben aus dem MOB SET-Befehl oder vom Befehl MOBCOL. Leider hat David Simons an dieser Stelle offenbar nicht gut genug nachgedacht, sonst hätte er die Verwirrung bei den Farbdefinitionsbuchstaben sicher vermieden, daher:

Beachten: Die Zuordnung der Buchstaben B und C bei den Befehlen MOB SET, DESIGN und @ ändert sich je nach Darstellungsmodus des Sprites.

Beispiel:

```
1700 PROC msprite
1710   DESIGN 1, 14*64
1720   @...bbbbb....
1721   @.bbb...bbb..
1722   @bb.cc.cc.bb.
1723   @bb...c...bb.
1724   @bb...c...bb.
1725   @.bbbcccbbb..
1726   @...bbbbb....
1727   @b...ddd.....
1728   @bb..ddd.....
1729   @bbbbdddbb...
1730   @..bbdddbb...
1731   @....ddd.bb..
1732   @....ddd..bb.
1733   @....ddd...b.
1734   @....ddd.....
1735   @..ccc.ccc...
1736   @.ccc...ccc..
1737   @.cc.....cc..
1738   @ccc.....ccc.
```

```
1739   @ccc.....ccc.
1740   @bbbbbbbbbbbb
1750   COLOR 9: MOB SET 2,14,8,0,1: CMOB 7,6: s8=300: z8=205:
       MMOB 2,0,0,s8,z8,2,200
1760 END PROC
```

Gelb (7) im Bild 9 ist die CMOB-Farbe Nr. 1, blau (6) die Nr. 2. Orange (8) kommt aus dem MOB SET-Befehl (dort Parameter Nr. 3). Der Hintergrund ist braun (9 in COLOR).

(entnommen aus dem „Trainingsbuch zum Simon's Basic“)

Befehl:	COLD	
Syntax:	COLD	
Zweck:	Reset des Interpreters	Programmierhilfen
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Kommando)	

COLD bewirkt unter Simons' Basic (und auch unter TSB) dasselbe wie die Eingabe von **SYS 64738**: Der Interpreter wird in den Startzustand zurückversetzt („Kaltstart“).

Im Einzelnen führt der Interpreter folgende Einstellungen durch:

- Das BASIC-ROM wird (in Speicherstelle $01) aktiviert.
- Die Standard-BASIC-Vektoren werden nach $0300 geladen.
- Das RAM wird für BASIC initialisiert (alle wichtigen Zeiger werden gesetzt und CHRGET aktiviert).
- Die Bildschirmfarben (bei TSB: Rahmen dunkelgrau, Hintergrund mittelgrau und Cursor schwarz) werden gesetzt.
- Die Flag- und Pufferbereiche ab $C400 werden gelöscht (bei TSB erheblich weniger umfangreich).

- Die interpretertypischen BASIC-Vektoren (ab $0300) werden angepasst (bei TSB auch LOAD- und SAVE-Vektor).
- Die (erste Zeile der) Einschaltmeldung wird ausgegeben.
- Der freie BASIC-Speicher wird festgelegt.
- Die BYTES-FREE-Meldung wird ausgegeben.
- Die NMI- und BRK-Vektoren werden so eingestellt, dass der Interpreter automatisch den Grafikmodus beenden kann.
- Der Stackpointer wird initialisiert.
- TSB setzt zusätzlich noch das Vorgabelaufwerk auf Laufwerk 8 zurück.
- „ tuned“ wird ausgegeben (dritte Zeile der Einschaltmeldung).

Befehl:	COLOR	
Syntax:	**COLOR {<border> [[,<backgr>]} [,<cursor>]]**	
Zweck:	Färben des Textbildschirms und des Cursors	Bearbeiten des Textbildschirms
Kürzel	**cO**	
Status:	Erweitertes Simons' Basic (Anweisung)	

Die Anweisung **COLOR** (bis v2.40.425: COLOUR) ersetzt die leidigen POKEs für die Bildschirmfarben des normalen Textmodus (53280, 53281 und 646). Wer für den Grafikmodus Hintergrund- und Rahmenfarbe ändern will, muss ebenfalls COLOR verwenden. Anders als bei Simons' Basic gibt es in TSB **vier** Syntax-Varianten:

- `COLOR 12,15,1`
  hiermit färbt man alle drei: Rahmen, Hintergrund und Cursor
- `COLOR 12,15`
  die ursprüngliche Simons'-Basic-Syntax färbt Rahmen (**<border>**) und Hintergrund (**<backgr>**)
- `COLOR 15`
  färbt nur den Rahmen (Parameter **<border>**)
- `COLOR ,1 (Sonderformat)`
  färbt nur den Cursor (hier in die Farbe Weiß, Parameter **<cursor>**)

Ein Parameter außerhalb des Byte-Bereichs führt zur Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Hinweis: Ein Wert über 15 lässt den bereits an dieser Parameterposition eingetragenen Wert weiter bestehen, d.h. ein `COLOR 16,2` ändert nur die Hintergrundfarbe, nicht aber den Rahmen.

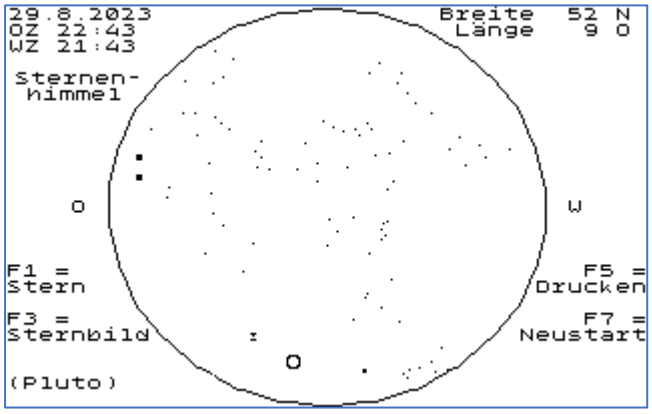
Befehl:	COPY	
Syntax:	COPY	
Zweck:	Ausgabe der hochauflösenden Grafik auf Drucker MPS 801 (und kompatible)	Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Kommando)	

COPY druckt die im Speicher vorhandene Bitmap auf einem angeschlossenen Drucker **MPS 801** (oder kompatibel) aus. Der Befehl benutzt dazu intern ein OPEN mit der laufenden Dateinummer 101 (`OPEN 101,4:CMD 101`) und schließt diesen Kanal auch wieder.

In TSB wird der Ausdruck von COPY horizontal zentriert auf dem Blatt Papier ausgegeben.

Hinweis: Um die Ausgabeposition zu ändern, kann man an die Werte an den Adressen **$8ADB** und **$8ADC** (bis v2.31.113: $BAB9 und $BABA) überschreiben, Vorgabe ist „51“, rückwärts für **15** (Anzahl Zeichen à 6 Pixel Einrückung, also 90 Pixel Einrückung).

Simons' Basic druckt ohne einen linken Rand, so dass Ausdrucke nicht wirklich beeindruckend aussehen.



Beispiel: Ausdruck mit **COPY**

Befehl:	CSET	
Syntax:	**CSET <wert>**	
Zweck:	Ändern der Anzeigemodi	Bearbeiten des Textbildschirms Grafik-Befehle
Kürzel	**cS**	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

CSET schaltet mit **<wert>** um zwischen den beiden Zeichensätzen des C64: Großschrift-Grafikzeichen (`CSET 0`) und Groß-Klein (`CSET 1`).

Außerdem kann man hiermit in den Grafikmodus schalten, ohne den Grafikspeicher dabei zu löschen (s. HIRES), dazu verwendet man `CSET 2`.

Anders als in Simons' Basic wirkt CSET in TSB auch nach der Anwendung des Befehls MEM (in Simons' Basic „vergisst“ der Interpreter bei CSET die Änderungen durch den MEM-Befehl).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, ein falscher Wert (größer als 2) wirkt wie `CSET 2`. Werte außerhalb des Byte-Bereichs erzeugen einen **ILLEGAL QUANTITY ERROR**.

Befehl:	D!	
Syntax:	**D! [<z1>] [-] [<z2>]**	
Zweck:	löscht Teile eines Programms	Programmierhilfen
Kürzel	-	
Status:	Neuer TSB-Befehl	

Mit **D!** hat der Programmierer die Möglichkeit, mehrere Zeilen eines im Speicher befindlichen Programms auf einmal zu löschen.

Die verschiedenen Syntaxformen haben folgende Ergebnisse:

`D! 100`
löscht eine einzelne Zeile

`D! -100`
löscht alle Zeilen bis einschl. Zeile 100

`D! 100-`
löscht alle Zeilen ab Zeile 100

`D! 100-200`
löscht von Zeile 100 bis einschließlich 200

Wird der Befehl ohne Parameter eingegeben, meldet der Interpreter einen **BAD MODE ERROR**.

Beispiel:

```
D! 2670-2890
```

(löscht in der Datei "tsb demo" auf der TSB-Diskette den Teil mit der TSB-Befehlsliste und der Shortcut-Prioritätsanzeige)

Befehl:	D!PEEK	
Syntax:	**a = D!PEEK(<adr>)** **PRINT D!PEEK(<adr>)**	
Zweck:	Auslesen eines 16-Bit-Wertes aus zwei aufeinanderfolgenden C64-Speicherstellen	Programmierhilfen
Kürzel	**d!peE**	
Status:	Neuer TSB-Befehl (numerische Funktion)	

Mit der numerischen Funktion **D!PEEK()** kann ein beliebiges Speicheradressenpaar von 0 bis 65534 ausgelesen werden. Die Funktion liest den Byte-Wert der angegebenen Adresse als Low-Byte, den Byte-Wert der darauffolgenden Speicherstelle als Hi-Byte und berechnet daraus als Ergebnis eine vorzeichenlose Zahl im Word-Bereich (0 bis 65535).

Generell sollte als Rückgabe-Variable (hier: a) keine Ganzzahlvariable (Integer) verwendet werden, da diese keine Werte größer als 32767 aufnehmen kann.

Nicht ganzzahlige Werte im Funktionsargument werden implizit analog zur Funktion INT in eine entsprechende Ganzzahl "gerundet". Liegt die eingegebene Speicheradresse nicht im Word-Bereich, erscheint die Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Beispiel:

```
PRINT D!PEEK(43)
```

(Gibt die Adresse des Anfangs des Basic-Speichers an. Er liegt üblicherweise bei Speicheradresse 2049.)

Befehl:	D!POKE	
Syntax:	**D!POKE <adr>, <wert>**	
Zweck:	Schreiben eines 16-Bit-Wertes in zwei aufeinanderfolgende C64-Speicherstellen (lo/hi)	Programmierhilfen
Kürzel	**d!poK**	
Status:	Neuer TSB-Befehl	

Mit D!POKE kann ein beliebiges Speicheradressenpaar von 0 bis 65534 beschrieben werden. Der Befehl teilt den 16-Bit-Wert <wert> in Low- und Hi-Byte und schreibt ihn an der angegebenen und der darauffolgenden Speicherstelle in den C64-Speicher (von 0 bis 65534).

Nicht ganzzahlige Werte in den Befehlsparametern werden implizit analog zur Funktion INT in eine entsprechende Ganzzahl "gerundet". Liegt die eingegebene Speicheradresse nicht im Word-Bereich, erscheint die Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Beispiel:

```
D!POKE 55,$7000
```

(Setzt den Vektor für das Ende des Basic-Speichers auf $7000. Er liegt in **TSB** üblicherweise bei Speicheradresse $8000.)

Befehl:	DELAY	
Syntax:	**DELAY <n>** **DELAY ON | OFF**	
Zweck:	Verzögern der LIST-Ausgabe	Programmierhilfen
Kürzel	**dE**	
Status:	Erweitertes Simons' Basic (Kommando)	

Mit **DELAY** kann man die Ausgabe der Zeichen bei LIST verlangsamen, wenn während des LIST-Vorgangs die **CTRL-Taste** gedrückt wird. Ein Wert von **10** für <n> entspricht dabei ungefähr der Verzögerung, die sich auch durch das Verwenden der CTRL-Taste beim LISTen in Basic V2 ergibt.

In TSB kann man mit der Shift-Taste das LISTen gänzlich anhalten, aber mit zusätzlich C= unverzögert fortsetzen. In Simons' Basic hält die C=-Taste dagegen das LISTen an. <STOP> bricht LIST ab.

Wenn in Simons' Basic der DELAY-Befehl vor dem ersten Aufruf nie verwendet wurde, ist automatisch die höchste Verzögerungsstufe aktiv. In TSB ist DELAY dagegen dann ausgeschaltet.

In TSB kann man zum Abschalten der LIST-Verzögerung auch DELAY OFF eingeben. DELAY ON schaltet mit einem Vorgabewert von 10 wieder ein. Die Kombination von Shift (LISTen anhalten) mit CTRL (LISTen in Einzelzeichen) und ab und zu C= (LIST normal fortfahren) ist bei einem Wert von 30 bis 40 für <n> nicht mehr so hektisch.

Befehl:	DESIGN	
Syntax:	**DESIGN <n>, <ad> [beliebiger nichtnumerischer Kommentar]**	
Zweck:	Definiert Typ und Speicherort eines Sprites oder Zeichens	Bearbeiten von Sprites Zeichen ändern
Kürzel	**deS**	
Status:	Erweitertes Simons' Basic (Kommando)	

Der Befehl **DESIGN** dient zwei unterschiedlichen Zwecken: Man kann damit 1. den Typ und den Speicherort von Sprites festlegen, aber auch 2. den Speicherort eines bestimmten Zeichens in einem Zeichensatz anwählen. In jedem Fall muss in den auf diesen Befehl unmittelbar folgenden Zeilen die Definition des Aussehens des Sprites oder Zeichens folgen. Definitionen werden mit dem Zeichen @ eingeleitet.

DESIGN und @ sind untrennbar miteinander verknüpft. Das eine ohne das andere führt zu **SYNTAX ERROR** (keine DESIGN-Zeile oder zu viele @-Zeilen) bzw. **TOO FEW LINES ERROR** (keine oder zu wenige @-Zeilen) oder **BAD CHAR ERROR** (falsche Definitionszeichen). Auf *<ad>* darf beliebiger Kommentar folgen.

---

1. Definition eines Sprites (DESIGN 0, DESIGN 1 und DESIGN 5)

Lautet der Wert des ersten DESIGN-Parameters (**<n>**) 0, 1 oder 5, dann befindet man sich als Programmierer in der „Sprite-Abteilung" des Befehls. Der Wert **0** legt fest, dass ein hochauflösendes Sprite aus 24×21 Pixeln definiert werden soll (Hires). Eine **1** oder **5** als Parameterwert teilt dem Interpreter dagegen mit, dass man ein Multicolor-Sprite aus 12×21 Doppelpixeln erzeugen möchte.

Der zweite Parameter (**<ad>**) selektiert den Ort im Speicher des C64, an dem die Pixel der @-Definitionszeilen abgelegt werden sollen (Anzahl: 21×3 = 63 Bytes). Welche Speicherbereiche dies sein können, hängt davon ab, welche Speicherbank der Videochip des C64 (VIC) gerade anzeigt. Dies können unter TSB standardmäßig zwei Bereiche sein: Bank 0 (Textmodus) von $0000 bis $3FFF und Bank 3 (Grafikmodus bzw. Textmodus mit eigenem Zeichensatz) von $C000 bis $FFFF. Die Adresse eines solchen Definitionsblocks errechnet sich nach folgender Formel: **ad = Bankbasisadresse + 64 * Block**. Für ein Sprite in Block 19 im Grafikmodus käme dann heraus: ad = $C000 + 64 * 19 = $C4C0.

Block	Adresse	dezimal
11	$02C0	704
13	$0340	832
14	$0380	896
15	$03C0	960
32	$0800	2048
..	+ $40	+ 64
255	$3FC0	16320

Die freien Speicherbereiche (Blöcke) in Bank 0 sind (fast) die gleichen wie auch im C64 ohne Basic-Erweiterung.

Achtung: alle Blöcke ab Block 32 liegen im Bereich des im Speicher befindlichen Basic-Programms! Sollen hier Sprites abgelegt werden, muss man die Adresse des Basic-Anfangs hochlegen.

Im TSB-Grafikmodus sieht die Verteilung etwas anders aus:

Block	Adresse	dezimal
16	$C400	50176
..	+ $40	+ 64
19	$C4C0	53184
20	$C500	50432
..	+ $40	+ 64
63	$CFC0	65344
253	$FF40	50176
..	+ $40	+ 64
255	$FFC0	65472

TSB verwendet den Bereich von $C400 bis $C4FF für CALL und EXEC. Wenn PROCs im Programm vorkommen, steht aufsteigend immer weniger Platz für Sprites zur Verfügung, also ist hier Vorsicht geboten (die Blöcke 17, 18 und 19 sind nur eingeschränkt verfügbar, je nach Anzahl der PROCs im Programm, Block 16 ist für Sprites tabu).

Die Blöcke 20 bis 63 dürfen in TSB **auf keinen Fall** verwendet werden, da dann Systemvariablen oder Code gelöscht würden. Ausnahme: Block 43, wenn LOCAL im Programm nicht verwendet wird.

Wenn man mit MEM einen neuen Zeichensatz aktiviert, sieht der Plan der freien Speicherblöcke wiederum anders aus:

Block	Adresse	dezimal
0	$C000	49152
..	+ $40	+ 64
16	$C400	50176
..	+ $40	+ 64
19	$C4C0	50368
20	$C500	50432
..	+ $40	+ 64
191	$EFC0	61376
..	+ $40	+ 64
255	$FFC0	65472

Auch hier trifft die Einschränkung im Bereich $C400 bis $C4FF (Blöcke 17 bis 19) unter TSB zu. Block 16 ist für Sprites immer tabu. Dasselbe gilt für die Blöcke 20 bis 63, hier würden Systemvariable oder sogar Code gelöscht. Die Blöcke 64 bis 127 liegen im IO-Bereich und sind ebenfalls nicht zugänglich. Und wenn man die Blöcke 128 bis 190 verwendet, wird womöglich der gerade in Benutzung befindliche Zeichensatz beeinträchtigt (es können aber evtl. freie Bereiche darin verwendet werden, z.B. wenn man nur den Groß-Klein-Zeichensatzbereich braucht).

Man kann auch im MEM-Modus die Grafik einschalten (und sieht dann oben im Bild den Zeichensatz): `CSET 2: MEM`. Die Farben der Grafik kommen dann nicht mehr aus dem Bereich ab $C000 und sondern berufen sich jetzt auf den Bereich ab $CC00 (dem Textscreen im MEM-Modus). Man kann sie aber mit `FCHR 0,0,40,25,Code` ändern. In Variable `Code` bestimmt das High-Nibble die Hintergrund- und das Low-Nibble die Schreibfarbe, für weiß auf schwarz lautet der Code also $01. Der bisherige Inhalt des Textbildschirms ist danach allerdings verloren.

Beispiel 1:

```
; definiert ein hochauflösendes Sprite für den Simons'-Basic-Textmodus
; die Zeile 1750 legt weitere Eigenschaften des Sprites fest und macht es schließlich
; sichtbar

1700 PROC sprite
1710   DESIGN 0, 15*64
1720   @bbbbbbbbbbbb............
1721   @bbbbbbbbbbbb............
1722   @bb........bb............
1723   @bb........bb............
1724   @bb........bb............
1725   @bb........bb............
```

```
1726   @bb........bb............
1727   @bb........bb............
1728   @bb........bb............
1729   @bb........bb............
1730   @bbbbbbbbbbbb............
1731   @bbbbbbbbbbbb............
1732   @........................
1733   @........................
1734   @........................
1735   @........................
1736   @........................
1737   @........................
1738   @........................
1739   @........................
1740   @........................
1750   MOB SET 1,15,1,0,0: s8=38: z8=80: MMOB 1,s8,z8,s8,z8,0,0
1760 END PROC
```

Ein komplettes Anwendungsbeispiel beim Simons-Basic-Befehl MOB SET (Beispiel 2).

Beispiel 2:

; definiert ein Multicolor-Sprite für den Simons'-Basic-Textmodus
; die Zeile 1750 legt weitere Eigenschaften des Sprites fest und macht
; es schließlich sichtbar

```
1700 PROC msprite
1710   DESIGN 1, 14*64
1720   @...bbbbb....
1721   @.bbb...bbb..
1722   @bb.cc.cc.bb.
1723   @bb...c...bb.
1724   @bb...c...bb.
1725   @.bbbcccbbb..
1726   @...bbbbb....
1727   @b...ddd.....
1728   @bb..ddd.....
1729   @bbbbdddbb...
1730   @..bbdddbb...
1731   @....ddd.bb..
1732   @....ddd..bb.
1733   @....ddd...b.
1734   @....ddd.....
1735   @..ccc.ccc...
1736   @.ccc...ccc..
1737   @.cc.....cc..
1738   @ccc.....ccc.
1739   @ccc.....ccc.
1740   @bbbbbbbbbbbb
1750   COLOR 9,9: MOB SET 2,14,8,0,1: CMOB 7,6: s8=300: z8=205:
       MMOB 2,0,0,s8,z8,2,200
1760 END PROC
```

(entnommen aus dem „Trainingsbuch zum Simon's Basic“)

2. Definition eines Zeichens (DESIGN 2, DESIGN 3 und DESIGN 7)

Werte von 2, 3 oder 7 für den ersten Parameter von DESIGN (<n>) führen in die Interpreter-Abteilung „Festlegen eines Zeichens für einen neuen Zeichensatz“. Auch hier gilt: der (mathematisch) gerade Parameter steht für den hochauflösenden Modus (Hires), der ungerade für Multicolor. DESIGN 2, 3 oder 7 machen nur Sinn nach Anwendung des Befehls MEM.

Beachten: Der Multicolormodus für den Textbildschirm ist in **Simons' Basic** nicht ohne Weiteres nutzbar, da der Auswahlbefehl für den Zeichensatz (CSET) die Änderungen von MEM bei Simons' Basic wieder rückgängig macht und die gewünschte Wirkung nicht erzielt wird (in TSB ist dieser Mangel behoben).

Der zweite DESIGN-Parameter (<ad>) selektiert die Adresse des Bitmusters des zu definierenden Zeichens. Die Simons'-Basic-Basisadresse für eigene Zeichensätze (normalerweise $E000) muss bei der Berechnung ausdrücklich eingeschlossen werden, so dass die folgende Formel für alle Zeichen geeignet ist: **ad = $E000 + 8 * Bildschirmcode**. Beispiel: Die Adresse des Zeichens „x“ lautet also ad = $E000 + 8 * 24 = $E0C0.

Beispiel 3:

; ersetzt das Zeichen Pfund mit dem Zeichenmuster für ß

```
1700 PROC eszett
1710   DESIGN 2, $E000 + 8 * 28
1720   @........
1721   @..BBBB..
1722   @.BB..BB.
1723   @.BBBBB..
1724   @.BB..BB.
1725   @.BB..BB.
1726   @.BBBBB..
1727   @.BB.....
1760 END PROC
```

Beachten: Da nach MEM der Bereich der Grafik ($E000 bis $FFFF) zur Hälfte (bis $EFFF) mit den beiden veränderbaren Standardzeichensätzen belegt ist, dürfen Grafikbefehle nur **ab Y-Koordinate 104** zur Anwendung kommen (Speicherbereich ab $F000), sonst zerstören sie den Zeichensatz (wenn der Groß-Klein-Zeichensatz nicht gebraucht wird, kann die Grafik **ab Y-Koordinate 56** verwendet werden). Der Befehl HIRES darf nach MEM gar nicht mehr eingesetzt werden, da er den neuen Zeichensatz löschen würde (in TSB zum Anzeigen der Grafik CSET 2 nehmen).

Da der Zeichensatz mit DESIGN aus dem ROM an eine frei veränderbare Stelle verschoben wird, kann man dorthin auch komplette andere Zeichensätze laden, die damit sofort aktiviert sind. In TSB geht das mit **LOAD „Zeichensatz“,use,0,$e000** oder (wenn man einen Zeichensatz mit SCRSV abgespeichert hat) auch mit **SCRLD 1,use,2,“Zeichensatz“** (mithilfe zweier POKEs, s. Tipp bei SCRSV) oder **SCRLD 1,use,3,“Zeichensatz“** (mithilfe von SCRLD|SV DEF).

Befehl:	DETECT	
Syntax:	**DETECT <n> (n = 0 | 1)**	
Zweck:	Sprite kontrollieren	Bearbeiten von Sprites
Kürzel	**deT**	
Status:	Simons' Basic (Anweisung)	

DETECT bereitet die Abfrage auf Kollision zwischen zwei Sprites bzw. zwischen einem Sprite und dem Hintergrund vor (s. CHECK, der Befehl liest den Inhalt der beiden VIC-Kollisionsregister aus, speichert ihn ab und löscht die Register). Dabei selektiert ein Wert von 0 für **<n>** die Sprite-Sprite-Kollision (VIC-Register $D01E) und ein Wert von 1 die Sprite-Hintergrund-Kollision (Register $D01F). Intern hält der Interpreter dann das Kollisionsergebnis in einer Speicherstelle ($C514) fest. Es wird dort erst durch die nächste Verwendung des DETECT-Befehls wieder geändert.

DETECT sollte für ein sicheres Erkennen einer Kollision mit dem Rasterstrahl synchronisiert werden (s. CHECK).

Beachten: Der in Simons' Basic hinter **DETECT** verwendete Wert (0 oder 1) wird nicht auf Plausibilität überprüft und akzeptiert Zahlenangaben bis 255. Falsche Angaben führen aber auf jeden Fall zu falschen Ergebnissen. Behoben in TSB (gerade Werte für <n> entsprechen hier der 0, ungerade der 1).

Befehl:	DIR	
Syntax:	**DIR [<string>]**	
Zweck:	Anzeige des Inhaltsverzeichnisses eines Diskettenlaufwerks	Ein-/Ausgabe
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Kommando)	

Mit **DIR ["$"]** gibt der Interpreter das Inhaltsverzeichnis der Floppy im Laufwerk mit der von USE eingestellten Geräteadresse aus, ohne dass das aktuelle BASIC-Programm dabei gelöscht wird. Statt der Angabe „$“ als <string>-Parameter können auch alle weiteren Syntaxformen des Directory-Floppy-Befehls angewendet werden. Die optionale (Doppellaufwerk-Drive-Angabe 0 oder 1) nach dem „$“ ist erlaubt, allerdings erzeugt bei einem Einfachlaufwerk die Anwahl des Drives 1 die Ausgabe der Zahl 3341 (das ist die Dezimalzahleninterpretation des hexadezimalen Wertes $0d0d, was zwei Returns entspricht) und im Fehlerkanal findet sich die Meldung **74,DRIVE NOT READY,18,00**.

DIR kann auch innerhalb eines Programms aufgerufen werden. Der Status der Ausführung des Befehls muss bei Bedarf durch Abfrage des Floppy-Fehlerkanals explizit ermittelt werden.

Leider kann Simons' Basic mittels DIR ausschließlich auf das Laufwerk mit Geräteadresse 8 zugreifen. Die Bildschirmausgabe lässt sich dabei nicht via Tastatur anhalten oder abbrechen, lediglich mit Hilfe der Taste <CTRL> verzögern. (Diese Mängel sind in TSB behoben.)

Ein fehlender Parameter oder zu viel angegebene Parameter führen zu einem **SYNTAX ERROR** (wobei der DIR-Befehl zunächst entsprechend abgearbeitet wird, bevor es zu Abbruch und Fehlerausgabe kommt). Beginnt der Parameter nicht mit dem „$“-Zeichen, bricht die Ausführung in Simons‘ Basic mit **BAD MODE** ab. Handelt es sich beim Parameter nicht um einen Ausdruck, der eine Zeichenkette ergibt, kommt es zur Fehlerausgabe von **TYPE MISMATCH ERROR**.

TSB bietet nicht nur eine erweiterte Tastatursteuerung (jede beliebige Taste hält die Ausgabe an und <RUN/STOP> bricht sie ab), sondern es findet sich bei der Ausgabe zusätzlich ein Doppelpunkt hinter allen angezeigten Dateinamen, damit das einfache Eintippen von LOAD vor dem angezeigten Namen und ein darauffolgendes <RETURN> zum Laden einer Datei bereits genügen (TSB braucht keine Laufwerksangabe bei Diskettenbefehlen).

Unter TSB kann der Parameterstring („$“) komplett weggelassen werden, sodass ein DIR als Eingabe für die Ausgabe des Standardinhaltsverzeichnisses (erstes Beispiel unten) ausreicht, ähnlich wie bei DIRECTORY in BASIC 3.5.

DIR "$:*=P" gibt mit Hilfe des Typselektors "=P" nur PRG-Dateien aus.

DIR "$:???.*" gibt alle Dateien aus, die an Position 4 im Namen einen Punkt haben.

DIR "$:?B*=S" zeigt alle sequentiellen Dateien, deren Dateiname als 2. Zeichen „B“ ist.

DIR "$?B,H*=P" zeigt alle PRG-Dateien, deren Dateiname zwei Zeichen lang ist und auf „B“ endet oder mit „H“ beginnt.

Der Doppelpunkt als Bestandteil der Drive-Angabe ist dabei optional.

Befehl:	DISABLE	
Syntax:	**DISABLE**	
Zweck:	Schaltet die programmierte Tastatur-Kontrolle (ON KEY) ab	Ein-/Ausgabe Struktur
Kürzel	**dI**	
Status:	Simons‘ Basic (Anweisung)	

TSB kann beliebige Tastendrücke unabhängig vom laufenden Programm abfangen (siehe ON KEY). Wenn eine solche Taste gedrückt wird, verzweigt der Interpreter in eine dafür vorzusehende Tastatur-Kontrollroutine innerhalb des Programms, in der dieser Tastendruck behandelt wird. Eine solche Routine sollte mit **DISABLE** beginnen, damit in ihr selbst keine weiteren derartigen Tastendrücke zu einer unkontrollierbaren Rekursion führen. Ein Prozeduraufruf statt des GOTO sollte direkt auf der ON-KEY-Zeile von DISABLE und RESUME umrahmt werden, da sonst der Prozedurstack beeinflusst würde.

Beispiel:

```
10 ON KEY "abc": GOTO 10000
```

... langes Programm ...

```
10000 DISABLE
10010 tt$ = CHR$(PEEK($c5ec))
10020 IF tt$ = "a" THEN PRINT AT(0,35)"ah! "
10030 IF tt$ = "b" THEN PRINT AT(0,35)"beh!"
10040 IF tt$ = "c" THEN PRINT AT(0,35)"zeh!"
10050 RESUME
```

(wenn der Benutzer "a", "b" oder "c" drückt, springt der Interpreter in die BASIC-Zeile 10000)

Beachten: In Simons‘ Basic muss nach dem String „abc“ zwingend ein Komma folgen.

Befehl:	DISAPA	
Syntax:	**DISAPA**	
Zweck:	Einzelne Programmzeilen vor Sichtzugriff schützen	Programmierhilfen
Kürzel	-	
Status:	Simons' Basic (Anweisung)	

Beachten: Dieser Befehl wurde aus dem Befehlssatz von TSB entfernt.

DISAPA (abgeleitet aus dem engl. „disappear" = verschwinden, unsichtbar machen) setzt der Programmierer vor diejenigen Zeilen des Programms, in denen Daten abgefragt werden, die verborgen bleiben sollen, wie z.B. Passwörter. Zusammen mit SECURE werden diese Zeilen beim Listen leer angezeigt. Ihr Inhalt wird dessen ungeachtet ganz normal ausgeführt.

Allerdings beruht der „Programmschutz" nur auf einer Besonderheit der LIST-Routine des Interpreters und bietet nicht wirklich die Sicherheit, die man erwarten würde. Die zu schützende Zeile wird z.B. nicht verschlüsselt und kann mit jedem Monitorprogramm eingesehen werden. Mit so einem Programm lässt sich sogar der Sichtschutz mit einer einzigen Eingabe wieder rückgängig machen, daher ist der Nutzen der Befehle DISPAPA und SECURE eher zweifelhaft.

Nach Eingabe des Befehls in die zu schützende Zeile hängt der Interpreter intern vier Doppelpunkte an den Befehl an, so dass zusammen mit dem ohnehin erforderlichen Doppelpunkt eine BASIC-Zeile in etwa so aussehen würde:

```
100 DISAPA:::::IF x$<>"geheim" THEN STOP
```

Der Befehl SECURE verwandelt nun den DISAPA-Befehl in ein Nullbyte, das zusammen mit den Doppelpunkten die LIST-Routine dermaßen irritiert, dass außer der Zeilennummer nichts vom Rest der Zeile angezeigt wird.

Befehl:	DISK	
Syntax:	DISK <string>	
Zweck:	Steuerung eines Diskettenlaufwerks	Ein-/Ausgabe
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Kommando)	

DISK sendet den Floppy-Befehl **<string>** an das aktuell eingestellte Diskettenlaufwerk, um Datei- und Laufwerksaktionen auszulösen. Man erspart sich dabei die sonst umständliche Formulierung `OPEN 1,8,15,"command...": CLOSE 1`, wobei „command...“ auch bei DISK jede erlaubte Syntax für einen Floppy-Befehl des Floppy-DOS enthalten darf (siehe Beispiele).

Fehlt eine Parameterangabe oder ist mehr als ein Parameter angegeben, mündet das in die Fehlermeldung **SYNTAX ERROR**. Ist der Parameter nicht vom Typ einer Zeichenkette, meldet der Interpreter einen **TYPE MISMATCH ERROR**.

Da in Simons' Basic im Direktmodus keine Rückmeldung erfolgt und auch kein Befehl vorgesehen ist, den Floppy-Fehlerkanal abzufragen, muss dort im Fehlerfall immer noch zur alten Form der Floppy-Ansprache gewechselt werden (siehe Beispiele). Allerdings enthält der Fehlerkanal im Falle des Dateilöschens mittels SCRATCH-Kommando nicht den wirklichen Status des Kommandos (der würde "**01,FILES SCRATCHED,NN,00**" lauten, wobei NN die Anzahl der tatsächlich gelöschten Dateien wäre), sondern es wird immer "**00,OK,00,00**" gemeldet. Darüber hinaus kann Simons' Basic mittels DISK ausschließlich auf das Laufwerk mit Geräteadresse 8 zugreifen.

In TSB sind **alle** diese Mängel behoben. Auch der Fehlerkanal kann mit einem eigenen Befehl ausgelesen und angezeigt werden (s. ERROR), allerdings nicht ohne Weiteres in speicherbarer Form.

Beispiele:

DISK "V0" validiert die im Drive 0 eines Laufwerks mit Geräteadresse 8 eingelegte Diskette (und gibt verloren gegangene Blöcke frei) - die Ausführung kann etwas Zeit beanspruchen ...

DISK "I" (großes i) Das Floppy-Laufwerk initialisieren.

DISK "N:DISKNAME,ID" Eingelegte Diskette auf Namen „DISKNAME“ mit Disk-ID „ID“ formatieren.

DISK "S:A*" Alle Dateien, die mit „A“ beginnen löschen.

DISK "R:NEU=ALT" Die Datei „ALT“ auf den Namen „NEU“ umbenennen.

DISK "C:DAT.COPY=DAT" Dupliziert die Datei „DAT“, die dann den Namen „DAT.COPY“ trägt.

Innerhalb eines Programms:

Laut dem angezeigten Inhaltsverzeichnis der Floppy löscht das folgende Programm interaktiv Dateien (auch mit Joker-Angabe). Beenden mit Leereingabe. Wegen des INPUT-Befehls ist u.a. die Verwendung der Zeichen Doppelpunkt und Komma nicht möglich (Meldung **EXTRA IGNORED** wird angezeigt).

```
100 DIR : PRINT
110 D$="" : INPUT "ZU LOESCHENDE DATEI";D$
120 IF D$="" THEN END
130 DISK "S:"+D$
140 ERROR: REM FEHLERKANAL AUSLESEN UND ANZEIGEN ...
160 GOTO 100
```

Beachten: Unter Simons‘ Basic gibt es den Befehl ERROR nicht, man müsste den Fehlerkanal auf die übliche umständliche Weise auslesen und anzeigen.

Befehl:	DISPLAY	1
Syntax:	**PRINT DISPLAY** **a = DISPLAY**	
Zweck:	Gibt die Basisadresse des aktuellen Video-RAMs zurück	Bearbeiten des Textbildschirms Programmierhilfen
Kürzel	**disP**	
Status:	Neue TSB-Systemvariable	

DISPLAY liefert die Basisadresse des aktuellen Video-RAMs zurück (gemäß dem Inhalt von Speicherstelle $0288). Je nach Einstellung von MEM ist das **1024** (hex $0400, Normalzustand, auch nach HI-RES) oder **52224** (hex $CC00, nach MEM).

Programme, die in den Bildschirmspeicher schreiben, können diesen nun beliebig verlegen, ohne weiter angepasst werden müssen, wenn sie von vornherein mit DISPLAY arbeiten.

Befehl:	DISPLAY	2
Syntax:	**DISPLAY**	
Zweck:	Belegung der F-Tasten anzeigen	Ein-/Ausgabe Programmierhilfen
Kürzel	**disP**	
Status:	Erweitertes Simons‘ Basic (Kommando)	

DISPLAY dient dazu, die Belegung der Funktionstasten (unter Simons' Basic/TSB gibt es 16) auf dem Bildschirm anzuzeigen. Der KEY-Befehl wird dabei vorangestellt, so dass die F-Tasten gleich editierbar sind.

Weniger bekannt ist, dass in Simons‘ Basic durch `POKE $C646,10` die Ausgabe der Funktionstastenbelegung abgeschaltet werden kann. Man schaltet sie mit `POKE $C646,0` wieder ein. In TSB schreibt man stattdessen einfach `KEY OFF` bzw. `KEY ON`.

Statt DISPLAY kann man unter Simons‘ Basic auch `KEY 0` eingeben. Unter TSB ist `KEY 0` eine andere Schreibweise von `KEY OFF` (F-Tastenbelegung ausschalten, wieder einschalten mit `KEY ON`).

Befehl:	DIV	1
Syntax:	**DIV**	
Zweck:	löscht den Textbildschirm	Bearbeiten des Textbildschirms
Kürzel	-	
Status:	Neuer TSB-Befehl (Kommando)	

DIV löscht den Textbildschirm. Der Cursor steht danach in der linken oberen Ecke an Position 0,0. DIV ist eine Kurzform für die Anweisung `PRINT CHR$(147);` (mit Semikolon).

In anderen BASIC-Dialekten heißt der Befehl oft CLS (für **CL**ear **S**creen), was auch in TSB anwendbar ist. DIV steht für **VID**eo, rückwärts gelesen.

Befehl:	DIV	2
Syntax:	**a = DIV(<z>,<n>)** **PRINT DIV(<z>,<n>)**	
Zweck:	ganzzahlige Division	Bearbeiten und Darstellen von Zahlen
Kürzel	-	
Status:	Simons‘ Basic (numerische Funktion)	

Die Funktion **DIV** dividiert den Zähler **<z>** durch den Nenner **<n>** und liefert ein ganzzahliges Ergebnis, z.B. DIV(9,4) = 2. Simons' Basic arbeitet an dieser Stelle nicht mit einem zusätzlichen Operator, sondern realisiert dies als numerische Funktion DIV. Sie entspricht dem Ausdruck `INT(INT(<z>)/INT(<n>))`.

Beachten: Alle Simons'-Basic-Funktionen arbeiten nicht, wenn sie zweiter Parameter eines POKE-Befehls sind. Die Werte müssen vor ihrer Verwendung bei POKE einer Variablen zugewiesen werden (behoben in TSB.)

Die Argumente dürfen dabei als Ganzzahl aufgefasst (implizite Umwandlung mit der INT-Funktion) nur im Bereich von 0 bis 65535 liegen, andernfalls meldet der Interpreter die Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Beispiele:

```
10 INPUT "POSITIVE ZAHL"; Z
20 A = DIV(Z,4): B = MOD(Z,4)
30 PRINT Z "DURCH 4 IST" A "MIT REST" B
```

Von einer Zahl wird berechnet, wie oft sie in einer anderen (hier: 4) enthalten ist und wie viel Rest bleibt.

```
10 Z=10: N=2.25
20 PRINT DIV(Z,N)
30 PRINT INT(INT(Z)/INT(N))
40 PRINT INT(Z/N)
```

liefert folgende Ausgabe:

```
 5
 5
 4
```

Zeile 30 ist die äquivalente BASIC-V2-Implementierung zu DIV() und die Variante in Zeile 40 zeigt, wie bei einer zu einfachen Nachbildung bei Argumenten, die durchaus Fließkommazahlen sein dürfen, ein falsches Ergebnis zustande kommt.

Befehl:	DO .. DONE	
Syntax:	**IF .. THEN DO** .. **[ELSE]** .. **DONE**	
Zweck:	umschließt mehrzeilige Bedingungszweige	Struktur
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

Mit **DO** und **DONE** hat der Programmierer die Möglichkeit, die beiden Bedingungszweige eines IF über mehrere BASIC-Zeilen auszuweiten, und so sein Programm erheblich übersichtlicher zu gestalten. ELSE trennt hier die Wahr- und Nicht-Wahr-Zweige voneinander, ist kein Nicht-Wahr-Zweig vorgesehen, kann ELSE weggelassen werden.

Der Stack dieser Konstruktion verträgt 10 Verschachtelungen. Das Befehlswort DO NULL hat keinen Einfluss auf DO..DONE.

Beispiel:

```
10 PRINT "test ";
20 FETCH "jn",1,x$
30 IF x$ = "j" THEN DO
40 :  PRINT "ja"
50 ELSE
60 :  PRINT "nein"
70 DONE
```

(je nachdem, ob der Benutzer „j“ oder „n“ drückt, erscheint eine andere Antwortausgabe)

Ein komplettes Anwendungsbeispiel beim Befehl MOB SET (Beispiel 2).

Beachten: Das Befehlswort **DO** darf auf keinen Fall in einem Prozedurnamen vorkommen (Beispiele: „frodo“, „dortmund“, „fundort“). Da dieses Schlüsselwort absoluten Vorrang bei der Tokenisierung einer Befehlszeile hat, führt eine Zuwiderhandlung zu unkontrollierbaren Zuständen des Interpreters bis hin zu einem Absturz des Systems.

Befehl:	DO NULL	
Syntax:	**DO NULL**	
Zweck:	Warten auf einen Tastendruck	Struktur
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

Kommt ein Programm an eine **DO-NULL**-Anweisung, wartet es, bis der Benutzer eine Taste drückt. Im Programm kann auf derselben BASIC-Zeile kein weiterer Befehl mehr folgen (er wird dann wie ein Kommentar behandelt).

Beachten: DO NULL darf **nicht** in einer IF-Zeile (nach THEN) verwendet werden. Ersatzweise kann man hier auf `WAIT 198,1:GET x$` oder einfach KEYGET x$ ausweichen.

Beispiel:

```
10 HIRES 0,1: CIRCLE 160,100,90,90,1
20 DO NULL
```

(schaltet die Grafik an, malt einen schwarzen Kreis auf weißem Grund und wartet)

Befehl:	DOWN	
Syntax:	**DOWNB / DOWNW <zl>,<sp>,<bt>,<ho>**	
Zweck:	Abwärtsscrollen eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	dO (für DOWNB, DOWNW hat keins)	
Status:	Erweitertes Simons' Basic (Anweisung)	

DOWNB bzw. **DOWNW** erlaubt es dem Programmierer, Bereiche des Textbildschirms inklusive der Farben zeilenweise nach unten zu scrollen. In der obersten, freiwerdenden Zeile werden je nach Typ des Befehls Leerzeichen aufgefüllt (DOWNB, das „B" steht für „blank"), was nur einen einzigen kompletten Scrollvorgang erlaubt, oder die unten herausfallenden Zeichen wieder eingefügt (DOWNW, das „W" steht für „wrap"), was mit dem gleichen Inhalt immer wieder durchgeführt werden kann.

Leider handelt es sich bei diesem Scrolling um ein zeichenweises Scrolling, das womöglich ruckelig wirken kann. Pixelweises Scrolling („Smooth Scrolling") ist mit Simons'-Basic-Befehlen nicht möglich.

Bei Über- oder Unterschreitung der zulässigen Werte (siehe Box) meldet der Interpreter die Fehlermeldung **BAD MODE ERROR**.

Beachten: Wenn DOWNW direkt am oberen Bildschirmrand beginnt (Zeile 0, Spalte 0), kann es dort zu störenden „Geisterzeichen" kommen, die allerdings keinen schwerwiegenden Folgefehler verursachen (Fehler behoben in TSB).

Beispiel:

```
100 CLS
110 COLOR 0,13: X=14: RV$=CHR$(18): SC$=CHR$(144): GR$=CHR$(152)
115 ST$=RV$+SC$+":"+GR$+"{11*SPACE}"+SC$+":": RR=39-LEN(ST$)+4
120 FILL 0,0,40,1,160,5: MOVE 0,0,40,1,24,0
125 FILL 0,0,1,25,160,5: MOVE 0,0,1,25,0,39
130 REPEAT: X=X+RND(1)*2-1: GET X$
140 IF X<1 THEN X=1
150 IF X>RR THEN X=R
160 DOWNB 1,1,38,23
170 PRINT AT(1,X) ST$
180 UNTIL X$>""
```

Das Beispiel erzeugt eine sich windende Straße. Abbruch ist per Tastendruck möglich. DOWNB lässt hier die Straße nach unten bewegen.

Beispiel übernommen und angepasst aus dem Buch „Das Trainingsbuch zum Simons' Basic".

Befehl:	DRAW TO	1
Syntax:	**DRAW TO <x>, <y>, <fq>**	
Zweck:	Ziehen einer Linie zwischen zwei Grafikpunkten	Grafik-Befehle
Kürzel	dR to	
Status:	Neuer TSB-Befehl (Anweisung)	

DRAW TO verbindet den Endpunkt einer zuvor gezeichneten hochauflösenden Grafik-Figur mit dem angegebenen Zielpunkt an der Position **<x>,<y>** in der Farbe, die durch **<fq>** bestimmt wird. Je nachdem, welcher Grafikmodus aktiviert ist (siehe HIRES und MULTI), hat man in x-Richtung 320 ansteuerbare Positionen (Hires-Modus) oder 160 (Multicolor-Modus). In y-Richtung beträgt die Auflösung immer 200 Pixel. Auch die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Hinweis: Nicht alle Grafikfiguren liefern den korrekten Startpunkt für DRAW TO. Bei ANGL, DUP und PAINT ist es deren Startpunkt, bei CIRCLE ist es der Startpunkt der letzten Einzellinie der Figur. BLOCK weicht völlig ab (gibt die vertauschte Startkoordinate weiter). Die Befehle TEXT und CHAR liefern die Position der linken unteren Ecke des letzten ausgegebenen Zeichens zurück. PLOT, ARC, REC und LINE verhalten sich korrekt (Endpunkte der jeweiligen Figur).

Auch die Befehle MMOB und RLOCMOB für Sprites beeinflussen DRAW TO (mit ihren jeweiligen Endpunkten). Der Befehl DRAW selbst (für festgelegte Linienzüge) hat keine Auswirkung auf DRAW TO.

Befehl:	DRAW	2
Syntax:	DRAW <str>, <x>, <y>, <fq>	
Zweck:	Definieren einer Figur aus Linien	Grafik-Befehle
Kürzel	dR	
Status:	Simons' Basic (Anweisung)	

DRAW ist das schlechte Ergebnis einer guten Idee. Der Befehl dient dazu, immer gleich aussehende Linienzüge platzsparend und flexibel in Simons' Basic eingeben zu können. Dazu hat sich David Simons einen Befehl ausgedacht, der ein bisschen die Turtle-Grafik der Programmiersprachen LOGO oder COMAL widerspiegeln sollte. Der Befehl enthält Anweisungen, um den Grafikcursor in eine bestimmte Richtung zu bewegen (in LOGO heißt das „vorwärts“ oder kurz „vw“), dabei den Schreibstift entweder hochzuheben (damit er *nicht* schreibt) oder abzusenken (dann geht's los mit dem Malen), was in LOGO „Stift hoch“ oder „Stift ab“ genannt wird. Alle diese Dinge enthält DRAW auch.

Der große Nachteil ist aber nun, dass man diesen Richtungsanweisungen nicht individuell sagen kann, wie weit sie gehen sollen. Dafür gibt es einen Extrabefehl (ROT), der dann für die gesamte Anweisungsfolge gilt. Es wäre auch schön, wenn man individuelle Winkel verwenden dürfte, aber das ist bei DRAW ebenfalls nicht vorgesehen. Man kann nur das gesamte „Objekt“ drehen (wiederum mit ROT), und nur in 45-Grad-Schritten, die zudem noch ein ums andere Mal linear vergrößern, also das geplante Objekt verändern.

Wie funktioniert DRAW nun? Der erste Parameter **(<str>**) ist ein String, der in Form von Ziffern zwischen 0 und 9 die Richtungsbefehle enthält, nach dem folgenden Muster:

Richtungsbefehl	ohne zu zeichnen, nach...	Richtungsbefehl	mit zeichnen, nach...
0	rechts	5	rechts
1	oben	6	oben
2	unten	7	unten
3	links	8	links
4	(unten)	9	Stop

Die Ziffer „9“ beendet das Abarbeiten eines Strings, auch wenn darin noch mehr folgen sollte. Das ist ganz nützlich, um in so einem String Kommentare unterzubringen, ohne den Interpreter zu stören. Wie immer in BASIC darf ein String jedoch insgesamt nicht länger sein als 255 Zeichen.

Die beiden folgenden Parameter **<x>** und **<y>** legen den Startpunkt des Objekts fest. Der letzte Parameter (**<fq>**) ist die in Simons' Basic übliche Angabe der Farbquelle bei Grafikbefehlen. Näheres dazu beim Befehl HIRES.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beispiel:

```
100 w$="777755662255666600"
110 i$="727700"
115 k$="1111777711562577001111"
120 DIM s%(1)
125 HIRES 1,0:
    x=160:
    y=100:
    s=8
130 s%(0)=s:
    s%(1)=s-s/4
135 d$=w$+i$+k$+i$
140 REPEAT :
      GET x$:
      FOR w=0 TO 7
150     ROT w,s%(w AND 1):
        DRAW d$,x,y,1
160     DRAW d$,x,y,0
170   NEXT
180 UNTIL x$>""
185 ROT 0,s:
    DRAW d$,x,y,1
190 DO NULL
```

Bild 10 DRAW und ROT(ate) in Aktion

(erzeugt das Beispielbild)

Kommentar: Die Größe des Objekts wird mit s in Zeile 125 definiert. Da beim Drehen mit Mehrfachen von 45 Grad Verzerrungen entstehen (das Objekt wird ein Viertel größer), werden diese in den Zeilen 130 und 150 wieder herausgerechnet. Korrekterweise müsste die Formel statt `s%(1)=s-s/4` lauten: `s%(1)=int(s/sqr(2)+.5)`, was in diesem Fall auf das gleiche Ergebnis hinausläuft.

Befehl:	DUMP	
Syntax:	**DUMP**	
Zweck:	Inhalt von Variablen anzeigen	Programmierhilfen
Kürzel	dU	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

DUMP gibt nach einem Programmlauf (bzw. nach einer Programmunterbrechung) alle Variablen und ihre aktuellen Werte aus. Dabei werden auch FN-Funktionen berücksichtigt und angezeigt (gekennzeichnet mit einem Ausrufezeichen nach dem Namen).

Die Ausgabe der Werte kann mit der <CTRL>-Taste verlangsamt und mit <RUN/STOP> angehalten werden.

Die groben Simons‘-Basic-Fehler gibt es in TSB nicht mehr:

DUMP ist dort derartig fehlerbehaftet, dass es nicht brauchbar ist. Ein negatives Vorzeichen bei Fließkommazahlen wird ignoriert, ebenso bei Ganzzahlen, dort wird aber zusätzlich noch 65536 zum Wert dazu addiert. Leere Strings (Zeichenketten) führen zu einer unkontrollierten Bildschirmausgabe von 256 zufälligen Zeichen irgendwo aus dem Speicher, die im harmlosesten Falle zu wilden Farbveränderungen führt, DEF-FN-Funktionsnamen (die schließlich auch im Variablenbereich abgelegt werden) werden überhaupt nicht berücksichtigt.

Befehl:	DUP	1
Syntax:	DUP <x1>,<y1>,<bt>,<ho>,<x2>,<y2>,<fq>,<zm>	
Zweck:	Umkopieren von Grafik an eine andere Stelle (mit Skalierungsmöglichkeit)	Grafik-Befehle
Kürzel	-	
Status:	Neuer TSB-Befehl(Anweisung)	

Mit **DUP** hat der Programmierer ein Werkzeug in der Hand, Bereiche der Grafik von einer Stelle an eine andere zu kopieren und dabei auch zu vergrößern oder zu verkleinern. Zulässige Werte für **<x1>** und **<x2>** sind 0..319 (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für **<y1>** und **<y2>** sind in beiden Fällen Werte von 0 bis 199 erlaubt. Der Punkt 0,0 ist in der linken oberen Ecke. **<bt>** (Breite) und **<ho>** (Höhe) markieren den zu kopierenden Bereich. Mit **<zm>** (Zoomfaktor) kann dieser Bereich dabei zunächst einmal (linear) vergrößert werden (der Wert 2 verdoppelt), sowohl in der Höhe als auch in der Breite. Der Punkt wird in Farbe **<fq>** gesetzt (s. HIRES bzw. MULTI).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beispiel 1 erzeugt den mittleren Teil der Animation im Bild (das Kreise-Demo):

```
230 CIRCLE 10,10,10,10,1
240 :
250 FOR i=0 TO 14
260 DUP 0,0,20,20,20*i+20,0,1,1
270 NEXT
280 FOR i=0 TO 7
290 DUP 0,0,20,20,300,20*i+20,1,1
300 NEXT
310 FOR i=15 TO 0 STEP -1
320 DUP 0,0,20,20,20*i,180,1,1
330 NEXT
340 FOR i=7 TO 0 STEP -1
350 DUP 0,0,20,20,0,20*i+20,1,1
360 NEXT
370 :
380 DUP 0,0,20,20,105,50,1,5
390 DO NULL
```

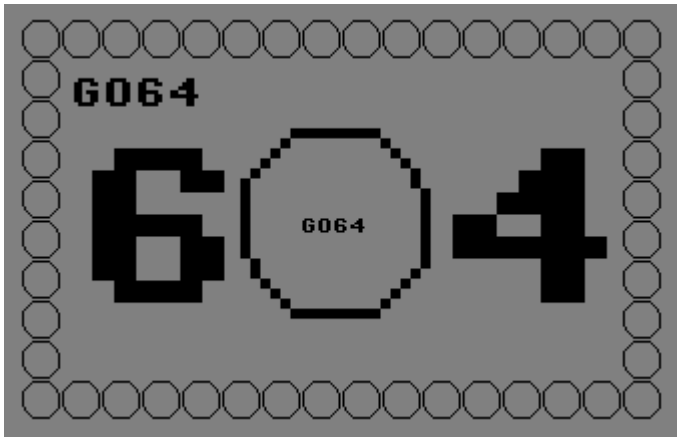


Bild 11 Mit DUP Pixel kopieren und ggf. dabei skalieren

Erläuterungen: Zeile 210 zeichnet einen Kreis mit Radius 10 Pixel in die linke obere Ecke. Dieser Kreis wird in 250 bis 360 rund um den Bildschirm kopiert und in fünffacher Vergrößerung einmal in die Mitte gesetzt.

Beispiel 2 ist ein komplettes TSB-Programm (zum Spiegeln von Text):

```
100 init: achsen: vorlage
180 nach unten: nach links unten: nach links
190 DO NULL
290 bildschirmfarben
900 END
```

```
1000 PROC init
1010 HIRES 11,12: COLOR: MULTI 12,11,1
1020 x=84: y=30: br=7*11: ho=3*20: f=0
1030 x1=76: y1=105
1040 END PROC

1100 PROC achsen
1110 LINE x-4,y-4,x-4,200-y,2
1120 LINE 4,96,156,96,2
1125 LINE 4,97,156,97,2
1130 END PROC

1200 PROC vorlage
1210 TEXT x,y,"{ctrl-b}Achsen-",3,2,11
1220 TEXT x,y+20,"{ctrl-b}Symme-",3,2,12
1230 TEXT x,y+40,"{ctrl-b}trie",3,2,12
1260 END PROC

1300 PROC nach links
1305 f=f+1
1310 FOR i=0 TO br-4
1320 DUP x+i,y,1,ho,x1-i,y,f,1
1330 NEXT
1340 END PROC

1400 PROC nach unten
1405 LOW COL 15,0,0
1410 FOR i=0 TO ho
1420 DUP x,y+ho-i,br,1,x,y1+i,1,1
1430 NEXT
1435 HI COL
1440 END PROC

1500 PROC nach links unten
1510 y=105: nach links: y=30
1540 END PROC

1600 PROC bildschirmfarben
1610 COLOR 11,12,0: FCOL 0,0,40,25,0
1620 END PROC
```

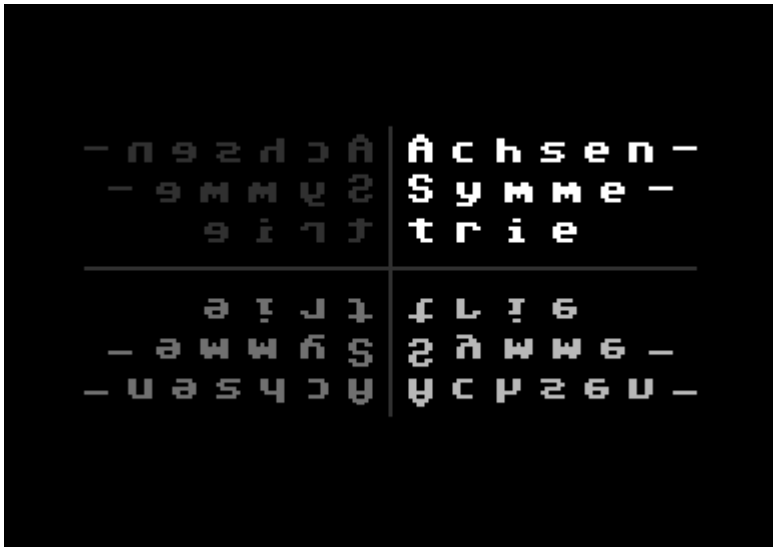


Bild 12 Aber auch spiegeln

Erläuterungen: Bei den beiden DUP-Befehlen (Zeilen 1320 und 1420) wird der Befehl wie ein Scanner eingesetzt. Er holt die Punkte an einer Stelle ab und schreibt sie in neuer Anordnung anderswo zurück. Die Prozedur „bildschirmfarben“ (Zeile 1600) rekonstruiert mit FCOL den Farbhintergrund des Textbildschirms, der von MULTI als Farbspeicher verwendet wurde.

Befehl:	DUP	2
Syntax:	**a$ = DUP(<string>,<n>)** **PRINT DUP(<string>,<n>)**	
Zweck:	Vervielfältigung eines Strings	Stringfunktionen
Kürzel	-	
Status:	Simons‘ Basic (Anweisung)	

DUP vervielfältigt den im ersten Argument **<string>** angegebenen String so oft, wie das zweite Argument **<n>** vorgibt. Für <n> ist (in TSB) auch der Wert 0 (null) erlaubt und erzeugt damit einen Leerstring.

Beachten: Wenn der resultierende String die Länge von 255 Zeichen übersteigt, stürzt Simons' Basic ab. Dasselbe passiert, wenn ein Leerstring dupliziert werden soll. TSB meldet dagegen in beiden Fällen einen **STRING TOO LONG ERROR**.

Beispiel:

```
10 PRINT "Ueberschrift"
20 PRINT DUP("-",40);
30 PRINT "Untertitel"
```

(unter dem Wort „Ueberschrift“ erscheint eine Linie von 40 Minuszeichen)

Befehl:	ELSE	
Syntax:	**ELSE**	
Zweck:	Leitet den Nicht-Wahr-Zweig einer Bedingungs-klausel ein	Struktur
Kürzel	eL	
Status:	Erweitertes Simons' Basic (Anweisung)	

ELSE ergänzt den BASIC-V2-Befehl IF (der nur einen Wahr-Zweig in seiner Bedingungsklausel kennt: das, was auf THEN folgt) um einen weiteren Zweig, den Nicht-Wahr-Zweig. Das bedeutet, dass in der gleichen BASIC-Zeile, in der IF..THEN steht, in TSB auch folgen kann, was passieren soll, wenn die Bedingung nach IF *nicht* zutrifft. (In BASIC V2 wird dieser Fall nicht ausdrücklich behandelt und muss extra abgefragt werden.) Der Nicht-Wahr-Zweig (mit ELSE) muss vom Wahr-Zweig mit einem Doppelpunkt abgetrennt werden.

Beachten: Ein ELSE am Ende eines kaskadierten IF-Gefüges (mehrere IFs auf einer Zeile) wird nur dann **nicht** ausgeführt, wenn **alle** Bedingungen in der Kaskade wahr sind (zutreffen). Das ELSE bezieht sich also auf **jede einzelne** Bedingungsklausel in der Kaskade. Ein ELSE hat am Ende einer IF-Kaskade daher nichts zu suchen (führt sogar bei mehrfachem Aufruf zu einem regelrechten Absturzfehler)!

IFs mit einem ELSE-Zweig sind jedoch beliebig verschachtelbar (etwa durch Aufrufen eines Unterprogramms oder einer Prozedur innerhalb einer Bedingungsklausel, wo weitere IFs mit ELSE auftreten).

In TSB hat der Programmierer darüber hinaus mit den Befehlen DO .. DONE die Möglichkeit, die beiden Bedingungszweige eines Bedingungsgefüges über beliebig viele BASIC-Zeilen zu verlängern. ELSE trennt auch hier die Wahr- und Nicht-Wahr-Zweige voneinander (Syntax siehe Beispiel 2). Das zu DO gehörige ELSE und das ELSE innerhalb einer IF-Klausel auf einer einzigen Zeile werden auf kollisionsfreie Weise verarbeitet. Sie kommen sich nicht in die Quere.

Beispiel 1:

```
10 PRINT "test ";
20 FETCH "jn",1,x$
30 IF x$ = "j" THEN PRINT "ja": ELSE PRINT "nein"
```

(je nachdem, ob der Benutzer „j“ oder „n“ drückt, erscheint eine andere Antwortausgabe. In Simons' Basic steht auch *nach* dem ELSE ein Doppelpunkt)

Beispiel 2:

```
10 PRINT "test ";
20 FETCH "jn",1,x$
30 IF x$ = "j" THEN DO
40 :  PRINT "ja"
50 ELSE
60 :  PRINT "nein"
70 DONE
```

(das Gleiche Ergebnis wie in Beispiel 1, aber diesmal verteilt auf mehrere Zeilen)

Befehl:	END LOOP	
Syntax:	**END LOOP**	
Zweck:	Markiert das Ende eines Schleifenkörpers	Struktur
Kürzel	end L	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

END LOOP definiert das physische Ende einer Schleife. Eine Schleife umschließt einen Teil eines Programms, der unter Umständen mehrfach durchlaufen wird. Das logische Ende der Schleife wird hingegen mithilfe des Befehls EXIT, mit dem eine vorher definierte Bedingung überprüft wird, an den Interpreter gemeldet.

Fehlt END LOOP, meldet der Interpreter einen **LOOP ERROR**.

Mit dem Konstrukt LOOP .. EXIT .. END LOOP lassen sich alle Schleifentypen nachbilden:

- Endlosschleife (kein EXIT)
- kopfgesteuerte Schleife (siehe Beispiel)
- fußgesteuerte Schleife (siehe Beispiel bei REPEAT)
- Zählschleife (siehe FOR und NEXT)

Beispiel für eine kopfgesteuerte Schleife:

```
10 A$ = ""
20 LOOP
30   EXIT IF PLACE(A$,"jnJN")
40   PRINT "Ja (j) oder Nein (n)? ";
50   FETCH "{crsr right}", 1, A$
60 END LOOP
```

Man kann alle Buchstaben eingeben, aber die Schleife wird erst verlassen, wenn „j“ oder „n“ getippt wurde (auch großgeschrieben). Wenn A$ vor der Schleife bereits einen der Tastendrücke enthielte, würde die Abfrage gar nicht erst stattfinden (z.B. das Beispiel mit A$="j" in Zeile 10 probieren).

Befehl:	END PROC	
Syntax:	**END PROC**	
Zweck:	Beenden einer Prozedur	Struktur
Kürzel	enD	
Status:	Simons‘ Basic (Anweisung)	

In TSB und Simons' Basic können Unterprogramme mit einem Namen versehen werden (siehe PROC). Sie werden dadurch unabhängig von ihrer Lage im Programm und der Programmierer kann leichter den Überblick bewahren (Namen lassen sich leichter einem Zweck zuordnen als Zeilennummern). Der Befehl **END PROC** beendet ein solches Unterprogramm. Er entspricht damit weitgehend dem BASIC-V2-Befehl RETURN.

Wenn ein TSB-Programm auf ein END PROC trifft, ohne eine Prozedur aufgerufen zu haben, erscheint die Meldung **END PROC W/O EXEC ERROR**.

Beispiel:

```
10 PRINT "bitte eine taste druecken!"
20 warten
30 PRINT "danke"
999 END

1000 PROC warten
1010 DO NULL
1020 END PROC
```

(Nach der Aufforderung, eine Taste zu drücken, wartet das Programm, gibt schließlich eine Rückmeldung aus und endet)

Befehl:	ENVELOPE	
Syntax:	**ENVELOPE <stimme>, <a>, <d>, <s>, <r>**	
Zweck:	Festlegen der Hüllkurve (ADSR-Kurve) eines Sounds	Sound
Kürzel	eN	
Status:	Simons' Basic (Anweisung)	

Mit **ENVELOPE** bestimmt man die sogenannte Hüllkurve der angewählten Stimme **<stimme>** mit vier Werten, die angeben, wie und wie stark der Ton am Anfang anschwillt (**A**ttack **<a>**) und danach wieder abfällt (**D**ecay **<d>**), woraufhin der Ton eine Zeit lang seine Lautstärke hält (**S**ustain **<s>**) und ganz zum Schluss mit einer bestimmten Dauer endgültig ausklingt (**R**elease **<r>**).

Für die Ausführung der Parameter ist das Key-Bit des WAVE-Befehls zuständig (Bit 0). Ist dieses Bit=1, werden <a>, <d> und <s> abgearbeitet, wird es auf 0 gesetzt, tritt Parameter <r> in Kraft. Aus diesem Grund sollte ENVELOPE auf jeden Fall **vor** WAVE im Programm ausgeführt werden.

Für die Parameter <a>, <d> und <r> sind Werte zwischen 0 und 15 möglich, wobei diesen intern eine festgelegte Dauer in tausendstel Sekunden zugeordnet ist. Diese Zeitdauern lauten (nach dem Handbuch):

Wert	Attack-Zeit Tausendstel Sek	Decay/Release-Zeit Tausendstel Sek
0	2	6
1	8	24
2	16	48
3	24	72
4	38	114
5	56	168
6	68	204
7	80	240
8	100	300
9	250	750
10	500	1500
11	800	2400
12	1000	3000
13	3000	9000
14	5000	15000
15	8000	24000

Der Sustain-Parameter **<s>** gibt laut Handbuch an, auf welchen Lautstärkepegel *Decay* abfallen soll. Auch hier sind Werte zwischen 0 und 15 möglich (ein Wert von 15 macht hier allerdings wenig Sinn, da es dann keinen Abfall gibt). Dieser Parameter beeinflusst also die Einstellungen bei VOL.

Wertangaben von über 15 bei den Hüllkurvenparametern führen zu einem **BAD MODE ERROR**. Ebenso andere Werte als 1, 2 oder 3 für den Parameter <stimme>.

Beispiel:

```
100 VOL 15
110 ENVELOPE 1,1,8,10,10
120 WAVE 1, 00100000
130 MUSIC 150, "{clear}1c2{f2}"
140 PLAY 1
150 VOL 0
```

(spielt mit Stimme 1 den Sägezahn-Ton c2 als Viertelnote)

Ein komplettes Anwendungsbeispiel beim Befehl MOB SET (Beispiel 2).

Hinweis: Gesteuert wird die Ausführung von **ENVELOPE** mit dem untersten Bit beim Parameter von WAVE (Gate-Bit). Ist es auf 1, werden die ersten drei Parameter von **ENVELOPE** abgearbeitet, fällt es auf 0, führt der Interpreter den letzten Parameter <r> aus. Der Befehl PLAY verwendet ebenfalls das Gate-Bit zum Einschalten eines Tones.

Befehl:	ERRLN	
Syntax:	**a = ERRLN** **PRINT ERRLN**	
Zweck:	Enthält die Zeilennummer des aktuellen Fehlers	Abfangen von Laufzeitfehlern
Kürzel	-	
Status:	Erweitertes Simons' Basic (Systemvariable)	

ERRLN enthält die Zeilennummer des zuletzt aufgetretenen Fehlers des Interpreters. Nach dem Abruf wird ERRLN nicht zurückgesetzt und bleibt bis zum nächsten Fehlerereignis bestehen.

Die Fehlerkontrolle insgesamt wurde in TSB so überarbeitet, dass sie nunmehr voll funktionsfähig ist, siehe ON ERROR.

Beispiel:

```
10 ON ERROR: GOTO 10000

15 PRIN "{clr/home}"
20 PRINT "ok"

10000 NO ERROR
10010 PRINT "in zeile " ERRLN "trat fehler nr." ERRN "auf."
10020 STOP
```

Bei einem Programmlauffehler springt der Interpreter in die BASIC-Zeile 10000.

Befehl:	ERRN	
Syntax:	**a = ERRN** **PRINT ERRN**	
Zweck:	Enthält aktuelle Fehlernummer	Abfangen von Laufzeitfehlern
Kürzel	-	
Status:	Erweitertes Simons' Basic (Systemvariable)	

ERRN enthält die Nummer des zuletzt aufgetretenen Fehlers des Interpreters. Nach dem Abruf wird ERRN zurückgesetzt (auf den Wert 128).

Leider ist die Simons'-Basic-Fehlerkontrolle nicht in der Lage, ihre eigenen Fehler zu identifizieren (Fehlernummern ab 32) , ein **NO PROC ERROR** z.B. (Nummer 33) wird als **SYNTAX ERROR** (Nummer 11) „verkauft" (siehe Beispiel).

Dieser Mangel wurde in TSB behoben (siehe Liste der (T)SB-Fehlermeldungen beim Befehl OUT). Die Fehlerkontrolle insgesamt wurde in TSB so überarbeitet, dass sie nunmehr voll funktionsfähig ist, siehe ON ERROR.

Beispiel:

```
10 ON ERROR: GOTO 10000

15 PRIN "{clr/home}"
20 PRINT "ok"

10000 NO ERROR
10010 PRINT "in zeile " ERRLN "trat fehler nr." ERRN "auf."
10020 STOP
```

Bei einem Programmlauffehler springt der Interpreter in die BASIC-Zeile 10000.

Befehl:	ERROR	
Syntax:	**ERROR**	
Zweck:	gibt die Fehlermeldung der Floppy aus	Abfangen von Laufzeitfehlern
Kürzel	-	
Status:	Neuer TSB-Befehl (Kommando)	

ERROR gibt die Floppy-Fehlermeldung aus dem Fehlerkanal auf dem Bildschirm aus. Der Befehl kann auch in Programmen verwendet werden. Das Ergebnis wird aber nur angezeigt und nirgends gespeichert.

Beispiel

```
PRINT AT(10,10)"";:ERROR
```

Zeigt den Floppy-Status an der Bildschirmposition Zeile 10, Spalte 10 an.

Befehl:	EXEC	
Syntax:	[EXEC][ ]<label>	
Zweck:	Aufrufen einer Prozedur	Struktur
Kürzel	eX	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

In TSB können genau wie in Simons' Basic Unterprogramme mit einem Namen versehen werden (siehe PROC). Sie werden dadurch unabhängig von ihrer Lage im Programm und der Programmierer kann leichter den Überblick bewahren (Namen lassen sich leichter einem Zweck zuordnen als Zeilennummern). Der Befehl **EXEC** ruft ein solches Unterprogramm auf, arbeitet es ab und kehrt an seinen Ausgangsort zurück. Er entspricht damit weitgehend dem BASIC-V2-Befehl GOSUB.

Anders als in Simons' Basic darf das Schlüsselwort EXEC in TSB weggelassen werden (ähnlich wie LET bei der Wertezuweisung an Variablen). Außerdem dürfen in der Zeile, in der ein Prozeduraufruf verwendet wird, nach einem Doppelpunkt hinter dem Label beliebige weitere BASIC-Befehle folgen. Auch ein Leerzeichen am Anfang eines Labels (die eventuelle Lücke zwischen PROC und dem Labelnamen wie im Beispiel unten) ignoriert das TSB-EXEC. Prozeduraufrufe sind damit viel flexibler zu handhaben als unter Simons' Basic (vgl. Beispiel 2 zum Befehl DUP).

Die Abarbeitungsgeschwindigkeit von EXEC und CALL ist unter Simons‘ Basic sehr viel langsamer als bei den vergleichbaren Befehlen GOSUB und GOTO in Basic V2, da bei der Suche des Sprungziels alle Zeichen des Labels verglichen werden müssen. Um diesen Nachteil auszugleichen, legt TSB während der Ausführung eines Programms eine Tabelle aller PROC-Adressen an (ab Adresse $C400) und sucht PROCs zuerst in dieser Tabelle (siehe CHECK). Die Abarbeitungsgeschwindigkeit eines Programms mit vielen Prozeduren erhöht sich dadurch während des Betriebs signifikant und ist schneller als Basic V2.

Wird das Label hinter EXEC im Programm nicht gefunden, so erscheint die Fehlermeldung **NO PROC ERROR**. Wenn ein TSB-Programm auf ein Prozedurende (END PROC) trifft, ohne eine Prozedur aufgerufen zu haben, erscheint die Meldung **END PROC W/O EXEC ERROR**. Bei zu vielen Verschachtelungen (bei mehr als zehn) meldet TSB einen **OVERFLOW ERROR**.

Beachten: Beginnt ein Labelname mit einem Schlüsselwort, darf EXEC nicht weggelassen werden (ein Dezimalpunkt als erstes Zeichen hilft dann).

Beispiel:

```
10 PRINT "bitte eine taste druecken!"
20 warten
30 PRINT "danke"
999 END

1000 PROC warten
1010 DO NULL
1020 END PROC
```

(Nach der Aufforderung, eine Taste zu drücken, wartet das Programm, gibt schließlich eine Rückmeldung aus und endet)

Ein komplettes Anwendungsbeispiel für EXEC beim Befehl MOB SET (Beispiel 2). Noch einmal: Mit einem führenden Punkt kann man das Problem aushebeln (z.B.: PROC .move).

Befehl:	EXIT	
Syntax:	**EXIT IF ..**	
Zweck:	die Bedingung hinter EXIT definiert die Abbruchsbedingung	Struktur
Kürzel	exI	
Status:	Erweitertes Simons' Basic (Anweisung)	

EXIT definiert das logische Ende einer LOOP-Schleife. Eine Schleife ist ein Teil eines Programms, der u.U. mehrfach durchlaufen wird. Eine vorher definierte Bedingung hinter EXIT wird überprüft und an den Interpreter gemeldet. Wenn sie zutrifft, wird die Schleife beendet.

Mit LOOP .. EXIT .. END LOOP lassen sich alle Schleifentypen nachbilden: Endlosschleife (kein EXIT enthalten), kopfgesteuerte Schleife (siehe Beispiel), fußgesteuerte Schleife (siehe Beispiel bei REPEAT) und Zählschleife (mit FOR).

Beispiel für eine kopfgesteuerte Schleife:

```
10 A$ = ""
20 LOOP
30   EXIT IF PLACE(A$,"jnJN")
40   PRINT "Ja (j) oder Nein (n)? ";
50   FETCH"{crsr right}", 1, A$
60 END LOOP
```

Man kann alle Buchstaben eingeben, aber die Schleife wird erst verlassen, wenn „j“ oder „n“ getippt wurde (auch großgeschrieben). Wenn A$ vor der Schleife bereits einen der Tastendrücke enthielte, würde die Abfrage gar nicht erst stattfinden (z.B. das Beispiel mit A$="j" in Zeile 10 probieren).

EXIT darf im Schleifenkörper mehrfach auftreten.

Befehl:	EXOR	
Syntax:	**a = EXOR(<z1>,<z2>)**	
Zweck:	zwei Zahlen bitweise Exklusiv-Oder verknüpfen	Bearbeiten und Darstellen von Zahlen
Kürzel	exO	
Status:	Erweitertes Simons‘ Basic (numerische Funktion)	

Mit der numerischen Funktion **EXOR** werden zwei beliebige vorzeichenlose Ganzzahlen **<z1>** und **<z2>** (im Bereich von 0 bis 65535) bitweise Exklusiv-Oder verknüpft (ein gesetztes Bit in einer Zahl kehrt das entsprechende Bit in der anderen Zahl um).

Für eine Verwendung der Funktion EXOR als logischer Operator „ausschließendes Oder“ („nur wenn beide Seiten falsch sind, ist auch das Ergebnis falsch“) muss man bedenken, dass beim C64 der Wert **-1** (minus eins) dem logischen „wahr“ entspricht und der Wert **0** (null) ein logisches „falsch“ darstellt.

Argumente mit einem Wert über 65535 führen zur Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Beachten: Alle Simons'-Basic-Funktionen mit 16-Bit-Argumenten arbeiten nicht ordnungsgemäß, wenn sie als ***zweiter*** Parameter in einem **POKE**-Befehl verwendet werden. Die Werte müssen vor ihrer Verwendung bei POKE einer Variablen zugewiesen werden. Der Grund liegt darin, dass die auch in SB benutzte GETADR-Routine des zugrunde liegenden BASIC-V2-Interpreters keine zwei 16-bittigen Argumente in einem Ausdruck verarbeiten kann. (Behoben in TSB.)

Dieser Hinweis gilt auch für den Basic-V2-Befehl **WAIT**.

Befehl:	FCHR	
Syntax:	**FCHR <zl>,<sp>, ,<ho>,<bc>**	
Zweck:	Füllen eines Bildschirmbereichs mit einem Zeichen	Bearbeiten des Textbildschirms
Kürzel	fC	
Status:	Simons' Basic (Anweisung)	

Mit **FCHR** kann man Bereiche des Textbildschirms (linke obere Ecke an **<zl>** und **<sp>**, Breite und Höhe mit **
** und **<ho>**) oder den ganzen Textbildschirm (Parameterwerte: 0,0,40,25) mit einem bestimmten Zeichen **<bc>** (Wert in Bildschirmcode, 0..255) füllen. Die Farben, die dort schon vorher waren, bleiben dabei erhalten.

Nützlich, wenn man bestimmte Stellen auf dem Bildschirm (z.B. selbstdefinierte Fenster) löschen möchte. Dazu verwendet man für <bc> den Code 32 (das Leerzeichen).

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter einen **BAD MODE ERROR**.

Beispiel:

```
10 CLS: CENTER "demo fchr"
20 sp=29: zl=14: br=10: ho=10: ra=160: in=32
30 FCHR zl,sp,br,ho,ra
40 FCHR zl+1,sp+1,br-2,ho-2,in
50 PRINT AT(zl+2,sp+2) "-demo-"
60 PRINT AT(zl+2,sp+ho-2) "taste!"
70 DO NULL
80 FCHR zl+1,sp+1,br-2,ho-2,in
90 PRINT "{home}"
```

schreibt einen Rahmen in die rechte untere Bildschirmecke (Zeilen 30 und 40), wartet und löscht den Inhalt (Zeile 80)

Tipp: Mit einem POKE kann man mithilfe von FCHR in der hochauflösenden Grafik (Hires) die Hintergrund- und Punktfarbe rechteckiger Bereiche umfärben:

```
POKE 648,$C0:FCHR zl,sp,br,ho,farben: POKE 648,4
```

Der Wert "farben" setzt sich aus den zwei Nibbles für Vordergrund- und Hintergrundfarbe bei HIRES zusammen, z.B. $26 für Pixelfarbe rot (2) und Hintergrund blau (6). Die Parameter bei **FCHR** beziehen sich auch in diesem Fall auf den Textscreen (es sind keine Grafikkoordinaten).

Mit diesem Trick wäre schnelles Highlighting von mit TEXT in der Grafik ausgegebenen Texten möglich.

Befehl:	FCOL	
Syntax:	**FCOL <zl>,<sp>,<bt>,<ho>,<f>**	
Zweck:	Einfärben eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	fcO	
Status:	Simons' Basic (Anweisung)	

FCOL dient dazu, Bereiche des Textbildschirms (linke obere Ecke an **<zl>** und **<sp>**, Breite und Höhe mit **<bt>** und **<ho>**) oder den ganzen Textbildschirm mit allem, was darauf zu sehen ist, neu einzufärben, Farbe in **<f>** (betrifft also das Farb-RAM ab $D800). Die Farbe des Cursors ändert sich dadurch nicht.

Nützlich, wenn nach dem Umschalten auf den Multicolor-Modus mit dem Befehl MULTI die vorherigen Textfarben rekonstruiert werden müssen (MULTI verändert auch den Farbspeicher des Textbildschirms) oder zum Nachfärben von Spiele-Screens, die mit MAP aufgebaut wurden.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter einen **BAD MODE ERROR**.

Beispiel

```
10 COLOR 11,12: CLS: CENTER "demo"
20 PRINT AT(2,0) DUP("{shift-*}",40)
30 DO NULL
40 FCOL 0,0,40,25,1
```

Dies gibt eine Überschrift aus, wartet und färbt die Stelle um.

Befehl:	FETCH	
Syntax:	FETCH [AT(zl,sp)] <string>, <len>, <variable$> [, <mode>]	
Zweck:	kontrollierte Eingabe	Ein-/Ausgabe
Kürzel	fE	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

Die Anweisung **FETCH** ist eine gelungene Kombination aus den Fähigkeiten von GET und INPUT. Bei GET hat man die Möglichkeit, vom Programm aus zu überwachen, welche Zeichen eingegeben werden dürfen und wie viele. Und INPUT weist Eingaben vorher festgelegten Variablen zu. FETCH bietet beides zugleich, man kann kontrollieren, welche und wie viele Zeichen eingetippt werden, und diese werden komfortabel vom Befehl direkt einer gewünschten Variablen zugewiesen.

Der String-Ausdruck in Argument **<string>** gibt an, welche Tasten während der Eingabe erlaubt sein sollen, **<len>** definiert die Höchstlänge dieser Eingabe und in **<variable$>** wird die String-Variable angegeben, die das Ergebnis aufnehmen soll. Während man in Simons' Basic die maximale Eingabelänge größer als den zulässigen Wert 88 für den Eingabepuffer wählen kann, wird dies in TSB mit einer Fehlermeldung abgefangen. Bei Erreichen der vorgegebenen maximalen Eingabelänge (Parameter <len>) verschwindet in TSB der Eingabe-Cursor. Man kann jedoch die schon getippten Zeichen auch jetzt noch löschen. Erst die RETURN-Taste macht eine Eingabe endgültig.

Es ist in TSB - anders als bei Simons' Basic - auch möglich, gar kein Zeichen einzugeben (und nur die <RETURN>-Taste zu drücken), was bei der Ergebniszuweisung den Wert der bei FETCH verwendeten Variablen so belässt wie vor dem Aufruf von FETCH. Der optionale letzte Parameter **<mode>** erzwingt bei Aktivierung (mit einem Wert größer als Null) den Verzicht auf diese Eigenschaft. Dann muss man auf jeden Fall einen Wert eintippen, RETURN allein reicht nicht mehr.

Mit dem ebenfalls optionalen Parameter **AT(zl,sp)** legt man fest, an welcher Stelle auf dem Bildschirm der FETCH-Eingabe-Cursor erscheinen soll.

Um Platz zu sparen, gibt es drei Kontroll-Sonderzeichen, die jeweils für einen ganzen Bereich zulässiger Zeichen stehen. "**{home}**" oder CHR$(19) steht für alle Kleinbuchstaben (im Klein-Groß-Modus) bzw. Großbuchstaben (im Groß-Grafik-Modus), "**{crsr down}**" oder CHR$(17) steht für alle Ziffern und Satzzeichen einschließlich „@“. Und "**{crsr right}**" oder CHR$(29) steht für alle Klein- und Großbuchstaben bzw. Großbuchstaben und Grafikzeichen (je nach Anzeigemodus). Diese drei Sonderzeichen sind beliebig mit anderen Tasten kombinierbar, z.B. hätte man mit "{crsr right}0123456789 ." alle Buchstaben, die Ziffern, den Punkt und das Leerzeichen (und keine weiteren Sonderzeichen) als zulässig ausgewählt. Ein Leerstring als Kontrollstring erzeugt einen **ILLEGAL QUANTITY ERROR**.

Hinweis: Der INPUT-Befehl des C64-Kernals wird bei FETCH **nicht** verwendet, daher kommen dessen große Nachteile nicht zum Tragen (das Zeichen Komma bei der Eingabe führt nicht mehr zur Fehlermeldung **EXTRA IGNORED**, ein Doppelpunkt nicht mehr zur Meldung **REDO FROM START**, führende Leerzeichen sind jetzt möglich).

FETCH in TSB beachtet den **RVS-Modus**. Ist er an, werden die Eingaben korrekt invertiert angezeigt und stören so nicht die Bildschirmmaske des laufenden Programms.

Beispiel:

```
10020 PRINT "Beenden (b)? Fortsetzen (f)? ";: FETCH "fb",1,x$: PRINT
10030 IF x$ = "b" THEN PRINT: PRINT "Abbruch nach Fehler!": STOP
```

Befehl:	FILL	
Syntax:	**FILL <zl>,<sp>,<bt>,<ho>,<bc>,<f>**	
Zweck:	Füllen eines Bildschirmbereichs mit einem Zeichen in Farbe	Bearbeiten des Textbildschirms
Kürzel	fI	
Status:	Simons' Basic (Anweisung)	

Mit **FILL** kann man Bereiche des Textbildschirms (linke obere Ecke an **<zl>** und **<sp>**, Breite und Höhe mit **<bt>** und **<ho>**) oder den ganzen Textbildschirm (Parameterwerte: 0,0,40,25) mit einem bestimmten Zeichen **<bc>** (Wert in Bildschirmcode, 0..255) in einer bestimmten Farbe **<f>** füllen. Damit ist FILL eine Kombination aus FCHR und FCOL.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter einen **BAD MODE ERROR**.

Beispiel:

```
10 CLS: CENTER "DEMO FILL"
20 SP=29: ZL=14: BT=10: HO=10: RA=160: IN=32: F1=1: F2=0
25 COLOR F2,15
30 FCHR ZL,SP,BT,HO,RA
40 FCHR ZL+1,SP+1,BT-2,HO-2,IN
50 PRINT AT(ZL+2,SP+2) "-DEMO-"
60 PRINT AT(ZL+HO-2,SP+2) "TASTE!"
70 DO NULL
80 FILL ZL+1,SP+1,BT-2,HO-2,RA,F1: PAUSE 1
85 FILL ZL+1,SP+1,BT-2,HO-2,IN,F2
90 PRINT AT(ZL+HO-2,SP+2) "DANKE!"
100 PRINT CHR$(19);
```

Das Beispiel schreibt einen Rahmen in die rechte untere Bildschirmecke, wartet und lässt den Inhalt einmal weiß blinken.

Befehl:	FIND	
Syntax:	**FIND[“]<zeichenfolge>**	
Zweck:	Suchen von Zeichenfolgen im Programm	Programmierhilfen
Kürzel	fiN	
Status:	Simons‘ Basic (Kommando)	

Wer in seinem Programm (alle) Stellen sucht, an denen ein bestimmter String oder ein bestimmter Befehl steht, der muss **FIND** benutzen. FIND wird im Direktmodus verwendet, im Programm funktioniert das Kommando auch, aber sein Aufruf endet auch dort im Direktmodus.

Bei der Angabe von Suchzeichen in **<zeichenfolge>** muss man wissen, was intern beim Absenden des Suchauftrags passiert, damit man auch das findet, was man sucht: Nach Eingabe von <Return> auf der FIND-Zeile wandelt der Interpreter das FIND und alles, was darauf noch folgen sollte, in Tokens um, soweit das möglich ist. Sucht man also nach dem BASIC-Befehl INT, dann wird FIND auch fündig, wenn er im Programm vorkommt.

Sucht man allerdings nach der Zeichenfolge „INT“, z.B. innerhalb eines Strings, sagen wir im Wort „HINTERGRUNDFARBE“, dann muss nach FIND zunächst ein Anführungszeichen folgen, da es sonst nach dem Token für den Befehl INT sucht (das den Wert $B5 bzw. 181 hat). FIND"INT (ohne zweites Anführungszeichen) würde funktionieren.

FIND gibt als Ergebnis die Zeilennummern der Fundstellen im Tabulatorabstand aus, also vier Fundstellen-Zeilennummern pro Bildschirmzeile.

Beachten: Auch ein Leerzeichen zwischen FIND und dem eigentlichen Suchbegriff ist signifikant und wird mitgesucht (bei FIND ohne Anführungszeichen).

Beispiele:

```
FINDwasser
```
(durchsucht das Programm nach der Zeichenfolge "wasser")

```
FIND
```
(listet ausnahmslos **alle** Programmzeilennummern)

```
FINDpoke
```
(zeigt die Stellen an, die den Basic-Befehl POKE enthalten)

FIND“int
(zeigt alle Stellen mit der Buchstabenfolge I, N und T)

Befehl:	FLASH	
Syntax:	**FLASH <f> [, <sp>]** **FLASH OFF**	
Zweck:	Blinken von Zeichen	Bearbeiten des Textbildschirms
Kürzel	fL	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

FLASH lässt alle Zeichen in der angegebenen Farbe **<f>** mit einer gewünschten Geschwindigkeit **<sp>** blinken. Die Alternativfarbe beim Blinkvorgang ergibt sich aus der Hintergrundfarbe des Zeichens, die von COLOR oder BCKGNDS abhängt. Der Parameter <sp> legt die Blinkgeschwindigkeit fest, wobei die eingegebene Zahl der Anzahl von **sechzigstel Sekunden** (Jiffies) entspricht. Bei einem Wert von 60 für <sp> blinken die Zeichen also einmal pro Sekunde. Dieser Parameter kann auch weggelassen werden. In diesem Fall verwendet der Interpreter die zuletzt gewählte Geschwindigkeit bzw. 0 (die langsamste), wenn vorher noch keine definiert war.

Beachten bei Simons‘ Basic: Die Ausführung des Befehls findet im Interrupt statt, das Basic-Programm läuft weiter. Der Programmierer hat nach der Aktivierung keinen Einfluss mehr auf FLASH. Auch das Ende des Befehlslaufs ist in Simons' Basic nicht synchronisiert, die zuletzt angezeigte Blinkphase hängt daher vom Moment des Ausführens des Befehls OFF ab. Wenn ein Programm vorzeitig abbricht (<RUN/STOP>-Taste gedrückt oder Laufzeitfehler), muss das Blinken von Hand mit OFF ausgeschaltet werden, da der Interpreter es auch im Direktmodus weiterlaufen lässt. Da der Befehl nach Ablauf der eingestellten Zeit <sp> jedes Mal den ganzen Bildschirm nach zu flashenden Zeichen durchsucht, wird das Updaten der TI-Zeit durch FLASH beeinträchtigt.

In TSB wurden die genannten Synchronisationsprobleme behoben. Zusätzlich akzeptiert es auch die klarere Formulierung FLASH OFF (Blinken beenden). FLASH ON ist nicht implementiert.

Farbangaben größer als 15 werden nicht akzeptiert und führen zu einem **SYNTAX ERROR**.

Beispiel:

```
100 COLOR 7,2,1: BFLASH 1,7,6: CLS
110 FOR x=0 TO 39
120 y=x/2
130 PRINT AT(y,x)"*" AT(y,39-x)"*"
140 PRINT AT(0,x)"*" AT(20,x)"*"
150 PRINT AT(y,0)"*" AT(y,39)"*"
160 NEXT
170 FLASH 1,15
180 PRINT AT(22,0)"";: CENTER "Demo von": PRINT
190 CENTER "Flash und BFlash"
200 PRINT AT(22,0)"";: PAUSE 5
210 FLASH OFF: PAUSE 5: BFLASH OFF: COLOR 11,12,0
```

Das Beispiel schreibt einen Kasten mit weißen Sternen auf rotem Grund mit gelbem Rahmen, lässt alles blinken und beendet nach jeweils 5 Sekunden das Blinken der Zeichen und des Rahmens.

Befehl:	FRAC	
Syntax:	**a = FRAC(<z>)** **PRINT FRAC(<z>)**	
Zweck:	Nachkommastellen einer Zahl ermitteln	Bearbeiten und Darstellen von Zahlen
Kürzel	fR	
Status:	Erweitertes Simons‘ Basic (numerische Funktion)	

Die numerische Funktion **FRAC** (von engl. „fraction“) ermittelt die Nachkommastellen (den Dezimalanteil) einer Fließkommazahl des Arguments **<z>**. Das Vorzeichen bleibt dabei erhalten. Die Berechnung entspricht dem recht aufwändigen BASIC-Ausdruck `SGN(<z>)*(ABS(<z>) – INT(ABS(<z>)))`, für positive Werte reicht hingegen `<z>–INT(<z>)`.

Beispiel:

```
10 INPUT "ZAHL";Z
20 F = FRAC(Z/4): G = INT(Z/4)
30 PRINT "DIVISION DURCH 4: GANZAHLANTEIL " G ", NACHKOMMASTELLEN " F
```

Von einer Zahl wird berechnet, was nach einer Division durch 4 **vor und hinter** dem Komma steht.

Befehl:	GLOBAL	
Syntax:	**GLOBAL**	
Zweck:	Deaktivieren lokal gültiger Variablen	Programmierhilfen
Kürzel	gL	
Status:	Simons‘ Basic (Anweisung)	

Mit **GLOBAL** werden alle lokalen Variablen deaktiviert und eventuell vorhandene gleichnamige Variablen des Hauptprogramms wieder zugänglich gemacht.

Beispiel:

```
100 a=1: b%=10 : c$="test"
110 PRINT "dies ist ein test:"
120 PRINT "a="a; "b%="b%; "c$= "c$
130 prozedur
140 PRINT "zurueck im hauptprogramm:"
150 PRINT "a="a; "b%="b%; "c$= "c$
160 PRINT "test beendet."
170 END
999 :
1000 PROC prozedur
1010 LOCAL a, b%, c$
1020 a=100: b%=-10: c$="lokal"
1030 PRINT "und hier innerhalb der prozedur: "
1040 PRINT "a="a; "b%="b%; "c$= "c$
1050 GLOBAL
1060 END PROC
```

(Die PRINT-Ausgabe wird dreimal wiederholt, vor, während und nach der lokalen Phase)

Befehl:	GRAPHICS	1
Syntax:	**GRAPHICS**	
Zweck:	installiert die Erweiterung High-Speed-Grafik	Ein-/Ausgabe
Kürzel	gR	
Status:	Neuer TSB-Befehl (Kommando)	

GRAPHICS lädt die Grafikerweiterung „High-Speed-Grafik“ („HSG“) von Wolfgang Lohwasser aus dem 64'er-Sonderheft „Grafik“ 6/1986 (in einer stark für TSB angepassten Version). Die HSG-Datei muss unter dem Namen „**tsb.hsg**“ auf der im aktuellen Laufwerk befindlichen Diskette sein. Die Erweiterungsbefehle (s.u.) funktionieren wie normale Interpreter-Befehle. Der freie BASIC-Speicher wird durch HSG noch einmal um drei KByte verkleinert.

HSG wird im Speicher ab $7400 abgelegt und von TSB automatisch eingebunden. Es wird mit dem Befehl COLD wieder abgeschaltet.

Wird die Datei nicht gefunden, meldet der Interpreter einen **FILE NOT FOUND ERROR** und alles bleibt, wie es vorher war.

HSG arbeitet ausschließlich im Hires-Modus (kein Multicolor!), dafür aber sind seine Zeichenroutinen - vor allem die Kreise - derartig schnell, dass man das Entstehen eines einzelnen Objekts nicht mehr nachvollziehen kann (was bei TSB und Simons' Basic immer geht). Das Beispiel bei CIRCLE ist mit HSG-Befehlen einfach atemberaubend.

Beachten: HSG stellt eigentlich zwei Grafikbildschirme zur Verfügung (einen zum verdeckten Aufbau von Grafik, damit es noch schneller aussieht, wenn etwas erscheint). Dieses Feature wurde in TSB aus Platzmangel komplett herausgenommen. Damit TSB-Grafikbefehle und HSG miteinander kooperieren können, sollte der Grafikbildschirm mit HIRES eingeschaltet werden.

Hinweis: Der Zeichensatz im Normaltextbereich des Splitscreens kann mit `POKE $79e0,21` auf Groß-Grafik gestellt werden, und mit `POKE $79e0,23` auf Groß-Klein (voreingestellt).

HSG-Befehle

Nach dem Aufruf von GRAPHICS erhält man einen Einschalt-Screen und kann jetzt mit mehr Befehlen programmieren (die Screen-Auswahlparameter in der eckigen Klammer sind jetzt nicht mehr erforderlich):

```
£h [,1]           ⇒  Grafikscreen an, ohne zu löschen (s. CSET 2)
£e [,1]           ⇒  löscht den Screen (s. HIRES)
£m [,1]           ⇒  Screen anwählen (nach FCHR und FCOL nötig!)
£c [,1],f         ⇒  färbt die Grafik (s. MOD)
£n                ⇒  Grafikscreen aus (s. NRM)
£s,x,y            ⇒  Grafikcursor auf x,y setzen
£d,x,y,m          ⇒  zeichnen bis zu Punkt x,y im Modus m
£p,x,y,m          ⇒  Punkt setzen im Modus m (s. PLOT)
£l,x1,y1,x2,y2,m  ⇒  Linie von x1,y1 nach x2,y2 im Modus m (s. LINE)
£r,x1,y1,x2,y2,m  ⇒  Rechteck zeichnen (s. REC)
                     (beachten: andere Parameter als bei REC!)
£b,x1,y1,x2,y2,m  ⇒  ausgefülltes Rechteck zeichnen (s. BLOCK)
£k,x,y,r1,r2,m    ⇒  Kreis, Ellipse zeichnen (s. CIRCLE)
```

```
£is,top,bottom    ⇒  Splitscreen, Grafik zwischen Textzeile <top>
                      und Textzeile <bottom> (max. <bottom> = 25)
£i1               ⇒  Splitscreen einschalten
£i0               ⇒  Splitscreen ausschalten
£v,v1,v2$         ⇒  numerischen Ausdruck im String <v2$> auswerten
                      und <v1> zuweisen
```

```
f = paper+16*ink, z.B. f=$75 für gelb auf grün
m:     =1: zeichnen
       =0: löschen
```

Vor allem **£l**, **£r** und **£k** sind erheblich schneller als ihre TSB-Pendants.

Programm in HSG:

```
10 i=10: HIRES i,0: COLOR i,0
15 REPEAT: £E,1: GET x$
20 FOR x=5 TO 165 STEP i: £K,159,99,x,165-x,1: NEXT
30 FOR x=5 TO 165 STEP i: £K,159,99,x,165-x,0: NEXT
40 i=INT(RND(ti)*20)+1: j=INT(RND(ti)*14)+1
50 MOD 0,j: COLOR 0,i
60 FOR x=0 TO 149 STEP 10+i
70 FOR y=189 TO 0 STEP -10-i
80 £L,x,y,310-x,190-y,1
90 NEXT y,x
100 FOR y=189 TO 0 STEP -10-i
110 FOR x=0 TO 149 STEP 10+i
120 £L,x,y,310-x,190-y,0
130 NEXT x,y
140 i=INT(RND(ti)*20)+1: j=INT(RND(ti)*14)+1
150 UNTIL x$>""
```

Unglaublich viele Ellipsen in unglaublich kurzer Zeit. (Programm mehrmals laufen lassen, die Farben verändern sich.)

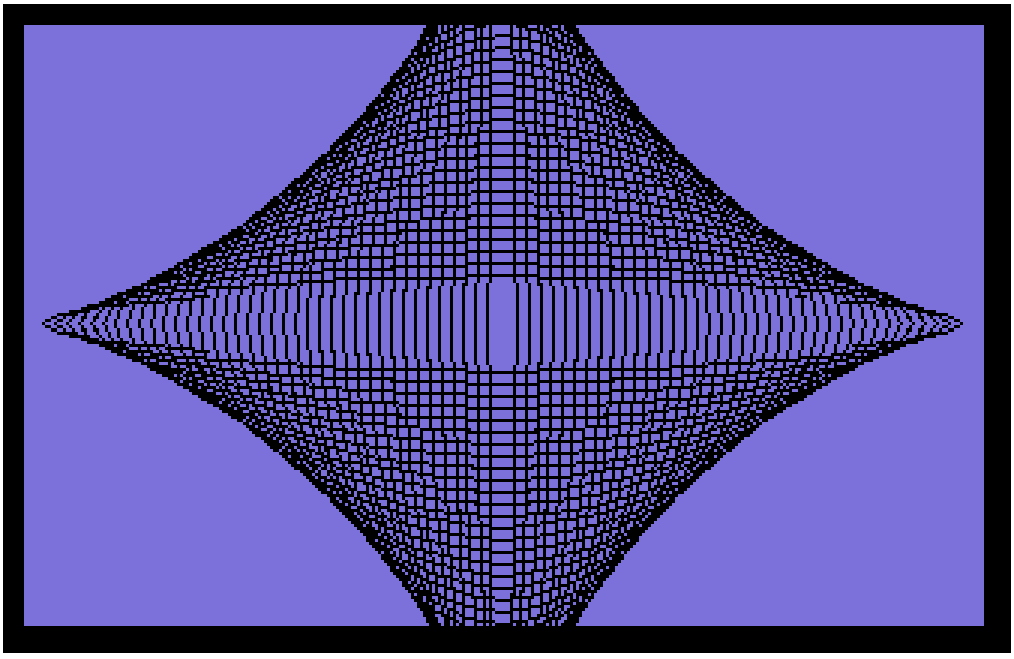


Bild 13 Wahnsinnsgeschwindigkeit mit Bresenham

Befehl:	GRAPHICS	2
Syntax:	**a = GRAPHICS** **PRINT GRAPHICS**	
Zweck:	Gibt die Basisadresse des VIC zurück	Programmierhilfen
Kürzel	gR	
Status:	Simons' Basic (Systemkonstante)	

GRAPHICS liefert als Systemkonstante die Speicheradresse zurück, an der der Video-Chip des C64 (VIC) liegt und kontrolliert werden kann: 53248 bzw. $D000.

Beispiel:

```
10 POKE GRAPHICS+17, PEEK(GRAPHICS+17) AND NOT 16
20 DO NULL
30 POKE GRAPHICS+17, PEEK(GRAPHICS+17) OR 16
```

Der Bildschirm wird abgeschaltet und nach Tastendruck wieder eingeschaltet.

Befehl:	HI COL	
Syntax:	**HI COL**	
Zweck:	Schaltet LOW COL aus	Grafik-Befehle
Kürzel	hi C	
Status:	Simons' Basic (Anweisung)	

HI COL schaltet die lokalen Farben von LOW COL wieder aus, so dass danach die generell gültigen Farbsetzungen durch COLOR, HIRES und MULTI wieder in Kraft treten.

Beispiel:

```
10 COLOR 5,7: HIRES 10,1: MULTI 12,15,1
20 BLOCK 0,0,50,100,2: LOW COL 5,15,0
30 LINE 0,100,50,0,1: HI COL
40 LINE 0,0,50,100,3: DO NULL
50 COLOR 11,12,0
```

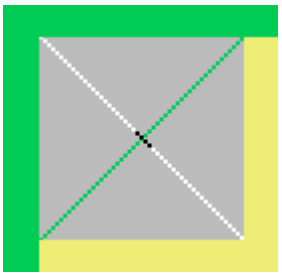


Bild 14 HI COL

Kommentar:

Zeile 10: schaltet die Grafik ein
Zeile 20: malt ein hellgraues Quadrat in <farbe2> (der letzte Parameter von BLOCK bezieht sich auf <farbe2> bei MULTI: 15) auf gelbem Hintergrund (7 von COLOR), dann wird per LOW COL umgefärbt in grün, hellgrau und schwarz;
Zeile 30: die erste Linie erscheint in der LOW COL-Farbe 1 (grün)
Zeile 40: und die zweite wegen HI COL in Farbe weiß (<farbe3> bei MULTI), wie der letzte Parameter von LINE angibt; die fehlerhaften schwarzen Punkte sind ein Überbleibsel von LOW COL <farbe3> (0 = schwarz) und unvermeidlich, da <farbe3> jetzt nicht mehr die HI-COL-Farbe weiß ist
Zeile 50: wechselt in den Textmodus und stellt die Textbildschirmfarben wieder her

Befehl:	HIRES	
Syntax:	**HIRES <ink>, <paper>**	
Zweck:	Einschalten und Färben der hochauflösenden Grafik, löschen der Grafik	Grafik-Befehle
Kürzel	hI	
Status:	Simons‘ Basic (Anweisung)	

HIRES dient dazu, den C64 vom Text- in den hochauflösenden Grafikmodus umzuschalten (320×200 Pixel) und die angegebenen Farben zu verwenden: **<ink>** für die Schreibfarbe und **<paper>** für die Farbe des Hintergrunds (beide mit Werten von 0 bis 15). Der Grafikspeicher (an Adresse $E000) wird dabei gelöscht und mit den angegebenen Farben vorbesetzt. Der Farbspeicher für Hires-Bilder liegt in Simons’ Basic an der Adresse $C000. Die Farbe des Bildschirmrahmens bleibt unbeeinflusst. Im Direktmodus springt Simons' Basic sofort nach Ausführung des Befehls zurück in den Textmodus. In einem Programm bleibt der gewählte Modus so lange aktiviert, bis er per Befehl aus- oder umgeschaltet wird (MULTI, CSET oder NRM).

Wichtig: Die Farbangaben fast aller weiteren Grafikbefehle beziehen sich auf die hier festgelegten Farben. Bei Angabe der **Farbquelle „0“** in einem Grafikbefehl wird die für **<paper>** angegebene Farbe ausgewählt, bei einer Angabe von „**1**“ dagegen die Farbe für **<ink>**, die Farbangabe „**2**“ **invertiert** den angesteuerten Pixel (wenn im Hires-Modus, ansonsten siehe MULTI).

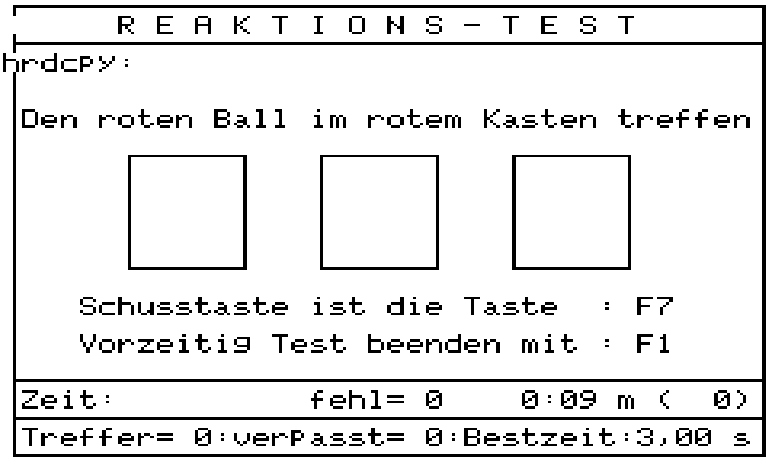
Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Befehl:	HRDCPY	
Syntax:	**HRDCPY**	
Zweck:	Ausgabe des Textbildschirms auf Drucker MPS 801	Ansprechen von Peripheriegeräten Ein-/Ausgabe
Kürzel	hR	
Status:	Erweitertes Simons' Basic (Kommando)	

HRDCPY druckt den Textbildschirm, der sich an der Speicherposition befindet, die die Adresse $0288 angibt auf einem angeschlossenen Drucker **MPS 801** (oder kompatibel) aus. Es benutzt dazu intern ein OPEN mit der laufenden Dateinummer 101 (`OPEN 101,4:CMD 101` – Simons' Basic verwendet Dateinummer 1) und schließt diesen Kanal auch wieder.

Simons' Basic druckt ohne einen linken Rand, außerdem werden nur die Zeichen auf dem Standard-Screen ($0400) erfasst. TSB druckt zentriert und jeden Textscreen.

Mit einem POKE kann man den Abstand der Ausgabezeilen so anpassen, dass untereinanderstehende Zeilen **ohne Lücke** gedruckt werden (s. Beispielbild „Reaktions-Test"): POKE $B734, 6 (statt 5, s. Beispielbild „Listing", bis v2.31.113 lautet die Adresse: $B54B).



Ohne Lücken zwischen den Zeilen (gut bei Grafikzeichen)

```
115 CENTREAT(0,24)„♠BBRUCH: LEERES <_ETU
RN>„:PRINTAT(2,4)„HEX„
120 A$=„„:FETCHAT(6,4)„0123456789ABCDEF„
,4,A$
130 L=LEN(A$):X=0:F=1:EG=0:PRINT:IFA$>„„
THENDO
140 FORI=1TOL:X$=MID$(A$,I,1)
145 X$=RIGHT$(„0„+X$,2):X1=NRM(X$):F=16↑
(L-I):X=X1*F:EG=EG+X
150 USEAT(4,LIN)„+## X #### = #####„,X1,
F,X
155 IFI=1THENPRINTAT(4,LIN-1)„ „
160 NEXT:PRINTAT(17,LIN)DUP(„-„,5)
170 USEAT(15,LIN)„= ##### DEZ„,EG
175 FCHR24,0,40,1,32:CENTREAT(0,24)„|AST
E„
180 DO NULL
190 DONE
200 UNTILA$=„„
210 CLS:CENTREAT(0,12)„|IS ZUM NAECHSTEN
 \AL!„:PRINTAT(0,21)„„
220 END
CSET0:POKE$B567,15:HRDCPY
```

Mit Lücken (bei reinen Textbildschirmen wie Listings).

Befehl:	INKEY	
Syntax:	**a = INKEY** **PRINT INKEY**	
Zweck:	F-Tasten abfragen	Programmierhilfen Struktur
Kürzel	inK	
Status:	Simons' Basic (Systemvariable)	

INKEY fragt die Nummer der gerade gedrückten Funktionstaste ab. Damit hat man die einfache Möglichkeit, Menüverzweigungen über Funktionstasten zu programmieren.

Weder die Anweisung KEY 0 bzw. KEY OFF noch der sich dahinter verbergende POKE zum Abschalten von DISPLAY (`POKE $C646,10`) haben Einfluss auf INKEY.

Beispiel:

```
10 a = INKEY: IF a = 0 THEN 10
20 PRINT "f-taste nr." a
30 GOTO 10
```

Wartet auf das Drücken einer F-Taste und gibt ihre Nummer aus.

Befehl:	INSERT	1
Syntax:	INSERT <string>,<zl>,<sp>,<bt>,<ho>,<f>	
Zweck:	Rahmen zeichnen (im Textmodus)	Bearbeiten des Textbildschirms
Kürzel	insE	
Status:	Neuer TSB-Befehl (Anweisung)	

Mit **INSERT** kann man im Textmodus Rahmen auf den Bildschirm malen.

Das Aussehen des Rahmens wird vom ersten Parameter **<string>** bestimmt. Er muss eine Zeichenkette enthalten, die **genau neun** Zeichen lang ist und aus den Zeichen besteht, die den Rahmen definieren: linke obere Ecke, obere Kante, rechte obere Ecke, linke Kante, Füllzeichen, rechte Kante, linke untere Ecke, untere Kante, rechte untere Ecke. Ist dieser String zu lang oder zu kurz, meldet TSB einen **ILLEGAL QUANTITY ERROR**. Bei aktiviertem RVS-Modus (CTRL-9) werden auch alle INSERT-Zeichen invertiert ausgegeben.

Zusätzlich eingebaut ist ein auf einzelne Zeichen bezogener RVS-Modus, der für Zeichen gedacht ist, die symmetrisch auf gegenüberliegenden Seiten eines Rahmens eingesetzt werden sollen. Dafür braucht man jeweils einmal ein invertiertes Zeichen, z.B. C= k für einen linken und rechten Rand oder C= i für einen oberen und unteren Rand. Da bei INSERT nur neun Rahmendefinitionszeichen zugelassen sind (Steuerzeichen fürs Invertieren also nicht vorkommen dürfen), sind für zu invertierende Zeichen die 32 Zeichen ab CHR$(224) vorgesehen (die Zeichen mit C=-Taste).

Die nächsten beiden Parameter legen den Ort der linken oberen Ecke des Rahmens fest: **<zl>** = Zeile (Wertebereich 0 bis 23) und **<sp>** = Spalte (Wertebereich 0 bis 38). Die beiden folgenden bestimmen seine Größe: **<bt>** = Breite (Wertebereich 2 bis 40) und **<ho>** = Höhe (Wertebereich 2 bis 25). Wird einer dieser Werte unter- oder überschritten, meldet der Interpreter einen **BAD MODE ERROR**. Eine Breiten- oder Höhenangabe von 1 ist dennoch möglich, wird aber automatisch auf 2 erhöht, weil sonst kein Kasten entstehen kann. Eine Breite oder Höhe von 2 lässt die Kanten weg und gibt nur die Ecken aus.

Der letzte Parameter (**<f>**) definiert die Farbe der Rahmen (Wertebereich: 0..15).

Bild 15 INSERT macht Rahmen

Fünf verschiedene INSERT-Rahmen im Bild.
Rand a**ußen, in hellgrau**: überall 4 Pixel dick,
in braun: nach innen hin 4 Pixel dick, mit einer Verzierung an den Ecken,
in dunkelgrau: mittig 2 Pixel dick, mit gefüllten Quadraten an den Ecken,
in rot: 2 Pixel dick nach außen, mit 4 Pixel dicken, abgesetzten Ecken, und wieder
in hellgrau: 3 Pixel dick nach außen, mit quadratisch gefüllten Ecken.

Will man **alle 256 Zeichen** eines Zeichensatzes verwenden, hilft **D!POKE $A117,$27D0** (rückgängig mit **$C7A5**) und ein Füllen des INSERT-Puffers (**$0140**) mit den neun (BCode-) Zeichen, aber von hinten nach vorn. In diesem Fall kann auch jedes der Zeichen einzeln invertiert erscheinen.

Befehl:	INSERT	2
Syntax:	a$ = INSERT(<string>,<altstring>,<pos>) PRINT INSERT(<string>,<altstring>,<pos>)	
Zweck:	Erweiterung eines Strings	Stringfunktionen
Kürzel	insE	
Status:	Erweitertes Simons‘ Basic (Stringfunktion)	

INSERT setzt die Zeichenkette **<string>** (erstes Argument) in die Zeichenkette **<altstring>** (zweites Argument) ab der Stelle **<pos>** (drittes Argument) ein, wobei die Zählung anders als in Simons‘ Basic mit 1 beginnt (1 = erstes Zeichen, Simons‘ Basic: 0!) Die resultierende Zeichenkette ist so lang wie die Summe der beiden Einzelzeichenketten.

Beachten: Wenn die resultierende Zeichenkette die Länge von 255 Zeichen übersteigt, melden Simons' Basic und TSB einen **INSERT TOO LARGE ERROR**. Derselbe Fehler wird bei SB gemeldet, wenn <string> an <altstring> angehängt werden soll (das Argument <pos> also den Wert `LEN(<altstring>)` oder größer) hat. Ist das zweite Argument ein Leerstring, besteht die resultierende Zeichenkette in Simons‘ Basic aus 255 zufälligen Zeichen, dieser Fall wird also nicht abgefangen und führt unter Umständen zum Absturz des Interpreters.

In TSB sind die beiden letzten Fehler behoben: INSERT kann Strings verlängern und das Einsetzen von Leerstrings liefert einen unveränderten Ausgangsstring zurück.

Beispiel:

```
10 a$="(spitzname) ": b$="vorname nachname"
20 c$=INSERT(a$,b$,9)
30 PRINT c$
```

Die resultierende Ausgabe ist "vorname (spitzname) nachname".

Beachten: Im Direktmodus unter Verwendung von literalen Strings (direkt verwendeter Text in Anführungszeichen) liefert diese Funktion falsche Ergebnisse, deshalb gibt es unter TSB im Direktmodus die Fehlermeldung **ILLEGAL DIRECT ERROR**. Hinweis: Man kann den Direktmodus mit POKE $9D,0 abschalten und so die Funktion auch ohne Programm ausführen, denn String**variablen** werden auch im Direktmodus korrekt verarbeitet.

Befehl:	INST	1
Syntax:	**INST**	
Zweck:	installiert DOS Wedge 5.1	Programmierhilfen Ein-/Ausgabe
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

INST lädt das DOS Wedge 5.1 nach (das Vorbild für die JiffyDOS-Befehlsabkürzungen). Die Wedge-Datei muss unter dem Namen „**tsb.ext**“ auf der im aktuellen Laufwerk befindlichen Diskette sein. Die Wedge-Befehle funktionieren nur im Direktmodus des Interpreters.

Das Wedge wird im Speicher ab $CC00 abgelegt und automatisch eingebunden. Die Befehle RENUMBER und PLACE werden dabei deaktiviert, da sie den gleichen Speicherplatz wie das Wedge belegen (ihr Aufruf führt bei installiertem DOS Wedge nun zu einem **BAD MODE ERROR**). Das DOS Wedge wird mit dem eigenen Befehl @q bzw. mit dem TSB-Befehl NRM abgeschaltet, was RENUMBER und PLACE wieder zugänglich macht (dazu muss die Datei „**tsb.mem**“ auf der aktuellen Diskette sein).

Wird eine der Dateien nicht gefunden, meldet der Interpreter einen **FILE NOT FOUND ERROR** und die beiden Befehle PLACE und RENUMBER bleiben deaktiviert.

DOS-Wedge-Befehle:

@	Floppy-Fehlermeldung abholen und ausgeben
@$	Directory ausgeben
@befehl	DOS-Befehl ausführen (Befehle: s, n, v, i, r, c)
@q	DOS Wedge 5.1 beenden
@#9	Umschalten auf Drive 9 (oder 10 oder…)
/datei	Laden eines Basic-Programms
↑datei	Laden und Starten eines Basic-Programms
%datei	Laden eines Maschinenspracheprogramms
←datei	Speichern eines Basic-Programms

Befehl:	INST	2
Syntax:	a$ = INST(<string>,<altstring>,<pos>) PRINT INST(<string>,<altstring>,<pos>)	
Zweck:	Veränderung eines Strings	Stringfunktionen
Kürzel	inS	
Status:	Erweitertes Simons‘ Basic (Stringfunktion)	

INST überschreibt mit der Zeichenkette **<string>** (erstes Argument) in der Zeichenkette **<altstring>** (zweites Argument) ab der Stelle **<pos>** (drittes Argument) die Zeichen der Zeichenkette <altstring>, wobei die Zählung anders als in Simons‘ Basic mit 1 beginnt (1 = erstes Zeichen; Simons‘ Basic: 0!) Die Länge der Zeichenkette <altstring> ändert sich nicht.

Beachten: Wenn die Zeichenkette <string> an eine Stelle in <altstring> geschrieben werden soll, die <altstring> verlängern würde, wird Simons' Basics String-Verwaltung gestört, was zu einem Absturz führen kann. Ist das zweite Argument ein Leerstring, besteht in Simons‘ Basic die resultierende Zeichenkette aus 255 zufälligen Zeichen. Dies kann ebenfalls zu einem Absturz führen.

Beides ist in TSB behoben, es wird im ersten Fall mit der Fehlermeldung **STRING TOO LONG ERROR** abgebrochen. Im zweiten Fall geschieht gar nichts.

Beispiel:

```
10 v$=" vor": n$="nachname"
20 r$=INST(v$,n$,1)
30 PRINT "(";r$;")

Die Variable r$ wird ausgegeben und zeigt dann „( vorname)“.
```

Beachten: Im Direktmodus und bei Verwendung von literalen Strings (direkt verwendeter Text in Anführungszeichen) liefert diese Funktion falsche Ergebnisse, deshalb gibt es unter TSB im Direktmodus die Fehlermeldung **ILLEGAL DIRECT ERROR**. Hinweis: Man kann den Direktmodus mit POKE $9D,0 abschalten und so die Funktion auch ohne Programm ausführen.

Befehl:	INV	
Syntax:	**INV <zl>,<sp>,<bt>,<ho>**	
Zweck:	Invertieren eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	-	
Status:	Simons' Basic (Anweisung)	

INV invertiert Bereiche des Textbildschirms (oder den ganzen Textbildschirm). Die Farben, die dort schon vorher waren, bleiben erhalten, da die Zeichen selbst manipuliert werden (Bit 7 der Bytes im Bildschirmspeicher wird umgeschaltet).

Nützlich, wenn man bestimmte Stellen auf dem Bildschirm (z. B. selbstdefinierte Fenster) blinken lassen oder hervorheben möchte, ebenso beim Hervorheben von Menütexten.

Bei Über- oder Unterschreitung der zulässigen Werte (siehe Box) gibt der Interpreter die Fehlermeldung **BAD MODE ERROR** aus.

Befehl:	JOY	
Syntax:	a = JOY[(<n>)] PRINT JOY [(<n>)]	
Zweck:	ermittelt den Zustand des Joysticks in einem der Ports	Ansprechen von Peripheriegeräten
Kürzel	j0	
Status:	Erweitertes Simons' Basic (numerische Funktion)	

JOY liest den Wert eines der beiden C64-Controlports (CIA 1, $DC00 und $DC01) aus, um einen dort angeschlossenen Joystick für das Programm nutzbar zu machen. Die Werte, die der Port liefert, werden übersetzt in Werte zwischen 0 und 8, wobei jeder Wert einer bestimmten Richtung zugeordnet ist. Wenn der Feuerknopf betätigt wird, addiert sich ein Wert von 128 zum Richtungswert:

Wert	mit Feuer	Richtung
0	128	-
1	129	N
2	130	NO
3	131	O
4	132	SO
5	133	S
6	134	SW
7	135	W
8	136	NW

Gegenüber Simons' Basic hat die TSB-Version des Befehls zwei Vorteile. Der wichtigste ist die Tatsache, dass beide Controlports ausgelesen werden können. Der zweite ist, dass unter TSB die Nummer des Controlports (1 oder 2) als Argument für die Funktion angegeben werden kann. Somit sind Verwechslungen nicht mehr so leicht möglich. Die Funktionsargumente löst TSB dahingehend auf, ob sie geradzahlig (Controlport 2) oder ungeradzahlig sind (Controlport 1), wobei der Wert 0 als geradzahlig betrachtet wird.

Beispiel:

```
1600 PROC stick
1610   x7=JOY(1):
       IF x7=0 THEN END PROC
1620   IF x7=1 THEN x$="{crsr up}"
1630   IF x7=3 THEN x$="{crsr right}"
1640   IF x7=5 THEN x$="{crsr down}"
1650   IF x7=7 THEN x$="{crsr left}"
1660   IF x7>127 THEN x$=CHR$(13)
1670 END PROC
```

Diese Abfrageroutine ist sinnvoll in Programmen mit kombinierter Joystick-Tastatur-Bedienung und liest einen Joystick in Port 1 aus. Eine solches Programm müsste die folgende Hauptabfrageroutine für Eingaben aufweisen (Ausschnitt):

```
1330     REPEAT:
           IF lg>0 THEN PRINT "_";
1332       GET x$:
           stick:
           IF x$="" THEN 1332
1335       IF x$=CHR$(20) THEN IF l THEN l=l-1:
             x9$=LEFT$(x9$,l):
             x$=x9$+" ":
             set:
             x$=""
1337     UNTIL x$>"":
```

Kommentar: In LG steht die zulässige Gesamtlänge der Eingabe, in L die aktuelle Länge. In X9$ wird die Summe der Eingabe-Tastendrücke zusammengestellt. CHR$(20) ist das Backspace-Zeichen und löscht den rechten Rand von X9$, solange dort etwas vorhanden ist. Der Befehl „stick“ fragt – wie oben beschrieben – den Joystick ab. Die Routine „set“ in Zeile 1335 (hier nicht abgedruckt) zeigt den aktuellen Ergebnisstring X9$ an.

Hinweis: Wer will, kann mit dem `POKE $91A1,208` die Reaktion von **JOY** so abändern, dass die JOY-Funktion bei Anwendung so lange wartet, bis der Joystick tatsächlich bewegt wird (ähnlich KEYGET). In dem Fall ist aber natürlich eine gleichzeitige Abfrage von Joystick und Tastatur wie im obigen Beispiel nicht mehr möglich. Der POKE wird mit dem Wert 103 an die gleiche Stelle wieder zurückgenommen.

(Die Formatierung der obigen Listings wurde mit dem Programm „BLister“ auf der TSB-Diskette vorgenommen.)

Befehl:	KEY	
Syntax:	**KEY <n>, <string>** **KEY ON | OFF**	
Zweck:	Belegen der F-Tasten	Ein-/Ausgabe Struktur
Kürzel	kE	
Status:	Erweitertes Simons‘ Basic (Kommando)	

Mithilfe von **KEY** können die Funktionstasten mit Texten belegt werden, die beim Drücken der entsprechenden F-Taste auf dem Bildschirm zur Anzeige kommen. Diese Texte dürfen aus allen zulässigen Zeichen zusammengesetzt sein. Sinnvoll sind z.B. F-Tastenbelegungen, die bestimmte, oft verwendete BASIC-Befehle produzieren, etwa eine persönliche Farbeinstellung (wie `COLOR 11,12,0` o.ä.) Ein zulässiges Zeichen ist auch die Return-Taste, die als „←“ (Linkspfeil) an den **<string>** angehängt wird. TSB stellt nun nicht nur acht, sondern mit **<n>** sogar 16 verschiedene Funktionstastenbelegungen zur Verfügung. Die F-Tasten ab Nummer 9 erreicht man, wenn man zusätzlich die Commodore-Taste gedrückt hält: F9 ist die Tastenkombination <C= F1> , F10 ist <C= Shift F1> usw. Eine Belegung kann bis zu 16 Zeichen umfassen, mehr werden mit **STRING TOO LONG ERROR** abgewiesen.

Eine falsche Angabe bei Parameter <n> führt zu einem **BAD MODE ERROR**.

KEY kann auch innerhalb von Programmen verwendet werden. Auf diese Weise kann man sich eine rekonstruierbare Funktionstastenbelegung schaffen. Eine Funktionstastenbelegung kann man (in TSB) mit einem Leer-String wieder löschen.

TSBs F-Tasten sind bereits vorbelegt mit diesen Befehlen:

```
KEY 1, "PAGE23←"
KEY 2, "RUN:←"
KEY 3, "LIST "
KEY 4, "LOAD"
KEY 5, "DUMP←"
KEY 6, "PLACE0{crsr left}"
KEY 7, "ERROR←"
KEY 8, "COLOR11,12,0←"
KEY 9, "DISPLAY←"
KEY 10,"DIR"+CHR$(34)+"$"+CHR$(34)←"
KEY 11,"FIND"
KEY 12,"DISK"+CHR$(34)+"s:"
KEY 13,"{ctrl-n}{ctrl-h}"
KEY 14,"SCRLD1,8,3,"+CHR$(34)
KEY 15,"CSET 2:"
KEY 16,"DO NULL←"
```

Mit dem Kommando KEY OFF werden alle F-Tastenbelegungen ausgeschaltet und mit KEY ON wieder eingeschaltet. Ein Druck auf eine F-Taste erzeugt nach KEY OFF keine Bildschirmausgabe mehr. Der Befehl DISPLAY und die Funktion INKEY arbeiten jedoch weiter wie gewünscht. KEY 0 schaltet die F-Tastenbelegung ebenfalls ab (wie es wohl von David Simons auch ursprünglich gedacht war) und führt nicht mehr wie in Simons‘ Basic stattdessen den Befehl DISPLAY aus.

In Simons' Basic muss man sich in Ermangelung der ON-/OFF-Parameter mit `POKE $C646,10` zum Abschalten und mit `POKE $C646,0` zum Wiedereinschalten zufriedengeben.

Befehl:	KEYGET	
Syntax:	**KEYGET <stringvar>**	
Zweck:	Tastendrücke für die Weiterverarbeitung erfassen	Ein-/Ausgabe
Kürzel	kEgE	
Status:	Neuer TSB-Befehl (Anweisung), ab v2.20.523	

KEYGET ermöglicht es, Tastendrücke unmittelbar annehmen und weiterverarbeiten zu können. Bei seiner Anwendung wartet das Programm an dieser Stelle auf die Eingabe per Tastatur, die dann sofort der angegebenen Variablen zugewiesen wird. Alle Tasten, die ein in PETSCII kodierbares Zeichen erzeugen, werden von KEYGET akzeptiert, auch die zusammen mit der Shift-, Control- oder Commodore-Taste. KEYGET a$ ersetzt die Konstruktion `POKE 198,0: WAIT 198,1: GET a$`.

Anders als beim Befehl GETKEY in höheren Commodore-Basic-Versionen (ab 3.5) kann KEYGET nicht mit mehreren Variablen als Parameter umgehen. Sie werden zwar vom Interpreter akzeptiert, aber bis auf die erste nicht ausgewertet.

Hinweis: Da intern der Basic-V2-Befehl GET zur Anwendung kommt, treten auch dessen Nachteile in Kraft. Man sollte bei KEYGET ausschließlich Stringvariablen verwenden, obwohl auch numerische Variablen funktionieren. Wenn man dort jedoch Buchstaben statt Ziffern eingibt, führt das bei den meisten Eingaben zu einem unumgänglichen **SYNTAX ERROR**.

Befehl:	LEFT	
Syntax:	**LEFTB / LEFTW <zl>,<sp>,<bt>,<ho>**	
Zweck:	Linksscrollen eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	lE (für LEFTW, LEFTB hat keins)	
Status:	Erweitertes Simons' Basic (Anweisung)	

LEFTB bzw. **LEFTW** erlaubt es dem Programmierer, Bereiche des Textbildschirms (Parameter: ab Zeile **<zl>** und Spalte **<sp>** mit der Breite **<bt>** und der Höhe **<ho>**) inklusive der Farben spaltenweise **nach links** zu scrollen. In der äußerst rechten, freiwerdenden Spalte werden je nach Typ des Befehls Leerzeichen aufgefüllt (LEFTB, das „B" steht für „blank", was nur einen einzigen kompletten Scrollvorgang erlaubt) oder die links herausfallenden Zeichen wieder eingefügt (LEFTW, das „W" steht für „wrap", was mit dem gleichen Inhalt immer wieder durchgeführt werden kann).

Leider handelt es sich bei diesem Scrolling um ein zeichenweises Scrolling, das unter Umständen ruckelig wirkt. Pixelweises Scrolling („Smooth Scrolling") ist mit Simons'-Basic-Befehlen nicht möglich.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter die Fehlermeldung **BAD MODE ERROR**.

Beachten: Wenn LEFTB oder LEFTW direkt am rechten Bildschirmrand enden (in Spalte 39), kann es in Simons' Basic dort zu „Geisterzeichen" kommen, die störend sein können, allerdings keinen schwerwiegenden Folgefehler verursachen (behoben in TSB).

Beispiel:

```
100 CLS: COLOR 11,12,0
110 FOR X= 0 TO 39
120 Y=12*SIN(X/3)+12: PRINT AT(Y,X)"*"
130 NEXT
135 :
140 FOR Y=1 TO 3
150 FOR X=1 TO 40
160 LEFTW 0,0,20,25: REM DIVERGIEREN
170 RIGHTW 0,20,20,25
180 NEXT
190 FOR X=1 TO 40
210 RIGHTW 0,0,20,25: REM KONVERGIEREN
220 LEFTW 0,20,20,25
230 NEXT
240 NEXT
```

Das Beispiel erzeugt eine Sinuskurve, die sich bewegt. Abbruch ist per Tastendruck möglich. LEFTW und RIGHTW lassen hier die Kurve konvergieren bzw. divergieren.

Beispiel übernommen und angepasst aus dem Buch „Das Trainingsbuch zum Simons' Basic".

Befehl:	LIN	1
Syntax:	**LIN [[<z1>] - [<z2>]], <name>[, <dr>]**	
Zweck:	speichert **Teile** eines Programms	Ein-/Ausgabe
Kürzel	-	
Status:	Neuer TSB-Befehl (Kommando)	

Mit **LIN** hat der Programmierer die Möglichkeit, Teile eines im Speicher befindlichen Programms abzuspeichern. Zusammen mit MERGE kann man auf diese Weise auf Programmbibliotheken zurückgreifen und damit schneller und effektiver arbeiten.

Die verschiedenen Syntaxformen haben folgende Ergebnisse:

`LIN ,"name"`
speichert alles, genauso wie SAVE

`LIN -100,"name"`
speichert bis zur Zeile 100

`LIN 100-,"name"`
speichert ab Zeile 100

`LIN 100-200,"name"`
speichert von Zeile 100 bis einschließlich 200

`LIN 100-100,"name"`
speichert genau die Zeile 100

Die Laufwerksangabe kann in TSB (wie bei allen Befehlen, die mit den Laufwerken zu tun haben) weggelassen werden, siehe dazu auch den Befehl USE.

Mögliche Fehlermeldungen können sein: **BAD MODE ERROR**, wenn die zweite Zeilennummer kleiner ist als die erste, **TYPE MISMATCH ERROR**, wenn das Komma vor dem Dateinamen weggelassen wird, und alle Fehlermeldungen, die auch SAVE verursachen würde.

Beispiel:

LIN `2670-2890,"help"`

(speichert in der Datei „tsb demo“ auf der TSB-Diskette den Teil mit der TSB-Befehlsliste und der Shortcut-Prioritätsanzeige)

Befehl:	LIN	2
Syntax:	**a = LIN** **PRINT LIN**	
Zweck:	ermittelt die **Cursorzeile**	Bearbeiten des Textbildschirms
Kürzel	-	
Status:	Simons' Basic (Systemvariable)	

LIN ergänzt die BASIC-V2-Funktion POS(0) und ermittelt (als Systemvariable, nicht als Funktion), auf welcher Bildschirmzeile der Cursor sich gerade befindet.

Beispiel

```
10 CLS: CENTER "lin-demo": ZL=2: SP=2: W=0
20 REPEAT
30 PRINT AT(ZL,SP)"Zeile " ZL: ZL=ZL+1
40 IF LIN=23 THEN KEYGET x$: SP=22: ZL=2: W=W+1
50 UNTIL W=2
```

Das Beispiel gibt zwei Spalten auf dem Bildschirm aus, wobei am Ende jeder Spalte auf einen Tastendruck gewartet wird (Zeile 40).

Befehl:	LINE	
Syntax:	**LINE <x1>,<y1>,<x2>,<y2>,<fq>**	
Zweck:	Ziehen einer Linie zwischen zwei Grafikpunkten	Grafik-Befehle
Kürzel	lI	
Status:	Simons' Basic (Anweisung)	

Mit **LINE** verbindet man zwei Orte auf dem Grafikbildschirm durch eine gerade Linie. Der Anfangspunkt der Linie wird durch die beiden ersten Parameter **<x1>** und **<y1>** bestimmt, der Endpunkt durch die Parameter drei und vier (**<x2>** und **<y2>**). Die Farbe der Linie wird durch den letzten Parameter (**<fq>**) bestimmt. Zulässige Werte sind 0..319 für x (im Hires-Modus) bzw. 0..159 für x (im Multicolor-Modus). Für y sind in beiden Fällen Werte von 0 bis 199 erlaubt. Auch die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Bei Überschreitung der Bildschirmgrenzen bei den Parametern begrenzt der Interpreter die Werte und zeichnet weiter. Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

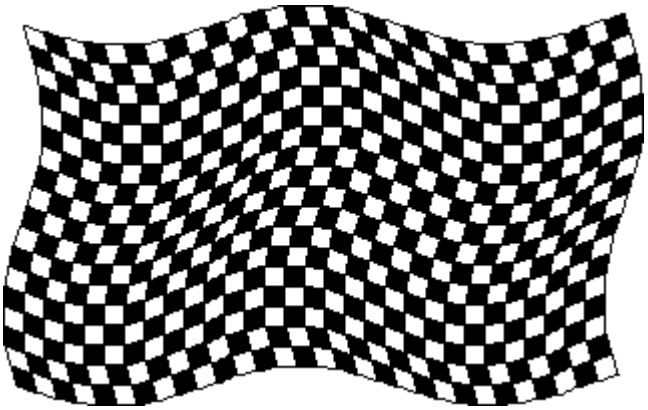


Bild 16 LINE macht scheinbar nicht nur gerade Linien: so…

Beispiel 1:

```
90 CLS: PRINT" 0  von 30";
100 DIM x(31,19), y(31,19)
110 FOR i=0 TO 30:FOR j=0 TO 18
120 x(i,j)=10+i*10+SIN(j/3)*10
125 y(i,j)=10+j*10+SIN(i/3)*10
130 PRINT AT(0,0)"";i;
140 NEXT:NEXT

145 HIRES 0,1
150 FOR i=1 TO 30: FOR j=0 TO 18
155 LINE x(i,j),y(i,j),x(i-1,j),y(i-1,j),1
160 NEXT:NEXT
170 FOR i=0 TO 30: FOR j=1 TO 18
175 LINE x(i,j),y(i,j),x(i,j-1),y(i,j-1),1
180 NEXT:NEXT

185 FOR i=0 TO 29:FOR j=0 TO 17
190 IF ((i+j) AND 1) THEN PAINT x(i,j)+4,y(i,j)+4,1
195 NEXT:NEXT
200 DO NULL
```

(erzeugt die abgebildete Grafik)

Beispiel 2:

```
100 HIRES 0,1
110 r=140:w={pi}/4
115 REPEAT
120 x=r*COS(w):y=r*SIN(w)
```

```
130 LINE 160+x,100-y,160-y,100-x,1
140 LINE 160-y,100-x,160-x,100+y,1
150 LINE 160-x,100+y,160+y,100+x,1
160 LINE 160+y,100+x,160+x,100-y,1
170 w=w+.1:r=r*.9
180 UNTIL w>2*{pi}
190 DO NULL
```

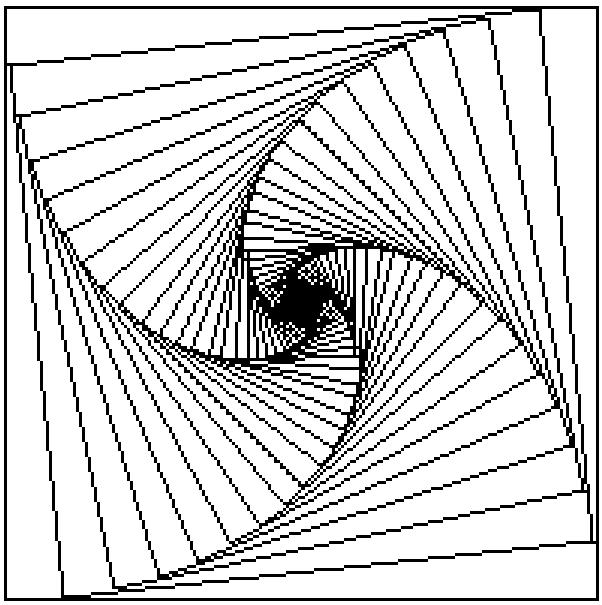


Bild 17 …oder so.

Befehl:	LOCAL	
Syntax:	LOCAL <v1> [, <v2>] ...	
Zweck:	Definieren lokal gültiger Variablen	Programmierhilfen
Kürzel	loC	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

LOCAL dient dazu, Variableninhalte zu definieren, deren Gültigkeit bei Bedarf wieder aufgehoben werden soll (die z.B. nur innerhalb einer Prozedur gültig sein sollen).

Normalerweise sind Variablen in der Programmiersprache BASIC programmweit gültig, einmal ein Wert zugewiesen, kann dieser Wert überall im Programm wieder abgerufen werden. Was wie ein Vorteil klingt, kann sehr schnell zum Nachteil werden: Bei großen Programmvorhaben darf man nämlich nicht den Überblick über den momentanen Inhalt einer Variablen verlieren, sonst reagiert das Programm irgendwann auf unvorhergesehene Weise und die Fehlersuche ist äußerst langwierig. Andere Programmiersprachen wie z.B. Pascal oder C sorgen schon von vornherein dafür, dass diese Hürde klein bleibt, denn dort gelten Variablen zunächst einmal ausschließlich lokal (nur innerhalb einer Prozedur oder Funktion).

Da auch Simons' Basic Prozedur-Befehle kennt (EXEC, PROC), liegt es nur nahe, dass auch eine Möglichkeit geschaffen wurde, nur dort gültige Variablen zu benutzen. Dies wird mit LOCAL zum Teil erreicht. Intern nämlich werden die lokalen Variablen in einer Liste geführt. Solange eine lokale Variable aktiviert ist, wird ihr globales Pendant unkenntlich gemacht, jedoch aber nicht gelöscht. Endet der lokale Bezug, tritt die alte, globale Variable wieder ein, mit ihren bis dahin gültigen Inhalten. Wird nun die lokale Variable erneut aktiviert, passiert das Gleiche wie oben beschrieben (der lokale Wert wird rekonstruiert).

Bedauerlich ist nun, dass die ausdrücklich lokalen Variablen nicht an eine bestimmte Prozedur gebunden sind, sondern überall sonst auch ihren nunmehrigen lokalen Wert führen! Somit ist die Lokalität von LOCAL höchst eingeschränkt. Außerdem beanspruchen LOCAL-Variablen zusätzlichen Platz im freien BASIC-Speicher.

Mit LOCAL können unter TSB bis zu 13 lokale Variablen deklariert werden. Darüberhinausgehende Zuweisungen werden (ohne einen Hinweis) nicht mehr angenommen und bleiben global.

Hinweis: Feldvariablen (Arrays) können nicht mit LOCAL bearbeitet werden. LOCAL darf bei gleichen Variablennamen nicht verschachtelt werden (durch Aufruf einer weiteren Prozedur, die den gleichen Variablennamen mit LOCAL verwendet).

Beispiel:

```
100 a=1: b%=10 : c$="test"
110 PRINT "dies ist ein test:"
120 PRINT "a="a; "b%="b%; "c$= "c$
130 prozedur
140 PRINT "zurueck im hauptprogramm:"
150 PRINT "a="a; "b%="b%; "c$= "c$
160 PRINT "test beendet."
170 END
999 :
1000 PROC prozedur
1010 LOCAL a, b%, c$
```

```
1020 a=100: b%=-10: c$="lokal"
1030 PRINT "und hier innerhalb der prozedur: "
1040 PRINT "a="a; "b%="b%; "c$= "c$
1050 GLOBAL
1060 END PROC
```

(Die PRINT-Ausgabe wird dreimal wiederholt, vor, während und nach der lokalen Phase)

Befehl:	LOOP	
Syntax:	**LOOP**	
Zweck:	Leitet eine Schleife ein	Struktur
Kürzel	lO	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

LOOP definiert den Anfang einer Schleife. Eine Schleife umschließt einen Teil eines Programms, der unter Umständen mehrfach durchlaufen wird.

Die Konstruktion LOOP .. END LOOP stellt letztlich die „Mutter aller Schleifen“ dar. Was hier im Schleifenkörper passiert, beschäftigt den Interpreter unentwegt. Nur mithilfe eines Befehls, der eine Abbruchbedingung überprüft, kann eine Schleife beendet werden. Der nötige und in Simons' Basic natürlich vorhandene (Ausstiegs-) Befehl lautet EXIT.

Mit diesen drei Befehlen (LOOP, END LOOP, EXIT) ist der Programmierer nun in der Lage, verschiedene Schleifentypen herzustellen:

- **Endlosschleifen** (die kann man nicht verlassen: es gibt darin kein EXIT),
- **kopfgesteuerte Schleifen** (die Ausstiegsbedingung wird gleich am Anfang abgefragt, so dass der Körper unter Umständen gar nicht erst durchlaufen wird),
- **fußgesteuerte Schleifen** (da wird erst am Ende des Schleifenkörpers die Ausstiegsbedingung geprüft, so dass er mindestens einmal durchlaufen wird)
- **Zählschleifen** (die werden verlassen, wenn ein Zählwert über- oder unterschritten ist).

Für die letzten beiden Schleifentypen gibt es eigene Befehlsfolgen (REPEAT-UNTIL für fußgesteuerte und FOR-NEXT für Zählschleifen), weshalb an dieser Stelle nur der Typ der kopfgesteuerten Schleifen betrachtet wird.

In anderen Hochsprachen (z.B. in Java und C) gibt es für diesen Typ das Befehlswort WHILE, in der Bedeutung „solange die Bedingung hinter WHILE zutrifft, durchlaufe die Schleife“. Der Ausstiegsbefehl ist dabei sozusagen gleich mit im WHILE eingebaut. WHILE stellen SB und TSB nicht zur Verfügung, es muss mit LOOP nachgebildet werden. Hier bei LOOP fragt der Interpreter leider anders, nämlich „solange die Bedingung hinter EXIT *nicht* zutrifft, durchlaufe die Schleife“, also logisch genau andersherum.

Die folgende Schleife (in Pseudocode):

```
A = 0
WHILE A<10 DO
  PRINT A;
  A = A+1
WEND
```

müsste in Simons' Basic so aussehen (mit NOT wird hier dem „Andersherum“ Rechnung getragen):

```
10 A = 0
20 LOOP
30   EXIT IF NOT(A<10)
40   PRINT A;
50   A = A+1
60 END LOOP
```

Ergebnisausgabe (bei beiden Programmen): 0 1 2 3 4 5 6 7 8 9. Wer sich nicht an negativ formulierte logische Terme (NOT) gewöhnen kann, müsste schreiben:

```
10 A = 0
20 LOOP
30   EXIT IF A>9
40   PRINT A;
50   A = A+1
60 END LOOP
```

Also mit einer Abbruchbedingung, die wirklich genau andersherum formuliert und nebenbei auch verständlicher und klarer ist.

Die TSB-Version des Befehls stellt die in Simons' Basic versprochene Stack-Tiefe von 20 auch tatsächlich zur Verfügung.

Beispiel für eine kopfgesteuerte Schleife:

```
10 A$ = ""
20 LOOP
30   EXIT IF PLACE(A$,"jnJN")
40   PRINT "Ja (j) oder Nein (n)? ";
50   FETCH "{crsr right}", 1, A$
60 END LOOP
```

Man kann alle Buchstaben eingeben, aber die Schleife wird erst verlassen, wenn „j“ oder „n“ getippt wurde (auch großgeschrieben). Wenn A$ bereits einen der Tastendrücke enthielte, würde die Abfrage gar nicht erst stattfinden (z.B. das Beispiel mit A$="j" in Zeile 10 probieren).

Wenn auf LOOP nicht irgendwo später ein END LOOP folgt, meldet TSB bei Aufruf von EXIT einen **LOOP ERROR**.

Befehl:	LOW COL	
Syntax:	**LOW COL <farbe1>, <farbe2>, [128 +] <farbe3>**	
Zweck:	Setzen abweichender Malfarben für Multicolor-grafik	Grafik-Befehle
Kürzel	loW	
Status:	Simons' Basic (Anweisung)	

Ohne **LOW COL** malt jeder schreibende Grafikbefehl ausschließlich in den Farben, die durch HIRES, COLOR und MULTI festgelegt wurden. Der C64 kann aber (in Multicolor) nicht nur mit vier aus 16 Farben für den ganzen Bildschirm arbeiten, sondern mit vier aus 16 Farben pro Kachel. Es dürfen also insgesamt alle 16 Farben auf dem Bildschirm vorkommen.

Da die eben genannten Befehle aber den ganzen Bildschirm betreffen, bietet TSB auch eine Anweisung, die eine „lokale" Malfarbenveränderung bewirkt. Die drei Farbangaben hinter LOW COL beziehen sich auf die gleichen Einstellungen wie bei MULTI, **<farbe1>** beeinflusst die Multicolor-Bitkombination %01, **<farbe2>** die Kombination %10 (beide in $C000) und schließlich **<farbe3>** diejenige, die aus %11 besteht (in $D800). Sie gelten, bis sie durch den Befehl HI COL wieder aufgehoben werden. Bei der Farbgebung muss man gut nachdenken, denn die MULTI- und LOW COL-Farben dürfen sich nicht „widersprechen" (siehe Beispiel).

In TSB (ab v2.20.324) kann man den Zugriff auf das Farb-RAM ($D800) verhindern, indem der dritte Parameter größer als 127 gewählt wird. Dann trägt LOW COL nichts ins Farb-RAM ein, wodurch z.B. Bildschirmmasken im Textmodus nicht zerstört werden.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beispiel:

```
10 COLOR 5,7: HIRES 10,1: MULTI 12,15,1
20 BLOCK 0,0,50,100,2: LOW COL 5,15,0
30 LINE 0,100,50,0,1
40 LINE 0,0,50,100,3: DO NULL
50 COLOR 11,12,0
```

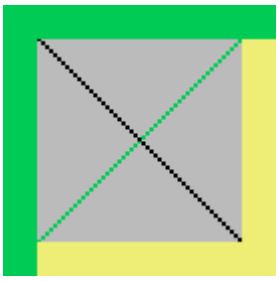


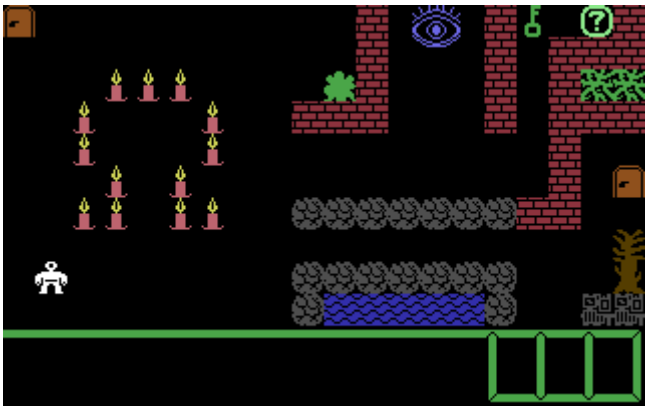
Bild 18 LOW COL

Kommentar:

Zeile 10: schaltet die Grafik ein, löscht sie und aktiviert den Multi-Modus
Zeile 20: malt ein hellgraues Quadrat in <farbe2> (der letzte Parameter von BLOCK bezieht sich auf <farbe2> bei MULTI: 15) auf gelbem Hintergrund (7 von COLOR), dann wird per LOW COL umgefärbt in grün, hellgrau und schwarz;
Zeile 30: die erste Linie erscheint in der LOW COL-Farbe 1 (Wert 5: grün)
Zeile 40: und die zweite in Farbe Schwarz (<farbe3> bei LOW COL, Wert 0), wie der letzte Parameter von LINE angibt.
Zeile 50: stellt die Textbildschirmfarben wieder her und wechselt (nach Programmende automatisch) in den Textmodus

Befehl:	MAP	
Syntax:	MAP <zl>,<sp>,<tile$>,<col-array%>	
Zweck:	Ausgabe von 2x2-Kacheln zur Bildung eines Spielfeldes aus Chars	Bearbeiten des Textbildschirms
Kürzel	mA	
Status:	Neuer TSB-Befehl (Anweisung)	

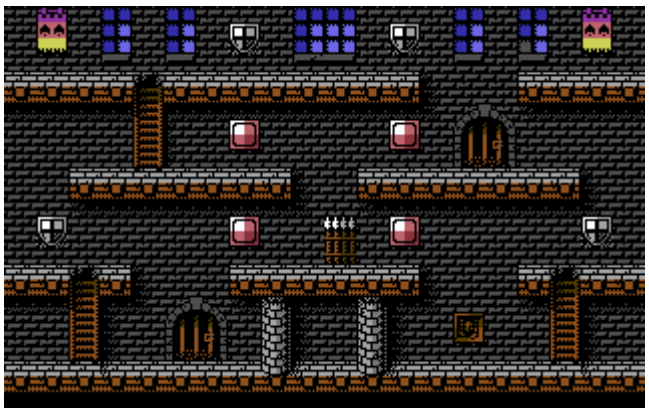
Genau wie X! gehört **MAP** zu den TSB-Befehlen, die nur temporär Teil des TSB-Befehlssatzes sind. **MAP** muss nachgeladen und ausdrücklich aktiviert werden, um es zu nutzen, sonst wird ein Aufruf des Befehls mit einem **NOT YET ACTIVE ERROR** beantwortet.



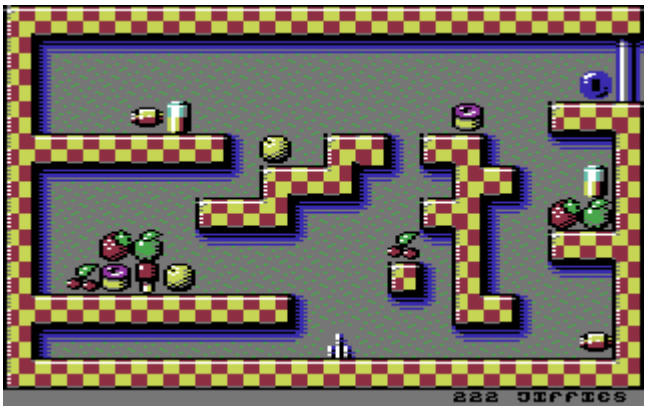
MAP wurde eingeführt im Zusammenhang mit dem in TSB geschriebenen Spiel „Ghost Valley“ von ***Christian Nennewitz***. Dort dient es dazu, das Spielfeld in großer Geschwindigkeit komplett aufzubauen (s. erstes Bild). Es setzt einen speziell für diesen Befehl konzipierten Zeichensatz voraus. In ihm müssen die dargestellten Gegenstände aus je vier Einzelzeichen zusammengesetzt sein, die im Speicher hintereinander liegen, z.B. die Eingangstür links oben im Bild. Jedes dieser Objekte hat eine Grundfarbe (die Tür ist orange, das Wasser ist blau usw.), die aber vom Programm aus mithilfe des Befehls FCOL noch ergänzt werden kann (ein schönes Beispiel dafür im dritten Bild, das aus einem kommerziellen Spiel („Danger Castle“) entnommen und mit **MAP** nachempfunden wurde).

Damit der **MAP**-Zeichensatz auch alphanumerische Zeichen für Beschriftungen und Interaktion zur Verfügung stellen kann, ist die Anzahl der 2x2-Objekte auf 48 begrenzt. Sie beginnen bei einem Byte-Offset von 512 Bytes (also ab dem Zeichen mit dem Bildschirmcode 64). Wenn der Zeichensatz z.B. an Position $E000 im Speicher liegt, beginnen die Objekte daher bei Position $E200.

Um einen zeitsparenden Weg für die Darstellung eines **MAP**-Objekts zu bieten, gehorcht der Aufruf eines Objekts in **MAP** einer Kodierung in einem String (**<tile$>**). Dabei erhält das erste Objekt den Code ‚#‘ (Wert: 35) und alle weiteren rücken im PETSCII-Code eine Stelle weiter, bis hin zum Wert (35 plus maximal 47 gleich) 82 (Code ‚r‘). Im zweiten Bild sieht man die Code-Strings für den vollständigen Bildaufbau für Level 5, Screen 2 im Spiel „Ghost Valley“, wie er im ersten Bild zu sehen ist. Die Tür hat den Code ‚&‘, das Wasser den Code ‚6‘ usw. Objekte, die in Bild 2 nicht zu sehen sind, dafür aber in Bild 1, sind Sprites. Zusätzlich: *Codes*, die im zweiten Bild zu sehen sind, nicht aber als Objekte in Bild 1 auftauchen, sind „geheime“, unsichtbare Objekte, die im Spiel bestimmte, zunächst überraschende Vorgänge auslösen (Codes ‚3‘ und ‚?‘).



Die Farbe der Objekte wird dem Befehl **MAP** über ein Integer-Array mitgeteilt (**<col-array%>**), dessen Inhalt der Anordnung bei der Objektkodierung folgt. Der erste Eintrag im Farb-Array ist dem Code ‚#' zugeordnet usw., der letzte dem Code ‚r'. Im Aufrufbefehl steht das Farbarray als letzter Parameter. In der Klammer für den Index steht immer die Indexnummer für den ersten Farbeintrag, normalerweise 0. Durch das Verwenden dieser Basisnummer könnte man für die Screens auch unterschiedliche Farben für gleiche Objekte vorsehen (nicht 0 als Basisindex angeben, sondern eine höhere Zahl, jenseits der Farben für den ersten Screen).



Selbstverständlich kann **MAP** auch Multicolor-Zeichensätze darstellen (viertes Bild, auch dieses aus einem kommerziellen Spiel, „Platman Worlds"). Wegen der besseren (und wahrscheinlich umfangreicheren) Einfärbungsmöglichkeiten könnte das Nachfärben mit FCOL einen spürbaren Moment dauern, das Ergebnis sollte jedoch diesen kleinen Mangel verschmerzbar machen.

Um den Multicolor-Textmodus zu aktivieren, braucht man zusätzlich die Befehle BCKGNDS und MULTI ON. Der folgende Code zeigt die Startsequenz für die Anzeige von Bild 4.

```
110 screen: multi on: bckgnds 128+12,0,1,x: .recolor
```

In der PROC SCREEN arbeitet der **MAP**-Befehl und gibt alle Objekte (hier noch nicht im Multimodus) aus, wie in der Screen-Map in Bild 5 zu sehen, was so schnell geht, dass der fehlende Multimodus nicht wahrgenommen wird. Danach wird Multi eingeschaltet (MULTI ON) mit den Farben 12 (mittelgrau) für den Hintergrund ($D021) und schwarz und weiß (BCKGNDS) für die für den ganzen Screen gültigen zwei Multifarben 0 (schwarz, wird bei den Objekträndern eingesetzt) und 1 (weiß, zeigt die hellen Lichter auf den Objekten). In der Prozedur .RECOLOR nimmt das Programm dann mit FCOL alle zusätzlichen Einfärbungen vor, es setzt z.B. das umfangreiche rot-gelbe Schachbrettmuster. Hierfür sind rund 160 Nachfärbungen erforderlich, die in der wirklich kurzen Zeit von etwa 220 Jiffies erfolgen (siehe Bild 4 unten rechts).

Die Beispielprogramme für **MAP** befinden sich auf der TSB-Disk („map.dmo" und „multimap.dmo").

Befehl:	MEM	
Syntax:	**MEM**	
Zweck:	Einschalten eines änderbaren Zeichensatzes	Zeichen ändern
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

MEM verlegt den ROM-Zeichensatz des C64 an die Speicherposition $E000, an der er mittels DESIGN editierbar ist. Diese Position liegt in der VIC-Speicherbank 3 ($C000 bis $FFFF), daher muss auch der Bildschirmspeicher verschoben werden. Er beginnt nach Anwendung von MEM an der Speicherstelle $CC00.

Da der DESIGN-Befehl zur Festlegung neuer Zeichen nicht dafür gedacht ist, einen kompletten Zeichensatz umzudefinieren (damit wäre der BASIC-Speicher bereits gefüllt und es bliebe kein Platz für das eigentliche Programm), sondern nur einzelne Zeichen, ist es sinnvoll, neue Zeichensätze mit geeigneten Editoren zu erstellen und dann innerhalb von Simons' Basic nach $E000 zu laden. Das untenstehende TSB-Beispielprogramm zeigt, wie das am besten erledigt werden sollte.

Der Anzeigemodus für geänderte Zeichensätze wird mit NRM wieder ausgeschaltet.

Beispiel:

Solche Zeichensätze sind kein Problem

```
  5 PRINT "{clear}{ctrl-n}Back at normal Basic..."
    AT(7,0)"OPTION10:LIST"
 10 IF DISPLAY<>$CC00 THEN MEM :
      I8=PEEK(186):
      DIM A$(2):
      LOAD "ZEICHENSATZ",0,0,$e800
 15 A$(0)="{neun Zeichen für INSERT}"
 20 F=11:
    PRINT "{clear}{ctrl-n}";:
    COLOR 0,1,F
 30 FILL 0,0,40,25,96,F
 40 PRINT " DESK  FILE  MISC  BASIC "
 50 INSERT A$(0),5,7,27,15,F
 80 PRINT AT(7,12)"Welcome to TSB!"
 85 PRINT AT(10,12)"If you notice any"
 90 PRINT AT(11,10)"'similarities',
they"
100 PRINT AT(12,8)"are absolutely intended."
110 PRINT AT(17,8)"(Please Press Any Key !)"
120 DO NULL
510 PRINT "{clear}Type NRM to return to normal Basic."
520 PRINT AT(4,0)"NRM" AT(0,1)""
```

Bild 19 Geänderter Zeichensatz

Kommentar: Die Zeile 5 sieht man eigentlich erst, wenn das Programm beendet wurde. Zeile 10 ist in TSB-Syntax. In Simons‘ Basic müsste dort IF PEEK($0288)<>204 THEN... stehen. Es wird getestet, ob der MEM-Befehl bereits aktiv ist (DISPLAY steht dann auf $CC00). Wenn nicht, wird MEM ausgeführt und der „zeichensatz“ vom aktuellen Laufwerk geladen. Der Rest erzeugt das, was im Beispielbild zu sehen ist. Das Programm geht im Original noch etwas weiter, es ist auf der TSB-Diskette unter dem Namen „fontdemo1.dmo“ zu finden.

Befehl:	MEMCLR	
Syntax:	**MEMCLR <adr>, <anz> [, <wert>]**	
Zweck:	Speicherbereiche mit einem vorgegebenen Wert überschreiben	Ein-/Ausgabe
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung), ab v2.30.910	

MEMCLR beschreibt alle **<anz>** Speicherstellen ab der Adresse **<adr>** mit dem vorgegebenen Wert 0 (null) oder optional mit einem beliebigen anderen Bytewert **<wert>**.

Die Angabe <anz> kann (theoretisch) beliebig groß sein, so dass der gesamte (RAM-) Speicher erreicht wird. Die Anzahl null wird ignoriert (es wird nichts in den Speicher geschrieben, es erscheint aber auch kein Fehler). Negative Werte erzeugen einen **ILLEGAL QUANTITY ERROR**.

Beispiele

Initialisieren des SID:

```
10000 PROC initsid
10010 MEMCLR SOUND, 29
10020 END PROC
```

Löschen eines Sprites mit der Blocknummer BL:

```
10000 PROC clearsprite
10010 MEMCLR bl*64, 64
10020 END PROC
```

Löschen des aktuellen Bildschirms:

```
10000 PROC initscreen
10010 MEMCLR DISPLAY, 1000, 32
10020 END PROC
```

Befehl:	MEMCONT	
Syntax:	**MEMCONT <type>**	
Zweck:	Festlegen der Art des RAM-Transfers	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMCONT legt mit **<type>** fest, wie sich die REU-Zählregister beim Transfer verhalten sollen:

- Ein Wert von **0** lässt die Zählregister für beide Seiten des Transfers automatisch nach jedem Byte weiterzählen.
- Der Wert **1** arretiert das REU-Zählregister, nur das C64-Register zählt weiter (gut zum schnellen Löschen von Speicherbereichen im C64).
- Der Wert **2** arretiert die C64-Adresse und lässt nur die REU-Seite weiterzählen.
- Mit dem Wert **3** werden beide Seiten des Transfers arretiert.

Befehl:	MEMDEF	
Syntax:	**MEMDEF <an> [, <c64ad> [, <reuad>, <reubk> [, <sw> [, <typ>]]]]**	
Zweck:	Sammelbefehl für alle für einen RAM-Transfer erforderlichen Einstellungen	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMDEF ist ein Sammelbefehl für alle Einstellungen der REU, die für einen Transfer benötigt werden. Zuerst legt man die Anzahl der zu übertragenden Bytes fest **(<an>**, s. MEMLEN), dann den Ort im C64 (**<c64ad>**, s. MEMOR). Darauf folgt die REU-Adresse mitsamt der Bank (**<reuad>, <reubk>**, s. MEMPOS). Dann legt man fest, ob diese Werte nach der Übertragung wieder rekonstruiert werden sollen (**<sw>**, s. MEMRESTORE) und schließlich steuert man mit **<typ>** die Art des Transfers (Arretierung eines Zählregisters oder nicht, s. MEMCONT).

Befehl:	MEMLEN	
Syntax:	**MEMLEN <anz>**	
Zweck:	Festlegen der Länge der zu übertragenden Daten	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMLEN sagt einer REU mit **<anz>**, wie viele Bytes transferiert werden sollen.

Beispiel: Unterroutine zum Laden eines Hires-Bildes aus Bank 3 in einer REU.

```
40100 PROC screenload
40110 MEMCONT 0
40120 MEMOR $c000: MEMPOS image,3
40130 MEMLEN 1000: MEMLOAD
40140 MEMOR $e000: MEMPOS image+$0400,3
40150 MEMLEN 8000: MEMLOAD
40160 END PROC
```

Kommentar: In IMAGE befindet sich die Zieladresse des Bildes innerhalb einer Bank (hier: Bank 3). Das zu übertragende Bild ist ein Hires-Bild, das aus 1000 Bytes Farben (Zeile 40130) und 8000 Bytes Bitmap besteht (Zeile 40150, ähnlich wie ein OCP-Art-Studio-Bild).

Befehl:	MEMLOAD	
Syntax:	**MEMLOAD**	
Zweck:	Laden von Speicherbereichen aus einer REU in den C64	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMLOAD holt einen Speicherbereich mit einer festgelegten Länge (s. MEMLEN) von einer vorher festgelegten Adresse in einer REU (s. MEMPOS) ab und speichert ihn an eine ebenso festgelegte Adresse des C64 (s. MEMOR). Der Befehl beachtet dabei die Einstellungen zur Transferart (s. MEMCONT) und zum Autoload der Adress- und Zählregister der REU-Steuereinheit (s. MEMRESTORE).

Mit dem folgenden kurzen Programm kann man nacheinander sieben (vorher dort abgelegte) Hires-Bilder aus der REU holen und anzeigen. Die Bilder befinden sich hier in Bank 3 der REU ab Adresse 1024. Wenn man sich auf die reinen Bilddaten beschränkt, passen in eine REU-Bank acht Hires-Bilder (mit Farbe wie hier: sieben). Nach jedem Bild wartet das Programm auf einen Tastendruck. Tippt man beim letzten Bild „x“, endet das Programm.

Beispiel: Programm "show images of reu" (auf der TSB-Demo-Diskette)

```
10 COLOR 11,12,0: PRINT "{clr}" AT(12,8) "End: 'x' on last image.":
DO NULL
20 CSET 2: REPEAT
30 image=$0400: screenload: DO NULL
40 image=$2800: screenload: DO NULL
41 image=$4c00: screenload: DO NULL
42 image=$7000: screenload: DO NULL
43 image=$9400: screenload: DO NULL
44 image=$b800: screenload: DO NULL
45 image=$dc00: screenload
50 REPEAT: GET x$: UNTIL x$>""
60 UNTIL x$="x"
70 CLS: PRINT AT(12,14) "End of Show!" AT(20,0) "";
999 END

40100 PROC screenload
40110 MEMCONT 0
40120 MEMOR $c000: MEMPOS image,3
40130 MEMLEN 1000: MEMLOAD
40140 MEMOR $e000: MEMPOS image+$0400,3
40150 MEMLEN 8000: MEMLOAD
40160 END PROC
```

Zur Verdeutlichung sind hier die von TSB erzeugten „Befehlsworte“ (nämlich: „screenload“) klein geschrieben, da sie keine eingebauten BASIC-Befehlsworte sind, sondern erst im BASIC-Programm definiert werden. Wie man Bilder in einer REU ablegt, ist beim Befehl MEMSAVE erklärt.

Befehl:	MEMOR	
Syntax:	**MEMOR <cadr> [, <radr>, <bnk>]**	
Zweck:	Festlegen der Adresse im C64 [und in der REU]	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMOR sagt einer REU, welche Adresse **<cadr>** im C64 vom Datentransfer betroffen sein soll (Start- bzw. Zieladresse des Transfers) [, optional auch die Adresse in der REU mit **<radr>** (Offset) und **<bnk>** (Bank)]. Diese Adressen können auch im Hex-Format angegeben werden.

Beispiel 1: Unterroutine zum Laden eines Hires-Bildes aus Bank 3 in einer REU.

```
40100 PROC screenload
40110 MEMCONT 0
40120 MEMOR $c000: MEMPOS image,3
40130 MEMLEN 1000: MEMLOAD
40140 MEMOR $e000: MEMPOS image+$0400,3
40150 MEMLEN 8000: MEMLOAD
40160 END PROC
```

Beispiel 2: Das gleiche Programm kürzer

```
40100 PROC screenload
40110 MEMCONT 0
40120 MEMOR $c000, image, 3
40130 MEMLEN 1000: MEMLOAD
40140 MEMOR $e000, image+$0400,3
40150 MEMLEN 8000: MEMLOAD
40160 END PROC
```

Befehl:	MEMPEEK	
Syntax:	**a = MEMPEEK(<adr>)** **PRINT MEMPEEK(<adr>)**	
Zweck:	Auslesen einer im C64-RAM gelegenen Adresse	Ein-/Ausgabe Programmierhilfen
Kürzel	-	
Status:	Neuer TSB-Befehl (numerische Funktion)	

Mit der numerischen Funktion MEMPEEK() kann eine RAM-Speicheradresse **<adr>** von 0 bis 65535 ausgelesen werden. Die Funktion liest den Wert der angegebenen Adresse als vorzeichenlose Zahl im Byte-Bereich (0 bis 255).

Liegt die angegebene Speicheradresse nicht im Word-Bereich, erscheint die Fehlermeldung **ILLEGAL QUANTITY ERROR**.

Hinweis: Möchte man den ROM-Zeichensatz des C64 auslesen (liegt unter dem I/O-Bereich), erreicht man diesen mit POKE $9519, $31 (bis v2.31.113: $924A) vor dem Einsatz von MEMPEEK (Patch rückgängig machen mit POKE $9519,$30).

Beachten: Alle Simons'-Basic-Funktionen mit 16-Bit-Argumenten arbeiten nicht ordnungsgemäß, wenn sie als ***zweiter*** Parameter in einem **POKE**-Befehl verwendet werden. Die Werte müssen vor ihrer Verwendung bei POKE einer Variablen zugewiesen werden. Der Grund liegt darin, dass die auch in SB benutzte GETADR-Routine des zugrunde liegenden BASIC-V2-Interpreters keine zwei 16-bittigen Argumente in einem Ausdruck verarbeiten kann. (Behoben in TSB.)

Dieser Hinweis gilt auch für den Befehl **WAIT**.

Befehl:	MEMPOS	
Syntax:	**MEMPOS <adr>, <bnk>**	
Zweck:	Festlegen der Adresse in der REU	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMPOS sagt einer REU, welche Adresse **<adr>** und welche Bank **<bnk>** in der REU vom Datentransfer betroffen sein sollen (Start- bzw. Zieladresse des Transfers).

Beispiel: Unterroutine zum Laden eines Hires-Bildes aus Bank 3 in einer REU.

```
40100 PROC screenload
40110 MEMCONT 0
40120 MEMOR $c000: MEMPOS image,3
40130 MEMLEN 1000: MEMLOAD
40140 MEMOR $e000: MEMPOS image+$0400,3
40150 MEMLEN 8000: MEMLOAD
40160 END PROC
```

Befehl:	MEMREAD	
Syntax:	**MEMREAD**	
Zweck:	Austauschen von Speicherbereichen aus einer REU mit dem C64	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMREAD tauscht einen Speicherbereich mit einer festgelegten Länge (s. MEMLEN) an einer vorher festgelegten Adresse in einer REU (s. MEMPOS) mit einem ebenso festgelegten Bereich im C64 (s. MEMOR) aus. Der Befehl beachtet dabei die Einstellungen zur Transferart (s. MEMCONT) und zum Autoload der Adress- und Zählregister der REU (s. MEMRESTORE).

Befehl:	MEMRESTORE	
Syntax:	**MEMRESTORE <swt> [+ 128]**	
Zweck:	Umschalter für das automatische Neuladen der Zähl- und Adressregister, Flag für DMA-Zugriff auf das Farb-RAM	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	memresT	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMRESTORE steuert mit **<swt>** das automatische Neuladen der Übertragungsregister nach dem REU-Transfer. Der Wert

- **0** bedeutet „nicht neu laden“ und belässt alle Register auf dem zuletzt bei der Übertragung erreichten Wert.
- **1** stellt alle Transferregister auf die vor dem Transfer eingestellten Werte zurück.

Außerdem sorgt ein gesetztes Bit 7 im Parameter dafür, dass die DMA-Zugriffe im IO-Bereich auf das Farb-RAM erfolgen und nicht auf das RAM darunter.

Beispiel: Programm zum Anzeigen des Directory-Inhalts der REU unter GoDot

```
80 CLS: CSET 1
85 d$="0123456789012345678": ls=LEN(d$): x$=CHR$(0)
90 stringadresse holen

100 MEMCONT 0: MEMRESTORE 1: MEMLEN LS
110 MEMPOS 0,0: MEMOR sa: MEMLOAD
120 PRINT "Directory of Drive 12: " MID$(d$,5,11): PRINT
130 nx=ASC(MID$(d$,2))+256*ASC(MID$(d$,3)): bk=ASC(MID$(d$,4))

210 FOR fl=1 TO 119
230 MEMPOS fl*19,0: MEMOR sa: MEMLOAD: p=fl:
240 IF ASC(d$)=$c0 THEN fl=119: GOTO 290
245 if place(x$,MID$(d$,5,14)) THEN MEMCLR sa,ls,$c0: GOTO 240
250 USE " ### ",fl;: PRINT MID$(d$,4) AT(LIN,28)"$" $$(ASC(d$) AND
15);
255 PRINT $$ASC(MID$(d$,3)); $$ASC(MID$(d$,2))
260 IF LIN=24 THEN KEYGET xx$: CLS
290 NEXT
300 PRINT: PRINT "Next entry at $" $$bk; $$nx; "," 120-p "entries
left"
999 END

1000 PROC stringadresse holen
1010 SYS $80be d$: p=(PEEK(780)+256*PEEK(782))+1
1020 sa=D!PEEK(p)
1040 END PROC
```

Befehl:	MEMSAVE	
Syntax:	**MEMSAVE**	
Zweck:	Ablegen von Speicherbereichen des C64 in einer REU	Ein-/Ausgabe Ansprechen von Peripheriegeräten
Kürzel	memsA	
Status:	Neuer TSB-Befehl (Anweisung)	

MEMSAVE legt einen vorher festgelegten Speicherbereich des C64 (s. MEMOR) von einer festgelegten Länge (s. MEMLEN) an einer ebenso festgelegten Adresse in einer REU (s. MEMPOS) ab. Der Befehl beachtet dabei die Einstellungen zur Transfer-Art (s. MEMCONT) und zum Autoload der Adress- und Zählregister der REU (s. MEMRESTORE).

Beispiel: TSB-Programm „write image“

```
5 f$="christmasday.art": image=$4c00

10 MOD 1,0:CSET 2:POKE $d020,15
15 OPEN 1,10,2,f$: FOR i=0 TO 8001: GET#1,x$: IF x$="" THEN
x$=CHR$(0)
20 IF (i AND 255)=0 THEN POKE $d020,f: f=f+1
25 x=ASC(x$): POKE $2ffe+i,x
30 POKE $dffe+i,x: NEXT
35 FOR i=0 TO 999: GET#1,x$: IF x$="" THEN x$=CHR$(0)
40 x=ASC(x$)
50 POKE $c000+i,x: NEXT
60 screensave: COLOR 11,12,0
99 DO NULL
999 END

40000 PROC screensave
40010 MEMCONT 0
40020 MEMOR $c000: MEMPOS image,3
40030 MEMLEN 1000: MEMSAVE
40040 MEMOR $3000: MEMPOS image+$0400,3
40050 MEMLEN 8000: MEMSAVE
40060 END PROC
```

Kommentar: In diesem Programm wird das OCP-Art-Studio-Bild „christmasday.art“ geladen (Zeile 5). Art-Studio-Bilder beginnen mit 8000 Bytes Bitmap, worauf 1000 Bytes Farben folgen. Diese Daten liest das Programm byteweise ein und speichert es einerseits im Grafikbereich ($E000, Zeile 30), damit man sieht, dass sich etwas tut (aus dem gleichen Grund schaltet sich auch die Randfarbe alle 256 eingelesene Bytes weiter, Zeile 20). Die Daten werden andererseits aber auch ins BASIC-RAM nach $3000 geschrieben (Zeile 25), da die REU im C64 nicht auf das RAM unter dem IO-Bereich zugreifen kann. Ist das Bild vollständig geladen, transferiert es das Programm von dort aus in die REU, hier an die Adresse $4C00 in Bank 3 (IMAGE in Zeile 5). Die Farben der Grafik liegen in TSB immer an der Adresse $C000 (Zeile 50). Wie man die Bilder wieder aus der REU herausbekommt, ist beim Befehl MEMLOAD beschrieben.

Befehl:	MERGE	
Syntax:	**MERGE <name>[, <dr> [,<sa>]]**	
Zweck:	lädt Teile eines Programms nach	Ein-/Ausgabe
Kürzel	mE	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

MERGE dient dazu, einem Programmierer das Entwickeln von Programmen mithilfe von Programm-Modulen zu ermöglichen.

MERGE lädt unter TSB ein BASIC-Programm zu einem bereits im Speicher vorhandenen **hinzu** und integriert es anhand der Zeilennummern. Das neue wird in das alte hineingemischt. Anschließend initialisiert der Interpreter den BASIC-Speicher neu, so dass alles zusammen lauffähig ist. Bei langen nachgeladenen Programmen dauert dieser Vorgang einige Sekunden.

MERGE lädt in Simons‘ Basic ein BASIC-Programm **hinter** ein bereits im Speicher vorhandenes (eigentlich also: APPEND). Das neue wird nicht in das alte hineingemischt.

Mögliche Fehlermeldungen sind alle diejenigen, die auch LOAD verursachen würde.

Beachten: Bei gleichen Zeilennummern im vorhandenen und im nachgeladenen Programm entsteht ein unangenehmer Wirrwarr. In so einem Fall sollte das Programm, wenn möglich, mit RENUMBER bearbeitet werden, damit Überschneidungen verschwinden. Da Referenzen auf gleiche Zeilennummern bei zwei verschiedenen Zeilen (mit vorher gleicher Zeilennummer) vorkommen können, muss man diese Stellen nach RENUMBER sorgfältig überprüfen.

Achtung! Mit einem JiffyDOS-ROM funktioniert MERGE nicht ohne Zusatzangaben! Dort wirkt es wie ein normales LOAD und löscht das womöglich in Arbeit befindliche Programm im Speicher! Zur Lösung des Problems muss man hier ausdrücklich die korrekte Sekundäradresse (null) angeben: **MERGE „name“,use,0**.

Befehl:	MMOB	
Syntax:	**MMOB** <n>, <sx>, <sy> [, <zx>, <zy>, [<gr>, [<sp>]]]	
Zweck:	Sprite steuern	Bearbeiten von Sprites
Kürzel	mM	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

Die Koordinaten für Sprites stimmen nicht mit denen für die Grafik überein. Die Fläche, auf der sich Sprites bewegen können, ist viel größer als die Grafik- bzw. Textfläche. Diese ist so auf dem Sprite-Bereich angeordnet, dass ein normal dargestelltes Sprite auf allen Seiten hinter dem Bildschirmrahmen verschwinden kann. Insgesamt überstreicht der Sprite-Bereich eine Fläche von 512×256 Pixeln. Der Grafik-/Textbereich darin (der 320×200 Pixel umfasst) beginnt bei der Koordinate x = 24 und y = 50 (für die linke obere Ecke des Sprites).

MMOB dient dazu, das Sprite mit der Nummer **<n>** von einem Startpunkt mit den Koordinaten **<sx>,<sy>** zu einem Zielpunkt mit den Koordinaten **<zx>,<zy>** zu bewegen. Der Parameter **<gr>** legt die Anzeigegröße des Sprites fest (und ändert damit die Basis-Einstellung bei MOB SET). Die Bewegungsgeschwindigkeit wird mit **<sp>** festgelegt, wobei größere Werte das Sprite langsamer fahren lassen.

Die verkürzte Schreibweise (ohne die Zielangabe <zx>/<zy>) **positioniert** einfach das angesprochene Sprite auf die gewählte Koordinate <sx>/<sy>.

Leider bewegen sich in Simons' Basic (und auch in TSB) Sprites nicht unabhängig voneinander (im Interrupt), sondern ausschließlich eins nach dem anderen (ein MMOB-Befehl steuert ein Sprite, der nächste Befehl das nächste Sprite usw.)

Die Einstellungen Größe <gr> und Geschwindigkeit <sp> sollten bei MOB SET eingetragen werden. Dort erfolgen dann gesammelt fast alle spritebezogenen Parameter.

Folgende Werte für <gr> sind zulässig:

Wert	Größe	Aussehen
0	24×21	normal
1	48×21	x-expandiert
2	24×42	y-expandiert
3	48×42	doppelt groß

Beachten: In Simons‘ Basic werden nur die Werte für die Sprite-Nummer, die Y-Koordinaten und für die Geschwindigkeit auf Plausibilität überprüft und erzeugen einen **ILLEGAL QUANTITY ERROR** bzw. **BAD MODE ERROR** bei Überschreitung der Höchstmarken. Wird die Sprite-Nummer zu groß gewählt, reagiert Simons' Basic mit unvorhersehbaren Anzeigen (Bildschirmflackern, Sprite-Flackern o.ä., alles behoben in TSB). Für die X-Koordinaten werden Werte bis 65535 akzeptiert (und auch abgearbeitet!) Werte größer als 3 bei der Anzeigegröße wirken alle wie der Wert 3.

Beispiel für DESIGN, MOB SET, MMOB und @:

```
1700 PROC SPRITE
1710   DESIGN 0, 15*64
1720   @BBBBBBBBBBBB............
1721   @BBBBBBBBBBBB............
1722   @BB........BB............
1723   @BB........BB............
1724   @BB........BB............
1725   @BB........BB............
1726   @BB........BB............
1727   @BB........BB............
1728   @BB........BB............
1729   @BB........BB............
1730   @BBBBBBBBBBBB............
1731   @BBBBBBBBBBBB............
1732   @........................
1733   @........................
1734   @........................
1735   @........................
1736   @........................
1737   @........................
1738   @........................
1739   @........................
1740   @........................
1750   MOB SET 1,15,1,0,0    : REM SPRITE DEFINIEREN
1760   S8=38: Z8=80          : REM SPRITE POSITION
1770   MMOB 1,S8,Z8          : REM SPRITE ANZEIGEN
1780 END PROC
```

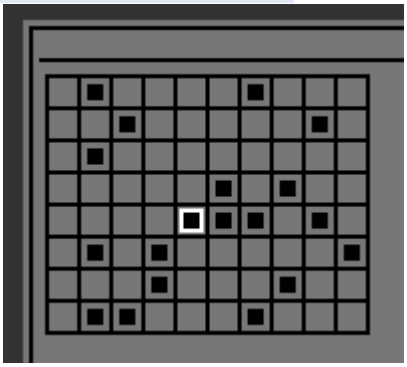


Bild 20 Ein Kreuzworträtsel-Gitter

(Definiert ein weißes Kästchen als Hires-Sprite mit einem zwei Pixel dicken Rand, in der Mitte bleiben 8x8 Pixel frei.)

Befehl:	MOBCOL	
Syntax:	**MOBCOL <n>,<f>**	
Zweck:	Sprites einfärben	Bearbeiten von Sprites
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MOBCOL setzt die individuelle Spritefarbe sowohl bei Hires-Sprites (Definitionsbuchstabe B bei Sprites) als auch bei Multicolor-Sprites (dann Definitionsbuchstabe C), die im VIC in den Registern $D027 bis $D02E verwaltet wird (siehe @ und DESIGN). Der Befehl entspricht dem Farbparameter des Befehls MOB SET, ohne aber dessen viele weiteren Parameter zu erfordern. **MOBCOL** überschreibt den Farbwert von MOB SET.

Hinweis: Das Token von **MOBCOL** war früher dem Befehl DISAPA zugeordnet. Seit TSB 2.40.215 gilt dies nicht mehr, der Befehl DISAPA wurde aus dem TSB-Befehlssatz entfernt.

Befehl:	MOB ON/OFF	
Syntax:	**MOB ON | OFF <n>**	
Zweck:	Sprite steuern	Bearbeiten von Sprites
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MOB ON|OFF schaltet das Sprite mit der Nummer **<n>** ein bzw. aus. Sprite-Nummern größer als 7 erzeugen einen **BAD MODE ERROR**.

Befehl:	MOB SET	
Syntax:	MOB SET <n>, <bl>, <f>, [128 +] <pr>, <type> [, <gr> [, <sp>]]	
Zweck:	Sprite einschalten und dessen Eigenschaften festlegen	Bearbeiten von Sprites
Kürzel	moB	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

Mit **MOB SET** schaltet man ein Sprite ein und erledigt den größten Teil der Sprite-Definitionen. Dieser Befehl legt zum ersten fest, welches der acht Sprites sich auf welchen mittels DESIGN gefüllten Speicherblock beziehen soll. Dafür sind die ersten beiden Parameter des Befehls zuständig (**<n>** und **<bl>**). Welche Blocknummern für <block> gewählt werden dürfen, kann man beim Befehl DESIGN 0 nachlesen. In TSB werden die Spritenummern auf Korrektheit überprüft (0..7). Zu große Sprite-Nummern erzeugen einen **BAD MODE ERROR**.

Die nächste Eigenschaft, die **MOB SET** festlegt, ist die Farbe **<f>** des Definitionsbuchstaben **B** bei hochauflösenden Sprites bzw. **C** bei Multicolor-Sprites (siehe @ und DESIGN). (Die anderen Definitionsbuchstaben färbt CMOB ein.) Mit diesem Parameter wird die individuelle Sprite-Farbe gesetzt (Bitmuster %10 in Multisprites), die ab $D027 (bis $D02E) in den VIC-II-Registern landet. *Diese Einstellung kann mit dem Befehl MOBCOL geändert werden*.

Der vierte Parameter (**<pr>**) steuert die Priorität eines Sprites gegenüber dem Hintergrund (die Priorität gegenüber den anderen Sprites regelt der C64 über die Sprite-Nummer). Ein Wert von 0 für <pr> bedeutet, dass das Sprite immer vor den Zeichen auf dem Bildschirm agiert. Eine 1 dagegen lässt ein Sprite hinter Bildschirmzeichen verschwinden. In TSB (nicht in Simons' Basic) kann man an dieser Stelle festlegen, ob **MOB SET** das Sprite einschalten soll oder nicht (wenn nicht, muss zu <pr> der **Wert 128** addiert werden). Dies hat den Vorteil, dass man einem Sprite bereits vor seinem Erscheinen die gewünschte Startposition mitgeben kann. Eingeschaltet wird es dann mit MOB ON/OFF.

Mit dem Parameter **<type>** wird endgültig festgelegt, ob ein Sprite hochauflösend sein soll (Wert **0** für <type>) oder aber in Multicolor-Darstellung erscheint (Wert **1**). Diese Angaben mussten zwar auch bei der Pixeldefinition (DESIGN und @) bereits gemacht werden, hatten dort aber nur beschreibenden Charakter. Erst mit **MOB SET** erhält ein Sprite seinen Darstellungsmodus tatsächlich zugewiesen.

Die beiden Parameter **<gr>** und **<sp>** sind in TSB aus Kompatibilitätsgründen optional, sie bestimmen die Größe des Sprites (<gr>) und seine Bewegungsgeschwindigkeit (<sp>). Erläuterungen hierzu bei MMOB (wo diese beiden Parameter in Simons‘ Basic stehen **müssen**, in TSB jedoch nicht).

Beachten: Die Zuordnung der Buchstaben B und C ändert sich je nach Darstellungsmodus des Sprites. Das ist ärgerlich und führt oft zu einer Fehlersuche, ist aber nicht zu ändern. Siehe dazu die Befehle @ und CMOB.

Beispiel 1:

```
1700 PROC msprite
1710   DESIGN 1, 14*64
1720   @...bbbbb....
1721   @.bbb...bbb..
1722   @bb.cc.cc.bb.
```

```
1723   @bb...c...bb.
1724   @bb...c...bb.
1725   @.bbbcccbbb..
1726   @...bbbbb....
1727   @b...ddd.....
1728   @bb..ddd.....
1729   @bbbbddddbb...
1730   @..bbdddbb...
1731   @....ddd.bb..
1732   @....ddd..bb.
1733   @....ddd...b.
1734   @....ddd.....
1735   @..ccc.ccc...
1736   @.ccc...ccc..
1737   @.cc.....cc..
1738   @ccc.....ccc.
1739   @ccc.....ccc.
1740   @bbbbbbbbbbbb
1750   COLOR 9,9: MOB SET 2,14,8,0,1: CMOB 7,6: s8=300: z8=205: MMOB
2,0,0,s8,z8,2,200
1760 END PROC
```

Gelb (7) ist die CMOB-Farbe Nr. 1, blau (6) die Nr. 2. Orange (8) kommt aus dem MOB-SET-Befehl (dort Parameter Nr. 3). Der Hintergrund ist braun (9 in COLOR).

(entnommen aus dem „Trainingsbuch zum Simon's Basic“)

Das an dieser Stelle früher abgedruckte Beispielprogramm „Reaktionstest“ ist wegen seines großen Umfangs aus dem Handbuch entfernt worden. Man kann es aber weiterhin im C64-Wiki (und auf der TSB-Disk) einsehen: Reaktionstest im C64-Wiki.

Befehl:	MOD	1
Syntax:	**MOD <ink>, <paper>**	
Zweck:	Umfärben der hochauflösenden Grafik, ohne sie zu löschen	Grafik-Befehle
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

MOD modifiziert die Farben der hochauflösenden Grafik. Der erste Parameter (**<ink>**) setzt die neue Schreibfarbe, der zweite (**<paper>**) steuert die Farbe des Hintergrunds. Die Farben werden für den ganzen Bildschirm geändert und dienen – wie beim Befehl HIRES – als Referenz bei allen Grafikbefehlen.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Befehl:	MOD	2
Syntax:	**a = MOD(<z>,<n>)** **PRINT MOD(<z>,<n>)**	
Zweck:	Rest einer Division ermitteln	Bearbeiten und Darstellen von Zahlen
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (numerische Funktion)	

Die Funktion **MOD** (math.: Modulo-Funktion) ist bei der Computersprache PASCAL ein Operator („mod“): Dieser dividiert den Zähler **<z>** durch den Nenner **<n>** und liefert **den Rest** der ganzzahligen Division, z.B. 9 mod 4 = 1. TSB bietet hier keinen zusätzlichen Operator an, sondern realisiert dies als numerische Funktion MOD. Sie entspricht dem recht aufwändigen Ausdruck
`a=INT(<z>)-INT(INT(<z>)/INT(<n>))*INT(<n>)`.

Beachten: Alle Simons'-Basic-Funktionen mit 16-Bit-Argumenten arbeiten nicht ordnungsgemäß, wenn sie als ***zweiter*** Parameter in einem **POKE**-Befehl verwendet werden. Die Werte müssen vor ihrer Verwendung bei POKE einer Variablen zugewiesen werden. Der Grund liegt darin, dass die auch in SB benutzte GETADR-Routine des zugrunde liegenden BASIC-V2-Interpreters keine zwei 16-bittigen Argumente in einem Ausdruck verarbeiten kann. (Behoben in TSB.)

Dieser Hinweis gilt auch für den Befehl **WAIT**.

Das Argument <z> darf dabei nur im 16-Bit-Bereich von 0 bis 65535 liegen, größere Werte meldet der Interpreter mit der Fehlermeldung **ILLEGAL QUANTITY ERROR**.

```
10 INPUT "POSITIVE ZAHL"; Z
20 A = DIV(Z,4): B = MOD(Z,4)
30 PRINT Z "DURCH 4 IST" A "MIT REST" B
```

Beispiel: Von einer Zahl wird berechnet, wie oft eine andere Zahl (hier: 4) in ihr enthalten ist und wie viel Rest bleibt.

Befehl:	MOVE	
Syntax:	**MOVE <zl>,<sp>,<bt>,<ho>,<zz>,<zs>**	
Zweck:	Umkopieren eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	mO	
Status:	Erweitertes Simons' Basic (Anweisung)	

MOVE dient dazu, einen Bereich des Textbildschirms mit der linken oberen Ecke **<zl>** (Zeile) und **<sp>** (Spalte) inklusive der Farben an eine andere Stelle des Textbildschirms (definiert durch **<zz>** und **<zs>**) zu kopieren.

Nützlich, um ohne großen Aufwand eine symmetrische Bildschirmgestaltung zu erzeugen.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter die Fehlermeldung **BAD MODE ERROR**.

Beachten: Die beiden letzten Parameter des Befehls (Zielzeile und Zielspalte) dürfen in Simons' Basic nur durch Konstanten angegeben werden. Der Interpreter wertet Variablen an dieser Stelle falsch aus, so dass unvorhersehbare Werte entstehen. Dieser Mangel ist unter TSB behoben.

Beispiel

```
100 CLS
110 COLOR 0,13: X=14: RV$=CHR$(18): SC$=CHR$(144): GR$=CHR$(152)
115 ST$=RV$+SC$+":"+GR$+"{11*space}"+SC$+":": RR=39-LEN(ST$)+4
120 FILL 0,0,40,1,160,5: MOVE 0,0,40,1,24,0
125 FILL 0,0,1,25,160,5: MOVE 0,0,1,25,0,39
130 REPEAT: X=X+RND(1)*2-1: GET X$
140 IF X<1 THEN X=1
150 IF X>RR THEN X=RR
160 DOWNB 1,1,38,23
170 PRINT AT(1,X) ST$
180 UNTIL X$>""
```

Erzeugt eine sich windende Straße, Abbruch per Tastendruck. MOVE vervollständigt hier den grünen Rahmen ums Bild.

Befehl:	MULTI	
Syntax:	**MULTI <farbe1>, <farbe2>, <farbe3>** **MULTI ON | OFF**	
Zweck:	Färben und Aktivieren der Multicolorgrafik	Grafik-Befehle Bearbeiten des Textbildschirms
Kürzel	mU	
Status:	Erweitertes Simons' Basic (Anweisung)	

MULTI legt drei der Farben des Multicolor-Modus des C64 fest und schaltet den Multicolor-Modus ein. Die vierte mögliche Farbe (die Hintergrundfarbe, Multicolor-Bitkombination %00, gesteuert durch Speicheradresse $D021) wird mit dem Befehl COLOR definiert. Der erste Parameter von MULTI (**<farbe1>**) steuert die Farbe der Bitkombination %01, **<farbe2>** entspricht der Farbe der Bitkombination %10 (beide im Speicherbereich ab $C000 zu finden) und **<farbe3>** färbt die Bitkombination %11 ein (im Farbspeicher ab $D800). Kehrt TSB in den Direktmodus zurück, wird MULTI automatisch deaktiviert.

Wichtig: Im Multicolor-Modus beziehen sich die Farbquellangaben aller Grafikbefehle auf die hier mit MULTI festgelegten Farben. Bei Angabe der **Farbquelle „0"** in einem Grafikbefehl wird die **Hintergrundfarbe** ausgewählt (siehe COLOR), die Angaben von „**1..3**" wählen von links nach rechts die **Farben hinter MULTI (bzw. LOW COL)** aus. Die Farbquellangabe „**4**" **invertiert** den angesteuerten Pixel, d.h., dass die Hintergrundfarbe (%00) nun in <farbe3> ausgegeben wird, <farbe1> (%01) erscheint nun als <farbe2>, <farbe2> (%10) als <farbe1> und <farbe3> (%11) als Hintergrundfarbe.

Beachten: MULTI verändert die Farben auch für den Textbildschirm (die Schreibfarbe erscheint in <farbe3> und die Hintergrundfarbe wurde geändert), so dass nach der Rückkehr aus dem Grafikmodus möglicherweise Texte unleserlich geworden sind. Der Befehl FCOL ist hier nützlich, um den Textscreen wieder lesbar einzufärben.

Mit **MULTI ON** wird in TSB der Multicolor-Modus aktiviert, ohne dabei Farben zu setzen. Ab sofort schreibt der Grafikmodus doppelt breite Punkte, wodurch auch deren Zählung beeinflusst wird (0 bis maximal 159). **MULTI OFF** schaltet den Multicolor-Modus aus (auch beim Multicolor-Textmodus).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beispiel:

```
10 COLOR 5,7: HIRES 10,1: BLOCK 1,0,100,100,1: DO NULL
30 MULTI 12,15,1: DO NULL

CSET 2: DO NULL
```

Das Beispiel schaltet die Grafik an, zeigt per BLOCK ein hellrotes Quadrat auf weißem Hintergrund (Farben 10,1 bei HIRES) und wartet auf einen Tastendruck, dann wird in Zeile 30 mit MULTI umgefärbt in mittelgrau (12, Streifen ganz links), hellgrau (15, Streifen rechts) und weiß (1) auf gelbem Hintergrund (7 bei COLOR); der letzte, interaktive CSET-Befehl zeigt die Grafik erneut, allerdings mit Farben wie nach HIRES 12,15, da zuvor MULTI ausgeführt wurde, was HIRES überschreibt.

Befehl:	MUSIC	
Syntax:	**MUSIC <d>, <str>**	
Zweck:	Festlegen und Steuern der Musik	Sound
Kürzel	muS	
Status:	Simons‘ Basic (Anweisung)	

MUSIC definiert ein Musikstück und seinen Ablauf. Es ist die Voraussetzung für PLAY. Simons'-Basic-Musikstücke sind (ohne den Einsatz von Tricks) nur einstimmig.

<d>: Der Parameter **<d>** legt die Basiseinheit der Dauer der kürzesten Note im Stück fest (da Notenwerte wie Achtel, Viertel usw. eine Angabe relativ zu den anderen Noten im jeweiligen Stück sind, ist die Zuordnung zu einer Notenwert-Bezeichnung an dieser Stelle beliebig). Alle anderen Notenlängen sind Vielfache von diesem Wert (s.u.) Der Parameter **<d>** bestimmt also das **Tempo** des Musikstücks.

<str>: Der Parameter **<str>** definiert das „Notenblatt“. Hier werden die SID-Stimmen, die zu spielenden Noten, ihre individuelle Dauer und ein paar weitere Eigenschaften des Musikstücks festgelegt. Es gibt dazu drei verschiedene Arten von Steuerzeichen:

1. Steuerung: Noten

Zuerst die Steuerzeichen für **Noten**. Sie entsprechen im Wesentlichen den üblichen Notennamen (wobei das h durch b dargestellt wird; h ist eine ausschließlich deutsche Notenbezeichnung).

```
c - d - e - f - g - a - b
```

Die **Halbtöne** (schwarze Tasten auf dem Klavier) erreicht man zusammen mit der Shift- (Tonerhöhung) oder der Commodore-Taste (Tonerniedrigung, steht nicht im Handbuch).

```
Shift   :  C - D - F - G - A
C=      :  c - d - e - f - g - a - b
```

Diese Tastenkombinationen stehen also für die Notenbezeichnungen

bei Tonerhöhung (mit Shift): **cis - dis - fis - gis - ais**
und
bei Tonerniedrigung (mit C=): **ces - des - es - fes - ges - as - bb**

Hinter dem Notennamen folgt als Ziffer die zu spielende **Oktave** (0..7). Akzeptiert werden auch Oktavangaben über 7 hinaus (8, 9, a, b, ...) Diese Werte sind allerdings nicht ausprobiert.

Eine **Pause** wird als Buchstabe **z** eingegeben. Auf z folgt ebenfalls eine (beliebige) Oktavangabe oder einfach noch ein z.

2. Steuerung: Notenlänge

Abschließend setzt man die zweite Art von Steuerzeichen, die Notenlänge. Sie werden durch die den Funktionstasten zugeordneten reversen PETSCII-Zeichen eingegeben. Dabei bedeuten:

F1	Achtel	bzw.:	Sechzehntel	(Anzeige:)	rvs **E**
F3	Viertel		Achtel		rvs **F**
F5	punktierte Viertel		punktierte Achtel		rvs **G**
F7	Halbe		Viertel		rvs **H**

F2	Halbe+Achtel	Viertel+Sechzehntel	rvs **I**
F4	punktierte Halbe	punktierte Viertel	rvs **J**
F6	doppelt punktierte Halbe	doppelt punktierte Viertel	rvs **K**
F8	Ganze	Halbe	rvs **L**

Der Notenwert der Taste F1 legt dabei das Bezugssystem fest und benennt den kürzesten Notenwert im zu bearbeitenden Musikstück. Abweichungen von diesen Notenlängen sind nicht vorgesehen, so dass Triolen und musikalische Verzierungen nicht ohne Tricks eingegeben werden können. Man kann aber **summieren**: F3F1 = punktierte Viertel (gleich F5), wenn die Bezugsnote die Achtel ist.

3. Steuerung: Ablauf

Weitere steuernde Angaben werden im MUSIC-String grundsätzlich mit dem {clr}-Code eingeleitet (die Taste <Shift Clr/Home>). Im Handbuch werden nur zwei dieser Steueranweisungen aufgeführt, es gibt jedoch zusätzlich drei undokumentierte Codes. Die Codes im Einzelnen:

- **{clr}+1** : Stimme selektieren (1, 2 oder 3). **Muss am Anfang des MUSIC-Strings stehen.**
- **{clr}+g** : aktuelle Stimme ausschalten. Sollte am Ende eines MUSIC-Strings stehen.
- **{clr}+t** : Sync-Bit für die aktuelle Stimme (s. WAVE) invertieren. (Zweck s.u.)
- **{clr}+r** : Wiederholen des ganzen MUSIC-Strings, Endlosspielen. Nur sinnvoll am String-Ende.
- **{clr}+c** : Löschen der WAVE-Definition der aktuellen Stimme. Nur sinnvoll am String-Ende.

Erläuterungen:

{clr}+t macht nur Sinn, wenn man zwei Stimmen kombiniert hat (was erst einmal mit Simons'-Basic-Befehlen gar nicht so einfach möglich ist, bei TSB ginge ein paralleles SOUND), und die zweite die erste beeinflussen soll. Immerhin kann man Stimme 3 unhörbar schalten (s. VOL) und sie einen Ton abspielen lassen, der nicht aufhört (bei entsprechenden ENVELOPE-Einstellungen) und diesen auf Stimme 1 wirken lassen. Das geht aber auch nur, wenn Stimme 1 mit Dreiecksschwingung arbeitet (s. WAVE).

{clr}+r lässt ein Musikstück immer wieder ablaufen. Das macht nur Sinn, wenn man mit PLAY 2 arbeitet, da sonst das ganze Programm blockiert wäre. Für diese Funktion war der Befehl PLAY 0 eigentlich implementiert worden, damit eine Endlosschleife abgebrochen werden kann. Funktionieren tut {clr}+r aber nur bei PLAY 1 (und nicht bei PLAY 2). Damit ist {clr}+r unbrauchbar.

{clr}+c schaltet eine Stimme komplett ab, da dann für sie keine Definitionswerte mehr existieren (s. WAVE). Nur sinnvoll bei Stücken mit Ringmodulation durch Stimme 3, wobei die Modulation irgendwo im Stück ausgeschaltet werden soll.

Fazit

Ein MUSIC-String sollte also prinzipiell diese Form aufweisen (ohne Leerzeichen eingeben!):

"`{clr}1 c4{F3} {clr}g`" : PLAY spielt dann mit **Stimme 1** den **Ton C** als **Viertelnote** in **Oktave 4** und beendet ihn nach Ablauf der ENVELOPE-Parameter.

Wie bei allen Strings ist auch der MUSIC-String auf eine maximale Länge von 255 Zeichen begrenzt. Lange Musikstücke kann man mit Simons' Basic also nicht so einfach erstellen. Insgesamt sind die Möglichkeiten der Musikprogrammierung unter Simons' Basic eher enttäuschend und man sollte Musiken weiterhin mit den entsprechenden Spezialprogrammen herstellen.

Wird bei MUSIC kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**.

Anwendungsbeispiel beim Befehl MOB SET (Beispiel 2).

Befehl:	NO ERROR	
Syntax:	**NO ERROR**	
Zweck:	Fehlerkontrolle abschalten	Abfangen von Laufzeitfehlern
Kürzel	nO	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

NO ERROR beendet die durch ON ERROR aktivierte Fehlerkontrolle des laufenden Programms.

Da Simons' Basic (im Gegensatz zu TSB) nicht automatisch beim Rückfall in den Direktmodus die Fehlerkontrolle beendet, muss eine Fehlerinfo-Routine im Programm auf jeden Fall NO ERROR enthalten, am besten gleich als erstes, damit keine ungewollten Selbstaufrufe erfolgen können. Bei Programmabbrüchen (z.B. durch die STOP-Taste) bei aktiver Fehlerkontrolle muss man sofort NO ERROR eingeben, damit nicht jede andere Eingabe in der Fehlerinfo-Routine landet.

Die Fehlerkontrolle insgesamt wurde in TSB so überarbeitet, dass sie nunmehr voll funktionsfähig ist, siehe ON ERROR.

Beispiel:

```
10 ON ERROR: GOTO 10000

15 PRIN "{clr/home}"
20 PRINT "ok"

10000 NO ERROR
10010 PRINT "in zeile " ERRLN "trat fehler nr." ERRN "auf."
10020 STOP
```

Bei einem Programmlauffehler springt der Interpreter in die BASIC-Zeile 10000.

Hinweis: Will man Fehlermeldungen komplett verhindern, schreibt man: ON ERROR : RESUME.

Befehl:	NRM	1
Syntax:	**a = NRM(<string>)** **PRINT NRM(<string>)**	
Zweck:	Umwandeln einer Hex- oder Binärzahl in einen Dezimalwert	Bearbeiten und Darstellen von Zahlen
Kürzel	nR	
Status:	Neuer TSB-Befehl (Stringfunktion)	

Mithilfe dieser Stringfunktion lassen sich zwei- oder vierstellige Hexzahlen bzw. acht- oder 16-stellige Binärzahlen in Dezimalzahlen umwandeln (Länge bei **<string>** ist zwingend vorgeschrieben). Die Strings dürfen dabei nur die jeweils zulässigen Ziffern enthalten (**0..9** und **a..f** bei **Hexzahlen** und **0** und **1** bei **Binärzahlen**). Innerhalb der Strings dürfen die Zeichen „**%**“ für „binär“ oder „**$**“ für „hex“ vorangestellt werden. Ein String darf auch als Konstante - z.B. `PRINT NRM("$c000")` - an die Funktion übergeben werden.

Kommt im String ein Zeichen vor, das nicht zur jeweiligen Zahlenbasis passt, führt dies zur Fehlermeldung **HEX CHAR ERROR** bzw. **BIN CHAR ERROR**.

Beispiel:

```
10 a$="123": a$=RIGHT$("0000"+a$,4)
20 PRINT "Hex $"a$;: b$=STR$(NRM(a$)): PRINT " ist dezimal "+b$
```

Anzeige: Hex $0123 ist dezimal 291

(die Hexzahl $123 wird zunächst vierstellig gemacht und danach mit STR$ in einen Dezimal-String verwandelt, um zu zeigen, dass sogar das funktioniert – wäre aber nicht erforderlich, für die Anzeige reicht NRM(a$) aus)

Beachten: Alle Simons'-Basic-Funktionen mit 16-Bit-Argumenten arbeiten nicht ordnungsgemäß, wenn sie als ***zweiter*** Parameter in einem **POKE**-Befehl verwendet werden. Die Werte müssen vor ihrer Verwendung bei POKE einer Variablen zugewiesen werden. Der Grund liegt darin, dass die auch in SB benutzte GETADR-Routine des zugrunde liegenden BASIC-V2-Interpreters keine zwei 16-bittigen Argumente in einem Ausdruck verarbeiten kann. (Behoben in TSB.)

Dieser Hinweis gilt auch für den Basic-V2-Befehl **WAIT**.

Befehl:	NRM	2
Syntax:	**NRM**	
Zweck:	Normalen Textmodus einschalten (Groß-Klein-schrift) und Rekonstruieren von PLACE und RENUMBER nach INST oder MEM	Bearbeiten des Textbildschirms Grafik-Befehle
Kürzel	nR	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

NRM stellt den VIC auf seine TSB-Vorgabewerte zurück: Der Grafikmodus wird abgeschaltet, die Groß-Klein-Schrift aktiviert, der Extended-Color-Modus wird abgeschaltet und ein Bildschirm mit eventuell verändertem Zeichensatz (siehe MEM) wird auf den Normalzeichensatz zurückgesetzt (Bildschirmspeicher an Adresse 1024).

Gleichzeitig rekonstruiert NRM nach Anwendung der Befehle INST oder MEM die dadurch deaktivierten Befehle RENUMBER und PLACE. Dazu muss auf der Diskette im aktuellen Laufwerk die Datei „**tsb.mem**“ zu finden sein. Ist sie nicht da, gibt es einen **FILE NOT FOUND ERROR** und die beiden Befehle bleiben deaktiviert.

Wenn TSB mit einer angeschlossenen REU gestartet wird, holt **NRM** sich die Datei „tsb.mem“ aus der REU.

Befehl:	OFF	
Syntax:	**OFF**	
Zweck:	Blinken von Zeichen beenden	Bearbeiten des Textbildschirms
Kürzel	oF	
Status:	Erweitertes Simons' Basic (Anweisung)	

Das mit FLASH hervorgerufene Blinken von Zeichen wird mit **OFF** (auch: **FLASH OFF**) beendet.

Beachten bei Simons' Basic: Die Ausführung des Befehls findet im Interrupt statt, das Programm läuft in dieser Zeit weiter. Der Programmierer hat keinen Einfluss mehr auf FLASH. Auch das Ende des Befehlslaufs ist nicht synchronisiert, die zuletzt angezeigte Blinkphase hängt daher vom Moment des Ausführens des Befehls OFF ab. Wenn ein Programm vorzeitig abbricht (<RUN/STOP>-Taste gedrückt oder Laufzeitfehler), muss das Blinken von Hand mit OFF ausgeschaltet werden, da der Interpreter es auch im Direktmodus weiterlaufen lässt.

In TSB wurden diese Synchronisationsprobleme behoben. Möchte man dort an einem laufenden FLASH-Screen zusätzliche Änderungen vornehmen, so synchronisiert man die Ausgabe mit `WAIT $C5C7,$80,$80` (wartet auf die **nicht** invertierte Phase, danach PRINT u.ä.)

Beispiel:

```
100 COLOR 7,2,1:
    BFLASH 1,7,6:
    PRINT "{clear}";
110 FOR x=0 TO 39
120   y=x/2
130   PRINT AT(y,x)"*" AT(y,39-x)"*"
140   PRINT AT(0,x)"*" AT(20,x)"*"
150   PRINT AT(y,0)"*" AT(y,39)"*"
160 NEXT
170 FLASH 1,15:
    COLOR ,10
180 PRINT AT(1,0)"";:
    CENTER "Demo von":
    PRINT
190 CENTER "Flash und BFlash"
200 PRINT AT(20,0)"";:
    PAUSE 6
210 OFF :
    PAUSE 6:
    BFLASH OFF
    COLOR 11,12,0:
```

(schreibt einen Kasten mit weißen Sternen auf rotem Grund und lässt alles blinken, Zeile 210 könnte auch **FLASH OFF** lauten)

Befehl:	OLD	
Syntax:	**OLD**	
Zweck:	Wiederherstellung eines im Speicher mit NEW gelöschten Programms	Programmierhilfen
Kürzel	oL	
Status:	Erweitertes Simons' Basic (Kommando)	

Mit **OLD** stellt der Interpreter ein im Speicher (z.B. durch NEW) gelöschtes Programm wieder her. Er nimmt alle nötigen Einstellungen vor, damit es sofort mit RUN gestartet werden kann.

Befehl:	ON ERROR	
Syntax:	**ON ERROR: <befehle>**	
Zweck:	Lauffehler während des Programmlaufs abfangen	Abfangen von Laufzeitfehlern
Kürzel	on E	
Status:	Erweitertes Simons' Basic (Anweisung)	

Mit **ON ERROR** wird die Kontrolle, ob Fehler zu einer Fehlermeldung und einem damit verbundenen Programmabbruch führen, vom Interpreter ans Programm selbst übergeben. Da Fehler jederzeit im Programm auftauchen können, sollte ON ERROR gleich am Anfang des Programms stehen, am besten in der ersten Zeile. In der gleichen Zeile folgt auf den ON-ERROR-Befehl ein Befehl, der festlegt, was bei einem Programmfehler geschehen soll. Da auf einer einzigen Zeile nicht Platz genug für viel Information ist, setzt man hier üblicherweise einen Sprung an eine andere Stelle im Programm ein, z.B. `ON ERROR: GOTO 10000`.

Dort kann man dem User mitteilen, welcher Fehler vorlag (OUT) und in welcher BASIC-Zeile er aufgetreten ist (ERRLN), damit die Ursache des Fehlers umrissen wird. Man sollte dem Benutzer Zeit zum Lesen geben und ihn dann entscheiden lassen, ob das Programm fortgesetzt (siehe RESUME) oder abgebrochen werden soll, um den Fehler zu beheben.

Bei Abbruch verlässt TSB den Fehlerabfang-Modus automatisch.

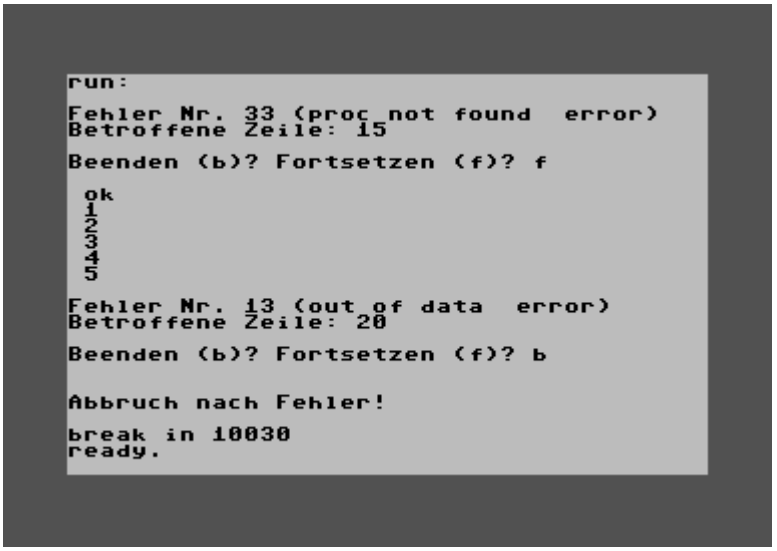


Bild 21 Eine ON-ERROR-Sitzung

Beispiel:

```
10 ON ERROR: GOTO 10000

15 PRIN "{clr/home}": PRINT "ok"
20 READ b
25 PRINT b: GOTO 20
30 DATA 1,2,3,4,5

10000 PRINT: PRINT "Fehler Nr." ERRN
"(";: OUT: PRINT ")"
10010 PRINT "Betroffene Zeile:" ERRLN:
PRINT
10020 PRINT "Beenden (b)? Fortsetzen
(f)? ";: FETCH "fb",1,x$: PRINT
10030 IF x$ = "b" THEN PRINT: PRINT
"Abbruch nach Fehler!": NO ERROR: STOP
10040 RESUME
```

(bei einem Programmlauffehler springt der Interpreter in die BASIC-Zeile 10000, gibt dort Informationen zum Fehler aus und stellt Handlungsalternativen zur Auswahl)

Tipp: Fehler einfach ignorieren und mit der gelieferten Fehlernummer fortfahren geht (z.B. beim Einlesen von DATA-Zeilen unbestimmter Menge) mit `ON ERROR:RESUME`.

Hinweis: Sollen auch Fehler beim Einlesen von Daten von der Floppy abgefangen werden, dann muss in der Fehlerabfangroutine (hier bei Zeile 10000) die Datei mit `SYS $FFCC: CLOSE <fn>` (<fn> steht für „*Filenummer*“) geschlossen werden, sonst gerät TSB in eine Endlosschleife.

Befehl:	ON KEY	
Syntax:	**ON KEY <string>: <befehle>**	
Zweck:	Schaltet die Tastatur-Kontrolle ein	Ein-/Ausgabe Struktur
Kürzel	oN oder on K	
Status:	Erweitertes Simons' Basic (Anweisung)	

TSB kann beliebige Tastendrücke unabhängig vom laufenden Programm abfangen. Wenn eine solche Taste gedrückt wird, verzweigt der Interpreter in eine dafür vorzusehende Tastatur-Kontrollroutine innerhalb des Programms, in der dieser Tastendruck behandelt wird. Alle von der Tastatur aus erreichbaren Zeichen sind zulässig (und auch nur sinnvoll). Die zuletzt gedrückte Taste wird in der Speicherstelle $C5EC festgehalten und kann dort abgefragt werden.

ON KEY dient hauptsächlich dazu, Hotkeys in menügeführten Programmen zu ermöglichen, z.B. könnte man auf Druck der Tastenkombination <C=b> aus jeder Programmsituation heraus in ein Menü „Bearbeiten“ wechseln.

Beispiel

```
10 ON KEY "abc": CALL auswertung

... langes Programm ...

10000 PROC auswertung
10010 DISABLE
10020 tt$ = CHR$(PEEK($c5ec))
10020 IF tt$ = "a" THEN PRINT AT(0,35)"ah! "
10030 IF tt$ = "b" THEN PRINT AT(0,35)"beh!"
10040 IF tt$ = "c" THEN PRINT AT(0,35)"zeh!"
10050 RESUME
```

Wenn der Benutzer "a", "b" oder "c" drückt, springt der Interpreter in die BASIC-Zeile 10000.

Eine Zeile 10 könnte auch wie folgt aufgebaut sein:

```
10 ON KEY "abc": DISABLE: auswertung: RESUME
...
```

Bei RUN würde also der Interpreter nach ON KEY den Rest der Zeile übergehen und dann während des Programmlaufs auf das Drücken einer der Tasten "a", "b" oder "c" warten. Wird eine davon gedrückt, arbeitet der Interpreter die Befehle ab DISABLE ab. RESUME führt dann zurück an die Stelle im Programm, an der der Tastendruck erfolgte (deshalb der ursprüngliche Aufruf mit CALL).

Beachten: In Simons' Basic muss nach dem Parameter <string> zwingend ein Komma gesetzt werden.

Befehl:	OPTION	
Syntax:	**OPTION ON | OFF** **OPTION <n>**	
Zweck:	Kennzeichnen von Simons'-Basic-Befehlen für die LIST-Ausgabe	Programmierhilfen
Kürzel	oP	
Status:	Erweitertes Simons‘ Basic (Kommando)	

Mit **OPTION** kann man die LIST-Ausgabe so beeinflussen, dass alle Simons'-Basic- und TSB-Schlüsselwörter dabei invertiert dargestellt und somit hervorgehoben werden. Eingeschaltet wird OPTION in Simons‘ Basic mit einem Wert von 10 für **<n>**, in TSB mit **OPTION ON**, abschalten kann man es mit **OPTION OFF** (in SB mit dem Wert 0).

Werden in TSB Zahlenwerte eingegeben, werden diese so interpretiert, dass beim LISTen auf jede Zeile, die ein END PROC enthält, eine Trennlinie aus Minuszeichen erscheint. Der Zahlenwert gibt an, wie viele Minuszeichen ausgegeben werden. Ein Wert von 0 (null) schaltet diese Funktion wieder ab.

Beachten: Die Trennlinie wird auch nach Zeilen, in denen END PROC Teil eines Bedingungsgefüges ist, angezeigt. Andere Zeichen als das Minuszeichen erreicht man mit **POKE $C9C1,<PETSCII-Code>**.

Befehl:	OUT	
Syntax:	OUT [<n>] [;]	
Zweck:	Fehlertext ausgeben	Abfangen von Laufzeitfehlern Programmierhilfen
Kürzel	oU	
Status:	Erweitertes Simons' Basic (Anweisung)	

OUT gibt den Text des letzten vom Interpreter gemeldeten Fehlers aus und setzt danach den Meldungspuffer zurück auf „kein Fehler“. Unter TSB werden alle Fehler abgefangen, auch diejenigen der BASIC-Erweiterung selbst (Nummern und Texte siehe unten).

Mit OUT **<n>** kann man gezielt einen bestimmten Fehlertext zur Anzeige bringen. Der Wert 0 und alle Werte über 45 werden dabei ohne besonderen Hinweis ignoriert (dem Wert 31 ist keine Fehlermeldung zugeordnet, hier erscheint nur „ERROR“). Werte über 255 führen zu einem **ILLEGAL QUANTITY ERROR**.

Fehlt hinter OUT das Semikolon, wechselt der Cursor nach der Ausgabe des Fehlertextes auf die folgende Bildschirmzeile. Mit Semikolon bleibt der Cursor – wie auch bei PRINT – hinter der Fehlermeldung stehen.

TSB-Fehlernummern und ihre Texte:

```
32 BAD MODE
33 NO PROC
34 INSERT TOO LARGE
35 STRING TOO LARGE
36 BIN CHAR
37 HEX CHAR
38 END PROC W/O EXEC
39 END LOOP W/O LOOP
40 LOOP
41 UNTIL W/O REPEAT
42 NOT YET ACTIVE
43 TOO FEW LINES
44 BAD CHAR
45 NO ARRAY
```

Beispiel: siehe ON ERROR.

Befehl:	PAGE	
Syntax:	**PAGE <n>** **PAGE ON | OFF**	
Zweck:	Anzahl Zeilen für die LIST-Ausgabe festlegen	Programmierhilfen
Kürzel	pA	
Status:	Erweitertes Simons' Basic (Anweisung)	

PAGE macht die LIST-Ausgabe angenehmer und brauchbarer. Der Wert hinter dem Befehl legt fest, wie viele Bildschirmzeilen bei LIST gefüllt werden sollen (eine mehr als **<n>** angibt). Bei Erreichen dieser Anzahl unterbricht der Interpreter das LISTen und wartet auf die Eingabe von <RETURN>, um fortzufahren, oder von <STOP>, um die Ausgabe zu beenden.

Der Wert 0 für <n> und Werte von über 23 schalten PAGE wieder aus. PAGE OFF schaltet das seitenweise LISTen ebenfalls ab. Wenn PAGE vorher schon einmal definiert wurde, tritt die vorherige Einstellung nach PAGE ON wieder in Kraft.

Die Shift-Taste hält die Ausgabe der Zeichen in TSB an jeder beliebigen Stelle an (anders als bei LIST in BASIC V2).

Wenn man die <C=>-Taste während des LISTens betätigt, wird die PAGE-Einstellung so lange ausgesetzt. Der Interpreter listet den Basic-Text dann **nicht mehr** seitenweise, sondern wie üblich fortlaufend.

Befehl:	PAINT	
Syntax:	**PAINT <x>,<y>,<fq>**	
Zweck:	Ausfüllen einer umschlossenen Fläche	Grafik-Befehle
Kürzel	paI	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

Mit **PAINT** werden umschlossene Fläche in der Farbe der angegebenen Farbquelle **<fq>** (s. dazu HI-RES), ausgehend vom Startpunkt **<x>, <y>**, ausgefüllt. Zulässige Werte für <x> sind **0..319** (im Hires-Modus) bzw**. 0..159** (im Multicolor-Modus). Für <y> sind in beiden Fällen Werte von **0 bis 199** erlaubt. Der Punkt 0,0 ist in der linken oberen Ecke.

PAINT kann durch Drücken einer der Tasten <CTRL>, <C=> oder <Shift> abgebrochen werden. Der interne Algorithmus arbeitet anders als in Simons' Basic (dort läuft der Füll-„Stift“ abwärts und aufwärts und vergisst schon mal etwas, in TSB geht es nur abwärts, ohne Vergesslichkeit). Der Invertiermodus (<fq> = 2 oder 4) arbeitet in TSB einwandfrei.

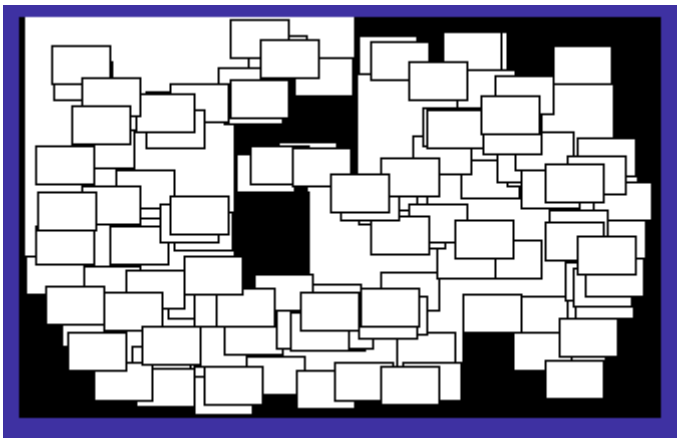


Bild 22 Flächen füllen mit PAINT

PAINT tut nichts, wenn der angewählte Startpunkt bereits die gewünschte Farbe hat. Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Ablauf im nebenstehenden Bild: Nachdem eine Reihe von „Flugblättern“ ausgeworfen wurde, tritt PAINT in Aktion und schwärzt den Hintergrund.

Beispiel:

```
100 HIRES 0,1
105 REPEAT: GET x$
110 x=290*RND(1): y=180*RND(1)
120 BLOCK x,y,x+29,y+19,0
130 REC x,y,29,19,1
140 UNTIL x$>"": PAINT 0,0,1
150 DO NULL
```

(erzeugt die abgebildete Grafik)

Befehl:	PAUSE	
Syntax:	**PAUSE [<string>,] <sek>**	
Zweck:	Programm für eine festgelegte Zeit anhalten	Struktur
Kürzel	paU	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

PAUSE ist dazu gedacht, einen Bildschirm für eine bestimmte Anzahl von Sekunden (**<sek>**) auf jeden Fall sichtbar zu halten. Der Befehl hält ein laufendes Programm also zeitkontrolliert an. Um eine einfache Möglichkeit zu haben, dem Benutzer ein solches Anhalten zu signalisieren, bietet PAUSE den optionalen **<string>**-Parameter an. Der String wird an der aktuellen Cursorposition ausgegeben und hängt keinen Zeilenvorschub an. Nach der Ausgabe bleibt der Cursor, so wie auch beim PRINT-Befehl, hinter dem ausgegebenen String stehen.

Eine Angabe von 0 als Wartezeit ist erlaubt, der Interpreter setzt dann seine Arbeit unverzüglich fort.

Die Wartezeit kann durch Betätigen einer beliebigen Taste abgebrochen werden. Diese Taste kann über PEEK(197) nach dem PAUSE-Befehl abgefragt werden (liefert Tastatur-Scancodes) und somit für Programm-Steuerzwecke dienen.

Beachten: Werte über 255 für <sek> werden akzeptiert, aber auf Byteformat reduziert, der Interpreter führt intern ein (<sek> AND 255) durch. Lässt man den <string> weg, muss der Wert von <sek> als Zahlenkonstante eingegeben werden, sonst reagiert (T)SB mit einem **TYPE MISMATCH ERROR**. Um diesen zu vermeiden und trotzdem Variablen für die Zeitangabe möglich zu machen, gibt man **PAUSE 0+var** ein (genau wie beim Befehl USE) oder **PAUSE “”,var**.

Beispiel 1:

```
90 a$="Bitte warten": b$="Danke       "
100 PRINT AT(23,23)"";: PAUSE a$,5
110 PAUSE 2: PRINT AT(23,23)b$
120 CLS
```

(in der Zeile 100 wird in alter Simons'-Basic-Syntax ein Hinweis ausgegeben, nach einer Wartezeit von 5 Sekunden mit einem weiteren PAUSE-Kommando kommentiert und in Zeile 120 der Bildschirm gelöscht)

Tipp: Um Zeitspannen zu bemessen, die kürzer sind als eine Sekunde, verwendet man die Speicherstelle **$C516**. Sie wird von (T)SB 60-mal in der Sekunde erhöht, zählt also 60tel Sekunden (Jiffys). Will man z.B. zur Aufmerksamkeitssteigerung den Bildschirmrahmen kurz flackern lassen, geht man so vor:

Beispiel 2:

```
1000 BFLASH 5,2,11
1010 POKE $C516,0: REPEAT: UNTIL PEEK($C516)>15
1020 BFLASH OFF
```

Lässt den Rahmen für etwa eine Viertelsekunde mit schneller Frequenz (5 Jiffys) zwischen Rot (2) und Dunkelgrau (11) blinken.

Befehl:	PENX	
Syntax:	**PENX**	
Zweck:	ermittelt die X-Position eines Lightpens	Ansprechen von Peripheriegeräten
Kürzel	pE	
Status:	Erweitertes Simons‘ Basic (Systemvariable)	

PENX fragt die X-Position eines am Controlport 1 angeschlossenen Lightpens ab (Speicherstelle $D013 im VIC). Die Spitze des Stifts ist dabei sensibel für das Aufblitzen des Elektronenstrahls des verwendeten Bildschirms/Monitors bei einem Durchgang an dieser Stelle, daher funktioniert so ein Gerät nicht mehr an modernen Flachbildschirmen.

Die Ordinatenangabe muss vor einer Verwendung in Programmen zunächst kalibriert werden, da nicht die Entfernung vom linken Rand des Grafik-/Textfensters auf dem Bildschirm, sondern von Rand des Bildschirms selbst vom Interpreter ausgegeben wird. Außerdem ist die X-Auflösung etwa doppelt so hoch wie die normale Pixelauflösung des C64, daher muss man die Rückgabe von PENX so auf Grafikkoordinaten umrechnen:

```
x = PENX*2-40
```

Wobei der Wert 40 der Kalibrierungswert ist und bei den Lightpens verschiedener Hersteller eben anders sein kann. Auch kann sich dieser Wert für PENY von dem für PENX unterscheiden.

Beachten: Der korrespondierende Befehl PENY liefert nur dann sinnvolle Werte, wenn vorher PENX ausgeführt wurde. (Die eigentliche Portabfrage findet bei PENX statt.)

Beispiel:

```
160 HIRES 6,7: xa=0: ya=0
170 PROC .plot
180 x=PENX*2-40
190 y=PENY-40
200 IF x<0 OR x>319 OR y<0 OR y>199 THEN CALL .plot
210 LINE xa,ya,x,y,1
220 xa=x: ya=y
230 GET a$: IF a$="←" THEN HIRES 6,7
240 CALL .plot
```

Beispiel entnommen aus „Trainingsbuch zum Simons' Basic“, Seite 368. PROC wird hier als Sprunglabel verwendet, nicht als Einleitung einer Prozedur.

Befehl:	PENY	
Syntax:	**PENY**	
Zweck:	ermittelt die Y-Position eines Lightpens	Ansprechen von Peripheriegeräten
Kürzel	-	
Status:	Erweitertes Simons' Basic (Systemvariable)	

PENY fragt die Y-Position eines am Controlport 1 angeschlossenen Lightpens ab (Speicherstelle $D014 im VIC). Die Spitze des Stifts ist dabei sensibel für das Aufblitzen des Elektronenstrahls des verwendeten Bildschirms/Monitors bei einem Durchgang an dieser Stelle, daher funktioniert so ein Gerät nicht mehr an modernen Flachbildschirmen.

Die Ordinatenangabe muss vor einer Verwendung in Programmen zunächst kalibriert werden, da nicht die Entfernung vom oberen Rand des Grafik-/Textfensters auf dem Bildschirm, sondern von Rand des Bildschirms selbst vom Interpreter ausgegeben wird. Laut „Trainingsbuch zum Simons' Basic“ soll man daher die Rückgabe von PENY so auf Grafikkoordinaten umrechnen:

```
y = PENY-40
```

Beachten: PENY liefert nur dann sinnvolle Werte, wenn vorher der korrespondierende Befehl PENX ausgeführt wurde. (Die eigentliche Portabfrage findet bei PENX statt.)

Beispiel:

```
160 HIRES 6,7: xa=0: ya=0
170 PROC .plot
180 x=PENX*2-40
190 y=PENY-40
200 IF x<0 OR x>319 OR y<0 OR y>199 THEN CALL .plot
210 LINE xa,ya,x,y,1
220 xa=x: ya=y
230 GET a$: IF a$="←" THEN HIRES 6,7
240 CALL .plot
```

Beispiel entnommen aus „Trainingsbuch zum Simons' Basic“, Seite 368. PROC wird hier als Sprunglabel verwendet, nicht als Einleitung einer Prozedur.

Befehl:	PLACE	1
Syntax:	PLACE 0 | <arrname>	
Zweck:	Anzeige von Array-Inhalten	Programmierhilfen
Kürzel	plA	
Status:	Neuer TSB-Befehl (Anweisung)	

PLACE gibt den Inhalt von Array-Variablen auf dem Bildschirm aus.

Bei der Eingabe von PLACE **0** werden nacheinander (in der Reihenfolge, wie sie im Programm definiert wurden) alle vorhandenen Arrays auf dem Bildschirm angezeigt. Zuerst erscheint der Name des Arrays mit seiner Dimensionierung, danach zeilenweise der Inhalt der „Zellen“, wobei die einzelnen Dimensionen von links nach rechts inkrementiert werden (zuerst die ganz links, zuletzt die ganz rechts). Damit liegen die Inhalte der einzelnen „Ebenen“ in der Ausgabe nahe beieinander.

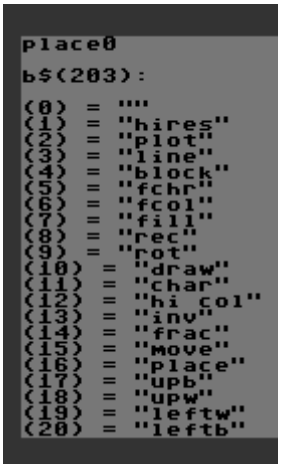


Bild 23 Die Ausgabe von PLACE

Will man nur ein ganz bestimmtes Array anzeigen lassen, gibt man PLACE **<arrname>** ein. Wird bei dieser Syntax das gesuchte Array nicht gefunden (Tippfehler oder falscher Name), meldet der Interpreter einen **NO ARRAY ERROR**.

Die Anzeige kann mit der <RETURN>-Taste angehalten werden und mit <STOP> kommt man in den Direktmodus zurück. Die angezeigten Array-Inhalte sind nicht direkt editierbar, da vor jeder Ausgabezeile der Arrayname fehlt. (Wer es trotzdem versucht, erhält einen **SYNTAX ERROR**, da der Name des Arrays am Zeilenanfang fehlt.)

Zusammen mit DUMP, ON ERROR (und den zugehörigen Systemvariablen ERRLN und ERRN), FIND, TRACE und der F-Tastenbelegung mit KEY hält der Programmierer hiermit nützliche Werkzeuge zur Programmentwicklung und zum Debugging in Händen. Erst TSB macht ihre problemlose Nutzung überhaupt möglich.

Befehl:	PLACE	2
Syntax:	a = PLACE(<string1>,<string2>) PRINT PLACE(<string1>,<string2>)	
Zweck:	Suchen eines Strings in einem anderen	Stringfunktionen
Kürzel	plA	
Status:	Simons' Basic (numerische Funktion)	

PLACE sucht Zeichenkette **<string1>** in Zeichenkette **<string2>** und gibt dessen Position innerhalb von <string2> an. Die Zählung beginnt dabei mit 1 (1 = erstes Zeichen). Lautet das Ergebnis 0, ist <string1> nicht in <string2> enthalten. Leerstrings – egal, an welcher Position – führen zum Ergebnis 0.

Im Beispiel wird nach der Position des Leerzeichens gesucht:

Beispiel:

```
10 a$=" ": b$="vorname nachname"
20 c=PLACE(a$,b$)
30 PRINT "der gesuchte string ";
40 IF c THEN PRINT "befindet sich an position" c:
   ELSE PRINT "ist nicht vorhanden"
```

Ergebnis: "der gesuchte string befindet sich an position 8".

Beachten: Im Direktmodus und bei Verwendung von literalen Strings (direkt verwendeter Text in Anführungszeichen) liefert diese Funktion falsche Ergebnisse, deshalb gibt es unter TSB im Direktmodus die Fehlermeldung **ILLEGAL DIRECT ERROR**.

Hinweis: Man kann den Direktmodus mit POKE $9D,0 abschalten und so die Funktion auch ohne Programm ausführen.

Befehl:	PLAY	
Syntax:	PLAY <n> PLAY ON | OFF	
Zweck:	Abspielen der Musik	Sound
Kürzel	-	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

PLAY startet das Abspielen des mit MUSIC definierten Musikstücks (dabei wird das Gate-Bit eingeschaltet, s. WAVE).

Der Parameter **<n>** steuert dabei die Art und Weise, wie der Interpreter abspielt.

- **n = 1** : Simons' Basic wartet, bis das Musikstück zu Ende ist, und fährt erst danach im Programm fort.
- **n = 2** : Simons' Basic setzt den Programmlauf ohne Pause fort (Interrupt-Betrieb). Das Stück spielt parallel zum Programmlauf.
- **n = 0** : Simons' Basic beendet den Interrupt-Betrieb. Entgegen den Angaben im Handbuch läuft die Musik weiter, jedoch nicht mehr parallel zum Programm, sondern wie nach PLAY 1. Außerdem beendet PLAY 0 ein ewig laufendes Musikstück (siehe MUSIC).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**. Der Interpreter akzeptiert Werte für <n> zwischen 0 und 255, wobei alle Werte außer 1 und 2 wie 0 behandelt werden.

In TSB kann man zum Beenden des Musikstückes auch **PLAY OFF** eingeben. **PLAY ON** schaltet sie wieder ein, wobei n = 2 aktiviert wird.

Beispiel:

```
100 VOL 15
110 ENVELOPE 1,1,8,10,10
120 WAVE 1, 00100000
130 MUSIC 10, "{clear}1c2{f2}"
140 PLAY 1
150 VOL 0
```

(VOL 0 wird erst durchgeführt, wenn die Musik beendet ist)

Hinweis: Wurde im MUSIC-String keine Stimme ausgewählt, stürzt Simons‘ Basic ab (behoben in TSB). Man kann dies mit einem **POKE $C57F,$D4** vor dem Auslösen des PLAY-Befehls verhindern.

Befehl:	PLOT	
Syntax:	**PLOT <x>, <y>, <fq>**	
Zweck:	Setzen eines Grafik-Punktes	Grafik-Befehle
Kürzel	pL	
Status:	Erweitertes Simons' Basic (Anweisung)	

PLOT malt einen einzelnen Grafik-Punkt (Pixel) an die Position **<x>,<y>** in der Farbe, die durch **<fq>** bestimmt wird.

Je nachdem, welcher Grafikmodus aktiviert ist (siehe HIRES und MULTI), hat man in x-Richtung 320 ansteuerbare Positionen (Hires-Modus) oder 160 (Multicolor-Modus). In y-Richtung beträgt die Auflösung immer 200 Pixel. Auch die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Das in Simons' Basics an dieser Stelle enthaltene Osterei (`PLOT 383,0,127`) gibt es unter TSB nicht mehr.

Beispiel 1:

```
100 HIRES 1,0
110 FOR X=.5 TO 159.5: FOR Y=.5 TO 99.5
120 R=SQR(X*X+Y*Y): H=.7: IF R<90 THEN
H=.7+.7*(90-R)/90
130 IF SGN(SIN(X/4)*SIN(Y/4))=>0 THEN
H=1-H
140 IF H<RND(1) THEN 160
150 PLOT 159.5+X, 99.5-Y,1:PLOT 159.5-
X, 99.5+Y,1: GOTO 170
160 PLOT 159.5+X, 99.5+Y,1: PLOT 159.5-
X, 99.5-Y,1
170 NEXT: NEXT
180 DO NULL
```

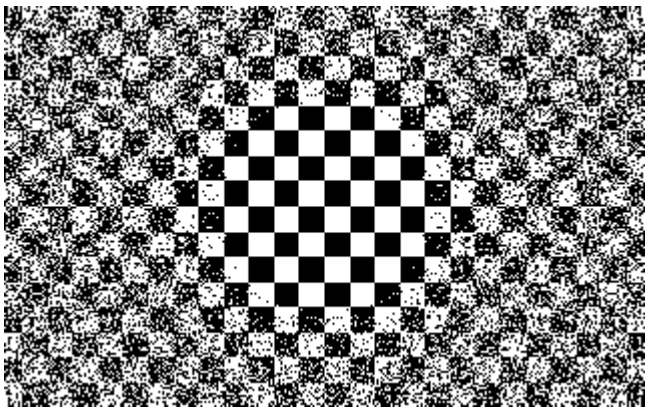


Bild 24 Mit PLOT kann man schöne Sachen machen

Beispiel 2:

```
100 HIRES 1,0
110 FOR X=0 TO 159: FOR Y=0 TO 99
120 R=X*X+Y*Y
130 IF (R/150 AND 1) THEN 160
140 PLOT 160+X,100+Y,1:PLOT 160-
X,100+Y,1
150 PLOT 160-X,100-Y,1:PLOT 160+X,100-
Y,1
160 NEXT: NEXT
170 DO NULL
```

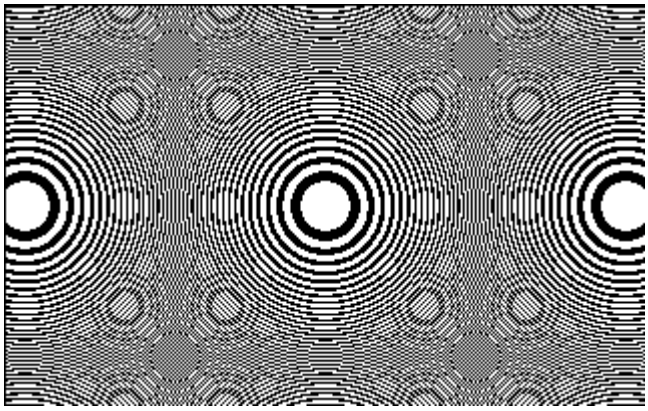


Bild 25 Noch ein Beispiel

Erzeugt die abgebildeten Grafiken (lange Laufdauer!)

Befehl:	POT	
Syntax:	**a = POT(<n>)** **PRINT POT(<n>)**	
Zweck:	ermittelt den Zustand eines Potentiometers (Paddle)	Ansprechen von Peripheriegeräten
Kürzel	p0	
Status:	Simons' Basic (numerische Funktion)	

POT eröffnet dem Programmierer die Möglichkeit, analoge Werte in digitale umzuwandeln, wobei der C64 nicht Spannungen, sondern Widerstandswerte erwartet. Diese Widerstände, am besten regelbare in Form von Potentiometern (Paddles) von z.B. 500 kOhm, kann man über die beiden Controlports mit dem Rechner verbinden. Jeder Port bietet Anschlüsse für zwei Potis (jeweils Pins 5 und 9 der Ports), deren Informationen zwei Analog-Digital-Wandlern im C64 (genauer: im Soundchip SID des C64) zugeführt werden. Pin7 stellt jeweils die erforderliche Spannung von +5 V zur Verfügung.

Simons' Basic verzichtet auf die Option, ein zweites Paddle-Paar an Port 2 abzufragen, und gibt mit POT nur die Default-Einstellung frei (Paddles/Potentiometer an Port 1). Mit ein paar POKEs könnte auch das zweite Paddle-Paar erfasst werden, dazu müsste man CIA 1 ($DC00 und $DC02, Bits 7 und 6) manipulieren.

Ein Wert von 0 für **<n>** in der Klammer von POT liest den Wert der Speicherstelle POTX ($D419) aus, ein Wert von 1 (oder größer) selektiert POTY ($D41A).

Beispiel: Motorsteuerung (Positionsregelung per Potentiometer)

```
900 DIM b(7),c(7)
910 FOR i=0 TO 7: b(i)=2^i: c(i)=255-b(i): NEXT i
1000 LOOP
1010   PRINT AT(16,1)"Sollwert der Drehung: ";
1020   FETCH "0123456789",3,soll
1030   EXIT IF soll>max
1040   EXIT IF soll<min
1050   LOOP
1060     y=POT(1)
1065     EXIT IF soll=y
1070     IF soll<y THEN bit=0
1080     IF soll>y THEN bit=1
1090     aus=aus OR b(bit)
1095     POKE $dd01,aus
1110     PRINT AT(18,1)"Istwert der Drehung: ";:USE "###",y
1120     aus=aus AND c(bit)
1130     POKE $dd01,aus
1140   END LOOP
1150   PRINT AT(16,1) DUP(" ",26)
1160 END LOOP
```

Kommentar: Hier wird mit den Potentiometern ein Roboterarm, der über den Userport ($DD01) mit dem C64 verbunden ist, in eine gewünschte Richtung bewegt (Zeilen 1050-1140). Der Programmausschnitt ist übernommen und angepasst aus dem Buch „Experimente zur Robotik“ von Roland Schulé, Franzis Verlag 1988, ISBN 3-7723-9461-2.

Befehl:	PROC	
Syntax:	PROC <label>	
Zweck:	Definieren eines (Prozedur-) Labels	Struktur
Kürzel	pR	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

In Simons' Basic können Programmzeilen mit einem Label (Namen) versehen werden. Auf diese Weise werden dadurch vor allem Unterprogramme unabhängig von ihrer Lage im Programm und der Programmierer kann leichter den Überblick bewahren (Namen lassen sich leichter einem Zweck zuordnen als Zeilennummern).

Der Befehl **PROC** dient dazu, den Namen eines solchen Unterprogramms (bzw. Sprungziels) zu definieren. Ein solcher Name darf andere BASIC-Schlüsselwörter enthalten (in TSB nicht an erster Position und nicht das Schlüsselwort DO .. DONE) und darf aus mehreren, durch Leerzeichen getrennten Wörtern bestehen. Ein führendes Leerzeichen ist signifikant, d.h. es muss dann auch bei EXEC verwendet werden (gilt **nicht** für TSB). In der gleichen Zeile wie PROC darf kein anderer BASIC-Befehl stehen (wird als zum Label gehörig betrachtet).

Beispiel:

```
10 PRINT "bitte eine taste druecken!"
20 warten auf taste
30 PRINT "danke"
999 END

1000 PROC warten auf taste
1010 DO NULL
1020 END PROC
```

Nach der Aufforderung, eine Taste zu drücken, wartet das Programm und gibt schließlich eine Rückmeldung aus.

Hinweis: **PROC** kann auch als einfaches Sprunglabel (ohne ein zugehöriges Unterprogramm) verwendet werden. Es wird dann mit CALL aufgerufen.

Befehl:	RCOMP	
Syntax:	**RCOMP** ..	
Zweck:	nimmt das Ergebnis der letzten durchlaufenen IF-Bedingungsklausel erneut auf	Struktur
Kürzel	rC	
Status:	Simons' Basic (Anweisung)	

RCOMP bildet das letzte Ergebnis von `IF <bedingung> THEN` erneut ab. Mit diesem Befehl kann man also eine Bedingungsklausel über mehrere Zeilen „verlängern". Auf RCOMP folgt ohne erneutes THEN der Wahr-Zweig der Ursprungsklausel (der sich dann auf jeder neuen Zeile unterscheiden kann), worauf (mit ELSE abgetrennt) auf jeder Zeile erneut auch ein Nicht-Wahr-Zweig folgen darf.

Beachten: Sowohl Wahr-Zweig als auch Nicht-Wahr-Zweig werden von RCOMP fortgeführt. Alle RCOMP-Zeilen verhalten sich also genauso wie die ursprüngliche Bedingungsklausel, es sei denn, hinter **RCOMP** folgt eine neue, zusätzliche IF-Klausel (dann wird deren Wahrheitswert weitertransportiert).

Beispiel:

```
10 PRINT "test ";
20 FETCH "jn",1,x$: PRINT
30 IF x$ = "j" THEN PRINT "ja": ELSE PRINT "nein"
40 RCOMP GOSUB 1000: ELSE GOSUB 1100
50 PRINT "ende"
50 END
1000 PRINT " zufrieden": RETURN
1100 PRINT " nicht zufrieden": RETURN
```

(je nachdem, ob der Benutzer „j" oder „n" drückt, erscheint eine andere Antwortausgabe, wenn er „n" drückt, werden z.B. alle ELSE-Zweige behandelt)

Befehl:	REC	
Syntax:	**REC <x>,<y>,<sa>,<sb>,<fq>**	
Zweck:	Zeichnen eines Rechtecks im Grafikmodus	Grafik-Befehle
Kürzel	rE	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

REC zeichnet ein Rechteck. Der Ort der linken oberen Ecke wird durch die beiden ersten Parameter **<x>** und **<y>** bestimmt, die Breite und Höhe des Rechtecks durch die Parameter drei und vier **(<sa>** und **<sb>**). Die Farbe der Kantenlinien wird durch den letzten Parameter (**<fq>**, Farbquelle) bestimmt. Lautet dieser Parameter „2“ (Invertieren), werden in TSB die vier Ecken des Rechtecks dennoch gezeichnet (in Simons‘ Basic nicht).

Zulässige Werte sind 0..319 für <x> und <sa> (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für <y> bzw. <sb> sind in beiden Fällen Werte von 0 bis 199 erlaubt. Auch die Farbe hängt vom Grafikmodus ab und bezieht sich auf die Farbangaben hinter HIRES einerseits bzw. MULTI und LOW COL andererseits. Der Punkt 0,0 ist in der linken oberen Ecke.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beachten: Bei Überschreitung der Bildschirmgrenzen bei den Kantenlängen des Rechtecks entstehen unkorrekte Ergebnisse (keine Rechtecke; dieser Fehler ist behoben in TSB).

Beispiel:

```
100 HIRES 0,1
105 REPEAT: GET x$
110 x=290*RND(1): y=180*RND(1)
120 BLOCK x,y,x+29,y+19,0
130 REC x,y,29,19,1
140 UNTIL x$>"": PAINT 0,0,1
150 DO NULL
```

(erzeugt die abgebildete Grafik)

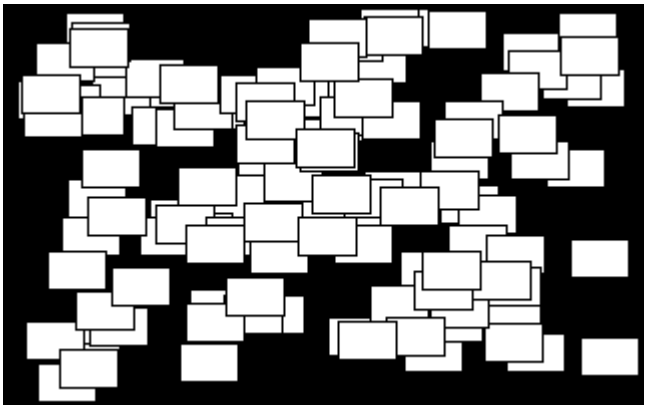


Bild 26 Rechtecke

Beispiel übernommen und angepasst aus dem Buch „Spiele mit Computergrafik“.

Befehl:	RENUMBER	
Syntax:	**RENUMBER [<start>, <step>]**	
Zweck:	Die Zeilennummern eines Programms umnummerieren	Programmierhilfen
Kürzel	reN	
Status:	Erweitertes Simons' Basic (Anweisung)	

Der Befehl **RENUMBER** ermöglicht, ein nach Spaghetti-Code aussehendes Programm (viele „Einschübe" zwischen dem normalen Abstand zweier BASIC-Zeilen, z. B. folgt auf Zeile 100 nicht 110, sondern 102, 104 und 105 usw.) wieder ansehnlich zu machen. Dazu gibt man hinter RENUMBER eine Startzeilennummer **<start>** und eine einzuhaltende Schrittweite **<step>** ein. Das Programm wird nach <RETURN> neu durchnummeriert.

Lässt man die Parameter weg, werden die Startzeile auf 100 und die Schrittweite automatisch auf 10 festgelegt.

Die Nummerierung wirkt auch auf Sprungzielangaben in Form von Zeilennummern innerhalb des Programms, und zwar bei folgenden Befehlen: **GOTO**, **GOSUB**, **THEN**, **LIST**, **RUN**, **RESET** und **RCOMP**.

Die Überschreitung der Höchstgrenze für Zeilennummern (63999) wird abgefangen, der Interpreter meldet dann einen **BAD MODE ERROR** und nummeriert nicht neu. Eine Schrittweite von **0** wird mit einem **ILLEGAL QUANTITY ERROR** abgewiesen.

Sollte im Programm ein Zeilennummernziel an eine nicht vorhandene Stelle führen, gibt RENUMBER die (noch nicht angepasste) Zeilennummer an, in der dieses fehlerhafte Ziel auftrat, ersetzt das unzutreffende Ziel durch die Zahl 0 (Null), nummeriert aber dennoch das Programm zu Ende, so dass in der Zeile mit dem falschen Ziel daraufhin z.B. ein `GOTO 0` erscheint (mit FIND danach suchen und ggf. korrigieren).

Hinweis: Möchte man nur Teile eines Programms umnummerieren, kann man den entsprechenden Teil mit LIN abspeichern, den eben gespeicherten Teil mit D! löschen und das Programm neu abspeichern, dann den Teilabschnitt gesondert laden, umnummerieren und mit dem ursprünglichen Programm per MERGE wieder zusammenfügen.

Befehl:	REPEAT	
Syntax:	**REPEAT**	
Zweck:	Fußgesteuerte Schleife einleiten	Struktur
Kürzel	reP	
Status:	Erweitertes Simons' Basic (Anweisung)	

REPEAT markiert den Anfang einer fußgesteuerten Schleife. Eine Schleife ist ein Teil eines Programms, der unter Umständen mehrfach durchlaufen wird (mehr zu Schleifen unter LOOP).

Beispiel 1, eine fußgesteuerte Schleife:

```
10 REPEAT
20   PRINT "Ja (j) oder Nein (n)? ";
30   FETCH "{crsr right}", 1, A$
40 UNTIL PLACE(a$, "jnJN")
```

Ja-Nein-Abfrage, die beliebige Zeichen zulässt, aber nur „j“ und „n“ (auch groß) durchreicht.

Beispiel 2, Tastaturabfrage:

```
10 REPEAT: GET X$: UNTIL X$>""
```

Tastaturabfrage, die in BASIC V2 mühsam mit der Speicherstelle 198, POKE und WAIT zusammengebastelt werden muss oder aber mit folgendem FOR-NEXT-Konstrukt umschrieben werden kann:

```
10 FOR WEITER = -1 TO 0
20 GET X$
30 WEITER = NOT X$>""
40 NEXT
```

Die Schleife terminiert erst, wenn der Ausdruck X$>"" wahr ist bzw. die Schleifenvariable WEITER den Wert für falsch bekommt, also diese den Wert 0 aufweist, den Endwert der Schleife.

Die Zeile 30 kann dabei auch kompakter formuliert werden (geht auch ohne Klammer):

```
30 WEITER = (X$ = "")
```

REPEAT kann zehn Mal verschachtelt werden.

Befehl:	RESET	
Syntax:	**RESET <zn>**	
Zweck:	gezieltes Anwählen von DATA-Zeilen	Ein-/Ausgabe
Kürzel	reS	
Status:	Simons‘ Basic (Anweisung)	

Mit **RESET** setzt man den internen DATA-Zeiger des Interpreters auf eine gewünschte BASIC-Zeile **<zn>**. Damit kann dieser Befehl sowohl Speicherplatz als auch Programmlaufzeit sparen, denn der bisherige RESTORE-Befehl setzt den DATA-Zeiger immer komplett auf den Anfang zurück. Nicht gewünschte DATA-Informationen müssen bei Verwendung von RESTORE u.U. erst in einer Schleife überlesen werden.

Beachten bei Simons‘ Basic: Eigentlich widerspricht ein solcher Befehl der Philosophie von Simons' Basic, denn zeilennummernbezogene Befehle will diese Befehlserweiterung ausdrücklich überflüssig machen. Wer sie dennoch verwendet, muss sich dessen bewusst sein, wenn er den Befehl RENUMBER benutzt oder die Programmstruktur auf andere Weise verändert. Die Befehle CGOTO, RESET, aber auch ON ERROR und ON KEY müssen dann von Hand an die neuen Verhältnisse angepasst werden (siehe dazu auch PROC, EXEC, ON ERROR, ON KEY, CGOTO und RENUMBER).

Auf TSB trifft diese Einschränkung nicht zu, da das TSB-RENUMBER auch die RESET-Werte anpasst.

Beachten: RESET fängt unmögliche Zeilennummern (größer als 63999) nicht ab, was aber zu keinem irreversiblen Zustand des Interpreters führt und nicht weiter schädlich ist.

Beispiel:

```
500 DATA arndt, annegrit, alid
510 DATA 1953, 1955, 1981

1010 RESET 510
1020 FOR i=1 TO 3: READ jahr(i): NEXT
```

(liest genau die Daten ein, die man lesen will und übergeht die anderen)

Befehl:	RESUME	1
Syntax:	**RESUME [NEXT | <ausdruck>]**	
Zweck:	setzt ON ERROR fort	Abfangen von Laufzeitfehlern
Kürzel	resU	
Status:	Neuer TSB-Befehl (Anweisung)	

RESUME ist hier Teil der ON-ERROR-Befehle von TSB. Es dient dazu, die Fehlerabfangroutine im BASIC-Programm zu beenden, ohne den Fehlerabfangmodus des Interpreters abzuschalten. RESUME kann dabei drei syntaktische Formen annehmen:

- **RESUME** (ohne Parameter) setzt das BASIC-Programm hinter dem fehlerauslösenden Befehl fort. Diese Syntax ist gut geeignet, um ein Programm auf Tippfehler und einfache logische Fehler zu überprüfen. Gedacht für den Programmierer.
- **RESUME NEXT** setzt das BASIC-Programm in der auf den Fehler folgenden Zeile fort. Diese Syntax sollte man einsetzten, wenn gleich eine ganze Serie von potenziell fehlergefährlichen Befehlen auf einer einzigen Zeile steht, z.B. Druckausgabebefehle.
- **RESUME <ausdruck>** ist gut, wenn man die fehlerauslösende Programmzeile noch ein weiteres Mal aufrufen möchte, nachdem man die Fehlerursache beseitigt hat, wenn z.B. der Drucker bei einer Druckausgabe nun eingeschaltet ist. In diesem Fall setzt man für <ausdruck> einfach ERRLN ein. Gedacht für den Einsatz beim Benutzer.

Beispiel:

```
10 ON ERROR: GOTO 10000

15 PRIN "{clr/home}": PRINT "ok"
20 READ b
25 PRINT b: GOTO 20
30 DATA 1,2,3,4,5

999 END
10000 PRINT: PRINT "Fehler Nr." ERRN "(";: OUT: PRINT ")"
10010 PRINT "Betroffene Zeile:" ERRLN: PRINT
10020 PRINT "Beenden (b)? Fortsetzen (f)? ";: FETCH "fb",1,x$: PRINT
10030 IF x$ = "b" THEN PRINT: PRINT "Abbruch nach Fehler!": NO ERROR:
STOP
10040 RESUME
```

Bei einem Programmlauffehler (in Zeile 15 ist statt „PRINT“ fälschlicherweise nur „PRIN“ angegeben) springt der Interpreter in die BASIC-Zeile 10000, gibt dort Informationen zum Fehler aus und stellt Handlungsalternativen zur Auswahl. Nach RESUME setzt das Programm fort in Zeile 20.

Befehl:	RESUME	2
Syntax:	**RESUME**	
Zweck:	Beendet eine ON-KEY-Tastatur-Kontrollroutine	Ein-/Ausgabe Struktur
Kürzel	resU	
Status:	Simons' Basic (Anweisung)	

RESUME bildet den Abschluss einer BASIC-Routine, die auf die Tastendrücke reagiert, die durch ON KEY abgefangen werden sollen. Der Interpreter fährt daraufhin hinter der Stelle fort, an der er durch den Tastendruck unterbrochen wurde. Da eine solche Routine mit DISABLE beginnen sollte, schaltet RESUME den ON KEY-Modus auch wieder ein.

Beispiel:

```
10 ON KEY "abc": GOTO 10000

... langes Programm ...

10000 DISABLE
10010 tt$ = CHR$(PEEK($c5ec))
10020 IF tt$ = "a" THEN PRINT AT(0,35)"ah!"
10030 IF tt$ = "b" THEN PRINT AT(0,35)"beh!"
10040 IF tt$ = "c" THEN PRINT AT(0,35)"zeh!"
10050 RESUME
```

(wenn der Benutzer "a", "b" oder "c" drückt, springt der Interpreter in die BASIC-Zeile 10000), RESUME setzt da im Programm fort, wo die Unterbrechung stattgefunden hat.

Befehl:	RETRACE	1
Syntax:	**RETRACE**	
Zweck:	Release-Informationen anzeigen	Programmierhilfen
Kürzel	reT	
Status:	Neuer TSB-Befehl (Kommando)	

RETRACE zeigt die Initialen des Autors (AD), das Release-Jahr des TSB-Interpreters (1986, bei TSBneo: 2023) und die aktuelle Versionsnummer.

Damit unterscheidet sich dieser Befehl völlig von seiner Vorlage in Simons' Basic.

Befehl:	RETRACE	2
Syntax:	**RETRACE**	
Zweck:	das letzte TRACE-Fenster anzeigen	Programmierhilfen
Kürzel	reT	
Status:	Simons‘ Basic (Kommando)	

RETRACE zeigt **in Simons‘ Basic** das letzte vom Befehl TRACE erzeugte Beobachtungsfenster (rechts am oberen Rand), allerdings nur dann, wenn der TRACE-Modus aktiv ist. Wenn nicht, erzeugt der Interpreter einen **BAD MODE ERROR**.

Das (RE-)TRACE-Fenster (rechts oben) überschreibt die dortige Bildschirmausgabe.

In TSB hat der Befehl eine andere Aufgabe.

Befehl:	RIGHT	
Syntax:	**RIGHTB | RIGHTW <zl>,<sp>,<bt>,<ho>**	
Zweck:	Rechtsscrollen eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	rI (für RIGHTB, RIGHTW hat keins)	
Status:	Erweitertes Simons' Basic (Anweisung)	

RIGHTB bzw. **RIGHTW** erlaubt es dem Programmierer, Bereiche des Textbildschirms ab der Stelle **<zl>** (Zeile) und **<sp>** (Spalte) mit einer Breite von **<bt>** und einer Höhe von **<ho>** inklusive der Farben spaltenweise nach rechts zu scrollen. In der äußerst linken, freiwerdenden Spalte werden je nach Typ des Befehls Leerzeichen aufgefüllt (RIGHTB, das „B" steht für „blank"), was nur einen einzigen kompletten Scrollvorgang erlaubt, oder die rechts herausfallenden Zeichen wieder eingefügt (RIGHTW, das „W" steht für „wrap"), was mit dem gleichen Inhalt immer wieder durchgeführt werden kann.

Es handelt sich bei diesem Scrolling um ein zeichenweises Scrolling, das u. U. ruckelig wirkt. Pixelweises Scrolling („Smooth Scrolling") ist mit Simons'-Basic-Befehlen nicht möglich.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter die Fehlermeldung **BAD MODE ERROR**.

Beachten: Wenn RIGHTW direkt am linken Bildschirmrand beginnt (Spalte 0, Zeile 0), kann es zu „Geisterzeichen" dort kommen, die störend sein können, allerdings keinen schwerwiegenden Folgefehler verursachen (Fehler behoben in TSB).

Beispiel:

```
100 CLS: COLOR 11,12,0
110 FOR X= 0 TO 39
120 Y=12*SIN(X/3)+12: PRINT AT(Y,X)"*"
130 NEXT
135 :
140 FOR Y=1 TO 3
150 FOR X=1 TO 40
160 LEFTW 0,0,20,25: REM DIVERGIEREN
170 RIGHTW 0,20,20,25
180 NEXT
190 FOR x=1 TO 40
210 RIGHTW 0,0,20,25: REM KONVERGIEREN
220 LEFTW 0,20,20,25
230 NEXT
240 NEXT
```

Das Beispiel erzeugt eine Sinuskurve, die sich bewegt. Abbruch ist per Tastendruck möglich. LEFTW und RIGHTW lassen hier die Kurve konvergieren bzw. divergieren.

Beispiel übernommen und angepasst aus dem Buch „Das Trainingsbuch zum Simons' Basic".

Befehl:	RLOCMOB	
Syntax:	RLOCMOB <n>, <zx>, <zy> [, <gr> [, <sp>]]	
Zweck:	Sprite steuern	Bearbeiten von Sprites
Kürzel	rL	
Status:	Erweitertes Simons' Basic (Anweisung)	

Mit **RLOCMOB** bewegt man das Sprite mit der Nummer **<n>** von seinem letzten Aufenthaltsort zu einem Zielpunkt mit den Koordinaten **<zx>,<zy>**. Ansonsten funktioniert dieser Befehl genau wie MMOB (weitere Informationen s. dort). Die Sprite-Nummer 7 darf man nicht überschreiten (wird von TSB überwacht, bei einem Fehler entsteht ein **BAD MODE ERROR**).

Die Koordinaten für Sprites stimmen nicht mit denen für die Grafik überein. Die Fläche, auf der sich Sprites bewegen können, ist viel größer als die Grafik- bzw. Textfläche. Diese ist so auf dem Sprite-Bereich angeordnet, dass ein normal dargestelltes Sprite auf allen Seiten hinter dem Bildschirmrahmen verschwinden kann. Insgesamt überstreicht der Sprite-Bereich eine Fläche von 512×256 Pixeln. Der Grafik-/Textbereich (der sichtbare Bildschirm) beginnt bei der Sprite-Koordinate x = 24 und y = 50.

Beachten: Nur die Werte für die Sprite-Nummer, die Y-Koordinate **<zy>** und für die Geschwindigkeit **<sp>** werden auf Plausibilität überprüft und erzeugen einen **ILLEGAL QUANTITY ERROR** (s. auch oben) bei Überschreitung der Höchstmarken. Wird die Sprite-Nummer zu groß gewählt (in TSB nicht möglich!), reagiert Simons' Basic mit unvorhersehbarem Verhalten des Programms. Für die X-Koordinate werden Werte bis 65535 akzeptiert (und auch abgearbeitet!) Werte größer als 3 bei der Anzeigegröße **<gr>** wirken alle wie der Wert 3.

Beispiel:

```
1800 PROC edit
1810   sprite
1815   kt$="{left arrow}{crsr right}{crsr left}{crsr down}{crsr up}":
       lg=0
1820   REPEAT:
         eingabe
1825     IF x$="{return}" THEN flip
1830     IF x$="{crsr down}" THEN z8=z8+16:
           IF z8>64+16*ch THEN z8=64+16*ch
1840     IF x$="{crsr up}" THEN z8=z8-16:
           IF z8<80 THEN z8=80
1850     IF x$="{crsr right}" THEN s8=s8+16:
           IF s8>22+16*cb THEN s8=22+16*cb
1860     IF x$="{crsr left}" THEN s8=s8-16:
           IF s8<38 THEN s8=38
1865     RLOCMOB 1,s8,z8,0,0
1870   UNTIL x$="{left arrow}"
1880   MOB ON/OFF 1
1890 END PROC
```

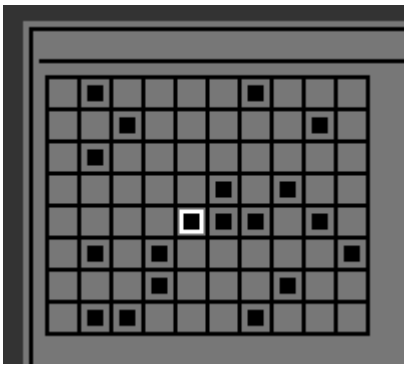


Bild 27 Editieren eines Kreuzwort-Gitters.

Kommentar: Der Prozeduraufruf „sprite“ in Zeile 1810 definiert das Sprite, hier: den weißen Kasten. Der Aufruf „eingabe“ fragt die Tastatur ab und liefert den Tastendruck in x$ zurück. Die Prozedur „flip“ dient dazu, ein Blindfeld in das Kreuzworträtsel einzufügen bzw. dieses wieder zu entfernen.

Drücken der Cursortasten bewegt das Sprite in die entsprechende Richtung, und zwar immer 16 Pixel (zwei Zeichen) weiter (um die Gitterlinien zu überspringen). Mit Drücken des Linkspfeils wird die Eingabeschleife beendet und das Sprite ausgeschaltet.

Befehl:	ROT	
Syntax:	ROT <r>, <g>	
Zweck:	Festlegen der Eigenschaften einer DRAW-Figur	Grafik-Befehle
Kürzel	rO	
Status:	Simons‘ Basic (Anweisung)	

ROT ergänzt den DRAW-Befehl, der eine ist ohne den anderen nicht verwendbar.

Mit dem zweiten Parameter von ROT (**<g>**) legt man fest, wie groß die Schrittweite der Richtungsanweisungen von DRAW sein soll. Die hier angegebene Schrittweite gilt für alle folgenden DRAW-Befehle bis zum nächsten ROT. Ein Wert von 0 entspricht in diesem Fall einer Schrittweite von 256 Pixeln. Punkte, die an einer Seite aus dem Grafikbereich herausragen, zeichnet der Interpreter entgegen seiner sonstigen Verhaltensweise an der gegenüberliegenden Seite weiter.

Der Parameter **<r>** bezeichnet den Rotationswinkel, den Simons' Basic bei der Darstellung der DRAW-Figur einhalten soll. Es ist nicht möglich, einen beliebigen Winkel einzugeben. Stattdessen ist man beschränkt auf acht verschiedene Winkel im Abstand von 45 Grad nach der folgenden Tabelle:

Befehl	Richtung	Winkel in Grad
0	N	0
1	NO	45
2	O	90
3	SO	135
4	S	180
5	SW	225
6	W	270
7	NW	315

Beachten: Die ungeraden Werte (also die Vielfachen von 45 Grad) führen dazu, dass die Figur **um etwa ein Viertel vergrößert** gezeichnet wird (genauer: es ist bei einer Länge von n um `n-n*1/SQR(2)` zu lang). Im Beispielprogramm sieht man, wie man diesem Problem entgegenwirken kann.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem Wert größer als 7 für die Rotation erscheint **BAD MODE ERROR**.

Beispiel:

```
100 w$="777755662255666600"
110 i$="727700"
115 k$="1111777711562577001111"
120 DIM s%(1)
```

```
125 HIRES 1,0:
    x=160:
    y=100:
    s=8
130 s%(0)=s:
    s%(1)=s-s/4
135 d$=w$+i$+k$+i$
140 REPEAT:
      GET x$:
      FOR w=0 TO 7
150     ROT w,s%(w AND 1):
        DRAW d$,x,y,1
160     DRAW d$,x,y,0
170   NEXT
180 UNTIL x$>""
185 ROT 0,s:
    DRAW d$,x,y,1
190 DO NULL
```

(erzeugt das Beispielbild)

Bild 28 DRAW und ROT(ate) in Aktion

Kommentar: Die Größe des Objekts wird mit s in Zeile 125 definiert. Da beim Drehen mit Mehrfachen von 45 Grad Verzerrungen entstehen (das Objekt wird etwa ein Viertel größer), werden diese in den Zeilen 130 und 150 wieder herausgerechnet. Korrekterweise müsste die Formel statt `s%(1)=s-s/4` lauten: `s%(1)=int(s/sqr(2)+.5)`, was in diesem Fall auf das gleiche Ergebnis hinausläuft.

Befehl:	SCRLD|SV DEF	
Syntax:	**SCRLD|SV DEF <ziel-hi>,<pages>,<modus>,<sa>**	
Zweck:	legt fest, wie (Bild-) Daten geladen/gespeichert werden sollen	Ein-/Ausgabe
Kürzel	scrL def | sC def	
Status:	Neuer TSB-Befehl (Anweisung)	

Mit dieser Anweisung steuert man das Verhalten eines nachfolgenden SCRLD oder SCRSV. Dabei spielen die Sekundäradresse und der Modus eine große Rolle bei dem, was genau geladen/abgespeichert wird.

Genaue Beschreibung von Modus und Sekundäradresse (Beispielwerte):

Zieladresse <ziel-hi>:	**Bedeutung**
$04	das Standard-Videoram beginnt an der Speicherposition $0400 (NRM)
$C0	die Farben eines Hires-Bildes liegen an Speicherposition $C000 (CSET 2)
$CC	das Videoram liegt nach MEM an Speicherposition $CC00 (MEM)
$E0	eine Bitmap (oder ein Zeichensatz) liegt in TSB an Speicherposition $E000 (CSET 2)

Pages <pages>:	**Bedeutung**
4	die beiden Textscreen-Komponenten sind je 4 Pages (je 1024 Bytes) lang
8	ein Zeichensatz ist üblicherweise 8 Pages (2048 Bytes) lang
16	der (zweifache) ROM-Zeichensatz des C64 ist 16 Pages (4096 Bytes) lang
32	eine Bitmap ist 32 Pages (8192 Bytes) lang

Modus <modus>:	**Bedeutung**
0	beide Textscreen-Komponenten laden/speichern (Videoram und Colorram)
1	nur Videoram laden/speichern (1024 Bytes, Colorram nicht beachten)
2	speichern im Append-Modus, an den Namen muss „,p,a“ angehängt werden (ansonsten wie Modus 1, beim Laden wirkt Modus 2 wie Modus 0)

Sek.-Adresse <sa>:	**Bedeutung**
2	Text-Screen laden/speichern (zwei Mal 1024 Bytes, byteweise alternierend)
3	Bitmap ohne Farben laden/speichern (8192 Bytes, Modus 0/1 wird nicht beachtet)
5	Bitmap mit Farben laden/speichern (unter Beachtung der Modus-Einstellung)

Beispiele:

1. **Textbildschirm *nach dem Einsatz von MEM* laden (bzw. speichern):**

```
100 MEM: sa=2
110 SCRLD DEF $cc,4,0,sa
```

```
120 SCRLD 2,dr,sa,"name"
130 SCRLD RESTORE
```

Die Datei „name“ ist eine Textscreen-Datei (Sekundäradresse **2**), der Textanteil wird aber nach **$CC00** geladen (dort liegt nach MEM das Videoram). Eine Komponente ist **4** Pages lang (1024 Bytes, beide zusammen 2048 Bytes) und SCRLD soll beide Dateikomponenten laden, sowohl den Text- als auch den Farbanteil (Modus: **0**).

2. **Einen Zeichensatz laden/speichern:**

Einen beliebigen fertigen Zeichensatz laden, der im Speicher z.B. an Position $E000 liegen soll (dem Ort, wo MEM seinen Zeichensatz hin kopiert) und eine Länge von 2048 Bytes hat (8 Pages):

```
100 sa=3
110 SCRLD DEF $e0,8,0,sa
120 SCRLD 1,dr,sa,“charset“
130 SCRLD RESTORE
```

3. **Hires-Grafik mit Farbe laden und *als Hi-Eddi-Bild* wieder speichern:**

```
laden (TSB-Format mit Farbe):

100 HIRES 1,0: sa=5
110 SCRLD 2,dr,sa,”name1"

speichern (Hi-Eddi-Format):

150 SCRSV DEF $e0,32,1,sa
160 SCRSV 3,dr,sa,"name2,p,w"
170 SCRSV RESTORE
```

Nachdem die Datei „name1“ (nach **$E000**) geladen ist, wird sie von ebendort wieder abgespeichert, wobei die Bitmap **32** Pages lang ist (8192 Bytes). Sie soll Farben enthalten (Sekundäradresse **5**), aber nur die Videoram-Komponente (aber nicht das Farbram) verwenden (Modus: **1**)

4. **Eine farbige Hires-Grafik *als Doodle-Bild* speichern:**

```
170 sa=2
180 SCRSV DEF $c0,4,1,sa
190 SCRSV 2,dr,sa,"name.dd,p,w"
...
230 sa=3
240 SCRSV DEF $e0,32,2,sa
250 SCRSV 3,dr,sa,"name.dd,p,a"
260 SCRSV RESTORE
```

Zuerst wird bei Doodle die Farbe gespeichert, das sind **4** Pages (Sekundäradresse **2**) ab **$C000**, aber ohne Colorram (Modus **1**). Im zweiten Durchgang kommt die Bitmap an die Reihe (**32** Pages, 8192 Bytes) ab **$E000**, dieser Durchgang ohne Farben (Sekundäradresse **3**), aber im Append-Modus (Modus: **2** und Suffix „,p,**a**“), sodass beide Durchgänge in einer einzigen Datei enden. Modus 2 verhindert den störenden Eintrag der Startadresse in die angehängte Datei.

5. **Kürzestmögliche Datei (eine Page, 256 Bytes) speichern:**

```
100 SCRSV DEF $c0,1,0,3: SCRSV 1,dr,3,“page-c0,p,w“
```

Befehl:	SCRLD|SV RESTORE	
Syntax:	**SCRLD|SV RESTORE**	
Zweck:	Macht die Änderungen durch SCRLD/SV DEF wieder rückgängig	Ein-/Ausgabe
Kürzel	scrL resT | sC resT	
Status:	Neuer TSB-Befehl (Anweisung)	

Nach dem Einsatz von SCRSV/LD DEF oder von POKEs zur Beeinflussung von SCRLD oder SCRSV sollte auf jeden Fall dieser Befehl eingesetzt werden, damit der Ursprungszustand von TSB wiederhergestellt wird.

Befehl:	SCRLD	
Syntax:	SCRLD <fn>,<dr>,<sa>,<name> [+ ",<ft> [,R] "]	
Zweck:	lädt einen Textbildschirm oder eine Grafik-Bitmap	Ein-/Ausgabe
Kürzel	scrL	
Status:	Erweitertes Simons' Basic (Anweisung)	

Mit **SCRLD** kann ein Programmierer vorher mit SCRSV gespeicherte **Textbildschirme** (z.B. Bildschirmmasken) direkt in den Screen laden. Dies geht schnell und spart Platz, denn der Code zum Aufbau einer solchen Maske braucht im Anwendungsprogramm nicht vorzukommen. Die 2048 Bytes lange Datei enthält sowohl den Textbildschirm ($0400) als auch die Farben der Zeichen darauf ($D800), byteweise abwechselnd Farbe und Screencode.

Der Befehl benötigt die gleichen Parameter, die auch der BASIC-Befehl OPEN verwendet:

- eine laufende Dateinummer **<fn>**, (1 bis 255)
- eine (Dummy-) Laufwerksangabe **<dr>** (es wird der mit USE eingestellte Drive verwendet)
- eine Sekundäradresse **<sa>**, wobei diese Zahl vom angeschlossenen Gerät abhängt; bei Floppys sind Werte von 2 bis 14 zulässig, für ***Textbildschirme*** muss die Sekundäradresse **gerade** sein (2, 4, ..), für ***Bitmaps*** **ungerade** (3, 5, .., s. unten)
- ein Dateiname **<name>**, mit einer Maximallänge von 16 Zeichen

Die folgenden Parameter Filetyp und Schreibmodus sind optional, vom System voreingestellt sind „P“ und „R“. Wenn mit „,P,W“ abgespeichert wurde, können sie beide wegfallen, ansonsten folgt also noch:

- eine Angabe über den gewünschten Dateityp **<ft>** der Datei, „S“ = SEQ, „U“ = USR und „P“ = PRG
- die letzte Angabe kann „R“ (für „Read“) lauten

Mögliche Fehlermeldungen sind alle diejenigen, die auch LOAD bzw. OPEN verursachen würden.

Beachten: Die momentane Cursorfarbe und die globalen Farben des Textbildschirms (Hintergrund und Rahmen) sind nicht in der Datei enthalten und sollten vielleicht im Dateinamen (oder in der Datei selbst an festgelegter Position) festgehalten werden, damit sie nicht in Vergessenheit geraten.

Unter TSB ist der Befehl **SCRLD** auch in der Lage, die **Bitmap** eines Grafikbildes zu laden. Dazu muss die Sekundäradresse **3** lauten (siehe auch Beispiel). TSB lädt 8192 Bytes, d.h. den kompletten Grafikpuffer (unter TSB ab $E000), erwartet also eine 33 Blocks große Datei. Bei Sekundäradresse **5** speichert TSB sowohl die Bitmap als auch die Farben (Videoram und Colorram) in eine Datei (41 Blocks).

Hinweis: Wenn man mit Sekundäradresse 3 gespeichert hat (nur die Bitmap), die Farben aber auch gebraucht werden, hat man die Möglichkeit, mit `POKE $A5DB,$C0: SCRSV 1,DR,2,"NAME,P,W": POKE $A5DB,4` die Grafikfarben in einer zweiten Datei zu speichern. Man kann sie dann mit den gleichen POKEs auch wieder laden (s. Beispiel). Bei Bildern im Multicolor-Modus steht die Hintergrundfarbe in der Datei an Position $C3FF (nach dem Laden, s. SCRSV).

Beispiel für zwei getrennte Dateien:

```
1550 PROC TSBLOAD
1560 CSET 2: MULTI ON
1565 SCRLD 1,x,3,"IMAGE *"
1570 POKE $a5db,$c0
1575 SCRLD 1,x,2,"IMAGECOLS"
1580 POKE $a5db,4
1585 COLOR 16, BG
1595 END PROC
```

Lädt eine Multicolor-Grafik samt Farben aus einer zweiten Datei unter dem angegebenen Namen von Disk. Die Variable BG enthält die Nummer der Hintergrundfarbe aus dem Dateinamen "IMAGE"+STR$(BG). Hat man Bild und Farben in eine einzige Datei gespeichert, fallen die Zeilen 1570 bis 1580 weg und Zeile 1565 muss SCRLD 1,x,5,“IMAGE *“ lauten.

Beispiel für eine Datei, die beides enthält, Bitmap und Farben:

```
1550 PROC TSBLOAD2
1560 SCRLD 1,x,5,“IMAGE BG“
1570 END PROC
```

Handelt es sich um ein Hires-Bild, braucht man die Angabe BG im Namen nicht. Bei einer Multicolordatei gibt man bei der Anzeige des Bildes den Befehl COLOR 16, bg ein.

Tipp: Mit einem weiteren POKE lassen sich **beliebige zwei** frei zugängliche **1-KByte-Bereiche**, z. B. Zeichensätze, laden. Angenommen, ein Zeichensatz liegt von Adresse $3000 bis $37ff im Speicher. Mit `POKE $A5DB,$30: POKE $A5D7,$34: SCRSV 1,x,2,“NAME,P,W“: POKE $A5DB,4: POKE $A5D7,$D8` kann er mit SCRSV gesichert und jederzeit mithilfe dieser POKEs mit SCRLD an beliebige andere Stellen zurück geladen werden (s. auch SCRLD|SV DEF).

Beliebige **8-KByte-Bereiche** speichert/lädt man mit Sekundäradresse 3 und dem POKE $A5DF,Highbyte. Der Defaultwert für $A5DF ist $E0.

Befehl:	SCRSV	
Syntax:	SCRSV <fn>,<dr>,<sa>,<name> + ",<ft>,W"	
Zweck:	speichert einen Textbildschirm oder eine Grafik-Bitmap	Ein-/Ausgabe
Kürzel	sC	
Status:	Erweitertes Simons' Basic (Anweisung)	

Mit **SCRSV** kann ein Programmierer Textbildschirme (z.B. Bildschirmmasken) abspeichern, um sie später bei Bedarf schnell und Platz sparend zur Verfügung zu stellen, denn der Code zum Aufbau einer solchen Maske bräuchte im endgültigen Anwendungsprogramm nicht vorzukommen. SCRSV speichert eine 2048 Bytes lange Datei, die sowohl den Textbildschirm ($0400) als auch die Farben der Zeichen darauf ($D800) enthält (in alternierenden Bytes, zuerst Farbe, dann Zeichen in Bildschirmcode).

Der Befehl benötigt die gleichen Parameter, die auch der BASIC-Befehl OPEN verwendet:

- eine logische Dateinummer **<fn>**, (1 bis 255)
- eine (Dummy-) Laufwerksangabe **<dr>** (es wird der mit USE eingestellte Drive verwendet)
- eine Sekundäradresse **<sa>**, wobei diese Zahl vom angeschlossenen Gerät abhängt; bei Floppys sind Werte von 2 bis 14 zulässig, für Textbildschirme muss die Sekundäradresse **gerade** sein (2, 4, ..), für Bitmaps **ungerade** (3,5, .., s. unten)
- ein Dateiname **<name>**, mit einer Maximallänge von 16 Zeichen
- eine Angabe über den gewünschten Dateityp **<ft>** der Datei, „S" = SEQ, „U" = USR und „P" = PRG
- die letzte Angabe muss zwingend „**W**" (für „Write") lauten, sonst wird nicht gespeichert

Mögliche Fehlermeldungen sind alle diejenigen, die auch SAVE bzw. OPEN verursachen würden.

Beachten: Die Multicolor-Hintergrundfarbe wird nicht gespeichert und sollte vor dem Speichern in der Datei per POKE an der Position $C3FF festgehalten werden, damit sie beim Wiedereinlesen mit SCRLD bei Sekundäradresse 5 (s. weiter unten) parat steht. Bei Hires-Bildern sollte dort vorsichtshalber eine Null eingetragen werden (ist aber standardmäßig dort zu finden).

Unter TSB ist der Befehl SCRSV in der Lage, die Bitmap des aktuellen Grafikbildes abzuspeichern. Dazu muss die Sekundäradresse **<sa>** ungerade sein, typischerweise wird „3" verwendet (siehe auch Beispiel). TSB speichert 8192 Bytes, d.h. den kompletten Grafikpuffer (unter TSB ab $E000) und erzeugt eine 33 Blöcke große Datei.

Hinweis: Die Bildfarben der Bitmapgrafik speichert TSB mit Sekundäradresse 3 nicht ab, es gibt aber zwei einfache Möglichkeiten, auch diesen Nachteil zu umgehen. Erstens, mit `POKE $A5DB, $C0: SCRSV 1,DR,2,"NAME,P,W": POKE $A5DB,4` speichert man die Grafikfarben in einer zweiten Datei ab (Dateilänge: 9 Blocks). Bei Bildern im Multicolor-Modus muss man sich dann die Hintergrundfarbe merken, am besten im Dateinamen der Farbdatei, damit die Farbnummer nicht verloren geht. Der Dateityp muss nicht zwingend angegeben werden. Fehlt die Angabe, wird eine PRG-Datei erzeugt.

Beispiel

```
1500 PROC TSBSAVE
1510 SCRSV 1,x,3,"IMAGE,P,W"
1520 POKE $A5DB,$C0
1530 SCRSV 1,x,2,"IMAGECOLS"+STR$(BG)+",P,W"
1540 POKE $A5DB,4
1545 END PROC
```

Das Beispiel speichert eine Grafik und in einer zweiten Datei deren Farben unter dem angegebenen Namen auf Disk. In Variable BG befindet sich die Nummer der Hintergrundfarbe. Statt beim Namen kann man natürlich auch hier die Speicherstelle $C3FF für die Hintergrundfarbe verwenden.

Zweitens kann man Bild und Farben mit der Sekundäradresse **5** in **einer einzigen** Datei ablegen (Länge: 41 Blocks), die sich mit SCRLD und der gleichen Sekundäradresse auch wieder laden lässt.

Tipp (unabhängig von Grafik): Mit einem weiteren POKE lassen sich **beliebige zwei** frei zugängliche **1-KByte-Bereiche**, z. B. Zeichensätze, laden. Angenommen, ein Zeichensatz liegt von Adresse $3000 bis $37ff im Speicher. Mit `POKE $A5DB,$30: POKE $A5D7,$34: SCRSV 1,X,2,"NAME,P,W": POKE $A5DB,4: POKE $A5D7,$D8` kann er gesichert und jederzeit mithilfe dieser POKEs an beliebige andere Stellen zurückgeladen werden (mit SCRLD). Die Defaultwerte für die Speicherstellen $A5D7 und $A5DB lauten $D8 und $04 (s. auch SCRLD|SV DEF).

Beliebige **8-KByte-Bereiche** speichert/lädt man mit Sekundäradresse 3 und POKE $A5DF, Highbyte. Der Defaultwert für $A5DF ist $E0.

Befehl:	SECURE	
Syntax:	**SECURE 0 | ON**	
Zweck:	Unsichtbarmachen einer BASIC-Zeile mit DISAPA	Programmierhilfen
Kürzel	sE	
Status:	Simons‘ Basic (Kommando)	

Mit **SECURE** wird in Simons‘ Basic eine BASIC-Zeile, die mit DISAPA vorbereitet wurde, für den LIST-Befehl unsichtbar gemacht.

Zur Wirkungsweise des Schutzes siehe Befehl DISAPA. Der Schutz ist allerdings wenig wirkungsvoll. SECURE wird im Direktmodus des Interpreters angewendet, funktioniert aber auch innerhalb eines Programms. Aus einer solchen Zeile:

```
100 DISAPA:IF x$<>"geheim" THEN STOP
```

macht der Interpreter nach Anwendung von SECURE die Zeile

```
100
```

SECURE und DISAPA sind als Programmschutz nicht empfehlenswert, da sie ohne Aufwand umgangen werden können.

Beachten: Der Befehl DISAPA ist mit v2.40.215 aus dem Befehlssatz von TSB entfernt worden, wodurch SECURE in TSB keine Funktion mehr erfüllt.

Hinweis: In der Cartridge-Version von TSB wurde mit v2.50.127 der Befehl SECURE aus dem Befehlssatz von TSB entfernt und durch das Schlüsselwort **MONITOR** zum Aufrufen von TSB.MON ersetzt.

Befehl:	SOUND	1
Syntax:	**a = SOUND** **PRINT SOUND**	
Zweck:	Gibt die Basisadresse des SID zurück	Sound
Kürzel	sO	
Status:	Simons' Basic (Systemkonstante)	

SOUND liefert die Speicheradresse zurück, an der der Sound-Chip des C64 (SID) liegt und kontrolliert werden kann: 53972 bzw. $D400.

Beispiel:

```
10 CLS
20 REPEAT: GET x$
30 a=PEEK(SOUND+27)
20 PRINT AT(0,0) a "{3xSpace}";
30 UNTIL x$>""
```

(Wenn zuvor der Rauschgenerator der Stimme drei aktiviert wurde, wird der Variablen a aus $D41B ständig ein zufälliger Wert von 0 bis 255 zugewiesen und dies wird angezeigt)

Befehl:	SOUND	2
Syntax:	**SOUND <n>, <fr>**	
Zweck:	Legt für Stimme <n> eine Tonhöhe <fr> fest.	Sound
Kürzel	sO	
Status:	Neuer TSB-Befehl (Anweisung)	

1. **SOUND** legt für einen zu spielenden Ton auf einem der drei SID-Oszillatoren **<n>** (1..3) die Tonhöhe fest. (Eingeschaltet wird er mit WAVE.)

Die Tonhöhe erstreckt sich beim C64 von 16Hz bis 3738Hz. Sie wird als 16Bit-Wert erwartet, der sich aus Frequenz und Prozessor-Taktrate ergibt. Für PAL-C64 lauten die Formeln für die Tonhöhe: **<fr> = Frequenz*17.028412** und umgekehrt **Frequenz = <fr>/17.028412**.

Beispiel

```
9 CSET 1: x$="":
10 s=SOUND: MEMCLR s,29: WAVE 1,0,$0800: w=RND(-ti)
11 REPEAT: get x$
12 CGOTO int(rnd(1)*4)+13
13 w=17 : w$="Dreieck"  : goto 17
14 w=33 : w$="Saegezahn": goto 17
15 w=65 : w$="Rechteck" : goto 17
16 w=129: w$="Rauschen"
17 VOL 15: ENVELOPE 1,6,1,12,8: WAVE 1,w
18 print w$
19 for x=0 to 255 step (rnd(1)*15)+1
20 f=x+256*(255-x): SOUND 1,f
21 for y=0 to 33 : next y,x
22 for x=0 to 200: next: vol 0
23 for x=0 to 100: next
24 UNTIL x$>"": WAVE 1,0
```

Das Programm spielt mit zufällig gewählten Schwingungsformen (Zeile 12) zufällig gewählte Töne (Zeile 19) mit abfallender Tonhöhe in zufällig gewählter Dauer (auch Zeile 19). Abbruch auf Tastendruck (die Reaktion kann einen Moment auf sich warten lassen). Der WAVE-Befehl in Zeile 10 gibt für die Rechteckschwingung eine (ideale) Pulsbreite vor, sonst würde man bei der Rechteckschwingung (w=65) im Verlauf des Programms nichts hören.

---

2. **SOUND 4** definiert die Filterfrequenz für Tief-, Band- oder Hochpassfilter in $D415/$D416.

Festlegen der Grenzfilterfrequenz mit SOUND 4,<fr> am Beispiel <fr>=2047:

```
10 def fn gf(x)=256*div(x,8)+(x and 7)
20 SOUND 4, fn gf(2047)
```

Die Grenzfrequenz wird vom [[SID]] als 11Bit-Wert aus den beiden Registern $D415/$D416 ermittelt, wobei die Bits 10 bis 3 aus $D416 und die Bits 2 bis 0 aus $D015 genommen werden. Sie kann Werte von 0 bis 2047 = (2^{11}) - 1 haben. Die Funktion gf(x) liefert zur Grenzfrequenz x den passenden 16Bit-Wert für $D415/$D416.

Befehl:	TEST	
Syntax:	**a = TEST(<x>,<y>)** **PRINT TEST(<x>,<y>)**	
Zweck:	Prüfen, ob ein Punkt gesetzt ist	Grafik-Befehle
Kürzel	tE	
Status:	Simons' Basic (Anweisung)	

TEST dient dazu, herauszufinden, ob ein Punkt gesetzt ist (in der Farbe <ink> erscheint, s. HIRES) oder nicht (also in der Hintergrundfarbe erscheint). Zulässige Werte für **<x>** sind 0..319 (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für **<y>** sind in beiden Fällen Werte von 0 bis 199 erlaubt. Der Punkt 0,0 ist in der linken oberen Ecke.

Im Multicolor-Modus liefert die Funktion den Wert der **Farbquelle** zurück: **0** ist die Hintergrundfarbe (gleichbedeutend mit „**Punkt nicht gesetzt**"), **1..3** sind die entsprechenden Farbquellen bei MULTI bzw. LOW COL. Diese Werte werden von TEST auch in der Systemvariablen ST (STATUS) zwischengespeichert und sind dort abrufbar, bis ST geändert wird. Das Handbuch erwähnt diese Eigenschaft von TEST nicht.

Wird kein Argument in der Funktion angegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **ILLEGAL QUANTITY ERROR**.

Beispiel:

```
100 HIRES 8,11: bb=-1: vv=.6
105 CIRCLE 160,100,100,100,1: PAINT 160,100,1

110 BLOCK 0,0,8,8,0
120 a=a+1: IF a>26 THEN a=1
130 CHAR 0,0,a,1,1

140 bb=b+.008/COS(bb): ll=ll+.2/COS(bb)
150 IF bb>1.45 THEN 240
160 FOR i=0 TO 8 STEP .5: FOR j=0 TO 8
STEP .5
170 IF TEST(i,8-j)=0 THEN 230
180 b=bb+.02*j: l=ll+.02/COS(bb)*i
190 x=SIN(l)*COS(b): y=COS(l)*COS(b):
z=SIN(b)
200 u=y*COS(vv)+z*SIN(vv): v=-
y*SIN(vv)+z*COS(vv)
210 IF u<0 THEN i=8: j=8: GOTO 230
220 PLOT 160+100*x,100-100*v,0
230 NEXT: NEXT: GOTO 110

240 CIRCLE 160,100,100,100,1
250 BLOCK 0,0,8,8,0
260 DO NULL
```

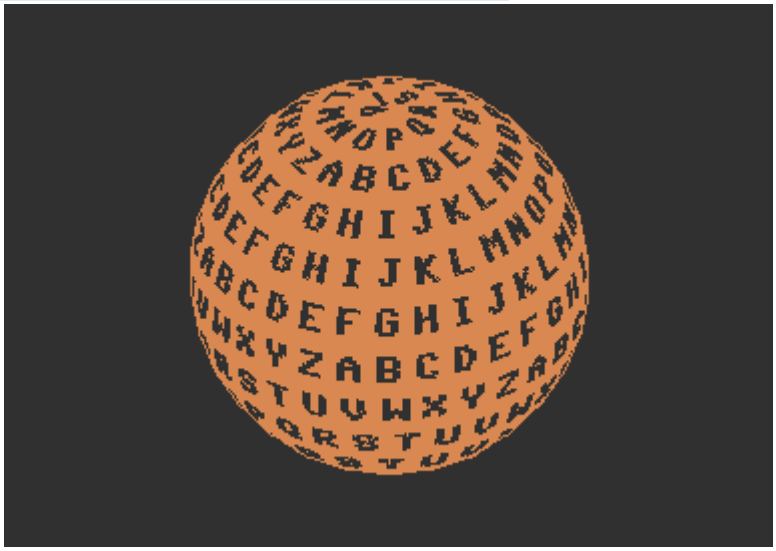


Bild 29 TEST, PLOT und ein paar Formeln

Das Beispiel erzeugt die abgebildete Grafik; vv steuert die Neigung der Kugel - (-π..+π), mit u wird festgestellt, ob ein Punkt auf der Kugel sichtbar ist; TEST prüft, ob an der aktuellen Stelle im Buchstaben selbst ein Bit gesetzt ist oder nicht. Lange Abarbeitungszeit.

Befehl:	TEXT	
Syntax:	**TEXT <x>,<y>,<string> [,<fq> [,<zm> [,<d>]]]**	
Zweck:	Schreiben von Texten im Grafikmodus	Grafik-Befehle
Kürzel	teX	
Status:	Erweitertes Simons' Basic (Anweisung)	

TEXT schreibt einen Textstring in der Farbe der angegebenen Farbquelle **<fq>** (s. dazu HIRES) in den Grafikbildschirm.

Zulässige Werte für **<x>** sind 0..319 (im Hires-Modus) bzw. 0..159 (im Multicolor-Modus). Für **<y>** sind in beiden Fällen Werte von 0 bis 199 erlaubt. Der Punkt 0,0 ist in der linken oberen Ecke.

Für **<string>** darf jede gültige Stringvariable eintreten, aber auch Textkonstanten in Anführungszeichen. Der Parameter **<d>** gibt die Distanz zwischen den linken Kanten zweier Zeichen des Textes an, also die Weite einer eventuellen Sperrschrift, der Wert des normalen Abstands ist 8. Mit **<zm>** kann die Schrift vergrößert werden (der Wert 2 verdoppelt die Größe der Zeichen).

Man kann innerhalb eines Strings angeben, ob die Zeichen in Groß-Grafik- oder in Groß-Klein-Schrift ausgegeben werden sollen. Zuständig dafür sind die „Umschaltzeichen" **<CTRL-a>** (Groß-Grafik) und **<CTRL-b>** (Groß-Klein) vor der entsprechenden Textpassage. Vorgabe-Schriftart ist Groß-Grafik. Leider wirkt sich der Zoom (<zm>) nur auf die Höhe der Zeichen aus, nicht jedoch auf deren Breite. Zooms größer als 5 wirken daher bereits unansehnlich (vgl. dazu DUP).

Beachten: Die drei letzten Parameter dürfen unter Simons' Basic nicht als Variable angegeben werden, da der Interpreter dadurch durcheinandergerät und unvorhersehbare Reaktionen zeigt. (Dieser Mangel wurde unter TSB behoben.) In TSB kann man diese Parameter von rechts her einzeln weglassen. Es werden dann die Defaultwerte 1, 1 und 8 gesetzt.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**, bei einem falschen Wert erscheint **BAD MODE ERROR**.

Beispiel:

```
100 HIRES 11,12
110 FOR y=1 TO 8
120 TEXT 64,25*y-20,"{ctrl-b}Commodore",1,2,20
130 NEXT
140 DO NULL
```

(erzeugt ein einfaches Demo)

Hinweis: Man kann beliebige Zeichensätze verwenden, wenn man sie an eine geeignete Stelle im Speicher lädt und ihre Position dann mit `POKE $B34B,hibyte` bekannt macht.

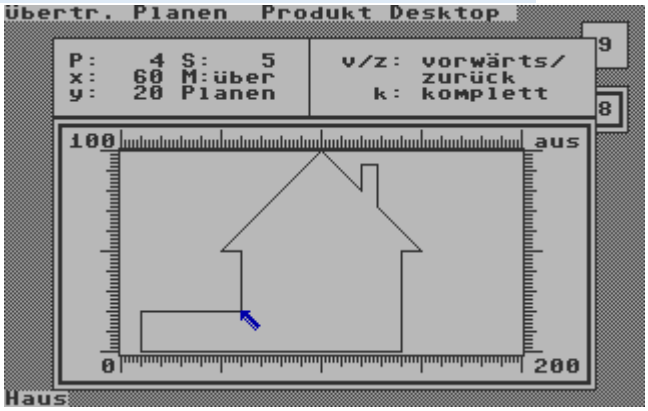


Bild 30 Ein Splitscreen, die Beschriftung im Grafikteil: mit TEXT

Befehl:	TRACE	
Syntax:	TRACE <n> TRACE ON | OFF	
Zweck:	momentan bearbeitete BASIC-Zeile anzeigen	Programmierhilfen
Kürzel	tR	
Status:	Erweitertes Simons' Basic (Anweisung)	

Mit **TRACE 10** (bzw. **TRACE ON**) schaltet man den Programm-„Verfolgungsmodus“ ein, d.h. der Interpreter zeigt am oberen Rand des Bildschirms die komplette aktuell vom Interpreter bearbeitete BASIC-Zeile. Ein mitlaufender Cursor markiert den momentan aktiven BASIC-Befehl (und springt bei Bedarf in der angezeigten Zeile hin und her). Da die Anzeige u.U. rasend schnell ablaufen kann, hat der Benutzer die Möglichkeit, mithilfe der **Commodore-Taste** die Geschwindigkeit des Programms zu drosseln. Drückt er zusätzlich noch die **Shift-Taste**, bleibt die Bearbeitung des Programms vollends stehen und setzt sich erst wieder fort, wenn Shift losgelassen wird.

Der TRACE-Befehl gibt uns ein Werkzeug, das es erlaubt zu überprüfen, ob das Programm überhaupt die vom Programmierer gewünschten Befehle durchläuft. Der TRACE-Modus wirkt nicht im Direktmodus (z.B. nach Programmende oder -abbruch), wird jedoch fortgesetzt nach einem erneuten RUN oder GOTO. Er endet, wenn TRACE mit einem anderen Parameter als dem Wert 10 aufgerufen wird (sinnvoll ist der Wert 0) bzw. mit **TRACE OFF**.

Die TRACE-Anzeige (am oberen Rand) überschreibt die dortige Bildschirmausgabe.

Hinweis: Im Direktmodus wird der TRACE-Modus ausgesetzt, aber nicht abgeschaltet. Nach RUN oder GOTO ist er wieder aktiv. Bei Simons' Basic werden nur die Zeilennummern angezeigt.

Befehl:	UNTIL	
Syntax:	**UNTIL <bedingung>**	
Zweck:	Markiert das Ende eines fußgesteuerten Schleifenkörpers	Struktur
Kürzel	uN	
Status:	Erweitertes Simons' Basic (Anweisung)	

UNTIL definiert das Ende einer fußgesteuerten Schleife. Eine Schleife ist ein Teil eines Programms, der unter Umständen mehrfach durchlaufen wird. Bei fußgesteuerten Schleifen wird erst am Ende der Schleife abgefragt, ob die Abbruchbedingung erfüllt ist, so dass der Schleifenkörper mindestens einmal durchlaufen wird. Mehr zu Schleifen findet sich bei LOOP.

Beispiel für eine fußgesteuerte Schleife:

```
10 REPEAT: GET x$: UNTIL x$>""
```

Eine Tastaturabfrage, bei der die Schleife verlassen wird, wenn eine Taste gedrückt wurde.

Befehl:	UP	
Syntax:	**UPB | UPW <zl>,<sp>,<bt>,<ho>**	
Zweck:	Aufwärtsscrollen eines Bildschirmbereichs	Bearbeiten des Textbildschirms
Kürzel	-	
Status:	Erweitertes Simons' Basic (Anweisung)	

UPB bzw. **UPW** erlaubt es dem Programmierer, einen Bereich des Textbildschirms ab **<zl>** (Zeile) und **<sp>** (Spalte) mit der Breite **<bt>** und der Höhe **<ho>** inklusive der Farben zeilenweise nach oben zu scrollen. In der untersten, freiwerdenden Zeile werden je nach Typ des Befehls Leerzeichen aufgefüllt (UPB, das „B" steht für „blank"), was nur einen einzigen kompletten Scrollvorgang erlaubt, oder die oben herausfallenden Zeichen wieder eingefügt (UPW, das „W" steht für „wrap"), was mit dem gleichen Inhalt immer wieder durchgeführt werden kann.

Leider handelt es sich bei diesem Scrolling um ein zeichenweises Scrolling, das u.U. (bei großen zu scrollenden Flächen) ruckelig wirkt. Pixelweises Scrolling („Smooth Scrolling") ist mit Simons'-Basic-Befehlen nicht möglich.

Bei Über- oder Unterschreitung der zulässigen Werte meldet der Interpreter die Fehlermeldung **BAD MODE ERROR**.

Beachten: Wenn UPB oder UPW direkt am unteren Bildschirmrand enden (Zeile 24), kann es in Simons' Basic zu „Geisterzeichen" dort kommen, die störend sein können, allerdings keinen schwerwiegenden Folgefehler verursachen (behoben in TSB).

Befehl:	USE	1
Syntax:	**USE <const>** **USE 0 + <var>**	
Zweck:	Systemweite Drivenummer festsetzen	Ein-/Ausgabe
Kürzel	uS	
Status:	Neuer TSB-Befehl (Anweisung)	

TSB hat (anders als Simons' Basic) die Eigenschaft, dass laufwerksbezogene Befehle auch auf andere als nur das Standardlaufwerk 8 zugreifen können. Weil dazu aber nötig ist, dass das gewünschte Laufwerk eingestellt wird, gibt es den Befehl **USE**. Folgt auf USE eine Zahlenkonstante, so wird diese als Laufwerksnummer interpretiert. Es sind Werte für **<const>** von 0 bis 31 zulässig (somit können auch CMD-Laufwerke wie z.B. ein RAMLink angesprochen werden).

Beachten: TSB prüft nicht, ob die Laufwerksangabe plausibel ist. Greift der Befehl auf ein nicht vorhandenes Laufwerk zu, meldet es einen **DEVICE NOT PRESENT ERROR**.

Hinweis: USE <const> kann auch als Anweisung in einem Programm verwendet werden, sogar Variablen kann man durch einen Trick bei USE <const> einbauen: Der USE-Parameter wird in Wirklichkeit schon als Ausdruck ausgewertet, der Zwang zur Konstante besteht nur deshalb, weil der Parameter vom Interpreter eindeutig seinem Zweck zugeordnet werden muss. Also erweitert man die Konstante mit der gewünschten Variablen, und zwar so: **USE 0+dr**, wobei DR die Laufwerksnummer enthält. Ansonsten muss man mit IF..THEN..ELSE arbeiten. Weitere Alternative: Der Wert der Konstanten wird in der Speicherstelle $BE (dez. 190) abgelegt und kann dort beliebig verändert oder abgerufen werden.

Befehl:	USE	2
Syntax:	**a = USE** **PRINT USE**	
Zweck:	Systemweite Drivenummer abfragen	Ein-/Ausgabe
Kürzel	uS	
Status:	Neuer TSB-Befehl (Systemvariable)	

Wenn man mit USE 9 ein neues Bezugslaufwerk auswählen kann, ist dies die Umkehrfunktion dazu: Mit a=USE (oder PRINT USE) lässt sich das aktuell eingestellte Bezugslaufwerk abfragen.

Befehl:	USE	3
Syntax:	**USE [#n,] [AT(zl,sp)] <string>, <zahl> [, zahl...] [;]**	
Zweck:	dezimalorientierte Ausgabe von Zahlen	Bearbeiten und Darstellen von Zahlen
Kürzel	uS	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

USE dient zum Formatieren von Zahlen, so dass sichergestellt ist, dass Dezimale immer an der gewünschten Position stehen. In BASIC-Dialekten auf PCs erscheint USE als „USING“ abhängig von PRINT (`PRINT USING <string>,<zahlen>`, zu deutsch: „Drucke die <zahlen>, indem du <string> als Vorlage benutzt“). In TSB ist der Befehl eigenständig. Mit dem optionalen Parameter AT legt man fest, an welcher Stelle auf dem Bildschirm USE eingesetzt werden soll.

Im Formatstring **<string>** kann beliebiger Text an beliebiger Stelle stehen, die Zahlen in der Zahlenliste werden auf Platzhaltersymbole (Rautenzeichen „#“ oder Stern „*“, für jede Ausgabeziffer steht ein Platzhaltersymbol) aufgeteilt. Das Trennzeichen zum Nachkommaanteil von Fließkommazahlen ist entweder der Dezimalpunkt („.“) oder das x Dezimalkomma („,“).

Im Formatstring können Platzhalter für mehrere Zahlen eingebaut werden, sodass man geordnete Listen mit einem einzigen Format erstellenkann. In der Liste der Zahlen (**<zahl> [, zahl...]**) werden die einzelnen Zahlen mit einem Komma getrennt. Sollten in der Liste Konstanten vorkommen, muss das Trennzeichen zum Nachkommateil bei ihnen der Basic-Syntax folgend natürlich ein Dezimalpunkt sein. Die Position des Vorzeichens bestimmt man durch Setzen des Vorzeichens an die gewünschte Stelle (vor oder nach dem Platzhalter), ein „**+**“ erzwingt ein Vorzeichen, ein „**-**“ lässt nur bei negativen Zahlen das Vorzeichen erscheinen. Kein Vorzeichen im Formatstring lässt den Interpreter vorzeichenlose Zahlen anzeigen. Ein „*****“ (Stern) als führender Platzhalter oder innerhalb des Vorkommaanteils des Formatstrings wird vor die formatierte Ausgabe gesetzt füllt alle von einer Zahl nicht benutzten führenden Platzhalterstellen mit Sternen.

Mit **AT(zl,sp)** kann man die Ausgabeposition des ausgefüllten Formatstrings auf dem Bildschirm festlegen.

Wenn am Ende des USE-Befehls ein „**;**“ (Semikolon) gesetzt wird, hat das den gleichen Effekt wie beim Befehl PRINT (kein Zeilenwechsel nach der Ausgabe).

Hinweise:

- USE gibt die Zahlen gerundet aus. Der zugrundeliegende Wert der gerundet angezeigten Zahl ändert sich dadurch nicht.
- Zahlen, die nicht in einen Platzhalter passen (z.B. vierstellige Zahlen in einem dreistelligen Platzhalter), werden nicht angezeigt. Stattdessen erscheint dort der Platzhalter selbst als Fehlerhinweis. Das betrifft auch Zahlen, die wegen eines Übertrags beim Runden nicht mehr in den Platzhalter passen, z.B. bei der Zahl 99.999 formatiert auf zwei Vor- und Nachkommastellen (was eigentlich 100.00 ergäbe).

- Die Ausgabe von USE kann auch auf einen Drucker umgelenkt werden. Dazu muss man vorher auf übliche Weise den Drucker öffnen (`OPEN 1,4`) und hinter USE muss der Ausgabekanal angegeben werden (`USE #1,<string>,<zahlen>`).

Beachten: Da „.“ und „,“ Dezimaltrennzeichen darstellen, können sie im Formatstring nicht einzeln stehen, z.B. als Satzzeichen oder Abkürzungszeichen (führt zu einem **BAD MODE ERROR**). Der „*“ darf ebenfalls nicht ohne direkt folgende Platzhalter verwendet werden, sonst gibt es auch hier einen **BAD MODE ERROR**.

Hinweis: Mit `POKE $A4C5,$30` vor dem USE-Befehl kann man Zahlen mit führenden Nullen erzeugen. Der Standardwert von $A4C5 lautet $20 (Adresse bis v2.31.113: $A607) .

Beispiel:

```
100 f$="nr ##: Euro -###,##"
110 FOR i=1 TO 10
120 z=(RND(1)*201)-50: vg$=STR$(z)
130 USE f$, i, z;: PRINT " :" vg$
140 NEXT

(Das Programm gibt 10 nummerierte Geldbeträge aus, negative Beträge erhalten
ein vorangestelltes Minus. Der Vergleichsstring VG$ zeigt die unformatierte
Version der Zahl.)

Ein komplettes Anwendungsbeispiel beim Befehl MOB SET (Beispiel 2).
```

Befehl:	VOL	
Syntax:	VOL <ld> VOL ON | OFF	
Zweck:	Festlegen von Lautstärke und Filtertypen	Sound
Kürzel	v0	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

VOL setzt den Lautstärkewert **<ld>** (0..15) in das SID-Register $D418.

Beachten: Der Interpreter lässt bei VOL auch Werte zu, die 15 übersteigen. Dies eröffnet die Möglichkeit, auch unter TSB mit den SID-Filtern zu arbeiten und die Stimme 3 zur Klangbeeinflussung heranzuziehen (wofür es in beiden Fällen aber keine ausdrücklichen Befehlswörter gibt).

Folgende Werte (zur Lautstärke dazu addiert) bewirken:

```
16 (Bit 4) : Tiefpassfilter ein
32 (Bit 5) : Bandpassfilter ein
64 (Bit 6) : Hochpassfilter ein
128(Bit 7) : Schaltet Stimme 3 unhörbar (s. dazu MUSIC, Unterpunkt 3).
```

Bei Addition der Werte können mehrere Aktionen gleichzeitig geschaltet werden.

Jedoch die SID-Register

- $D415/16 (Filterfrequenz lo/hi, in TSB mit SOUND 4),
- $D417 (Bit 0..3: Filteraktivierung, Bit 4..7: Resonanzfrequenz definieren),
- $D418 (Bit 4..6: Filtertyp),
- $D41B (Rauschgenerator Stimme 3, Einschalten mit WAVE) und
- $D41C (Hüllkurve von Stimme 3)

werden unter Simons' Basic nicht ausdrücklich unterstützt (in TSB nur Filterfrequenz mit SOUND 4).

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**.

Ein Anwendungsbeispiel beim Befehl MOB SET (Beispiel 2).

Befehl:	WAVE	
Syntax:	**WAVE <stimme>, <bitmuster> [, <pulsbreite>]**	
Zweck:	Festlegen einer Wellenform für eine der drei Stimmen	Sound
Kürzel	wA	
Status:	Erweitertes Simons‘ Basic (Anweisung)	

WAVE legt die Wellenform der zu spielenden Töne fest und schaltet eine Stimme ein oder aus. Jede Stimme kann direkt angesprochen werden (erster Parameter **<stimme>**, Wert: 1..3), sodass die SID-Register $D404 (Stimme 1), $D40B (Stimme 2) und $D412 (Stimme 3) betroffen sind.

Das **<bitmuster>** stellt Schalter für die oben genannten Register dar. TSB erwartet hier eine binäre Zahlenkonstante mit oder ohne einleitendem %-Präfix oder eine beliebige Byte-Zahlenkonstante oder -Variable (Wert 0..255). In Simons‘ Basic ist ausschließlich eine binäre Zahlenkonstante ohne %-Präfix erlaubt. Eine „1“ im Bitmuster (ein gesetztes Bit) aktiviert folgende SID-Features für die angesprochene Stimme:

```
Bit 7 : Rauschgenerator
Bit 6 : Rechteck
Bit 5 : Sägezahn
Bit 4 : Dreieck
Bit 3 : Testbit
Bit 2 : Ringmodulation
Bit 1 : Sync (Stimme 1 mit 3, 2 mit 1 bzw. 3 mit 2)
Bit 0 : Steuer- oder Gatebit (Stimme ein/aus)
```

Von den Bits 4 bis 7 sollte immer nur eines auf „1“ stehen, weil sich die Werte sonst gegenseitig per AND auslöschen.

Für das Setzen der Pulsbreite, die zum Erzeugen einer klingenden Rechteckschwingung (Bit 6) erforderlich ist, bietet Simons' Basic keinen Befehl, so dass man sich mit zwei POKEs behelfen muss: `POKE SOUND+(stimme-1)*7+2, lobyte: POKE SOUND+(stimme-1)*7+3, hibyte.` Die Pulsbreite ist maximal 12 Bits breit (also lautet der Höchstwert für die Pulsbreite 4095). **In TSB kann man den Wert der Pulsbreite als dritten Parameter an den WAVE-Befehl anhängen.**

Die unteren vier Bits haben folgende Bedeutung:

- **Testbit**: Wenn eine Schwingungsform gleichzeitig mit dem Rauschgenerator läuft (also zwei der vier oberen Bits gleichzeitig aktiviert wurden), kann es vorkommen, dass der Rauschgenerator blockiert. Das Testbit aktiviert ihn auf jeden Fall. Sollte sonst 0 sein.
- **Ringmodulation**: Die Dreiecksschwingung von Stimme 1 wird durch ein Gemisch aus Stimme 1 und 3 ersetzt (auch wenn Stimme 3 stumm ist). Entsprechendes gilt für die anderen Stimmen.
- **Sync**: Vermeidet Schwebungen, wenn zwei Stimmen parallel arbeiten.
- **Steuerbit**: (Gate-Bit) Schaltet eine Stimme ein oder aus. Mit diesem Bit arbeitet PLAY. In Stücken, die PLAY nicht benutzen, macht man hiermit die Töne hörbar.

Wird kein Parameter eingegeben, so erscheint die Fehlermeldung **SYNTAX ERROR**. Eine falsche Stimmennummer (kleiner als 1 oder größer als 3) führt zu einem **BAD MODE ERROR**. Andere Zeichen als „0“ oder „1“ oder zu wenige Zeichen im Bitmuster erzeugen einen **BIN CHAR ERROR**.

Beispiel:

```
100 VOL 15
110 ENVELOPE 1,1,8,10,10
120 WAVE 1, 00100000
130 MUSIC 10, "{clear}1c2{f2}"
140 PLAY 1
150 VOL 0
```

(spielt mit Stimme 1 den Sägezahn-Ton c2 als Viertelnote)

Ein komplettes Anwendungsbeispiel beim Simons-Basic-Befehl MOB SET (Beispiel 2).

Befehl:	X!	
Syntax:	X!	
Zweck:	Bereitstellen von neuen, temporären TSB-Befehlen für spezielle Zwecke	Programmierhilfen
Kürzel	-	
Status:	Neuer TSB-Befehl (Anweisung)	

X! eröffnet dem Programmierer die Möglichkeit, TSB z.B. für ein Spieleprojekt mit speziell für dieses Projekt erstellten Befehlen temporär zu ergänzen. Dafür stehen einige unter bestimmten Umständen ungenutzte Speicherbereiche zur Verfügung. Nach Umfang geordnet sind das diese:

Bereich	Umfang	Bedingungen
$0400 .. $07FF	1024	der MEM-Modus ist aktiviert
$C000 .. $C3F7	1015	der HIRES-Modus wird nicht verwendet, aber Sprites
$C64D .. $C74C	256	die F-Tastenbelegung wird nicht gebraucht (KEY OFF)
$CA00 .. $CAFF	256	normalerweise frei (explizit für X!)
$CB00 .. $CBFF	256	MAP wird nicht verwendet
$0334 .. $03FF	204	normalerweise frei (Sprites 13 bis 15)
$C9CF .. $C9FF	49	LOCAL wird nicht verwendet (die Cartridge-Version von TSB nutzt diesen Bereich für den Befehl MONITOR)

Solange **X!** nicht eingebunden ist, erzeugt der Aufruf von **X!** die Fehlermeldung **NOT YET ACTIVE ERROR**. Wenn der Code an die geplante Stelle geladen wurde, aktiviert man den Befehl mit `D!POKE $897C,<Startadresse-1>`. Wenn der Code an $0400 liegt, also mit `D!POKE $897C,$03FF`. Das liegt daran, dass **X!** nun ein echter, vom Interpreter akzeptierter TSB-Befehl ist. Aus diesem Grund darf der Befehl auch nicht mit `RTS` abgeschlossen sein, sondern muss zurück zum Interpreter führen. Wenn der selbstgeschriebene X!-Befehl Parameter einliest, lautet die Endadresse des Befehls daher $9567, wenn nicht $9564.

Es können gleichzeitig mehrere X!-Befehle integriert werden, wenn das Token um einen weiteren Bezeichner erweitert wird, der die Befehle unterscheidet, wie z.B. bei **X!INST** (ein Befehl, der Strings manipuliert, ohne String-Müll zu erzeugen, er befindet sich – zusammen mit weiteren – auf der TSB-Disk). Bei mehreren X!-Befehlen gleichzeitig muss die X!-Routine das berücksichtigen und diese unterscheiden.

Der Code des X!-Befehls sollte auf der Projekt-Release-Diskette zu finden und im Namen als nachladbare Erweiterung („Extension“, Kürzel „ext.“) gekennzeichnet sein, z.B. „ext.inst“ für X!INST. Eingebunden wird er dann etwa mit folgenden Zeilen am Programmanfang (hier einschließlich einer Sicherheitsabfrage, die die ersten beiden Bytes des Codes überprüft, in diesem Fall mit dem Wert 160):

```
100 if a=0 then a=1: load “ext.inst”,use,0,$ca00
110 if d!peek($ca00)=160 then d!poke $897c,$c9ff:else  ”fail!”: end
…
```

Alle TSB-Befehle in Übersicht

Diese Tabelle enthält alle Befehle in alphabetischer Folge.

@	%	%%	$	$$	ANGL	ARC	AT (AT)
AUTO	BCKGNDS	BFLASH	BLOCK	CALL	CENTER	CGOTO	CHAR
CHECK (CHECK)	CIRCLE	CLS	CMOB	COLD	COLOR	COPY	CSET
D!	D!PEEK	D!POKE	DELAY	DESIGN	DETECT	DIR	DISABLE
DISAPA	DISK	DISPLAY (DISPLAY)	DIV	DO .. DONE	DO NULL	DOWN b/w	DRAW
DRAW TO	DUMP	DUP (DUP)	ELSE	END LOOP	END PROC	ENVELOPE	ERRLN
ERRN	ERROR	EXEC	EXIT	EXOR	FCHR	FCOL	FETCH
FILL	FIND	FLASH	FRAC	GLOBAL	GRAPHICS (GRAPHICS)	HI COL	HIRES
HRDCPY	INKEY	INSERT (INSERT)	INST (INST)	INV	JOY	KEY	KEYGET
LIN (LIN)	LINE	LOCAL	LOOP	LOW COL	MAP	MEM	MEMCLR
MEM-CONT	MEMDEF	MEMLEN	MEMLOAD	MEMOR	MEMPEEK	MEMPOS	MEMREAD
MEMRES-TORE	MEMSAVE	MERGE	MMOB	MOBCOL	MOB ON/OFF	MOB SET	MOD (MOD)
MOVE	MULTI	MUSIC	NO ERROR	NRM	OFF	OLD	ON ERROR
ON KEY	OPTION	OUT	PAGE	PAINT	PAUSE	PENX	PENY
PLACE (PLACE)	PLAY	PLOT	POT	PROC	RCOMP	REC	RENUM-BER
REPEAT	RESET	RESUME (RESUME)	RETRACE (RETRACE)	RIGHT b/w	ROT	RLOCMOB	SCRLD
SCRLD|SV DEF	SCRLD|SV RESTORE	SCRSV	SCRLD|SV DEF	SCRLD|SV RESTORE	SECURE	SOUND (SOUND)	TEST
TEXT	TRACE	UNTIL	USE (USE)	VOL	WAVE	X!	

Alle von TSB beeinflussten Adressen

In dieser Tabelle kann man nachschauen, an welchen C64-Adressen die TSB-Befehle Einfluss genommen haben:

Farben	*Beeinflusst durch*	*C64-Adresse*
Hintergrundfarbe(n)	COLOR ra, **bg** BCKGNDS **bg**, **b1**, **b2**, **b3**	$D020, $D021 $D021, $D022, $D023, $D024
Rahmenfarbe	COLOR **ra** BFLASH sp, **r1**, **r2**	$D020 $D020
Cursor-Farbe	COLOR ,**cs**	$0286
Farbe an Position sp, zl (Text-Screen)	AT(zl,sp)	($D800+40*zl+sp) AND 15
Farbe an Position x, y (Grafik-Screen) Da die Grafikfarben von den Bitmustern in der Bitmap abhängen, ist das Ermitteln der zugehörigen Farben nicht ganz trivial.	PLOT x, y	***Hires (Ergebnis in p):*** 100 k=DIV(x,8)*8+DIV(y,8)*320: pz=y AND 7: ad=$e000+k+pz: px=2^(7-(x AND 7)) 110 p=PEEK($c000+DIV(x,8)+DIV(y,8)*40) 120 b=MEMPEEK(ad) AND px 130 IF b THEN p=DIV(p,16) 140 p= p AND 15 ***Multi (Ergebnis in p):*** 100 p=$c000+DIV(2*x,8)+DIV(y,8)*40 110 k=DIV(2*x,8)*8+DIV(y,8)*320: pz=(y AND 7): ad=$e000+k+pz 120 b=MEMPEEK(ad): x$=%%b: px=2*(x AND 3) + 1: x$="000000"+MID$(x$,px,2): a=NRM(x$) 130 if a=0 THEN p=PEEK($d021) AND 15 140 if a=1 THEN p=DIV(PEEK(p),16) 150 if a=2 THEN p=PEEK(p) AND 15 160 if a=3 THEN p=PEEK($d800+DIV(2*x,8) + DIV(y,8)*40) AND 15

Sound	*Beeinflusst durch*	*C64-Adresse*
Tonhöhe	SOUND sn, **fr** MUSIC l, **string**	$D400/1, $D407/8, $D40E/F, $D415/6
Pulsbreite (Rechteck)	WAVE sn, cn, **pu**	$D402/3, $D409/A, $D410/1
Tonsteuerung	WAVE sn, **cn** PLAY **n**	$D404/0B/12
Hüllkurve	ENVELOPE sn, **a**, **d**, **s**, **r**	$D405/6, $D40C/D, $D413/4
Lautstärke	VOL **x**	$D418 (Bits 0..3)
Filter		(Bits 4..7)
A/D-Wandler	POTX POTY	$D419 $D41A

Sprites	*Beeinflusst durch*	*C64-Adresse*
Sprite-Farben	MOB SET n, bl, **f**, p, m oder MOBCOL n, **f** (die individuellen Farben) und CMOB **f1**, **f2** (Multi) (die allen Sprites gemeinsamen Farben)	$D027..$D02E (gemäß lfd. Nummer **n**) $D025 und $D026
Sprite-Koordinaten	RLOCMOB n, **ax**, **ay**, **ex**, **ey**, sz, sp oder MMOB n, **ex**, **ey**, sz, sp	Sprite 0 x-low: $D000, x-hi: $D010, Bit 0, y: $D001 Sprite 1 x-low: $D002, x-hi: $D010, Bit 1, y: $D003 Sprite 2 x-low: $D004, x-hi: $D010, Bit 2, y: $D005 Sprite 3 x-low: $D006, x-hi: $D010, Bit 3, y: $D007 Sprite 4 x-low: $D008, x-hi: $D010, Bit 4, y: $D009 Sprite 5 x-low: $D00A, x-hi: $D010, Bit 5, y: $D00B Sprite 6 x-low: $D00C, x-hi: $D010, Bit 6, y: $D00D Sprite 7 x-low: $D00E, x-hi: $D010, Bit 7, y: $D00F
Sprite-Modi: *Einschalten* *Aus* *Hires / Multicolor* *X-Vergrößerung* *Y-Vergrößerung* *Priorität gegenüber dem Hintergrund* *Kollision Spr-Spr* Kollision Spr-Char	 MOB SET **n**, bl, f, p, m MOB ON/OFF **n** MOB SET n, bl, f, p, **m** RLOCMOB n, ex, ey, **sz**, sp oder MMOB n, ax, ay, ex, ey, **sz**, sp dito MOB SET n, bl, f, **p**, m DETECT x und CHECK(n1, n2) DETECT x und CHECK(n)	 $D015 (Bit gemäß lfd. Nummer **n**) $D01C (wenn Multi: Bit gemäß lfd. Nummer **n** gesetzt) $D01D (Bit gemäß lfd. Nummer **n**) $D016 (Bit gemäß lfd. Nummer **n**) $D01B (Bit gemäß lfd. Nummer **n**) $D01E (Bits gemäß lfd. Nummern **n**) $D01F (Bit gemäß lfd. Nummer **n**)

Zeichen	*Beeinflusst durch*	*C64-Adresse*
Welches Zeichen an Bildschirmposition zl, sp	AT(zl, sp)	z=PEEK(DISPLAY+40*zl+sp) oder z=PEEK(DISPLAY+40*LIN+POS(0)) (aktuelle Cursorposition)

Alle Befehle

Abkürzungen von Basic-Befehlen

(Wenn keine Abkürzung eingetragen ist, ist eine Abkürzung nicht sinnvoll)

Alphabetisch geordnet

A

V2	ABS	aB	-
V2	AND		geändert gegenüber Basic V2
	ANGL	aN	
	ARC	aR	
V2	ASC	aS	-
	AT(	aT	
V2	ATN		geändert
	AUTO	aU	

B

	BCKGNDS	bC	
	BFLASH	bF	
	BLOCK	bL	

C

	CALL	cA	
	CENTER	cE	
	CGOTO	cG	
	CHAR	cH	
	CHECK	chE	
V2	CHR$	chR	-
	CIRCLE	cI	
V2	CLOSE	clO	-
V2	CLR		geändert
	CLS	cL	
V2	CMD		geändert
	CMOB	cM	
	COLD	coL	
	COLOR	cO	
V2	CONT	coN	-
	COPY	coP	
V2	COS		-
	CSET	cS	

D

	D!		
V2	DATA	dA	-
V2	DEF		geändert

	DELAY	dE	
	DESIGN	deS	
	DETECT	deT	
V2	DIM		geändert
	DIR		
	DISABLE	dI	
	DISAPA	disA	
	DISK		
	DISPLAY	disP	
	DIV		
	DO .. DONE		
	DOWNB	dO	
	DOWNW		
	D!PEEK		
	D!POKE		
	DRAW TO	dR	
	DUMP	dU	
	DUP		

E

	ELSE	eL	
V2	END		geändert
	END LOOP	end L	
	END PROC	enD	
	ENVELOPE	eN	
	ERROR		
	EXEC	eX	
	EXIT	exI	
	EXOR	exO	
V2	EXP		-

F

	FCHR	fC	
	FCOL	fcO	
	FETCH	fE	
	FILL	fI	
	FIND	fiN	
	FLASH	fL	
V2	FN		-
	FRAC	fR	
V2	FRE		geändert

G

V2	GET	gE	-
	GLOBAL	gL	
V2	GO		-

V2	GOSUB	goS	-
V2	GOTO	gO	-
	GRAPHICS	gR	

H

	HI COL	hi C	
	HIRES	hI	
	HRDCPY	hR	

I

V2	IF		-
	INKEY	iN	
V2	INPUT		-
V2	INPUT#	inP	
	INSERT	insE	
	INST	inS	
V2	INT		geändert
	INV	iN	

J

	JOY	jO	

K

	KEY	kE	
	KEYGET		

L

V2	LEFT$		geändert
	LEFTB		
	LEFTW	lE	
V2	LEN		-
V2	LET		geändert
	LIN		
	LINE	lI	
V2	LIST	liS	geändert
V2	LOAD	loA	geändert
	LOCAL	loC	
V2	LOG		-
	LOOP	lO	
	LOW COL	loW	

M

	MAP	mA	
	MEM		
	MEMCLR		

	MEMCONT		
	MEMDEF		
	MEMLEN		
	MEMLOAD		
	MEMOR		
	MEMPEEK		
	MEMPOS		
	MEMREAD		
	MEMRESTORE	memresT	
	MEMSAVE	memsA	
	MERGE	mE	
V2	MID$	mI	-
	MMOB	mM	
	MOB ON/OFF		
	MOB SET	moB	
	MOD		
	MOVE	mO	
	MULTI	mU	
	MUSIC	muS	

N

V2	NEW		-
V2	NEXT	nE	-
	NO ERROR	nO	
V2	NOT		geändert
	NRM	nR	

O

	OFF	oF	
	OLD	oL	
V2	ON		-
	ON ERROR	on E	
	ON KEY	oN	
V2	OPEN	opE	geändert
	OPTION	oP	
V2	OR		-
	OUT	oU	

P

	PAGE	pA	
	PAINT	paI	
	PAUSE	paU	
V2	PEEK	peE	geändert
	PENX	pE	
	PENY		
	PLACE	plA	

	PLAY		
	PLOT	pL	
V2	POKE	poK	geändert
V2	POS		-
	POT	pO	
V2	PRINT	?	-
V2	PRINT#	prinT	geändert
	PROC	pR	

R

	RCOMP	rC	
V2	READ	reA	geändert
	REC	rE	
V2	REM		-
	RENUMBER	reN	
	REPEAT	reP	
	RESET	reS	
V2	RESTORE	resT	geändert
	RESUME	resU	
	RETRACE	reT	
V2	RETURN	retU	geändert
V2	RIGHT$		geändert
	RIGHTB RIGHTW	rI	
	RLOCMOB	rL	
V2	RND	rN	-
	ROT	rO	
V2	RUN	rU	-

S

V2	SAVE	sA	-
	SCRLD SCRLD|SV DEF	scrL	
	SCRSV	sC	
	SECURE	sE	
V2	SGN	sG	-
V2	SIN	sI	-
	SOUND	sO	
V2	SPC(	sP	-
V2	SQR	sQ	-
V2	STEP	stE	-
V2	STOP	sT	-
V2	STR$	stR	-
V2	SYS	sY	-

T

V2	TAB(	tA	-
V2	TAN		-
	TEST	tE	
	TEXT	teX	
V2	THEN	tH	-
V2	TO		-
	TRACE	tR	

U

	UNTIL	uN	
	UPB	uP	
	UPW		
	USE	uS	
V2	USR		geändert

V

V2	VAL	vA	-
V2	VERIFY	vE	-
	VOL	v0	

W

V2	WAIT	waI	geändert
	WAVE	wA	

X

	X!		

Ein Reim auf TSB

Seit ich den 64 hab
Da lacht sich meine Freundin schlapp
Das Basic kommt nicht um die Ecke
Sie nennt mich nur noch "Häuptling Schnecke"
Doch heut' mach' ich den Rechner flott
Und nehm das BASIC von GoDot!

tada *tada* *tada*

Die Bootdisk hab ich flugs im Schacht
Mal sehen, wer als letzter lacht!
Ich lade die Objektdatei
(Die Freundin kichert noch dabei.)
Dann starte ich es sehr sensibel
Ach du Schreck - INKOMPATIBEL!

tada *tada* *tada*

Ich muss das nur schnell umkopieren
(Ja, ja, tu' dich nur amüsieren...)
Und diesmal passt's, es meldet sich
Der Startbildschirm, mit grauem Stich
Das BASIC hat jetzt neue Kraft
Mein erster Schritt zur Weltherrschaft!

tada *tada* *tada*

Musik spiel' ich nach Noten nun
Ich bin kein DATA-Zeilen-Huhn!
Die Sprite-Befehle sind so leicht
Die Freundin ist vor Neid erbleicht
"Wie fängst du das denn bitte an?"
Ach, weißt du, Schatz: Wer kann, der kann!

tada *tada* *tada*

Ich stell' noch schnell die Größe ein
Doch das geht mit dem Teufel ein!
Die Sprites sind gleichermaßen groß?
(Die Freundin guckt schon kurios.)
Sie darf nicht merken, dass ich log!
Doch dann die Rettung: Ein kleiner POKE!

tada *tada* *tada*

Von Forum64-User goloMAK

TSB by Arndt Dettke 2024
Released 1986, permanently updated
Text first published on C64-Wiki
Revised for print