BEDIENUNGSANLEITUNG FOR XBC S 1.ØX

XBC:
Extended BASIC compiler, standard version, release 1.ØX

Includes compiler, editor and runtime system.

Written by

Bryan Hayes

distributed by

BBG Software
Beimoorweg 2-4
2Ø7Ø Ahrensburg
West Germany
Tel. Ø41Ø2/43948

Da XBC auf verschiedenen Rechnern läuft, besteht die Anleitung aus einem allgemeinem Teil und aus einem systemspezifischem Anhang. In diesem system-spezifischen Teil steht alles, was nur den jeweiligen Computer bestrifft.

Die allgemeine Anleitung gilt für alle XBC S 1.ØX (X=Ø, 1, 2, ... Z) Versionen. Letzte Überarbeitung der allgemeinen Anleitung am 4.1.1987.

ALLGEMEINE ANLEITUNG FOR XBC S 1.ØX

INHALTSVERZEICHNIS

Ø ALLGEMEINES

Ø.1 Urheberrecht

XBC S 1.ØX und diese Bedienungsanleitung sind urheberrechtlich geschützt. Kein Teil daraus darf ohne schriftliche Genehmigung des Urheberrechtsinhabers in irgendeiner Weise kopiert oder an Dritte weitergegeben werden.

Kompilierte XBC-BASIC-Programme, die mit dem Befehl "JS" inkl. des Laufzeitsystems (Befehl "JSR") abgespeichert werden, dürfen ohne Einschränkung kopiert und vertrieben werden.

Urheberrechtsinhaber von XBC S 1.ØX und dieser Bedienungsanleitung ist (1.1.1986)

Bryan Hayes

Die Rechte für den weltweiten Vertrieb hat das Software-Unternehmen

BBG Software
Beimoorweg 2-4
D-2Ø7Ø Ahrensburg
Tel. Ø4182/4394Ø

Ø.2 Gewährleistung

XBC ist ein sehr komplexes Programm (etwa 17ØØØ-18ØØØ Zeilen), so daß ein absolut einwandfreies Funktionieren nicht garantiert werden kann. Für irgendwelche Schäden, die durch eventuell vorhandene Fehler des Compilers versursacht wurden, wird keine Haftung übernommen.

Falls Sie einen Fehler entdeckt haben sollten, melden Sie dies bitte an eine der oben genannten Adressen (möglichst schriftlich).

Außerdem Irrtum und Änderung vorbehalten.

Ø.3 Kurzbeschreibung

XBC ist ein sehr schneller, leistungsfähiger, komfortabler und professioneller BASIC-Compiler mit einigen sinnvollen Erweiterungen.

XBC besteht aus einem Editor-/Compilergespann inkl. Laufzeitsystem, so daß man Programme sehr schnell schreiben und testen kann.

Es sind nicht nur der vollständige Standard-BASIC-Befehlssatz (inkl. Realzahlen und Fehlerbehandlung) und die üblichen Erweiterungen für die Ansteuerung von Grafik und Ton implementiert, sondern auch eine Reihe von Erweiterungen und Verbesserungen: z.B.:

- Labels statt Zeilennummern ("GOSUB gzykord" statt "GOSUB 1948")
- Schleifen: REPEAT ... UNTIL x, WHILE x ... WEND, LOOP ... END LOOP, DO n ... END DO

- Block-IF x-(ELSE IF x)-(ELSE)-END IF (Läßt sich ohne weiteres als Ersatz für den Pascal-Befehl CASE verwenden)
- Zusätzliche Befehle und Funktionen (z.B. EDIT und KEYP#)
- Datentypen Byte, Cardinalzahlen (Ø-65535), Integerzahlen, Realzahlen, Strings
- Beliebig lange Variablennamen in Groß- und Kleinschreibung

Ø.4 Lieferumfang

Zum Lieferumfang von XBC gehört ein Speichermedium mit XBC 1.ØX und diese Bedienungsanleitung.

Ø.5 Logische/Physikalische Geräte

XBC kennt eine Reihe von logischen und physikalischen Geräten. Ein logisches Gerät ist ein Programmteil von XBC, der für bestimmte Dinge verantwortlich ist, ein physikalisches Gerät dagegen ist tatsächlich vorhanden (z.B. FD oder CMT). Bei jeder Fehlermeldung wird das entsprechende Gerät mit angegeben. Die Nummer des Gerätes kann mit der Funktion ERRDEV# von einem Programm abgefragt werden.

Nr	dv	Abk.	Name	
1		ED	Editor	Editor
2		COM	Compiler	
3		LPT	Line printer (Centronics)	Drucker
4		RS	Serial interface	Serielle Schnittstelle
5	c	CMT	Cassette with magnetic tape	
6	f1	FD1	Floppy disk 1	
7	f2	FD2	Floppy disk 2	
8	f3	FD3	Floppy disk 3	
9	f4	FD4	Floppy disk 4	
1Ø	ct1	CT1	Cartridge 1	
11	ct2	CT2	Cartridge 2	
12	ct3	CT3	Cartridge 3	
13	ct4	CT4	Cartridge 4	
14	q	QD	Quisk disk	
15	r	RD	Ram disk	
16	h	HD	Hard disk	
17	m	MDL	Module	
18		BCP	Basic control program	Programm für Speicherverw. etc.
19		ML	Machine language	Kompiliertes Maschinenprogramm
2Ø		RR	Runtime routines	Laufzeitsystem/routinen

0.6 Fehlermeldungen

Bei jeder Fehlermeldung wird das entsprechende Gerät mit angegeben (siehe "ALLGEMEINES: Logische/Physikalische Geräte"). Die Fehlernummer kann von einem Programm mit der Funktion ERRN# abgefragt werden.

Nr	Meldung	Ausfürliche Form	Bedeutung
0	BRK	Break	Programm wurde mit <BREAK> abgebrochen; dies ist kein Fehler im üblichem Sinn; siehe hierzu ON ERROR/BREAK ...
1	S	Syntax	Der Befehl existiert nicht (in dieser Form) oder eine Eingabe ist grundsätzlich falsch (z.B. 3E% statt 3E5 (Realzahl))
2	V	Value	Der Wert einer Zahl liegt im falschen Bereich (z.B. die 0 bei LN!(0))
3	SYMNE	Symbol not existing	Das Symbol (Label/Array) ist undefiniert
4	LNE	Line not existing	Die Zeile existiert nicht
5	MCAP	Memory capacity	Zu wenig Speicherplatz vorhanden
6	MPRO	Memory protection	Der Speicherbereich ist schreibgeschützt
7	STRL	String length	Der Text ist entweder zu kurz/zu lang
8	SYMSTCAP	Symbol stack capacity	Der Speicher für die Symboltabelle ist voll; siehe Kapitel "DER COMPILER"
9	LD	Load	Beim Laden ist ein Fehler aufgetreten
10	SV	Save	Beim Abspeichern ist ein Fehler aufgetr.
11	VER	Verify	Die Datei wurde fehlerhaft aufgezeichnet
12	FTYP	Filetype	Die Datei ist nicht vom richtigen Typ
13	FNE	File not existing	Die Datei existiert nicht
14	FAE	File already existing	Der Dateiname existiert bereits
15	NSP	No space	Auf dem Gerät ist zu wenig Speicherplatz
16	NE	Not existing	Das Gerät ist nicht vorhanden
17	NFRM	Not formated	Die Diskette ist nicht formatiert
18	WPRO	Write protection	Das Gerät ist gegen Schreibzugriffe geschützt
19	TMF	To many files	Zu viele Dateien auf dem Gerät
20	DVNE	Device not existing	Das Gerät ist prinzipiell unbekannt
21	CHK	Checksum	Es ist ein Prüfsummenfehler aufgetreten
22	HRD	Hardware	Das Gerät ist nicht in Ordnung
23	NRED	Not ready	Das Gerät ist nicht einsatzbereit
24	MOD	Mode not existing	Der Modus (z.B. 80 Zeichen/Zeile) ist nicht möglich
25	STO	Stack overflow	Der Stack ist übergelaufen (z.B. wegen zu vieler Klammerebenen); extrem schwerwiegender Fehler!!
26	CMSQ	Command sequence	Die Schleifenbefehle sind in falscher Reihenfolge (z.B. LOOP:NEXT)
27	STYP	Symbol type	Eine Variable wurde mit einem Label verwechselt bzw. umgekehrt
28	O	Overflow	Ein Über- oder Unterlauf ist aufgetreten (z.B. bei $200+$100 oder 5-7)
29	SYMDD	Symbol double defined	Ein Symbol (Array/Label) ist doppelt definiert
30	NES	Nesting	Schleifen sind unvollständig verschachtelt (z.B. nur NEXT)
31	RD	Read	Beim READ-Befehl sind keine Daten mehr da
32	TYPM	Type mismatch	Falscher Datentyp (z.B. a$=5)
33	A	Array index	Ein Arrayelement wurde falsch indiziert (z.B. DIM a((9)):b=a((10)))
34	NPRNT	Not printable	Kann eigentlich nicht auftreten

0.7 Grundsätzliche Eigenschaften

Es steht ein Zeileneditor zu Verfügung, der maximal 255 Zeichen hereinlesen kann.

Alle Listvorgänge lassen sich mit <SPACE> anhalten und wieder starten, mit <BREAK> dagegen abbrechen.

Mit <BREAK> ist das Drücken der Tasten gemeint, die auf dem Computer im Allgemeinen für den Abbruch verantwortlich sind.
Mit <SPACE> ist das Drücken der Leertaste gemeint.
Mit <CR> ist das Drücken der CR/RETURN/ENTER-Taste gemeint.

Befehle können groß oder klein geschrieben werden.

Eckige Klammern werden im Rahmen dieser Anleitung durch zwei Spitze Klammern ausgedrückt: << Text in eckigen Klammern >>.

Bei eventuell etwas länger andauernden Operationen gibt XBC eine Meldung über die gerade laufende Operation in eckigen Klammern aus, z.B. "<< Trans. lines >>", "<< Search >>" etc. .

Bei allen Daten, die bei den Befehlsbeschreibungen angegeben werden, bedeuten spitze Klammern (<>), daß diese Daten optional sind, d.h. sie können angegeben werden, müssen aber nicht.

Wenn der Editor/Compiler auf einen Befehl wartet, wird dies durch ein ")"-Zeichen angezeigt.

Bei Beispielen sind Eingaben unterstrichen.

0.8 Grundsätzliche Speicheraufteilung

Im Speicher folgen <Quelltext>, <kompiliertes Maschinenprogramm>, freier Speicher, <Variablenspeicher>, <Symbolstacks>, <Compiler/Editor>, Runtimeroutinen aufeinander. Quelltext, Symbolstacks und Compiler/Editor sind nicht vorhanden, wenn man das Programm mit "JS" kompiliert ("JS" kann ohne "JSR" nicht angewandt werden) und dann als eigenständiges Maschinenprogramm in den Speicher geladen hat.
Der freie Speicher kann mit Hilfe der Systemfunktionen MAXADR und SIZE verwaltet werden.

1 DATEN

hh, hl, ...
1-2stellige Hexadezimalzahl von Ø-FF: 7, 1E

hhhh, hhhl, ...
1-4stellige Hexadezimalzahl von Ø-FFFF: D, 18C, FØØ1

dddd, dddl, ...
1-4stellige Dezimalzahl von 1-9999

symbol
Symbol (Variable oder Label). Eine Folge von Groß-/Kleinbuchstaben, Ziffern und Unterstreichungsstrichen; das erste Zeichen darf keine Ziffer sein. Symbole sind auf alle Stellen signifikant, d.h. das Symbol symbolØ123456789 wird vom Symbol symbolØ12345678a unterschieden. Groß- und Kleinbuchstaben werden unterschieden (a<>A): primenumber, Errhandling, d99, T, SYS_adr

lnr, lnrl, ... ;Line number
Zeilennummer, welche sich folgendermaßen darstellen läßt:

1. als dddd
2. als ';' oder '/'. Beides sind Pseudovariablen: ';' enthält die aktuelle, '/' die letzte Zeilennummer
3. als ';' oder '/' +/- dddd. Zum Wert der Pseudovariablen wird dddd addiert oder subtrahiert: ;-1 , /-9 , ;+2
4. als symbol. Der Computer sucht nach der Programmzeile, in der das Label vorkommt, und ersetzt symbol durch die entsprechende Zeilennummer. symbol ist in diesem Fall also auch eine Pseudovariable, allerdings besonderer Art.
5. als symbol +/- dddd. Siehe 4. . Zum Wert der Pseudovariblen wird dddd addiert oder subtrahiert.

Beispiel: Wenn man das Programm

```
1 FOR prim=5 TO 1000 STEP 2
2   f#=TRUE#
3   FOR t=3 TO SQRT!(prim+1) STEP 2
4     IF prim/t*t=prim THEN f#=FALSE#:GOTO e
5   NEXT
6 e::
7   IF f# THEN PRINT prim,
8 NEXT
```

hat und gerade die Zeile 2 mit "12 <CR>" gelistet hat, so daß die Zeile 3 jetzt die aktuelle Zeile ist, dann listen die folgenden Listbefehle die angegebenen Zeilenbereiche:

1	1-8
1 /	8
1 ;	3
1 e	6
1 3,7 oder 1 3 7	3-7
1 ; /	3-8
1 2/	2-8
1 e-2 /-1	4-7
1 ;+1 5	4-5

txt
Eine Folge von ASCII-Zeichen, eingeschlossen in '"': "text"

lbl
Label. Muß an erster Stelle einer Zeile stehen:
9 eofnxt::
Das Label eofnxt markiert die Zeile 9.

var#
Variable vom Typ Byte (Ø-255 bzw. TRUE#/FALSE#). Bytevariablen werden mit "#" gekennzeichnet: bv#, arx#, bytevar#.

var
Variable vom Typ Cardinal (Ø ... 65535). Cardinalvariablen werden nicht gekennzeichnet: hfk, mz, file_number

var%
Variable vom Typ Integer (-32768 ... +32767). Integervariablen werden mit "%" gekennzeichnet: fjg%, quark%, MT%

var!
Variable vom Typ Real (+/- 2.938736E-39 - 3.402823E+38). Realvariablen werden mit "!" gekennzeichnet: rv!, Ffjif!, _z!

var$
Variable vom Typ String (Länge Ø-255). Stringvariablen werden mit "$" gekennzeichnet: H$, first_file_name$

varn
Variable vom Typ Byte, Cardinal, Integer oder Real.

varx
Variable beliebigen Typs (var#, var, var%, var!, var$).

exp#
Arithmetischer Ausdruck, der nach der Berechnung in den Datentyp Byte umgewandelt wird.

exp
Arithmetischer Ausdruck, der nach der Berechnung in den Datentyp Cardinal umgewandelt wird.

exp%
Arithmetischer Ausdruck, der nach der Berechnung in den Datentyp Integer umgewandelt wird.

exp!
Arithmetischer Ausdruck, der nach der Berechnung in den Datentyp Real umgewandelt wird.

exp$
Arithmetischer Ausdruck, der nach der Berechnung in den Datentyp String umgewandelt wird.

expn
Arithmetischer Ausdruck, der als Ergebnis den Datentyp Byte, Cardinal, Integer oder Real hat.

x, x1, ...
Wird bei Gebrauch definiert.

Der XBC-Compiler (nicht Editor) unterstützt fünf verschiedene Datentypen:
- Bytes. Diese beanspruchen 1 Byte im Speicher und haben einen Wert von #Ø ... #255 (#$Ø ... #$FF). Dieser Wert kann entweder dezimal (z.B. '#5Ø'), hexadezimal (z.B. '#$A7') oder als ASCII dargestellt werden (z.B. '#"M"'). Byte-Variablen werden mit einem '#' gekennzeichnet (z.B. 'a#'). Bytes werden außerdem benutzt, um boolsche Werte darzustellen (wahr/falsch). Hierbei wird jeder Wert ungleich #Ø als wahr (TRUE#) aufgefaßt, der Wert #Ø als falsch (FALSE#). Z.B. liefert der Vergleich 4<5 das Byte #255 (#$FF), den Wert für wahr; der Vergleich 4=3 dagegen liefert das Byte #Ø, den Wert für falsch.
- Cardinalzahlen. Diese beanspruchen 2 Bytes im Speicher und haben einen Wert von Ø ... 65535 ($Ø ... $FFFF). Dieser Wert kann entweder dezimal (z.B. '1Ø1') oder hexadezimal dargestellt werden (z.B. $CØØØ). Cardinalvariablen werden nicht gekennzeichnet.
- Integer-Zahlen. Diese beanspruchen 2 Bytes im Speicher und haben einen Wert von -32768 ... +32767. Dieser Wert kann entweder dezimal (z.B. '%999', '%-1Ø'(!)) oder hexadezimal dargestellt werden (z.B. '%$5F', '%$-3ØØØ'(!)). Integer-Variablen werden mit '%' gekennzeichnet (z.B. 'XXORD%').
- Realzahlen. Diese beanspruchen 4 Bytes im Speicher und haben einen Wert von +/- 2.938736E-39 ... 3.4Ø2823E+38 (7-stellig). Dieser Wert kann nur dezimal dargestellt werden. Hierbei muß unbedingt entweder ein Punkt oder ein Exponent vorhanden sein, da der Compiler die Zahl sonst nicht als Realzahl erkennt. '1ØØØØØ' wäre z.B. eine Cardinalzahl, die gar nicht dargestellt werden kann und somit zu einer Fehlermeldung führt. Es muß hier entweder '1ØØØØØ.' oder '1E5' heißen. Real-Variablen werden durch ein '!' gekennzeichnet (z.B. 'rv!').
- Strings. Ein Standardstring belegt 1+8Ø=81 Bytes im Speicher (in dem einem zusätzlichem Byte ist die Länge gespeichert). Die maximale Länge beträgt bei einem Standardstring 8Ø Zeichen. Siehe auch das Kapitel "DER COMPILER: Compileroptionen". Strings werden wie üblich in Anführungszeichen dargestellt. Strings-Variablen werden durch '$' gekennzeichnet (z.B. 'a$').

Arithmetische Operationen:

Bytes	+, -, (,), =, <>, (=,)=, AND, OR, XOR, NOT
Cardinal	+, -, *, /, (,), =, <>, (=,)=, AND, OR, XOR, NOT
Integer	+, -, *, /, Negation, (,), =, <>, (=,)=
Real	+, -, *, /, ^, Negation, (,), =, <>, (=,)=
Strings	+, (,), =, <>, (=,)=

Es gelten die üblichen Prioritäten:

1. Funktionen, Klammern, Zahlen, Variablen, Arrayelemente
2. ^
3. *, /
4. Negation
5. +, -
6. NOT
7. (,), =, <>, (=,)=

8. AND
9. OR
10. XOR

Wenn bei einer arithmetischen Operation zwei Zahlen verschiedenen Typs benutzt werden, wird die Zahl mit dem geringerwertigerem Typ in die entsprechende Zahl des anderen (höherwertigeren) Typs umgewandelt. Die Typen haben hierbei folgende Priorität (Wertigkeit):

1. Real
2. Integer
3. Cardinalzahlen
4. Bytes

Bei der Operation '#5+2' wird '#5' in die Cardinalzahl '5' umgewandelt.
Bei der Operation '3.*%5' wird '%5' in die Realzahl '5.' umgewandelt.
etc.
Für Strings gilt folgendes: Strings werden nicht umgewandelt. Falls Bytes und Strings addiert werden, wird das Byte in ein String der Länge 1 gewandelt ("X"+#65+#66 => "X"+"A"+#66 => "XA"+#66 => "XA"+"B" => "XAB").
Im Gegensatz zu Funktionen können Rechenoperatoren keine Zahlentypen anfordern (wenn z.B. Real angefordert wird, wird der Ausdruck ausgerechnet und, falls das Ergebnis nicht vom Typ Real ist, das Ergebnis in Real umgewandelt (konvertiert)), so daß "2^a" Type mismatch error ergibt (es müßte '2.^a' heißen.
Grundsätzlich ist es besser, auf diese automatische Typenkonversion zu verzichten und gleich die richtigen Typen zu benutzen, da dies Zeit und Platz spart. Dies gilt allerdings nur beschränkt für Variablen. Hier ist es meist sinnvoller, die einfachen Typen zu benutzen, wo es nur geht, da man dann wahrscheinlich mehr Platz und Zeit einspart als bei gelegentlichen Typenkonversionen verlorengeht.

Die Variablen v#, v, v%, v!, v$, v#((...)), v((...)), v%((...)), v!((...)), v$((...)) und das Label v werden unterschieden. Ein Programm wie

```
1 DIM a((9))
2 FOR a=Ø TO 9
3   a#=a*a:a((a))=a+1
4   IF KEYP# THEN GOTO a
5 NEXT
6 a::
7 END
```

wird also wie

```
1 DIM a((9))
2 FOR b=Ø TO 9
3   c#=b*b:a((b))=b+1
4   IF KEYP# THEN GOTO d
5 NEXT
6 d::
7 END
```

ausgeführt.

2 DER EDITOR

2.Ø Allgemeines

Der Editor ist der Programmteil von XBC, der die Eingabe des Quelltextes möglich macht. Dieser Quelltext wird in kodierter Form ab einer bestimmten Adresse gespeichert (siehe Anhang). Die Zeilen werden von 1-9999 fortlaufend (1, 2, 3, ...) durchnummeriert.

Zeilen, bei denen schon bei der Eingabe ein Fehler entdeckt wurde (z.B. Integerzahl > 32767) werden nicht angenommen, sondern, außer im Direktmodus, zur Korrektur bereitgestellt. Dies geschieht in der Form, daß der Fehler und die fehlerhafte Zeile ausgedruckt werden und der Cursor dann am Ende der Zeile wartet.

Leerzeichen werden, außer sie stehen am Anfang der Zeile, grundsätzlich übersprungen, so daß man sie meist weglassen kann. Da innerhalb von Variablen nicht nach BASIC-Wörtern gesucht wird, muß man zwischen einer Cardinalvariablen und einem BASIC-Wort ein Leerzeichen einfügen ("for n=ØtoxstepЗ" wird sonst als "FOR n=Ø TO xstep 3" aufgefaßt, was beim Kompilieren dann als Syntax error gemeldet wird). Beim Listen werden Leerzeichen an den richtigen Stellen automatisch ergänzt. Die Leerzeichen am Anfang einer Zeile werden so ausgegeben, wie sie eingegeben wurden, d.h. Einrückungen, die die Struktur des Programmes anzeigen, sind möglich.

Die benutzte Maschinenroutine, die einen String "hereinliest", ist ein Zeileneditor, d.h.

- der Cursor kann nur dorthin bewegt werden, wo man schon etwas (bei der gerade aktuellen Eingabe) eingegeben hat
- daß nur das hereingelesen wird, was bei der gerade aktuellen Eingabe auch eingegeben wurde (es spielt keine Rolle, was sonst auf dem Bildschirm steht)
- daß die Eingabe auf ein bestimmtes Feld beschränkt ist (beim Editor sind es 255 Zeichen)

Die gleiche Routine wird auch beim INPUT-Befehl benutzt.

Während der Eingabe gibt es folgende Steuerbefehle: (^A=CTRL-A)

- ^M Die Eingabe wird beendet (<CR>)
- ^C Die Eingabe wird abgebrochen (<BREAK>)
- ^S/^H Der Cursor wird eine Position nach links bewegt
- ^D Der Cursor wird eine Position nach rechts bewegt
- ^A Der Cursor wird vier Positionen nach links bewegt
- ^F Der Cursor wird vier Positionen nach rechts bewegt
- ^V Der Einfügemodus wird ein-/ausgeschaltet. Im Einfügemodus werden bei jeder Eingabe alle rechts vom Cursor stehenden Zeichen um eine Position weiter nach rechts geschoben
- ^G Alle rechts von dem Cursor stehenden Zeichen werden um eine Position nach links geschoben; das Zeichen unter dem Cursor wird gelöscht; der Cursor verändert seine Position nicht (Delete)
- ^T Viermal Delete
- ^Q Alle rechts und unter dem Cursor stehenden Zeichen werden um eine Position nach links geschoben; das Zeichen links neben dem Cursor wird gelöscht; der Cursor wird eine Position nach links bewegt (Backspace)
- ^W Viermal Backspace
- ^Z Die ganze Eingabe wird gelöscht

Die Steuerbefehle werden natürlich auch von den Cursortasten etc. ausgelöst (siehe Anhang).

Wenn der Editor/Compiler einen Befehl erwartet, wird dies durch das ")"-Zeichen angezeigt.

2.1 Befehle des Editors

i <lnr> ;Input lines
Ermöglicht die Eingabe von Zeilen ab lnr oder der letzten Zeile. Alle nachfolgenden Zeilen werden neu nummeriert. Dieser Inputmodus wird mit <BREAK> abgebrochen.
Beispiel: Folgende Eingaben werden gemacht:

```
)i <CR>
   1 for n=Ø to 99 <CR>
   2   ? n,n*n <CR>
   3 n. <CR>
   4 <BREAK>
BRK
)i 1 <CR>
   1 ? "Start" <CR>
   2 <BREAK>
BRK
)
```

Jetzt hat man folgendes Programm:

```
   1 PRINT "Start"
   2 FOR n=Ø TO 99
   3   PRINT n,n*n
   4 NEXT
```

l <lnr1> <,lnr2> <:x> ;List
Listet die Zeilen lnr1-lnr2. Die Option ":x" ist nur beim Ausdrucken des Programmes wirksam. x ist der Titel des Programmes. Siehe hierzu das Kapitel "DRUCKERANSTEUERUNG".
Beispiel: siehe Beipiel beim Datentyp lnr.

c <lnr> ;Correct
Ermöglicht die Korrektur von Zeilen ab lnr oder 1. Dieser Korrekturmodus wird mit <BREAK> abgebrochen.
Beispiel:

```
)c 2 <CR>
*   2 FOR n=Ø TO 99* wird jetzt gelistet, der Cursor steht hinter
dem  "99". Man ergänzt jetzt noch "step 2" und dann <CR>. Jetzt
steht folgendes da:
   2 FOR n=Ø TO 99step 2
   3   PRINT n,n*n
```

usw.

d <lnr1> <,lnr2> ;Delete
Löscht nach Bestätigung mit "y <CR>" die Zeilen lnr1-lnr2.

t lnr1,lnr2,lnr3 ;Transfer
Verschiebt die Zeilen lnr1-lnr2 nach lnr3. Es werden keine Zeilen verdoppelt oder gelöscht.
Beispiel: (Das Beispielprogramm bei "i" wird vorrausgesetzt)

```
>t 3,4,1 <CR>
<< Transf.lines >>
>l <CR>
   1   PRINT n,n*n
   2 NEXT
   3 PRINT "Start"
   4 FOR n=0 TO 99
>
```

f <lnr1> <,lnr2> <,> txt ;Find
Durchsucht die Zeilen lnr1-lnr2 nach txt. In txt ist das Zeichen '&', wenn es nicht an erster Stelle steht, ein Dummyzeichen, d.h. es wird an dieser Stelle nicht geprüft. Falls txt gefunden wird, kann die Zeile wie beim C-Befehl korrigiert werden. Nach der Korrektur wird weitergesucht. Die Suche kann mit <BREAK> abgebrochen werden.
Beispiel: Mit 'f "PRI" <CR>' werden die Zeilen

```
1 PRINT "Start"
3   PRINT n,n*n
```

zur Korrektur bereitgestellt.

x x1 dv <x2> ;External memory management
x1=l: ;Load
Bei allen Geräten außer CMT muß x2=<lnr> <,> txt sein, sonst <lnr>. Die Datei wird vom Gerät dv entweder an die letzte Zeile angehängt, oder an lnr geladen.
x1=s: ;Save
x2=<lnr1> <,lnr2> <,> txt: Der Zeilenbereich lnr1-lnr2 wird auf dem Gerät dv unter dem Namen txt abgespeichert.
x1=D: ;Directory
Listet das Directory des Gerätes dv.
x1=F: ;Format
Formatiert nach Bestätigung mit 'Y <CR>' das Gerät dv.
x1=V: ;Verify
Das zuletzt aufgezeichnete Programm wird auf dem Gerät dv auf einwandfreie Aufzeichnung hin untersucht.
x1=K: ;Kill
x2=txt: Löscht das File mit dem Namen txt auf dem Gerät dv.
x1=R: ;Rename
x2=txt1=txt2: Benennt auf dem Gerät dv die Datei txt1 in txt2 um.

Welche Geräte bei welchen Befehlen unterstützt werden, steht im Anhang.
Beispiele:

'x1c <CR>' lädt das nächste Programm von Kassette und hängt es an das vorhandene an.
'x1f2 51 "T1" <CR>' lädt das Programm "T1" von der zweiten

Floppy disk vor die Zeile 51.
'xsq 200/ "ROUT5" <CR>' Speichert alle Zeilen ab 200 unter dem Namen "ROUT5" auf der QD ab.
'xrr "X1"="TP2" <CR>' benennt das Programm "TP2" auf der Ramdisk in "X1" um.
'xkf "TESTPROG" <CR>' löscht das File "TESTPROG" auf der ersten Floppy disk.
'xdq <CR>' listet das Directory der Quickdisk.

q ;Quit
Springt nach Bestätigung mit "y <CR>" ins Rom. Die Warmstartadresse steht im Anhang.

z ;Clear screen
Löscht den Bildschirm (der Übersicht wegen).

v x ;Set videomode
x=4: Der Bildschrim wird falls mögl. auf 40 Zeichen/Zeile eingestellt.
x=8: Der Bildschirm wird falls mögl. auf 80 Zeichen/Zeile eingestellt.

s ;Print status
Listet die Anzahl der freien Bytes für den Quelltext (hexadezimal), die Anzahl der Zeilen/Seite, die Codes für den Seitenvorschub (hexadezimal) und die Symbolstackgröße (hexadezimal).

3 DER COMPILER

3.Ø Allgemeines

Der Compiler übersetzt den BASIC-Quelltext in ein Maschinenprogramm, das dann direkt vom Prozessor ausgeführt werden kann und somit sehr schnell ist. Ein mit XBC kompiliertes Programm ist etwa 1-5ØØ mal schneller als ein gleichartiges, aber von einem BASIC-Interpreter ausgeführtes Programm (je einfacher die Operation ist, desto größer ist der Geschwindigkeitsgewinn); realistisch ist ein Faktor von etwa 4-4Ø.

Da ein Compiler kein Interpreter ist, gibt es auch einige fundamentale Unterschiede, die auf jeden Fall zu beachten sind:

- Beim Compiler wird das Programm erst übersetzt, dann führt es der Prozessor direkt aus. Beim Interpreter wird es vom Interpreter ausgeführt (der Interpreter selbst ist ein Maschinenprogramm und wird vom Prozessor ausgeführt). Ein kompiliertes Programm ist also auch unabhängig vom Compiler lauffähig.
- Beim Compiler sind Editor, Compiler und kompiliertes Maschinenprogramm an sich voneinander unabhängig; daher gibt es auch keine BASIC-Befehle zum editieren des Programms. Es ist auch nicht möglich, ein unterbrochenes Programm wieder zu starten oder vom Editor aus Variableninhalte auszudrucken.
- Beim Übersetzen/Kompilieren geht ein Teil der Struktur des Programms verloren, so daß bei der Fehlerbehandlung gewisse Schwierigkeiten auftauchen.
- Da XBC keine dynamische Variablenverwaltung hat (wie die meisten anderen Compiler), muß der Speicherbedarf von Variablen, insbesondere von Strings und Arrays, vor dem Programmstart feststehen (DIM a(<n>) ist also nicht möglich; anstelle von n muß eine konkrete Zahl angegeben werden, z.B. 49). Strings haben standardmäßig eine maximale Länge von 8Ø, was sich allerdings mit entsprechendn Befehlen ändern läßt.

Beim Kompilieren werden folgende Angaben gamacht:
<< ML:niedrigste Adresse-höchste Adresse, free:freier Speicher >>
Diese Angaben werden hexadezimal gamacht. Der "freie Speicher" kann mit Hilfe der Systemvariablen MAXADR und SIZE verwaltet werden.
Der Programmlauf wird mit der Meldung << Run >> gestartet.
Siehe Anhang für Informationen über Programmabbruch.
Wenn ein Fehler auftritt, wird dies folgendermaßen gemeldet:
Fehlergerät:Fehlertext error at PC=Fehleradresse (hexadezimal)
<< Search.line >>
Zeile, in der der Fehler auftrat

Beim Kompilieren können zwei besondere Fehler auftauchen:

1. "COM:MCAP error". Dies bedeutet, daß nicht genügend Speicherplatz für das kompilierte Programm da ist. Abhilfe kann man durch Benutzen der Befehle "jsr" und "js" schaffen, wodurch die Entwicklungsgeschwindigkeit allerdings erheblich herabgesetzt wird. Eine andere Möglichkeit ist das Verkleinern des Symbol-Stacks mit dem "st"-Befehl (siehe dort).
2. "COM:SYMSTCAP error". Dies bedeutet, daß nicht genügend Platz für die Label und Variablenverwaltung vorhanden ist. Sie müssen entweder weniger Lables und/oder Variablen benutzen oder den Symbolstack vergrößern (siehe Befehl "st").

3.1 Befehle des Compilers

st hhhh ;Set symbol stack size
Definiert die Größe des Symbolstacks. Die aktuelle Größe wird mit "s <CR>" gelistet.

j ;Do job
Kompiliert das Programm und startet es. Dies ist der Standardbefehl zum Starten eines Programmes.

jt ;Job: test
Testet, ob einwandfrei kompiliert wird. Es wird kein Maschinenprogramm erzeugt (und natürlich auch keines gestartet).

jsr dv txt ;Job: save runtime system
Speichert das Laufzeitsystem unter dem Namen txt auf dem Gerät dv ab. Dieser Befehl sollte möglichst nur direkt nach dem Starten des Compilers benutzt werden. Insbesondere darf kein Quelltext eingegeben, keine Funktionstasten anders belegt worden sein, Gerät vorher benutzt worden sein, etc. .

js ;Job: save program
Vor Anwenden dieses Befehls muß der Befehl "jsr" ausgeführt worden sein. Dieser Befehl kompiliert das Programm und speichert es auf dem selben Gerät wie beim jsr-Befehl unter dem gleichem Namen, der allerdings um ein "C" ergänzt wurde.

Anmerkung zu jsr und js:

Diese beiden Befehle werden benutzt, um ein eigenständiges, d.h. vom Compiler und Editor unabhängiges Maschinenprogramm zu erzeugen, das beliebig kopiert und verkauft werden darf. Dieses Programm besteht aus zwei Teilen:

1. Aus dem Laufzeitsystem. In diesem Laufzeitsystem befinden sich z.B. alle Rechen-, Ein/Ausgabe- und Grafikroutinen. Dieses Laufzeitsystem versucht, nach dem Laden das kompilierte Maschinenprogramm zu laden und zu starten. Auf einer Kassette muß sich deshalb das Laufzeitsystem vor dem kompiliertem Maschinenprogramm befinden.
2. Aus dem kompiliertem Maschinenprogramm. Dieses Programm wird vom Laufzeitsystem geladen. Auf einer Kassette muß es sich deshalb hinter dem Laufzeitsystem befinden. Ohne das Laufzeitsystem ist es nicht lauffähig.

Beispiel:

Man hat auf der Kassette CASSOURCE ein XBC-BASIC-Programm namens "G76". Dieses Programm will man jetzt unter dem Namen "GAME" auf der Kassette CASML als eigenständiges Maschinenprogramm abspeichern. Hierbei geht man folgendermaßen vor:

1. XBC-BASIC-Compiler laden.
2. Kassette CASML einlegen (und zurückspulen).
3. 'jsr c "GAME" (CR)' eingeben. Kassette nicht zurückspulen!
4. Kassette CASSOURCE einlegen und XBC-BASIC-Programm "G76" mit 'xlc "G76" (CR)' laden.
5. Kassette CASML einlegen (nicht zurückspulen). Programm "G76" mit 'js (CR)' kompilieren und auf Kassette CASML unter dem Namen 'GAMEC' abspeichern. Kassette zurückspulen.
6. Reset drücken oder Computer aus-ein-schalten. Dann versuchen, das Programm "GAME" zu laden und starten.

Bei einer Diskette geht man analog vor. Allerdings muß man nicht mit zwei Diskette arbeiten.

Bei einem so abgespeichertem Programm (das aus zwei Files besteht), wird man nach Programmende und nach jedem Fehler folgendes abgefragt:
Quit, Run, Cont:(Cursor wartet)
Jetzt kann man 'q', 'r' oder 'c' eingeben. Bei 'q' wird in das Rom gesprungen, bei 'r' wird das Programm neu gestartet. 'c' sollte nur nach einem einem (BREAK) (oder eventuell nach einem Fehler) eingegeben werden, da es sonst zu einem Absturz kommen kann. Das Programm wird dann fortgeführt.

3.2 Compileroptionen

Compileroptionen sind bestimmte Befehle, die im Quelltext stehen und die während des Kompillierens den Compiler veranlassen, etwas Bestimmtes zu tun. Bei Compileroptionen wird kein Maschinenkode erzeugt, sondern sie beeinflussen die Kompilation.

OPTION "x"
Anstelle von x kann folgendes stehen (oder eine Kombination davon):
- 'b+':Ab hier wird hinter jedem Befehl ein RST-Aufruf zu einer Break-test-routine generiert, so daß man das Programm mit (BREAK) anhalten kann (was sonst nicht möglich ist).
- 'b-':Ab hier wird kein RST-Aufruf zur Break-test-routine generiert (das Programm kann also nicht durch (BREAK) abgebrochen werden)
- 'o+':Ab hier wird jeder Über-/Unterlauf etc. bei Arithmetischen Operationen durch eine entsprechende Fehlermeldung angezeigt.
- 'o-':Ab hier werden nur noch bestimmte Fehler abgefangen (alle Real-, Multipliations- und Divisions-fehler und eine Reihe weiterer; nicht abgefangen werden Additions-, Subtraktions- und Zahlentypumwandlungsfehler bei den Datentypen Byte, natürliche Zahlen, Integer)
- 'a+':Ab hier werden alle Fehler bei der Arrayindizierung abgefangen (z.B. bei "DIM a((9)):b=a((20))")
- 'a-':Ab hier werden diese Fehler nicht abgefangen, was zu Programmabstürzen etc. führen kann.

Bei allen diesen Optionen gilt, daß die '-'-Option das Programm kürzer und schneller macht. Bei Verwendung der 'o-' und 'a-' Option sollte man sich unbedingt davon überzeugen, daß keine Über-/Unterläufe und falsche Arrayindizierungen vorkommen, da das Programm sonst nicht einwandfrei funktio-

niert. Die Option 'b-' sollte auch mit Vorsicht angewandt werden, da man sonst das Programm u.U. gar nicht abbrechen kann (wenn die RESET-Taste fehlt); siehe auch BASIC-Befehl 'TEST BREAK'.
Zu Beginn der Kompilation wird intern der Befehl 'OPTION "b+o+a+"' ausgeführt.
Beispiel:

```
1 OPTION "b-o-a-"
2 DIM a((99))
3 FOR n=Ø TO 99
4   a((n))=n+n+n
5   OPTION "b+"
6 NEXT
```

Bei diesem Beispiel treten garantiert keine Über-/Unterläufe und falsche Arrayindizierungen auf. In Zeile 5 bewirkt die Option "b+", daß man dieses Programm in der Schleife auch abbrechen kann (was allerdings angesichts der kurzen Ausführungszeit in diesem Fall ziemlich sinnlos ist; bei Endlos-Scheifen sieht die Sache aber schon ganz anders aus). Statt 'OPTION "b+"' könnte man auch 'TEST BREAK' schreiben. Dies wäre wahrscheinlich besser, da hier nur an einer Stelle auf (BREAK) geprüft wird, während im obigem Beispiel bei jedem nachfolgendem Befehl auf (BREAK) geprüft werden würde.

LENGTH x,var$1 (,var$2) (, ...)
x ist eine Zahl von 1-255. Die Stringvariablen var$1 (var$2, ...) können jetzt maximal x Zeichen enthalten. Ohne Anwendung dieses Befehls ist die maximale Länge für jede Stringvariable 8Ø. Für jede Stringvariable werden (1+maximale Länge) Bytes im Speicher belegt.
Beispiel:
LENGTH 1Ø,i$,c0$,c1$ sorgt dafür, daß die Stringvariablen i$, c0$ und c1$ nur noch 1Ø Zeichen enthalten können (und jeweils nur noch 11 Bytes im Speicher belegen statt 81).

Zusatz beim DIM-Befehl
Hinter einem Stringarray kann eine Zahl stehen, die die maximale Länge der einzelnen Strings angibt.
Beispiel:
DIM TXT$((79)),8,i$((5)),4Ø dimensioniert zwei Stringarrays:
- TXT$ mit 8Ø Strings, mit jeweils nur bis zu 8 Zeichen
- i$ mit 6 Strings, mit jeweils nur bis zu 4Ø Zeichen

Das TXT$-Array belegt 8Ø*9=72Ø Bytes im Speicher statt 8Ø*81=648Ø.
Das i$-Array belegt 6*41=246 Bytes im Speicher statt 6*81=486.

4 DAS BASIC

4.Ø Allgemeines

Grundsätzlich sollte man im Programm alles klein schreiben, denn beim Listen werden alle Schlüsselwörter groß ausgegeben, so daß man auf den ersten Blick sieht, was Schlüsselwörter und was eigene Variablen und Labels sind. Da XBC ein erweiterter BASIC-Compiler ist, sind bestimmte Standard-BASIC-Wörter durch andere ersetzt worden (z.B. "ABS" durch "ABS!"). Wenn man alles klein eingibt, sieht man dann beim Listen sofort, daß "abs" nicht als Schlüsselwort erkannt worden ist.

Befehle und Funktionen können abgekürzt werden. Z.B. kann man "NEXT" durch "n." abkürzen oder "RETURN" durch "ret.". Ein Sonderfall ist "PRINT": hier kann man wie in BASIC allgemein üblich mit "?" abkürzen.

Hier werden nicht alle Befehle und Funktionen aufgeführt; Die systemspezifischen sind im Anhang zu finden.

4.1 Allgemeine Befehle und Funktionen

'x
x ist irgendein Kommentar. Dies ist die einzige Möglichkeit, Kommentare darzustellen; es gibt keinen REM-Befehl.
Beispiel:
```
8 'print all data
```

TEST BREAK
Prüft auf <BREAK>. Dieser Befehl ist nur dann sinnvoll, wenn man mit 'OPTION "b-"' alle überprüfungen auf <BREAK> unterdrückt und jetzt an einer Stelle doch auf <BREAK> testen möchte.

Labels
Labels werden benutzt, um bestimmte Zeilen im Programm zu markieren. Sie können nur am Anfang einer Zeile stehen und haben zwei Doppelpunkte hinter sich:
```
32 start::
```
Jetzt kann die Zeile 32 durch "GOTO start" angesprungen werden.

<LET> varx=expx
Die übliche Variablenzuweisung (z.B. a=a*6).

DIM varx1((x11 <,x12> <...>)) (,varx2((x21 <,x22> <...>))) <...>
Dies ist der bekannte DIM-Befehl (leider etwas unübersichtlich dargestellt). Es müssen immer eckige Klammern benutzt werden. x11, x12 , ... sind einzelne konkrete Cardinalzahlen, keine Variablen oder arithmetische Ausdrücke! Die Anzahl der Indizes (Dimensionen) ist beliebig. Unter "Compileroptionen" ist eine Erweiterung des DIM-Befehls aufgeführt (nur für Stringarrays).
Beispiel:
```
DIM a((99)),cf$((7)),p$((3,5)) dimensioniert
  - Ein Cardinal-Array (a) mit 100 Elementen
  - Ein Byte-Array (cf$) mit 8 Elementen
```



```
  - Ein String-Array (p$) mit 4*6 Elementen
```

CLR
Löscht alle Variableninhalte (Zahlen => Ø, Strings => ""); wird auch bei Programmstart ausgeführt.

READ varx1 <,varx2> <...>
Die bekannte READ-Anweisung.

DATA x1 <,x2> <...>
Die bekannte DATA-Anweisung. Strings müssen unbedingt in Anführungszeichen stehen.

RESTORE lbl
Die bekannte RESTORE-Anweisung (nur mit Label statt Zeilennummer). Der nächste Befehl nach dem benutzten Label muß aber unbedingt ein DATA-Befehl sein. Statt

```
11 dtØ::
12 a=b
13 DATA 4,5,67
14 RESTORE dtØ
```

müßte es

```
11 a=b
12 dtØ::
13 DATA 4,5,67
14 RESTORE dtØ
```

heißen, da sonst beim nächsten READ ein 'Syntax error' ausgegeben werden würde.

RESTORE
Setzt den READ-DATA-Pointer auf das erste DATA-Element im ganzen Programm.

RND!
Entspricht der Funktion RND(1). RND! ist eine Pseudozufallszahl.

RESET RND!
Setzt den Pseudozufallsgenerator auf den Startwert.

RANDOM
Setzt den Pseudozufallsgenerator auf einen mehr oder minder zufälligen Startwert.

LEFT$(exp$,exp#)
Die bekannte LEFT$-Funktion (LEFT$("abcdefgh",3)="abc").

RIGHT$(exp$,exp#)
Die bekannte RIGHT$-Funktion (RIGHT$("abcdefgh",4)="efgh").

MID$(exp$,exp#1 <,exp#2>)
Die bekannte MID$-Funktion (MID$("abcdefgh",5,2)="ef"; MID$("abcdefgh",3)="cdefgh").

POS#(<exp$,> exp$1,exp$2)

Gibt die Position des Strings exp$1 im String exp$2 zurück. Wenn der Substring exp$1 nicht im String exp$2 gefunden wird, wird #0 zurückgegeben. Wenn exp# mit angegeben wird, wird ab der exp#-ten Stelle in exp$2 gesucht.

Beispiel:

```
POS#("abz","4xttabz99")=#4
POS#("abz","4xttaby99")=#0
POS#(#3,"dir","dir dir")=#5
```

CHR$(exp#1 (,exp#2) (...))

Wandelt die Bytes exp#1 (, exp#2, ...) in einen String um.

Beispiel:

```
CHR$(65,66)="AB"
```

ASC#(exp$)

Die bekannte ASC-Funktion (ASC#("BY")=66; ASC#("")=0; ASC#("!")=33).

STRING$(exp#, exp$)

Addiert exp#-mal exp$.

Beispiel:

```
STRING$(3,"TEST")="TESTTESTTEST"
STRING$(5,"fd ")="fd fd fd fd fd "
```

SPC$(exp#)

Gibt exp# Leerzeichen zurück (SPC$(3)=" ").

UP$(exp$)

Wandelt alle Kleinbuchstaben in exp$ in Großbuchstaben.

Beispiel:

```
UPCASE$("Jf8Hld")="JF8HLD"
```

LOW$(exp$)

Wandelt alle Großbuchstaben in exp$ in Kleinbuchstaben.

Beispiel:

```
LOWCASE$("Jf8Hld")="jf8hld"
```

HEX$(exp (,exp#))

Wandelt exp in eine 4-stellige Hexadezimalzahl. Wenn exp# mit angegeben wird, werden die exp# letzten Stellen zurückgegeben. Wenn exp# gleich #0, werden soviel Stellen wie notwendig zurückgegeben, d.h. führende Nullen werden unterdrückt.

Beispiel:

```
HEX$(100)="0064"
HEX$(10000,2)="10"
HEX$(1000,#0)="3E8"
```

BIN$(exp (,exp#))

Wandelt exp in eine 16-stellige Binärzahl. Sonst wie HEX$.

DEC$(exp (,exp#))

Wandelt exp in eine 5-stellige Dezimalzahl. Sonst wie HEX$.

STR$(expn)

Die bekannte STR$-Funktion (STR$(#6)="6"; STR$(4.3-1.1)="3.2").

LEN#(exp$)

Die bekannte LEN-Funktion (LEN#("hf8")=3; LEN#("")=0).

ODD#(expn)

Gibt wahr (#255) zurück, wenn expn ungerade ist, sonst #0 (falsch).

LB#(exp)

Gibt das Lowbyte von exp zurück (die niederen 8 Bits).

Beispiel:

```
LB#($1234)=$34
```

HB#(exp)

Gibt das Highbyte von exp zurück (die höheren 8 Bits).

Beispiel:

```
HB#($1234)=$12
```

INT!(exp!)

Die bekannte INT-Funktion (INT!(5.6)=5.; INT!(-6.7)=-7.).

CUT!(exp!)

Die Nachkommastellen werden entfernt.

ROUND!(exp!)

Führt die 4/5-Rundung durch.

Beispiel:

```
ROUND!(12.3)=12.
ROUND!(12.6)=13.
```

SQR!(exp!)

Liefert die Quadratwurzel (entspricht der bekannten SQR-Funktion).

LN!(exp!)

Liefert den natürlichen Logarithmus.

LOG!(exp!)

Liefert den 10-er Logarithmus.

EXP!(exp!)

Liefert e^exp!.

POW10!(exp!)

Liefert 10^exp!.

SIN!(exp!)

Die bekannte SIN-Funktion (Bogenmaß).

DSIN!(exp!)

Liefert den Sinus von exp! (Gradmaß).

COS!(exp!)

Die bekannte COS-Funktion (Bogenmaß).

DCOS!(exp!)

Liefert den Cosinus von exp! (Gradmaß).

TAN!(exp!)

Die bekannte TAN-Funktion (Bogenmaß).

DTAN!(exp!)

Liefert den Tangens von exp! (Gradmaß).

ATN!(exp!)
Die bekannte ATN-Funktion (Bogenmaß).

DATN!(exp!)
Liefert den Arcustangens von exp! (Gradmaß).

RAD!(exp!)
Rechnet exp! ins Bogenmaß um (*PI!/18Ø.).

DEG!(exp!)
Rechnet exp! ins Gradmaß um (*18Ø./PI!).

VAL#(exp$)
Die bekannte VAL-Funktion, hier für Bytes.

VAL(exp$)
Wie VAL#, aber für Cardinalzahlen.

VAL%(exp$)
Wie VAL#, aber für Integerzahlen.

VAL!(exp$)
Wie VAL#, aber für Realzahlen.

SGN#(exp#)
Die bekannte SGN-Funktion, hier für Bytes.

SGN(exp)
Wie SGN#, aber für Cardinalzahlen.

SGN%(exp%)
Wie SGN#, aber für Integerzahlen.

SGN!(exp!)
Wie SGN#, aber für Realzahlen.

ABS%(exp%)
Die bekannte ABS-Funktion, hier für Integerzahlen.

ABS!(exp!)
Wie SGN%, aber für Realzahlen.

SQR(exp)
Berechnet exp*exp, hier für Cardinalzahlen.

SQR%(exp)
Wie SQR, aber für Integerzahlen.

SQR!(exp)
Wie SQR, aber für Realzahlen.

MULT2#(exp#)
Schnelle Berechnung von exp#*exp#, hier für Bytes.

MULT2(exp)
Wie MULT2#, aber für Cardinalzahlen.

MULT2%(exp%)

Wie MULT2#, ber für Integerzahlen.

MULT2!(exp!)
Wie MULT2#, aber für Realzahlen.

DIV2#(exp#)
Schnelle Berechnung von exp#/2, hier für Bytes.

DIV2(exp)
Wie DIV2#, aber für Cardinalzahlen.

DIV2%(exp%)
Wie DIV2#, aber für Integerzahlen.

DIV2!(exp!)
Wie DIV2#, aber für Realzahlen.

MOD(exp1, exp2)
Berechnet den Divisionsrest von exp1/exp2, hier für Cardinalzahlen.

MOD%(exp%1, exp%2)
Wie MOD, aber für Integerzahlen.

MOD!(exp!1, exp!2)
Wie MOD, aber für Realzahlen.

CONV#(expx)
Wandelt expx in den Datentyp Byte um.

CONV(expx)
Wandelt expx in den Datentyp Cardinal um.

CONV%(expx)
Wandelt expx in den Datentyp Integer um.

CONV!(expx)
Wandelt expx in den Datentyp Real um.

CONV$(expx)
Wandelt expx in den Datentyp String um.

TRUE#
Diese Funktion liefert #255, den Wert für "wahr".

FALSE#
Diese Funktion liefert #Ø, den Wert für "falsch".

PI!
Diese Funktion liefert PI (etwa 3.14159265).

4.2 I/O-Befehle und -Funktionen

FORMAT exp$
Definiert das Ausgabeformat für Zahlen. exp$ enthält genau 4 Zeichen:
1. "#", " ", "%" oder "!". Hier wird festgelegt, welches Ausgabeformat für welchen Zahlentyp definiert werden soll.
2. " " oder "-". Hier wird festgelegt, ob ein Leerzeichen oder gar nichts ("-") vor der Zahl ausgegeben werden soll.
3. " ", "+" oder "-". Hier wird festgelegt, ob bei einer positiven Zahl ein Leerzeichen, ein Pluszeichen oder gar nichts ("-") ausgegeben werden soll.
4. " " oder "-". Hier wird festgelegt, ob ein Leerzeichen oder gar nichts ("-") hinter der Zahl ausgegeben werden soll.

Beispiel:

```
1 FORMAT "!-+ "
2 PRINT ")>)";PI!;"(("
```

würde ")>)+3.141592 ((" ausgeben. Wenn man in Zeile 1 "!---" schreiben würde, bekäme man als Ergebnis ")>)3.141592(("

RESET FORMAT
Führt 'FORMAT "x- -"' für x="#", " ", "%", "!" aus, d.h. bei positiven Zahlen steht ein Leerzeichen vor der Zahl. Dieser Befehl wird automatisch bei Programmstart ausgeführt.

INPUT <x;> ((exp$);) varx1 <,varx2> <...>
Der bekannte INPUT-Befehl (x ist ein String in Anführungzeichen). Mit (exp$) kann die Länge des Eingabefeldes definiert werden (Standardlänge: 255). Bei Byte-, Cardinal- und Integervariablen können Hexadezimalzahlen dadurch eingegeben werden, daß man ein "$" voranstellt. Bei Bytevariablen kann man zudem ASCII-Zeichen durch Voranstellen eines Anführungszeichens eingeben, außdem auch TRUE$ und FALSE$ durch Eingeben von 'true' oder 'false' (Groß-/Kleinschreibung erlaubt; ohne die "'"; muß nicht ausgeschrieben werden).
Beispiel:
INPUT (3);a erlaubt nur 3-stellige Zahlen

EDIT ((exp$);) var$
Erlaubt das Editieren des Strings in var$. Hierbei werden alle Eingegeben Zeichen übernommen, auch Kommata, führende und folgende Leerzeichen. Mit (exp$) kann die Länge des Eingabefeldes definiert werden; ohne (exp$) wird die Länge automatisch auf 255 gesetzt.
Beispiel: Man hat das Programm

```
1 a$="abcd"
2 EDIT (6);a$
3 PRINT ")>)";a$;"(("
```

und startet es:

```
>j <CR>
abcd            erscheint jetzt mit dem Cursor dahinter.
                Wenn Sie versuchen, mehr als 6 Zeichen einzugeben,
                werden diese nicht angenommen. Sie geben jetzt
                folgendes ein:
,cds <CR>
```


)> ,cds ((wird dann ausgedruckt.

GET var$
Der bekannte GET-Befehl. Es muß aber eine Bytevariable angegeben werden und keine Stringvariable. Wenn keine Taste gedrückt ist, wird $0 zurückgegeben, sonst der ASCII-Wert der gerade gedrückten Taste.

GETC var$
Wie GET var$, aber der Cursor wartet auf eine Eingabe.

KEYP$
Diese Funktion gibt TRUE$ ($255) zurück, wenn gerade eine Taste gedrückt wird, sonst FALSE$ ($0).

PRINT (((exp))) <x1> <x2> <...>
x1, x2, ... ist entweder expx, ";", "," oder "TAB(exp$)". Abgesehen davon, daß durch "," der 8-er und nicht der 10-er Tabulator ausgelöst wird, entspricht der PRINT-Befehl dem Standard. Mit "((exp))" kann die Ausgabe auf verschiedene Geräte umgeleitet werden. Jedes der 16 Bits in exp steht für ein bestimmtes Gerät. Wenn das entsprechende Bit gesetzt ist, wird die Ausgabe auf das Gerät geleitet. Auf diese Weise ist es möglich, eine Ausgabe auf mehreren Geräten gleichzeitig auszugeben.

```
Bit  0: Bildschirm
Bit  1: Drucker (LPT) (nur Text, Tabulator, CR)
Bit  2: Drucker direkt (für Steuerzeichen, Grafik)
Bit  3 ... 15: Siehe Anhang
```

Wenn man in exp ein bestimmtes Bit setzten will, muß man 2^Bitnummer dazuaddieren. Bildschirm+Drucker: 2^0+2^1=1+2=3.
Beispiele:
"PRINT ((4));CHR$(#27,#$45);" gibt das Epson-Steuerzeichen für "Fettdruck ein" auf dem Drucker direkt aus.
"PRINT ((3));"Number";a,b" gibt "Number:" und die Variablen a und b auf dem Bildschirm und auf dem Drucker aus.

CURSOR exp$1, exp$2
Setzt den Cursor auf die Position exp$1, exp$2.

CURX$
Diese Funktion liefert die Cursor-X-Position.

CURY$
Diese Funktion liefert die Cursor-Y-Position.

CLS
Löscht den Bildschirm und bewegt den Cursor in die linke, obere Ecke.

HOME
Bewegt den Cursor in die linke, obere Ecke.

BELL
Erzeugt einen kurzen Piepton.

4.3 Programmsteuerbefehle

END
Das Programm wird beendet.

RUN
Das Programm wird von der ersten Zeile nochmal abgearbeitet. Hierbei werden implizit folgende Befehle abgearbeitet:
CLR, RESTORE, RESET SP, RESET FORMAT, ON ERROR ERROR, ON BREAK BREAK.

BOOT
Springt ins ROM (das Programm wird dementsprechend abgebrochen).

WAIT exp
Wartet exp Millisekunden. Dies funktioniert immer richtig, egal ob auf <BREAK> geprüft wird oder nicht.

GOTO lbl
Springt zum Label lbl.

ON exp% GOTO/GTO lbl1 <,lbl2> <, ...>
Der bekannte ON GOTO Befehl. Es ist besser GTO statt GOTO zu schreiben, da sonst kein Leerzeichen vor GOTO beim Listen gedruckt wird.

<GOSUB> lbl
Der bekannte GOSUB-Befehl. GOSUB kann auch weggelassen werden, d.h. die Routine lbl wird einfach durch Nennung des Namens lbl aufgerufen.

RETURN
Kehrt aus einer Routine zurück, die mit GOSUB aufgerufen wurde.

RETURN lbl
Entfernt die Rücksprungadresse auf dem Stack, die von dem GOSUB-Befehl dort abgelegt wurde und springt dann nach lbl. (nur für Profis und solche, die glauben, sie wären welche).

ON exp% GOSUB/GSUB lbl1 <,lbl2> <...>
Der bekannte ON GOSUB Befehl. Siehe GOSUB und ON exp% GOTO ...

LOOP ... END LOOP
Die Befehle zwischen LOOP und END LOOP werden endlos ausgeführt, d.h. beim Erreichen des Befehls END LOOP wird zum Befehl hinter LOOP gesprungen.

REPEAT ... UNTIL exp%
Wiederholt die Befehle zwischen REPEAT und UNTIL solange, bis exp% wahr wird. Die Befehle zwischen REPEAT und UNTIL werden mindenstens einmal ausgeführt.
Beispiel:

```
1 REPEAT
2   INPUT "Number:";a
3 UNTIL a=10
```

Man wird solange aufgefordert, eine Zahl einzugeben, bis man "10" eingibt.

WHILE exp% ... WEND
Wiederholt die Befehle zwischen WHILE und WEND solange, wie exp% wahr ist. Wenn exp% beim ersten Erreichen von WHILE falsch ist, werden die Befehle zwischen WHILE und WEND keinmal ausgeführt.
Beispiel:

```
1 WHILE KEYP%
2   PRINT "Key is pressed"
3 WEND
```

Solange man eine Taste drückt, wird "Key is pressed" ausgegeben.

DO exp ... END DO
Wiederholt die Befehle zwischen DO und END DO exp-mal. Wenn exp gleich Ø ist, werden die Befehle zwischen DO und END do keinmal ausgeführt. Diese Schleife ist schneller und kürzer als die FOR ... NEXT Schleife.
Beispiel:

```
1 DO 7
2   PRINT "??"
3 END DO
```

Es wird 7-mal "??" ausgegeben.

FOR varn=expn1 <DOWN>TO expn2 <STEP expn3> ... NEXT
Die bekannte FOR ... NEXT Schleife. Alle Zahlentypen sind als Laufvariable zugelassen. Wenn man Byte- und Cardinalvariablen abwärts zählen lassen will, muß man statt TO DOWNTO verwenden.
Hinter NEXT darf im Gegensatz zu den meisten Interpretern keine Variable stehen.
Die FOR ... NEXT Schleife erfordert einen relativ hohen Verwaltungsaufwand und ist somit entsprechend langsam und "platzfressend" (aber dafür leistet sie auch entsprechend viel). Die Konstruktion

```
54 n=1
55 DO 1000
56   ...
57   n=n+1
58 END DO
```

ist etwa 1.8 mal schneller als die FOR ... NEXT Schleife

```
54 FOR n=1 TO 1000
55   ...
56 NEXT
```

(erfordert aber etwa gleichviel Platz)
Noch ein Beispiel mit DOWNTO:

```
1 FOR a%=%200 DOWNTO %194 STEP %2
2   PRINT a%
3 NEXT
```

Die Zahlen 200, 198, 196 und 194 werden ausgegeben. Hinter DOWNTO kann auf keinen Fall Ø stehen!

IF exp% THEN ... <:ELSE ...>

Der normale IF ... THEN ... ELSE Befehl. Vor ELSE muß ":" stehen. Hinter THEN kann zwar durchaus ein Label stehen (IF ... THEN target), aber es wird nicht als "IF ... THEN GOTO target" aufgefaßt, sondern als "IF ... THEN GOSUB target" (siehe GOSUB).

IF exp$1 ... <ELSE IF exp$2 ... > <ELSE IF exp$i ... > <ELSE ... > END IF
Strukturiertes IF ... ELSE IF ... ELSE ... END IF. Das ELSE IF kann mehrfach vorkommen, ELSE allein nur einmal, END IF muß immer stehen. Hinter keinem der IF exp$ steht ein THEN. Es wird grundsätzlich maximal eine der Befehlsfolgen "..." ausgeführt. Wenn die erste Bedingung nicht zutrifft, wird die nächste geprüft, falls sie zutrifft, wird die dahinter liegende Befehlsfolge ausgeführt und dann sofort zum END IF gesprungen, etc. . Wenn keine der Bedingungen zutrifft, wird entweder zum END IF gesprungen, oder, wenn ein ELSE vorhanden ist, die Befehlsfolge hinter ELSE ausgeführt. (Falls es aus der obigen Definition nicht klar geworden ist: die ELSE IFs müssen nicht immer benutzt werden, das ELSE auch nicht; IF ... END IF oder IF ... ELSE ... END IF ist auch möglich)
Beispiel:

```
 1 INPUT a$
 2 IF a$=#$D
 3   PRINT "CR"
 4 ELSE IF a$=#$A
 5   PRINT "LINEFEED"
 6 ELSE IF a$=#$C
 7   PRINT "CLEAR SCREEN"
 8 ELSE
 9   PRINT "???"
10 END IF
```

Wenn man 13 oder $d eingibt, wird "CR", bei 1Ø oder $a wird "LINEFEED", bei 12 oder $c wird "CLEAR SCREEN", bei allen anderen Werten wird "???" ausgegeben.

Die Schleifen und strukturierten IF ... END IFs können (fast) beliebig geschachtelt werden.

4.4 Maschinennahe Befehle und Funktionen

POKE$ exp, exp$1 <,exp$2> <...>
Lädt das Byte/die Bytes exp$1 (,exp$2, ...) in die Speicherstelle exp (exp+1, exp+...). Dieser Befehl entspricht dem Standard-BASIC-Befehl POKE.

PEEK$(exp)
Die entsprechende Funktion zu POKE$. PEEK$ entspricht der Standard-BASIC-Funktion PEEK.

POKE exp, exp1 <, exp2> <...>
Wie POKE$, aber es werden immer zwei Bytes gepoked (Reihenfolge Low-, Highbyte). Dieser Befehl entspricht nicht dem Standard-BASIC-Befehl POKE (siehe POKE$).

PEEK(exp)
Die entsprechende Funktion zu POKE. PEEK entspricht nicht der Standard-BASIC-Funktion PEEK!

POKE% exp, exp%1 <, exp%2> <...>
Wie POKE, nur mit Integer-zahlen.

PEEK%(exp)
Die entsprechende Funktion zu POKE%.

POKE! exp, exp!1 <, exp!2> <...>
Wie POKE$, aber es werden immer 4 Bytes abgelegt.

PEEK!(exp)
Die entsprechende Funktion zu POKE!.

POKE$ exp, exp$
Lädt den String exp$ in den Speicher ab exp, die Stringlänge wird nicht mit abgespeichert.

PEEK$(exp)
Die entsprechende Funktion zu POKE$ exp, exp$. An exp ist die Länge des Strings gespeichert, ab exp+1 befindet sich der String.

POKE$ <(exp)>, exp$
Wie POKE$, aber der String wird ab exp+1 abgespeichert, die Länge an exp.

PEEK$(exp, exp#)
Die entsprechende Funktion zu POKE$ <(exp)>, exp$. Es werden exp# Bytes ab exp aus dem Speicher gelesen und in einen String umgewandelt.

CALL exp
Ruft die Maschinenroutine ab exp auf.

ML x1 <,x2> <...>
x1, x2, ... entweder Variablennamen, konkrete Zahlen (Konstanten) mit einem Wert von Ø-255 oder /lbl. Diese Bytes werden direkt in das Maschinenprogramm übernommen, anstelle der Variablennamen wird die Adresse (Reihenfolge Low-, High-Byte), an der sich der Variableninhalt befindet, übernommen. Bei /lbl wird die Adresse der Routine lbl übernommen. Wenn mit Hilfe der Compileroption "b+" nach jedem Befehl auf <BREAK> geprüft wird, wird auch hinter jedem ML-/CALL-Befehl ein RST-Befehl angehängt, der auf <BREAK> prüft; hierbei werden aber keine Register zerstört.
Beispiel:

```
ML $2A,xk,$ED,$5B,yk,$CD,$6,$BC,$CD,/tlx,$CD,$7,$BD
```

Lädt HL mit dem Wert der Variablen xk, DE mit dem von yk und ruft dann die Routine ab $BCØ6 auf. Dann wird die BASIC-Routine tlx aufgerufen und schließlich die Routine ab $BDØ7.

OUTP exp$1, exp$2
Gibt das Byte exp$2 auf dem Port exp$1 aus (LD C,exp$1; LD A,exp$2; OUT (C),A).

INP exp$, var$
Liest ein Byte vom Port exp$ und speichert es in var$ (LD C,exp$1; INP A,(C); LD (var$),A).

OUTPH exp, exp$
Gibt das Byte exp$ auf dem Port exp aus. Im Gegensatz zum OUTP-Befehl wird der Port durch 16 Bits adressiert (LD BC,exp; LD a,exp$; OUT (C),A).

INPH exp, var$
Liest ein Byte vom Port exp und speichert es in var$. Im Gegensatz zum INP-Befehl wird der Port durch 16 Bits adressiert (LD BC,exp; INP A,(C); LD (var$),A).

MAXADR
Diese Funktion liefert die höchste Adresse, die vom kompilierten Maschinenprogramm belegt wird. Ab dieser Adresse sind eine Reihe von Bytes, deren Anzahl von der Systemfunktion SIZE geliefert wird, frei verfügbar.

SIZE
Diese Funktion liefert die Anzahl der Bytes, die ab MAXADR frei verfügbar sind.

ADRS(varx)
Die Speicheradresse des Variableninhaltes wird zurückgegeben.

ADRSL(label)
Die Adresse/Der Wert des Labels wird zurückgegeben.

4.5 Fehlerbehandlungs-Befehle und Funktionen

ON ERROR GOTO lbl
Wenn ein Fehler (<BREAK> ist kein Fehler) auftritt, wird nach lbl gesprungen. Der Stack wird hierbei nicht zurückgesetzt. Es stehen jetzt einige Fehlerfunktionen zur Verfügung, um den Fehler zu analysieren.

ON ERROR ERROR
Wenn ein Fehler auftritt, wird der Fehler auch ausgegeben (Standardzustand).

ON BREAK ERROR
Wenn ein <BREAK> festgestellt wird, wird ein Fehler mit der Nummer 8 generiert, so daß man den Abruch des Programmes durch <BREAK> verhindern kann.

ON BREAK BREAK
Wenn ein <BREAK> festgestellt wird, wird dies dann auch ausgegeben (Standardzustand).

RESUME lbl
Gleiche Wirkung wie GOTO lbl. Wird zur Fortführung des Programmes benutzt, nachdem ein Fehler durch ON ERROR GOTO abgefangen wurde und die Fehlerbehandlungsroutine fertig ist. Siehe unbedingt "SP=exp".

RESUME NEXT
Führt das Programm genau dort fort, wo der Fehler auftrat. Wenn der

Fehler also mitten in einem arithmetischem Ausdruck auftrat, wird das Programm genau in diesem arithmetischem Ausdruck fortgeführt, wobei aufgrund falscher Registerwerte natürlich ein falsches Ergebnis berechnet wird. Bei einem Interpreter würde der Befehl hinter dem fehlerverursachendem Befehl ausgeführt werden. Diesen Befehl sollte man nur benutzen, wenn man genau weiß, was man tut.

ERROR exp$
Der Fehler exp$ wird generiert ('ERROR $1' erzeugt also ein "S error"). Das entsprechende Gerät ist "ML".

TEST SP
Prüft, ob der Stack übergelaufen ist (über die Sicherheitsgrenze). Wenn das der Fall ist, wird ein "STO error" generiert. Dieser Befehl ist eigentlich nur dann sinnvoll, wenn man sich in rekursiver Programmierung versucht.

RESET SP
Setzt den Stackpointer zurück. Dies kann bei der Fehlerbehandlung sinnvoll sein (besonders, wenn unabhängig vom Fehlerort eine bestimmte Stelle angesprungen wird).

SP=exp
Setzt den Stackpointer auf exp. Wenn man hier den falschen Wert zuweist, kann dies einen Absturz zu Folge haben. Vor und nach dem Abarbeiten eines Befehl hat der Stackpointer immer den gleichen Wert (Ausnahme: GOSUB, RETURN, RESET SP und eben SP=exp). Dagegen kann sich während der Abarbeitung eines Befehls der Wert des Stackpointers sehr wohl verändern. Wenn nun nach einer solchen Veränderung ein Fehler auftritt (z.B. bei Klammern) und der Stackpointer noch nicht wieder den Wert hat, den er am Ende des Befehls haben würde, dann kann die einfache Fehlerbehandlung mit "RESUME lbl" einen Absturz oder sonstige extrem schwere Pannen oder Fehler zur Folge haben. Um dies zu vermeiden, gibt es die Funktion "SP", die den Wert des Stackpointers zurückgibt und den Befehl "SP=exp". Vor einem fehlergefährdetem Befehl schreibt man dann z.B. "errsp=SP" und bei der Fehlerbehandlungsroutine vor dem RESUME lbl dann "SP=errsp" (statt "errsp" kann man auch eine andere Variable benutzen). Dieses etwas aufwendige Vorgehen ist nur dann sinnvoll, wenn

1. direkt auf den Fehler reagiert werden soll. Wenn bei einem Fehler z.B. einfach in ein Menü o.Ä. gesprungen werden soll, ist RESET SP der richtige Befehl vor dem RESUME.
2. und tatsächlich der Wert des Stackpointers verändert wird. Bei "a=1/Ø" ist dies z.B. nicht der Fall, bei "a=1+(1/Ø)" dagegen sehr wohl. Solche Stackveränderungen kommen innerhalb eines Befehls bei Klammern (wenn vor der Klammer noch was steht; also nicht bei "a=(1/Ø)+1)") (Funktionen sind Klammern gleichgestellt) und dort vor, wo implizit Klammern gesetzt werden (z.B. wird "a+b*c" eigentlich wie "a+(b*c)" berechnet).

Beispiele zur Fehlerbehandlung:

```
1 ON ERROR GOTO e
2 FOR x!=Ø. TO 319.
3   y!=1./x!
4   PRINT x!,y!
5   et::
6 NEXT
```

```
  7 END
  8 e::
  9   y!=3.4E38
 1Ø RESUME et
```

Wenn hier also 1./Ø. zum Value error führt, wird y! auf 3.4E38 gesetzt.

```
  1 ON ERROR GOTO e
  ...
 55 elnØ::
 56 INPUT a!
 57 eln1::
 58 PRINT LN!(a!)
 59 eln2::
  ...
131 e::
132 IF ERRADR>=eln1 AND ERRADR<=eln2 THEN PRINT "Eingabe>Ø!":RESUME elnØ
  ...
```

Hier wird gezeigt, wie man den Zeilenbereich feststellen kann, in dem der Fehler auftrat. In diesem Fall ist wäre es natürlich besser, direkt beim INPUT den Fehler abzufangen.

```
 1 ON ERROR GOTO e
 2 a::
 3 GOSUB routine
 4 END
 5 routine::
 6 PRINT LN!(Ø.)
 7 RETURN
 8 e::
 9 RESUME a
```

Bei diesem Beispiel wird beim Autreten eines Fehlers zu einer bestimmten Stelle, nämlich a, gesprungen. Wenn Sie dieses Beispiel so eingeben, werden Sie schnell feststellen, daß dies so nicht gut geht. Denn bei jedem GOSUB-Befehl werden zwei Bytes auf dem Stack abgelegt, die hier nicht wieder (durch den RETURN-Befehl) entfernt werden. Der Stack wird also in Millisekunden überlaufen und einen Absturz herbeiführen. Vor dem RESUME müßte man den Befehl RESET SP einfügen, damit dies nicht passiert.

```
 1 ON ERROR GOTO e
 2 FOR a=Ø TO 9
 3   errsp=SP
 4 te::
 5   PRINT a;". number:";
 6   INPUT n
 7   PRINT 1.3+LN1(1+(n-9)*54)
 8 NEXT
 9 END
10 e::
11   PRINT "Error!  n>=9!"
12   SP=errsp
13   RESUME te
```

Wenn hier ein Fehler dadurch auftritt, daß n<9 ist, wird n noch einmal

angefordert. Der Fehler wird bei der Operation n-9 entdeckt (Cardinalzahlen können nicht negativ sein); da diese Operation in Klammern steht und zudem noch 1+ davor steht, wird der SP beim Abarbeiten dieses Ausdrucks verändert. Daher können die beiden Zeilen 'errsp=SP' und 'SP=errsp' nicht wie beim ersten Beispiel weggelassen werden.

ERRN$
Diese Funktion gibt die Fehlernummer zurück (siehe "ALLGEMEINES: Fehlermeldungen".

ERRDEV$
Diese Funktion gibt die Nummer des betroffenen Gerätes zurück (siehe "ALLGEMEINES: Logische/Physikalische Geräte".

ERRADR
Diese Funktion gibt die Adresse zurück, an der Fehler auftrat.

5 DRUCKERANSTEURUNG

5.Ø Allgemeines

Der Drucker wird über die Centronicsschnittstelle angesteuert. Falls der Drucker für mehr als etwa drei Sekunden im Busy-zustand ist, d.h. kein Zeichen annimmt, wird die Meldung "LPT:NRED error" ausgegeben. Das Drucken kann man grundsätzlich mit <BREAK> abbrechen.

Das einzige Steuerzeichen, welches XBC an den Drucker ausgibt, ist das CR-Zeichen (Code ØDH). Dieses Steuerzeichen muß auch einen Zeilenvorschub bewirken. Alle anderen Steuerzeichen lassen sich fast beliebig einstellen.

Der Rest dieses Kapitels bezieht sich nur auf den Editor. Für Programmgesteuerte Druckausgaben (BASIC) siehe "PRINT".

Das Programm wird Seitenweise ausgegeben. Dabei steht in der ersten Zeile die Seitennummer und, falls angegeben, der Name des Programmes.

5.1 Befehle zur Steuerung des Druckers

p ;Print on printer
Gibt alles hinter P stehende auf dem Drucker aus.
Beispiel:
"pXBC compiler <CR>" gibt "XBC compiler" und dann den Code ØDH (CR) auf dem Drucker aus, damit der Drucker einen Wagenrücklauf ausführt.

p:x ;Set printer parallel
x ist der Listbefehl. Der Drucker läuft jetzt parallel.
Beispiel:
"p:l 1Ø/ :Print-prime <CR>" gibt alle Zeilen ab 1Ø auf dem Drucker aus. Hierbei steht "Print-prime" in der Titelzeile.

p+b h1 <,h2> <, ...) ;Print bytes on printer
Die Bytes h1 <, h2, ...) werden direkt auf den Drucker ausgegeben.
Beispiel:
"p+b 42 41 <CR>" würde "BA" auf den Drucker ausgeben

p+s <h1><,h2><, ...) ;Set printer skip codes
Es können maximal 5 Bytes angegeben werden. Diese Bytes müssen einen Seitenvorschub auf dem Drucker auslösen. Die aktuellen Codes lassen sich mit "s <CR>" listen (Standard ØCH).

p+f ;Form feed
Die mit "p+s <h1><,h2><, ...> <CR>" eingestellten Steuerzeichen für den Seitenvorschub werden auf den Drucker ausgegeben, um einen Seitenvorschub auszulösen.

n dd ;Set number of lines/page
dd>9. Setzt die Anzahl der Zeilen/Seite fest. Hierbei müssen die zwei Titelzeilen berücksichtig werden. Standard 62. Die aktuelle Seitenlänge kann mit "s <CR>" gelistet werden.

6 INTERNES

Interne Darstellung von Daten

Bytes
Bytes benötigen jeweils ein Byte Speicherplatz und können Werte von Ø bis 255 ($FF) enthalten. Bytes werden bei arithmetischen Operationen in den Registern A und B gespeichert.

Cardinalzahlen
Cardinalzahlen bestehen aus 16 Bits, benötigen jeweils 2 Bytes Speicherplatz und können Werte von Ø bis 65535 ($FFFF) enthalten. Die beiden Bytes werden in der Reihenfolge Low-, Highbyte abgespeichert. Cardinalzahlen werden bei arithmetischen Operationen in den Registern HL und DE gespeichert.

Integerzahlen
Integerzahlen bestehen aus 16 Bits und werden im 2-er-Komplement dargestellt, d.h. das 15. Bit enthält das Vorzeichen (Ø=+; 1=-). Sie benötigen 2 Bytes Speicherplatz und können Werte von -32768 (-$4ØØØ) bis +32767 ($7FFF) enthalten. Die beiden Bytes werden in der Reihenfolge Low-, Highbyte abgespeichert. Integerzahlen werden bei arithmetischen Operationen in den Registern HL und DE gespeichert.

Realzahlen
Realzahlen bestehen aus 32 Bits: 1 Bit für das Vorzeichen (Ø=+; 1=-), 23 Bits für die Mantisse, wobei das Komma direkt hinter dem ersten Bit steht, und 8 Bits für den Exponenten im 2-er-Komplement. Der Wert der Zahl ist Mantisse*(2^Exponent) (dazu kommt noch das Vorzeichen).
Beispiel:

```
1Ø.:    1.Ø1Ø ØØØØ  ØØØØ ØØØØ  ØØØØ ØØØ * 2^ØØØØ ØØ11
Ø.1:    1.1ØØ 1ØØ1  1ØØ1 1ØØ1  1ØØ1 1Ø1 * 2^1111 1ØØ1
```

Es können Zahlen von +/- 2.938736E-39 bis 3.4Ø2823E+38 dargestellt werden. Realzahlen werden bei arithmetischen Operationen in den Registern DE, HL und DE', HL' gespeichert. Dabei enthält das Ø. Bit in L das Vorzeichen, das D-Register enthält den Exponenten, das E-Register enthält die ersten 8 Bits der Mantisse, H die nächsten 8 und L (außer L(Ø)) die restlichen 7 Bits. Realzahlen benötigen 4 Bytes Speicherplatz und werden in der Reihenfolge E, D, L, H abgespeichert.

Strings
Strings werden durch eine Anzahl von Bytes dargestellt. Im ersten Byte steht jedoch die aktuelle Länge des Strings, so daß ein Standardstring, der maximal 8Ø Zeichen enthalten kann, effektiv 81 Bytes im Speicher belegt; der eigentliche String steht dann in den folgenden Bytes. Es werden immer (maximale Anzahl)+1 Bytes im Speicher belegt, auch wenn der String leer ist. Die maximale Länge eines Strings wird nicht abgespeichert, sie wird nur beim Kompilieren benötigt.

Arrays
Beim DIM-Befehl muß mit x1 (x2, ...) die Anzahl der Dimensionen und der jeweilige Indexbereich festgelegt werden. Der Speicherbedarf berechnet sich dann folgendermaßen: (x1+1)*(x2+1)...*(Länge eines Elementes).

Die Speicheradresse eines beliebigen Elementes wird durch Offset+(x1+(x1+1)*x2+(x1+1)*(x2+1)*x3+...)*(Länge eines Elementes) berechnet (Offset ist die niedrigste Adresse, die das Array belegt). Je weniger Dimensionen ein Array hat, desto schneller wird also die Adresse berechnet (und desto kürzer ist das entsprechenden Maschinenprogramm); bei Bytes, Cardinalzahlen, Integerzahlen und Realzahlen ist die Multiplikation "...*(Länge eines Elementes)" nicht nötig, da hier Additionen genügen.

Weitere Informationen stehen im Anhang.

7 HILFE FOR ANFANGER

BASIC- und Programmierkenntnisse werden selbstverständlich vorausgesetzt.

Zuerst müssen Sie sich mit dem Zeileneditor vertraut machen. Dies tun Sie am besten dadurch, daß Sie sich die Beschreibung dieses Editors am Anfang des Kapitels "DER EDITOR" durchlesen (nicht mit den Editorbefehlen zu verwechseln); Ergänzungen hierzu (Tastaturbelegung) finden Sie im Anhang.
Versuchen Sie, Folgendes einzugeben (Cursorposition ist unterstrichen; Steuerzeichen in Klammern):

```
1. >1 for a=0 to 999 step 2
2. >1 for a=0 to 999 step 2        (^A ^A ^A ^A ^D)
3. >1 for a=100 to 999 step 2      (^V "10" ^V)
4. >1 for a=100 to 999_step 2      (^F ^F)
5. >1 for a=100 to 999_            (^T ^T)
6. >_                              (^Z (Eingabe gelöscht))
```

Das Nächstwichtigste ist der Editor (siehe Kapitel "DER EDITOR"). Zunächst brauchen Sie nur die Befehle "i", "l", "c" und natürlich "d". Sehen Sie sich außerdem den Datentyp "lnr" im Kapitel "DATEN" an; vorläufig können Sie sich allerdings damit begnügen, für lnr immer Zeilennummern oder gar nichts einzusetzen. Da die Befehle selbst gut genug erklärt sind, soll hier nur kurz auf die Reaktion des Compilers/Editors bei Fehlern eingegangen werden (zum Starten eines Programmes brauchen Sie nur "j <CR>" einzugeben).

Editor:

Wenn Sie

```
>i <CR>
  1 # <CR>
```

eingeben, wird der Editor mit

```
ED:S error (Syntax error)
  1 # (Cursor wartet)
```

reagieren, denn hinter einem "#" wird immer eine Zahl erwartet. Fehler dieser Art (Syntax oder Overflow) werden immer dann ausgegeben, wenn der Editor die Zeile nicht kodieren kann.
Die zwei anderen Fehler, die auftreten können, sollen auch kurz angeführt werden. Der eine ist der Fehler

ED:MCAP error (Memory capacity error: kein Speicher mehr frei)

Dieser Fehler sollte besser nicht auftreten, denn in diesem Falle müssen Sie irgendwelche Zeilen etc. löschen, wenn Sie weiterarbeiten wollen. Der andere Fehler muß förmlich provoziert werden; er tritt nur dann auf, wenn eine gelistete Zeile länger als 255 Zeilen werden würde. Dies kann nur geschehen, wenn die Ausgabe länger als die Eingabe wird, z.B. bei folgender Eingabe:

```
>1 ?????????????????????????????????????????????????????????????? <CR>
```

Gelistet werden würde

```
1 PRINT PRINT PRINT PRINT ... (330 Zeichen)
```

Da nicht mehr als 255 Zeichen/Zeile gelistet werden können, wird
```
ED:STRL error (String length error: Text zu lang(/kurz))
```
ausgegeben, sobald versucht wird, diese Zeile zu listen.

Compiler:

Fehler wie
```
1 PRINT PRINT
```
oder
```
1 a="text"
```
werden nicht vom Editor, sondern vom Compiler erkannt. Das Kompillieren wird mit "j <CR>" gestartet. Wenn der Compiler einen Fehler entdeckt, dann sieht das so aus:
```
COM:S error (Syntax error)
  1 PRINT PRINT
```
oder
```
COM:TYPM error (Type mismatch error: Falscher Datentyp)
  1 a="text"
```
D.h. der Fehler und die fehlerhafte Zeile werden ausgedruckt. Da die eben gelistete Zeile jetzt aktuelle Zeile ist, kann man sie mit "c; <CR>" korrigieren.

Die Reaktion auf Fehler während des Programmablaufs sollen ebenfalls kurz dargestellt werden. Beim Ablauf des Mini-Programms
```
1 PRINT 1/Ø
```
passiert folgendes:
```
ML:V error at PC=???? (Value error: Wert falsch)
<< Search.line >>
  1 PRINT 1/Ø
```
Hierbei ist ???? die Adresse, an der der Fehler auftrat (hexadezimal). Jetzt kann man die Zeile mit "c; <CR>" korrigieren. U.U. könnte anstelle der Zeile auch
```
<< Line not ex.(?) >>
```
gemeldet werden. Dies ist ein schwerer Fehler, der entweder durch "herum-POKEn" etc. oder durch einen Compilerfehler hervorgerufen wurde. Wenn diese Meldung ausgegeben wird und Sie glauben, daß Sie selbst nicht für diesen Fehler verantwortlich sind, dann melden Sie dies bitte BBG.

Nachdem Sie sich nun (hoffentlich) ein wenig mit dem Editor und Compiler vertraut gemacht haben, sollen einige Besonderheiten des Compilers behandelt werden:

1. Es gibt keine Zeilennummern im üblichem Sinne (die Zahlen vor der Zeile dienen lediglich zur Nummerierung). So etwas wie "GOTO 1790" ist also nicht möglich. Statt der Zeilennummern müssen sogenannte Labels benutzt werden. Dies sieht dann so aus:

```
1 FOR n=Ø TO 9
2   GOSUB subrout
3 NEXT
4 END
5
6 subrout::
```



```
7   PRINT n,n*n
8 RETURN
```

Dieses XBC-Programm ersetzt das Standard-BASIC-Programm

```
1Ø FOR N=Ø TO 9
2Ø   GOSUB 6Ø
3Ø NEXT
4Ø END
5Ø :
6Ø PRINT N,N*N
7Ø RETURN
```

Die Zeile, die man "anspringen" will, muß also durch ein Label gekennzeichnet werden, bevor man diese Zeile mit GOTO/GOSUB anspringen kann. Ein solches Label kann - wie auch alle Variablen - beliebig lang sein; um Speicherplatz zu sparen und weil bei langen Labals/Variablen häufiger Schreibfehler vorkommen, ist es besser, nicht mehr als etwa 8 Zeichen/Label/Variable zu benutzen. Übrigens sollte man alle Labels und Variablen zu besseren Unterscheidung von den Schlüsselwörtern nur klein schreiben.

Das XBC-Programm könnte übrigens auch so aussehen:

```
1 FOR n=Ø TO 9:subrout:NEXT:END
2 subrout:: PRINT n,n*n:RETURN
```

D.h. das GOSUB ist nicht nötig und nach einem Label können auch weitere Befehle folgen. Wichtig ist, das Labels nur am Anfang einer Zeile stehen können (das obige Programm kann also nicht in einer Zeile geschrieben werden).

Wenn ein Label nicht definiert ist, es aber trotzdem benutzt wird, meldet der Compiler dies. Nehmen wir an, daß die Zeile 6 (6 subrout::) im obigen Beispiel nicht eingegeben wurde. Dann würde der Compiler
```
COM:SYMNE error
  2 GOSUB subrout
```
melden (natürlich erst beim Kompilieren (also "j <CR>" eingeben)).

2. Es gibt eine Reihe von verschiedenen Zahlentypen. Diese sind im Kapitel "DATEN" aufgeführt. Sehen Sie sich auch die entsprechenden Zahlen und Variablen an! Das Anwenden von falschen Datentypen kann zu schweren Fehlern führen. Hierbei sind auch Schreibfehler zu beachten. Wichtig ist, daß Zahlen und Variablen ohne Kennzeichnung als Cardinalzahlen/-variablen aufgefaßt werden. Das Programm

```
1 FOR a=1 TO 1Ø STEP .5
2   PRINT a
3 NEXT
```

würde also keineswegs "1 1.5 2 ..." ausgeben, sondern vielmehr "1 1 1 ...". Dies kommt daher, daß ".5" eine Realzahl ist, die beim FOR-Befehl erst einmal in eine Cardinalzahl umgewandelt wird. Da hierbei quasi die INT-Funktion benutzt werden muß, wird also Ø daraus. Das Programm müßte so aussehen:

```
1 FOR a!=1. TO 1Ø. STEP .5
2   PRINT a!
3 NEXT
```

ALLGEMEINE ANLEITUNG FÜR XBC S 1.ØX

Der Computer kann immer nur gleichartige Zahlen miteinander verrechnen, d.h., daß z.B. "a=a*7.7" eigentlich nicht möglich ist, da hier eine Cardinal- und eine Realzahl miteinander multipliziert werden. Der Compiler erkennt dies aber und sorgt dafür, daß während des Programmlaufes der Inhalt der Cardinalvariablen "a" in eine Realzahl verwandelt wird. Dann wird mit Realzahlen weitergerechnet. Da das Ergebnis natürlich auch eine Realzahl ist, aber die Variable "a", der das Ergebnis zugewiesen werden soll, eine Cardinalvariable ist, muß das Ergebnis vor der Zuweisung in eine Cardinalzahl umgewandelt werden. Dafür sorgt der Compiler automatisch. (Man könnte diese Umwandlungen auch "von Hand" machen, indem man "a=CONV(CONV!(a)*7.7)" schreibt).

3. Der Compiler bietet eine Reihe von zusätzlichen Befehlen an, die vor allem den Programmablauf steuern (LOOP ... END LOOP, REPEAT ... UNTIL x etc.). Wenn Sie diese Befehle benutzen, wird das Programm erstens übesichtlicher und zweitens können Sie auf etwas unübersichtliche Konstruktionen mit IF und GOTO weitgehend verzichten.

Wenn Sie jetzt noch irgendwelche Fragen haben sollten, dann lesen Sie sich die Anleitung bitte nochmals gründlich durch. Falls das nicht ausreichen sollte, wenden Sie sich am besten an einen "Experten", immerhin steht alles Wichtige in der Anleitung. Im Notfall wenden Sie sich (möglichst schriftlich) an BBG.

ANHANG FÜR XBC S 1.ØX AUF MZ-7ØØ/8ØØ

Letzte Überarbeitung des Anhanges am 19.1.1987.

INHALTSVERZEICHNIS

0 ALLGEMEINES

0.1 Laden des Compilers

Kassette:
XBC wird mit 'L (CR)' vom Rommonitor geladen und gestartet.
QD:
XBC wird mit 'QL (CR)' vom Rommonitor geladen und gestartet.

0.2 Unterstützte Geräte

Auf MZ-700:
c (CMT) und q (QD).
Auf MZ-800:
c (CMT), q (QD) und r (Ramdisk).

Für den MZ-800 gilt: Kassettenoperationen können nur im MZ-700-Modus durchgeführt werden. Dieser Modus wird vom Editor mit 'v4 (CR)' eingeschaltet, von einem kompilliertem XBC-BASIC-Programm mit 'CRT #0'. Wenn der MZ-700-Modus nicht eingeschaltet ist, wird 'MOD error' ausgegeben.

Bei der Kassette können folgende Operationen durchgeführt werden:
l/LOAD, s/SAVE und v/VERIFY. Die Speicherbereiche 0-$FFF und $D000-$FFFF können nicht abgespeichert werden.

Bei der Quickdisk können folgende Operationen durchgeführt werden:
l/LOAD, s/SAVE, d/DIR und f/INIT (Formatiert die QD). Der Speicherbereich $D000 (MZ-700-Modus) / $E010 (MZ-800-Modus) - $FFFF kann nicht abgespeichert werden.

Bei der Ramdisk können folgende Operationen durchgeführt werden:
l/LOAD, s/SAVE, d/DIR, f/INIT (Formatiert die RD; unbedingt zur Initialisierung durchführung!), k/DELETE (ein File löschen) und r/RENAME.

0.3 Hinweise zum MZ-700- und MZ-800-Modus

Auf dem MZ-700 gibt es nur den MZ-700-Modus, auf dem MZ-800 dagegen beide Moden. Manche systemspezifische Befehle funktionieren nur in einem bestimmtem Modus, andere in beiden, wobei die Wirkung allerdings etwas unterschiedlich sein kann, und bestimmte setzen sogar eine RAM-Erweiterung (64 KB Ramdisk; 16 KB Videoram) voraus.

Der MZ-700-Modus:
Hier ist immer die PCG-Grafik eingeschaltet. Der Bildschirm ist 40*25 Zeichen groß. Jedes Zeichen ist 8*8 Punkte groß, die immer gemeinsam gesetzt werden. Es gibt 512 verschiedene 8*8-Punkte große Muster (siehe VPOKE und VPEEK). Für jedes Zeichen können Vorder- und Hintergrundfarbe beliebig gesetzt werden. Eine hochauflösende Grafik ist nicht möglich, nur eine Spezialgrafik mit 80*50 Punkten, die durch Ausnutzen der Zeichen 240 ($F0) bis 255 ($FF) zustande kommt. Beim MZ-800 können die einzelnen Zeichen beliebig definiert werden (siehe PCG-Befehl), beim

MZ-700 sind sie vordefiniert und nicht zu ändern. Diese Grafik wird für viele Spiele benutzt.

Der MZ-800-Modus:
Im MZ-800-Modus gibt es vier verschiedene Grafikmoden (siehe CRT-Befehl). Hier ist immer die hochauflösende Grafik eingeschaltet. Zeichen müssen in diesem Modus dadurch erzeugt werden, daß einzelne Punkte gesetzt werden.

Grundsätzlich ist zu beachten, daß sich die Speicheraufteilung und z.T. auch die I/O-Ports je nach Modus ändern. Dies ist aber nur für Maschinensprachprogrammierer interessant.

0.4 Speicheraufteilung

Siehe auch "Unterstützte Geräte" und "Hinweise zum MZ-700- und MZ-800-Modus". XBC arbeitet immer mit 64 KB RAM. Bei LOAD-/SAVE-...Operationen wird der Speicherbereich $1000-$1200 immer z.T. zerstört. Ab $1200 befinden sich bestimmte Stacks und Routinen, dahinter befindet sich der Quelltext in kodierter Form.

Warmstartadresse ist $1200.

0.5 ASCII-Zeichensatz

Es wird nicht der teilweise seltsamme SHARP-ASCII-Zeichensatz, sondern der Standard-ASCII-Zeichensatz benutzt, den jeder normale Drucker sofort "versteht". Daher werden insbesondere die Kleinbuchstaben vom ROM (Befehl "QD"; Meldung 'WRITING ???') und vom BASIC-Interpreter als Grafikzeichen ausgegeben. Dieser Standardzeichensatz sieht so aus:

	00	01	02	03	04	05	06	07	Erläuterung:
00				0	$	P	`	p	LE: Cursor left
01			!	1	A	Q	a	q	RI: Cursor right
02			"	2	B	R	b	r	BR: Break (CONTROL-C)
03	BR	LE	#	3	C	S	c	s	UP: Cursor up
04	RI		$	4	D	T	d	t	DN: Cursor down
05	UP		%	5	E	U	e	u	BL: Bell
06			&	6	F	V	f	v	TB: Tabulator
07	BL		'	7	G	W	g	w	ES: ESCape
08		DN	(	8	H	X	h	x	FF: Form feed (CLS)
09	TB		)	9	I	Y	i	y	CR: Carridge return
0A	DN	FF	*	:	J	Z	j	z	HM: Home
0B		ES	+	;	K	Ä	k	ä	NL: Next line (wenn CURX#(>0: CR)
0C	FF		,	<	L	Ö	l	ö	
0D	CR	HM	-	=	M	Ü	m	ü	
0E		NL	.	>	N	^	n	ß	Der Drucker versteht von diesen Steuer-
0F			/	?	O	_	o		zeichen aber nur CR, ES, TB, FF.

Alle ASCII-Zeichen über $7E (126) werden auf dem Bildschirm als Leerzeichen gedruckt. Wenn Sie einen SHARP-Drucker besitzen, können Sie mit Hilfe des SHARP-Befehls die ASCII-Zeichen über $5E in SHARP-ASCII umkodieren lassen; gleiches geht auch mit dem Befehl "p+p (CR)" vom Editor aus.

Die obige Tabelle zeigt den deutschen Zeichensatz (siehe "PRINT"). Normalerweise wird im US-Zeichensatz gearbeitet; folgende Zeichen sehen anders aus:

$40: Klammeraffe
$5B: Eckige Klammer auf
$5C: Backslash: umgekehrter Divisionstrich ($2F)
$5D: Eckige Klammer zu
$7B: Geschweifte Klammer auf
$7C: Senkrechter Strich
$7D: Geschweifte Klammer zu
$7E: Geschlängeltes Minuszeichen

0.6 Tastaturbelegung

F1-F10:	Definierbarer String.
GRAPH:	Nichts.
TAB:	^I (8-er Tabulator; (CHR$(9)).
ALPHA:	Schaltet zwischen Groß- und Kleinschrift um.
BREAK:	ESC (CHR$(27)).
SHIFT+BREAK:	^C ((BREAK); CHR$(3)).
Leere Taste:	"_".
CR:	^M (CR; CHR$(13)).
INST:	^V (Insert on/off; CHR$(22)).
SHIFT+INST:	^N (CHR$(12)).
DEL:	^G (Delete; CHR$(7)).
SHIFT+DEL:	CHR$(29).
CURSOR UP:	^E (CHR$(5)).
SHIFT+CURSOR UP:	^W (CHR$(23)).
CURSOR DOWN:	^X (CHR$(24)).
SHIFT+CURSOR DOWN:	^Z (CHR$(26)).
CURSOR RIGHT:	^D (CHR$(4)).
SHIFT+CURSOR RIGHT:	^F (CHR$(6)).
CURSOR LEFT:	^S (CHR$(19)).
SHIFT+CURSOR LEFT:	^A (CHR$(1)).

0.7 Programmabbruch

Ein Programm kann nur abgebrochen werden, wenn auf Abbruch ((BREAK)) geprüft wird (siehe 'TEST BREAK' und 'OPTION'). Wenn auf die Taste 'BREAK' gedrückt wird, blinkt der Cursor und das Programm ist angehalten. Wenn jetzt eine beliebige Taste außer 'SHIFT-BREAK' gedrückt wird, wird das Programm fortgesetzt. Wenn 'SHIFT-BREAK' ((BREAK)) gedrückt wird, wird das Programm abgebrochen.

1 BEFEHLE UND FUNKTIONEN

1.1 Befehle vom Editor aus

v4 ;Videomode with 40 characters/line
Schaltet in den MZ-700-Modus mit 40 Zeichen/Zeile (entspricht 'CRT #0').

v8 ;Videomode with 80 characters/line
Schaltet in MZ-800-Modus mit 80 Zeichen/Zeile (entspricht 'CRT #3').
Dieser Befehl ist nur beim MZ-800 möglich.

p+x ;ASCII-converting for x-printer (Epson-kompatible)
Sorgt dafür, daß ASCII-Zeichen nicht umkodiert werden, bevor sie an den Drucker ausgegeben werden. Entspricht dem Befehl 'SHARP #0'.

p+p ;ASCII-converting for SHARP-Plotter/Printer
Sorgt dafür, daß ASCII-Zeichen bei der Ausgabe an den Drucker so umkodiert werden, daß keine Probleme entstehen. Nur die Steuerzeichen CR, FF, TB werden akzeptiert. Entspricht dem Befehl 'SHARP #1'.

1.2 Allgemeine Befehle und Funktionen

TI$

Die Systemfunktion TI$ enthält die Zeit als String im Format "hhmmss".

TI$=exp$

Setzt die Zeit neu. 12 Stunden nach diesem Befehl kann leider nicht mehr garantiert werden, daß die Zeit korrekt zurückgegeben wird.

Beispiel:

```
TI$="015959"
```

TIME

Diese Sonderfunktion gibt direkt die Anzahl der Sekunden nach dem letzten 'TI$=exp$'-Befehl. Diese Funktion ist häufig zum Messen von Zeiten besser geeignet.

MZ800$

Gibt TRUE$ ($255) zurück, wenn der Computer ein MZ-800 und, sonst FALSE$ ($0).

Beispiel:

```
1 IF MZ800$ THEN PRINT "SHARP MZ-800":ELSE PRINT "SHARP MZ-700"
```

1.3 I/O-Befehle und Funktionen

FKEY exp$, exp$

Weist der Funktionstaste exp$ ($1-$10) den String exp$ (Länge 0-7) zu.

Beispiel:

```
FKEY $5,"text"+CHR$(13)
```

JOY$(exp$)

Fragt die Tastatur (exp$=$0), den Joystick 1 (exp$=$1) oder den Joystick 2 (exp$=$2) ab. Nur die MZ-800-Joysticks werden unterstützt. Folgender Wert wird zurückgegeben:

Nichts: $0.
Links: $1; Links, hoch: $2; Hoch: $3 ...

FIRE$(exp$)

Siehe JOY$(exp$). Es wird entweder TRUE$ ($255) oder FALSE$ ($0) zurückgegeben, je nachdem, ob die Leer-/Feuertaste gedrückt ist.

FREQ exp

Dieser Befehl funktioniert nur im MZ-700-Modus. Der Tongenerator wird auf exp gesetzt. Wenn exp=0, wird der Tongenerator abgeschaltet. Die Hz-Frequenz errechnet sich folgendermaßen:

Hz:=895000/exp

SOUND exp$1, exp, exp$2

Dieser Befehl funktioniert nur auf dem MZ-800. Der Tongenerator exp$1 ($0-$2) wird auf die Frequenz exp (0-1023) und auf die Lautstärke exp$2 ($0-$15) gesetzt. Wenn exp$2=$0, wird der Tongenerator unhörbar (quasi abgeschaltet). Die Hz-Frequenz errechnet sich folgendermaßen:

Hz:=113600/exp

Beispiel:

```
SOUND $1,113600./880,$8
```

stellt den Tongenerator $1 auf 880 Hz und mittlere Lautstärke.

SSOUND exp$1, exp$2, exp$3

Dieser Befehl funktioniert nur auf dem MZ-800. exp$3 ist wieder die Lautstärke-($0-$15). Wenn exp$2=$0, wird kein Zufallsgenerator benutzt, um das Ergebnis zu beeinflussen, wenn exp$2=$1, geschieht dies. exp$1 ist die Betriebsart:

exp$1=$0: 6930 Hz
exp$1=$1: 3470 Hz
exp$1=$2: 1730 Hz
exp$1=$3: (Frequenz in Hz von Tongenerator $2)/16

Beispiele:

SSOUND $1,$0,$15 gibt einen 6930-Hz-Ton aus. Wenn die $0 zu $1 geändert wird, ergibt sich eine Art Rausch-effekt.

SSOUND $3,$1,$15: Die Basisfrequenz ist um den Faktor 16 tiefer als die Frequenz des 2. Tongenerators, außerdem ist der Rauscheffekt eingeschaltet. Der 2. Tongenerator wird hiervon nie beeinflußt. Mit 'SOUND $2,1023,$0' schaltet man die tiefste Frequenz des 2. Tongenerators an, wobei dieser aber unhörbar ist.

Hinweis zu PRINT

XBC kann auf dem Bildschirm drei Attribute und drei Zeichensätze darstellen, die folgendermaßen ein- und ausgeschaltet werden:

```
PRINT CHR$($1B,$45);: Fettdruck einschalten
PRINT CHR$($1B,$46);: Fettdruck ausschalten
```



```
PRINT CHR$($1B,$7D);: Inversdruck einschalten
PRINT CHR$($1B,$7E);: Inversdruck ausschalten
PRINT CHR$($1B,$2D,1);: Unterstreichen einschalten
PRINT CHR$($1B,$2D,0);: Unterstreichen ausschalten
PRINT CHR$($1B,$52,0);: US-Zeichensatz einschalten
PRINT CHR$($1B,$52,2);: Deutschen Zeichensatz einschalten
PRINT CHR$($1B,$52,3);: Englischen Zeichensatz einschalten
```

Im MZ-700-Modus ist Fettdruck und Unterstreichen nicht möglich.
Die Bits 3-15 beim 'PRINT (<exp>) ...'-Befehl haben keine Funktion.

CRT exp$

Stellt den MZ-Modus und den Grafikmodus ein. Grundsätzlich werden hierbei folgendes Befehle intern abgearbeitet: CLS; SETORG 0,0; US-Zeichensatz, kein Unterstreichen, keine Fettschrift bei 40 Zeichen/Zeile, bei 80 Zeichen/Zeile Fettschrift, keine Inversschrift; GMODE $0; PLOT FRAME $0; SHOW FRAME $0; SYNC $1; BORDER $0; außerdem werden die Farben wie unten angegeben gesetzt. Der Bildschirm wird mit CLS gelöscht.

Bei der hochauflösenden Grafik ist folgendes zur Farbgebung zu sagen: Es gibt zum einen den internen Farbkode. Dies ist eine Zahl von 0-1, 0-3 oder 0-15, je nach Grafikmodus. Ein solcher Farbkode ist für jeden Punkt auf dem Bildschirm im Grafikspeicher vorhanden. Zum anderen gibt es die absolute Farbnummer (immer 0-15). Mit dem PAL-Befehl kann man nun jedem internem Farbkode eine absolute Farbnummer zuordnen, mit 'PAL 1,15' z.B. würde man alle Punkte mit dem internem Farbkode 1 hellweiß auf dem Bildschirm erscheinen lassen.

Der (0;0)-Punkt befindet sich in der unteren, linken Ecke. Es steht eine virtuelle Grafik zur Verfügung, die interne Koordinaten von -16384 - +16383 erlaubt. Durch den SETORG-Befehl wird ein Offset angegeben, der immer zu den angegebenen Koordinaten addiert wird. Wenn Punkte außerhalb des sichtbaren Bereiches gezeichnet werden, passiert nichts.

exp$=$0: Der MZ-700-Modus mit der PCG-Grafik wird eingeschaltet. Die Befehle 'COLOR 7, 1: COLOR 1' werden durchgeführt. Die 80 * 50 Punkte große niedrigauflösende Grafik ist zweifarbig, d.h., beim COLOR-Befehl sind nur Werte von 0-1 zugelassen. Der PAL-Befehl hat keine Wirkung.

exp$=$1: Der MZ-800-Modus mit der hochauflösenden Grafik mit 320 * 200 Punkten und 4 Farben wird eingeschaltet. Beim COLOR-Befehl sind Werte von 0-3 möglich, beim PAL-Befehl kann der erste Parameter ebenfalls Werte von 0-3 annehmen. Die Befehle 'PAL 0,9: PAL 1, 11: PAL 2, 13: PAL 3, 15: COLOR 3' werden intern ausgeführt.

exp$=$2: Der MZ-800-Modus mit der hochauflösenden Grafik mit 320 * 200 Punkten und 16 Farben wird eingeschaltet. Dieser Befehl ist nur möglich, wenn die 16 KB VRAM Erweiterung eingebaut ist. Beim COLOR-Befehl sind Werte von 0-15 möglich, beim PAL-Befehl kann der erste Parameter ebenfalls Werte von 0-15 annehmen. Der Befehl 'COLOR 15' wird intern ausgeführt. In diesem Modus gibt es hinsichtlich der Farbgebung eine Besonderheit: Jedem internem Farbkode ist genau die gleiche absolute Farbnummer zugeordnet. Mit dem PAL-Befehl kann man aber dennoch bis zu 4 internen Farbkodes andere absolute Farbnummern zuordnen. Diese 4 internen Farbkodes müssen aber entweder 0-3, 4-7, 8-11 oder 12-5 sein, d.h., es ist nicht möglich z.B. den Farbkodes 5 und 13 gleichzeitig absolute Farbnummern zuzuordnen. Sobald man mit dem PAL-Befehl eine neue Gruppe (0-3, 4-7, 8-11, 12-15) anspricht, bekommt die alte Gruppe

wieder die absoluten Farbnummern zugewiesen.

exp$=$3: Der MZ-800-Modus mit der hochauflösenden Grafik mit 640 * 200 Punkten und 2 Farben wird eingeschaltet. Beim COLOR-Befehl sind Werte von 0-1 möglich, beim PAL-Befehl kann der erste Parameter ebenfalls Werte von 0-1 annehmen. Die Befehle 'PAL 0,9: PAL 1, 15: COLOR 1' werden intern ausgeführt.

exp$=$4: Der MZ-800-Modus mit der hochauflösenden Grafik mit 640 * 200 Punkten und 4 Farben wird eingeschaltet. Dieser Befehl ist nur möglich, wenn die 16 KB VRAM Erweiterung eingebaut ist. Beim COLOR-Befehl sind Werte von 0-3 möglich, beim PAL-Befehl kann der erste Parameter ebenfalls Werte von 0-3 annehmen. Die Befehle 'PAL 0,9: PAL 1, 11: PAL 2,13: PAL 3,15: COLOR 3' werden intern ausgeführt.

COLOR exp$1, exp$2
Stellt die Vorder- und Hintergrundfarben im MZ-700-Modus ein (exp$1, exp$2=$0-$7).

PCG exp, exp$1, exp$2, ..., exp$8
Dieser Befehl funktioniert nur beim MZ-800 im MZ-700-Modus. Das Zeichen exp (0-511) wird mit Hilfe der Bytes exp$1, ..., exp$8 definiert.
Beispiel:
PCG 0,$F3,$F3,$F3,0,$CF,$CF,$CF,0 definiert alle Leerzeichen in Mauerstücke um.

VPOKE exp$1, exp$2, exp, exp$3, exp$4
Schreibt das Videozeichen exp (0-511) an die Koordinate exp$1 ($0-$39), exp$2 ($0-$24) mit der Vordergrundfarbe exp$3 ($0-$7) und der Hintergrundfarbe exp$4 ($0-$7).
Beispiel:
VPOKE 39,24,1,0,7 schreibt ein 'A' in die rechte, untere Ecke, wobei das Zeichen schwarz und der Hintergrund weiß ist. Die Zahlen in diesem Beispiel sollten bis auf die 1 alle Bytes sein, d.h. ein '$' sollte davor stehen. So, wie das Beispiel jetzt dasteht, läuft es langsammer und verbraucht mehr Platz.

VPEEK(exp$1, exp$2)
Diese Funktion funktioniert nur im MZ-700-Modus. Das Zeichen (0-511) an der Koordinate exp$1, exp$2 wird zurückgegeben. Die Farbe kann nicht ermittelt werden.

BVPOKE exp$1, exp$2, exp$3, exp$4, < << > exp < >> >, exp$5, exp$6
Dieser Befehl funktioniert nur im MZ-700-Modus. exp kann von eckigen Klammern umgeben sein. Dieser Befehl schriebt ein Block der Größe exp$1 * exp$2 an die Koordinaten exp$3, exp$4 mit den Farben exp$5, exp$6. Wenn exp nicht in eckigen Klammern steht, ist das Zeichen in der linken, oberen Ecke exp, das rechts daneben exp+1, ... Wenn exp in eckigen Klammern steht, wird immer das gleiche Zeichen geschrieben. Dieser Befehl eignet sich gut zum Spieleschreiben.
Beispiele:
BVPOKE $17,$0,$5,$2,1,$7,$0 schreibt oben in der Mitte des Bildschirms folgendes:
ABCDE
FGHIJ
Wenn die 1 in eckigen Klammern stehen würde, würden nur 'A's ge-

schrieben werden.

SHARP exp$
Siehe die Editorbefehle 'p+x' und 'p+x'.

BORDER exp$
Setzt die Randfarbe beim MZ-800 auf exp$ ($0-$15).

COLOR exp$
Stellt die aktuelle Zeichenfarbe im MZ-800-Modus ein. Siehe CRT.

PAL exp$1, exp$2
Weist einem internem Farbkode (exp$1, siehe CRT) eine absolute Farbnummer (exp$2=$0-$15) zu. Der interne Farbkode $0 ist für den Hintergrund reseviert.
Beispiel:
PAL $0,$1 weist dem Hintergrund die absolute Farbe blau zu.

SYNC exp$
Dieser Befehl ist nur im MZ-800-Modus wirksam. Er bewirkt, wenn exp$=$1, daß ein Farbwechsel mit dem Bildschirmaufbau (auch scrollen) synchronisiert wird. Diese Synchronisation kostet aber Zeit. Ein Farbwechsel wird bei folgenden Befehlen angenommen: PLOT, LINE, PAL, BORDER, SHOW FRAME.
Folgendes Beispiel zieht den Effekt deutlich:

```
1 CRT $1:w=7
2 SYNC $0:GOSUB r
3 SYNC $1:GOSUB r:END
4 r::
5 DO 100
6   PAL $0,$0:WAIT w:PAL $0,$15:WAIT w
7 END DO:RETURN
```

SETORG exp%1, exp%2
Setzt den (0;0)-Punkt auf die absolute Koordinate exp%1, exp%2.
Beispiel:
SETORG %160,%100 setzt den (0;0)-Punkt, wenn man 'CRT $1' ausgeführt hat, in die Mitte des Bildschirmes.

GMODE exp$
Setzt die Überlagerungsbetriebsart fest (gültig for PLOT und LINE):
exp$=$0: Punkt setzen
exp$=$1: AND
exp$=$2: OR
exp$=$3: XOR

GMOVE exp%1, exp%2
Setzt den internen Grafikcursor auf die Koordinate exp%1, exp%2.

PLOT exp%1, exp%2
Setzt einen Punkt an die Koordinate exp%1, exp%2. Der interne Grafikcursor wird auf exp%1, exp%2 gesetzt.

LINE TO exp%1, exp%2
Zieht eine Linie vom internen Grafikcursor nach exp%1, exp%2. Der interne Grafikcursor wird auf exp%1, exp%2 gesetzt.

LINE exp%1, exp%2 TO exp%3, exp%4
Zieht eine Linie von exp%1, exp%2 nach exp%3, exp%4. Der interne Grafikcursor wird auf exp%3, exp%4 gesetzt.

POINT(exp%1, exp%2)
Diese Funktion gibt den internen Farbkode des Punktes exp%1, exp%2 zurück.

VR32#
Gibt TRUE# (#255) zurück, wenn die 16 KB VRAM Erweiterung beim MZ-800 eingebaut ist, sonst FALSE# (#0).

PLOT FRAME exp#
Dieser Befehl funktioniert nur bei eingebauter 16 KB VRAM Erweiterung und im 'CRT #1' und 'CRT #3' Modus. Durch exp# wird angegeben, ob im Frame #0 oder #1 gezeichnet werden soll (PLOT, LINE, POINT). Die Frames sind jeweils eigenständige Grafikbilder, von denen aber nur eines sichtbar ist.

CLS FRAME exp#
Dieser Befehl funktioniert nur bei eingebauter 16 KB VRAM Erweiterung und im 'CRT #1' und 'CRT #3' Modus. Der Frame exp# (#0, #1) wird gelöscht. Der normale CLS-Befehl löscht immer nur den gerade sichtbaren Frame. Ähnliches gilt für den PRINT-Befehl.

SHOW FRAME exp#
Dieser Befehl funktioniert nur bei eingebauter 16 KB VRAM Erweiterung und im 'CRT #1' und 'CRT #3' Modus. Durch exp# wird angegeben, ob der Frame #0 oder #1 gezeigt werden soll.

Durch geschickte Ausnutzung dieser FRAME-Befehle kann man bei bewegter Grafik einen sehr sauberen Bildwechsel erreichen.
Beispiel:

```
1 CRT #1:IF NOT VR32# THEN PRINT "NOT POSSIBLE":END
2 FOR y%=%0 TO %199 STEP %3
3   fr#=#1-fr#:CLS FRAME fr#:PLOT FRAME fr#
4   LINE %0,%0 TO %160,y%:LINE TO %319,%0
5   LINE %0,%199 TO %160,y%:LINE TO %319,%199
5   SHOW FRAME fr#
6 NEXT
```

DEFAULT exp$
exp$ ("c", "q" oder "r") setzt das aktuelle Gerät fest. Nur mit Hilfe dieses Befehls kann ein Gerät angesprochen werden. Kein Gerät ist voreingestellt.

DIR
Gibt das Directory aus.
Beispiel:
DEFAULT "q":DIR gibt das Directory der QD aus.

INIT
Formatiert das Gerät.

DELETE exp$
Löscht das File mit dem Namen exp$.

RENAME exp$1, exp$2
Bennent das File exp$1 in exp$2 um.
Beispiel:
DEFAULT "R":RENAME "P88","P89" bennent das File "P88" auf der Ramdisk in "P89" um.

SAVE exp$, exp#, exp1, exp2, exp3, exp4
Speichert einen Speicherbereich ab exp1 der Länge exp2 mit dem Namen exp$, dem Filetypen exp#, der Autostartadresse exp3 ab. Mit exp4 wird angegeben, wo das File beim Laden hingeladen werden soll. Der Filetyp #0 sollte nicht benutzt werden. #1 ist der Filetyp OBJ.
Beispiele:
SAVE "DTA",#200,MAXADR,100,0,0 speichert 100 Bytes ab MAXADR mit der Ladeadresse und der Autostaradresse 0 unter dem Namen "DTA" und dem Filetypen #200 auf dem voreingestelltem Gerät ab.
SAVE "VAR",#201,ADRS(z!(<0>)),4*80,0,0 speichert den gesammten Inhalt des Arrays z! unter dem Namen "VAR" und dem Filetypen #201 auf dem voreingestelltem Gerät ab. Das Array muß vorher schon mit 'DIM v!(<79>)' definiert worden sein. Mit 'LOAD "VAR", ADRS(v!(<0>))' wird der Inhalt des Arrays wieder geladen.

VERIFY
Prüft, ob das zuletzt abgespeicherte Programm korrekt aufgezeichnet wurde.

LOAD exp$ (,exp)
Lädt das File exp$ an die vorgegebene Ladeadresse oder an exp (wenn angegeben). Hierbei können Speicherbereiche zerstört werden, es wird nicht geprüft, ob der Compiler etc. überschrieben wird. Auch bei der CMT muß ein Name angegeben werden, der allerdings ignoriert wird, d.h. man gibt als Namen am besten einfach " " an.

LOADH exp$
Lädt den Header des Files exp$. Auch bei der CMT muß ein Name angegeben werden, der allerdings ignoriert wird, d.h. man gibt als Namen am besten einfach " " an. Jetzt stehen in folgenden Funktionen einige Informationen über das File bereit:
FNAME$: Der Name des Files
FTYPE#: Der Filetyp
FADR: Der Ladeadresse
FSIZE: Die Länge
FEXADR: Die Autostartadresse
Nach LOADH muß unbedingt sofort der Befehl LOADD oder ABORT folgen!

LOADD exp
Lädt die Daten des Files, dessen Header mit LOADH geladen wurde, an exp. Hierbei können Speicherbereiche zerstört werden.

ABORT
Bricht die Operation ab, die mit LOADH begonnen wurde (CMT und QD abschalten).

Beispiel zu LOADH, LOADD und ABORT:
Das File "FDATA" auf der QD soll an MAXADR geladen werden, wenn genügend Speicherplatz vorhanden ist.

```
DEFAULT "Q":LOADH "FDATA":IF FSIZE<=SIZE THEN LOADD MAXADR: PRINT
"OK": ELSE ABORT: PRINT "NO SPACE"
```

ZUSATZ FÜR XBC F 1.ØX AUF MZ-7ØØ/8ØØ

Letzte Überarbeitung des Zusatzanhanges am 27.Ø4.1987.

Dieser Zusatzanhang beschriebt die Abweichungen der FD-Version (F 1.ØX) von der Standardversion (S 1.ØX). Die allgemeine Anleitung und der Anhang für die Standardversion sind bis auf die hier beschriebenen Ausnahmen gültig.

Laden der Programme

XBC wird mit "F <CR>" vom Rommonitor aus geladen und gestartet.
Die Beispielprogramme werden mit 'xlf "name" <CR>' von XBC aus geladen (vorher altes Programm mit "d <CR>" löschen) und mit "j <CR>" gestartet.

Unterstützte Geräte

Die QD und CMT werden nicht unterstützt, dafür aber die FD. Es gibt eine neue Fehlermeldung: LCK (LOCK), d.h., das File is gelockt (=keine Änderung möglich).

Bei der FD (FD1-FD4) können folgende Operationen ausgeführt werden:
L (load), S (save), B (backup), D (directory), F (formatprogramm aufrufen), K (kill), R (rename), P (protect=lock), U (unprotect=unlock) und X (execute).

Der BACKUP-Befehl wird wie der SAVE-Befehl benutzt; im Gegensatz zu diesem wird dabei das genannte File überschreiben. Bei den LOCK- und UNLOCK-Befehlen wird der Filename angegeben (z.B. "XPF2 'TEST' <CR>": Auf der 2. FD wird das File "TEST" gelockt).
Das FORMAT-Programm befindet sich auf der XBC-Disk unter dem Namen "FORMAT", wird mit "xff <CR>" und dann "y <CR>" aufgerufen (XBC und eigene Programme (im Speicher) werden dabei zerstört) und bietet folgendes:

1. XBC booten.
2. BOOT-Programm generieren. Man wird nach dem Namen des (OBJ)Files und nach dem Laufwerk gefragt, auf dem es sich befindet. Dieses File wird dann beim BOOTen vom Rommonitor geladen und gestartet. Es erscheint nicht mehr im Inhaltsverzeichnis.
3. Ein File kopieren (Maximallänge etwa ØAEØØH). Man wird nach dem Laufwerk und Namen gefragt, dann nach dem Zeillaufwerk und Zielnamen. Wird nur <CR> beim Zielnamen eingegeben, wird der gleiche Name angenommen.
4. Formatieren. Man wird nach dem Laufwerk gefragt. Es werden keine Files kopiert.

Der Execute-Befehl (x) arbeitet genau wie der FORMAT-Befehl, man kann aber ein beliebiges Programm starten (Ladeadresse muß <= $18ØØ sein), indem man den Namen angibt (z.B. 'xxf2 "xcopy" <CR>': Das Programm "XCOPY" auf der 2. Disk wird geladen und gestartet).

Grundsätzlich werden alle Filenamensangaben intern in Großschrift umgewandelt, so daß im Directory nur groß geschriebene Namen erscheinen. Diese Files lassen sich durch Angabe eines klein geschriebenen Filenamens laden.

P.S.: Vor dem Eingeben des jsr-Befehls muß das einzeilige Programm
```
1 DEFAULT "F"
```
eingegeben, ausgeführt und dann gelöscht werden.

ANHANG FÜR XBC S 1.ØX AUF CPC 464

Letzte Überarbeitung des Anhanges am 22.4.1987.

INHALTSVERZEICHNIS

Ø ALLGEMEINES

Ø.1 Laden des Compilers

Kassette:
XBC wird mit 'run" (CR)' vom BASIC geladen und gestartet.

Ø.2 Unterstützte Geräte

c (CMT).

Bei der Kassette können folgende Operationen durchgeführt werden:
l/LOAD, s/SAVE und d/DIR.

Ø.3 Speicheraufteilung

XBC arbeitet immer mit 64 KB RAM. Ab $6Ø befindet sich der Quelltext in kodierter Form.

Warmstartadresse ist $73ØØ.

Ø.4 ASCII-Zeichensatz

Es wird der Standard-ASCII-Zeichensatz benutzt, den jeder normale Drucker sofort "versteht". Der Zeichensatz stimmt im Wesentlichen mit dem CPC-Zeichensatz überein, jedenfalls bis auf die Steuerzeichen und die Zeichen jenseits von 126 ($7E). Dieser Standardzeichensatz sieht so aus:

	ØØ	Ø1	Ø2	Ø3	Ø4	Ø5	Ø6	Ø7	Erläuterung:
ØØ				Ø	S	P	`	p	LE: Cursor left
Ø1			!	1	A	Q	a	q	RI: Cursor right
Ø2			"	2	B	R	b	r	BR: Break (CONTROL-C)
Ø3	BR	LE	#	3	C	S	c	s	UP: Cursor up
Ø4	RI		$	4	D	T	d	t	DN: Cursor down
Ø5	UP		%	5	E	U	e	u	BL: Bell
Ø6			&	6	F	V	f	v	TB: Tabulator
Ø7	BL		'	7	G	W	g	w	ES: ESCape
Ø8		DN	(	8	H	X	h	x	FF: Form feed (CLS)
Ø9	TB		)	9	I	Y	i	y	CR: Carridge return
ØA	DN	FF	*	:	J	Z	j	z	HM: Home
ØB		ES	+	;	K	Ä	k	ä	NL: Next line (wenn CURX$<>Ø: CR)
ØC	FF		,	<	L	Ö	l	ö	
ØD	CR	HM	-	=	M	Ü	m	ü	
ØE		NL	.	>	N	^	n	ß	Der Drucker versteht von diesen Steuer-
ØF			/	?	O	_	o		zeichen aber nur CR, ES, TB, FF.

Alle ASCII-Zeichen über $7E (126) werden auf dem Bildschirm als Leerzeichen

gedruckt.
Die obige Tabelle zeigt den deutschen Zeichensatz. Da im US-Zeichensatz gearbeitet wird, sehen folgende Zeichen anders aus:
$4Ø: Klammeraffe
$5B: Eckige Klammer auf
$5C: Backslash: umgekehrter Divisionstrich ($2F)
$5D: Eckige Klammer zu
$7B: Geschweifte Klammer auf
$7C: Senkrechter Strich
$7D: Geschweifte Klammer zu
$7E: Geschlängeltes Minuszeichen

Ø.5 Tastaturbelegung

TAB:	^I (8-er Tabulator; (CHR$(9)).
CAPS LOCK:	Schaltet zwischen Groß- und Kleinschrift um.
ESC:	^C ((BREAK); CHR$(3)).
SHIFT+ESC:	^C ((BREAK); CHR$(3)).
ENTER:	^M (CR; CHR$(13)).
COPY:	^V (Insert on/off; CHR$(22)).
SHIFT+COPY:	^V (Insert on/off; CHR$(22)).
CLR:	^G (Delete; CHR$(7)).
SHIFT+CLR:	^G (Delete; CHR$(7)).
DEL:	^Q (Backspace; CHR$(17)).
SHIFT+DEL:	^Q (Backspace; CHR$(17)).
CURSOR UP:	^E (CHR$(5)).
SHIFT+CURSOR UP:	^W (CHR$(23)).
CURSOR DOWN:	^X (CHR$(24)).
SHIFT+CURSOR DOWN:	^Z (CHR$(26)).
CURSOR RIGHT:	^D (CHR$(4)).
SHIFT+CURSOR RIGHT:	^F (CHR$(6)).
CURSOR LEFT:	^S (CHR$(19)).
SHIFT+CURSOR LEFT:	^A (CHR$(1)).

Ø.6 Programmabbruch

Ein Programm kann nur abgebrochen werden, wenn auf Abbruch ((BREAK)) geprüft wird (siehe 'TEST BREAK' und 'OPTION'). Wenn auf die Taste 'ESC' gedrückt wird, wartet der Cursor und das Programm ist angehalten. Wenn jetzt eine beliebige Taste außer 'ESC' gedrückt wird, wird das Programm fortgesetzt. Wenn 'ESC' ((BREAK)) gedrückt wird, wird das Programm abgebrochen.

1 BEFEHLE UND FUNKTIONEN

1.1 Befehle vom Editor aus

v4 ;Videomode with 40 characters/line
Schaltet in den Modus mit 40 Zeichen/Zeile (entspricht 'CRT #1').

v8 ;Videomode with 80 characters/line
Schaltet in Modus mit 80 Zeichen/Zeile (entspricht 'CRT #2').

1.2 Allgemeine Befehle und Funktionen

TIME!
Die Systemfunktion TIME! entspricht der BASIC-Funktion TIME.

1.3 I/O-Befehle und Funktionen

JOY#(exp#)
Fragt die Tastatur (exp#=#0) oder den Joystick (exp#=#1) ab. Folgender Wert wird zurückgegeben:
Nichts: #0.
Links: #1; Links, hoch: #2; Hoch: #3 ...

FIRE#(exp#)
Siehe JOY#(exp#). Es wird entweder TRUE# (#255) oder FALSE# (#0) zurückgegeben, je nachdem, ob die Leer-/Feuertaste gedrückt ist.

SOUND exp#1, exp, exp#2, exp#3
Der Tongenerator exp#1 (#0-#2) wird auf die Frequenz exp (0-4095) und auf die Lautstärke exp#2 (#0-#15) gesetzt. Wenn exp#2=#0, wird der Tongenerator unhörbar (quasi abgeschaltet). Wenn die Lautstärke auf #16 eingestellt wird, wird die Lautstärke vom ENV-Generator übernommen. Wenn exp#3=#0, wird kein Rauschen des Rauschgenerators dazugemischt; dies geschieht, wenn exp#3=#1. Nur Rauschen erhält man, wenn man exp#3=#1 und exp=0 setzt. Die Hz-Frequenz errechnet sich folgendermaßen:
Hz:=62500/exp
Beispiel:
SOUND #1,62500/880,#8,#0 stellt den Tongenerator #1 auf 880 Hz und mittlere Lautstärke.

NOISE exp#
Setzt den Rauschgenerator auf exp# (#0-#31). Je niedriger exp# ist, desto höher ist die Rauschbasisfrequenz. Der Rauschgenerator selbst ist unhörbar. Das Rauschen kann nur zu einem der 3 Kanäle dazugemischt werden (siehe SOUND).

ENV exp#, exp
Stellt den ENV-Generator (Lautstärkengeber) auf den Modus exp# (#0-#15) und den Takt exp (0-65535). Je niedriger exp ist, desto schneller ändert sich die Lautstärke. Wenn beim SOUND-Befehl die Lautstärke auf 16 gestellt wird, wird die Lautstärke vom ENV-Generator übernommen.
Beispiel:
NOISE 0:SOUND 0,0,16,1:LOOP:GETC a$:ENV 0,1000:END LOOP erzeugt nach jedem Tastendruck eine Art Schußgeräusch.

Hinweis zu PRINT und INPUT
Die Bits 3 und 4 beim 'PRINT (<exp>) ...'-Befehl haben folgende Funktionen:
Bit 3: Die PRINT-Routine des Betriebssystems wird benutzt.
Beispiel:
PRINT (<8>);CHR$(#18); invertiert die aktuellen Farben.
Bit 4: Der Kassettenrekorder (CMT) wird angesprochen.
Beim 'INPUT (<exp>) ...'-Befehl können nur die Bits 0 und 4 benutzt werden, aber nicht gleichzeitig. Die Bedeutung ist genau wie bei PRINT.
Beispiel:
ROPEN "TXT1":INPUT (<16>);a$ öffnet ein File namens "TXT1" und liest den ersten String aus dieser Datei in a$.

CRT exp#
exp#=#0-#2. Entspricht dem BASIC-Befehl MODE.

COLOR exp#1, exp#2
Stellt die Vorder- und Hintergrundfarben ein (exp#1, exp#2=#0-#15).

COLOR exp#
Stellt die Vordergrundfarbe ein (besonders für Grafik) (exp#=#0-#15).

BORDER exp#1, exp#2
Setzt die Randfarbe auf exp#1, exp#2 (exp#1, exp#2=#0-#31).

INK exp#1, exp#2, exp#3
Weist einem internem Farbkode (exp#1) zwei absolute Farbnummern (exp#2, exp#3=#0-#31) zu. Der interne Farbkode #0 ist für den Hintergrund reseviert.
Beispiel:
PAL #0,#0,#27 weist dem Hintergrund die absoluten Farben 0 (sehr dunkel) und 27 (sehr hell) zu.

SETORG exp%1, exp%2
Setzt den (0;0)-Punkt auf die absolute Koordinate exp%1, exp%2.
Beispiel:
SETORG %320,%200 setzt den (0;0)-Punkt, wenn man 'CRT #1' ausgeführt hat, in die Mitte des Bildschirmes.

GMODE exp#
Setzt die Überlagerungsbetriebsart fest (gültig for PLOT und LINE):
exp#=#0: Punkt setzen
exp#=#1: XOR
exp#=#2: AND
exp#=#3: OR

GMOVE exp%1, exp%2
Setzt den internen Grafikcursor auf die Koordinate exp%1, exp%2.

PLOT exp%1, exp%2
Setzt einen Punkt an die Koordinate exp%1, exp%2. Der interne Grafikcursor wird auf exp%1, exp%2 gesetzt.

LINE TO exp%1, exp%2
Zieht eine Linie vom internen Grafikcursor nach exp%1, exp%2. Der interne Grafikcursor wird auf exp%1, exp%2 gesetzt.

LINE exp%1, exp%2 TO exp%3, exp%4
Zieht eine Linie von exp%1, exp%2 nach exp%3, exp%4. Der interne Grafikcursor wird auf exp%3, exp%4 gesetzt.

POINT(exp%1, exp%2)
Diese Funktion gibt den internen Farbkode des Punktes exp%1, exp%2 zurück.

DEFAULT exp$
exp$ ("c") setzt das aktuelle Gerät fest. Nur mit Hilfe dieses Befehls kann ein Gerät angesprochen werden. Kein Gerät ist voreingestellt.

DIR
Gibt das Directory aus.
Beispiel:
DEFAULT "c":DIR gibt das Directory der CMT aus.

SAVE exp$, exp#, exp1, exp2, exp3, exp4
Speichert einen Speicherbereich ab exp1 der Länge exp2 mit dem Namen exp$, dem Filetypen exp#, der Autostartadresse exp3 ab. Mit exp4 wird angegeben, wo das File beim Laden hingeladen werden soll (funktioniert nicht beim CPC: exp4 auf exp1 setzen). Beim Filetypen sollte das Bit Ø auf Ø stehen. #2 ist der ML-Filetyp.
Beispiele:
SAVE "DTA",#2ØØ,MAXADR,1ØØ,Ø,Ø speichert 1ØØ Bytes ab MAXADR mit der Autostaradresse Ø unter dem Namen "DTA" und dem Filetypen #2ØØ auf dem voreingestelltem Gerät (beim CPC 464 nur CMT) ab.
SAVE "VAR",#2ØØ,ADRS(z!(Ø)),4*8Ø,Ø,Ø speichert den gesammten Inhalt des Arrays z! unter dem Namen "VAR" und dem Filetypen #2ØØ auf dem voreingestelltem Gerät ab. Das Array muß vorher schon mit 'DIM v!(79)' definiert worden sein. Mit 'LOAD "VAR", ADRS(v!(Ø))' wird der Inhalt des Arrays wieder geladen.

LOAD exp$ (,exp)
Lädt das File exp$ an die vorgegebene Ladeadresse oder an exp (wenn angegeben). Hierbei können Speicherbereiche zerstört werden, es wird nicht geprüft, ob der Compiler etc. überschrieben wird. Auch bei der CMT muß ein Name angegeben werden, der allerdings ignoriert wird, d.h. man gibt als Namen am besten einfach " " an.

LOADH exp$
Lädt den Header des Files exp$. Auch bei der CMT muß ein Name angegeben werden, der allerdings ignoriert wird, d.h. man gibt als Namen am besten einfach " " an. Jetzt stehen in folgenden Funktionen einige Informationen über das File bereit:
FNAME$: Der Name des Files
FTYPE#: Der Filetyp
FADR: Der Ladeadresse
FSIZE: Die Länge
FEXADR: Die Autostartadresse

Nach LOADH muß unbedingt sofort der Befehl LOADD oder ABORT folgen!

LOADD exp
Lädt die Daten des Files, dessen Header mit LOADH geladen wurde, an exp. Hierbei können Speicherbereiche zerstört werden.

ABORT
Bricht die Operation ab, die mit LOADH begonnen wurde (CMT abschalten).

Beispiel zu LOADH, LOADD und ABORT:
Das File "FDATA" auf der CMT soll an MAXADR geladen werden, wenn genügend Speicherplatz vorhanden ist.
DEFAULT "C":LOADH "FDATA":IF FSIZE<=SIZE THEN LOADD MAXADR: PRINT "OK": ELSE ABORT: PRINT "NO SPACE"

WOPEN exp$
Öffnet ein sequentielles File zum Schreiben mit dem Namen exp$ auf der CMT. Die Daten werden mit PRINT (<16>) ... geschrieben.

ROPEN exp$
Öffnet ein sequentielles File zum Lesen mit dem Namen exp$ auf der CMT. Die Daten werden mit INPUT (<16>) ... gelesen, wobei hinter INPUT nur eine Variable stehen sollte.

CLOSE
Schließt ein File, das mit WOPEN oder ROPEN geöffnet wurde.

EOF#
Diese Systemfunktion zeigt das Ende der sequentiellen Datei an, die gerade gelesen wird.
Beispiel:
ROPEN "??":WHILE NOT EOF#:INPUT (<16>);a$:PRINT a$:WEND zeigt den gesammten Inhalt der Datei "??" an.

Anmerkung zu sequentiellen Files

Sequentielle Files sind dazu gedacht, Texte zu speichern (Zahlen werden automatisch in Texte umgewandelt). Es dürfen auf keinen Fall die Steuerzeichen $D, $A und $1A mit PRINT ausgegeben werden, da es sonst zu Fehlfunktionen kommen kann.
$1A wird an jede Datei am Ende angehängt, um das Ende zu kennzeichnen. Da der BASIC-Interpreter dies nicht tut (obwohl dies ein Standard ist), erkennt XBC auch das physikalische Ende beim Lesen. Die Kodes $D und $A werden als Wagenrücklaufkodes benutzt. Sie werden ausgegeben, wenn das letzte Zeichen beim PRINT-Befehl nicht ";" oder "," ist. Das Komma wird bei der CMT nicht als Tabulator aufgefaßt, sondern als Wagenrücklauf. Wenn XBC beim Lesen einer Datei auf $D trifft, wird dies als Zeilen/Stringende aufgefaßt und das nächste Zeichen überlesen, da sich dort (hoffentlich) $A befindet.
Beispiel:
WOPEN "tt":PRINT (<16>);"AB","CC";:CLOSE schreibt die Hexcodes 41 42 ØD ØA 43 43 1A auf die CMT unter dem Namen "tt".
lesen kann man das ganze mit
ROPEN "tt":INPUT (<16>);a$:INPUT (<16>);b$:PRINT a$:PRINT b$:CLOSE

SPEED exp#
exp#=#Ø-#1. Entspricht dem BASIC-Befehl SPEED WRITE.

1.4 Maschinenbefehle

SCALL exp

Ruft eine Maschinenroutine ab exp auf. Im Gegensatz zum normalen CALL-Befehl werden hierbei erstens keine Register zerstört (die z.B. mit Hilfe des ML-Befehls einen Wert bekommen haben) und zweitens werden die Zweitregister mit den aktuellen Werten geladen, so daß auch Betriebssystemroutinen aufgerufen werden können. SCALL ist allerdings wesentlich langsammer als CALL.

Anmerkung zu maschinennaher Programmierung

Beim CPC wird 300 mal pro Sekunde ein Interrupt ausgelöst (RST 38H). Die Routine, die auf die Interrupts reagiert, setzt unverständlicherweise bestimmte Werte (Bankswitching, CRT-Mode) in den Registern AF' und BC' voraus. Da XBC diese Register (und HL', DE') selbst benötigt, wurde eine Routine vor die eigentliche Interruptroutine geschrieben und eine dahinter, die diese Probleme beseitigen. Wenn nun eine Betriebssystemroutine im ROM aufgerufen werden soll, müssen die Register AF' und BC' mit den nötigen Werten geladen werden, da auch im ROM Interrupts auftreten können, die nicht "behandelt" werden. Genau dafür sorgt der Befehl SCALL. Man kann mit Hilfe des ML-Befehls den Registern AF, HL, DE und BC irgendwelche Werte zuweisen und dann die Routine mit SCALL aufrufen. Bei CALL dagegen werden einige Register zerstört (meist nur HL) und AF' und BC' haben irgendwelche Werte, die beim Interrupt im ROM einen Absturz auslösen könnten.

Das IX-Register darf in keiner Subroutine zerstört werden, da XBC es intern verwendet.

Der RST 38H wird von XBC für die (BREAK)-Überprüfung benutzt, der RST 8H wird beim Kompilieren benutzt, so daß ein RST 8H oder JP 0 zum Rücksetzen des Computers nicht unbedingt zum gewünschten Effekt führt.