



**Commodore
Sachbuch**

Florian Müller

CC64 **für Insider**

- ★ Ausführlich dokumentiertes ROM-Listing
- ★ System-Handbuch ★ Memory Map

Florian Müller

**Commodore
Sachbuch**



c64 für Insider

- ★ Ausführlich dokumentiertes ROM-Listing
- ★ Memory Map
- ★ System-Handbuch

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Müller, Florian:

C 64 für Insider : ausführl. dokumentiertes ROM-Listing,
Memory Map, System-Handbuch / Florian Müller. –
Haar bei München : Markt-u.-Technik-Verl., 1988
(Commodore-Sachbuch)
ISBN 3-89090-481-5

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.

Trotzdem können Fehler nicht vollständig ausgeschlossen werden.

Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische

Verantwortung noch irgendeine Haftung übernehmen.

Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.

Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

C-Commodore® ist ein eingetragenes Warenzeichen der Commodore Büromaschinen GmbH, Frankfurt.

»Commodore 64« ist eine Produktbezeichnung der Commodore Büromaschinen GmbH, Frankfurt,
die ebenso wie der Name »Commodore« Schutzrechte genießt.

Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis der Rechteinhaberin.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
91 90 89 88

ISBN 3-89090-481-5

© 1988 by Markt & Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany

Alle Rechte vorbehalten

Einbandgestaltung: Grafikdesign Heinz Rauner

Druck: Jantsch, Günzburg

Printed in Germany

Inhaltsverzeichnis

Vorwort	7
----------------	---

1 ROM-Listing	9
----------------------	---

2 So verwendet man das ROM-Listing	289
---	-----

2.1 Symbole	290
-------------	-----

2.1.1 Geschweifte Klammern	290
----------------------------	-----

2.1.2 Pfeile	290
--------------	-----

2.1.3 Waagrechte Linien	290
-------------------------	-----

2.2 Aufbau des Disassemblerlistings	290
-------------------------------------	-----

2.2.1 Disassemblerformat	290
--------------------------	-----

2.2.2 Weitere Informationen im Disassemblerlisting	291
--	-----

2.2.2.1 Andere Zahlenformate	291
------------------------------	-----

2.2.2.2 Low-High-Format	291
-------------------------	-----

2.2.2.3 Zeropage-Adressen	291
---------------------------	-----

2.2.2.4 ASCII-Codes	291
---------------------	-----

2.2.2.5 Anführungszeichen hinter Mnemonics	291
--	-----

2.2.2.6 Der Bit-Trick	292
-----------------------	-----

2.3 Aufbau der Kommentare	292
---------------------------	-----

2.4 Cross-Reference	292
---------------------	-----

3 Die Firmware des C64	381
-------------------------------	-----

3.1 Grundbegriffe »Hardware«, »Software« und »Firmware«	381
--	-----

3.2 Begriffe »Betriebssystem«, »Interpreter« und »Compiler«	381
--	-----

3.3 Das Betriebssystem (Kernal)	382
---------------------------------	-----

3.3.1 Die Kernal-Sprungtabelle	382
--------------------------------	-----

3.3.2 Die IRQ-Routinen	383
------------------------	-----

3.3.3 Die Funktionsweise der universellen Routinen	384
--	-----

3.3.4 Die Initialisierung (Reset)	384
-----------------------------------	-----

3.3.5 Die Fehlermeldungen und ihre Übermittlung	384
---	-----

3.3.6 Das Statusbyte (ST)	384
---------------------------	-----

3.3.7 Die Steuermeldungen	385
---------------------------	-----

3.3.8 Die Filetabelle	385
-----------------------	-----

3.3.9 I/O-Beispiel: Drucker-Ausgabe	385
-------------------------------------	-----

3.3.10 Steuerzeichen	387
----------------------	-----

3.4 Der Basic-Interpreter	387
---------------------------	-----

3.4.1 Die Initialisierung	388
---------------------------	-----

3.4.2 Der Aufbau von Basic-Programmen im Speicher	388
---	-----

3.4.2.1 Ober- und Untergrenze des Basic-Speichers	388
---	-----

3.4.2.2 Nullbyte vor dem Programmbeginn	388
---	-----

3.4.2.3 Überblick über einen Zeileneintrag	388
--	-----

3.4.2.4 Linkpointer	389
---------------------	-----

3.4.2.5 Zeilennummer	389
----------------------	-----

3.4.2.6 Zeileninhalt	389
----------------------	-----

3.4.2.7 Programmende	390
----------------------	-----

3.4.3 Der Aufbau von Basic-Variablen im Speicher	390
--	-----

3.4.3.1 Zeiger für den Variablenbereich	390
---	-----

3.4.3.2 Die Gliederung in Array- und Variableneinträge	391
--	-----

3.4.3.3 Eintrag einer Fließkomma-Variablen	391
--	-----

3.4.3.4 Eintrag einer Integer-Variablen	392
---	-----

3.4.3.5	Eintrag einer String-Variablen	392
3.4.3.6	Aufbau von indizierten Variablen (Arrays)	393
3.4.4	Der Editor	394
3.4.5	Die Interpreterschleife	394
3.4.6	Die Garbage-Collection	394
3.4.7	Die Parameterauswertung	396
3.4.7.1	Die CHRGET/CHRGOT-Routine	396
3.4.7.2	Die FRMEVL-Routine	396
3.4.7.3	Sonderfall für numerische Parameter: Basic-Zeilenummer	396
3.4.7.4	Auswertung numerischer Parameter innerhalb eingeschränkter Bereiche	396
3.4.7.5	Syntaktische Erfordernisse	396
3.4.8	Das Fließkommaformat	397
3.4.8.1	Zahlenformate	397
3.4.8.2	Mantisse und Exponent	397
3.4.8.3	Beispiel zur Berechnung der Mantissenbytes	398
3.4.8.4	FLPT- und MFLPT-Format	398
3.4.9	Polynome und das Horner-Schema	399
3.4.10	Fehler- und Steuermeldungen	399
3.4.11	Der Stapel als Hilfsmittel des Interpreters	400

3.5	Verknüpfungsstellen zwischen Betriebssystem und Basic-Interpreter	401
3.5.1	Speicherbereiche von Interpreter und Betriebssystem	401
3.5.2	Basic-Kernal-Aufrufe	402
3.5.3	Basic-ROM-Vektoren	402
4	Die ROM-Routinen im Detail	403
5	Die ROM-Routinen im Überblick	495
6	Memory Map	501
7	Ausblick: GEOS und C64-verwandte Betriebssysteme	509
7.1	GEOS – Graphical Environment Operating System	509
7.2	VC20 – Der Vorläufer	511
7.3	C16, C116 und Plus/4 – Aus der Art geschlagen	511
7.4	C128 – Der Nachfolger	512
	Stichwortverzeichnis	513
	Hinweise auf weitere Markt&Technik-Produkte	517

Vorwort

Liebe Leser,

gestatten Sie mir zuerst eine Vorbemerkung. Das Vorwort mag vielleicht streckenweise wie ein Werbetext wirken, aber die aufgestellten Behauptungen entsprechen durchaus den Tatsachen, wie Sie noch sehen werden. Und da Sie dieses Buch offensichtlich bereits erworben haben, dürfen Sie auch – ohne Hintergedanken meinerseits – erfahren, worüber Sie nun verfügen.

Daß dieses Buch weit aus dem üblichen Rahmen fällt, zeigt allein schon das äußere Format (die geplagten Buchhändler, die dieses Buch in ihre Regale einordnen wollen, mögen es dem Verlag und mir verzeihen). Dies ist kein Werbegag, sondern eine recht kostspielige Angelegenheit, um Ihnen äußerst viele Informationen in möglichst ansprechender Form auf selbem Raum anzubieten. Wenn Sie einen Blick in Kapitel 1, das ROM-Listing (»Weltrekord« von bis zu 120 Zeichen pro Zeile!) werfen, fällt es Ihnen sicher leicht zu verstehen, warum dieses Buch nicht die Standardgröße hat:

Es muß **breiter** sein, weil es mehr in die **Tiefe** geht!

Das vorliegende ROM-Listing wurde über einen Zeitraum von etwa 20 Monaten (!), nämlich von November 1985 bis Juli 1987, erstellt, dabei sogar einmal von Grund auf neu verfaßt und immer wieder verbessert; es hat also eine bewegte Geschichte hinter sich, in der es nach und nach zu dem gereift ist, was man es jetzt ohne Übertreibung nennen kann:

Das gründlichste und praxisbezogenste ROM-Listing, das je zum C64 oder einem anderen Computer erstellt wurde!

Dem großen Einsatz des Markt&Technik Verlags ist es nun zu verdanken, daß mit revolutionären grafischen Methoden rundum durch zahlreiche Symbole (Raster, Verzweigungspfeile, geschweifte Klammern über mehrere Zeilen hinweg, 120 Zeichen pro Zeile) eine »optimale Optik« erzielt wurde, wie sie bei anderen ROM-Listings nicht im entferntesten vorliegt.

Wie Sie am restlichen Text merken werden, liegt es mir ganz und gar nicht, große Sprüche zu klopfen, aber um Ihnen die Vorzüge dieses wahrhaft sensationellen ROM-Listings zu schildern, wäre Understatement fehl am Platz. Ich würde gerne auch sachliche Vergleiche mit anderen C64-ROM-Listings anstellen, doch ist dies leider aufgrund der Wettbewerbsgesetze nicht erlaubt. Nur ein Satz

zu diesem Thema: Wenn jemand bereits ein ROM-Listing besitzt, sollte er lieber auf dieses Buch umwechseln; es wird sich in einer besseren Beherrschung des C64 und größerem Komfort bei der Programmentwicklung bezahlt machen.

Zeigen Sie doch dieses Buch einmal einem Bekannten, der bereits ein anderes ROM-Listing besitzt; er wird vor Neid erblassen. . . . Das ROM-Listing ist übrigens deshalb schon in Kapitel 1, damit das Finden einer bestimmten Adresse so schnell wie möglich vor sich geht.

Würde sich dieses Werk jedoch auf das ROM-Listing beschränken, wäre der Titel »C64: ROM-Listing« angebrachter gewesen. »C64 für Insider« heißt aber auch, daß Sie ein umfassendes Informationswerk zur Verfügung haben. Darunter fallen beispielsweise folgende Glanzpunkte:

- Cross-Reference

Damit wird endlich Schluß gemacht mit vagen Behauptungen wie »Diese Adresse wird normalerweise nicht benötigt« oder »eine selten aufgerufene Routine«, die in der Regel nichts weiter als leeres Geschwafel zum Ausdruck der Unwissenheit ihrer Verfasser sind, um es so deutlich zu formulieren.

Klipp und klar erfahren Sie in Kapitel 2.4, welche Zusammenhänge zwischen den einzelnen Adressen und Routinen im C64-Speicher bestehen.

Daß eine solche Liste bislang noch nicht für den C64 veröffentlicht wurde, ist den bisherigen Autoren sogenannter Systemhandbücher anzulasten.

- Erklärung der wichtigen Begriffe und Systemüberblick

Taucht ein unbekannter Begriff auf, so wird er in Kapitel 3 nicht nur definiert, sondern Sie erfahren auch vieles über den grundsätzlichen Aufbau von Betriebssystem und Basic-Interpreter. Von diesem Überblick ausgehend, können Sie auch die Details besser verstehen. Wie schon ein griechischer Philosoph sagte: »Wenn man das Ganze erkennt, kann man auch die Teile begreifen.«

- Dokumentation der Routinen

Kapitel 4 behandelt als parallele Ergänzung zum ROM-Listing noch einmal jede ROM-Routine. Es gibt sowohl Beispiele zur

Anwendung als auch Erklärungen zu bestimmten Programm-besonderheiten. Vor allem werden Ihnen die vielen Fluß-diagramme und sonstigen Abbildungen zum größtmöglichen Verständnis verhelfen.

Eine solche Routinenbeschreibung in dieser Ausführlichkeit ist ebenfalls eine echte Neuheit.

Der anschließende Routinenüberblick (Kapitel 5) hingegen ist wieder Standard.

– Memory Map

Eine Beschreibung aller Adressen des C64-Speichers bietet Kapitel 6. In Verbindung mit der Cross-Reference und dem ROM-Listing können Sie damit alles über die Verwendungsweise einer bestimmten Adresse erfahren.

Sie sehen also am bisherigen Überblick, daß der beträchtliche Umfang dieses Buches keineswegs durch Seitenfüllerei zustande gekommen ist. Im Gegenteil, es platzt auch inhaltlich aus allen Nähten. Sie verfügen somit über ein umfassendes Werk zum C64, das bestimmt **jedem** C64-Insider sowohl kurz- als auch mittel- und langfristig etwas zu bieten vermag – auch wenn er schon jedes andere C64-Buch gelesen haben sollte.

Obwohl ich sehr stolz darauf bin, dieses Buch alleine verfaßt zu haben (normalerweise werden solche Bücher von Autorentams, in denen jedes Mitglied sein eigenes Spezialgebiet behandelt, verfaßt), wäre dies wohl nicht ohne die große Unterstützung von allen Seiten

möglich gewesen, für die ich mich an dieser Stelle herzlichst bedanken möchte:

Klaus Schrödl und Roland Fieger von der Redaktion 64'er haben mir wichtige Programme zur Verfügung gestellt, zum Beispiel für die Cross-Reference.

Thomas Fenzl hat die mühevollen Arbeit auf sich genommen, über den Text dieses Buches Korrektur zu lesen.

Ganz herzlichen Dank möchte ich an alle mit diesem Projekt befaßten Mitarbeiter des Markt&Technik Buchverlages richten. Besonders dankbar bin ich dabei Georg Weiherer, der dieses Projekt in der Anfangsphase betreut hat, und Frank Hergenröder, der es in der weiteren Bearbeitung übernahm und durch die problematische Weiterverarbeitung des ROM-Listings leider sehr stark in Anspruch genommen werden mußte.

Beide haben sich weit über das übliche Maß dafür eingesetzt, daß dieses »extreme« Buch trotz vieler technischer Probleme möglich wurde.

Nun bleibt mir noch Ihnen, lieber Leser, viel Spaß und Erfolg mit diesem Buch zu wünschen. Es würde mir auch große Freude bereiten, wenn Sie mir Ihre Meinung zu diesem Buch mitteilen könnten; Sie erreichen mich schriftlich über den Markt&Technik Verlag AG (Hans-Pinsel-Str. 2, 8013 Haar bei München).

Der Autor

Kapitel 1

ROM-Listing

Das ROM-Listing ist das wichtigste Werkzeug eines ernsthaften Programmierers, denn es dokumentiert die eigentlichen Grundlagen der C64-Programmierung: Basic-Interpreter und Betriebssystem.

Der besondere Wert des ROM-Listings auf den folgenden fast 300 Seiten liegt nicht nur in der großen Ausführlichkeit (für jede Kommentarzeile wurden 120 Zeichen benötigt, manche Kommentare reichen über mehrere Zeilen), sondern auch in der optischen Verarbeitung.

Die grafische Aufbereitung erleichtert die praktische Arbeit in erheblichem Maße, so daß das Auffinden bestimmter Informationen optimal unterstützt wird.

In Kapitel 2 werden alle Besonderheiten in der äußeren Form des ROM-Listings erläutert. Beachten Sie auch die umfangreiche »Cross-Reference«, die gewissermaßen eine Analyse des ROM-Listings darstellt.

Die einzelnen Kommentarzeilen im ROM-Listing sind bewußt so aufgebaut, daß Sie diese aus zwei Richtungen lesen können:

1. Von links nach rechts gelangen Sie von der Adresse und den Befehlscodes (Opcodes) über die mnemonische Darstellung (Disassembler-Listing) zum Zeilenkommentar; ganz rechts steht unter Umständen ein zeilenübergreifender Hinweis.
Allgemein gesprochen, gelangt man von links nach rechts jeweils eine Dokumentationsebene höher und bewegt sich mehr von den Maschinencodes weg.
2. Von rechts nach links arbeitet man sich immer mehr ins Detail vor: von den allgemeinen zu den speziellen Kommentaren, dann zu den Mnemonics und zuletzt zu Adreßangabe und Befehlscodes.

Es bleibt Ihnen in jeder Situation überlassen, auf welcher Seite Sie beginnen. Im Laufe der Zeit werden Sie sich von selbst daran gewöhnen, diese Freiheit sinnvoll zu nutzen.

; ROM-Vektoren für Basic-Kaltstart und Basic-NMI

:a000 94 e3 \$e394	ROM-Vektor für Basic-2.0-Kaltstart; zeigt auf \$e394; wird bei \$fcff berücksichtigt
:a002 7b e3 \$e37b	ROM-Vektor für Basic-2.0-NMI; zeigt auf \$e37b; wird bei \$fe6f berücksichtigt (wenn <RESTORE> ausgelöst wird, erfolgt ein NMI; ist gleichzeitig <RUN/STOP> gedrückt, springt die NMI-Routine über diesen Vektor)

; ROM-Kennung

:a004 43 42 4d 42 41 53 49 43 c b m b a s i c	Text "cbmbasic" im ASCII-Code; sogenannte "ROM-Kennung"; hat keinerlei Einfluß auf die Funktionsweise des Interpreters: rein informativer Charakter
--	---

; ROM-Tabelle der Adressen der Routinen zu den Basic-Kommandos; wird bei \$a7f9 und \$a7fd (GONE-Routine) ausgelesen (in Reihenfolge der Tokens)

:a00c 30 a8	Adresse der Routine zum Basic-Befehl END:	\$a831 (im ROM um 1 dekrementiert)
:a00e 41 a7	Adresse der Routine zum Basic-Befehl FOR:	\$a742 (im ROM um 1 dekrementiert)
:a010 ld ad	Adresse der Routine zum Basic-Befehl NEXT:	\$ad1e (im ROM um 1 dekrementiert)
:a012 f7 a8	Adresse der Routine zum Basic-Befehl DATA:	\$a8f8 (im ROM um 1 dekrementiert)
:a014 a4 ab	Adresse der Routine zum Basic-Befehl INPUT#:	\$aba5 (im ROM um 1 dekrementiert)
:a016 be ab	Adresse der Routine zum Basic-Befehl INPUT:	\$abbf (im ROM um 1 dekrementiert)
:a018 80 b0	Adresse der Routine zum Basic-Befehl DIM:	\$b081 (im ROM um 1 dekrementiert)
:a01a 05 ac	Adresse der Routine zum Basic-Befehl READ:	\$ac06 (im ROM um 1 dekrementiert)
:a01c a4 a9	Adresse der Routine zum Basic-Befehl LET:	\$a9a5 (im ROM um 1 dekrementiert)
:a01e 9f a8	Adresse der Routine zum Basic-Befehl GOTO:	\$a8a0 (im ROM um 1 dekrementiert)
:a020 70 a8	Adresse der Routine zum Basic-Befehl RUN:	\$a871 (im ROM um 1 dekrementiert)
:a022 27 a9	Adresse der Routine zum Basic-Befehl IF:	\$a928 (im ROM um 1 dekrementiert)
:a024 1c a8	Adresse der Routine zum Basic-Befehl RESTORE:	\$a81d (im ROM um 1 dekrementiert)
:a026 82 a8	Adresse der Routine zum Basic-Befehl GOSUB:	\$a883 (im ROM um 1 dekrementiert)
:a028 d1 a8	Adresse der Routine zum Basic-Befehl RETURN:	\$a8d2 (im ROM um 1 dekrementiert)
:a02a 3a a9	Adresse der Routine zum Basic-Befehl REM:	\$a93b (im ROM um 1 dekrementiert)
:a02c 2e a8	Adresse der Routine zum Basic-Befehl STOP:	\$a82f (im ROM um 1 dekrementiert)
:a02e 4a a9	Adresse der Routine zum Basic-Befehl ON:	\$a94b (im ROM um 1 dekrementiert)
:a030 2c b8	Adresse der Routine zum Basic-Befehl WAIT:	\$b82d (im ROM um 1 dekrementiert)
:a032 67 e1	Adresse der Routine zum Basic-Befehl LOAD:	\$e168 (im ROM um 1 dekrementiert)
:a034 55 e1	Adresse der Routine zum Basic-Befehl SAVE:	\$e156 (im ROM um 1 dekrementiert)
:a036 64 e1	Adresse der Routine zum Basic-Befehl VERIFY:	\$e165 (im ROM um 1 dekrementiert)

:a038 b2 b3	Adresse der Routine zum Basic-Befehl DEF:	\$b3b3 (im ROM um 1 dekrementiert)
:a03a 23 b8	Adresse der Routine zum Basic-Befehl POKE:	\$b824 (im ROM um 1 dekrementiert)
:a03c 7f aa	Adresse der Routine zum Basic-Befehl PRINT#:	\$aa80 (im ROM um 1 dekrementiert)
:a03e 9f aa	Adresse der Routine zum Basic-Befehl PRINT:	\$aaa0 (im ROM um 1 dekrementiert)
:a040 56 a8	Adresse der Routine zum Basic-Befehl CONT:	\$a857 (im ROM um 1 dekrementiert)
:a042 9b a6	Adresse der Routine zum Basic-Befehl LIST:	\$a69c (im ROM um 1 dekrementiert)
:a044 5d a6	Adresse der Routine zum Basic-Befehl CLR:	\$a65e (im ROM um 1 dekrementiert)
:a046 85 aa	Adresse der Routine zum Basic-Befehl CMD:	\$aa86 (im ROM um 1 dekrementiert)
:a048 29 e1	Adresse der Routine zum Basic-Befehl SYS:	\$e12a (im ROM um 1 dekrementiert)
:a04a bd e1	Adresse der Routine zum Basic-Befehl OPEN:	\$elbe (im ROM um 1 dekrementiert)
:a04c c6 e1	Adresse der Routine zum Basic-Befehl CLOSE:	\$elc7 (im ROM um 1 dekrementiert)
:a04e 7a ab	Adresse der Routine zum Basic-Befehl GET:	\$ab7b (im ROM um 1 dekrementiert)
:a050 41 a6	Adresse der Routine zum Basic-Befehl NEW:	\$a642 (im ROM um 1 dekrementiert)

; ROM-Tabelle der Adressen der Routinen zu den Basic-Funktionen; wird bei \$afd6 und \$afdb ausgelesen

:a052 39 bc	Adresse der Routine zur Basic-Funktion SGN:	\$bc39 (nicht dekrementiert!)
:a054 cc bc	Adresse der Routine zur Basic-Funktion INT:	\$bccc (nicht dekrementiert!)
:a056 58 bc	Adresse der Routine zur Basic-Funktion ABS:	\$bc58 (nicht dekrementiert!)
:a058 10 03	Adresse der Routine zur Basic-Funktion USR:	\$0310 (nicht dekrementiert!)
:a05a 7d b3	Adresse der Routine zur Basic-Funktion FRE:	\$b37d (nicht dekrementiert!)
:a05c 9e b3	Adresse der Routine zur Basic-Funktion POS:	\$b39e (nicht dekrementiert!)
:a05e 71 bf	Adresse der Routine zur Basic-Funktion SQR:	\$bf71 (nicht dekrementiert!)
:a060 97 e0	Adresse der Routine zur Basic-Funktion RND:	\$e097 (nicht dekrementiert!)
:a062 ea b9	Adresse der Routine zur Basic-Funktion LOG:	\$b9ea (nicht dekrementiert!)
:a064 ed bf	Adresse der Routine zur Basic-Funktion EXP:	\$bfed (nicht dekrementiert!)
:a066 64 e2	Adresse der Routine zur Basic-Funktion COS:	\$e264 (nicht dekrementiert!)
:a068 6b e2	Adresse der Routine zur Basic-Funktion SIN:	\$e26b (nicht dekrementiert!)
:a06a b4 e2	Adresse der Routine zur Basic-Funktion TAN:	\$e2b4 (nicht dekrementiert!)
:a06c 0e e3	Adresse der Routine zur Basic-Funktion ATN:	\$e30e (nicht dekrementiert!)
:a06e 0d b8	Adresse der Routine zur Basic-Funktion PEEK:	\$b80d (nicht dekrementiert!)
:a070 7c b7	Adresse der Routine zur Basic-Funktion LEN:	\$b77c (nicht dekrementiert!)
:a072 65 b4	Adresse der Routine zur Basic-Funktion STR\$:	\$b465 (nicht dekrementiert!)
:a074 ad b7	Adresse der Routine zur Basic-Funktion VAL:	\$b7ad (nicht dekrementiert!)
:a076 8b b7	Adresse der Routine zur Basic-Funktion ASC:	\$b78b (nicht dekrementiert!)
:a078 ec b6	Adresse der Routine zur Basic-Funktion CHR\$:	\$b6ec (nicht dekrementiert!)
:a07a 00 b7	Adresse der Routine zur Basic-Funktion LEFT\$:	\$b700 (nicht dekrementiert!)
:a07c 2c b7	Adresse der Routine zur Basic-Funktion RIGHT\$:	\$b72c (nicht dekrementiert!)
:a07e 37 b7	Adresse der Routine zur Basic-Funktion MID\$:	\$b737 (nicht dekrementiert!)

; Prioritätsflags und Adressen der Basic-2.0-Operatoren

Prioritätsflags werden bei \$adfl, \$ael9 und \$ae35, Routinenadressen bei \$ae20 (HB) und \$ae24 (LB) berücksichtigt.

:a080 79	Priorität von + (Addition):	\$79 (vierthöchste von 8 Prioritäten)
:a081 69 b8	Adresse der Routine zu + (Addition):	\$b86a (im ROM um 1 dekrementiert!)
:a083 79	Priorität von - (hier: Subtraktion):	\$79 (vierthöchste von 8 Prioritäten)
:a084 52 b8	Adresse der Routine zu - (Subtraktion):	\$b853 (im ROM um 1 dekrementiert!)
:a086 7b	Priorität von * (Multiplikation):	\$7b (dritthöchste von 8 Prioritäten)
:a087 2a ba	Adresse der Routine zu * (Multiplikation):	\$ba2b (im ROM um 1 dekrementiert!)
:a089 7b	Priorität von / (Division):	\$7b (dritthöchste von 8 Prioritäten)
:a08a 11 bb	Adresse der Routine zu / (Division):	\$bb12 (im ROM um 1 dekrementiert!)
:a08c 7f	Priorität von ↑ (Potenzierung):	\$7f (höchste aller 8 Prioritäten)
:a08d 7a bf	Adresse der Routine zu ↑ (Potenzierung):	\$bf7b (im ROM um 1 dekrementiert!)
:a08f 50	Priorität von AND (logische Konjunktion):	\$50 (zweitniedrigste von 8 Prioritäten)
:a090 e8 af	Adresse der Routine zu AND (log. Konj.):	\$afe9 (im ROM um 1 dekrementiert!)
:a092 46	Priorität von OR (logische Disjunktion):	\$46 (niedrigste aller 8 Prioritäten)
:a093 e5 af	Adresse der Routine zu OR (log. Disj.):	\$afe6 (im ROM um 1 dekrementiert!)
:a095 7d	Priorität von - (hier: Vorzeichenwechsel):	\$7d (zweithöchste von 8 Prioritäten)
:a096 b3 bf	Adresse der Routine zu - (Vorz.-wechsel):	\$bfb4 (im ROM um 1 dekrementiert!)
:a098 5a	Priorität von NOT (logisches Komplement):	\$5a (drittniedrigste von 8 Prioritäten)
:a099 d3 ae	Adresse der Routine zu NOT (log. Kompl.):	\$aed4 (im ROM um 1 dekrementiert!)
:a09b 64	Priorität von =, <, > usw. (Vergleich):	\$64 (fünfhöchste von 8 Prioritäten)
:a09c 15 b0	Adresse der Routine zu =,<,> usw. (Vergl.):	\$b016 (im ROM um 1 dekrementiert!)

; Tabelle der Basic-2.0-Befehlswörter für Tokenisierung - in Reihenfolge der dazugehörigen Tokens

(Bit 7 im letzten Buchstaben jedes Befehlswortes als Endmarkierung gesetzt)

Berücksichtigung bei Tokenisierung (CRUNCH: \$a5bc, \$a5fa, \$a5ff) und Ent-Tokenisierung (LIST: \$a730, \$a738)

:a09e 45 4e c4	enD	"end" (b7 in "d" als Endmarkierung)	Token: \$80
:a0a1 46 4f d2	foR	"for" (b7 in "r" als Endmarkierung)	Token: \$81
:a0a4 4e 45 58 d4	next	"next" (b7 in "t" als Endmarkierung)	Token: \$82

:a0a8 44 41 54 c1	datA	"data" (b7 im 2. "a" als Endmarkierung)	Token: \$83
:a0ac 49 4e 50 55 54 a3	input#	"input#" (b7 in "#" als Endmarkierung)	Token: \$84
:a0b2 49 4e 50 55 d4	inpuT	"input" (b7 in "t" als Endmarkierung)	Token: \$85
:a0b7 44 49 cd	diM	"dim" (b7 in "m" als Endmarkierung)	Token: \$86
:a0ba 52 45 41 c4	reaD	"read" (b7 in "d" als Endmarkierung)	Token: \$87
:a0be 4c 45 d4	leT	"let" (b7 in "t" als Endmarkierung)	Token: \$88
:a0c1 47 4f 54 cf	got0	"goto" (b7 im 2. "o" als Endmarkierung)	Token: \$89
:a0c5 52 55 ce	ruN	"run" (b7 in "n" als Endmarkierung)	Token: \$8a
:a0c8 49 c6	iF	"if" (b7 in "f" als Endmarkierung)	Token: \$8b
:a0ca 52 45 53 54 4f 52 c5	restorE	"restore" (b7 im 2. "e" als Endmarkierung)	Token: \$8c
:a0d1 47 4f 53 55 c2	gosuB	"gosub" (b7 in "b" als Endmarkierung)	Token: \$8d
:a0d6 52 45 54 55 52 ce	returN	"return" (b7 in "n" als Endmarkierung)	Token: \$8e
:a0dc 52 45 cd	reM	"rem" (b7 in "m" als Endmarkierung)	Token: \$8f
:a0df 53 54 4f d0	stoP	"stop" (b7 in "p" als Endmarkierung)	Token: \$90
:a0e3 4f ce	oN	"on" (b7 in "n" als Endmarkierung)	Token: \$91
:a0e5 57 41 49 d4	waitT	"wait" (b7 in "t" als Endmarkierung)	Token: \$92
:a0e9 4c 4f 41 c4	load	"load" (b7 in "l" als Endmarkierung)	Token: \$93
:a0ed 53 41 56 c5	savE	"save" (b7 in "e" als Endmarkierung)	Token: \$94
:a0f1 56 45 52 49 46 d9	verifY	"verify" (b7 in "y" als Endmarkierung)	Token: \$95
:a0f7 44 45 c6	deF	"def" (b7 in "f" als Endmarkierung)	Token: \$96
:a0fa 50 4f 4b c5	pokE	"poke" (b7 in "e" als Endmarkierung)	Token: \$97
:a0fe 50 52 49 4e 54 a3	print#	"print#" (b7 in "#" als Endmarkierung)	Token: \$98
:a104 50 52 49 4e d4	prinT	"print" (b7 in "t" als Endmarkierung)	Token: \$99
:a109 43 4f 4e d4	conT	"cont" (b7 in "t" als Endmarkierung)	Token: \$9a
:a10d 4c 49 53 d4	list	"list" (b7 in "t" als Endmarkierung)	Token: \$9b
:a111 43 4c d2	clR	"clr" (b7 in "r" als Endmarkierung)	Token: \$9c
:a114 43 4d c4	cmD	"cmd" (b7 in "d" als Endmarkierung)	Token: \$9d
:a117 53 59 d3	syS	"sys" (b7 im 2. "s" als Endmarkierung)	Token: \$9e
:a11a 4f 50 45 ce	opeN	"open" (b7 in "n" als Endmarkierung)	Token: \$9f
:a11e 43 4c 4f 53 c5	closE	"close" (b7 in "e" als Endmarkierung)	Token: \$a0
:a123 47 45 d4	geT	"get" (b7 in "t" als Endmarkierung)	Token: \$a1
:a126 4e 45 d7	neW	"new" (b7 in "w" als Endmarkierung)	Token: \$a2
:a129 54 41 42 a8	tab(	"tab(" (b7 in "(" als Endmarkierung)	Token: \$a3
:a12d 54 cf	t0	"to" (b7 in "o" als Endmarkierung)	Token: \$a4
:a12f 46 ce	fN	"fn" (b7 in "n" als Endmarkierung)	Token: \$a5
:a131 53 50 43 a8	spc(	"spc(" (b7 in "(" als Endmarkierung)	Token: \$a6
:a135 54 48 45 ce	theN	"then" (b7 in "n" als Endmarkierung)	Token: \$a7
:a139 4e 4f d4	noT	"not" (b7 in "t" als Endmarkierung)	Token: \$a8
:a13c 53 54 45 d0	steP	"step" (b7 in "p" als Endmarkierung)	Token: \$a9
:a140 ab	+	"+" (b7 als Endmarkierung)	Token: \$aa
:a141 ad	-	"-" (b7 als Endmarkierung)	Token: \$ab

:a142 aa	*	"*" (b7 als Endmarkierung)	Token: \$ac
:a143 af	/	"/" (b7 als Endmarkierung)	Token: \$ad
:a144 de	↑	"↑" (b7 als Endmarkierung)	Token: \$ae
:a145 41 4e c4	and	"and" (b7 in "d" als Endmarkierung)	Token: \$af
:a148 4f d2	or	"or" (b7 in "r" als Endmarkierung)	Token: \$b0
:a14a be	<	"<" (b7 als Endmarkierung)	Token: \$b1
:a14b bd	=	"=" (b7 als Endmarkierung)	Token: \$b2
:a14c bc	>	">" (b7 als Endmarkierung)	Token: \$b3
:a14d 53 47 ce	sgN	"sgn" (b7 in "n" als Endmarkierung)	Token: \$b4
:a150 49 4e d4	inT	"int" (b7 in "t" als Endmarkierung)	Token: \$b5
:a153 41 42 d3	abS	"abs" (b7 in "s" als Endmarkierung)	Token: \$b6
:a156 55 53 d2	usR	"usr" (b7 in "r" als Endmarkierung)	Token: \$b7
:a159 46 52 c5	frE	"fre" (b7 in "e" als Endmarkierung)	Token: \$b8
:a15c 50 4f d3	poS	"pos" (b7 in "s" als Endmarkierung)	Token: \$b9
:a15f 53 51 d2	sqR	"sqr" (b7 in "r" als Endmarkierung)	Token: \$ba
:a162 52 4e c4	rnD	"rnd" (b7 in "d" als Endmarkierung)	Token: \$bb
:a165 4c 4f c7	loG	"log" (b7 in "g" als Endmarkierung)	Token: \$bc
:a168 45 58 d0	exP	"exp" (b7 in "p" als Endmarkierung)	Token: \$bd
:a16b 43 4f d3	coS	"cos" (b7 in "s" als Endmarkierung)	Token: \$be
:a16e 53 49 ce	siN	"sin" (b7 in "n" als Endmarkierung)	Token: \$bf
:a171 54 41 ce	taN	"tan" (b7 in "n" als Endmarkierung)	Token: \$c0
:a174 41 54 ce	atN	"atn" (b7 in "n" als Endmarkierung)	Token: \$c1
:a177 50 45 45 cb	peek	"peek" (b7 in "k" als Endmarkierung)	Token: \$c2
:a17b 4c 45 ce	leN	"len" (b7 in "n" als Endmarkierung)	Token: \$c3
:a17e 53 54 52 a4	str\$	"str\$" (b7 in "\$" als Endmarkierung)	Token: \$c4
:a182 56 41 cc	vaL	"val" (b7 in "l" als Endmarkierung)	Token: \$c5
:a185 41 53 c3	asC	"asc" (b7 in "c" als Endmarkierung)	Token: \$c6
:a188 43 48 52 a4	chr\$	"chr\$" (b7 in "\$" als Endmarkierung)	Token: \$c7
:a18c 4c 45 46 54 a4	left\$	"left\$" (b7 in "\$" als Endmarkierung)	Token: \$c8
:a191 52 49 47 48 54 a4	right\$	"right\$" (b7 in "\$" als Endmarkierung)	Token: \$c9
:a197 4d 49 44 a4	mid\$	"mid\$" (b7 in "\$" als Endmarkierung)	Token: \$ca
:a19b 47 cf	g0	"go" (b7 in "o" als Endmarkierung)	Token: \$cb
:a19d 00	[nul]	Endmarkierung der gesamten Tabelle	(Token wäre \$cc, weshalb der sogenannte <SHIFT>+<L>-Trick funktioniert, auf den Kapitel 4 ausführlich eingeht)

; Basic-2.0-Fehlermeldungen als ROM-Tabelle in Reihenfolge der Fehlernummern (1-29, weil BREAK als Nummer 30 in dieser Tabelle fehlt)

Das jeweils letzte Zeichen hat ein gesetztes Bit 7 als Endkennung der Fehlermeldung.

Tip: Die Fehlercodes können durch "POKE 781,Fehlercode:SYS 42039" getestet werden (in Maschinensprache:

"ldx #Fehlercode:jmp \$a437"). Die Tabelle ab \$a328 zeigt auf die Adressen der Fehlertexte im Speicher.

:a19e 54 4f 4f 20 4d 41 4e 59 20 46 49 4c 45 d3	TOO MANY FILES	Fehlercode #01 (\$01)
:alac 46 49 4c 45 20 4f 50 45 ce	FILE OPEN	Fehlercode #02 (\$02)
:alb5 46 49 4c 45 20 4e 4f 54 20 4f 50 45 ce	FILE NOT OPEN	Fehlercode #03 (\$03)
:alc2 46 49 4c 45 20 4e 4f 54 20 46 4f 55 4e c4	FILE NOT FOUND	Fehlercode #04 (\$04)
:ald0 44 45 56 49 43 45 20 4e 4f 54 20 50 52 45 53 45 4e d4	DEVICE NOT PRESENT	Fehlercode #05 (\$05)
:ale2 4e 4f 54 20 49 4e 50 55 54 20 46 49 4c c5	NOT INPUT FILE	Fehlercode #06 (\$06)
:alf0 4e 4f 54 20 4f 55 54 50 55 54 20 46 49 4c c5	NOT OUTPUT FILE	Fehlercode #07 (\$07)
:alff 4d 49 53 53 49 4e 47 20 46 49 4c 45 20 4e 41 4d c5	MISSING FILENAME	Fehlercode #08 (\$08)
:a210 49 4c 4c 45 47 41 4c 20 44 45 56 49 43 45 20 4e 55 4d 42 45 d2	ILLEGAL DEVICE NUMBER	Fehlercode #09 (\$09)
:a225 4e 45 58 54 20 57 49 54 48 4f 55 54 20 46 4f d2	NEXT WITHOUT FOR	Fehlercode #10 (\$0a)
:a235 53 59 4e 54 41 d8	SYNTAX	Fehlercode #11 (\$0b)
:a23b 52 45 54 55 52 4e 20 57 49 54 48 4f 55 54 20 47 4f 53 55 c2	RETURN WITHOUT GOSUB	Fehlercode #12 (\$0c)
:a24f 4f 55 54 20 4f 46 20 44 41 54 c1	OUT OF DATA	Fehlercode #13 (\$0d)
:a25a 49 4c 4c 45 47 41 4c 20 51 55 41 4e 54 49 54 d9	ILLEGAL QUANTITY	Fehlercode #14 (\$0e)
:a26a 4f 56 45 52 46 4c 4f d7	OVERFLOW	Fehlercode #15 (\$0f)
:a272 4f 55 54 20 4f 46 20 4d 45 4d 4f 52 d9	OUT OF MEMORY	Fehlercode #16 (\$10)
:a27f 55 4e 44 45 46 27 44 20 53 54 41 54 45 4d 45 4e d4	UNDEF'D STATEMENT	Fehlercode #17 (\$11)
:a290 42 41 44 20 53 55 42 53 43 52 49 50 d4	BAD SUBSCRIPT	Fehlercode #18 (\$12)
:a29d 52 45 44 49 4d 27 44 20 41 52 52 41 d9	REDIM'D ARRAY	Fehlercode #19 (\$13)
:a2aa 44 49 56 49 53 49 4f 4e 20 42 59 20 5a 45 52 cf	DIVISION BY ZERO	Fehlercode #20 (\$14)
:a2ba 49 4c 4c 45 47 41 4c 20 44 49 52 45 43 d4	ILLEGAL DIRECT	Fehlercode #21 (\$15)
:a2c8 54 49 50 45 20 4d 49 53 4d 41 54 43 c8	TYPE MISMATCH	Fehlercode #22 (\$16)
:a2d5 53 54 52 49 4e 47 20 54 4f 4f 20 4c 4f 4e c7	STRING TOO LONG	Fehlercode #23 (\$17)
:a2e4 46 49 4c 45 20 44 41 54 c1	FILE DATA	Fehlercode #24 (\$18)
:a2ed 46 4f 52 4d 55 4c 41 20 54 4f 4f 20 43 4f 4d 50 4c 45 d8	FORMULA TOO COMPLEX	Fehlercode #25 (\$19)
:a300 43 41 4e 27 54 20 43 4f 4e 54 49 4e 55 c5	CAN'T CONTINUE	Fehlercode #26 (\$1a)
:a30e 55 4e 44 45 46 27 44 20 46 55 4e 43 54 49 4f ce	UNDEF'D FUNCTION	Fehlercode #27 (\$1b)
:a31e 56 45 52 49 46 d9	VERIFY	Fehlercode #28 (\$1c)
:a324 4c 4f 41 c4	LOAD	Fehlercode #29 (\$1d)

; ROM-Tabelle der Adressen der Texte zu den Fehlermeldungen (geordnet nach Fehlercodes, Adressen im Lo-Hi-Format);
wird bei \$a43d und \$a442 ausgelesen

:a328 9e al	\$a19e	Fehlercode #01 (\$01)	TOO MANY FILES
:a32a ac al	\$alac	Fehlercode #02 (\$02)	FILE OPEN
:a32c b5 al	\$alb5	Fehlercode #03 (\$03)	FILE NOT OPEN
:a32e c2 al	\$alc2	Fehlercode #04 (\$04)	FILE NOT FOUND
:a330 d0 al	\$ald0	Fehlercode #05 (\$05)	DEVICE NOT PRESENT
:a332 e2 al	\$ale2	Fehlercode #06 (\$06)	NOT INPUT FILE
:a334 f0 al	\$alf0	Fehlercode #07 (\$07)	NOT OUTPUT FILE

:a336 ff a1	\$alff	Fehlercode #08 (\$08)	MISSING FILENAME
:a338 10 a2	\$a210	Fehlercode #09 (\$09)	ILLEGAL DEVICE NUMBER
:a33a 25 a2	\$a225	Fehlercode #10 (\$0a)	NEXT WITHOUT FOR
:a33c 35 a2	\$a235	Fehlercode #11 (\$0b)	SYNTAX
:a33e 3b a2	\$a23b	Fehlercode #12 (\$0c)	RETURN WITHOUT GOSUB
:a340 4f a2	\$a24f	Fehlercode #13 (\$0d)	OUT OF DATA
:a342 5a a2	\$a25a	Fehlercode #14 (\$0e)	ILLEGAL QUANTITY
:a344 6a a2	\$a26a	Fehlercode #15 (\$0f)	OVERFLOW
:a346 72 a2	\$a272	Fehlercode #16 (\$10)	OUT OF MEMORY
:a348 7f a2	\$a27f	Fehlercode #17 (\$11)	UNDEF'D STATEMENT
:a34a 90 a2	\$a290	Fehlercode #18 (\$12)	BAD SUBSCRIPT
:a34c 9d a2	\$a29d	Fehlercode #19 (\$13)	REDIM'D ARRAY
:a34e aa a2	\$a2aa	Fehlercode #20 (\$14)	DIVISION BY ZERO
:a350 ba a2	\$a2ba	Fehlercode #21 (\$15)	ILLEGAL DIRECT
:a352 c8 a2	\$a2c8	Fehlercode #22 (\$16)	TYPE MISMATCH
:a354 d5 a2	\$a2d5	Fehlercode #23 (\$17)	STRING TOO LONG
:a356 e4 a2	\$a2e4	Fehlercode #24 (\$18)	FILE DATA
:a358 ed a2	\$a2ed	Fehlercode #25 (\$19)	FORMULA TOO COMPLEX
:a35a 00 a3	\$a300	Fehlercode #26 (\$1a)	CAN'T CONTINUE
:a35c 0e a3	\$a30e	Fehlercode #27 (\$1b)	UNDEF'D FUNCTION
:a35e 1e a3	\$a31e	Fehlercode #28 (\$1c)	VERIFY
:a360 24 a3	\$a324	Fehlercode #29 (\$1d)	LOAD

; Adresse der Fehlermeldung "BREAK", deren ASCII-Text nicht in der Fehlertabelle ab \$a19e steht

:a362 83 a3	\$a383	Fehlercode #30 (\$1e)	BREAK
-------------	--------	-----------------------	-------

; In mehreren Zusammenhängen auftretende Texte von Fehler- und Steuermeldungen

(\$00-Byte als Endmarkierung, da zur Ausgabe die Routine STROUT \$AB1E herangezogen wird, die dieses erfordert)

:a364 0d 4f 4b 0d 00	[cr]ok[cr][null]	OK	Beispiel: VERIFYING OK (s. \$e18d)
:a369 20 20 45 52 52 4f 52 00	[2space]error[null]	ERROR	Beispiel: Fehlermeldung (s. \$a465)
:a371 20 49 4e 20 00	[space]in[space][null]	IN	Beispiel: Fehlermeldung (s. \$bdc2)

; Steuermeldung "READY." als Eingabe-Prompt

(\$00-Byte als Endmarkierung, da zur Ausgabe die Routine STROUT \$AB1E herangezogen wird, die dieses erfordert)

:a376 0d 0a 52 45 41 44 59 2e	[cr,lf]ready.	READY.	Beispiel: Steuermeldung (s. \$a474)
0d 0a 00	[cr,lf,null]		

; Fehlermeldung "BREAK" (isoliert von regulärer Fehlertabelle ab \$a19e)

(\$00-Byte als Endmarkierung, da zur Ausgabe die Routine STROUT \$AB1E herangezogen wird, die dieses erfordert)

:a381 0d 0a 42 52 45 41 4b 00 [cr,lf]break[null] BREAK Beispiel: Fehlercode #30 (s. \$a362)

; Routine zur Suche bestimmter Stapeleinträge des Basic-Interpreters bei der Verwaltung von FOR/NEXT und GOSUB/RETURN
(wird von FOR bei \$a749, von NEXT bei \$ad2b und von RETURN bei \$a8d8 als Unterprogramm aufgerufen)

,a38a	ba	tsx	Stapelzeiger ins X-Register, um Offset auf Stapeleinträge berechnen zu können
,a38b	e8	inx	Offset=Offset+1
,a38c	e8	inx	Offset=Offset+1
,a38d	e8	inx	Offset=Offset+1
,a38e	e8	inx	Offset=Offset+1
,a38f	bd 01	lda 0101,x	Headerbyte des letzten Basic-Stapeleintrags in Akkumulator holen
,a392	c9 81	cmp #81	und mit FOR-Code (sinnvollerweise wird das FOR-Token \$81 verwendet) vergleichen
,a394	d0 21	bne a3b7	keine Übereinstimmung (Z=0): RTS-Befehl am Ende der Routine anspringen (aufrufende Routine bekommt Z-Flag übermittelt!)
,a396	a5 4a	lda 4a	HB des FOR/NEXT-Variablenzeigers auslesen
,a398	d0 0a	bne a3a4	<> 0 (Z=0): weiter (effektiv wird Prüfen des nächsten Eintrags bewirkt, s. Text)
,a39a	bd 02 01	lda 0102,x	LB von Stapel
,a39d	85 49	sta 49	in LB des FOR/NEXT-Variablenzeigers holen
,a39f	bd 03 01	lda 0103,x	HB von Stapel
,a3a2	85 4a	sta 4a	in HB des FOR/NEXT-Variablenzeigers holen
,a3a4	dd 03 01	cmp 0103,x	Vergleich fällt positiv aus, wenn vorher \$a39f abgearbeitet wurde, sonst negativ
,a3a7	d0 07	bne a3b0	Vergleich negativ (siehe \$a398!) (Z=0): nächsten Eintrag bearbeiten
,a3a9	a5 49	lda 49	LB des FOR/NEXT-Variablenzeigers auslesen
,a3ab	dd 02 01	cmp 0102,x	mit LB des auf dem Stapel liegenden FOR/NEXT-Variablenzeigers vergleichen
,a3ae	f0 07	beq a3b7	Vergleich positiv (Z=1): RTS anspringen (aufrufende Routine erhält gesetztes Z-Flag)
,a3b0	8a	txa	Offset zwecks Addition zeitweise in Akkumulator holen
,a3b1	18	clc	Carry vor Addition löschen
,a3b2	69 12	adc #12	18 (Länge eines FOR/NEXT-Stapелеintrags) addieren
,a3b4	aa	tax	Ergebnis der Addition wieder in Offset-Register X
,a3b5	d0 d8	bne a38f	noch nicht =0, also noch nicht ganzer Stapel durchsucht (Z=0): nächsten Eintrag bearbeiten, indem an Schleifenbeginn gesprungen wird
,a3b7	60	rts	Rücksprung von Unterroutine, Z-Flag hat entsprechenden Wert (0=Suche erfolglos); im Akku steht \$8d bei Auffinden eines GOSUB/RETURN-Eintrags

; Unterroutine zur Bereitstellung von Variablen-Speicherplatz ab einer Adresse, die im Akkumulator (LB) und im Y-Register (HB) übergeben wird. Dabei erfolgt eine Verschiebung des Variablenspeichers.

Diese Unterroutine wird nur bei \$a50a (Basic-Zeileneinfügung) und \$b15d (Variablenbehandlung) aufgerufen.

,a3b8	20 08 a4	jsr a408 "chkfvm"	Prüfroutine auf ausreichenden Speicherplatz aufrufen (löst bei Bedarf die gefürchtete Garbage Collection oder den "?OUT OF MEMORY ERROR" aus)
,a3bb	85 31	sta 31	LB setzen } Zeiger auf Endadresse der Basic-Arrays (+1) mit neuem Wert belegen,
,a3bd	84 32	sty 32	HB setzen } um zusätzlich angelegte Variable als geschützt zu deklarieren

; Speicherblockverschiebungsroutine BLTUC

(Anfangsadresse des Originalbereichs in \$5f/\$60, dazugehörige Endadresse in \$5a/\$5b und Endadresse des Zielbereichs in \$58/\$59 als vorausgesetzte Parameter)

Aufruf im ROM nur von \$b628 (innerhalb der Garbage-Collection-Routine).

,a3bf	38	sec	Carry vor Subtraktion setzen	} Länge des Kopierbereichs nach Y/X berechnen; LB auch in \$22 merken. Y- und X-Register werden in Verschiebeschleifen als Zähler eingesetzt.
,a3c0	a5 5a	lda 5a	LB der Quell-Endadresse holen	
,a3c2	e5 5f	sbc 5f	davon LB der Quell-Anfangsadresse subtrahieren	
,a3c4	85 22	sta 22	Ergebnis in \$22 für später merken (s. \$a3d3, \$a3de)	
,a3c6	a8	tay	Ergebnis ins Y-Register für unmittelbare Auswertung	} Verschiebeschleifen als Zähler eingesetzt.
,a3c7	a5 5b	lda 5b	HB der Quell-Endadresse	
,a3c9	e5 60	sbc 60	davon HB der Quell-Anfangsadresse subtrahieren	
,a3cb	aa	tax	Ergebnis ins X-Register für unmittelbare Auswertung	
,a3cc	e8	inx	um 1 erhöhen, damit Ergebnis als Page-Zähler in Dekrementierschleife verwendbar ist	
,a3cd	98	tya	LB des Ergebnisses (s. \$a3c6) testhalber in Akkumulator transportieren	
,a3ce	f0 23	beq a3f3	LB=0 (Z=1): nur komplette Pages verschieben, da Bereichslänge "ganz-seitig" ist	
,a3d0	a5 5a	lda 5a	LB der Quell-Endadresse holen	} Anfangsadresse des Quell-Restbereichs berechnen und nach \$5a/\$5b schreiben (für Verschiebeschleife)
,a3d2	38	sec	Carry vor Subtraktion setzen	
,a3d3	e5 22	sbc 22	LB der Bereichslänge (s. \$a3c4) abziehen	
,a3d5	85 5a	sta 5a	Ergebnis in LB der Quell-Endadresse schreiben	
,a3d7	b0 03	bcs a3dc	Subtraktion ohne Übertrag (C=1): nicht HB=HB-1	} Anfangsadresse des Ziel-Restbereichs berechnen und nach \$58/\$59 schreiben (für Verschiebeschleife)
,a3d9	c6 5b	dec 5b	HB der Quell-Endadresse dekrementieren	
,a3db	38	sec	Carry vor Subtraktion setzen	
,a3dc	a5 58	>lda 58	LB der Ziel-Endadresse holen	
,a3de	e5 22	sbc 22	LB der Bereichslänge (s. \$a3c4) abziehen	} Anfangsadresse des Ziel-Restbereichs berechnen und nach \$58/\$59 schreiben (für Verschiebeschleife)
,a3e0	85 58	sta 58	und somit neues LB der Ziel-Endadresse setzen	
,a3e2	b0 08	bcs a3ec	kein Subtraktionsübertrag (C=1): nicht HB=HB-1	
,a3e4	c6 59	dec 59	HB dekrementieren (Übertrag berücksichtigen)	
,a3e6	90 04	bcc a3ec "jmp"	an Schleifenkontrolle von Kopierschleife für Restbereich springen	

; hier beginnt die Verschiebeschleife für den Restbereich

(Länge im Y-Register, Anfang der Quelle in \$5a/\$5b und Anfang des Ziels in \$58/\$59 zu übergeben)

Diese Schleife dient auch beim Verschieben ganzer Pages zum Bearbeiten einer Page, wofür im Y-Register \$00 steht

,a3e8	b1	5a	→lda (5a),y	Byte aus Quell-Restbereich holen
,a3ea	91	58	sta (58),y	und in Ziel-Restbereich schreiben
,a3ec	88		→dey	Schleifenzähler (s. \$a3c6) dekrementieren
,a3ed	d0	f9	↳bne a3e8	Restbereich noch nicht fertig umkopiert (Z=0): an Schleifenanfang

; hier beginnt die Verschiebeschleife für die ganzen Pages (Speicherseiten)

(Anzahl der Pages +1 im X-Register, Endadresse der Quelle in \$5a, Endadresse des Ziels in \$58/\$59 zu übergeben)

,a3ef	b1	5a	lda (5a),y	Byte aus Quellbereich holen
,a3f1	91	58	sta (58),y	und in Zielbereich schreiben
,a3f3	c6	5b	→dec 5b	HB des Zeigers auf aktuelle Quell-Page dekrementieren
,a3f5	c6	59	dec 59	HB des Zeigers auf aktuelle Ziel-Page dekrementieren
,a3f7	ca		dex	Page-Zähler (zu Beginn: Anzahl der Pages +1) dekrementieren
,a3f8	d0	f2	↳bne a3ec	noch nicht heruntergezählt (Z=0): in Verschiebeschleife für Restbereich einsteigen, die dann die aktuelle Quell-Page in die aktuelle Ziel-Page kopiert und dadurch gewissermaßen zweckentfremdet wird)
,a3fa	60		rts	Rücksprung bei abgeschlossenem Kopiervorgang

; GETSTK: Prüfroutine auf Stapelplatz für 2-Byte-Einträge einer im Akkumulator übergebenen Anzahl

(Verwendung bei \$a757 von FOR, \$a885 von GOSUB und \$adae von FRMEVL)

,a3fb	0a		asl	Akku mit 2 multiplizieren, um Anzahl der erforderlichen Bytes zu berechnen
,a3fc	69	3e	adc #3e	62 (vom Interpreter geforderte Mindestmenge an freiem Stapelspeicher) addieren
,a3fe	b0	35	↳bcs a435	Übertrag bei Addition (C=1): Fehlermeldung OUT OF MEMORY erzeugen
,a400	85	22	sta 22	Ergebnis der Addition (= benötigter Stapelzeiger) in \$22 merken
,a402	ba		tsx	Stapelzeiger in X-Register für Berechnungen holen
,a403	e4	22	cpx 22	mit Additionsergebnis (s. \$a400) vergleichen
,a405	90	2e	↳bcc a435	realer Stapelzeiger kleiner als benötigter (C=0): Fehlermeldung OUT OF MEMORY
,a407	60		rts	ansonsten ordnungsgemäßer Rücksprung von Routine, wenn genügend Platz vorhanden ist

; CHKFVM: Prüfroutine auf freien Speicherplatz im Variablenspeicher (löst bei Bedarf die Garbage Collection aus, um auf diese Weise Speicherplatz bereitstellen zu können). In Akkumulator (LB) und Y-Register (HB) hat die höchste Adresse des benötigten Bereiches zu stehen.

Aufruf bei \$a3b8 (BLTUC-Vorspann), \$b264 (Anlage einer Array-Variablen), \$b269 (ebenfalls bei Anlage einer Array-Variablen) und \$e426 (MSGNEW: Einschaltmeldung)

,a408	c4 34	cpy 34	Vergleich des HB der übergebenen Adresse mit HB des Zeigers auf Anfang des Stringbereichs (= für andere Variablen gesperrter Bereich!)
,a40a	90 28	bcc a434	HB des Zeigers kleiner (C=0): Rücksprung über RTS, da genügend Platz vorhanden
,a40c	d0 04	bne a412	keine Übereinstimmung, also HB des Zeigers größer (Z=0): LB-Vergleich überspringen
,a40e	c5 33	cmp 33	LB-Vergleich des übergebenen Akku mit LB des Zeigers auf Anfang des Stringbereichs
,a410	90 22	bcc a434	LB des Zeigers kleiner (C=0): Rücksprung über RTS, da genügend Platz vorhanden
,a412	48	→pha	Akkumulator (übergebenes LB) merken
,a413	a2 09	ldx #09	Dekrementierzähler der Rettungsschleife für Zwischenspeicher \$58-\$5f initialisieren
,a415	98	tya	übergebenes HB in Akku als erster zu rettender Wert enthalten
,a416	48	→pha	Akku auf Stapel legen (zunächst HB, dann Werte aus \$57-\$5F)
,a417	b5 57	lda 57,x	Zwischenspeicher auslesen (wird im nächsten Durchgang auf Stapel gerettet)
,a419	ca	dex	Dekrementierzähler=Dekrementierzähler-1 (dient auch als Offset von \$57 aus!)
,a41a	10 fa	bpl a416	noch nicht fertig (N=0): Rettungsschleife fortsetzen
,a41c	20 26 b5	jsr b526 "garcol"	Garbage Collection, da noch nicht ausreichend Speicherplatz vorhanden
,a41f	a2 f7	ldx #f7	Trick (s. Routinenbeschreibung im Fließtext!): Offset / Inkrementierzähler
,a421	68	→pla	Wert von Stapel holen
,a422	95 61	sta 61,x	und in Bereich \$58-\$5f zurückholen
,a424	e8	inx	Inkrementierzähler/Offset erhöhen
,a425	30 fa	bmi a421	noch nicht fertig (N=1): weiter in Schleife
,a427	68	pla	Akku holen } an diese Routine übergebenes HB
,a428	a8	tay	und nach Y } vom Stapel zurückholen
,a429	68	pla	Akku (übergebenes LB) vom Stapel zurückholen
,a42a	c4 34	cpy 34	HB mit Zeiger-HB auf Stringbereich vergleichen
,a42c	90 06	bcc a434	benötigtes HB kleiner (C=0): Routine verlassen
,a42e	d0 05	bne a435	HBS ungleich (Z=0): OUT OF MEMORY hervorrufen
,a430	c5 33	cmp 33	LBs vergleichen (Akku mit LB des Zeigers)
,a432	b0 01	bcs a435	benötigtes LB größer oder gleich (C=1): OUT OF MEMORY
,a434	60	→rts	Ende der Routine, wenn jetzt Platz vorhanden ist

} nach Garbage Collection:
 letzte Feststellung, ob
 wenigstens jetzt Platz
 vorhanden ist; wennnicht,
 wird OUT OF MEMORY erzeugt

; OUT-OF-MEMORY-Einsprung

(wird über JMP bei \$b30b und über Verzweigungen bei \$a3fe, \$a405, \$a42e und \$a432 genutzt)

,a435	a2 10	ldx #10	Fehlermeldung OUT OF MEMORY durch Fehlernummer #16 vorbereiten (im Speicher folgt unmittelbar der Sprung über den Fehlervektor)
-------	-------	---------	--

; ERROR: Einsprung in Fehlerbehandlungsroutine des Basic-2.0-Interpreters über Fehlervektor

(Aufruf von \$a573, \$a857, \$a8e5, \$ab68, \$ad32, \$ad9b, \$b65a, \$b980, \$bb86, \$el09 und \$el9e aus)

,a437	6c 00 03	jmp(0300)	Sprung über Fehlervektor IERROR \$0300/\$0301 normalerweise wird nach \$a43a gesprungen (sofern der Vektor nicht verstellt wurde)
-------	----------	-----------	--

im X-Register wird die Fehlernummer übergeben (1-30; andere Werte werden nicht ordnungsgemäß behandelt!)

```

-----
,a43a 8a      txa      Fehlernummer zwecks Multiplikation in Akku      } Fehlernummer in X mit 2
,a43b 0a      asl      Akku mit 2 multiplizieren                } multiplizieren, da Tabelle
,a43c aa      tax      Ergebnis zurück ins X-Register           } aus 2-Byte-Einträgen besteht
,a43d bd 26 a3 lda a326,x  LB der Adresse des Fehlertextes in ROM-Tabelle } Adresse, ab der der Text zur
,a440 85 22    sta 22    nach $22 (LB des Hilfszeigers) schreiben } Fehlermeldung im ROM steht,
,a442 bd 27 a3 lda a327,x  HB der Adresse des Fehlertextes in ROM-Tabelle } aus ROM-Tabelle ab $a328 (!)
,a445 85 23    sta 23    nach $23 (HB des Hilfszeigers) schreiben } in Zeiger $22/$23 holen
,a447 20 cc ff jsr ffcc "clrchn" Tastatur als Eingabe-, Bildschirm als Ausgabegerät setzen (Standardzustand)
,a44a a9 00    lda #00    $00 (Flag für "Standard-I/O") laden } Tastatur und Bildschirm als Basic-I/O-Geräte
,a44c 85 13    sta 13     und nach $13 schreiben                } setzen (Basic-Ergänzung zu "a447 jsr clrchn")
,a44e 20 d7 aa jsr aad7    Ausgaberroutine für [CR] und ggf. [LF] aufrufen
,a451 20 45 ab jsr ab45 "qumout" Fragezeichen (ASCII-Code $3f) ausgeben
,a454 a0 00    ldy #00    Offset mit 0 initialisieren
,a456 b1 22    →lda (22),y Zeichen aus Fehlertext holen (s. $a43d—$a445) } Ausgabe des zur Fehlermeldung
,a458 48      pha      und auch auf Stapel sichern              } gehörenden Textes, dessen
,a459 29 7f    and #7f %01111111 b7 löschen (also mögliche Endmarkierung herausnehmen) } Adresse in $22/$23 berechnet
,a45b 20 47 ab jsr ab47 "bbsout" Zeichen ausgeben (Basic-Einsprung für BSOUT) } wurde. Die Endmarkierung ist
,a45e c8      iny      Offset erhöhen (auf nächstes Zeichen stellen) } ein gesetztes b7, das bei der
,a45f 68      pla      gesichertes Zeichen holen (s. $a458)      } Textausgabe herausgefiltert
,a460 10 f4    bpl a456   b7 war gelöscht (N=0): Fehlertext-Ausgabe fortsetzen } und später - nach Rettenauf
,a462 20 7a a6 jsr a67a   in Basic-Routine zu NEW einsteigen, um Zeiger auf temporären Stringstapel, } den Stapel - getestet wird.
                               Stapelzeiger des Prozessors und Flag für Benutzerfunktionsaufruf zu initialisieren
                               sowie den CONT-Befehl durch Löschen von $3e zu sperren (bereitet CAN'T CONTINUE vor)

,a465 a9 69    lda #69 <($a369) LB der Adresse von Text "error" im Speicher } Ausgabe des Textes
,a467 a0 a3    ldy #a3 d($a369) HB der Adresse von Text "error" im Speicher } [2SPACE]error
,a469 20 1e ab jsr able "strout" Text über BSOUT-Schleife ausgeben } unter Zuhilfenahme von STROUT $able
,a46c a4 3a    ldy 3a     HB der aktuellen Zeilennummer nach Y holen
,a46e c8      iny      um 1 erhöhen, um auf $ff (Flag für Direktmodus) testen zu können
,a46f f0 03    beq a474    HB war = $ff (ist jetzt = $00) (Z=1): Ausgabe von "in<Zeilennummer>" überspringen
,a471 20 c2 bd jsr bdc2    gibt Text "[space]in" und Zeilennummer über NUMOUT $BD CD aus; vor der Zeilennummer
                               wird von der NUMOUT-Routine ein Leerzeichen ausgegeben, das dann nach "in" erscheint

,a474 a9 76    →lda #76 <($a376) LB der Adresse von Text "ready." im Speicher } Ausgabe des Textes
,a476 a0 a3    ldy #a3 d($a376) HB der Adresse von Text "ready." im Speicher } [CR,LF]ready.[CR,LF]
,a478 20 1e ab jsr able "strout" Text über BSOUT-Schleife ausgeben } unter Zuhilfenahme von STROUT $able
,a47b a9 80    lda #80 %10000000 Flag für "Direktmodus" laden } auf Direktmodus umstellen, weil im Speicher der
,a47d 20 90 ff jsr ff90 "setmsg" und über SETMSG setzen        } Einsprung für Basic-Warmstart folgt

```

; MAIN: Basic-Warmstart (Sprung in Eingabe-Modus);
über JMP wird hierher bei \$a530 verzweigt.

<pre> ,a480 6c 02 03 >jmp(0302) ----- ,a483 20 60 a5 jsr a560 "getsyb" ,a486 86 7a stx 7a ,a488 84 7b sty 7b ,a48a 20 73 00 jsr 0073 "chrget" ,a48d aa tax ,a48e f0 f0 beq a480 "main" ,a490 a2 ff ldx #ff ,a492 86 3a stx 3a ,a494 90 06 bcc a49c ,a496 20 79 a5 jsr a579 "crunch" ,a499 4c e1 a7 jmp a7e1 "gone" </pre>	Sprung über Warmstart-Vektor IMAIN \$0302/\$0303 normalerweise wird nach \$a483 gesprungen (wenn IMAIN nicht verstellt wurde) holt Eingabe (Programmzeile oder Direktanweisung) in System-Eingabepuffer ab \$0200 LB des Programmzählers in CHRGET-Routine schreiben } CHRGET-Zeiger auf System- HB des Programmzählers in CHRGET-Routine schreiben } Eingabepuffer stellen holt nächstes Zeichen aus Eingabepuffer in Akku und setzt dabei die Flags Zeichen zwecks Test in X-Register transportieren = 0 (Z=1): Warmstart und Eingabe wiederholen, da Puffer leer ist (1.Byte=Ende-Marke) Flag für "Direktmodus" laden } mögliche Direktausführung der Eingabe und in HB der Zeilennummer schreiben } (falls keine Zeilennummer) vorbereiten erstes Zeichen ist Ziffer (s. \$a48a) (C=0): Basic-Zeile in Programm aufnehmen Tokenisierung der im Eingabepuffer befindlichen Eingabe Ausführung der Eingabe durch Sprung in Interpreterschleife
--	--

; Aufnahme der im Eingabepuffer stehenden Eingabezeile in das aktuelle Basic-Programm

<pre> ,a49c 20 6b a9 >jsr a96b "linget" ,a49f 20 79 a5 jsr a579 "crunch" ,a4a2 84 0b sty 0b ,a4a4 20 13 a6 jsr a613 "fndlin" ----- ,a4a7 90 44 bcc a4ed ,a4a9 a0 01 ldy #01 ,a4ab b1 5f lda (5f),y ,a4ad 85 23 sta 23 ,a4af a5 2d lda 2d ,a4b1 85 22 sta 22 ,a4b3 a5 60 lda 60 ,a4b5 85 25 sta 25 ,a4b7 a5 5f lda 5f ,a4b9 88 dey ,a4ba f1 5f sbc (5f),y ,a4bc 18 clc ,a4bd 65 2d adc 2d ,a4bf 85 2d sta 2d ,a4c1 85 24 sta 24 </pre>	gültige (!) Zeilennummer (am Pufferanfang) über LINGET nach \$14/\$15 holen Tokenisierung der nach der Zeilennummer im Eingabepuffer befindlichen Eingabe Eingabepufferzeiger mit Länge der neuen Programmzeile (ohne Linkpointer usw.) laden Zeile mit eingegebener Nummer im Speicher suchen (Test, ob und wo vorhanden) falls Zeile vorhanden: C=1 und vorhandene Adresse in \$5f/\$60; sonst C=0 entsprechende Zeile existiert noch nicht (C=0): Löschen der alten Zeile überspringen Offset mit \$01 initialisieren (auf HB des Linkpointers stellen) HB des Linkpointers (Adresse der nächsten Zeile) in HB des Zeigers \$22/\$23 holen LB des Basic-Variablen-Beginns (= Programmende) in LB des Zeigers \$22/\$23 holen HB der Adresse der aktuellen Zeile in HB des Zeigers \$24/\$25 schreiben LB der Adresse der aktuellen Zeile holen Offset auf \$00 (s. \$a4a9), also auf LB des Linkpointers LB des Linkpointers von LB der Zeilenadresse abziehen Carry vor Addition löschen LB des Basic-Variablen-Beginns (= Programmende) addieren Ergebnis als neues LB des Ende-Zeigers setzen auch als LB des Zeigers \$24/\$25 setzen	Berechnung der Zeiger \$22/\$23, \$24/\$25 und Y/X. Diese enthalten nach der Berechnung die entsprechenden Werte, damit durch das Verschieben von (Y/X) Bytes von (\$22/\$23) nach (\$24/\$25) die alte Zeile durch
--	---	--

,a4c3	a5 2e	lda 2e	HB des Basic-Variablen-Beginns (= Programmende) holen	die folgenden
,a4c5	69 ff	adc #ff	bei C=0 wird 1 subtrahiert, bei C=1 wird 0 addiert	überschrieben und
,a4c7	85 2e	sta 2e	Ergebnis in HB des Basic-Variablen-Beginn-Zeigers	dadurch gelöscht
,a4c9	e5 60	sbc 60	davon HB der Adresse der aktuellen Zeile abziehen	wird. Die
,a4cb	aa	tax	Ergebnis in X-Register merken	Linkpointer
,a4cc	38	sec	Carry vor Subtraktion setzen	werden dabei
,a4cd	a5 5f	lda 5f	LB der Adresse der aktuellen Zeile holen	zunächst nicht
,a4cf	e5 2d	sbc 2d	LB des neuen Zeigers auf Programmende abziehen	angepaßt, was
,a4d1	a8	tay	Ergebnis ins Y-Register	aber später
,a4d2	b0 03	bcs a4d7	kein Subtraktionsübertrag (C=1): HBs nicht ändern	erfolgt.
,a4d4	e8	inx	HB von Wert in Y/X inkrementieren	
,a4d5	c6 25	dec 25	HB von Zeiger \$24/\$25 dekrementieren	
,a4d7	18	→clc	Carry vor Addition löschen	
,a4d8	65 22	adc 22	LB des Zeigers \$22/\$23 addieren	
,a4da	90 03	bcc a4df	kein Übertrag (C=0): HBs nicht ändern	
,a4dc	c6 23	dec 23	HB des Zeigers \$22/\$23 dekrementieren	
,a4de	18	clc	Carry löschen	
,a4df	b1 22	→lda (22),y	Wert ab nächster Programmzeile auslesen	Schleife zum
,a4e1	91 24	sta (24),y	und so verschieben, daß alte Zeile überschrieben wird	Verschiebeneiner
,a4e3	c8	iny	Low-Zähler erhöhen	in Y (LB) und X
,a4e4	d0 f9	bne a4df	noch nicht fertig (Z=0): weiter in Verschiebeschleife	(HB) angegebenen
,a4e6	e6 23	inc 23	HB von \$22/\$23 erhöhen	Zahl von Bytes ab
,a4e8	e6 25	inc 25	HB von \$24/\$25 erhöhen	der Adresse in
,a4ea	ca	dex	HB von Y/X herunterzählen	\$22/\$23 zu der
,a4eb	d0 f2	bne a4df	noch nicht auf 0 heruntergezählt (Z=0): weiter	Adresse in\$24/\$25

; hier ist eine möglicherweise vorher vorhandene Zeile mit gleicher Zeilennummer wie der im Eingabemodus eingegebenen Nummer schon gelöscht. Es erfolgt das Einfügen der neuen Eingabe in das aktuelle Programm.

,a4ed	20 59	a6→jsr a659	in NEW/CLR-Routine einsteigen (Variablen löschen und Basic-Zeiger anpassen, dadurch Löschen/Einfügen alter Variablenbereich verlorengeht)	
,a4f0	20 33 a5	jsr a533 "lnkprg"	Neuberechnung der Linkpointer, da dies bei eventuellem Löschen der alten Zeile nicht erfolgte, weil lediglich die Programmblöcke im Speicher verschoben wurden	
,a4f3	ad 00 02	lda 0200	erstes Byte des Eingabepuffers auslesen (enthält tokenisierte Zeile)	
,a4f6	f0 88	↑beq a480 "main"	= 0, also nichts im Eingabepuffer (Z=1): erneuter Warmstart, da nur eine Zeilennummer zum Löschen einer Zeile eingegeben wurde	
,a4f8	18	clc	Carry vor Addition bei \$a4fd löschen	Platz im aktuellen Programm ab
,a4f9	a5 2d	lda 2d	LB des Zeigers auf Programmende	der Einfüge-Adresse schaffen (für
,a4fb	85 5a	sta 5a	in LB des Hilfszeigers \$5a/\$5b schreiben	so viele Bytes, wie die neue
,a4fd	65 0b	adc 0b	und Länge der Eingabezeile addieren (s. \$a4a2)	Zeile benötigt). Für die
,a4ff	85 58	sta 58	und Ergebnis als LB von \$58/\$59 schreiben	Speicherblockverschiebungsroutine

,a501	a4 2e	ldy 2e	HB des Zeigers auf Programmende	} werden die Zeiger \$58/\$59 und \$5a/\$5b auf Ende von Ziel- und Quellbereich gestellt. \$5f/\$60 (Adresse der Zeile) ist schon seit \$a4a4 richtig eingestellt.
,a503	84 5b	sty 5b	in HB des Hilfszeigers \$5a/\$5b schreiben	
,a505	90 01	bcc a508	kein Übertrag (s. \$a4fd) (C=0): HB nicht erhöhen	
,a507	c8	iny	HB erhöhen	
,a508	84 59	sty 59	und in HB des Hilfszeigers \$58/\$59 schreiben	
,a50a	20 b8 a3	jsr a3b8	Basic-Speicherblockverschiebung aufrufen	
,a50d	a5 14	lda 14	LB der Zeilennummer in Akku	} Zeilennummer im Low-High-Format vor den Eingabepuffer (!), also
,a50f	a4 15	ldy 15	HB der Zeilennummer in Y	
,a511	8d fe 01	sta 01fe	LB aus Akku nach \$01fe schreiben	} in \$01fe und \$01ff schreiben, damit die Zeile von dort in den Speicher übernehmbar ist
,a514	8c ff 01	sty 01ff	HB aus Y nach \$01ff schreiben	
,a517	a5 31	lda 31	LB des Zeigers auf Ende der Basic-Arrays +1 holen	} Zeiger auf Ende des Basic-Programms (=Variablenbeginn) neu setzen
,a519	a4 32	ldy 32	HB des Zeigers auf Ende der Basic-Arrays +1 holen	
,a51b	85 2d	sta 2d	LB in LB des Zeigers auf Programmende (= Anfang der Variablen)	
,a51d	84 2e	sty 2e	HB in HB des Zeigers auf Programmende (= Anfang der Variablen)	
,a51f	a4 0b	ldy 0b	Anzahl der Bytes im Eingabepuffer in Y-Register holen	
,a521	88	dey	um 1 verringern, um Dekrementierzähler für Schleife zu erhalten	
,a522	b9 fc 01	lda 01fc,y	Bytes von \$01fc an (\$01fc/\$01fd wäre Linkpointer, \$01fe/\$01ff ist Zeilennummer, \$0200 ist die tokenisierte Zeile) auslesen	
,a525	91 5f	sta (5f),y	in Programm ab Zieladresse in \$5f/\$60 (seit \$a4a4 gesetzt) kopieren	
,a527	88	dey	Dekrementierzähler herunterzählen	
,a528	10 f8	bpl a522	noch nicht auf \$ff heruntergezählt (N=0): weiter in Schleife	

; hierher wird nach erfolgreichem Laden bei \$e19c verzweigt

,a52a	20 59 a6	jsr a659	In NEW/CLR-Routine einsteigen (Variablen löschen und Basic-Zeiger anpassen, dadurch Löschen/Einfügen alter Variablenbereich verlorengeht)
,a52d	20 33 a5	jsr a533 "lnkprg"	Neuberechnung der Linkpointer, da in \$01fc/\$01fd nur ein zufälliger Wert stand
,a530	4c 80 a4	jmp a480 "main"	erneuter Basic-Warmstart (Eingabe der nächsten Zeile)

; LNKPRG-Unterprogramm zur Neuberechnung der Linkpointer im aktuellen Programm

Wird bei \$a4f0 nach Löschen einer Programmzeile, bei \$a52d nach Einfügen einer neuen Zeile und bei \$e1b8 nach Laden eines Basic-Programms (über den LOAD-Befehl) als Unterroutine aufgerufen.

,a533	a5 2b	lda 2b	LB des Zeigers auf den Programmanfang in Akku	} Hilfszeiger \$22/\$23 enthält immer die Basisadresse der gerade relevanten Zeile; hier: Belegung mit Adresse der 1. Programmzeile
,a535	a4 2c	ldy 2c	HB des Zeigers auf den Programmanfang in Y	
,a537	85 22	sta 22	LB in LB des Hilfszeigers \$22/\$23	
,a539	84 23	sty 23	HB in HB des Hilfszeigers \$22/\$23	
,a53b	18	clc	Carry vor möglicher Addition bei \$a54b löschen	
,a53c	a0 01	ldy #01	Offset auf HB des Linkpointers stellen	
,a53e	b1 22	lda (22),y	HB des Linkpointers der aktuellen Zeile holen	
,a540	f0 1d	beq a55f	= 0, also Programmende (Z=1): RTS anspringen	

```

,a542 a0 04      ldy #04      Offset auf erstes Byte des Zeileninhalts stellen
,a544 c8        >iny          Offset=Offset+1 (dient Berechnung der Zeilenlänge)
,a545 b1 22      lda (22),y    Byte aus Zeileninhalt auslesen
,a547 d0 fb      bne a544      noch nicht Endmarkierung (Z=0): weiter prüfen
,a549 c8        iny          Offset=Offset+1, um Länge der Zeile in Bytes zu erhalten
,a54a 98        tya          Ergebnis in Akku zwecks Addition
,a54b 65 22      adc 22        LB der aktuellen Basisadresse addieren
,a54d aa        tax          und Ergebnis in X-Register
,a54e a0 00      ldy #00      Offset auf LB des Linkpointers stellen
,a550 91 22      sta (22),y    LB des Linkpointers auf LB von Basis + Zeilenlänge setzen
,a552 a5 23      lda 23        HB der aktuellen Basisadresse holen
,a554 69 00      adc #00       bei Übertrag um 1 erhöhen (s. $a54b)
,a556 c8        iny          Offset auf HB des Linkpointers stellen
,a557 91 22      sta (22),y    Hb des Linkpointers auf HB der Basis + evtl. Übertrag stellen
,a559 86 22      stx 22        X-Register (Ergebnis der Addition, s. $a54d) als neues LB der Basis setzen
,a55b 85 23      sta 23        Akku (s. $a552, $a554) als neues HB der Basis setzen
,a55d 90 dd      bcc a53c "jmp" sicher kein Übertrag (HB nie $ff!), also Schleife fortsetzen mit neuer Basis
-----
,a55f 60        >rts          Routine verlassen (s. $a540), wenn alle Linkpointer neu berechnet wurden
-----

```

; GETSYB-Routine zur Einholung einer Tastatureingabe in den Systemeingabepuffer ab \$0200
ans Ende des Puffers wird \$00 als Endmarkierung geschrieben

```

,a560 a2 00      ldx #00      Offset auf erstes Pufferbyte stellen (Initialisierung)
,a562 20 12 e1 >jsr ell2      Basic-Einsprung für BASIN aufrufen (Zeichen in Akku holen)
,a565 c9 0d      cmp #0d      eingegebenes Zeichen mit [RETURN] vergleichen
,a567 f0 0d      beq a576      Übereinstimmung (Z=1): $00 an Pufferende schreiben, [CR] ausgeben, Ende der Routine
,a569 9d 00 02   sta 0200,x    Zeichen in Eingabepuffer unter Berücksichtigung des Offset schreiben
,a56c e8        inx          Offset erhöhen (auf nächstes Zeichen stellen)
,a56d e0 59      cpx #59      Offset jetzt auf #89, also schon #88 Zeichen eingegeben?
,a56f 90 f1      bcc a562      nein (C=0): weiter in Leseschleife

```

; bei mehr als 88 Zeichen langen Eingaben wird hier STRING TOO LONG bewirkt:

```

,a571 a2 17      ldx #17      Fehlermeldung STRING TOO LONG vorbereiten
,a573 4c 37 a4   jmp a437 "error" Sprung in Fehlerbehandlung ERROR
-----

```

```

,a576 4c ca aa->jmp aaca      Sprung zur Fortsetzung der Routine bei $aaca; nach $aaca wird nur von hier verzweigt!
-----

```

; Einsprung für Tokenisierungsroutine CRUNCH: wandelt im Systemeingabepuffer befindliche Eingabe in Tokens um;
 Aufruf von \$a496 und \$a49f (beide Aufrufstellen liegen in der MAIN-Routine)

,a579	6c 04 03	jmp (0304)	Sprung über Vektor ICRUNCH \$0304/\$0305; führt normalerweise nach \$a57c

,a57c	a6 7a	ldx 7a	Offset mit CHRGET-Programmzähler (LB) laden, da dieser auf erstes Byte nach der über LINGET eingeholten Zeilennummer zeigt
,a57e	a0 04	ldy #04 %00000100	Flag für "gerade kein Quote Mode" laden; Y dient aber auch - vor allem! - als Offset: im X-Register steht der Offset von \$0200 zur untokenisierten Eingabe, in Y von \$0200 zu den tokenisierten Codes
,a580	84 0f	sty 0f	und in \$0f (Flag der CRUNCH-Routine) schreiben
,a582	bd 00 02	→ lda 0200,x	Byte aus Systemeingabepuffer in Akku holen
,a585	10 07	bpl a58e	kein Zeichen mit ASCII-Code > 127 (N=0): nicht auf π prüfen, nicht überlesen
,a587	c9 ff	cmp #ff	Vergleich mit π -Code (einziger Code über 127, der nicht ignoriert werden soll!)
,a589	f0-3e	beq a5c9	Vergleich positiv (Z=1): Zeichen unverändert in Speicher übernehmen
,a58b	e8	inx	Offset erhöhen, ohne Zeichen weiterzuverarbeiten
,a58c	d0 f4	bne a582 "jmp"	nächstes Zeichen holen, um geschiftete Zeichen (außer π) zu überlesen

,a58e	c9 20	→ cmp #20	Leerzeichen ([SPACE])?
,a590	f0-37	beq a5c9	ja (Z=1): Zeichen unverändert in Speicher übernehmen
,a592	85 08	sta 08	ansonsten Wert in \$08 (Flag für Quote Mode) aufnehmen
,a594	c9 22	cmp #22	Anführungszeichen ([QUOTE])?
,a596	f0 56	beq a5ee	ja (Z=1): Zeichen in Speicher übernehmen und Quote-Mode-Flag modifizieren
,a598	24 0f	bit 0f	Flag für Quote Mode testen
,a59a	70-2d	bvs a5c9	b6 gesetzt, also Quote Mode (V=1): Zeichen unverändert in Speicher übernehmen
,a59c	c9 3f	cmp #3f	Fragezeichen (dient als PRINT-Abkürzung)?
,a59e	d0 04	bne a5a4	nein (Z=0): Sonderbehandlung für Fragezeichen überspringen
,a5a0	a9 99	lda #99	Umwandlung von \$3f (ASCII-Code für Fragezeichen) in \$99 (Token für PRINT)
,a5a2	d0-25	bne a5c9 "jmp"	Zeichen unverändert in Speicher übernehmen

,a5a4	c9 30	→ cmp #30	mit ASCII-Code für "0" vergleichen
,a5a6	90 04	bcc a5ac	kleiner (C=0): Sonderbehandlung für \$30-\$3b kommt nicht in Frage, also weiter
,a5a8	c9 3c	cmp #3c	mit \$3c (\$3b + 1) vergleichen
,a5aa	90-1d	bcc a5c9	Zeichen im Bereich \$30-\$3b (C=1): Zeichen unverändert in Speicher übernehmen
,a5ac	84 71	→ sty 71	aktuellen Offset in Y-Register (Offset für Tokenisierungsergebnis) in \$71 retten
,a5ae	a0 00	ldy #00	\$00 als Initialisierungswert laden
,a5b0	84 0b	sty 0b	und als Anzahl der Zeichen im Eingabepuffer setzen
,a5b2	88	dey	auf \$ff setzen, damit nach \$a5b6 am Schleifenbeginn der Initialisierungswert zu 0 wird und somit das erste Byte erfaßt
,a5b3	86 7a	stx 7a	Offset in X-Register in LB des CHRGET-Zeigers schreiben
,a5b5	ca	dex	und um 1 herunterzählen, um Initialisierungswert für X zu haben (s. \$a5b7!)

; Tokenisierungsschleife: versucht, im eingegebenen Text Befehlswörter zu finden, um diese durch Tokens zu ersetzen;
erhält im Akku den nicht-tokenisierten Wert und sucht byteweise nach Übereinstimmungen mit Tokenisierungstabelle

,a5b6	c8	→iny	Offset in Token-Tabelle erhöhen
,a5b7	e8	inx	Offset für nicht-tokenisierte Codes erhöhen
,a5b8	bd 00 02	lda 0200,x	nicht-tokenisiertes Byte aus Systemeingabepuffer in Akku holen
,a5bb	38	sec	Carry vor Subtraktion setzen
,a5bc	f9 9e a0	sbc a09e,y	Subtraktion eines Bytes aus der ASCII-Tabelle der Befehle (zwecks Vergleich)
,a5bf	f0 f5	beq a5b6	Übereinstimmung (Z=1): weiter am Schleifenbeginn, nächstes Byte prüfen
,a5c1	c9 80	cmp #80 %10000000	Ergebnis gleich \$80 (nur bei Endmarkierungen von zutreffendem Tabelleneintrag) ?
,a5c3	d0 30	↙bne a5f5	nein (Z=0): in Befehlstabelle weiter nach Übereinstimmung suchen
,a5c5	05 0b	ora 0b	da jetzt \$80 im Akku steht (s. \$a5c1, \$a5c3), wird hier b7 gesetzt (Offset für Tokens) und die Nummer des Befehlstabelleneintrages addiert, um Token zu erhalten
,a5c7	a4 71	ldy 71	Offset für tokenisiertes Byte zurückholen (s. \$a5ac, \$a5f7)
,a5c9	e8	→inx	Offset für nicht-tokenisiertes Byte aus Systemeingabepuffer erhöhen
,a5ca	c8	iny	Offset für tokenisiertes Byte (s. \$a5c7) erhöhen
,a5cb	99 fb 01	sta 01fb,y	tokenisiertes Byte in Eingabepuffer schreiben
,a5ce	b9 fb 01	lda 01fb,y	und von dort nur zwecks Test (CPU-Flags!) wieder in Akku holen (also keine Änderung des Akkumulator-Inhalts, sondern nur der Prozessorflags)
,a5d1	f0 36	↙beq a609	tokenisiertes Byte ist \$00 = Endmarkierung (Z=1): Ende der Tokenisierungsschleife
,a5d3	38	sec	Carry vor Subtraktion setzen
,a5d4	e9 3a	sbc #3a	mittels Subtraktion auf Doppelpunkt (ASCII-Code \$3a) prüfen, da dieser die Endmarkierung eines einzelnen Befehls ist
,a5d6	f0 04	↙beq a5dc	Übereinstimmung (Z=1): \$0f löschen (s. \$a5dc), da Akku bei Z=1 hier den Wert 0 hat
,a5d8	c9 49	cmp #49	war es der Code \$49 + \$3a (s. \$a5d4!), also \$83 (Token für DATA; nimmt Sonderstellung ein, da DATA-Zeilen nicht tokenisiert werden dürfen)?
,a5da	d0 02	↙bne a5de	nein (Z=0): nicht \$0f mit \$49 (Flag für DATA) belegen, sondern unverändert lassen
,a5dc	85 0f	→sta 0f	wird dieser Befehl durchlaufen (s. \$a5d8/\$a5da), hat Akku den Wert \$49 (DATA-Flag)
,a5de	38	→sec	Carry vor Subtraktion setzen
,a5df	e9 55	sbc #55	war es der Code \$55 + \$3a (s. \$a5d4!), also \$8f (Token für REM; nimmt Sonderstellung ein, da REM-Zeilen nicht tokenisiert werden dürfen)
,a5e1	d0 9f	↙bne a582	nein (Z=0): alle Tests durchlaufen, also nächstes Zeichen aus Systemeingabepuffer holen
,a5e3	85 08	sta 08	Flag für REM setzen (Akku hat hier den Wert \$00, siehe \$a5de-\$a5f1)
,a5e5	bd 00 02	→lda 0200,x	nicht-tokenisiertes Byte aus Systemeingabepuffer holen
,a5e8	f0-df	beq a5c9	Endmarkierung (\$00) (Z=1): unverändert in Speicher übernehmen
,a5ea	c5 08	cmp 08	im Akku steht anderer Wert als \$00 (s. \$a5e5, \$a5e8), also REM-Flag auf \$55 testen
,a5ec	f0-db	beq a5c9	Wert = \$55 (Z=1): Wert unverändert in Speicher übernehmen
,a5ee	c8	iny	Offset für tokenisierte Werte inkrementieren
,a5ef	99 fb 01	sta 01fb,y	Wert in Speicher schreiben

,a5f2	e8	inx	Offset für nicht-tokenisierte Bytes erhöhen
,a5f3	d0 f0	bne a5e5 "jmp"	weiter so mit nächstem Byte aus Systemeingabepuffer

; erfolglos gesuchte Endmarkierung eines Token bearbeiten (hierher wird nur von \$a5c3 gesprungen)

,a5f5	a6 7a	ldx 7a	LB des CHRGET-Zeigers, der zur Auswertung des Systemeingabepuffers dient, holen
,a5f7	e6 0b	inc 0b	Nummer des gesuchten Tabelleneintrags erhöhen (erhöht effektiv das Token)
,a5f9	c8	iny	Offset in Befehlstabelle erhöhen
,a5fa	b9 9d a0	lda a09d,y	nächstes Byte aus Befehlstabelle holen
,a5fd	10 fa	bpl a5f9	keine Endmarkierung von Befehlswort (N=0): weiter nach Endmarkierung suchen
,a5ff	b9 9e a0	lda a09e,y	darauffolgendes Byte aus Befehlstabelle holen
,a602	d0 b4	bne a5b8	kein Nullbyte als Endmarkierung der Befehlswörtertabelle gefunden (Z=0): zurück in Tokenisierungsschleife
,a604	bd 00 02	lda 0200,x	nicht-tokenisiertes Byte aus Systemeingabepuffer holen
,a607	10 be	bpl a5c7 "jmp"	zurück in Suchschleife; jetzt wird nächstes Befehlswort gesucht

; Endmarkierung \$00 behandeln (hierher wird nur von \$a5dl gesprungen)

,a609	99 fd 01	sta 01fd,y	Byte ist hier immer \$00 und wird in Speicher übernommen
,a60c	c6 7b	dec 7b	HB des CHRGET-Zeigers herunterzählen (war vorher immer \$02 und wird hier zu \$01)
,a60e	a9 ff	lda #ff <(\$01ff)	\$ff (LB von \$01ff = Anfang der tokenisierten Eingabe) laden
,a610	85 7a	sta 7a	und in LB des CHRGET-Zeigers schreiben
,a612	60	rts	Ende der CRUNCH-Routine, Tokenisierungsergebnis steht im Systemeingabepuffer, Y enthält den Offset zur \$00-Endmarkierung des tokenisierten Textes

; FNDLIN-Routine

sucht zu einer Zeilennummer, die in \$14/\$15 übergeben wird, die Position im Speicher ist das Carry-Flag danach gesetzt, wurde die Zeile gefunden, bei C=0 ist sie noch nicht vorhanden; in \$5f/\$60 steht die Adresse, ab der die Zeile steht bzw. bei Nichtvorhandensein der Zeile die Adresse, ab der die Zeile mit der nächsthöheren Nummer steht, d.h., ab wo eine Einfügung in den Basic-Speicher erfolgen müßte; wird bei \$a4a4 (Einfügung einer Basic-Zeilenangabe) und \$a6a7 (LIST-Routine) verwendet.

,a613	a5 2b	lda 2b	LB des Zeigers auf den Basic-Programmanfang im Speicher holen	} Suche am Basic- Anfang beginnen
,a615	a6 2c	ldx 2c	HB des Zeigers auf den Basic-Programmanfang im Speicher holen	
,a617	a0 01	ldy #01	Offset auf HB des Linkpointers stellen (enthält mögliche Endmarkierung)	
,a619	85 5f	sta 5f	neues LB in Hilfszeiger \$5f/\$60 schreiben	} Hilfszeiger \$5f/\$60 (Basis der zu prüfenden Basic-Zeile) setzen
,a61b	86 60	stx 60	neues HB in Hilfszeiger \$5f/\$60 schreiben	
,a61d	b1 5f	lda (5f),y	HB des Zeilenlinkpointers auslesen	
,a61f	f0 lf	beq a640	= \$00 (Programmende) (Z=1): Routine verlassen, vorher wird Carry gelöscht	

,a621	c8	iny "ldy #02"	Offset=Offset+1	} Offset auf 3 stellen (vorher 1), um ihn auf HB der Zeilennummer der gerade zu untersuchenden Basic-Zeile zu stellen
,a622	c8	iny "ldy #03"	Offset=Offset+1	
,a623	a5 15	lda 15	HB der gesuchten Zeilennummer in Akku holen	
,a625	d1 5f	cmp (5f),y	Vergleich mit HB der gerade untersuchten Zeilennummer im Speicher	
,a627	90 18	bcc a641	gewünschte Zeilennummer-HB ist kleiner (C=0): RTS bei gelöschtem Carry-Flag	
,a629	f0 03	beq a62e	gewünschte Zeilennummer-HB = aktuell untersuchte Zeilennummer-HB (Z=1): LBs testen	
,a62b	88	dey "ldy #02"	Offset=Offset-1, also auf LB der Zeilennummer stellen	
,a62c	d0 09	bne a637 "jmp"	LBs testen	

,a62e	a5 14	lda 14	LB der gesuchten Zeilennummer holen	
,a630	88	dey "ldy #02"	Offset=Offset-1, also auf LB der Zeilennummer stellen	
,a631	d1 5f	cmp (5f),y	Vergleich von gesuchtem LB mit aktuellem LB der Zeilennummer	
,a633	90 0c	bcc a641	gesuchtes LB < aktuelles LB (C=0): RTS bei gelöschtem Carry-Flag	
,a635	f0 0a	beq a641	gesuchtes LB = aktuelles LB (Z=1): RTS bei gesetztem (s. \$a633) Carry-Flag	
,a637	88	dey "ldy #01"	Offset=Offset-1, also auf HB des Linkpointers (vorher Y=2, jetzt Y=1) stellen	
,a638	b1 5f	lda (5f),y	HB der Adresse der nächsten Zeile aus Linkpointer-HB entnehmen	
,a63a	aa	tax	in X-Register laden (wird bei \$a61b vorausgesetzt)	
,a63b	88	dey "ldy #00"	Offset=Offset-1, also auf LB des Linkpointers (vorher Y=1, jetzt Y=0) stellen	
,a63c	b1 5f	lda (5f),y	LB der Adresse der nächsten Zeile aus Linkpointer-LB entnehmen (LB im Akku wird bei \$a619 vorausgesetzt)	
,a63e	b0 d7	bcs a617 "jmp"	Suche mit nächster Zeile im Speicher fortsetzen (Adresse in A/X)	

,a640	18	clc	Carry löschen (Zeichen für "Zeile nicht vorhanden")	
,a641	60	rts	Rücksprung von Routine (C=0: Zeile nicht vorhanden/C=1: Zeile gefunden)	

; Routine zum Basic-Befehl NEW (Token: \$a2)				
,a642	d0 fd	bne a641	nächstes Zeichen nach NEW kein Trennzeichen (\$00 oder Doppelpunkt \$3a) (Z=0): RTS dadurch wird eine Eingabe wie "NEW 5" mit der Meldung SYNTAX ERROR bedacht, da die über RTS angesprungene Interpreterschleife den Befehl "5" nicht ausführen kann	

; hier: Einstieg von \$e444 aus der MSGNEW-Routine

,a644	a9 00	lda #00	Nullbyte (Endmarkierung eines Programms) für Schreiben an Programmstart laden	
,a646	a8	tay "ldy #00"	auch Y-Register (Offset von Programmstart für Schreiben von \$00-Bytes) mit 0 laden	
,a647	91 2b	sta (2b),y	\$00 in erstes Byte am Programmstart schreiben (\$2b/\$2c: Zeiger auf Basic-Anfang)	
,a649	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf zweites Byte am Basic-Anfang stellen)	
,a64a	91 2b	sta (2b),y	\$00 in zweites Byte am Programmstart schreiben	
,a64c	a5 2b	lda 2b	LB des Basic-Anfangs aus Zeiger \$2b/\$2c auslesen	
,a64e	18	clc	Carry vor Addition löschen	

,a64f	69 02	adc #02	2 (Anzahl der Nullbytes am Anfang = Länge des gelöschtenProgramms) addieren	
,a651	85 2d	sta 2d	als Programmende-LB setzen	
,a653	a5 2c	lda 2c	HB des Basic-Anfangs aus Zeiger \$2b/\$2c auslesen	
,a655	69 00	adc #00	eventuellen Übertrag der Addition von 2 (s. \$a64f) berücksichtigen	
,a657	85 2e	sta 2e	als Programmende-HB setzen	
; hier: NEWCLR-Einsprung				
,a659	20 8e a6	jsr a68e "stxtpt"	CHRGET-Zeiger auf Basic-Anfang setzen (Initialisierung der CHRGET-Zeiger)	
,a65c	a9 00	lda #00	Akku mit \$00 laden, um Zero-Flag zu setzen und gleichzeitig Endekennzeichnung der Befehlszeile bzw. Direkteingabe vorzutauschen, damit auf NEW folgender Befehl ignoriert wird; im Speicher folgt unmittelbar die Routine zum CLR-Befehl	
; Routine zum Basic-Befehl CLR (Token: \$9c)				
,a65e	d0 2d	bne a68d	weder \$00 noch Doppelpunkt \$3a hinter CLR, also unerlaubte Parameter (Z=0): RTS, wodurch letztendlich in der Interpreterschleife ein SYNTAX ERROR erzeugt wird, da Eingaben nach CLR ungültig sind	
; hier: Einsprung aus RUN-Routine				
,a660	20 e7 ff	jsr ffe7 "clall"	alle möglicherweise offenen Kanäle aus Filetabelle entfernen	
; hier: Einsprung aus EREXIT (Fehlerbehandlung nach I/O-Routinen des Basic-Interpreters)				
,a663	a5 37	lda 37	LB des Zeigers auf die Basic-RAM-Obergrenze laden	} oberste Adresse des Basic-RAM in Zeiger für Stringbereich-Obergrenze schreiben Zeiger auf Anfangs- und Endadresse der Arrays im Speicher mit dem Ende des Basic-Programms laden (Initialisierung)
,a665	a4 38	ldy 38	HB des Zeigers auf die Basic-RAM-Obergrenze laden	
,a667	85 33	sta 33	LB des Zeigers auf Anfang der Stringspeicherung setzen	
,a669	84 34	sty 34	HB des Zeigers auf Anfang der Stringspeicherung setzen	
,a66b	a5 2d	lda 2d	LB des Zeigers auf das Basic-Programmende laden	
,a66d	a4 2e	ldy 2e	HB des Zeigers auf das Basic-Programmende laden	
,a66f	85 2f	sta 2f	LB des Zeigers auf die Array-Anfangsadresse setzen	
,a671	84 30	sty 30	HB des Zeigers auf die Array-Anfangsadresse setzen	
,a673	85 31	sta 31	LB des Zeigers auf die Endadresse der Arrays + 1 setzen	
,a675	84 32	sty 32	HB des Zeigers auf die Endadresse der Arrays + 1 setzen	
,a677	20 ld a8	jsr a8ld	Routine zum Basic-Befehl RESTORE aufrufen	

; Aufruf von \$a462 (ERROR) und \$e382 (Basic-NMI)

,a67a	a2 19	ldx #19 *(\$19)	25 (Initialisierungswert für Stringstapelzeiger) laden	} Zeiger für temporären Stringstapel initialisieren
,a67c	86 16	stx 16	in Zeiger für temporären Stringstapel schreiben	
,a67e	68	pla	LB der Rücksprungadresse der aufrufenden Routine vom Stapel holen	} Rücksprung- adresse nach
,a67f	a8	tay	und im Y-Register merken, da Akku gleich wieder benötigt wird	
,a680	68	pla	HB der Rücksprungadresse der aufrufenden Routine vom Stapel holen	} Y/A holen
,a681	a2 fa	ldx #fa	Initialisierungswert für Stapelzeiger laden	
,a683	9a	txs	und in Stapelzeiger der CPU schreiben	} mit \$fa initialisieren
,a684	48	pha	HB der Rücksprungadresse der aufrufenden Routine wieder auf Stapel legen (s. \$a680)	
,a685	98	tya	LB der Rücksprungadresse der aufrufenden Routine in Akku (s. \$a67e/\$a67f) holen	} bei \$a67e-\$a680 vom Stapel geholte Rücksprungadresse wieder auf den Stapel legen; Rettung war erforderlich wegen Stapelzeiger-Initialisierung
,a686	48	pha	und dann wieder auf den Stapel legen	
,a687	a9 00	lda #00	Initialisierungswert \$00 für CONT- und FN-Flag laden	
,a689	85 3e	sta 3e	HB des CONT-Zeigers mit \$00 belegen = CONT sperren (CAN'T-CONTINUE-Zustand)	
,a68b	85 10	sta 10	Flag für benutzerdefinierte Funktion FN mit \$00 belegen (keine Funktion verfügbar)	
,a68d	60	→rts	Rücksprung von Routine; wegen Rücksprungadresse: s. \$a67e-\$a686 !	

; STXTPT-Routine (setzt CHRGET-Zeiger auf Anfangsadresse des Basic-Programms);

wird vom NEW-Befehl bei \$a659 und vom LOAD-Befehl bei \$elb5 als Unteroutine aufgerufen

,a68e	18	clc	Carry vor Addition bei \$a691 löschen
,a68f	a5 2b	lda 2b	LB des Zeigers auf den Basic-Programmstart in Akkumulator holen
,a691	69 ff	adc #ff	\$ff addieren; ist Ersatz für Subtraktion von 1!
,a693	85 7a	sta 7a	LB des CHRGET-Zeigers setzen
,a695	a5 2c	lda 2c	HB des Zeigers auf den Basic-Programmstart in Akkumulator holen
,a697	69 ff	adc #ff	eventuellen Übertrag der Subtraktion von 1 bei \$a691 berücksichtigen
,a699	85 7b	sta 7b	HB des CHRGET-Zeigers setzen
,a69b	60	rts	Rücksprung von der Routine; jetzt enthält \$7a/\$7b die Anfangsadresse - 1 (!)

; Routine zum Basic-Befehl LIST (Token: \$9b)

,a69c	90 06	bcc a6a4 beq a6a4 cmp #ab bne a68d	auf LIST folgt eine Ziffer (C=0): LIST-Parametereauswertung für "Ziffer nach LIST"
,a69e	f0 04		Endmarkierung folgt (\$00 oder \$3a=Doppelpunkt) (Z=1): LIST-Parametereauswertung
,a6a0	c9 ab		auf LIST folgt Token für "-" ?
,a6a2	d0 e9		nein (Z=0): RTS anspringen, da weder Ziffer noch "-" noch Endmarkierung nach LIST
,a6a4	20 6b a9	→jsr a96b "linget"	erste Zeilennummer nach LIST nach \$14/\$15 holen
,a6a7	20 13 a6	jsr a613 "fndlin"	diese oder (falls nicht vorhanden) nächste Zeile suchen, Adresse nach \$5f/\$60

,a6aa	20 79 00	jsr 0079 "chrgot"	letztes Zeichen erneut in Akku holen
,a6ad	f0 0c	beq a6bb	Endmarkierung (\$00 oder \$3a=Doppelpunkt) (Z=1): nur angegebene Zeile listen
,a6af	c9 ab	cmp #ab	war es das Token für "-"?
,a6b1	d0 8e	↖ bne a641	nein (Z=0): RTS anspringen, da unerlaubtes Zeichen; Interpreterschleife sorgt für SYNTAX ERROR
,a6b3	20 73 00	jsr 0073 "chrget"	CHRGET-Zeiger auf nächstes Byte stellen (Vorbereitung für "jsr linget")
,a6b6	20 6b a9	jsr a96b "linget"	zweite Zeilennummer nach \$14/\$15 holen
,a6b9	d0 86	↖ bne a641	keine Endmarkierung hinter zweiter Zeilennummer (Z=0): RTS anspringen (SYNTAX ERROR)
,a6bb	68	↗ pla	LB der Rücksprungadresse vom Stapel holen } Rücksprungadresse vom Stapel löschen,
,a6bc	68	pla	HB der Rücksprungadresse vom Stapel holen } ohne sie irgendwie zu speichern
,a6bd	a5 14	lda 14	LB der Startzeilennummer holen } testet, ob \$14 und \$15 den Wert 0 haben,
,a6bf	05 15	ora 15	mit HB verknüpfen } also, ob in \$14/\$15 der Wert \$0000 steht
,a6c1	d0-06	bne a6c9	nicht \$0000 als Startzeilennummer angegeben (Z=0): Sonderbehandlung "Liste bis Ende" überspringen, die nur bei "LIST 0" in Kraft tritt
,a6c3	a9 ff	lda #ff	\$ffff ist die höchste 2-Byte-Zahl und dient dem Listen bis zum Programmende
,a6c5	85 14	sta 14	LB auf \$ff setzen } \$14/\$15 mit
,a6c7	85 15	sta 15	HB auf \$ff setzen } \$ffff belegen
,a6c9	a0-01	→ ldy #01	Offset und Initialisierungswert für Quote-Mode-Flag der LIST-Routine laden
,a6cb	84 0f	sty 0f	Quote-Mode-Flag der LIST-Routine löschen (initialisieren)
,a6cd	b1 5f	lda (5f),y	HB des Linkpointers der aktuell zu listenden Zeile holen
,a6cf	f0-43	beq a714	Endmarkierung \$00 (Z=1): LIST beenden und in Basic-Warmstart springen (!)
,a6d1	20 2c a8	jsr a82c "bstop"	Basic-Routine für Testen der STOP-Taste (ggf. Abbruch über "BREAK IN xxxx")
,a6d4	20 d7 aa	jsr aad7	Ausgabe von [CR] und ggf. zusätzlich [LF] als Interpreter-Unterroutine
,a6d7	c8	iny "ldy #02"	Offset erhöhen (von 1 auf 2, also auf LB der Zeilennummerstellen)
,a6d8	b1 5f	lda (5f),y	LB der Zeilennummer in Akku holen
,a6da	aa	tax	und in X-Register merken
,a6db	c8	iny "ldy #03"	Offset erhöhen (von 2 auf 3, also auf HB der Zeilennummer stellen)
,a6dc	b1 5f	lda (5f),y	HB der Zeilennummer in Akku holen
,a6de	c5 15	cmp 15	Vergleich der HBs von aktueller Zeilennummer (Akku) und Endzeilennummer (\$14/\$15)
,a6e0	d0 04	bne a6e6	keine Übereinstimmung (Z=0): auch Carry-Flag auf "größer" testen, ggf. Ende von LIST
,a6e2	e4 14	cpx 14	LBs von aktueller Zeilennummer (X) und Endzeilennummer (\$14/\$15) vergleichen
,a6e4	f0 02	beq a6e8	auch hier Übereinstimmung (Z=1): Zeile noch ausgeben
,a6e6	b0 2c	↗ bcs a714	aktueller Wert größer als Endwert (C=1): LIST beenden
,a6e8	84 49	→ sty 49	Offset aus Y-Register in \$49 zwischenspeichern
,a6ea	20 cd bd	jsr bdc4 "numout"	Zeilennummer über NUMOUT (gibt Integerzahl in X=LB/A=HB aus) ausgeben
,a6ed	a9 20	lda #20	Code für Ausgabe von Leerzeichen hinter Zeilennummer vorbereiten (s. \$a6f3)
,a6ef	a4 49	ldy 49	geretteten Offset wieder aus \$49 entnehmen (s. \$a6e8)
,a6f1	29 7f	and #7f %01111111	b7 im auszugebenden Byte löschen (Endmarkierung nicht ausgeben)
,a6f3	20 47 ab	jsr ab47 "bbsout"	Zeichen über Basic-Einsprung für BSOUT ausgeben
,a6f6	c9 22	cmp #22	war das Zeichen ein Anführungszeichen (ASCII-Code #34 = \$22)?
,a6f8	d0 06	↖ bne a700	nein (Z=0): Überspringen des Quote-Mode-Flag-Umdrehens

,a6fa	a5 0f	lda 0f	Quote-Mode-Flag der LIST-Routine auslesen	} bei Ausgabe eines Anführungszeichens Quote-Mode-Umschaltung
,a6fc	49 ff	eor #ff %l1l1l1l1l1	flippen (invertieren, umdrehen)	
,a6fe	85 0f	sta 0f	und geflippten Wert in Quote-Mode-Flag zurückschreiben	
,a700	c8	→iny	Offset=Offset+1 (auf nächstes Byte zur Ausgabe stellen)	
,a701	f0 11	beq a714	Offset war vorher \$ff und ist jetzt \$00 (Z=1): LIST-Routine verlassen, da 255 Byte das absolute Limit für Basic-Zeilen im Speicherformat ist	
,a703	b1 5f	lda (5f),y	Byte aus Basic-Zeile holen	
,a705	d0 10	bne a717 "qplop"	keine \$00-Zeilenendmarkierung (Z=0): Token entschlüsseln	
,a707	a8	tay "ldy #00"	Akku muß hier dank \$a705 auf \$00 stehen, also "LDY#0"-Simulation	
,a708	b1 5f	lda (5f),y	LB des Linkpointers der aktuellen Zeile holen	
,a70a	aa	tax	und in X-Register merken (s. \$a70e)	
,a70b	c8	iny "ldy #01"	Offset erhöhen (von 0 auf 1, also auf HB des Linkpointers stellen)	
,a70c	b1 5f	lda (5f),y	HB des Linkpointers der aktuellen Zeile holen (s. \$a710)	
,a70e	86 5f	stx 5f	LB der Adresse der nächsten Zeile (Linkpointer-LB) in Hilfszeiger \$5f/\$60 schreiben	
,a710	85 60	sta 60	HB der Adresse der nächsten Zeile (Linkpointer-HB) in Hilfszeiger \$5f/\$60 schreiben	
,a712	d0 b5	bne a6c9	HB keine Endmarkierung \$00 (Z=0): weiter in LIST-Schleife	
,a714	4c 86	e3→jmp e386	Sprung in Basic-Warmstart (READY.), da Rücksprungadresse vom Stapel gelöscht wurde und somit die Möglichkeit des RTS-Rücksprungs verbaut ist (s. \$a6bb/\$a6bc)	

; QPLOP-Routine: gibt Byte in Akku aus, wobei Tokens in Klartext ausgegeben werden

,a717	6c 06	03→jmp(0306)	Sprung über Vektor IQPLOP; zeigt normalerweise auf \$a71a
,a71a	10 d7	←bpl a6f3	b7 im Ausgabe-Byte gelöscht (N=0): Zeichen uncodiert über Basic-BSOUT ausgeben
,a71c	c9 ff	cmp #ff	π-Token?
,a71e	f0 d3	←beq a6f3	ja (Z=1): Zeichen ebenfalls über Basic-BSOUT-Einsprung ausgeben
,a720	24 0f	bit 0f	Quote-Mode-Flag der LIST-Routine testen
,a722	30 cf	←bmi a6f3	gesetzt (N=1): Zeichen ebenfalls über Basic-BSOUT-Einsprung ausgeben
,a724	38	sec	Carry vor Subtraktion setzen
,a725	e9 7f	sbc #7f	Tokens beginnen bei \$80, \$7f wird zwecks Offset-Entfernung subtrahiert
,a727	aa	tax	Ergebnis ins X-Register als Zähler für Position in Befehlswörtertable
,a728	84 49	sty 49	und Offset in Basic-Programmcode in \$49 zwischenspeichern
,a72a	a0 ff	ldy #ff	Initialisierungswert für Y (Offset in Befehlswörtertable), wird bei \$a72f zu \$00
,a72c	ca	→dex	Zähler in Token-Tabelle dekrementieren
,a72d	f0 08	beq a737	schon richtigen Eintrag in Tabelle gefunden (Z=1): zur Klartext-Ausgabeschleife
,a72f	c8	→iny	Offset in Befehlswörtertable erhöhen, um nächstes Byte zu untersuchen
,a730	b9 9e	a0 lda a09e,y	Byte aus Befehlswörtertable holen
,a733	10 fa	bpl a72f	kein Byte mit b7-Endmarkierung (N=0): weiter suchen, ohne X-Zähler zu dekrementieren
,a735	30 f5	bmi a72c "jmp"	Offset erst dekrementieren, dann eventuell (!) weiter suchen

; gibt ab \$a09e,y stehenden Befehlswort-Klartext (b7 als Endmarkierung) aus

,a737	c8	→iny	Offset in Befehlswörtertable erhöhen
,a738	b9 9e	a0 lda a09e,y	Klartext-Byte aus ROM-Tabelle entnehmen
,a73b	30 b2	↖bmi a6ef	Endmarkierung enthalten (N=1): b7 ausblenden, ausgeben und weiter in LIST-Schleife
,a73d	20 47	ab jsr ab47	Zeichen über Basic-Einsprung für BSOUT ausgeben
,a740	d0 f5	↖bne a737 !"jmp!"	hier liegt kleine Ungenauigkeit vor (s. Fließtext); ist bei einem Token von \$cckkein (!!!!!) Pseudo-JMP, ansonsten aber funktioniert es; Grund für <SHIFT L>-Trick

; Routine zum Basic-Befehl FOR (Token: \$8f)

,a742	a9 80	lda #80 %10000000	Flag für "Parameter dürfen nicht Integerzahlen sein" laden	} Verhinderung von "FOR A%=" usw.
,a744	85 10	sta 10	Flag für LET-Routine setzen	
,a746	20 a5 a9	jsr a9a5 "let"	Routine zum LET-Befehl aufrufen, damit Schleifenindexvariable angelegt wird	
,a749	20 8a a3	jsr a38a "srcstk"	Stapelsuchroutine des Basic-Interpreters aufrufen; sucht jetzt FOR-NEXT-Eintrag mit gleicher Indexvariablen wie der übergebenen	
,a74c	d0 05	bne a753	nicht gefunden (Z=0): keine Sonderbehandlung erforderlich	
,a74e	8a	txa	X-Register enthält nach \$a749 Stapelzeiger, also kommt effektiv Stapelzeiger in Akku	} Stapelzeiger um 15 erhöhen
,a74f	69 0f	adc #0f	15 zum Stapelzeiger addieren (Carry ist seit \$a749 gelöscht)	
,a751	aa	tax	Ergebnis ins X-Register bringen	
,a752	9a	txs	und von dort in den Stapelzeiger	
,a753	68	↗pla	LB der Rücksprungadresse auf dem Stapel löschen	} Rücksprungposition vom Stapel entfernen
,a754	68	pla	HB der Rücksprungadresse auf dem Stapel löschen	
,a755	a9 09	lda #09	auf \$09*2 = #18 Byte Speicherplatz auf Stapel prüfen	
,a757	20 fb a3	jsr a3fb "getstk"	Prüfroutine auf freien Stapelplatz (erzeugt ggf. OUT OF MEMORY)	
,a75a	20 06 a9	jsr a906 "gofnxt"	Offset zum nächsten Basic-Befehl (von CHRGET-Zeiger aus) in Y-Register holen	
,a75d	18	clc	Carry vor Addition bei \$a75f löschen	
,a75e	98	tya	Offset in Akku zwecks Addition bringen	} CHRGET-Zeiger auf nächsten Befehl hinter FOR, also auf den Inhalt der Schleife, auf den Stapel legen
,a75f	65 7a	adc 7a	zum LB des CHRGET-Zeigers addieren	
,a761	48	pha	Ergebnis als Rücksprungposition für Schleifeninhalt setzen	
,a762	a5 7b	lda 7b	HB des CHRGET-Zeigers holen	
,a764	69 00	adc #00	möglichen Übertrag (s. \$a75f) berücksichtigen	
,a766	48	pha	Ergebnis als Rücksprungposition für Schleifeninhalt setzen	
,a767	a5 3a	lda 3a	HB der aktuellen Basic-Zeilenummer holen	} Nummer der aktuellen Basic-Zeile, die den FOR-Befehl enthält, auf den Stapel legen
,a769	48	pha	und auf Stapel legen	
,a76a	a5 39	lda 39	LB der aktuellen Basic-Zeilenummer holen	
,a76c	48	pha	und auf Stapel legen	
,a76d	a9 a4	lda #a4	Token für T0 als Testbyte laden	} stellt sicher, daß T0-Befehl hinter "FOR .=" steht
,a76f	20 ff ae	jsr aeef "chkbyt"	auf Vorhandensein im Basic-Text prüfen	


```

,a772 20 8d ad jsr ad8d "chknum" prüft, ob auf T0 folgender Parameter numerisch ist (sonst SYNTAX ERROR)
,a775 20 8a ad jsr ad8a "frmnum" holt numerischen Wert (hier: Schleifenendwert) in den FAC #1
,a778 a5 66 lda 66 Vorzeichen von FAC #1 in Akku holen
,a77a 09 7f ora #7f %01111111 b0-b6 setzen, b7 von $a778 übernehmen
,a77c 25 62 and 62 mit erstem Mantissenbyte ANDen (enthält ebenfalls Vorzeichen)
,a77e 85 62 sta 62 und Ergebnis ins erste Mantissenbyte schreiben
,a780 a9 8b lda #8b <($a78b) LB von $a78b (Rücksprungadresse) laden
,a782 a0 a7 ldy #a7 >($a78b) HB von $a78b (Rücksprungadresse) laden
,a784 85 22 sta 22 LB der Rücksprungadresse übergeben
,a786 84 23 sty 23 HB der Rücksprungadresse übergeben
,a788 4c 43 ae jmp ae43 "facstk" FAC #1 auf Stapel legen und an Rücksprungadresse in $22/$23 (hier $a78b) springen

```

; hierher wird über \$a780-\$a788 gesprungen

```

,a78b a9 bc lda #bc <($b9bc) LB von $b9bc (Adresse von ROM-Konstante 1) laden
,a78d a0 b9 ldy #b9 >($b9bc) HB von $b9bc (Adresse von ROM-Konstante 1) laden
,a78f 20 a2 bb jsr bba2 "movmf" Konstante 1 von $bcb9 in FAC #1 holen (als Voreinstellung für STEP-Wert)
,a792 20 79 00 jsr 0079 "chrgot" letztes Zeichen in Akku holen, um auf STEP zu testen
,a795 c9 a9 cmp #a9 STEP-Token?
,a797 d0 06 bne a79f nein (Z=0): keinen numerischen Parameter holen, Default-Wert aus FAC #1
(Konstante 1) übernehmen
,a799 20 73 00 jsr 0073 "chrget" CHRGET-Zeiger auf nächstes Byte nach STEP stellen
,a79c 20 8a ad jsr ad8a "frmnum" numerischen Parameter (STEP-Wert) in FAC #1 anstelle von Konstante 1 holen
,a79f 20 2b bc jsr bc2b "sign" STEP-Wert (ggf. Default-Wert 1) aus FAC #1 auswerten: Vorzeichen in Akku ($01 steht
für positiv, $ff für negativ und $00 für Null-Wert)
,a7a2 20 38 ae jsr ae38 Vorzeichen aus Akku und STEP-Wert aus FAC #1 auf den Stapel legen
,a7a5 a5 4a lda 4a HB des FOR/NEXT-Variablenzeigers holen
,a7a7 48 pha und auf den Stapel legen
,a7a8 a5 49 lda 49 LB des FOR/NEXT-Variablenzeigers holen
,a7aa 48 pha und auf den Stapel legen
,a7ab a9 81 lda #81 FOR-Token als FOR/NEXT-Stapeleintrag-Kennzeichnung laden
,a7ad 48 pha und auf den Stapel legen; im Speicher folgt jetzt unmittelbar die
Interpreter-Schleife, weshalb kein Rücksprung erfolgt und die Rücksprungadresse am
Stapel entfernt wurde (s. $a753/$a754)

```

; INTPRT: Interpreter-Schleife (liest über CHRGET nächstes Zeichen aus Basic-Text ein und sorgt für Befehlsausführung);
außer unmittelbarer Ausführung hinter \$a7ad auch JMP von \$a7ea (Rücksprung an Start von INTPRT), \$a89d (GOSUB) und
\$ad75 (NEXT)

```

,a7ae 20 2c a8 jsr a82c "bstop" Basic-Einsprung für Prüfung auf STOP-Taste (ggf. "BREAK IN xxxx")

```

,a7b1	a5 7a	lda 7a	LB des CHRGET-Zeigers in Akku	} CHRGET-Zeiger nach A/Y } holen
,a7b3	a4 7b	ldy 7b	HB des CHRGET-Zeigers in Y	
,a7b5	c0 02	cpy #02	HB mit 2 (HB von \$0200 = Systemeingabepuffer) vergleichen, also Test auf Direktmodus	
,a7b7	ea	nop	dieser Befehl wurde wahrscheinlich bei späterer Änderung eingefügt	
,a7b8	f0 04	beq a7be	Vergleich bei \$a7b5 positiv (Z=1): CHRGET-Zeiger nicht als Zeiger für CONT übernehmen	
,a7ba	85 3d	sta 3d	LB des CHRGET-Zeigers (s. \$a7b1) als LB des CONT-Zeigers setzen	
,a7bc	84 3e	sty 3e	HB des CHRGET-Zeigers (s. \$a7b3) als HB des CONT-Zeigers setzen	
,a7be	a0 00	ldy #00	Offset mit 0 belegen	
,a7c0	b1 7a	lda (7a),y	Byte an CHRGET-Position auslesen; wird nicht über CHRGET gelesen, da die Flags nicht von der CHRGET-Prüfung, sondern allein von der CPU gesetzt werden sollen	
,a7c2	d0 43	bne a807	Byte <> 0 (Z=0): weitere Prüfungen, zuallererst auf Doppelpunkt (ASCII-Code \$3a)	
,a7c4	a0 02	ldy #02	Offset = 2, also auf insgesamt 3 Nullbytes prüfen (das erste Nullbyte ist wegen \$a7c2 vorhanden)	
,a7c6	b1 7a	lda (7a),y	eventuelles 3. Nullbyte (= Programmendmarkierung) zwecks Test auslesen	
,a7c8	18	clc	Flag für möglicherweise erforderliche Ausführung von END setzen (0=END, C=1 wäre STOP)	
,a7c9	d0 03	bne a7ce	kein 3. Nullbyte, also nur Zeilen- und nicht schon Programmende (Z=0): Zeilenwechsel	
,a7cb	4c 4b a8	jmp a84b	Programmende, da Endmarkierung (letzte Zeile) erreicht, also END ausführen (s. \$a7c8)	

; Sonderbehandlung für den Fall, daß Zeilenende (nicht Programmende, s. \$a7be-\$a7c2) erreicht ist und CHRGET-Zeiger auf die nächste Zeile umzustellen sind (s. \$a7c9)

,a7ce	c8	iny "ldy #03"	Offset erhöhen: von 2 (s. \$a7c4, \$a7c8) auf 3, somit auf Zeilennummer-LB richten	
,a7cf	b1 7a	lda (7a),y	LB der Zeilennummer mit Y als Offset von der Endmarkierung der vorhergehenden Zeile auslesen	
,a7d1	85 39	sta 39	und in LB der aktuellen Basic-Zeilennummer (Hilfszeiger) schreiben	
,a7d3	c8	iny "ldy #04"	Offset erhöhen von 3 (s. \$a7ce) auf 4, somit auf HB der Zeilennummer richten	
,a7d4	b1 7a	lda (7a),y	HB der Zeilennummer mit Y als Offset von Endmarkierung der vorhergehenden Zeile auslesen	
,a7d6	85 3a	sta 3a	und in HB der aktuellen Basic-Zeilennummer (Hilfszeiger) schreiben	
,a7d8	98	tya "lda #04"	Akku mit 4 laden (s. \$a7d3), 4 = Anzahl der Bytes am Zeilenanfang	
,a7d9	65 7a	adc 7a	4 zum LB des CHRGET-Zeigers addieren	
,a7db	85 7a	sta 7a	und Ergebnis in LB	
,a7dd	90 02	bcc a7e1 "gone"	kein Additionsübertrag (C=0):	
,a7df	e6 7b	inc 7b	CHRGET-Zeiger-HB erhöhen	
			CHRGET-Zeiger-HB inkrementieren (als Einbeziehung von Überträgen)	
			CHRGET-Zeiger um 4 (Länge der Kopf-Markierung von 2 Linkpointer-Bytes und 2 Zeilennummer-Bytes addieren, um CHRGET-Zeiger von Endmarkierung der einen auf den ersten Befehl der anderen Zeile zu stellen	

; GONE-Routine: führt aktuelles Befehlsbyte (Kriterium: CHRGET-Zeiger) aus, wenn es sich um einen interpretierbaren Befehl (meist Basic-Kommando oder ähnliches) handelt, ansonsten Behandlung als Variablenzuweisung (über LET-Routine); wird auch von \$a499 am Ende der MAIN-Routine angesprungen

,a7e1 6c 08 03 → jmp(0308) Sprung über Vektor IGONE \$0308/\$0309, zeigt normalerweise auf \$a7e4

,a7e4 20 73 00 jsr 0073 "chrget" auszuführenden Befehlscode in Akku holen

,a7e7 20 ed a7 jsr a7ed Ausführung des Befehls im Akku

,a7ea 4c ae a7 jmp a7ae "intprt" zurück zur Interpreter-Schleife, nächsten Befehl bearbeiten und vorher <STOP> testen

; Routine zur Ausführung des im Akku befindlichen Befehlscodes, kehrt über RTS zurück, wenn Befehl ausgeführt wurde

,a7ed f0 3c ↙ beq a82b = 0, also Befehlsendmarkierung (Z=1): RTS anspringen, Ausführung unmöglich

; hierher auch Einstieg von \$a95e (Ende der Routine zum ON-Befehl)

,a7ef e9 80 sbc #80 %10000000 Offset für Tokens abziehen, um Vergleich und Auswertung über indizierte Tabellenbehandlung zu erleichtern

,a7f1 90 11 bcc a804 Subtraktionsübertrag (C=0): LET-Routine ausführen, da es kein Token war

; Befehl, dessen Nummer (Token - \$80) im Akku steht, über Befehlsroutinentabelle ausführen

,a7f3 c9 23 cmp #23 war Token größer als \$80 (s. \$a7ef) + \$23 = \$a3 (höchstes Token für Kommando ist \$a2, ab \$a3 bis \$cc handelt es sich um Basic-Funktionen)?

,a7f5 b0 17 bcs a80e ja (C=1): Funktionen ablehnen (nicht für GONE verwendbar), mögliches "GO TO" erkennen

,a7f7 0a asl Offset gilt auch für Tabelle und wird hier mit 2 multipliziert, da es sich um eine 2-Byte-Tabelle (Low-High-Format für Routinenadressen der Kommandos) handelt

,a7f8 a8 tay Ergebnis als Offset in Y-Register

,a7f9 b9 0d a0 lda a00d,y HB aus Tabelle auslesen } Adresse zur Befehlsroutine aus

,a7fc 48 pha und auf Stapel legen } Tabelle ab \$a00c auslesen und

,a7fd b9 0c a0 lda a00c,y LB aus Tabelle auslesen } auf den Stapel legen, damit sie beim

,a800 48 pha und auf Stapel legen } nächsten RTS ausgeführt wird

,a801 4c 73 00 jmp 0073 "chrget" CHRGET anspringen: nächstes Zeichen hinter Befehl in Akku holen; JMP statt JSR, damit beim CHRGET-JSR-Befehl am Ende der CHRGET-Routine zur am Stapel liegenden Adresse (s. \$a7f9—\$a800) verzweigt und die Befehlsroutine aufgerufen wird; beideren RTS erfolgt Rücksprung in die Routine, die \$a7f3 aufgerufen hat, also nach \$a7ea

; Sonderbehandlung: auszuführender Code ist kein Token, sondern eine Variablenzuweisung ohne LET oder ein Syntax-Fehler, den die LET-Routine erkennen muß
(Aufruf über \$a7f1)

,a804 4c a5 a9 → jmp a9a5 "let" LET-Befehl ausführen, um Konstruktionen wie "A=5" anstelle von "LET A=5" auszuführen

; Sonderbehandlung: Befehls-Endmarkierung wurde an die GONE-Routine übergeben und bei \$a7c2 erkannt

,a807 c9 3a cmp #3a Doppelpunkt und \$00 durch Vergleich mit ASCII-Code von Doppelpunkt unterscheiden
,a809 f0 d6 ^_beq a7e1 "gone" Doppelpunkt statt \$00 (Z=1): zurück an Interpreter-Schleife, daraufhin wird nächstes Zeichen bearbeitet
,a80b 4c 08 af → jmp af08 "synerr" \$00: SYNTAX ERROR (GONE arbeitet \$00 nicht ab, dafür ist die Interpreter-Schleife zuständig, wie man bei \$a7c0 usw. sieht!)

; Sonderbehandlung: gültiges Token wurde an GONE übergeben, das aber kein Kommando, sondern eine Funktion oder einen Operator repräsentiert

,a80e c9 4b → cmp #4b ist Token = \$80 (s. \$a7ef) + \$4b = \$cb (Token von G0 = höchstes Befehlstoken)
,a810 d0 f9 → bne a80b nein (Z=0): dann Funktion (Tokens \$a3-\$ca) oder unerlaubt (\$cc-\$ff), also SYNTAX ERROR erzeugen
,a812 20 73 00 jsr 0073 "chrget" CHRGET-Zeiger auf nächsten Code hinter G0-Token stellen
,a815 a9 a4 lda #a4 Token für T0 laden } auf T0 hinter G0
,a817 20 ff ae jsr ae ff "chkbyt" und nächstes Byte prüfen, ggf. SYNTAX ERROR } prüfen
,a81a 4c a0 a8 jmp a8a0 "gotoln" Routine für GOT0-Befehl ausführen

; Routine zum Basic-Befehl RESTORE (Token: \$8c);
wird auch von \$a677 (CLR-Routine) aufgerufen

,a81d 38	sec	Carry vor Subtraktion setzen (s. \$a820)	} belegt den DATA-Zeiger \$41/\$42 mit der Anfangsadresse des aktuellen Basic-Programms minus 1 zur Initialisierung
,a81e a5 2b	lda 2b	LB des Zeigers auf Anfang des Basic-Programms im Speicher auslesen	
,a820 e9 01	sbc #01	davon 1 abziehen und Ergebnis in Akku lassen	
,a822 a4 2c	ldy 2c	HB des Zeigers auf Anfang des Basic-Programms im Speicher auslesen	
,a824 b0 01	bcs a827	kein Übertrag bei Subtraktion von 1 bei \$a820 (C=1): nicht HB=HB-1	
,a826 88	dey	HB dekrementieren, um Subtraktionsübertrag zu berücksichtigen	
,a827 85 41	→ sta 41	LB (s. \$a820) in LB des DATA-Zeigers schreiben	
,a829 84 42	sty 42	HB (s. \$a822-\$a826) in HB des DATA-Zeigers schreiben	
,a82b 60	rts	Rücksprung von Routine	

; BSTOP: Basic-Einsprung für Testen und Reagieren auf gedrückte STOP-Taste;
 Aufruf aus LIST-Routine bei \$abdl und INTPRT bei \$a7ae

,a82c 20 e1 ff jsr ffel "stop" STOP-Taste durch Kernall-Routine testen lassen
 danach: Z=0 und C=0, wenn <STOP> nicht betätigt; Z=1 und C=1, falls <STOP> betätigt

; Routine zum Basic-Befehl STOP (Token: \$90)

,a82f b0 01 bcs a832 STOP-Prüfung von \$a82c stellte <STOP> fest (C=1): Löschen von C-Flag überspringen
 Carry-Flag löschen als Flag für Ausführung des END-Befehls im Gegensatz zur
 Ausführung der Testroutine ab \$a82c

; Routine zum Basic-Befehl END (Token: \$80)

,a831 18 clc Carry-Flag löschen als Flag für END-Befehl
 ,a832 d0 3c bne a870 Doppelbedeutung: bei "JSR \$a82c" ohne gedrückte STOP-Taste führt RTS zum Rücksprung
 der aufrufenden Test-Routine ohne weitere Auswirkungen; nach Aufruf der Routinen zu
 STOP (\$a82f) oder END (\$a831) ist das Z-Flag für den Fall, daß keine
 Befehlsendmarkierung folgt, gesetzt, was als Syntax-Verstoß nach einem späteren RTS
 von der Interpreter-Schleife erkannt wird

; Ausführung von STOP oder END je nach gesetztem (STOP) oder gelöschtem (END) Carry-Flag; wurde die STOP-Taste
 gedrückt, ist das STOP-Befehlsflag seit \$a82c von der Kernall-Routine aus gesetzt

,a834 a5 7a lda 7a LB des CHRGET-Zeigers holen } CHRGET-Zeiger nach A/Y
 ,a836 a4 7b ldy 7b HB des CHRGET-Zeigers holen } einlesen
 ,a838 a6 3a ldx 3a HB der aktuellen Zeilennummer holen
 ,a83a e8 inx um 1 erhöhen, um auf \$ff (Direktmodus-Flag) zu testen
 ,a83b f0 0c beq a849 war vorher \$ff (Z=1): Sonderbehandlungen für Programm-Modus überspringen
 ,a83d 85 3d sta 3d LB der Abbruch-Adresse im Basic-Text in Zeiger auf letzte Programmunterbrechung
 ,a83f 84 3e sty 3e HB der Abbruch-Adresse im Basic-Text in Zeiger auf letzte Programmunterbrechung
 ,a841 a5 39 lda 39 LB des Zeigers auf aktuelle Basic-Zeilenummer holen } Zeiger auf letzte
 ,a843 a4 3a ldy 3a HB des Zeigers auf aktuelle Basic-Zeilenummer holen } Abbruchzeile mit
 ,a845 85 3b sta 3b LB in LB des Zeigers auf Unterbrechungs-Zeilenummer schreiben } aktueller Zeilen-
 ,a847 84 3c sty 3c HB in HB des Zeigers auf Unterbrechungs-Zeilenummer schreiben } nummer laden
 ,a849 68 pla LB der Rücksprungadresse vom Stapel holen } Rücksprungadresse vom
 ,a84a 68 pla HB der Rücksprungadresse vom Stapel holen } Stapel löschen

; Einsprung von \$a7cb bei Programmende

```

,a84b a9 81    lda #81 <($a381)  LB der Adresse vom Text BREAK laden } mögliche Ausgabe von
,a84d a0 a3    ldy #a3 >($a381)  HB des Adresse vom Text BREAK laden } BREAK vorbereiten
,a84f 90 03    bcc a854          Abbruch über END-Befehl (C=0): Basic-Warmstart, Text BREAK nicht ausgeben
,a851 4c 69 a4 jmp a469          in Fehlerbehandlungsroutine einspringen, wo BREAK-Meldung und Abbruch erfolgt
-----
,a854 4c 86 e3 jmp e386          Einsprung für Basic-Warmstart (= Programmende) aufrufen
-----

```

; Routine zum Basic-Befehl CONT (Token: \$9a)

```

,a857 d0 17    bne a870          keine Endmarkierung hinter CONT-Befehl (Z=0): RTS anspringen zwecks SYNTAX ERROR
,a859 a2 1a    ldx #1a          Fehlercode für CAN'T CONTINUE vorbereiten
,a85b a4 3e    ldy 3e          HB des Zeigers auf CONT-Fortsetzungsstelle im Speicher holen
,a85d d0 03    bne a862          HB <> 0, also CONT nicht gesperrt (Z=0): Aufruf der Fehlermeldung überspringen
,a85f 4c 37 a4 jmp a437 "error" Fehlereinsprung des Basic-Interpreters aufrufen, erzeugt (wegen $a859) CAN'T CONTINUE
-----

```

; hier erfolgt Ausführung von CONT, wobei schon auf CAN'T CONTINUE geprüft wurde

```

,a862 a5 3d    lda 3d          LB des CONT-Zeigers laden
,a864 85 7a    sta 7a          und in LB des CHRGET-Zeigers schreiben
,a866 84 7b    sty 7b          HB des CHRGET-Zeigers mit HB des CONT-Zeigers (s. $a85b) belegen
,a868 a5 3b    lda 3b          LB des CONT-Zeilenzeigers laden
,a86a a4 3c    ldy 3c          HB des CONT-Zeilenzeigers laden
,a86c 85 39    sta 39          LB in LB des Zeigers für aktuelle Zeilennummer schreiben
,a86e 84 3a    sty 3a          HB in HB des Zeigers für aktuelle Zeilennummer schreiben
,a870 60      rts             Rücksprung von Routine
-----

```

; Routine zum Basic-Befehl RUN (Token: \$8a)

```

,a871 08      php            Prozessorstatus zunächst auf Stapel retten (wegen CHRGET-Prüfungsergebnis aus
                             Interpreter-Schleife: Z=0, wenn keine Endmarkierung auf END folgt)
,a872 a9 00    lda #00        0 (Flag für Programm-Modus) laden
,a874 20 90 ff jsr ff90 "setmsg" und in Interpreter-Flag schreiben
,a877 28      plp            Prozessorstatus wieder holen (s. $a871)
,a878 d0 03    bne a87d        auf RUN folgt weitere Angabe (Zeilennummer!) (Z=0): Sprung zur Sonderbehandlung
,a87a 4c 59 a6 jmp a659        in CLR einsteigen, wobei Programmzeiger gesetzt werden und nach dem RTS der
                             CLR-Routine die Interpreter-Schleife am Programmanfang weiterarbeitet
-----

```

```
,a87d 20 60 a6↳jsr a660 CLR-Routine ausführen lassen
,a880 4c 97 a8 jmp a897 an Ende der GOSUB-Routine zwecks GOTO-Aufruf einspringen; keine GOSUB-Nebenwirkungen
```

; Routine zum Basic-Befehl GOSUB (Token: \$8d)

```
,a883 a9 03 lda #03 6/2 in Akku laden
,a885 20 fb a3 jsr a3fb "getstk" auf 3 (6/2) freie Bytes auf Stapel prüfen (ggf. OUT OF MEMORY)
,a888 a5 7b lda 7b HB des CHRGET-Zeigers holen } CHRGET-Zeiger
,a88a 48 pha und auf den Stapel legen } auf den
,a88b a5 7a lda 7a LB des CHRGET-Zeigers holen } Stapel
,a88d 48 pha und auf den Stapel legen } legen
,a88e a5 3a lda 3a HB der aktuellen Zeilennummer holen } Zeiger auf aktuelle
,a890 48 pha und auf den Stapel legen } Zeilennummer
,a891 a5 39 lda 39 LB der aktuellen Zeilennummer holen } auf den Stapel
,a893 48 pha und auf den Stapel legen } legen
,a894 a9 8d lda #8d GOSUB-Stapeleintrag-Kennzeichnung (sinnvollerweise das GOSUB-Token $8d) laden
,a896 48 pha und auf den Stapel legen
```

; von \$a880: Einsprung aus RUN-Routine

```
,a897 20 79 00 jsr 0079 "chrgot" Zeichen nach GOSUB in Akku holen
,a89a 20 a0 a8 jsr a8a0 "gotoin" Routine zum Basic-Befehl GOTO aufrufen
,a89d 4c ae a7 jmp a7ae "intprt" unmittelbarer Sprung zur Interpreter-Schleife
```

; Routine zum Basic-Befehl GOTO (Token: \$89)

wird unmittelbar bei \$a89a (GOSUB) als UP aufgerufen, bei \$a8la (GONE) und \$a945 (THEN-Behandlung) angesprungen

```
,a8a0 20 6b a9 jsr a96b "linget" Zeilennummer hinter GOTO holen
,a8a3 20 09 a9 jsr a909 "gosend" Offset zur nächsten Basic-Zeile holen
,a8a6 38 sec Carry vor Subtraktion bei $a8a9 setzen
,a8a7 a5 39 lda 39 LB der aktuellen Zeilennummer holen } Subtraktion der
,a8a9 e5 14 sbc 14 und LB der Ziel-Zeilennummer abziehen } Ziel-Zeilennummer
,a8ab a5 3a lda 3a HB der aktuellen Zeilennummer holen } von der aktuellen
,a8ad e5 15 sbc 15 und HB der Ziel-Zeilennummer abziehen } Zeilennummer zwecks 16-Bit-Vergleich
,a8af b0 0b bcs a8bc kein Subtraktionsübertrag, also Ziel-Zeilennummer nicht größer (C=1): keine
Sonderbehandlung durch Addition des Offset zum CHRGET-Zeiger
,a8b1 98 tya Offset zwecks Addition in Akku holen
,a8b2 38 sec Carry vor Addition setzen, damit zusätzlich 1 addiert wird
,a8b3 65 7a adc 7a LB des CHRGET-Zeigers addieren
```

,a8b5	a6	7b	ldx 7b	HB des CHRGET-Zeigers holen
,a8b7	90	07	bcc a8c0	kein Additionsübertrag (C=0): neu berechneten Zeiger A/X weiterverarbeiten
,a8b9	e8		inx	HB um 1 erhöhen, damit Additionsübertrag berücksichtigt wird
,a8ba	b0	04	bcs a8c0 "jmp"	jetzt neu berechneten Zeiger A/X weiterverarbeiten

,a8bc	a5	2b	→lda 2b	LB des Zeigers auf Programmanfang holen	} Zeiger auf Programmanfang auslesen
,a8be	a6	2c	ldx 2c	HB des Zeigers auf Programmanfang holen	

; hier steht in A/X die Adresse der nächsten Zeile, wenn eine später im Speicher liegende Zeile anzuspringen ist, ansonsten in A/X die Adresse des Programmanfangs (erste Zeile), wenn die Zeile vor der aktuellen Zeile liegt. A/X entscheidet darüber, wo die Suche nach der entsprechenden Zeile begonnen wird.

,a8c0	20	17	a6	→jsr a617	in FNDLIN so einsteigen, daß die Suche bei A/X begonnen wird (eventuelle Beschleunigung der Suche, wenn nicht unbedingt am Programmanfang gestartet wird!)
,a8c3	90	1e		bcc a8e3	Zeile nicht gefunden, also nicht vorhanden (C=0): zu Einsprung für UNDEF'D STATEMENT
,a8c5	a5	5f		lda 5f	LB der berechneten Zeilenadresse in Akku holen
,a8c7	e9	01		sbc #01	1 subtrahieren, Carry ist seit \$a8c3 logischerweise gesetzt
,a8c9	85	7a		sta 7a	Ergebnis in CHRGET-Zeiger-LB
,a8cb	a5	60		lda 60	HB der berechneten Zeilenadresse in Akku holen
,a8cd	e9	00		sbc #00	eventuellen Übertrag von \$a8c7 berücksichtigen
,a8cf	85	7b		sta 7b	Ergebnis in CHRGET-Zeiger-HB
,a8d1	60			→rts	Rücksprung von Routine nach Veränderung des nun relevanten CHRGET-Zeigers

; Routine zum Basic-Befehl RETURN (Token: \$8e)

,a8d2	d0	fd		bne a8d1	auf RETURN folgt weiteres Zeichen außer Endmarkierung (Z=0): RTS zwecks SYNTAX ERROR
,a8d4	a9	ff		lda #ff %11111111	Flag für "FOR/NEXT-Variablenzeiger bedeutungslos" laden
,a8d6	85	4a		sta 4a	und in FOR/NEXT-Variablenzeiger-HB schreiben
,a8d8	20	8a	a3	jsr a38a "srcstk"	Stapelhilfsroutine aufrufen, um GOSUB/RETURN-Eintrag zu suchen
,a8db	9a			txs	richtigen Stapelzeiger auf gefundene Adresse stellen
,a8dc	c9	8d		cmp #8d	Vergleich des Stapeleintrag-Kopfbytes mit GOSUB/RETURN-Flag (GOSUB-Token \$8d)
,a8de	f0	0b		beq a8eb	Übereinstimmung (Z=1): keine Fehlermeldung "RETURN WITHOUT GOSUB" auslösen
,a8e0	a2	0c		ldx #0c	Fehlercode für "RETURN WITHOUT GOSUB" laden
,a8e2	2c	a2	11	→"bit" ldx #11	Fehlercode für UNDEF'D STATEMENT laden, falls Einsprung bei \$a8e3
,a8e5	4c	37	a4	jmp a437 "error"	Fehlermeldung - "RETURN WITHOUT GOSUB" oder "UNDEF'D STATEMENT" - ausgeben

,a8e8	4c	08	af	jmp af08 "synerr"	SYNTAX ERROR auslösen
-------	----	----	----	-------------------	-----------------------

,a8eb	68			→pla	Kopfbyte des Eintrags vom Stapel tilgen	} Stapeleintrag auslesen, wobei das Kopfbyte ohne
,a8ec	68			pla	LB der Nummer der Rücksprunzeile holen	

,a8ed	85 39	sta 39	und in LB des Zeigers auf die aktuelle Zeile schreiben	} Ersatz entfernt wird, während die Adresse und Zeilennummer des für den Rücksprunganzusteuern den Befehls vom Stapel in die entsprechenden Zeiger des Interpreters kommen
,a8ef	68	pla	HB der Nummer der Rücksprungzeile holen	
,a8f0	85 3a	sta 3a	und in HB des Zeigers auf die aktuelle Zeile schreiben	
,a8f2	68	pla	LB des CHRGET-Zeigers für Rücksprungbefehl holen	
,a8f3	85 7a	sta 7a	und in LB des CHRGET-Zeigers schreiben	
,a8f5	68	pla	HB des CHRGET-Zeigers für Rücksprungbefehl holen	
,a8f6	85 7b	sta 7b	und in HB des CHRGET-Zeigers schreiben	

Im Speicher folgt jetzt die Routine zu DATA, die den nächsten Befehl aufruft

; Routine zum Basic-Befehl DATA (Token: \$83)

Überliest alle nach DATA stehenden Angaben und springt somit den hinter DATA stehenden Befehl an;
als IGNORC-Einsprung von \$abe7 (INPUT) und \$b3db (DEF) genutzt.

,a8f8	20 06 a9	jsr a906 "gosnxt"	jetzt Offset zum nächsten Befehl berechnen lassen	} Offset zum nächsten nach
-------	----------	-------------------	---	----------------------------

; ADCGPT-Einsprung: Y-Register zum CHRGET-Zeiger addieren;
Nutzung von \$a93e (IF), \$abf6 (INPUT) und \$acd1 (READ)

,a8fb	98	tya	Ergebnis in Akku zwecks Addition	} GOSUB gestandenen Befehl berechnen, zum CHRGET-Zeiger addieren und diesen dadurch auf Rücksprungbefehl und -zeile anpassen
,a8fc	18	clc	Carry vor Addition löschen	
,a8fd	65 7a	adc 7a	LB des CHRGET-Zeigers zu Offset addieren	
,a8ff	85 7a	sta 7a	und Ergebnis in CHRGET-Zeiger-LB setzen	
,a901	90 02	bcc a905	kein Übertrag (C=0): Ende der Routine	
,a903	e6 7b	inc 7b	Übertrag von \$a8fd hier miteinbeziehen	
,a905	60	>rts	Rücksprung von Routine, CHRGET-Zeiger und Zeiger auf aktuelle Zeile sind jetzt auf die Rücksprungadresse eingestellt	

; GOSNXT-Routine: holt den Offset vom aktuellen CHRGET-Zeiger zur nächsten Zeilenendmarkierung oder Befehlsendmarkierung (je nach Einsprung bei \$a906 oder \$a909) in das Y-Register;
Verwendung bei \$a75a (FOR), \$a8f8 (DATA bzw. ADCPT), \$abf3 (INPUT) und \$acb8 (READ)

,a906	a2 3a	ldx #3a	ASCII-Code für Doppelpunkt laden, wenn auch Doppelpunkt als Endmarkierung gültig ist
-------	-------	---------	--

; GOSEND-Einsprung bei \$a909: Zeilenende suchen; Nutzung von \$a8a3 (GOTO) und \$a93b (IF)

,a908	2c a2 00	"bit" ldx #00	\$00 laden, wenn nur \$00 und nicht auch Doppelpunkt als Endmarkierung gelten soll
,a90b	86 07	stx 07	Suchbyte #1 in \$07 schreiben
,a90d	a0 00	ldy #00	\$00 (Wert für Suchbyte #2) laden; gleichzeitig Offsetinitialisierung für \$a919
,a90f	84 08	sty 08	und in \$08 schreiben

; Suchschleife; vorher enthalten \$07 und \$08 die beiden Suchbytes (entweder beide \$00 oder \$3a und \$00)

,a911	a5 08	→lda 08	Suchbyte #2 in Akku	} Hilfsspeicher \$07 und \$08 für die Suchbytes 1 und 2 vertauschen, um \$3a nur außerhalb von Anführungszeichen zu suchen
,a913	a6 07	ldx 07	Suchbyte #1 in X	
,a915	85 07	sta 07	Suchbyte #2 (s. \$a911) in Suchbyte #1	
,a917	86 08	stx 08	Suchbyte #1 (s. \$a913) in Suchbyte #2	
,a919	b1 7a	→lda (7a),y	Byte an CHRGET-Adresse holen (s. \$a90d!), um es auf \$00 zu testen	
,a91b	f0 e8	beq a905	ja (Z=1): RTS-Befehl anspringen, da Zeilenende erreicht	
,a91d	c5 08	cmp 08	Vergleich des Bytes mit Suchbyte #2	
,a91f	f0 e4	beq a905	Übereinstimmung (Z=1): ebenfalls beendet, Y enthält Fundstellen-Offset	
,a921	c8	iny	Offset auf nächstes Zeichen stellen	
,a922	c9 22	cmp #22	Akku mit \$22 (ASCII-Code des Anführungszeichens) vergleichen, da Doppelpunkte in Anführungszeichen nicht als Endmarkierung festgestellt werden dürfen (!)	
,a924	d0 f3	bne a919	keine Übereinstimmung (Z=0): zurück zur Schleife ohne Sonderbehandlung für QUOTE	
,a926	f0 e9	beq a911 "jmp"	Suchbytes 1 und 2 austauschen, damit \$3a mit \$00 vertauscht wird und somit erst nach erneutem Anführungszeichen wieder nach \$3a-Code gesucht wird	

; Routine zum Basic-Befehl IF (Token: \$8b)

,a928	20 9e ad	jsr ad9e "frmevl"	Auswertung von Parametern aller Art in numerisches Ergebnis
,a92b	20 79 00	jsr 0079 "chrgot"	Zeichen hinter IF-Bedingung holen
,a92e	c9 89	cmp #89	Vergleich mit GOTO-Token (außer THEN-Token als einziges nach IF-Bedingung zulässig)
,a930	f0 05	beq a937	Vergleich positiv (Z=1): Sonderbehandlung für "IF ... GOTO" ohne "THEN"
,a932	a9 a7	lda #a7	Token für THEN als Prüfbyte laden } THEN hinter "IF ..."
,a934	20 ff ae	jsr aeff "chkbyt"	auf Prüfbyte (THEN-Token) testen } verlangen, sonst SYNTAX ERROR
,a937	a5 61	→lda 61	Ergebnis der Auswertung aus FAC-Exponent beziehen
,a939	d0 05	bne a940	numerisches Vergleichsergebnis <> 0 (Z=0): IF-Bedingung wahr, dann eigene Behandlung

; Behandlung: IF-Bedingung nicht erfüllt, also falsch

,a93b	20 09 a9	jsr a909 "gosend"	Offset zur nächsten Endmarkierung holen
,a93e	f0 bb	beq a8fb "jmp adcpt"	wie DATA-Befehl, also alles hinter IF ignorieren und danach fortfahren

; Behandlung: IF-Bedingung wahr

,a940	20 79 00	→jsr 0079 "chrgot"	Zeichen hinter THEN holen
,a943	b0 03	bcs a948	keine Ziffer (C=1): nicht Sonderbehandlung für "IF...THEN 500" usw. ausführen
,a945	4c a0 a8	jmp a8a0 "gotoln"	GOTO-Befehl für Zeilennummer hinter THEN ausführen

; Behandlung: "IF bedingung THEN zeilennummer" bei erfüllter IF-Bedingung

,a948 4c ed a7 > jmp a7ed Fortsetzung in Interpreter-Schleife; jetzt hinter THEN

; Routine zum Basic-Befehl ON (Token: \$91)

<pre> ,a94b 20 9e b7 jsr b79e "getbyt" ,a94e 48 pha ,a94f c9 8d cmp #8d ,a951 f0 04 beq a957 ,a953 c9 89 cmp #89 ,a955 d0 91 bne a8e8 ,a957 c6 65 dec 65 ,a959 d0 04 bne a95f ,a95b 68 pla ,a95c 4c ef a7 jmp a7ef </pre>	<pre> Bytewert (ganzzahlig 0-255) in X-Register aus Basic-Parameter holen; kommt als Nebeneffekt auch nach \$65, was sich die ON-Routine zunutze macht (s. \$a957) und Akku (Zeichen hinter Parameter) retten ist es GOSUB-Token gewesen (nur bei ON x GOSUB)? ja (Z=1): Sonderbehandlung ON x GOSUB anspringen Vergleich mit GOTO-Token (nur bei ON x GOTO) keine Übereinstimmung (Z=0): SYNTAX ERROR auslösen, bei \$a8e8 steht "jmp synerr" Zähler für Suche der richtigen Zeilennummer dekrementieren noch nicht auf 0 heruntergezählt (Z=0): weiter suchen bei \$a94e gerettetes Byte in Akku holen und Interpreter-Schleife anspringen, wodurch GOTO x bzw. GOSUB x (x=richtige Zeilennummer) ausgeführt wird; sehr trickreiche Programmierung </pre>
---	---

<pre> ,a95f 20 73 00 > jsr 0073 "chrget" ,a962 20 6b a9 jsr a96b "linget" ,a965 c9 2c cmp #2c ,a967 f0 ee beq a957 ,a969 68 pla ,a96a 60 rts </pre>	<pre> CHRGET-Zeiger auf nächstes Byte im Basic-Text stellen Zeilennummer einlesen folgt ein Komma auf die Zeilennummer (ASCII-Code \$2c)? ja (Z=1): weiter in Suchschleife auf richtige Zeilennummer nein, dann Ende der Suche; Akku wird vom Stapel zurückgeholt, so daß jetzt die erste angegebene Sprungzeile angesprungen wird Rücksprung von Routine, führt letztlich zur Ausführung von "GOTO zeilennummer1" aus "ON x GOTO zeilennummer1,..." </pre>
--	---

; LINGET-Routine: holt Zeilennummer < 64000 ab CHRGET-Zeigerposition aus Basic-Text nach \$14/\$15 im Integerformat

Vorher muß CHRGET aufgerufen worden sein, damit die CPU-Flags und der Akku entsprechend belegt sind;

Aufruf von \$a49c (MAIN-Routine), \$a6a4 und \$a6b6 (beides aus LIST), \$a8a0 (GOTO) und \$a962 (ON)

<pre> ,a96b a2 00 ldx #00 ,a96d 86 14 stx 14 ,a96f 86 15 stx 15 ,a971 b0 f7 bcs a96a ,a973 e9 2f sbc #2f ,a975 85 07 sta 07 </pre>	<pre> Initialisierungswert für \$14/\$15 holen; X dient nicht als Offset \$14 (LB der Zeilennummer) mit 0 belegen \$15 (HB der Zeilennummer) mit 0 belegen Zeichen aus Basic-Text ist keine Ziffer (C=1): RTS anspringen, \$14/\$15 wird bei Eingabe keines Zeichens als Vorbelegung übernommen ASCII-Code von 0 (\$30) minus 1 subtrahieren; zusätzlich wird wegen gelöschten Carry-Flags (s. \$a971) noch einmal 1 abgezogen, so daß Subtraktion von \$30 erfolgt Ergebnis (Ziffer \$00-\$09 statt ASCII-Code) in \$07 (Hilfsspeicher) merken </pre>	<pre> } \$14/\$15 } mit \$0000 } vorbelegen </pre>
--	--	--

,a977	a5 15	lda 15	HB der derzeitigen Zeilennummer holen		
,a979	85 22	sta 22	und in Hilfsspeicher \$22 schreiben		
,a97b	c9 19	cmp #19	Vergleich mit #25 (nur bei Zeilennummer >= 64000 ist \$15-Inhalt >= #25)		
,a97d	b0 d4	↖ bcs a953	größer, also unerlaubte Zeilennummer (C=1): SYNTAX ERROR aufrufen (funktioniert nur "fast immer"; näheres in Kapitel 4)		
,a97f	a5 14	lda 14	LB der Zeilennummer holen	\$14/\$22 (LB/HB	letzten Wert der
,a981	0a	asl	mit 2 multiplizieren	der Zeilennummer	Zeilennummer
,a982	26 22	rol 22	HB-Hilfsspeicher mit 2 multiplizieren	mit	*4, dann
,a984	0a	asl	LB der Zeilennummer mit 2 multiplizieren	4	+Zeilennummer,
,a985	26 22	rol 22	HB-Hilfsspeicher mit 2 multiplizieren	multiplizieren	schließlich
,a987	65 14	adc 14	LB zu sich selbst addieren, also verdoppeln	zu Ergebnis	noch einmal
,a989	85 14	sta 14	und Ergebnis in Zeilennummer-LB	den doppelten	*2, also
,a98b	a5 22	lda 22	Hilfsspeicher \$22 holen (Ergebnis von \$a985)	Wert der alten	effektiv *10;
,a98d	65 15	adc 15	altes (s. \$a977) HB der Zeilennummer addieren	Zeilennummer	dadurch werden
,a98f	85 15	sta 15	und setzen, also effektiv HB*2 addieren	addieren	vorausgehende
,a991	06 14	asl 14	LB der Zeilennummer mit 2 multiplizieren	dieses Ergebnis	Ziffern um eine
,a993	26 15	rol 15	HB der Zeilennummer mit 2 multiplizieren	auch verdoppeln	Stelle links
,a995	a5 14	lda 14	LB der Zeilennummer in Akku holen	jetzt die neue	versetzt, damit
,a997	65 07	adc 07	und neue Ziffer (s. \$a973, \$a975) addieren	Stelle zur	neue Ziffer an
,a999	85 14	sta 14	Ergebnis in LB der Zeilennummer schreiben	Zeilennummer	unterste Stelle
,a99b	90 02	bcc a99f	kein Übertrag (C=0): HB nicht inkrementieren	als letzte Stelle	(10↑0-Stelle)
,a99d	e6 15	inc 15	HB der Zeilennummer wegen Übertrag erhöhen	addieren	addiert wird.
,a99f	20 73 00	jsr 0073	"chrget"	nächstes Zeichen aus Basic-Text holen, Flags setzen	
,a9a2	4c 71 a9	jmp a971		Rücksprung an Schleifenanfang	

; Routine zum Basic-Befehl LET (Token: \$88)

Wird bei \$a746 von der FOR-, bei \$a804 von der GONE-Routine aufgerufen

,a9a5	20 8b b0	jsr b08b	"fndvar"	Adresse der LET-Variablen ermitteln; falls nicht vorhanden: neu anlegen	Adresse der Zuweisungsvariablen suchen (falls nicht vorhanden: neuen Eintrag
,a9a8	85 49	sta 49		LB der Adresse in LB von Zeiger \$49/\$4a	im Variablenspeicher errichten) und
,a9aa	84 4a	sty 4		HB der Adresse in HB von Zeiger \$49/\$4a	in \$49/\$4a (Hilfszeiger) merken
,a9ac	a9 b2	lda #b2		Token für "=" (Zuweisungszeichen) laden	SYNTAX ERROR auslösen, wenn hinter
,a9ae	20 ff ae	jsr ae ff	"chkbyt"	und als Prüfbyte verwenden	"LET variable" nicht "=" folgt
,a9b1	a5 0e	lda 0e		Integerflag für numerische Variablen holen	Datentyp-Flags für Variablen
,a9b3	48	pha		und auf den Stapel retten	(Integer/Fließkomma; String/numerisch)
,a9b4	a5 0d	lda 0d		String/Numerisch-Flag für Variablen holen	aus Zeropage-Hilfsspeichern auslesen
,a9b6	48	pha		und auf den Stapel retten	und auf den Stapel retten
,a9b7	20 9e ad	jsr ad9e	"frmevl"	Ausdruck hinter Zuweisungszeichen auswerten (Zuweisungswert holen)	
,a9ba	68	pla		bei \$a9b4/\$a9b6 gerettetes String/Numerisch-Flag zurückholen	


```

,a9bb 2a      rol      b7 durch Bitverschiebung zwecks Test in Carry holen
,a9bc 20 90 ad  jsr ad90 "chktyp" Test auf Variablentyp (C=0: numerisch; C=1: String, s. $a9bb), ggf. TYPE MISMATCH
,a9bf d0 18      bne a9d9 String (Z=0): Variablenzuweisungsroutine für String anspringen

; Zuweisungsroutine für numerische Variable (Integer/Fließkomma)

,a9c1 68      pla      Integerflag (bei $a9b1/$a9b3 gerettet) vom Stapel holen

; hier: Aufruf von $ac82 (INPUT/READ/GET-Routine)

,a9c2 10 12      bpl a9d6 keine Integerzahl (N=0): Zuweisungsroutine für Fließkommavariablen anspringen

; Zuweisungsroutine für Integervariablen

,a9c4 20 1b bc  jsr bclb "round" FAC zunächst runden (Rundungsbyte berücksichtigen)
,a9c7 20 bf bl  jsr blbf      und in Integerformat (mit Vorzeichen!) nach $64/$65 holen
,a9ca a0 00      ldy #00      Offset mit 0 initialisieren
,a9cc a5 64      lda 64      LB von $a9c7 holen
,a9ce 91 49      sta (49),y    und in LB des Variableneintrags schreiben
,a9d0 c8      iny "ldy #01"    Offset von 0 auf 1 erhöhen (auf HB stellen)
,a9d1 a5 65      lda 65      HB von $a9c7 holen
,a9d3 91 49      sta (49),y    und in HB des Variableneintrags schreiben
,a9d5 60      rts      Routine beenden, Variable steht im Speicher

-----

; Zuweisungsroutine für Fließkomma-Variablen anspringen

,a9d6 4c d0 bb  jmp bbd0 "facvar" FAC-Inhalt (Fließkomma-Zuweisungswert) in Variable transportieren

-----

; Zuweisungsroutine für Stringvariablen

,a9d9 68      pla      Datentyp-Flag Integer/Fließkomma vom Stapel löschen, da bei Strings bedeutungslos

; hier: Aufruf von $ac83 (INPUT/READ/GET-Routine)

,a9da a4 4a      ldy 4a      HB der Variablenadresse holen
,a9dc c0 bf      cpy #bf      mit $bf (HB von FAC-Inhalt bei TI$-Zuweisung, hat Flag-Charakter) vergleichen
,a9de d0 4c      bne aa2c     nicht TI$ (Z=0): Wertzuweisung an herkömmliche Stringvariable aufrufen

```

; Wertzuweisung an TI\$

,a9e0	20 a6 b6	jsr b6a6 "frestr"	in FRESTR-Routine einsteigen, damit String auf temporären Stringstapel kommt
,a9e3	c9 06	cmp #06	Stringlänge (nach "\$a9e0 jsr \$b6a6" im Akku) mit 6 (TI\$-Länge) vergleichen
,a9e5	d0 3d	bne aa24	nicht exakt geforderte Stringlänge 6 (Z=0): ILLEGAL QUANTITY ERROR auslösen
,a9e7	a0 00	ldy #00	Initialisierungswert für Exponent und Vorzeichen von FAC #1 sowie Hilfszähler laden
,a9e9	84 61	sty 61	Exponent von FAC #1 initialisieren
,a9eb	84 66	sty 66	Vorzeichen von FAC #1 initialisieren
,a9ed	84 71	sty 71	Hilfszähler \$71 (Zeiger auf Stelle in TI\$) setzen
,a9ef	20 ld aa	jsr aald "strcgt"	nächstes Zeichen im String auf Ziffer prüfen (wenn nicht: ILLEGAL QUANTITY; sonst: von ASCII-Code in numerisches Format \$00-\$09 umwandeln und zum FAC addieren)
,a9f2	20 e2 ba	jsr bae2 "facml0"	FAC-Inhalt mit 10 multiplizieren, um vorhergehende Ziffern nach links zu bewegen
,a9f5	e6 71	inc 71	Hilfszähler \$71 erhöhen, also auf nächste Stelle in TI\$ positionieren
,a9f7	a4 71	ldy 71	Y-Register (dient als Offset-Register) mit Hilfszähler laden
,a9f9	20 ld aa	jsr aald "strcgt"	nächstes Zeichen im String auf Ziffer prüfen (wenn nicht: ILLEGAL QUANTITY; sonst: von ASCII-Code in numerisches Format \$00-\$09 umwandeln und zum FAC addieren)
,a9fc	20 0c bc	jsr bc0c "movfa"	FAC #1 in ARG (FAC #2) übertragen, damit er als Argument dienen kann
,a9ff	aa	tax	Akku (Flag, ob FAC #1 = 0 ist) in X-Register schreiben
,aa00	f0 05	beq aa07	FAC = 0 (Z=1): Addition überspringen
,aa02	e8	inx	X-Inhalt erhöhen (Exponent um 1 erhöhen)
,aa03	8a	txa	und Ergebnis in Akku
,aa04	20 ed ba	jsr baed	in FACML0-Routine so einsteigen, daß Exponent in Akku verwendet wird (anstelle der 10er-Multiplikation)
,aa07	a4 71	ldy 71	Hilfszähler \$71 holen
,aa09	c8	iny	um 1 erhöhen
,aa0a	c0 06	cpy #06	mit 6 (erster zu hoher Wert) vergleichen
,aa0c	d0 df	bne a9ed	keine Übereinstimmung, noch nicht fertig (Z=0): Schleife mit neuem Hilfszeiger
,aa0e	20 e2 ba	jsr bae2 "facml0"	FAC mit 10 multiplizieren, was aufgrund vorzeitigen Schleifenabbruchs bei \$a9f2 nicht mehr erledigt wird
,aal1	20 9b bc	jsr bc9b "facint"	FAC in 3-Byte-Integerformat nach \$63-\$65 umwandeln; nur positive Zahlen
,aal4	a6 64	ldx 64	mittelwertiges Byte nach X
,aal6	a4 63	ldy 63	niederwertigstes Byte nach Y
,aal8	a5 65	lda 65	höherwertigstes Byte nach A
,aala	4c db ff	jmp ffdb "settim"	interne Uhr (TI/TI\$) über Kernel-Routine setzen

} ermittelte
3 Byte
als Uhrzeit
einstellen

; STRCGT-Unterroutine zur Auswertung des nächsten Zeichens eines Strings (Adresse in \$22, Offset in Y); Zeichen wird auf numerisch geprüft, von ASCII-Code in numerischen Wert \$00-\$09 umgewandelt und zum FAC addiert;
Aufruf von \$a9ef und \$a9f9 (jeweils bei "LET TI\$=")

,aal	bl 22	lda (22),y	Byte anhand von Zeiger \$22/\$23 und Offset in Y auslesen
,aalf	20 80 00	jsr 0080	so in CHRGET/CHRGOT-Routine einsteigen, daß nur ein Test des Akku und
			entsprechendes Setzen des Carry-Flags erfolgt (C=0: Ziffer; C=1: keine Ziffer)
,aa22	90 03	bcc aa27	Ziffer (C=0): keine Fehlermeldung, sondern ASCII-Code der Ziffer auswerten
,aa24	4c 48 b2	jmp b248 "illqua"	ILLEGAL QUANTITY anspringen

,aa27	e9 2f	>sbcs #2f	da C=0 ist (s. \$aa22!) und bei gelöschtem Carry zusätzlich zum Operanden auch 1
			subtrahiert wird, wird hier vom Akku \$2f+1 = \$30 (ASCII-Code von "0") abgezogen
,aa29	4c 7e bd	jmp bd7e "addafc"	im Akku stehende Ziffer (\$00-\$09) zu FAC addieren

; Zuweisungsroutine für herkömmliche Stringvariable (alle außer TI\$)

,aa2c	a0 02	ldy #02	Offset mit 2 initialisieren (auf HB der Stringadresse stellen)
,aa2e	bl 64	lda (64),y	HB der Stringadresse auslesen
,aa30	c5 34	cmp 34	und mit HB des Zeigers auf Stringgrenze +1 vergleichen
,aa32	90 17	bcc aa4b	HB der Stringadresse < HB des String-Grenz-Zeigers (C=0): String in Speicher kopieren
,aa34	d0 07	bne aa3d	HB der Stringadresse <> HB des String-Grenz-Zeigers (Z=0): Vergleich der HBs
,aa36	88	dey "ldy #01"	Offset dekrementieren (von 2 auf 1, also auf LB der Stringadresse stellen)
,aa37	bl 64	lda (64),y	LB der Stringadresse auslesen
,aa39	c5 33	cmp 33	und mit LB des Zeigers auf Stringgrenze +1 vergleichen
,aa3b	90 0e	bcc aa4b	LB der Stringadresse < LB des String-Grenz-Zeigers (C=0): String in Speicher kopieren
,aa3d	a4 65	>ldy 65	HB der Adresse des String-Variableneintrags holen
,aa3f	c4 2e	cpy 2e	und mit HB des Anfangs des Variablenspeichers (= Programmende) vergleichen
,aa41	90 08	bcc aa4b	HB der Stringeintragsadresse < HB des Variablenanfangs (C=0): String in Speicher
,aa43	d0 0d	bne aa52	HB der Stringeintragsadresse = HB des Variablenanfangs (Z=0): Sonderbehandlung
,aa45	a5 64	lda 64	LB der Adresse des String-Variableneintrags holen
,aa47	c5 2d	cmp 2d	und mit LB des Anfangs des Variablenspeichers (= Programmende) vergleichen
,aa49	b0 07	bcs aa52	LB der Stringeintragsadresse >= LB des Variablenanfangs (C=0): Sonderbehandlung
,aa4b	a5 64	>lda 64	LB der Stringeintragsadresse in Akku holen } Adresse des Stringeintrags im
,aa4d	a4 65	ldy 65	HB der Stringeintragsadresse in Y holen } Variablenspeicher in A/Y setzen
,aa4f	4c 68 aa	jmp aa68	String in Speicher kopieren (A/Y enthalten Adresse des Stringvariableneintrags)

; Sonderbehandlung: Wertzuweisung an Stringvariable, für die Speicherplatz zu schaffen ist

,aa52	a0 00	>ldy #00	Offset mit 0 initialisieren (auf Stringlänge stellen)
,aa54	bl 64	lda (64),y	Stringlänge dem Stringvariableneintrag entnehmen
,aa56	20 75 b4	jsr b475	Hilfsroutine zur Schaffung von Platz durch Ändern der Stringzeiger aufrufen

,aa56	20 75 b4	jsr b475	Hilfsroutine zur Schaffung von Platz durch Ändern der Stringzeiger aufrufen	
,aa59	a5 50	lda 50	LB der Adresse für neuen String holen	} Zieladresse
,aa5b	a4 51	ldy 51	HB der Adresse für neuen String holen	
,aa5d	85 6f	sta 6f	LB in LB des Hilfszeigers auf Zieladresse schreiben	} Strings
,aa5f	84 70	sty 70	HB in HB des Hilfszeigers auf Zieladresse schreiben	
,aa61	20 7a b6	jsr b67a	String in dafür geschaffenen Speicherplatz kopieren	
,aa64	a9 61	lda #61 <(\$0061)	LB von \$0061 (Adresse des Deskriptors) laden	} Adresse des Deskriptors
,aa66	a0 00	ldy #00 >(\$0061)	HB von \$0061 (Adresse des Deskriptors) laden	
				(\$0061) von vorher laden

; ab hier: gemeinsam für alle herkömmlichen Strings, unabhängig davon, ob vorher Platz im Stringspeicher organisiert werden mußte oder nicht: Löschen des Stringdeskriptors im temporären Stringstapel und Umkopieren des Stringdeskriptors; auch Aufruf von \$aa4f (Zuweisungsroutine für herkömmliche Stringvariable)

,aa68	85 50	sta 50	LB in LB von Übergabe-Zeiger \$50/\$51 schreiben	
,aa6a	84 51	sty 51	HB in HB von Übergabe-Zeiger \$50/\$51 schreiben	
,aa6c	20 db b6	jsr b6db	Stringdeskriptor im temporären Stringstapel löschen	
,aa6f	a0 00	ldy #00	Kopierzähler initialisieren	} Stringdeskriptor
,aa71	b1 50	lda (50),y	erstes Byte (Stringlänge) aus altem Deskriptor holen	
,aa73	91 49	sta (49),y	und in erstes Byte des neuen Deskriptors schreiben	} aus
,aa75	c8	iny "ldy #01"	Kopierzähler von 0 auf 1 (LB der Stringadresse) stellen	
,aa76	b1 50	lda (50),y	zweites Byte (LB der Stringadresse) aus altem Deskriptor holen	} Bereich auslesen
,aa78	91 49	sta (49),y	und in zweites Byte des neuen Deskriptors schreiben	
,aa7a	c8	iny "ldy #02"	Kopierzähler von 1 auf 2 (HB der Stringadresse) stellen	} und in neuen
,aa7b	b1 50	lda (50),y	drittes Byte (HB der Stringadresse) aus altem Deskriptor holen	
,aa7d	91 49	sta (49),y	und in drittes Byte des neuen Deskriptors schreiben	} Descriptor
,aa7f	60	rts	Routine beenden, da Stringdeskriptor umkopiert wurde	
				(Stringlänge und
				Stringadresse)

; Routine zum Basic-Befehl PRINT# (Token: \$98)

,aa80	20 86 aa	jsr aa86	Routine für Basic-Befehl CMD aufrufen
,aa83	4c b5 ab	jmp abb5	Basic-Einsprung für CLRCH anspringen (CMD von \$aa80 rückgängig machen)

; Routine zum Basic-Befehl CMD (Token: \$9d)

,aa86	20 9e b7	jsr b79e "getbyt"	Filenummer hinter CMD in X-Register einlesen
,aa89	f0 05	beq aa90	auf Byte folgt Endmarkierung (Doppelpunkt oder \$00) (Z=1): keine weiteren Parameter
,aa8b	a9 2c	lda #2c	Komma-Code als Prüfbyte laden
,aa8d	20 ff ae	jsr ae ff "chkbyt"	umständliche Schreibweise
,aa90	08	>php	und als Syntax-Bedingung erfordern } von "jsr chkcom"
			Prozessorstatus (Z-Flag!) retten


```

,aa91 86 13    stx 13      bei $aa86 geholte Filenummer in Flag für aktuelles I/O-Gerät schreiben
,aa93 20 18 e1  jsr e1l8    und als Ausgabegerät über CKOUT-Einsprung des Interpreters setzen
,aa96 28       plp         Prozessorstatus von $aa90 wieder holen
,aa97 4c a0 aa  jmp aaa0    zum PRINT-Befehl für Ausgabe von [CR,LF] und ggf. Text hinter "CMD filenummer,"
-----

```

; Anfang der PRINT-Befehlsroutine, allerdings nicht Einsprungsadresse (diese liegt bei \$aaa0)

```

,aa9a 20 21 ab → jsr ab21 "prtstr" String über Einsprung in STROUT-Routine ausgeben
,aa9d 20 79 00 → jsr 0079 "chrgot" letztes Zeichen hinter String bearbeiten

```

; Einsprung der Routine zum Basic-Befehl PRINT (Token: \$99)

```

,aaa0 f0 35    ↓ beq aad7      bei Einsprung über Interpreterschleife oder nach Ausführung von $aa9d: Endmarkierung
                                der PRINT-Anweisung (Z=1): [CR,LF]-Ausgabe aufrufen
,aaa2 f0 43    ↓ beq aae7      bei Schleifenaufruf bei $abl6: Endmarkierung der PRINT-Anweisung (Z=1): Sprung zu RTS
,aaa4 c9 a3    cmp #a3        Vergleich des auszuführenden Zeichens mit TAB-Token
,aaa6 f0 50    ↓ beq aaf8      Übereinstimmung (Z=1): TAB ausführen, Carry muß logischerweise gesetzt (!) sein
,aaa8 c9 a6    cmp #a6        Vergleich des auszuführenden Zeichens mit SPC-Token
,aaaa 18       clc            Carry löschen (Flag für SPC)
,aaab f0 4b    ↓ beq aaf8      Übereinstimmung (Z=1): SPC ausführen
,aaad c9 2c    cmp #2c        Vergleich des auszuführenden Zeichens mit Komma-Code (Pseudo-Tabulator)
,aaaf f0 37    ↓ beq aae8      Übereinstimmung (Z=1): Komma berücksichtigen (Pseudo-Tabulator)
,aabl c9 3b    cmp #3b        Vergleich des auszuführenden Zeichens mit Semikolon-Code
,aab3 f0 5e    ↓ beq abl3      Übereinstimmung (Z=1): Semikolon berücksichtigen
,aab5 20 9e ad  jsr ad9e "frmevl" Ausdruck für Ausgabe holen (String oder numerisch)
,aab8 24 0d    bit 0d         Datentyp-Flag für String/numerisch testen
,aaba 30 de    bmi aa9a       String (N=1): String ausgeben und weiter in Schleife
,aabc 20 dd bd  jsr bddd "flpstr" FAC-Inhalt (numerischer Ausgabeparameter) in ASCII-Code-String umwandeln
,aabf 20 87 b4  jsr b487 "strlit" Stringparameter (Länge und Adresse im Speicher) ermitteln; Adresse nach $22/$23
,aac2 20 21 ab  jsr ab21 "prtstr" in STROUT-Routine so einsteigen, daß Ergebnis ausgegeben wird
,aac5 20 3b ab  jsr ab3b "rgtspc" Unterroutine zur Ausgabe von [SPACE/CRSR RIGHT] aufrufen
,aac8 d0 d3    bne aa9d "jmp" zurück in PRINT-Schleife, nächsten Parameter auswerten

```

; Unterroutine für Eingabeschleife einer Basic-Zeile: \$00 an letzte Pufferposition (Offset in X, Systemeingabepuffer ab \$0200) schreiben, um Endmarkierung am Pufferende zu schaffen;
Verwendung von \$a576 (MAIN) aus

```

,aaca a9 00    lda #00        $00 (Endmarkierung für Systemeingabepuffer) laden $00
,aacc 9d 00 02 sta 0200,x     und an $0200+x, also hinter letztes Byte im Puffer, schreiben

```

} an Pufferende
schreiben

,aacf	a2 ff	ldx #ff <(\$01ff)	LB von \$01ff (Anfangsadresse des Systemeingabepuffers -1) laden	\$01ff nach X/Y laden
,aad1	a0 01	ldy #01 >(\$01ff)	HB von \$01ff (Anfangsadresse des Systemeingabepuffers -1) laden	
,aad3	a5 13	lda 13	Flag für Basic-Ausgabe laden	
,aad5	d0 10	bne aae7	gesetzt, Ausgabe wurde umgelenkt (Z=0): RTS anspringen	
,aad7	a9 0d	lda #0d	ASCII-Code für [CR] laden	
,aad9	20 47	ab jsr ab47 "bbsout"	Zeichen über Basic-Einsprung für BASOUT ausgeben	
,aadc	24 13	bit 13	Flag für Basic-Ausgabe testen	
,aade	10 05	bpl aae5	gelöscht, Filenummer hat b7=0 (N=0): kein [LF] ausgeben	
,aae0	a9 0a	lda #0a	[LF]-Code laden	
,aae2	20 47	ab jsr ab47 "bbsout"	und über Basic-Einsprung für BASOUT ausgeben	
,aae5	49 ff	eor #ff %11111111	Byte im Akku invertieren, um Flags entsprechend zu setzen (z.B. Z=0, N=1)	
,aae7	60	rts	Rücksprung von Routine	

; PRINT-Befehl: Unterroutine zur Ausführung des Kommas (Pseudo-Tabulator)

,aae8	38	sec	Carry setzen, Flag für "Cursor-Position auslesen"
,aae9	20 f0 ff	jsr fff0 "plot"	Cursor-Position holen
,aaec	98	tya	Spalte (Y) in Akku zwecks Weiterverarbeitung
,aaed	38	sec	Carry vor Subtraktion setzen
,aaee	e9 0a	sbc #0a	10 subtrahieren, um Restbetrag der Division durch 10 zu erhalten
,aaf0	b0 fc	bcs aaaa	kein Übertrag bei Subtraktion (C=1): weiter subtrahieren
,aaf2	49 ff	eor #ff %11111111	Restbetrag invertieren, um negatives Ergebnis in positive Spaltenzahl umzuwandeln
,aaf4	69 01	adc #01	invertierten Restbetrag um 1 erhöhen (Ergebniskorrektur)
,aaf6	d0 16	bne ab0e "jmp"	in Ausgabeschleife einsteigen

; Ausführung von TAB bei gesetztem und SPC bei gelöschtem Carry-Flag

,aaf8	08	php	Prozessorstatus auf Stapel retten (Carry-Flag !)
,aaf9	38	sec	Carry setzen (Flag für Cursorposition holen)
,aafa	20 f0 ff	jsr fff0 "plot"	Cursorposition auslesen
,aafd	84 09	sty 09	Spaltenposition als Spalte vor letztem TAB- oder SPC-Befehl setzen
,aaff	20 9b b7	jsr b79b	Byte hinter TAB durch indirekten GETBYT-Aufruf ermitteln
,ab02	c9 29	cmp #29	folgt "Klammer zu" (ASCII-Code \$29) auf Zahl nach TAB?
,ab04	d0 59	bne ab5f	nein (Z=0): SYNTAX ERROR auslösen
,ab06	28	plp	Prozessorstatus von \$aaf8 vom Stapel holen (Carry-Flag !)
,ab07	90 06	bcc ab0f	SPC-Befehl (C=0): zur SPC-Routine verzweigen, in X steht SPC-Parameter (seit \$aaff)

; Sonderbehandlung TAB-Befehl

,ab09	8a	txa	TAB-Parameter in Akku holen
,ab0a	e5 09	sbc 09	davon die letzte Spaltenposition für TAB/SPC abziehen (C=1 dank \$ab07!)
,ab0c	90 05	bcc abl3	Übertrag bei Subtraktion, also TAB-Parameter kleiner als Cursorposition vor TAB (C=0): weiter in PRINT-Schleife
,ab0e	aa	→tax	Ergebnis ist Anzahl der zu überspringenden Spalten und kommt nach X

; Sonderbehandlung SPC-Befehl oder Fortsetzung der Routine zu TAB

,ab0f	e8	→inx	Ergebnis um 1 erhöhen, was in Schleife rückgängig gemacht wird
,ab10	ca	→dex	Zähler für Anzahl der zu überspringenden Spalten verringern
,ab11	d0 06	bne abl9	Zähler <> 0 (Z=0): in Ausgabeschleife springen

; Ausführung von TAB beenden

,ab13	20 73 00	jsr 0073 "chrget"	nächstes Zeichen für Ausführung von PRINT vorbereiten
,ab16	4c a2 aa	jmp aaa2	wieder in PRINT-Schleife springen

; Ausgabe von [CRSR RIGHT] oder [SPACE] in TAB/SPC-Ausgabeschleife

,ab19	20 3b ab	→jsr ab3b "rgtspc"	Ausgaberroutine für [CRSR RIGHT] oder [SPACE] aufrufen
,ab1c	d0 f2	bne abl0 "jmp"	weiter in TAB/SPC-Ausgabeschleife

; STROUT-Routine: Ausgabe eines Strings ab der in A/Y enthaltenen Adresse;

Verwendung von \$a469 (ERROR), \$a478 (ERROR), \$ab6f (INPUT), \$acf8 (READ/INPUT), \$bdda (NUMOUT), \$e191 (LOAD/VERIFY), \$elaf (LOAD), \$e42d (MSGNEW) und \$e441 (MSGNEW)

,able 20 87 b4 jsr b487 "strlit" Stringparameter holen

; PRTSTR-Einsprung: Ausgabe eines eingeholten Strings

Verwendung von \$aa9a (PRINT-Befehlsroutine), \$aac2 (PRINT) und \$abcb (INPUT)

,ab21	20 a6 b6	jsr b6a6 "frestr"	und soweit auswerten, daß jetzt die Ausgabe erfolgen kann
,ab24	aa	tax	Stringlänge in X-Register
,ab25	a0 00	ldy #00	Ausgabe-Zähler mit \$00 initialisieren
,ab27	e8	inx	Stringlänge erhöhen, da sie gleich wieder am Schleifenbeginn dekrementiert wird
,ab28	ca	→dex	Zähler für Stringlänge dekrementieren
,ab29	f0 bc	↖beq aae7	schon auf 0 = alle Zeichen ausgegeben (Z=1): RTS anspringen
,ab2b	bl 22	lda (22),y	Zeichen aus String holen

```

,ab2d 20 47 ab jsr ab47 "bbsout" Zeichen über Basic-Einsprung für BSOUT ausgeben
,ab30 c8      iny      Ausgabe-Zähler inkrementieren
,ab31 c9 0d      cmp #0d  war auszugebendes Zeichen [CR]?
,ab33 d0 f3      bne ab28  nein (Z=0): zurück in Schleife springen
,ab35 20 e5 aa jsr aae5  /// eigentlich müßte es "jsr $aadc" heißen, offensichtlich ist den Programmierern
                        hier ein Fehler unterlaufen; dieser JSR hat keine Auswirkungen, er invertiert nur
                        den Akku-Inhalt, der danach ohnehin nicht mehr berücksichtigt wird
,ab38 4c 28 ab jmp ab28  in Ausgabeschleife zurückspringen
-----

```

; Ausgaberroutine für Basic; enthält Basic-BSOUT-Einsprung bei \$ab47
wird hier eingesprungen, erfolgt die Ausgabe von [SPACE] bei auf File umgelenkter Ausgabe, bei Standard-Ausgabe auf
Bildschirm wird [CRSR RIGHT] ausgegeben; Nutzung von \$aac5, \$abl9 (PRINT) und \$ac00 (INPUT)
bei Einsprung zu \$ab3f: Ausgabe von [SPACE]
bei Einsprung zu \$ab42: Ausgabe von [CRSR RIGHT]
bei Einsprung zu \$ab45: Ausgabe von Fragezeichen

```

,ab3b a5 13      lda 13      Flag für Basic-I/O auslesen
,ab3d f0 03      beq ab42     gelöscht, also Standard-I/O auf Bildschirm (Z=1): [CRSR RIGHT] ausgeben

```

; Einsprung für Ausgabe von [SPACE]

```

,ab3f a9 20      lda #20      ASCII-Code von [SPACE] laden, wird bei $ab47 ausgegeben

```

; Einsprung für Ausgabe von [CRSR RIGHT] (bei \$ab41+1 = \$ab42)

```

,ab41 2c a9 1d >"bit" lda #1d  ASCII-Code von [CRSR RIGHT] laden, wird bei $ab47 ausgegeben

```

; QUMOUT: Einsprung für Ausgabe von Fragezeichen (bei \$ab44+1 = \$ab45); wird von \$a451, \$abfd und \$ac47 genutzt

```

,ab44 2c a9 3f "bit" lda #3f  ASCII-Code von Fragezeichen laden, wird bei $ab47 ausgegeben

```

; BBSOUT: Basic-Einsprung für BASOUT (ruft weiteren Einsprung bei \$el0c auf, der nur hier vom Interpreter benutzt wird);
Verwendung von \$a45b, \$a6f3, \$a73d, \$aad9, \$aae2, \$ab2d

```

,ab47 20 0c e1 jsr el0c      Basic-BSOUT-Einsprung aufrufen
,ab4a 29 ff      and #ff %11111111 verändert Akku nicht, setzt aber CPU-Flags gemäß Akkuinhalt
,ab4c 60          rts          Rückkehr von Routine, Flags sind entsprechend gesetzt
-----

```


; Fehlerbehandlung bei INPUT, GET oder READ;
wird von \$ac9a aus angesprungen

```
,ab4d a5 11 lda 11      Flag für Eingabebefehl ($00=INPUT, $40=GET, $80=READ)
,ab4f f0 11 beq ab62     INPUT (Z=1): Sonderbehandlung anspringen
,ab51 30 04 bmi ab57     READ (N=1): Sonderbehandlung anspringen
```

; Fehlersonderbehandlung für GET

```
,ab53 a0 ff ldy #ff %11111111 $ff als HB für Zeiger auf aktuelle Zeilennummer laden
,ab55 d0 04 bne ab5b "jmp" und in READ-Sonderbehandlung einsteigen
```

; Fehlersonderbehandlung für READ

```
,ab57 a5 3f lda 3f      LB der aktuellen DATA-Zeilenummer holen } Nummer der aktuellen DATA-Zeile
,ab59 a4 40 ldy 40      HB der aktuellen DATA-Zeilenummer holen } in Zeiger auf
,ab5b 85 39 sta 39      LB der aktuellen Zeilennummer setzen      } aktuelle
,ab5d 84 3a sty 3a      HB der aktuellen Zeilennummer setzen      } Zeilennummer schreiben
,ab5f 4c 08 af jmp af08 "synerr" SYNTAX ERROR melden
```

; Fehlersonderbehandlung für INPUT

```
,ab62 a5 13 lda 13      Flag für Datentyp bei Eingabe auslesen
,ab64 f0 05 beq ab6b     numerische Variable (Z=1): REDO FROM START auslösen und Eingabe wiederholen
,ab66 a2 18 ldx #18      Fehlercode für FILE DATA laden } FILE DATA - Fehlermeldung
,ab68 4c 37 a4 jmp a437 "error" Fehlereinsprung aufrufen } wegen fehlerhafter INPUT-Stringeingabe
```

```
,ab6b a9 0c lda #0c <($ad0c) LB von $ad0c (Adresse für Text "REDO FROM START") laden } Ausgabe der
,ab6d a0 ad ldy #ad >($ad0c) HB von $ad0c (Adresse für Text "REDO FROM START") laden } Steuermeldung
,ab6f 20 1e ab jsr able "strout" Text REDO FROM START ausgeben } "REDO FROM START"
,ab72 a5 3d lda 3d      LB des CONT-Zeigers holen } CHRGET-Zeiger wieder auf den
,ab74 a4 3e ldy 3e      HB des CONT-Zeigers holen } fehlerverursachenden INPUT-Befehl
,ab76 85 7a sta 7a      LB des CHRGET-Zeigers setzen } stellen, dessen Adresse dem
,ab78 84 7b sty 7b      HB des CHRGET-Zeigers setzen } CONT-Zeiger entnommen wird
,ab7a 60 rts           Rücksprung von Routine, jetzt erfolgt Fortsetzung durch nochmalige Eingabe
```

; Routine zum Basic-Befehl GET (Token: \$a1); arbeitet auch den GET#-Befehl, der kein eigenes Token hat, ab

```
,ab7b 20 a6 b3 jsr b3a6 "chkdir" gibt Meldung ILLEGAL DIRECT ERROR aus, wenn GET im Direktmodus erfolgen soll
,ab7e c9 23 cmp #23      folgt auf GET-Befehl das Zeichen "#" (Doppelkreuz)?
```

,ab80	d0 10	bne ab92	nein (Z=0): GET nicht von File, sondern von Tastatur durchführen
,ab82	20 73 00	jsr 0073 "chrget"	CHRGET-Zeiger auf erstes Byte hinter Doppelkreuz stellen
,ab85	20 9e b7	jsr b79e "getbyt"	Bytewert (\$00-\$ff) hinter GET# einlesen (in X-Register)
,ab88	a9 2c	lda #2c	Komma-Code als Prüfbyte laden
,ab8a	20 ff ae	jsr aeff "chkbyt"	Komma als syntaktisches Erfordernis testen
,ab8d	86 13	stx 13	Eingabefile setzen; gleichzeitig Flag, daß GET# abgearbeitet wird, da \$13 bei GET (ohne #) gelöscht ist
,ab8f	20 le el	jsr elle	Basic-Einsprung für CHKIN aufrufen (Eingabe von gewünschtem File)

; ab hier laufen GET und GET# gemeinsam ab, da vorher ggf. das Eingabefile gesetzt wurde

,ab92	a2 01	>ldx #01	Obergrenze für Eingabepuffer ist 1 (GET/GET# holt nur 1 Zeichen!)
,ab94	a0 02	ldy #02 >(\$0200)	HB des Eingabepuffers \$0200 laden
,ab96	a9 00	lda #00 <(\$0200)	LB des Eingabepuffers \$0200 laden
,ab98	8d 01 02	sta 0201	\$00 auch an Ende des 1 Byte langen Eingabepuffers ab \$0200 schreiben
,ab9b	a9 40	lda #40 %01000000	Flag für GET laden
,ab9d	20 0f ac	jsr ac0f	in Eingaberoutine einsteigen
,aba0	a6 13	ldx 13	Nummer des Eingabefiles laden
,aba2	d0 13	bne abb7	Eingabe von File, also GET# (Z=0): Eingabegerät zurücksetzen
,aba4	60	rts	Rücksprung von Routine

; Basic-Befehl INPUT# (Token: \$85)

,aba5	20 9e b7	jsr b79e "getbyt"	Bytewert (\$00-\$ff) für Filenummer des Eingabefiles laden
,aba8	a9 2c	lda #2c	Komma-Code als Prüfbyte laden
,abaa	20 ff ae	jsr aeff "chkbyt"	Komma als syntaktisches Erfordernis prüfen
,abad	86 13	stx 13	Eingabefile setzen; gleichzeitig Flag für Eingabe von File
,abaf	20 le el	jsr elle	Basic-Einsprung für CHKIN aufrufen
,abb2	20 ce ab	jsr abce	von "normaler" INPUT-Routine Eingabe einholen lassen

; hier: bei \$aa83 (PRINT#) und \$abe4 (INPUT) genutzter Einsprung

,abb5	a5 13	lda 13	Nummer des Eingabefiles holen; würde sich erübrigen, da der Akku bei \$abb7 wieder geändert wird und somit keine Berücksichtigung findet
,abb7	20 cc ff	>jsr ffcc "clrchn"	Zurücksetzen der I/O-Kanäle
,abba	a2 00	ldx #00 %00000000	\$00 (Flag für "Eingabe von Tastatur, nicht File")
,abbc	86 13	stx 13	in Flag/Hilfsspeicher für Basic-Eingaben schreiben
,abbe	60	rts	Rücksprung von Routine

; Routine zum Basic-Befehl INPUT (Token: \$86)

```

,abff c9 22      cmp #22      folgt Anführungszeichen auf INPUT-Befehl?
,abcl d0 0b      bne abce     nein (Z=0): nicht Kommunikationstext von INPUT bearbeiten
,abc3 20 bd ae    jsr aebd     String rechts von Anführungszeichen aus Basic-Text holen
,abc6 a9 3b      lda #3b      Semikolon als Prüfbyte laden      } Prüfung, ob
,abc8 20 ff ae    jsr aeff "chkbyt" syntaktisches Erfordernis überprüfen } Semikolon folgt
,abcb 20 21 ab    jsr ab21 "prtstr" String (s. $abc3) ausgeben
,abce 20 a6 b3→jsr b3a6 "chkdir" gibt Meldung ILLEGAL DIRECT ERROR aus, wenn INPUT im Direktmodus erfolgen soll
,abd1 a9 2c      lda #2c      Komma als DATA-Zwischenmarkierung laden
,abd3 8d ff 01    sta 01ff     und vor Systemeingabepuffer (ab $0200) schreiben
,abd6 20 f9 ab→jsr abf9       Fragezeichen über Basic-Einsprung ausgeben und Eingabe in Puffer holen
,abd9 a5 13      lda 13       Eingabe-Flag (File/Tastatur) laden
,abdb f0 0d      beq abea     Eingabe von Tastatur, also INPUT ohne # (Z=1): INPUT#-Sonderbehandlung überspringen
,abdd 20 b7 ff    jsr ffb7 "readst" Statusbyte holen
,abe0 29 02      and #02 %00000010 alle Bits bis auf bl löschen, also bl testen
,abe2 f0 06      beq abea     bl gelöscht, also kein TIMEOUT (Z=1): Sonderbehandlung überspringen
,abe4 20 b5 ab    jsr abb5     I/O-Kanäle zurücksetzen, Eingabeflag $13 löschen
,abe7 4c f8 a8    jmp a8f8 "ignorc" in DATA-Befehl einsteigen, so daß nächster Befehl ausgeführt wird
-----
,abea ad 00 02→lda 0200       erstes Byte des Eingabepuffers holen
,abed d0 1e      bne ac0d     nicht Endmarkierung (Z=0): in READ-Befehl einsteigen, damit Puffer ausgewertet wird
,abef a5 13      lda 13       Flag für Eingabe von Tastatur oder File holen
,abf1 d0 e3      bne abd6     Eingabe von File, also INPUT# (Z=0): weiter mit nächster Eingabe
,abf3 20 06 a9    jsr a906 "gosnxt" Offset zum nächsten Befehl in Y-Register holen } zum nächsten Befehl springen
,abf6 4c fb a8    jmp a8fb "adcgpt" CHRGET-Zeiger um Offset in Y erhöhen      } (durch CHRGET-Zeigeränderung)
-----

```

; Routine zur Ausgabe des Fragezeichens als Eingabe-Aufforderung bei INPUT (nicht INPUT#!)
und zur Eingabe einer Zeile über BASIN-Schleife

```

,abf9 a5 13      lda 13       Flag für Eingabegerät (Tastatur oder File) auslesen
,abfb d0 06      bne ac03     von File, also INPUT# (Z=0): Eingabezeile über BASIN-Schleife holen
,abfd 20 45 ab    jsr ab45 "qumout" Basic-Einsprung für Ausgabe eines Fragezeichens aufrufen
,ac00 20 3b ab    jsr ab3b "rgtspc" Basic-Einsprung für Ausgabe eines Leerzeichens aufrufen
,ac03 4c 60 a5→jmp a560 "getsyb" Eingabe einer Zeile in Systemeingabepuffer ab $0200 über BASIN-Schleife
-----

```

; Routine zum Basic-Befehl READ (Token: \$87)

,ac06	a6 41	ldx 41	LB des DATA-Zeigers auslesen	} Zeiger auf aktuelle DATA-Adresse nach X/Y aus Zeiger holen Flag für Eingabebefehl (\$98=READ/\$00=INPUT) laden
,ac08	a4 42	ldy 42	HB des DATA-Zeigers auslesen	
,ac0a	a9 98	lda #98	READ-Flag laden	
,ac0c	2c a9 00	"bit" lda #00	INPUT-Flag laden	
,ac0f	85 11	sta 11	Flag für READ (Einsprung bei \$ac06) oder INPUT (Einsprung bei \$ac0c) setzen	
,ac11	86 43	stx 43	LB des Zeigers für READ/INPUT-Eingabe setzen	} READ/INPUT-Eingabezeiger mit Adresse in X/Y belegen
,ac13	84 44	sty 44	HB des Zeigers für READ/INPUT-Eingabe setzen	
,ac15	20 8b b0	jsr b08b "fndvar"	Eingabevariable im Speicher suchen, falls nicht vorhanden: anlegen	
,ac18	85 49	sta 49	LB des FOR/NEXT-Variablenzeigers setzen	} FOR/NEXT-Variablenzeiger \$49/\$4a zeigt jetzt auf die Variable
,ac1a	84 4a	sty 4a	HB des FOR/NEXT-Variablenzeigers setzen	
,ac1c	a5 7a	lda 7a	LB des CHRGET-Zeigers holen	} CHRGET-Zeiger auslesen und in Zwischenspeicher
,ac1e	a4 7b	ldy 7b	HB des CHRGET-Zeigers holen	
,ac20	85 4b	sta 4b	LB des Zwischenspeichers \$4b/\$4c setzen	} \$4b/\$4c schreiben
,ac22	84 4c	sty 4c	HB des Zwischenspeichers \$4b/\$4c setzen	
,ac24	a6 43	ldx 43	LB des Zeigers für READ/INPUT-Eingabe holen	} CHRGET-Zeiger auf die Adresse, ab der die
,ac26	a4 44	ldy 44	HB des Zeigers für READ/INPUT-Eingabe holen	
,ac28	86 7a	stx 7a	LB des CHRGET-Zeigers auf READ/INPUT-Eingabe richten	} READ/INPUT-Eingabe im Speicher ist, stellen
,ac2a	84 7b	sty 7b	HB des CHRGET-Zeigers auf READ/INPUT-Eingabe richten	
,ac2c	20 79 00	jsr 0079 "chrgot"	Zeichen an neuer CHRGET-Zeigerposition holen	
,ac2f	d0 20	bne ac51	noch keine Endmarkierung (Z=0): Sonderbehandlung überspringen	

; Sonderbehandlung: Endmarkierung der Eingabe an erster Position des Systemeingabepuffers
(tritt ein, wenn bei Eingabe nur <RETURN> betätigt wurde)

,ac31	24 11	bit 11	Eingabeflag (GET/INPUT/READ) testen
,ac33	50 0c	bvc ac41	nicht GET, sondern hier INPUT oder READ (V=0): zu READ/INPUT-Behandlung bei nicht erfolgter Eingabe

; Sonderfall: Endmarkierung der Eingabe an erster Position des Systemeingabepuffers bei Befehl GET

,ac35	20 24 e1	jsr e124	Basic-Einsprung für GETIN-Routine aufrufen
,ac38	8d 00 02	sta 0200	und Zeichen in Systemeingabepuffer schreiben (umfaßt bei GET nur die Adresse \$0200)
,ac3b	a2 ff	ldx #ff <(\$01ff)	LB der Adresse des Systemeingabepuffers - 1 laden
,ac3d	a0 01	ldy #01 >(\$01ff)	HB der Adresse des Systemeingabepuffers - 1 laden
,ac3f	d0 0c	bne ac4d "jmp"	CHRGET-Zeiger mit X/Y laden und weiter

; Sonderfall: Endmarkierung der Eingabe an erster Position des Systemeingabepuffers bei Befehl READ oder INPUT

,ac41 30 75 →bmi acb8 READ (N=1): Sonderbehandlung für READ anspringen

; Sonderfall: Behandlung von INPUT

,ac43 a5 13 lda 13 Flag für INPUT-Kommunikationsstring holen
 ,ac45 d0 03 bne ac4a keinKommentarstring (Z=0): Sonderbehandlung "Ausgabe von Fragezeichen" überspringen
 ,ac47 20 45 ab jsr ab45 "qumout" weiteres Fragezeichen ausgeben (über Basic-Einsprung für Ausgabe von "?")
 ,ac4a 20 f9 ab →jsr abf9 bei Bedarf zweites Fragezeichen ausgeben lassen
 ,ac4d 86 7a →stx 7a LB des CHRGET-Zeigers setzen } CHRGET-Zeiger mit X/Y (Zeiger auf Adresse der
 ,ac4f 84 7b sty 7b HB des CHRGET-Zeigers setzen } Eingabe im Speicher) belegen
 ,ac51 20 73 00 jsr 0073 "chrget" CHRGET-Zeiger auf nächstes Zeichen stellen
 ,ac54 24 0d bit 0d Datentyp-Flag für Variable testen
 ,ac56 10 31 ↓bpl ac89 numerisch, nicht String (N=0): zur Eingabe-Auswertung für numerische Variable

; String-Variable aus Eingabepuffer beziehen

,ac58 24 11 bit 11 Flag für Eingabebefehl testen
 ,ac5a 50 09 bvc ac65 nicht GET (V=0): GET-Sonderbehandlung überspringen
 ,ac5c e8 inx LB des Zeigers in Eingabepuffer erhöhen
 ,ac5d 86 7a stx 7a und als CHRGET-Zeiger-LB setzen
 ,ac5f a9 00 lda #00 Initialisierungswert für Hilfsspeicher \$07 laden
 ,ac61 85 07 sta 07 und in Hilfsspeicher \$07 schreiben (als Flag für GET)
 ,ac63 f0 0c beq ac71 "jmp" weiter bei \$ac71; Akku enthält \$00 auch als Initialisierungswert für Speicher \$08

 ,ac65 85 07 →sta 07 Hilfsspeicher \$07 setzen (Akku hat hier wegen \$ac5a seit \$ac51 anderen Wert als \$00)
 ,ac67 c9 22 cmp #22 Vergleich des über CHRGET gehalten Zeichens mit Anführungszeichen
 ,ac69 f0 07 beq ac72 Übereinstimmung (Z=1): QUOTE-MODE-Flag für Eingabe ändern
 ,ac6b a9 3a lda #3a ASCII-Code für Doppelpunkt laden
 ,ac6d 85 07 sta 07 und in Hilfsspeicher \$07 schreiben
 ,ac6f a9 2c lda #2c ASCII-Code für Komma laden
 ,ac71 18 →clc Carry vor Addition bei \$ac78 löschen
 ,ac72 85 08 →sta 08 Hilfsspeicher \$08 belegen (s. \$ac6f)
 ,ac74 a5 7a lda 7a LB des CHRGET-Zeigers holen
 ,ac76 a4 7b ldy 7b HB des CHRGET-Zeigers holen
 ,ac78 69 00 adc #00 Addition von 0 zum LB (falls vorher \$ac71 abgearbeitet wurde) oder 1 (s.\$ac67/\$ac69)
 ,ac7a 90 01 bcc ac7d kein Übertrag (C=1): Erhöhung des HB überspringen
 ,ac7c c8 iny HB (s. \$ac76) erhöhen
 ,ac7d 20 8d b4 →jsr b48d String in temporären Stringstapel kopieren
 ,ac80 20 e2 b7 jsr b7e2 CHRGET-Zeiger hinter String positionieren

```
,ac83 20 da a9 jsr a9da      Zuweisungsroutine für Stringvariable aufrufen
,ac86 4c 91 ac jmp ac91      weiter mit allgemeinem Parameter-Auswertungsteil für READ/INPUT
```

```
-----
; numerische Variable aus Systemeingabepuffer auswerten
```

```
,ac89 20 f3 bc jsr bcf3 "strflp" VAL-Funktion aufrufen, um numerischen Wert eines Strings in FAC zu holen
,ac8c a5 0e lda 0e          Datentypflag Integer/Fließkomma auslesen
,ac8e 20 c2 a9 jsr a9c2      numerische Variable mit FAC-Inhalt belegen
,ac91 20 79 00 jsr 0079 "chrget" letztes Zeichen hinter Variable holen
,ac94 f0 07 beq ac9d        Endmarkierung (Z=1): Fortsetzung der Befehle READ/INPUT
,ac96 c9 2c cmp #2c        folgt Komma?
,ac98 f0 03 beq ac9d        ja (Z=1): Fortsetzung der Befehle READ/INPUT
,ac9a 4c 4d ab jmp ab4d     ansonsten Fehlerbehandlung bei Eingabebefehlen aufrufen
```

```
-----
; Fortsetzung der Befehle READ/INPUT, wenn weitere Parameter zu bearbeiten sind
```

```
,ac9d a5 7a >lda 7a        LB des CHRGET-Zeigers holen } aktuelle CHRGET-Zeiger-Position
,ac9f a4 7b ldy 7b        HB des CHRGET-Zeigers holen } (Systemeingabepuffer oder DATA-Position)
,aca1 85 43 sta 43        LB des INPUT-Zeigers setzen } in Eingabezeiger $43/$44 für INPUT/READ
,aca3 84 44 sty 44        HB des INPUT-Zeigers setzen } übertragen, um auf nächsten Parameter zu stellen
,aca5 a5 4b lda 4b        LB des bei $ac20/$ac22 geretteten CHRGET-Zeigers holen } alten Zustand des
,aca7 a4 4c ldy 4c        HB des bei $ac20/$ac22 geretteten CHRGET-Zeigers holen } CHRGET-Zeigers
,aca9 85 7a sta 7a        LB des CHRGET-Zeigers darauf richten } (vor INPUT/READ)
,acab 84 7b sty 7b        HB des CHRGET-Zeigers darauf richten } wiederherstellen
,acad 20 79 00 jsr 0079 "chrget" letztes Zeichen holen
,acb0 f0 2d beq acdf        Endmarkierung (Z=1): Schlußbedingungen überprüfen und ggf. Ende
,acb2 20 fd ae jsr aefd "chkcom" auf Komma als syntaktisches Erfordernis prüfen
,acb5 4c 15 ac jmp ac15     zurück an Beginn der READ/INPUT-Schleife
```

```
-----
; Sonderbehandlung für READ
```

```
,acb8 20-06-a9->jsr a906 "gosnxt" Offset zur nächsten Befehlsendmarkierung in Y berechnen
,acbb c8 iny             Offset um 1 erhöhen
,acbc aa tax             entsprechendes Befehls-Trennzeichen zwecks Test in X-Register bringen
,acbd d0 12 bne acdl      war nicht $00, sondern $3a (Z=0): Sonderbehandlung für Zeilenende überspringen
```

```
; Sonderbehandlung: Ende der DATA-Zeile, DATA-Zeiger auf nächstes DATA richten
```

```
,acbf a2 0d ldx #0d        Fehlercode für OUT OF DATA laden, falls er benötigt wird (s. $acc4)
,accl c8 iny             Offset noch ein Byte weiter vorrücken lassen
,acc2 b1 7a lda (7a),y     HB des Linkpointers der nächsten Zeile holen
```

,acc4	f0 6c	beq ad32	Endmarkierung für Programm (Z=1): Fehlereinsprung indirekt aufrufen, s. \$acbf
,acc6	c8	iny	Offset auf LB der nächsten Zeilennummer richten
,acc7	b1 7a	lda (7a),y	LB der nächsten Zeilennummer holen
,acc9	85 3f	sta 3f	und in LB des Zeigers auf aktuelle DATA-Zeilennummer
,accb	c8	iny	Offset auf HB der nächsten Zeilennummer richten
,accc	b1 7a	lda (7a),y	HB der nächsten Zeilennummer holen
,acce	c8	iny	Offset erhöhen, falls bei \$acdl erforderlich
,accf	85 40	sta 40	HB des Zeigers auf aktuelle DATA-Zeilennummer setzen
,acd1	20 fb a8	→jsr a8fb "adcgpt"	CHRGPT-Zeiger um Offset in Y-Register erhöhen, also auf erstes Befehlsbyte der neuen Zeile richten (s. \$acce)
,acd4	20 79 00	jsr 0079 "chrgot"	Zeichen an neuer Position holen
,acd7	aa	tax	und in X-Register bringen
,acd8	e0 83	cpx #83	Vergleich mit Token für DATA
,acda	d0-dc	bne acb8	Vergleich negativ (Z=0): Suchschleife fortsetzen, bis DATA-Befehl gefunden
,acdc	4c 51 ac	jmp ac51	weiter in READ/INPUT-Schleife

; Schlußbedingungen für READ/INPUT überprüfen

,acdf	a5 43	lda 43	LB des Zeigers für READ/INPUT holen
,acel	a4 44	ldy 44	HB des Zeigers für READ/INPUT holen
,ace3	a6 11	ldx 11	Eingabeart-Flag holen (zwecks Test)
,ace5	10 03	bpl acea	nicht READ, sondern INPUT/GET (N=0): noch kein Abbruch
,ace7	4c 27 a8	jmp a827	A/Y in DATA-Adreßzeiger abspeichern und Ende

,acea	a0 00	→ldy #00	Offset mit \$00 initialisieren
,acec	b1 43	lda (43),y	Byte an aktueller READ/INPUT-Position holen
,acee	f0 0b	beq acfb	Endmarkierung \$00 (Z=1): RTS anspringen, Ende der Routine
,acf0	a5 13	lda 13	INPUT#-Flag auslesen
,acf2	d0 07	bne acfb	Kommentar (Z=0): RTS anspringen, Ende der Routine
,acf4	a9 fc	lda #fc <(\$acfc)	LB von \$acfc (Adresse von EXTRA IGNORED) laden
,acf6	a0 ac	ldy #ac >(\$acfc)	HB von \$acfc (Adresse von EXTRA IGNORED) laden
,acf8	4c 1e ab	jmp able "strout"	Text "?EXTRA IGNORED" ausgeben

,acfb	60	→rts	Rücksprung von Routine
-------	----	------	------------------------

; Steuermeldungen für INPUT, auszugeben über STROUT (nicht über ERROR!)

,acfc	3f 45 58 54 52 41 20 49 47 4e 4f 52 45 44 0d 00	?extra ignored[cr,null]
,ad0c	3f 52 45 44 4f 20 46 52 4f 4d 20 53 54 41 52 54 0d 00	?redo from start[cr,null]

; Routine für Basic-Befehl NEXT (Token: \$82)

,ad1e	d0 04	bne ad24	keine Endmarkierung hinter NEXT-Befehl, also Syntax "NEXT variable" (Z=0): nach \$ad24
,ad20	a0 00	ldy #00	HB des Variablenzeigers mit \$00 belegen (Flag für "keine Variable")
,ad22	f0 03	beq ad27 "jmp"	und in weitere NEXT-Routine einsteigen

; Sonderbehandlung: Syntax "NEXT variable", also z.B. "NEXT I"

,ad24	20 8b b0	>jsr b08b "fndvar"	Variable suchen
,ad27	85 49	>sta 49	LB der Adresse nach \$49 (LB des FOR/NEXT-Variablenzeigers) } FOR/NEXT-Variablen-
,ad29	84 4a	sty 4a	HB der Adresse nach \$4a (HB des FOR/NEXT-Variablenzeigers) } zeiger setzen
,ad2b	20 8a a3	jsr a38a "srcstk"	auf Stapel nach FOR/NEXT-Variable suchen
,ad2e	f0 05	beq ad35	gefunden (Z=1): keine Fehlermeldung NEXT WITHOUT FOR generieren
,ad30	a2 0a	ldx #0a	Fehlercode für NEXT WITHOUT FOR laden } Erzeugung von
,ad32	4c 37 a4	jmp a437 "error"	Fehlereinsprung aufrufen } NEXT WITHOUT FOR ERROR

,ad35	9a	>txs	Stapelzeiger auf gefundenen Eintrag richten
,ad36	8a	txa	und zwecks Addition in Akku transportieren
,ad37	18	clc	Carry vor Addition löschen
,ad38	69 04	adc #04	4 addieren
,ad3a	48	pha	Ergebnis auf Stapel bis \$ad3f merken
,ad3b	69 06	adc #06	6 addieren (mittlerweile wurde schon 10 addiert)
,ad3d	85 24	sta 24	Ergebnis in \$24 als LB der Adresse des Schleifenendwertes, der sich im Stapeleintrag befindet, für \$ad57 vormerken
,ad3f	68	pla	bei \$ad3a gemerkten Stapelzeiger+4 holen (als LB der Adresse der Schrittweite)
,ad40	a0 01	ldy #01 >(\$0100)	HB der Adresse laden
,ad42	20 a2 bb	jsr bba2 "movmf"	Schrittweite, die auf Stapel gefunden wurde, in FAC #1 holen
,ad45	ba	tsx	Stapelzeiger in X-Register als Offset holen
,ad46	bd 09 01	lda 0109,x	Wert vom Stapel holen
,ad49	85 66	sta 66	und Vorzeichen dementsprechend setzen
,ad4b	a5 49	lda 49	LB des FOR/NEXT-Variablenzeigers holen } FOR/NEXT-Variablenzeiger
,ad4d	a4 4a	ldy 4a	HB des FOR/NEXT-Variablenzeigers holen } nach A/Y auslesen
,ad4f	20 67 b8	jsr b867 "addmem"	Schrittweite (wurde vom Stapel geholt und ist jetzt in ARG) zu Schleifenvariable addieren
,ad52	20 d0 bb	jsr bbd0 "facvar"	FAC #1-Inhalt in Schleifenvariable kopieren (ist jetzt um Schrittweite erhöht)
,ad55	a0 01	ldy #01 >(\$0100)	HB der Konstanten laden (\$0100 = Prozessorstapel)
,ad57	20 5d bc	jsr bc5d	und Vergleich von FAC #1 und Konstante "Endwert der Schleife"
,ad5a	ba	tsx	Stapelzeiger in X-Register als Offset holen
,ad5b	38	sec	Carry vor Subtraktion setzen
,ad5c	fd 09 01	sbc 0109,x	vom Akku (Vergleichsergebnis, s. Routinenbeschreibung \$bc5d) Schleifenrichtung (= Vorzeichen der Schrittweite) zwecks Vergleich subtrahieren

,ad5f	f0 17	beq ad78	beide hatten denselben Wert (Z=1): Schleife beenden	
,ad61	bd 0f 01	lda 010f,x	LB der Zeilennummer des ersten Befehls nach FOR holen	} Zeilennummer des ersten Befehls nach
,ad64	85 39	sta 39	und als LB der aktuellen Zeilennummer setzen	
,ad66	bd 10 01	lda 0110,x	HB der Zeilennummer des ersten Befehls nach FOR holen	} FOR als aktuelle Zeilennummer setzen
,ad69	85 3a	sta 3a	und als HB der aktuellen Zeilennummer setzen	
,ad6b	bd 12 01	lda 0112,x	LB der Adresse des ersten Befehls nach FOR holen	} Adresse des ersten Befehls nach
,ad6e	85 7a	sta 7a	und als LB des CHRGET-Zeigers setzen	
,ad70	bd 11 01	lda 0111,x	HB der Adresse des ersten Befehls nach FOR holen	} FOR als aktuelle Adresse setzen
,ad73	85 7b	sta 7b	und als HB des CHRGET-Zeigers setzen	
,ad75	4c ae a7	→ jmp a7ae "intprt"	zur Interpreterschleife springen, die jetzt den Schleifenrumpf ausführt	

; NEXT-Befehl: Sonderfall "Schleife beenden"

,ad78	8a	→ txa	Stapelzeiger zwecks Addition in Akku transportieren
,ad79	69 11	adc #11	#17 (Länge eines FOR/NEXT-Stapeleintrags) und 1 (da C=1 seit \$ad5c) addieren
,ad7b	aa	tax	Ergebnis zurück ins X-Register
,ad7c	9a	txs	und von dort in den Stapelzeiger
,ad7d	20 79 00	jsr 0079 "chrget"	letztes Zeichen hinter "NEXT variable" holen
,ad80	c9 2c	cmp #2c	Komma?
,ad82	d0 f1	bne ad75	nein (Z=0): zurück zur Interpreterschleife, Ende

; NEXT-Befehl: Sonderfall "NEXT variable1,variable2..."

,ad84	20 73 00	jsr 0073 "chrget"	CHRGET-Zeiger auf erstes Byte hinter Abgrenzungskomma stellen
,ad87	20 24 ad	jsr ad24 "jmp"	Sonderfall "NEXT variable" bearbeiten;
			JSR täuscht: kein Rücksprung, da Stapel manipuliert wird, und statt über RTS mit "jmp \$a7ae" zur Interpreterschleife zurückgesprungen wird

; FRMNUM-Routine: Auswertung numerischer Ausdrücke (auch Variablen und logische Ausdrücke erlaubt);
Aufruf von \$a775 (FOR), \$a79c (FOR), \$b438 (FN), \$b79e (GETBYT), \$b7eb (GETWRB) und \$e12a (SYS)

,ad8a	20 9e ad	jsr ad9e "frmevl"	beliebigen Ausdruck holen;
			im Speicher folgt CHKNUM-Routine

; CHKNUM-Routine: Prüfung über FRMEVL eingeholter Ausdrücke auf numerischen Datentyp

,ad8d	18	clc	C=0 als Flag für "Prüfung auf numerisch" setzen; Im Speicher folgt CHKTYP-Routine.
-------	----	-----	--

; bei \$ad8f: CHKSTR-Einsprung (prüft auf String);

,ad8e	24 38	"bit" sec	C=1 setzen als Flag für "Prüfung auf String"
-------	-------	-----------	--

; CHKTYP-Routine: prüft auf numerischen Ausdruck, wenn C=0 (s. z.B. \$ad8d), auf Strings, wenn C=1
wenn erforderter Datentyp nicht zutrifft, erfolgt Fehlermeldung TYPE MISMATCH;
Verwendung von \$a9bc (LET) und \$b016 (Variablenvergleich) aus

,ad90	24 0d	bit 0d	Datentyp-Flag String/Numerisch testen
,ad92	30 03	bmi ad97	String (N=1): Fehler, wenn C=0 war; richtig, wenn C=1 war
,ad94	b0 03	bcs ad99	numerisch wegen \$ad92; Fehler (C=1): String gewünscht und Zahl gefunden
,ad96	60	→rts	richtig, wenn C=0 und numerisch oder C=1 und String

,ad97	b0 fd	→bcs ad96	String gewünscht und gefunden (C=1): RTS anspringen, da Übereinstimmung
,ad99	a2 16	→ldx #16	Fehlercode für TYPE MISMATCH laden } Fehlermeldung TYPE MISMATCH
,ad9b	4c 37 a4	jmp a437 "error"	Fehlermeldung ausgeben } (wegen falschen Datentyps) erzeugen

; FRMEVL-Routine: Auswertung beliebiger Ausdrücke (Strings/numerische; auch Variablen und logische Ausdrücke erlaubt)
Aufruf von \$a928 (IF), \$a9b7 (LET), \$aab5 (PRINT), \$ad8a (FRMNUM), \$acf4 (EVAL), \$afb4 (Stringauswertung in FRMEVL)
und \$blb5 (INTEVL)

,ad9e	a6 7a	ldx 7a	LB des CHRGET-Zeigers auslesen	} CHRGET-Zeiger um 1 dekrementieren, damit er vor dem auszuwertenden Parameter steht
,ada0	d0 02	bne ada4	<> 0 (Z=0): nur LB dekrementieren	
,ada2	c6 7b	dec 7b	HB des CHRGET-Zeigers dekrementieren	
,ada4	c6 7a	→dec 7a	LB des CHRGET-Zeigers dekrementieren	
,ada6	a2 00	ldx #00	\$00 als Initialisierungswert für Prioritätsflag laden (\$00 als Prioritätsflag heißt, daß neuer Ausdrucksbestandteil auszuwerten ist)	
,ada8	24 48	"bit" pha	Akku (Operatormaske) merken, Beginn der FRMEVL-Schleife (bei erstem Durchlauf wegen "bit" ignoriert); \$ada9 wird nur von \$ae2d aus angesprochen	
,adaa	8a	txa	Akku mit Wert im X-Register belegen (beim ersten Mal \$00, s. \$ada6)	
,adab	48	pha	und Akku auf Stapel legen	
,adac	a9 01	lda #01	Hälfte von 2 (= benötigter Stapelplatz) laden	
,adae	20 fb a3	jsr a3fb "getstk"	und auf freien Platz auf Stapel testen	
,adb1	20 83 ae	jsr ae83 "eval"	nächsten Ausdrucksbestandteil auswerten	
,adb4	a9 00	lda #00 %00000000	\$00 als Initialisierungswert für Operatormaske laden	
,adb6	85 4d	sta 4d	und als aktuelle Operatormaske setzen	
,adb8	20 79 00	jsr 0079 "chrgot"	Zeichen hinter letztem ausgewerteten Ausdrucksbestandteil holen	
,adbb	38	sec	Carry vor Subtraktion setzen	
,adbc	e9 b1	sbc #b1	Token für ">" (niedrigstes Token von Vergleichsoperator) subtrahieren	
,adbe	90 17	→bcc add7	kleineres Token als Vergleichsoperator (C=0): keine Sonderbehandlung für "<=>"	
,adc0	c9 03	cmp #03	Token auch < \$b1+\$03=\$b4 (Token von "<" + 1)?	
,adc2	b0 13	→bcs add7	nein (C=1): Sonderbehandlung für "<=>" nicht auslösen	

; Sonderbehandlung: Vergleichsoperator wie <, = oder >. Im Akku steht \$00 für >, \$01 für = und \$02 für <.

,adc4	c9 01	cmp #01	Carry löschen, wenn nicht \$b1 gewesen (Token für >)
,adc6	2a	rol	Carry-Flag in Akku übernehmen (Ergebnis von \$adc4 berücksichtigen)
,adc7	49 01	eor #01 %00000001	b0 invertieren; ist danach nur bei "<" oder ">" gesetzt
,adc9	45 4d	eor 4d	Verknüpfung mit Operatormaske, um doppelt vorkommende Operatoren auszuschalten
,adcb	c5 4d	cmp 4d	Vergleich mit Operatormaske
,adcd	90 61	↓ bcc ae30	doppelt auftretender Operator (C=0): SYNTAX ERROR indirekt auslösen
,adcf	85 4d	sta 4d	kein Fehler, also neue Operatormaske (neue Verknüpfung enthalten) merken
,add1	20 73 00	jsr 0073 "chrget"	CHRGET-Zeiger auf nächstes Byte nach Operator richten
,add4	4c bb ad	jmp adbb	zurück in FRMEVL-Operator-Auswertungsschleife

; Fortsetzung von \$adc2 aus, wenn kein Vergleichsoperator zu bearbeiten war

,add7	a6 4d	ldx 4d	Operatormaske holen
,add9	d0 2c	↓ bne ae07	mindestens 1 Bit gesetzt, also <, = oder > oder Kombinationen (Z=0): Prioritätsberechnung für andere Operatoren überspringen
,addb	b0 7b	↓ bcs ae58	Token war vor \$adbc größer als \$b3 (C=1): Funktion ausführen lassen
,addd	69 07	adc #07	ansonsten \$07 (Carry ist seit \$addb gelöscht) addieren (dient Test auf +, -, *, /, ↑, AND, OR als Operatoren)
,addf	90 77	↓ bcc ae58	kein solcher Operator (C=0): Funktion auswerten lassen
,adel	65 0d	adc 0d	Datentyp-Flag String/numerisch addieren (zwecks Test) bei x\$+y\$ ergibt dies \$01 ("+"-Code) + \$ff (String-Flag) = 0
,ade3	d0 03	↓ bne ade8	nicht String-Addition (Z=0): Sonderbehandlung Stringverknüpfung überspringen
,ade5	4c 3d b6	jmp b63d	Stringverkettung anspringen

; Fortsetzung von \$ade3 aus, wenn keine Stringverkettung zu bearbeiten war

Es erfolgt die Berechnung eines Zeigers Y auf das Prioritätsflag in der ROM-Tabelle für Operationen (ab \$a080)

,ade8	69 ff	↓ adc #ff %11111111	bewirkt Subtraktion von 1, da Übertrag vernachlässigt wird; Ergebnis: Nummer des Eintrag in ROM-Tabelle ab \$a080 für entsprechende Operation
,adea	85 22	sta 22	Ergebnis merken
,adec	0a	asl	} Akku (Nummer der numerischen Operation in ROM-Tabelle für Operationen) mit 3 multiplizieren, um Offset innerhalb der Tabelle zu erhalten
,aded	65 22	adc 22	
,adef	a8	tay	Ergebnis in Offset-Register Y
,adf0	68	→ pla	altes Prioritätsflag wieder vom Stapel holen
,adf1	d9 80 a0	cmp a080,y	Vergleich mit neuem Prioritätsflag
,adf4	b0 67	↓ bcs ae5d	altes Prioritätsflag >= neues (C=1): Sonderbehandlung ab \$ae5d
,adf6	20 8d ad	jsr ad8d "chknun"	auf numerischen Parameter testen
,adf9	48	pha	und Akku-Inhalt (Operand) auf Stapel legen

,adfa	20 20 ae	jsr ae20	Operand auf Stapel legen
,adfd	68	pla	Akku-Inhalt (Operand) von \$adf9 vom Stapel holen
,adfe	a4 4b	ldy 4b	Prioritätsflag-Zeiger auslesen
,ae00	10 17	bpl ae19	Zeiger auf Prioritätsflag war kleiner als \$80 (N=0): Sonderbehandlung
,ae02	aa	tax	Akku (s. \$adfd) zwecks Test in X-Register transportieren
,ae03	f0 56	beq ae5b	Akku = \$00 (Z=1): Ende der Auswertung
,ae05	d0 5f	bne ae66 "jmp"	bei \$adfa geretteten Operand vom Stapel zurückholen

; Prioritätsberechnung für >,=,<

,ae07	46 0d	lsr 0d	Datentyp-Flag String/numerisch auswerten (b7 in Carry-Flag holen)
,ae09	8a	txa	und alte Operatormaske in Akku schreiben
,ae0a	2a	rol	Carry-Flag von \$ae07 in Operatormaske nehmen
,ae0b	a6 7a	ldx 7a	LB des CHRGET-Zeigers auslesen
,ae0d	d0 02	bne ae11	<> 0 (Z=0): HB nicht dekrementieren
,ae0f	c6 7b	dec 7b	HB dekrementieren
,ae11	c6 7a	dec 7a	LB dekrementieren
,ae13	a0 1b	ldy #1b	Zeiger für Prioritätsflag bei Vergleichsoperatoren laden
,ae15	85 4d	sta 4d	Operatormaske (s. \$ae09) merken
,ae17	d0-d7	bne adf0 "jmp"	Zeiger auf Prioritätsflag berechnen

; Sonderbehandlung: Zeiger auf Prioritätsflag war kleiner als \$80

,ael9	d9 80-a0	→cmp a080,y	Vergleich mit Prioritätsflag aus ROM-Tabelle
,aelc	b0 48	→bcs ae66	Akku >= ROM-Tabelleninhalt (C=1): Operand vom Stapel holen
,aele	90 d9	↑bcc adf9 "jmp"	in Behandlung "aktuelles Prioritätsflag größer" so einsteigen, daß kein Test auf numerisch erfolgt

; Unterroutine: Operand auf den Stapel retten; Aufruf von \$adfa aus

,ae20	b9 82 a0	lda a082,y	HB der Funktionsadresse aus ROM-Tabelle entnehmen	Funktionsadresse aus ROM-Tabelle ermitteln und auf den Stapel legen
,ae23	48	pha	und auf den Stapel legen	
,ae24	b9 81 a0	lda a081,y	LB der Funktionsadresse aus ROM-Tabelle entnehmen	
,ae27	48	pha	und auf den Stapel legen	
,ae28	20 33 ae	jsr ae33 "facstk"	Unterroutine "FAC auf Stapel legen" aufrufen	
,ae2b	a5 4d	lda 4d	Operatormaske holen	
,ae2d	4c a9 ad	jmp ada9	und in FRMEVL-Auswertungsschleife springen	

,ae30	4c 08 af	jmp af08 "synerr"	SYNTAX ERROR hervorrufen
-------	----------	-------------------	--------------------------

; FACSTK-Routine: FAC auf Stapel legen; Aufruf von \$ae28 aus

,ae33	a5 66	lda 66	Vorzeichen von FAC #1 auslesen	
,ae35	be 80 a0	ldx a080,y	Prioritätsflag aus ROM-Tabelle holen	
,ae38	a8	tay	Vorzeichen (s. \$ae33) nach Y	
,ae39	68	pla	LB der Rücksprungadresse vom Stapel holen	} Rücksprungadresse vom Stapel in den Hilfsvektor \$0022/\$0023 schreiben
,ae3a	85 22	sta 22	und in LB von \$22/\$23 merken	
,ae3c	e6 22	inc 22	LB von \$22/\$23 um 1 erhöhen	
,ae3e	68	pla	HB der Rücksprungadresse vom Stapel holen	
,ae3f	85 23	sta 23	und in HB von \$22/\$23 merken	
,ae41	98	tya	Vorzeichen (s. \$ae38) wieder in Akku	
,ae42	48	pha	und auf den Stapel legen	
,ae43	20 1b bc	jsr bclb "roundf"	FAC #1 runden	
,ae46	a5 65	lda 65	Mantissenbyte #1 holen	} alle 4 Mantissenbytes und das Exponentenbyte von FAC #1 auf den Stapel legen lassen
,ae48	48	pha	und auf den Stapel legen	
,ae49	a5 64	lda 64	Mantissenbyte #2 holen	
,ae4b	48	pha	und auf den Stapel legen	
,ae4c	a5 63	lda 63	Mantissenbyte #3 holen	
,ae4e	48	pha	und auf den Stapel legen	
,ae4f	a5 62	lda 62	Mantissenbyte #4 holen	
,ae51	48	pha	und auf den Stapel legen	
,ae52	a5 61	lda 61	Exponentenbyte holen	
,ae54	48	pha	und auf den Stapel legen	
,ae55	6c 22 00	jmp(0022)	Rücksprung über \$22/\$23 (s. \$ae39-\$ae3f)	

; FRMEVL-Sonderfall: Ausführung einer Funktion

,ae58	a0 ff	ldy #ff %11111111	Flag für Zeiger auf Operatormaske laden	
,ae5a	68	pla	altes Prioritätsflag vom Stapel holen	
,ae5b	f0 23	beq ae80	Ende-Flag \$00 (Z=1): Ende der Auswertung veranlassen	
,ae5d	c9 64	cmp #64	Vergleich mit Prioritätsflag für Vergleichsoperatoren (<,<=,> und Kombinationen)	
,ae5f	f0 03	beq ae64	positiv (Z=1): keine Prüfung auf numerisch, da auch für Strings möglich	
,ae61	20 8d ad	jsr ad8d "chknum"	prüft, ob Datentyp-Flag auf "numerisch" steht	
,ae64	84 4b	sty 4b	Zeiger auf Prioritätsflag mit \$ff (s. \$ae58) laden (höchste Priorität!!)	
,ae66	68	pla	Vorzeichenbyte vom Stapel holen	} Vorzeichenbyte für TAN-Funktion, alle 4 Mantissenbytes, Exponenten- und Vorzeichenbyte
,ae67	4a	lsr	durch 2 dividieren (b0 ins Carry)	
,ae68	85 12	sta 12	und ins TAN-Vorzeichenflag übernehmen	
,ae6a	68	pla	Exponentenbyte vom Stapel holen	
,ae6b	85 69	sta 69	und in FAC #2/Exponentenbyte übernehmen	
,ae6d	68	pla	Mantissenbyte #4 vom Stapel holen	

,ae6e	85 6a	sta 6a	und in FAC #2/Mantissenbyte #4 übernehmen	von FAC #2
,ae70	68	pla	Mantissenbyte #3 vom Stapel holen	(auch ARG
,ae71	85 6b	sta 6b	und in FAC #2/Mantissenbyte #3 übernehmen	genannt)
,ae73	68	pla	Mantissenbyte #2 vom Stapel holen	schreiben;
,ae74	85 6c	sta 6c	und in FAC #2/Mantissenbyte #2 übernehmen	} die Werte werden
,ae76	68	pla	Mantissenbyte #1 vom Stapel holen	
,ae77	85 6d	sta 6d	und in FAC #2/Mantissenbyte #1 übernehmen	geholt und direkt
,ae79	68	pla	Vorzeichenbyte # vom Stapel holen	in den Speicher
,ae7a	85 6e	sta 6e	und in FAC #2/Vorzeichenbyte übernehmen	übernommen.
,ae7c	45 66	eor 66	Verknüpfung mit FAC #1-Vorzeichen	
,ae7e	85 6f	sta 6f	Ergebnis als Ergebnis des Vorzeichenvergleichs merken	
,ae80	a5 61	>lda 61	Exponentenbyte von FAC #1 laden	
,ae82	60	rts	und Rücksprung von Routine	

; EVAL-Routine: nächsten Ausdrucksbestandteil auswerten;
 Aufruf von \$adbl (FRMEVL) sowie \$b643 (Stringverknüpfung)

,ae83 6c 0a 03 jmp(030a) Sprung über Vektor IEVAL \$030a/\$030b; zeigt normalerweise nach \$ae86

,ae86	a9 00	lda #00	Flag für "numerische Variable" laden	} auf "numerisch"
,ae88	85 0d	sta 0d	und in Flag für Variablen-Datentyp (String/numerisch) schreiben	
,ae8a	20 73 00	jsr 0073 "chrget"	nächstes Zeichen für Auswertung holen	
,ae8d	b0 03	bcs ae92	keine Ziffer (C=1): nicht Konvertierung von Zahl in Fließkomma-Format aufrufen	
,ae8f	4c f3 bc	>jmp bcf3 "strflp"	ASCII-Codes als Zahl interpretieren und in FAC #1 holen	
,ae92	20 13 b1	>jsr b113 "chkltr"	Prüfroutine, ob Buchstabencode im Akku steht	
,ae95	90 03	bcc ae9a	kein Buchstabe (C=0): nicht Auswertung einer Variablen veranlassen	
,ae97	4c 28 af	jmp af28	Variablenwert holen und in FAC #1 bringen	
,ae9a	c9 ff	>cmp #ff	folgt das Token für π ?	
,ae9c	d0 0f	bne aead	nein (Z=0): π -Sonderbehandlung überspringen	

; Sonderbehandlung: π in numerischen Wert umwandeln

,ae9e	a9 a8	lda #a8 <(\$aea8)	LB von \$aea8 (Adresse der Konstanten π) laden
,aea0	a0 ae	ldy #ae >(\$aea8)	HB von \$aea8 (Adresse der Konstanten π) laden
,aea2	20 a2 bb	jsr bba2 "movmf"	Konstante ab \$aea8 (π -Wert) in FAC holen
,aea5	4c 73 00	jmp 0073 "chrget"	CHRGET-Zeiger auf nächstes Zeichen stellen und Ende

; Fließkomma-Konstante π im MLFPT-Format

(wird bei \$ae9e-\$aea2 zur Auswertung von π berücksichtigt)

:aea8 82 49 0f da al MFLPT-Format von 3.14159265 (π)

```

,aead c9 2e    →cmp #2e      folgt Dezimalpunkt "." (ASCII-Code $2e)?
,aeaf f0 de    →beq ae8f      ja (Z=1): ASCII-Codes als Zahl interpretieren und in FAC #1 holen
,aeb1 c9 ab     cmp #ab       folgt das Token für "-"?
,aeb3 f0 58    →beq af0d      ja (Z=1): Vorzeichenwechsel auslösen und Auswertung fortführen
,aeb5 c9 aa     cmp #aa       folgt das Token für "+"?
,aeb7 f0 d1    →beq ae8a      ja (Z=1): normale Auswertung von Parametern, "+" vor Ausdruck ist bedeutungslos
,aeb9 c9 22     cmp #22       folgt das Anführungszeichen zur Einleitung eines Strings?
,aebb d0 0f    →bne aecc      nein (Z=0): Anführungszeichen-Sonderbehandlung überspringen

```

; Sonderbehandlung: Anführungszeichen als String-Anfangsmarkierung

```

,aebd a5 7a    lda 7a        LB des CHRGET-Zeigers holen
,aebf a4 7b    ldy 7b        HB des CHRGET-Zeigers holen
,aec1 69 00    adc #00       hier wird 1 addiert (siehe $aebb!), da C hier immer den Wert 1 hat
,aec3 90 01    bcc aec6      kein Übertrag bei Erhöhung um 1 (C=0): HB nicht erhöhen
,aec5 c8       iny          HB um 1 erhöhen (Übertrag berücksichtigen)
,aec6 20 87    b4→jsr b487 "strlit" String ab in A/Y enthaltener Adresse auswerten
,aec9 4c e2    b7 jmp b7e2    Hilfsroutine zur Erhöhung des CHRGET-Zeigers um Stringlänge anspringen

```

} CHRGET-Zeiger
auslesen und
um 1 erhöhen,
weil String
beginnt

```

,aecc c9 a8    →cmp #a8      Vergleich mit Token für NOT-Operator
,aece d0 13    →bne aee3     Vergleich negativ (Z=0): NOT-Sonderbehandlung überspringen

```

; Sonderbehandlung für NOT-Operator bei Auswertung

```

,aed0 a0 18    ldy #18       Offset für NOT-Priorität in Tabelle ab $a080 laden
,aed2 d0 3b    →bne af0f "jmp" Entfernung der Rücksprungadresse vom Stapel und Ausführung des Operators

```

; Routine zur Basic-Funktion NOT (Token: \$a8)

```

,aed4 20 bf b1 jsr blbf      FAC-Inhalt in vorzeichenbehafteten 2-Byte-Integerwert umwandeln
,aed7 a5 65    lda 65        LB auslesen
,aed9 49 ff    eor #ff %11111111 und alle Bits invertieren (NOT-Funktion!)
,aedb a8       tay          Ergebnis nach Y als LB merken
,aedc a5 64    lda 64        HB auslesen
,aede 49 ff    eor #ff %11111111 und alle Bits invertieren (NOT-Funktion!)

```

} LB
und
HB
einzeln
invertieren

```
,aee0 4c 91 b3 jmp b391 "intfac" Ergebnis aus Y (LB) und A (HB) in FAC zurückschreiben
```

```
; Fortsetzung der EVAL-Routine
```

```
,aee3 c9 a5 -> cmp #a5      Vergleich mit Token für FN
,aee5 d0 03   bne aeea      keine Übereinstimmung (Z=0): FN-Sonderbehandlung überspringen
,aee7 4c f4 b3 jmp b3f4      FN-Sonderbehandlung anspringen (benutzerdefinierte Funktion ausführen)
```

```
,aeea c9 b4   -> cmp #b4      Vergleich mit Token für SGN (niedrigstes Token einer Basic-Funktion)
,aeec 90 03   bcc aef1      kleiner, also kein Token (C=0): Klammerausdruck holen
,aeee 4c a7 af jmp afa7      Auswertung einer Stringfunktion anspringen
```

```
; BRCEVL (Fortsetzung der EVAL-Routine): in Klammern stehenden Ausdruck auswerten (Vorbereitung für Funktionsauswertung)
```

```
,aef1 20 fa ae -> jsr aefa "chkbro" Prüfroutine für "(" (offene Klammer, ASCII-Code $28)
,aef4 20 9e ad jsr ad9e "frmev1" beliebigen Ausdruck holen (rekursives Element, siehe Fließtext!)
                                im Speicher folgt die Prüfroutine für ")" (geschlossene Klammer, ASCII-Code $29)
```

```
; Syntax-Prüfeinsprünge für bestimmte Zeichen:
```

```
$aef7: ")" (CHKBCL)
$aefa: "(" (CHKBRO)
$aefd: ",", (CHKCOM)
$aeff: beliebiges Zeichen (CHKBYT)
```

Wenn das gewünschte Zeichen nicht an der Position der CHRGET-Zeiger befindlich ist, ergibt dies einen SYNTAX ERROR.

```
,aef7 a9 29   lda #29      ASCII-Code von ")" als Prüfbyte laden (CHKBCL)
,aef9 2c a9 28 "bit" lda #28 ASCII-Code von "(" als Prüfbyte laden (CHKBRO)
,aefc 2c a9 2c "bit" lda #2c ASCII-Code von "," als Prüfbyte laden (CHKCOM)
```

```
; CHKBYT-Routine: Prüfung auf beliebiges Zeichen (Token im Akku übergeben!);
```

Aufruf auch von \$aa8d, \$ab8a, \$abaa, \$abc8, \$b3cb und \$b3e3

```
,aeff a0 00   ldy #00      Offset 0 von CHRGET-Zeiger aus laden
,af01 d1 7a   cmp (7a),y    Vergleich: Akku mit Byte an CHRGET-Zeiger-Position
,af03 d0 03   bne af08 "syterr" keine Übereinstimmung (Z=0): SYNTAX ERROR auslösen
,af05 4c 73 00 jmp 0073 "chrget" CHRGET-Zeiger auf nächste Position stellen und ordnungsgemäßer Rücksprung
```


; SYNERR: Einsprung für SYNTAX ERROR;

Aufruf von \$af03 über BNE, von \$a80b, \$a8e8, \$ab5f, \$ae30, \$b09c, \$b138, \$b446 und \$e216 über JMP

<pre>,af08 a2 0b >ldx #0b ,af0a 4c 37 a4 jmp a437 "error"</pre>	<pre>Fehlernummer für SYNTAX laden und zu Fehlereinsprung springen</pre>	}	<pre>SYNTAX ERROR über Fehlereinsprung erzeugen</pre>
---	--	---	---

; Sonderbehandlung für Vorzeichenwechsel (Einsprung bei \$af0d) oder andere Funktion wie NOT (Einsprung bei \$af0f, im Y-Register hat der Offset für das Prioritätsflag zu stehen)

<pre>,af0d a0 15 ldy #15 ,af0f 68 pla ,af10 68 pla ,af11 4c fa ad jmp adfa</pre>	<pre>Offset des Prioritätsflags für Vorzeichenwechsel laden LB der Rücksprungadresse vom Stapel holen HB der Rücksprungadresse vom Stapel holen zur Auswertung anhand der Prioritätsflag-Offsets springen; Funktion wie Vorzeichenwechsel oder NOT wird jetzt ausgeführt</pre>	}	<pre>Rücksprungadresse am Stapel tilgen</pre>
--	--	---	---

; Hilfsroutine zur Prüfung, ob die geforderte Basic-Variable im ROM liegt

(in \$64/\$65 steht die Variablenadresse; nach "jsr \$af14" ist C=0, wenn Variable im ROM / C=1, wenn nicht)

<pre>,af14 38 sec ,af15 a5 64 lda 64 ,af17 e9 00 sbc #00 <(\$a000) ,af19 a5 65 lda 65 ,af1b e9 a0 sbc #a0 >(\$a000) ,af1d 90 08 bcc af27 ,af1f a9 a2 lda #a2 <(\$e3a2) ,af21 e5 64 sbc 64 ,af23 a9 e3 lda #e3 >(\$e3a2) ,af25 e5 65 sbc 65 ,af27 60 rts</pre>	<pre>Carry vor Subtraktion setzen LB der Variablenadresse holen zwecks Vergleich 0 (LB von \$a000) abziehen HB der Variablenadresse holen zwecks Vergleich \$a0 (HB von \$a000) abziehen Subtraktionsübertrag, also Variable im ROM (C=0): RTS bei gelöschtem Carry-Flag LB von \$e3a2 (höchste Basic-ROM-Adresse) holen davon zwecks Vergleich LB der Variablenadresse abziehen HB von \$e3a2 (höchste Basic-ROM-Adresse) holen davon zwecks Vergleich HB der Variablenadresse abziehen Rücksprung: C=0, wenn Adresse im ROM, sonst C=1</pre>	}	<pre>\$a000 (niedrigste ROM-Adresse) zwecks Vergleich von Variablenadresse subtrahieren \$e3a2 (höchste Basic-ROM-Adresse) zwecks Vergleich von Variablenadresse subtrahieren</pre>
---	--	---	---

; GETVAR-Routine: Variable holen (Aufruf von \$ae97 aus der EVAL-Routine)

<pre>,af28 20 8b b0 jsr b08b "fndvar" ,af2b 85 64 sta 64 ,af2d 84 65 sty 65 ,af2f a6 45 ldx 45 ,af31 a4 46 ldy 46 ,af33 a5 0d lda 0d</pre>	<pre>Variableneintrag im Speicher suchen LB der Adresse des Eintrags setzen HB der Adresse des Eintrags setzen Byte #1 des Variablennamens holen Byte #2 des Variablennamens holen Datentyp-Flag (String/numerisch) zwecks Test auslesen</pre>	}	<pre>Zeiger \$64/\$65 mit Adresse des Variableneintrags belegen Variablennamen nach X (Byte#1) und Y (Byte #2) holen</pre>
--	--	---	--

```
,af35 f0-26 beq af5d    numerische Variable (Z=1): Sonderbehandlung für numerische Variable anspringen
,af37 a9 00 lda #00     Löschwert laden
,af39 85 70 sta 70      Rundungsbyte des FAC #1 löschen
,af3b 20 14 af jsr af14 "chkrom" Prüfroutine, ob Variable laut "fndvar" (s. $af28) im ROM liegt, aufrufen
,af3e 90 1c bcc af5c    nein, also keine besondere Variable (C=0): RTS anspringen
```

; Sonderbehandlung: Variableneintrag liegt laut "fndvar" (s. \$af28) im ROM, es handelt sich also um einen Spezialfall

```
,af40 e0 54 cpx #54     Byte #1 des Variablennamens (s. $af2f) mit T vergleichen
,af42 d0 18 bne af5c    keine Übereinstimmung (Z=0): Variable ist nicht TI$, also RTS anspringen
,af44 c0 c9 cpy #c9     Byte #2 des Variablennamens (s. $af31) mit I vergleichen
                        ($c9 = "i"; mit gesetztem b7 als Kennzeichnung für Stringvariable)
,af46 d0 14 bne af5c    keine Übereinstimmung (Z=0): Variable ist nicht TI$, also RTS anspringen
,af48 20 84 af jsr af84  Unterroutine zum Auslesen der Systemuhr (TI/TI$) aufrufen
,af4b 84 5e sty 5e      Hilfszeiger $5e löschen (Y enthält seit $af48 den Wert 0; s. $af8d!)
,af4d 88 dey "ldy #ff"  Y mit $ff laden (von 0 auf $ff dekrementieren)
,af4e 84 71 sty 71      und als Flag in $71 merken
,af50 a0 06 ldy #06     Länge des TI$-Strings laden } TI$-Länge auf 6
,af52 84 5d sty 5d      und in Übergaberegister für Stringlänge schreiben } festlegen
,af54 a0 24 ldy #24     Vorbelegungswert für Einstieg in Fließkomma-Routine laden (Offset für Zeitkonstanten
                        bei FLPSTR-Umwandlung in der Tabelle ab $bfl6)
,af56 20 68 be jsr be68  durch $af48 ausgelesene Systemuhr in Fließkomma-Format umwandeln
,af59 4c 6f b4 jmp b46f  in Routine zur Basic-Funktion STR$ einsteigen, so daß TI$ entsteht
```

```
-----
,af5c 60 rts           Rücksprung von Routine
-----
```

; GETVAR-Sonderfall: numerische Variable auswerten

```
,af5d 24-0e bit 0e      Datentyp-Flag (Fließkomma/Integer) auslesen
,af5f 10 0d bpl af6e    Fließkomma-Variable (N=0): Sonderbehandlung für Fließkomma anspringen
```

; GETVAR-Sonderfall: Integervariable auswerten

```
,af61 a0 00 ldy #00     Offset mit 0 initialisieren (auf LB des Integerwertes stellen)
,af63 b1 64 lda (64),y   LB des Integerwertes aus Variablenspeicher holen
,af65 aa tax       und vorerst in X-Register merken
,af66 c8 iny "ldy #01"  Offset von 0 auf 1 erhöhen (auf HB des Integerwertes stellen)
,af67 b1 64 lda (64),y   HB des Integerwertes aus Variablenspeicher holen
,af69 a8 tay       und im Y-Register aufnehmen
,af6a 8a txa        LB in Akku holen (s. $af65)
```

} Integer-
Variableninhalt
nach A/Y
auslesen

```
,af6b 4c 91 b3 | jmp b391 "intfac" Integerwert aus A/Y in FAC #1 bringen
```

```
-----
```

```
; GETVAR-Sonderfall: Fließkomma-Variable auswerten
```

```
,af6e 20 14 af | jsr afl4 "chkrom" Prüfroutine, ob Variable laut "fndvar" (s. $af28) im ROM liegt, aufrufen
,af71 90 2d | bcc afa0      nein, normale Variable (C=0): Fließkommawert aus Variablenspeicher in FAC #1 holen
,af73 e0 54 | cpx #54      Byte #1 mit T vergleichen
,af75 d0 1b | bne af92     keine Übereinstimmung, also nicht TI als Variablenname (Z=0): auf ST prüfen
,af77 c0 49 | cpy #49      Byte #2 mit I vergleichen
,af79 d0 25 | bne afa0     keine Übereinstimmung, also nicht TI als Variablenname (Z=0): Fließkommawert aus
               Variablenspeicher in FAC #1 holen
```

```
; Sonderbehandlung für Variable TI
```

```
,af7b 20 84 af | jsr af84      Unteroutine zum Auslesen der Systemuhr aufrufen
,af7e 98      | tya "lda #00" Akku mit 0 belegen, da Y seit $af8d (s. $af7b!) mit 0 belegt ist
,af7f a2 a0 | ldx #a0      Exponent für TI-Ergebnis laden
,af81 4c 4f bc | jmp bc4f "setfac" TI-Wert in FAC #1 bringen, da jetzt alle Vorbereitungen abgeschlossen sind
```

```
-----
```

```
; Unteroutine zum Auslesen der Systemuhr (Aufruf von $af48 bei Auslesen von TI$ und $af7b bei Auslesen von TI)
```

```
,af84 20 de ff | jsr ffde "rdtime" Kernall-Einsprung für Auslesen der Systemuhr aufrufen
,af87 86 64 | stx 64      mittelwertiges Byte ablegen
,af89 84 63 | sty 63      niederwertigstes Byte ablegen
,af8b 85 65 | sta 65      höchstwertiges Byte ablegen
,af8d a0 00 | ldy #00     Inhalt für erste Mantisse laden
,af8f 84 62 | sty 62      erstes Mantissenbyte löschen
,af91 60      | rts         Rücksprung von Routine
```

```
-----
```

```
; GETVAR-Routine (Fortsetzung von $af75): Prüfung auf ST als Variablenname
```

```
,af92 e0 53 | cpx #53      Byte #1 mit S vergleichen
,af94 d0 0a | bne afa0     keine Übereinstimmung (Z=0): normale Variable, also Fließkommawert aus
               Variablenspeicher in FAC #1 bringen
,af96 c0 54 | cpy #54      Byte #2 mit T vergleichen
,af98 d0 06 | bne afa0     keine Übereinstimmung (Z=0): normale Variable, also Fließkommawert aus
               Variablenspeicher in FAC #1 bringen
```

```
,afa9a 20 b7 ff jsr ffb7 "rdstat" Status-Byte des Betriebssystems über Kern- Einsprung auslesen
,afa9d 4c 3c bc jmp bc3c und diesen Byte-Wert in den FAC #1 als Fließkomma-Zahl bringen
```

; Einsprung: Inhalt einer Fließkomma-Variablen (ab in \$64/\$65 enthaltener Adresse befindlich) in FAC #1 holen;
wird über Verzweigungsbeefehle von \$af71 und \$af79 aufgerufen

```
,afa0 a5 64 → lda 64 LB der Adresse des MFLPT-Wertes holen
,afa2 a4 65 ldy 65 HB der Adresse des MFLPT-Wertes holen
,afa4 4c a2 bb jmp bba2 "movmf" MFLPT-Wert (in diesem Fall Variableninhalt) in FAC #1 bringen
```

} Adresse der
} aktuellen Variablen
} an MOVMF übergeben

; Auswertung einer Funktion während der Ausführung von FREMVL
(Funktionscode muß sich im Akku befinden; Aufruf von \$aeae aus EVAL-Routine)

```
,afa7 0a asl Funktionscode mit 2 multiplizieren; b7 wird dabei entfernt
,afa8 48 pha und Ergebnis auf den Stapel legen
,afa9 aa tax außerdem im X-Register merken
,afaa 20 73 00 jsr 0073 "chrget" nächstes Zeichen aus dem Basic-Text holen
,afad e0 8f cpx #8f Vergleich mit Multiplikationsergebnis (s. $afa7), um Art der Funktion
("String" oder "numerisch") zu ermitteln
,afaf 90 20 bcc afd1 numerische Funktion (C=0): Sonderbehandlung anspringen
```

; Sonderbehandlung: Auswertung einer Stringfunktion

```
,afb1 20 fa ae jsr aefa "chkbro" auf "(" prüfen (syntaktisches Erfordernis)
,afb4 20 9e ad jsr ad9e "frmev1" Auswertung eines beliebigen Ausdrucks
,afb7 20 fd ae jsr aefd "chkcom" auf "," prüfen (syntaktisches Erfordernis)
,afba 20 8f ad jsr ad8f "chkstr" sicherstellen, daß bei $afb4 ausgewerteter Ausdruck ein String war
,afbd 68 pla bei $afa8 gerettetes Multiplikationsergebnis wieder vom Stapel holen
,afbe aa tax und auch ins X-Register bringen
,afbf a5 65 lda 65 HB der Adresse des Variableneintrags holen
,afc1 48 pha und auf den Stapel legen
,afc2 a5 64 lda 64 LB der Adresse des Variableneintrags holen
,afc4 48 pha und auf den Stapel legen
,afc5 8a txa bei $afbe im X-Register abgelegtes Multiplikationsergebnis in Akku holen
,afc6 48 pha und dann auf den Stapel legen
,afc7 20 9e b7 jsr b79e "getbyt" Bytewert (0-255) aus Basic-Text ins X-Register holen
,afca 68 pla bei $afc6 gemerktes Multiplikationsergebnis vom Stapel holen
,afcb a8 tay und ins Y-Register als Offset bringen
,afcc 8a txa bei $afc7 eingelesenen Bytewert in den Akku bringen
```

} Adresse des
} Variableneintrags
} auf den Stapel
} legen


```
,afcd 48 pha und auf dem Stapel merken
,afce 4c d6 af jmp afd6 Funktion ausführen lassen
-----

; Sonderbehandlung: Auswertung einer numerischen Funktion

,afdl 20 fl ae >jsr aefl "brcevl" in Klammern stehenden Ausdruck auswerten
,afd4 68 pla bei $afa8 gerettetes Multiplikationsergebnis wieder vom Stapel holen
,afd5 a8 tay und als Offset ins Y-Register holen

; Ausführung der Routine zu einer beliebigen Funktion, deren Funktionscode in Y steht

,afd6 b9 ea 9f lda 9fea,y LB aus Tabelle ab $a052 holen
,afd9 85 55 sta 55 und als LB des Sprungziels setzen
,afdb b9 eb 9f lda 9feb,y HB aus Tabelle ab $a052 holen
,afde 85 56 sta 56 und als HB des Sprungziels setzen
,afe0 20 54 00 jsr 0054 bei $0054 steht JMP-Opcode; also wird die vorher ermittelte Adresse angesprungen
,afe3 4c 8d ad jmp ad8d "chknum" sicherstellen, daß der letzte Parameter numerisch war
-----

; Routine zur Basic-Funktion OR (Token: $b0)

,afe6 a0 ff ldy #ff %11111111 Flag für OR-Funktion laden

; Routine zur Basic-Funktion AND (Token: $af)

,afe8 2c a0 00 bit "ldy #00" Flag für AND-Funktion laden
,afeb 84 0b sty 0b Flag für AND oder OR in $0b merken; dient später als Bitmaske für Verknüpfung
,afed 20 bf bl jsr blbf "facint" FAC in 2-Byte-Integerformat umwandeln
,aff0 a5 64 lda 64 LB des Parameters vor AND/OR holen
,aff2 45 0b eor 0b mit Bitmaske verknüpfen
,aff4 85 07 sta 07 und in LB von $07/$08 merken
,aff6 a5 65 lda 65 HB des Parameters vor AND/OR holen
,aff8 45 0b eor 0b mit Bitmaske verknüpfen
,affa 85 08 sta 08 und in HB von $07/$08 merken
,affc 20 fc bb jsr bbfc "movaf" FAC #2 (ARG) in FAC #1 bringen
,afff 20 bf bl jsr blbf "facint" FAC in Integerformat wandeln
,b002 a5 65 lda 65 HB des zweiten Parameters holen
,b004 45 0b eor 0b mit Bitmaske verknüpfen (keine Veränderung bei AND)
,b006 25 08 and 08 AND-Verknüpfung mit HB des Zwischenwertes in $07/$08
,b008 45 0b eor 0b mit Bitmaske verknüpfen (keine Veränderung bei AND)
```

Modifikation des
JMP-Befehls in \$0054-\$0056,
indem er mit Adresse der
Funktionsroutine belegt wird

Bei AND wird der erste Parameter mit
0 verknüpft (bleibt also unverändert),
bei OR wird das Komplement des ersten
Parameters ermittelt.
Das Ergebnis kommt nach \$07/\$08 und
wird später berücksichtigt.

Im
Fließtext
steht
eine umfangreiche
Beschreibung

,b00a	a8	tay	Ergebnis in Y-Register merken	} der relativ undurchsichtigen Arbeitsweise dieser Befehle.
,b00b	a5 64	lda 64	LB des zweiten Parameters holen	
,b00d	45 0b	eor 0b	mit Bitmaske verknüpfen (keine Veränderung bei AND)	
,b00f	25 07	and 07	AND-Verknüpfung mit LB des Zwischenwertes in \$07/\$08	
,b011	45 0b	eor 0b	mit Bitmaske verknüpfen (keine Veränderung bei AND)	
,b013	4c 91 b3	jmp b391 "intfac"	Ergebnis in FAC #1 schreiben	

; Vergleich zweier Variablen (Behandlung von Vergleichsoperand "=")

,b016	20 90 ad	jsr ad90 "chktyp"	Prüfung auf Datentyp des ersten Vergleichswertes	
,b019	b0 13	bcs b02e	String (C=1): Stringvergleichsroutine anspringen	
,b01b	a5 6e	lda 6e	Vorzeichenbyte von FAC #2 (ARG) holen	} Umwandlung des FAC #2 (ARG) ins MFLPT-Format, um MFLPT-Vergleich zu ermöglichen Vergleich des FAC #1 mit dem ins MFLPT-Format konvertierten FAC #2 (ARG)
,b01d	09 7f	ora #7f %01111111	alle Bits bis auf b7 setzen	
,b01f	25 6a	and 6a	mit erstem Mantissenbyte des FAC #2 (ARG) ANDen	
,b021	85 6a	sta 6a	und Ergebnis in erstes Mantissenbyte des FAC #2	
,b023	a9 69	lda #69 <(\$0069)	LB der Adresse des FAC #2 (ARG) laden	
,b025	a0 00	ldy #00 >(\$0069)	HB der Adresse des FAC #2 (ARG) laden	
,b027	20 5b bc	jsr bc5b "cmpfac"	FAC #1 mit MFLPT-Zahl vergleichen	
,b02a	aa	tax	Ergebnis des Vergleichs ins X-Register bringen	
,b02b	4c 61 b0	jmp b061	und dieses in den FAC #1 schreiben	

; Routine zum Vergleich zweier Strings

,b02e	a9 00	lda #00	Flag für "numerisch" laden
,b030	85 0d	sta 0d	und in Datentyp-Flag (String/numerisch) schreiben
,b032	c6 4d	dec 4d	Bitmuster für Vergleichsoperatoren dekrementieren
,b034	20 a6 b6	jsr b6a6 "frestr"	ersten zu vergleichenden String auswerten
,b037	85 61	sta 61	Stringlänge in \$61 merken
,b039	86 62	stx 62	LB der Adresse des Strings im Speicher in LB von \$62/\$63 schreiben
,b03b	84 63	sty 63	HB der Adresse des Strings im Speicher in HB von \$62/\$63 schreiben
,b03d	a5 6c	lda 6c	LB des Eintrags des zweiten zu vergleichenden Strings holen
,b03f	a4 6d	ldy 6d	HB des Eintrags des zweiten zu vergleichenden Strings holen
,b041	20 aa b6	jsr b6aa	in FRESTR-Routine einsteigen, damit Auswertung des Strings erfolgt
,b044	86 6c	stx 6c	LB der Adresse des zweiten Strings in LB von \$6c/\$6d schreiben
,b046	84 6d	sty 6d	HB der Adresse des zweiten Strings in HB von \$6c/\$6d schreiben
,b048	aa	tax	Stringlänge ins X-Register bringen
,b049	38	sec	Carry vor Subtraktion setzen
,b04a	e5 61	sbc 61	Länge des ersten Strings zwecks Vergleich von erster Stringlänge abziehen
,b04c	f0 08	beq b056	beide Strings haben dieselbe Länge (Z=1): mit \$00 im Akku weiterarbeiten

```

,b04e a9 01    lda #01 %00000001 Flag für "erster String länger" laden
,b050 90 04    bcc b056          erster String länger (C=0): mit diesem Flag weiterarbeiten
,b052 a6 61    ldx 61           Länge des ersten Strings als Anzahl der zu vergleichenden Zeichen laden
,b054 a9 ff    lda #ff %11111111 Flag für "zweiter String länger" laden
,b056 85 66    >sta 66          Flag für "Strings gleich lang" ($00), "String #1 länger" ($01) oder "String #2
                                länger" ($ff) setzen
,b058 a0 ff    ld y #ff        Offset für Vergleichsbyte vorbereiten
,b05a e8       inx             Dekrementierzähler für Anzahl der zu vergleichenden Zeichen vorbereiten
,b05b c8       >iny            Offset für Vergleichsbyte erhöhen
,b05c ca       dex             Dekrementierzähler für Anzahl der zu vergleichenden Zeichen herunterzählen
,b05d d0 07    bne b066        noch nicht heruntergezählt (Z=0): Vergleich der beiden Bytes
,b05f a6 66    ldx 66          Flag für Stringlängenverhältnis holen
,b061 30 0f    bmi b072        String #2 ist länger (N=1): Vergleich mit $ff im X-Register abbrechen
,b063 18       clc             Carry löschen, um Ergebnis bei $b074 nicht zu verfälschen
,b064 90 0c    bcc b072 "jmp"  Vergleich mit $00 oder $01 im X-Register abbrechen
-----
,b066 b1 6c    >lda (6c),y      Byte aus String #2 holen
,b068 d1 62    cmp (62),y      und mit Byte aus String #1 vergleichen
,b06a f0 ef    beq b05b        Übereinstimmung (Z=1): Vergleich bei nächstem Byte fortsetzen
,b06c a2 ff    ldx #ff         Flag für "String #1 < String #2" laden
,b06e b0 02    bcs b072        String #1 hat kleineren ASCII-Code als String #2 (C=1): Vergleich mit $ff im
                                X-Register abbrechen
,b070 a2 01    ldx #01         Flag für "String #1 > String #2" laden
,b072 e8       >inx            Flag für Vergleichsergebnis erhöhen
,b073 8a       txa             und in den Akku bringen
,b074 2a       rol             eventuell gesetztes Carry-Flag in Ergebnis miteinbeziehen
,b075 25 12    and 12          Verknüpfung des Vergleichsergebnisses mit den gewünschten Vergleichsoperatoren
,b077 f0 02    beq b07b        tatsächliches Stringverhältnis stimmt nicht mit gewünschtem überein (Z=1):
                                Routine mit $00 im Akku als Wahrheitsflag für "nicht wahr" verlassen
,b079 a9 ff    lda #ff         Wahrheitsflag für "wahr" laden
,b07b 4c 3c bc >jmp bc3c       Wahrheitsflag (0 oder $ff) in FAC #1 bringen (numerisches Ergebnis des Vergleichs)
-----
; Anfang (nicht Einsprung!) der Routine zum Basic-Befehl DIM

,b07e 20 fd ae >jsr aefd "chkcom" auf ", " prüfen (Abgrenzung zwischen DIM-Parametern)

; Einsprungadresse zum Basic-Befehl DIM (Token: $86)

,b081 aa       tax             erstes Zeichen des DIM-Parameters ins X-Register bringen
,b082 20 90 b0 jsr b090        Dimensionierungsroutine aufrufen

```

```
,b085 20 79 00 | jsr 0079 "chrgot" letztes Zeichen noch einmal in den Akku holen und dabei die CPU-Flags berechnen
,b088 d0 f4 | bne b07e keine Endmarkierung (Z=0): weiter in DIM-Schleife
,b08a 60 | rts Rücksprung von Routine zum DIM-Befehl
```

; FNDVAR: Variable holen oder Array dimensionieren (\$b08b: Einsprung für "nicht dimensionieren"; \$b090 bei X-Register mit anderem Inhalt als \$00: Einsprung für "dimensionieren")

```
,b08b a2 00 | ldx #00 Flag für "keine Dimensionierung" laden
,b08d 20 79 00 | jsr 0079 "chrgot" letztes Zeichen (= erstes Zeichen des Variablennamen) holen
,b090 86 0c | stx 0c Flag für Standardvariable oder Arraydimensionierung nach Wunsch setzen
,b092 85 45 | sta 45 erstes Byte des Variablennamen setzen
,b094 20 79 00 | jsr 0079 "chrgot" letztes Zeichen (= erstes Zeichen des Variablennamen) holen
,b097 20 13 b1 | jsr b113 "chkltr" Prüfroutine, ob es sich um einen Buchstaben handelt, aufrufen
,b09a b0 03 | bcs b09f Buchstabe (C=1): erlaubter Variablenname, also keine Fehlermeldung erzeugen
,b09c 4c 08 af | jmp af08 "synerr" SYNTAX ERROR zur Zurückweisung eines Variablennamen, der nicht mit Buchstabe beginnt

-----
,b09f a2 00 | ldx #00 Initialisierungswert für Datentyp-Flags laden ($00 = "numerisch" und "Fließkomma")
,b0a1 86 0d | stx 0d und in Datentyp-Flag (String/numerisch) schreiben
,b0a3 86 0e | stx 0e ebenso in Datentyp-Flag (Fließkomma/Integer) schreiben
,b0a5 20 73 00 | jsr 0073 "chrget" nächstes Zeichen bearbeiten
,b0a8 90 05 | bcc b0af Ziffer (C=0): kein Fehler (Ziffer als zweites Zeichen eines Variablennamen zulässig)
,b0aa 20 13 b1 | jsr b113 "chkltr" Prüfroutine, ob es sich um einen Buchstaben handelt, aufrufen
,b0ad 90 0b | bcc b0ba kein Buchstabe (C=0): Test auf Variablentyp-Markierung ("$", "%")
,b0af aa | tax zweites Zeichen des Variablennamen in X-Register merken
,b0b0 20 73 00 | jsr 0073 "chrget" nächstes Zeichen aus Basic-Text holen
,b0b3 90 fb | bcc b0b0 Ziffer (C=0): kein Fehler, Ziffer als weiteres Zeichen eines Variablennamen zulässig
,b0b5 20 13 b1 | jsr b113 "chkltr" Prüfroutine, ob es sich um einen Buchstaben handelt, aufrufen
,b0b8 b0 f6 | bcs b0b0 Buchstabe (C=1): weiter auf Ende des Variablennamen warten
,b0ba c9 24 | cmp #24 letztes Zeichen im Variablennamen mit "$" vergleichen
,b0bc d0 06 | bne b0c4 keine Übereinstimmung (Z=0): Sonderbehandlung für Entdeckung einer Stringvariablen überspringen
,b0be a9 ff | lda #ff Flag für "String" laden
,b0c0 85 0d | sta 0d und in Datentyp-Flag (String/numerisch) schreiben
,b0c2 d0 10 | bne b0d4 "jmp" b7 in Byte #2 des Variablennamen setzen und weitere Behandlung

-----
,b0c4 c9 25 | cmp #25 letztes Zeichen im Variablennamen mit "%" vergleichen
,b0c6 d0 13 | bne b0db keine Übereinstimmung (Z=0): Sonderbehandlung für Entdeckung einer Integervariablen überspringen
,b0c8 a5 10 | lda 10 Flag für FN-Benutzerfunktion zwecks Test auslesen
,b0ca d0 d0 | bne b09c Flag für "Integer gesperrt" ist gesetzt (Z=0): SYNTAX ERROR erzeugen
```


,b0cc	a9 80	lda #80 %10000000	Flag für "Integerzahl" laden (dient bei \$b0d0 auch als Bitmaske)	} jeweils Bit 7 in beiden Bytes des Variablennamen setzen
,b0ce	85 0e	sta 0e	und in Datentyp-Flag (Integer/Fließkomma) schreiben	
,b0d0	05 45	ora 45	b7 (s. \$b0cc) in Byte #1 des Variablennamen setzen	
,b0d2	85 45	sta 45	und Ergebnis in Byte #1 des Variablennamen schreiben	
,b0d4	8a	→txa	bei \$b0af gemerktes Byte #2 des Variablennamen holen	
,b0d5	09 80	ora #80 %10000000	auch hier b7 setzen	
,b0d7	aa	tax	und zurück ins X-Register als Byte #2 des Variablennamen	
,b0d8	20 73 00	jsr 0073 "chrget"	nächstes Zeichen aus dem Basic-Text holen	
,b0db	86-46	→stx 46	bei \$b0d7 im X-Register gemerktes Byte #2 des Variablennamen in Hilfsspeicher für Variablennamen schreiben	
,b0dd	38	sec	Carry vor Subtraktion bei \$b0e0 setzen	
,b0de	05 10	ora 10	Verknüpfung des Akku mit dem Flag für "Integer gesperrt"	
,b0e0	e9 28	sbc #28	Subtraktion des ASCII-Codes von "(", um auf Arrayvariable zu testen	
,b0e2	d0 03	bne b0e7	keine Übereinstimmung (Z=0): nicht zur Bearbeitung einer Arrayvariablen springen	
,b0e4	4c d1 b1	jmp b1d1	weiter bei Routine zur Auswertung einer Arrayvariablen	

,b0e7	a0 00	→ldy #00	Flag für "Integer erlaubt" laden und gleichzeitig Offset initialisieren (s. \$b0fd!)	
,b0e9	84 10	sty 10	und in Sperrflag für Integerzahlen schreiben	
,b0eb	a5 2d	lda 2d	LB der Anfangsadresse des Variablenbereichs holen	
,b0ed	a6 2e	ldx 2e	HB der Anfangsadresse des Variablenbereichs holen	
,b0ef	86 60	→stx 60	HB in HB des Hilfszeigers \$5f/\$60 für Variablensuche schreiben	
,b0f1	85 5f	→sta 5f	LB in LB des Hilfszeigers \$5f/\$60 für Variablensuche schreiben	
,b0f3	e4 30	cpx 30	Vergleich des HB mit HB des Zeigers auf den Anfang der Basic-Arrays	
,b0f5	d0 04	bne b0fb	keine Übereinstimmung (Z=0): Suche fortsetzen	
,b0f7	c5 2f	cmp 2f	Vergleich des LB mit LB des Zeigers auf den Anfang der Basic-Arrays	
,b0f9	f0 22	beq b11d	Übereinstimmung (Z=1): nicht im Variablenbereich gefundene Variable anlegen	
,b0fb	a5 45	→lda 45	Byte #1 des Variablennamen holen	
,b0fd	d1 5f	cmp (5f),y	Vergleich mit Byte #1 im aktuell untersuchten Variableneintrag	
,b0ff	d0 08	bne b109	keine Übereinstimmung (Z=0): Vergleich bei nächstem Variableneintrag fortsetzen	
,b101	a5 46	lda 46	Byte #2 des Variablennamen holen	
,b103	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Byte #2 des Variablennamen im aktuell untersuchten Variableneintrag stellen)	
,b104	d1 5f	cmp (5f),y	Vergleich mit Byte #2 im aktuell untersuchten Variableneintrag	
,b106	f0 7d	→beq b185	Übereinstimmung, gesuchte Variable gefunden (Z=1): Zeiger auf erstes Byte des Variableninhalts stellen	
,b108	88	dey "ldy #00"	Offset wieder von 1 auf 0 herunterzählen	
,b109	18	→clc	Carry vor Addition bei \$b10c löschen	
,b10a	a5 5f	lda 5f	LB des Hilfszeigers für untersuchten Variableneintrag stellen	
,b10c	69 07	adc #07	Länge eines Variableneintrags addieren	
,b10e	90 e1	bcc b0f1	kein Additionsübertrag (C=0): HB nicht erhöhen	
,b110	e8	inx	HB (s. \$b0ed) erhöhen, um Übertrag zu berücksichtigen	

,b111 d0 dc bne b0ef "jmp" Suche mit neuem Zeiger (auf nächsten Variableneintrag gestellt) fortsetzen

; CHKLTR-Hilfsroutine: ermittelt, ob im Akkumulator der ASCII-Code eines Buchstaben steht (C=1) oder nicht (C=0);
 Aufruf von \$ae92 (EVAL), \$b097 (FNDVAR), \$b0aa (FNDVAR) und \$b0b5 (FNDVAR)

,b113 c9 41 cmp #41 Vergleich mit ASCII-Code von "a" (kleinster ASCII-Code eines Buchstaben)
 ,b115 90 05 bcc b11c ASCII-Code im Akku ist kleiner (C=0): kein Buchstabe, daher RTS bei gelöschtem Carry
 ,b117 e9 5b sbc #5b ASCII-Code von "z" plus 1 abziehen; Carry ist wegen \$b115 hier immer gesetzt
 ,b119 38 sec Carry vor Subtraktion setzen
 ,b11a e9 a5 sbc #a5 Berichtigung der Subtraktion bei \$b117 (\$5b+\$a5 = \$0100); jetzt werden Flags
 entsprechend dem Vergleichsergebnis gesetzt (C=0: kein Buchstabe; C=1: Buchstabe)
 ,b11c 60 rts Rücksprung von Prüfroutine; Carry enthält das Vergleichsergebnis

; Nicht im Variablenspeicher gefundenen Variableneintrag anlegen

,b11d 68 pla LB der Rücksprungadresse der aufrufenden Routine vom Stapel holen
 ,b11e 48 pha und dorthin wieder zurücklegen, also effektiv keine Stapelveränderung
 ,b11f c9 2a cmp #2a <(\$af2a) wurde dieser Programmteil aus der GETVAR-Routine aufgerufen?
 ,b121 d0 05 bne b128 nein (Z=0): Variable anlegen
 ,b123 a9 13 lda #13 <(\$bf13) LB der Adresse der ROM-Konstanten 0 laden
 ,b125 a0 bf ldy #bf >(\$bf13) HB der Adresse der ROM-Konstanten 0 laden (gleichzeitig Flag für "noch nicht
 angelegte Variable wurde angetroffen")
 ,b127 60 rts Rücksprung von Routine

,b128 a5 45 lda 45 Byte #1 des Variablennamen holen
 ,b12a a4 46 ldy 46 Byte #2 des Variablennamen holen
 ,b12c c9 54 cmp #54 Byte #1 mit "T" vergleichen
 ,b12e d0 0b bne b13b keine Übereinstimmung (Z=0): weiter mit Prüfung auf "ST"
 ,b130 c0 c9 cpy #c9 Byte #2 mit "I" (b7 gesetzt als Flag für Stringvariable) vergleichen
 ,b132 f0 ef beq b123 Übereinstimmung (Z=1): 0 übergeben (auch Flag für "angelegte Variable im ROM")
 ,b134 c0 49 cpy #49 Byte #2 mit "I" (b7 nicht gesetzt) vergleichen
 ,b136 d0 03 bne b13b keine Übereinstimmung (Z=0): weiter mit Prüfung auf "ST"
 ,b138 4c 08 af jmp af08 "synerr" SYNTAX ERROR auslösen, da TI und ST nicht über "TI=" bzw. "ST=" belegt werden dürfen

,b13b c9 53 cmp #53 Byte #1 mit "S" vergleichen
 ,b13d d0 04 bne b143 keine Übereinstimmung (Z=0): Eintrag für neue Variable errichten
 ,b13f c0 54 cpy #54 Byte #2 mit "T" vergleichen
 ,b141 f0 f5 beq b138 Übereinstimmung (Z=1): SYNTAX ERROR auslösen, da "ST=..." unzulässig ist

,b143	a5 2f	↳lda 2f	LB der Anfangsadresse der Basic-Arrays holen	} Anfangsadresse des Array-Bereichs als
,b145	a4 30	ldy 30	HB der Anfangsadresse der Basic-Arrays holen	
,b147	85 5f	sta 5f	LB als LB des Anfangs des Quellbereichs setzen	} Anfangsadresse des Quellbereichs setzen
,b149	84 60	sty 60	HB als HB des Anfangs des Quellbereichs setzen	
,b14b	a5 31	lda 31	LB der Endadresse der Basic-Arrays holen	} Endadresse des Array-Bereichs als
,b14d	a4 32	ldy 32	HB der Endadresse der Basic-Arrays holen	
,b14f	85 5a	sta 5a	LB als LB des Endes des Quellbereichs setzen	} Endadresse des Quellbereichs setzen
,b151	84 5b	sty 5b	HB als HB des Endes des Quellbereichs setzen	
,b153	18	clc	Carry vor Addition löschen	
,b154	69 07	adc #07	Länge eines Variableneintrags addieren, da der Bereich durch den neuen Variableneintrag um dessen Speichermenge größer werden muß	
,b156	90 01	bcc b159	kein Additionsübertrag (C=0): HB nicht erhöhen	
,b158	c8	iny	HB erhöhen, um Übertrag zu berücksichtigen	
,b159	85 58	↳sta 58	LB des Additionsergebnisses als LB der Endadresse des Zielbereichs setzen	
,b15b	84 59	sty 59	HB des Additionsergebnisses als HB der Endadresse des Zielbereichs setzen	
,b15d	20 b8 a3	jsr a3b8	Routine zum Schaffen von ausreichend Platz im Variablenspeicher schaffen (enthält BLTUC-Speicherblockverschiebung nach den ermittelten Parametern)	
,b160	a5 58	lda 58	LB der neuen Anfangsadresse des Array-Bereichs holen	
,b162	a4 59	ldy 59	HB der neuen Anfangsadresse des Array-Bereichs holen	
,b164	c8	iny	erhöhen, da HB in BLTUC-Routine verändert und nicht wiederhergestellt wurde	
,b165	85 2f	sta 2f	LB als LB der Anfangsadresse des Array-Bereichs setzen	
,b167	84 30	sty 30	HB als HB der Anfangsadresse des Array-Bereichs setzen	
,b169	a0 00	ldy #00	Offset initialisieren (auf Byte #1 des Variablennamen im Variableneintrag stellen)	
,b16b	a5 45	lda 45	Byte #1 des Variablennamen holen	
,b16d	91 5f	sta (5f),y	und als Byte #1 des Variablennamen in neu einzurichtenden Variableneintrag schreiben	
,b16f	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Byte #1 des Variablennamen im Eintrag stellen)	
,b170	a5 46	lda 46	Byte #2 des Variablennamen holen	
,b172	91 5f	sta (5f),y	und als Byte #2 des Variablennamen in neu einzurichtenden Variableneintrag schreiben	
,b174	a9 00	lda #00	Initialisierungswert für Variableninhalt laden (MFLPT-Darstellung von 0 besteht nur aus \$00-Bytes)	
,b176	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf Byte #1 des Variableninhalts stellen)	
,b177	91 5f	sta (5f),y	Byte #1 des Variableninhalts mit \$00 initialisieren	
,b179	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf Byte #2 des Variableninhalts stellen)	
,b17a	91 5f	sta (5f),y	Byte #2 des Variableninhalts mit \$00 initialisieren	
,b17c	c8	iny "ldy #04"	Offset von 3 auf 4 erhöhen (auf Byte #3 des Variableninhalts stellen)	
,b17d	91 5f	sta (5f),y	Byte #3 des Variableninhalts mit \$00 initialisieren	
,b17f	c8	iny "ldy #05"	Offset von 4 auf 5 erhöhen (auf Byte #4 des Variableninhalts stellen)	
,b180	91 5f	sta (5f),y	Byte #4 des Variableninhalts mit \$00 initialisieren	
,b182	c8	iny "ldy #06"	Offset von 5 auf 6 erhöhen (auf Byte #5 des Variableninhalts stellen)	
,b183	91 5f	sta (5f),y	Byte #5 des Variableninhalts mit \$00 initialisieren	
,b185	a5 5f	lda 5f	LB der Adresse des neu eingerichteten Variableneintrags holen	

```

,bl87 18      clc          Carry vor Addition löschen
,bl88 69 02    adc #02      2 (Anzahl der Bytes des Variablennamens, also Offset zu Variablenwert) addieren
,bl8a a4 60    ldy 60       HB der Adresse des neu eingerichteten Variableneintrags holen
,bl8c 90 01    bcc bl8f     kein Additionsübertrag bei $bl88 (C=0): HB nicht erhöhen
,bl8e c8       iny         HB erhöhen, um Additionsübertrag zu berücksichtigen
,bl8f 85 47    sta 47       LB des Zeigers auf die MFLPT-Darstellung des aktuellen Variableninhalts setzen
,bl91 84 48    sty 48       HB des Zeigers auf die MFLPT-Darstellung des aktuellen Variableninhalts setzen
,bl93 60       rts         Rücksprung von Routine

```

```

; FIRARY-Hilfsroutine zur Ermittlung der Adresse des ersten Eintrags im aktuellen Array;
  Aufruf von $b253 und $b261

```

```

,bl94 a5 0b    lda 0b       Anzahl der Indizes (= Anzahl der Dimensionen) holen
,bl96 0a       asl          und mit 2 multiplizieren
,bl97 69 05    adc #05      5 addieren; Carry ist davor immer 0, da bei $bl96 der Akku nur Werte von 0—3
                        beinhalten kann (mehr Dimensionen sind nicht möglich)
,bl99 65 5f    adc 5f       LB des Hilfszeigers $5f/$60 (Anfangsadresse des Arrays) addieren
,bl9b a4 60    ldy 60       HB des Hilfszeigers $5f/$60 (Anfangsadresse des Arrays) holen
,bl9d 90 01    bcc bla0     kein Additionsübertrag bei $bl99 (C=0): HB nicht erhöhen
,bl9f c8       iny         HB erhöhen, um Additionsübertrag zu berücksichtigen
,bla0 85 58    sta 58       LB des Ergebnisses in LB des Zeigers $58/$59 übergeben } Ergebnis
,bla2 84 59    sty 59       HB des Ergebnisses in HB des Zeigers $58/$59 übergeben } nach $58/$59
,bla4 60       rts         Rücksprung von Hilfsroutine

```

```

; ROM-Konstante -32768 (Untergrenze für Integerzahlen)

```

```

:bla5 90 80 00 00 00      MFLPT-Darstellung von -32768

```

```

; Einsprung zum Umwandeln des FAC #1 in eine Integerzahl, die in Akku (LB) und Y (HB) zurückgegeben wird;
  dieser Einsprung wird im C 64-ROM nie aufgerufen, er steht nur für Benutzersoftware zur Verfügung!

```

```

,blaa 20 bf b1  jsr blbf     Einsprung in Routine zum Auswerten einer Integerzahl aus dem Basic-Text, wobei aber
                        das Einlesen des Parameters aus dem Basic-Text durch Einsteigen an späterer Stelle
                        übersprungen wird
,blad a5 64    lda 64       LB des Ergebnisses in den Akku holen } Ergebnis der Umwandlung
,blaf a4 65    ldy 65       HB des Ergebnisses in den Akku holen } nach A/Y holen
,blbl 60       rts         Rücksprung von Routine

```


; INTEVL: Auswertung einer im Basic-Text stehenden Integerzahl; Ergebnis wird in \$65/\$64 zurückgegeben;
 Besonderheit dieser Routine: Parameter wird ausgelesen und auf zulässigen Bereich für Integervariablen überprüft

```
,blb2 20 73 00 jsr 0073 "chrget" CHRGET-Zeiger auf nächstes Zeichen stellen
,blb5 20 9e ad jsr ad9e "frmev1" beliebigen Ausdruck auswerten
,blb8 20 8d ad jsr ad8d "chknum" nur numerische Ausdrücke zulassen (es gibt keine "Integer-Strings")
,blbb a5 66 lda 66 Vorzeichenbyte des FAC #1 zwecks Test einlesen
,blbd 30 0d bmi blcc negatives Vorzeichen (N=1): ILLEGAL QUANTITY ERROR auslösen
,blbf a5 61 lda 61 Exponentenbyte von FAC #1 holen
,blcl c9 90 cmp #90 mit Exponent für 32768 (Betragswert der Grenze des Integerbereichs) vergleichen
,blc3 90 09 bcc blce kleinerer Wert im FAC #1 (C=0): FAC #1 in Integerformat umwandeln lassen
,blc5 a9 a5 lda #a5 <($bla5) LB der Adresse der MFLPT-Konstanten -32768 laden
,blc7 a0 b1 ldy #b1 >($bla5) HB der Adresse der MFLPT-Konstanten -32768 laden
,blc9 20 5b bc jsr bc5b "cmpfac" FAC #1 mit MFLPT-Konstante -32768 (Untergrenze des Integerbereichs) vergleichen
,blcc d0 7a bne b248 keine Übereinstimmung (Z=0): ILLEGAL QUANTITY ERROR auslösen, da -32768 die absolute
                          Untergrenze für Integerzahlen ist
,blce 4c 9b bc jmp bc9b "facint" FAC #1 als Integerzahl nach $64/$65 bringen
```

; Sonderbehandlung: Auswertung einer Arrayvariablen (Aufruf von \$b0e4)

```
,bld1 a5 0c lda 0c Flag für Arraydimensionierung holen
,bld3 05 0e ora 0e Datentyp-Flag (Integer/Fließkomma) ebenfalls aufnehmen (Flag besteht nur aus b7)
,bld5 48 pha Array- und Datentyp-Flag als 1 Byte auf den Stapel retten
,bld6 a5 0d lda 0d Datentyp-Flag (String/numerisch) holen
,bld8 48 pha und auf den Stapel legen
,bld9 a0 00 ldy #00 Anzahl der Indizes (= Anzahl der Dimensionen) initialisieren
,bldb 98 >tya Anzahl der Indizes in Akku holen
,bldc 48 pha und dann auf den Stapel retten
,bldd a5 46 lda 46 Byte #2 des Variablennamens holen
,bldf 48 pha auf den Stapel retten
,ble0 a5 45 lda 45 Byte #1 des Variablennamens holen
,ble2 48 pha auf den Stapel retten
,ble3 20 b2 b1 jsr blb2 "intevl" Auswertung einer im Basic-Text stehenden Integerzahl
,ble6 68 pla Byte #1 des Variablennamens (s. $ble0/$ble2) wieder holen
,ble7 85 45 sta 45 und an die ursprüngliche Adresse schreiben
,ble9 68 pla Byte #2 des Variablennamens (s. $bldd/$bldf) wieder holen
,blea 85 46 sta 46 und an die ursprüngliche Adresse schreiben
,blec 68 pla Anzahl der Indizes (s. $bldb/$bldc) wieder vom Stapel holen
,bled a8 tay und ins Y-Register als Hilfsregister bringen
,blee ba tsx Stapelzeiger als Offset ins X-Register holen
```

,b1ef	bd 02 01	lda 0102,x	Datentyp-Flag (Integer/Fließkomma) und Array-Flag direkt aus Stapel auslesen
,b1f2	48	pha	und an jeweils oberste Stapelposition schreiben
,b1f3	bd 01 01	lda 0101,x	Datentyp-Flag (String/numerisch) direkt aus Stapel auslesen
,b1f6	48	pha	und an jeweils oberste Stapelposition schreiben
,b1f7	a5 64	lda 64	HB des eingelesenen Indexwertes holen
,b1f9	9d 02 01	sta 0102,x	und an vorher mit Datentyp-Flag belegte Stapelposition schreiben (s. \$b1ef)
,b1fc	a5 65	lda 65	LB des eingelesenen Indexwertes holen
,b1fe	9d 01 01	sta 0101,x	und an vorher mit Datentyp-Flag belegte Stapelposition schreiben (s. \$b1f3)
,b201	c8	iny	Anzahl der übergebenen Indizes (= Anzahl der Dimensionen) erhöhen
,b202	20 79 00	jsr 0079 "chrgot"	letztes Zeichen hinter ausgewertetem Index wieder in den Akku holen
,b205	c9 2c	cmp #2c	handelte es sich um ein Komma als Abgrenzung zum nächsten Index?
,b207	f0 d2	beq b1db	ja (Z=1): Prozedur der Parameterübergabe mit neuem Index fortsetzen

; hier sind alle Indizes ausgewertet

,b209	84 0b	sty 0b	Hilfsspeicher für Array mit Anzahl der übergebenen Indizes (= Anzahl der Dimensionen) belegen
,b20b	20 f7 ae	jsr aef7 "chkbrc"	auf ")" (Klammer zu) als syntaktisches Erfordernis testen
,b20e	68	pla	Datentyp-Flag (String/numerisch) vom Stapel holen (s. \$b1fc/\$b1fe)
,b20f	85 0d	sta 0d	und in den Speicher schreiben
,b211	68	pla	Datentyp-Flag (Integer/Fließkomma) und Array-Dimensionierungsflag vom Stapel holen
,b212	85 0e	sta 0e	und als Datentyp-Flag in den Speicher schreiben
,b214	29 7f	and #7f %01111111	b7 löschen, da dies nur das Datentyp-Flag ist und nicht ins Array-Flag gehört
,b216	85 0c	sta 0c	und Ergebnis als Array-Dimensionierungsflag in den Speicher schreiben
,b218	a6 2f	ldx 2f	LB des Zeigers auf den Anfang der Basic-Arrays holen
,b21a	a5 30	lda 30	HB des Zeigers auf den Anfang der Basic-Arrays holen
,b21c	86-5f	→ stx 5f	LB als LB in Hilfszeiger für Suche des Variableninhalts schreiben
,b21e	85 60	sta 60	HB als HB in Hilfszeiger für Suche des Variableninhalts schreiben
,b220	c5 32	cmp 32	HB des Hilfszeigers mit HB der Endadresse der Basic-Arrays (+1) vergleichen
,b222	d0 04	bne b228	keine Übereinstimmung (Z=0): Suche fortsetzen
,b224	e4 31	cpx 31	LB des Hilfszeigers mit LB der Endadresse der Basic-Arrays (+1) vergleichen
,b226	f0 39	↓ beq b261	Übereinstimmung (Z=1): Sonderbehandlung für Anlegen einer Array-Variablen anspringen, da gesuchte Array-Variable nicht gefunden wurde
,b228	a0 00	→ ldy #00	Offset zum Hilfszeiger \$5f/\$60 mit 0 initialisieren
,b22a	b1 5f	lda (5f),y	Byte #1 des Variablennamen im aktuell untersuchten Variableneintrag holen
,b22c	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Byte #2 des Variablennamen stellen)
,b22d	c5 45	cmp 45	Vergleich vom jeweils ersten Byte im gesuchten und gefundenen Variablennamen
,b22f	d0 06	bne b237	keine Übereinstimmung (Z=0): Suche fortsetzen, da Suche noch erfolglos
,b231	a5 46	lda 46	Byte #2 des gesuchten Variablennamen holen
,b233	d1 5f	cmp (5f),y	Vergleich mit Byte #2 im gefundenen Variablennamen
,b235	f0 16	beq b24d	Übereinstimmung (Z=1): gesuchte Variable ist gefunden, weitere Auswertung bei \$b24d

,b237	c8		↳iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf LB der Array-Länge stellen)	} Hilfszeiger um Array-Länge vergrößern, damit er auf nächstes Array zum Untersuchen gestellt wird
,b238	b1	5f	lda (5f),y	LB der Array-Länge holen	
,b23a	18		clc	Carry vor Addition löschen	
,b23b	65	5f	adc 5f	LB (s. \$b238) zu LB des Hilfszeigers \$5f/\$60 addieren	
,b23d	aa		tax	Ergebnis-LB im X-Register speichern	
,b23e	c8		iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf HB der Array-Länge stellen)	
,b23f	b1	5f	lda (5f),y	HB der Array-Länge holen	
,b241	65	60	adc 60	und zu HB des Hilfszeigers \$5f/\$60 addieren	
,b243	90	d7	bcc b21c	kein Übertrag bei Addition (C=0): Suche mit neu berechnetem Hilfszeiger fortsetzen	
,b245	a2	12	↳ldx #12	Fehlernummer für BAD SUBSCRIPT laden	
,b247	2c	a2	0e "bit" ldx #0e	Fehlernummer für ILLEGAL QUANTITY laden (\$b248 ist der ILLQUA-Einsprung, der bei \$aa24, \$blcc, \$b798 und \$b9f1 aufgerufen wird)	
,b24a	4c	37	a4 ↳jmp a437 "error"	Fehlereinsprung aufrufen; Fehlernummer wurde bei \$b245 oder \$b248 nach X geladen	

; Variableneintrag in gefundenem Array suchen

,b24d	a2	13	↳ldx #13	Fehlermeldung REDIM'D ARRAY durch Laden des Fehlercodes vorbereiten
,b24f	a5	0c	lda 0c	Array-Dimensionierungsflag holen
,b251	d0	f7	↳bne b24a	nicht 0, obwohl Array bereits vorhanden (Z=0): REDIM'D ARRAY auslösen, da bereits vorhandenes Array dimensioniert werden soll
,b253	20	94	b1 jsr b194 "firary"	Hilfsroutine zur Ermittlung der Adresse des ersten Eintrags im aktuellen Array aufrufen
,b256	a5	0b	lda 0b	Anzahl der Indizes (= Anzahl der Dimensionen) im aktuellen Array holen
,b258	a0	04	ldy #04	Offset auf Anzahl der Dimensionen im Array-Eintrag richten
,b25a	d1	5f	cmp (5f),y	Vergleich der Anzahl der angegebenen Indizes (s. \$b256) mit tatsächlicher Anzahl der Indizes (= Anzahl der Dimensionen)
,b25c	d0	e7	↳bne b245	keine Übereinstimmung (Z=0): BAD SUBSCRIPT auslösen, da Parameter falsch angegeben wurden (Array hat andere Anzahl von Dimensionen, als der gewünschten Array-Variablen Indizes folgen)
,b25e	4c	ea	b2 jmp b2ea	weiter mit Suchroutine für Array-Variable in gefundenem Array

; Array-Variable anlegen

,b261	20	94	b1 jsr b194 "firary"	Ermittlung der Adresse des ersten Eintrags im aktuellen Array
,b264	20	08	a4 jsr a408 "chkfvm"	Variablenspeicher organisieren (wenn nicht genügend vorhanden: OUT OF MEMORY)
,b267	a0	00	ldy #00	Initialisierungswert für Offset (s. \$b26f) und Hilfsspeicher \$72 laden
,b269	84	72	sty 72	Hilfsspeicher \$72 initialisieren
,b26b	a2	05	ldx #05	Ausgangswert für Speicherbedarf der Variablen laden (5 ist die Länge einer Fließkomma-Variablen im Speicher)

,b26d	a5 45	lda 45	Byte #1 des Variablennamen aus Hilfsspeicher holen
,b26f	91 5f	sta (5f),y	und in den neuen Variableneintrag schreiben
,b271	10 01	bpl b274	b7 war gelöscht (N=0): keine Integer- oder Stringvariable, also keine Veränderung des Speicherbedarfs (s. \$b26b)
,b273	ca	dex "ldx #04"	Speicherbedarf verringern, da es sich um Integer- oder Stringvariable handelt
,b274	c8	→iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Byte #2 des Variablennamen stellen)
,b275	a5 46	lda 46	Byte #2 des Variablennamen aus Hilfsspeicher holen
,b277	91 5f	sta (5f),y	und in den neuen Variableneintrag schreiben
,b279	10 02	bpl b27d	b7 gelöscht, also keine Integervariable (N=0): Speicherbedarf (hier: 4) unverändert übernehmen, da es sich um eine Stringvariable handelt
,b27b	ca	dex "ldx #03"	Speicherbedarf von 4 auf 3 herunterzählen } Speicherbedarf einer
,b27c	ca	dex "ldx #02"	Speicherbedarf von 3 auf 2 herunterzählen } Integervariablen einstellen
,b27d	86 71	→stx 71	Speicherbedarf der Variablen (2, 4 oder 5) in Hilfsspeicher \$71 merken
,b27f	a5 0b	lda 0b	Hilfsspeicher für Anzahl der Dimensionen im aktuellen Array holen
,b281	c8	iny	Offset um 1 erhöhen } Offset
,b282	c8	iny	Offset um 1 erhöhen } um 3
,b283	c8	iny	Offset um 1 erhöhen } erhöhen
,b284	91 5f	sta (5f),y	Anzahl der Dimensionen des Arrays in Array-Eintrag ablegen
,b286	a2 0b	→ldx #0b	11 als Index-Wert für noch nicht dimensioniertes, aber dennoch benutztes Array laden
,b288	a9 00	lda #00	Ausgangswert für Anzahl der Dimensionen in noch nicht dimensioniertem, aber dennoch benutztem Array laden
,b28a	24 0c	bit 0c	Array-Dimensionierungsflag testen
,b28c	50 08	bvc b296	b6 gelöscht, also keine Dimensionierung gewünscht (V=0): DIM-Sonderbehandlung überspringen
,b28e	68	pla	Anzahl der Dimensionen vom Stapel holen (s. Parameterauswertung bei \$b1d1-\$b20b, v.a. \$b209)
,b28f	18	clc	Carry vor Addition löschen
,b290	69 01	adc #01	1 addieren, um tatsächliche Anzahl der Dimensionen zu erhalten
,b292	aa	tax	Ergebnis ins X-Register bringen
,b293	68	pla	nächstes Byte vom Stapel holen
,b294	69 00	adc #00	bei Additionsübertrag (s. \$b290) wird hier 1 addiert
,b296	c8	→iny	Offset um 1 erhöhen
,b297	91 5f	sta (5f),y	Ergebnis in Array-Bereich schreiben
,b299	c8	iny	Offset um 1 erhöhen
,b29a	8a	txa	bei \$b292 gemerktes Ergebnis wieder in Akku holen
,b29b	91 5f	sta (5f),y	und Ergebnis in Array-Bereich schreiben
,b29d	20 4c b3	jsr b34c	Hilfsroutine zur Berechnung des benötigten Speicherplatzes aufrufen
,b2a0	86 71	stx 71	LB des Ergebnisses in LB von \$71/\$72 merken } Ergebnis (benötigtes Ende des
,b2a2	85 72	sta 72	HB des Ergebnisses in HB von \$71/\$72 merken } Variablenbereichs) merken
,b2a4	a4 22	ldy 22	Offset zum Array-Eintrag holen
,b2a6	c6 0b	dec 0b	Nummer der aktuellen Dimension um 1 dekrementieren

,b2a8	d0 1dc	— bne b286	noch nicht heruntergezählt (Z=0): Suche nach richtiger Position für Variableneintrag fortsetzen
,b2aa	65 59	adc 59	zum HB des Ergebnisses (s. \$b29d, \$b2a2) die Array-Länge addieren
,b2ac	b0 5d	↓ bcs b30b	HB-Additionsübertrag (C=1): OUT OF MEMORY ERROR auslösen
,b2ae	85 59	sta 59	Additionsergebnis in \$59 merken
,b2b0	a8	tay	und ins Y-Register transportieren
,b2b1	8a	txa	LB des Ergebnisses (s. \$b29d, \$b2a0) in Akku bringen
,b2b2	65 58	adc 58	LB der Array-Länge addieren
,b2b4	90 03	— bcc b2b9	kein Übertrag (C=0): keine Erhöhung des HB
,b2b6	c8	iny	HB (s. \$b2b0) erhöhen, um Additionsübertrag der LBs zu berücksichtigen
,b2b7	f0 52	↓ beq b30b	HB würde den Wert 0 annehmen (Z=1): OUT OF MEMORY ERROR auslösen
,b2b9	20 08	a4 → jsr a408 "chkfvm"	Prüfroutine, ob ausreichend Platz im Variablenspeicher vorhanden ist; gleichzeitig wird Speicherplatz organisiert
,b2bc	85 31	sta 31	LB des Zeigers auf Ende des Array-Bereichs aktualisieren
,b2be	84 32	sty 32	HB des Zeigers auf Ende des Array-Bereichs aktualisieren
,b2c0	a9 00	lda #00	Initialisierungswert für neuen Array-Eintrag laden
,b2c2	e6 72	inc 72	HB des Hilfszeigers \$71/\$72 (s. \$b2a2) erhöhen
,b2c4	a4 71	ldy 71	LB des Hilfszeigers \$71/\$72 holen
,b2c6	f0 05	— beq b2cd	schon auf 0 gesetzt (Z=1): Fortsetzung der Füllschleife mit neuem HB des Hilfszeigers
,b2c8	88	→ dey	Offset herunterzählen (gleichzeitig Schleifenzähler)
,b2c9	91 58	sta (58),y	Initialisierungswert in Array-Speicher schreiben (s. \$b2c0; Akku wird in Schleife nicht verändert)
,b2cb	d0 fb	— bne b2c8	Offset wurde bei \$b2c8 noch nicht auf 0 heruntergezählt (Z=0): innere Schleife fortsetzen (der STA-Befehl bei \$b2c9 hat die CPU-Flags nicht beeinflusst)
,b2cd	c6 59	→ dec 59	HB des Füllzeigers \$58/\$59 dekrementieren
,b2cf	c6 72	dec 72	HB des Schleifenzählers \$71/\$72 dekrementieren
,b2d1	d0 f5	— bne b2c8	noch nicht auf 0 heruntergezählt (Z=0): weiter mit neuem HB des Füllzeigers
,b2d3	e6 59	inc 59	HB des Füllzeigers \$58/\$59 erhöhen
,b2d5	38	sec	Carry vor Subtraktion bei \$b2d8 setzen
,b2d6	a5 31	lda 31	LB des Zeigers auf Ende des Array-Bereichs holen
,b2d8	e5 5f	sbc 5f	davon das LB des Füllzeigers (= Anfang des Array-Eintrags) abziehen
,b2da	a0 02	ldy #02	Offset innerhalb des Array-Kopfes auf LB der Array-Länge stellen
,b2dc	91 5f	sta (5f),y	Ergebnis als LB der Array-Länge in Array-Kopf schreiben
,b2de	a5 32	lda 32	HB des Zeigers auf Ende des Array-Bereiches holen
,b2e0	c8	iny "ldy #03"	Offset innerhalb des Array-Kopfes auf HB der Array-Länge stellen
,b2e1	e5 60	sbc 60	vom bei \$b2de eingelesenen HB das HB der Array-Anfangsadresse abziehen
,b2e3	91 5f	sta (5f),y	und als HB der Array-Länge in Array-Kopf schreiben
,b2e5	a5 0c	lda 0c	Array-Dimensionierungsflag zwecks Test auslesen
,b2e7	d0 62	↓ bne b34b	Array-Dimensionierungsflag war gesetzt (Z=0): RTS-Befehl anspringen, da Array ordnungsgemäß dimensioniert wurde

Array-Länge
durch
Subtraktion
der Anfangs-
adresse des
Arrays v.d.
Endadresse
ermitteln

; Suche einer Arrayvariablen in einem Array, auf dessen Kopf der Hilfszeiger \$5f/\$60 zeigt; Y-Register enthält den Offset zum HB der Array-Länge im Array-Kopf

,b2e9	c8	iny	Offset erhöhen (auf Dimension des Arrays stellen)	} Dimension des aktuellen
,b2ea	b1 5f	lda (5f),y	Dimension des Arrays (Anzahl der Indizes) holen	} Arrays inentsprechenden
,b2ec	85 0b	sta 0b	und in dafür verwendeten Hilfsspeicher \$0b schreiben	} Hilfsspeicher holen
,b2ee	a9 00	lda #00	Initialisierungswert für Hilfszeiger \$71/\$72 laden	} Hilfszeiger \$71/\$72
,b2f0	85 71	sta 71	LB des Hilfszeigers \$71/\$72 initialisieren	} mit \$0000
,b2f2	85-72	→sta 72	HB des Hilfszeigers \$71/\$72 initialisieren	} initialisieren
,b2f4	c8	iny	Offset erhöhen (auf "Ausdehnung" in Dimension stellen)	
,b2f5	68	pla	LB des gewünschten Index in aktueller Dimension holen	
,b2f6	aa	tax	und ins X-Register bringen	
,b2f7	85 64	sta 64	als LB des Index in \$64/\$65 setzen	
,b2f9	68	pla	HB des gewünschten Index in aktueller Dimension holen	
,b2fa	85 65	sta 65	und als HB des Index in \$64/\$65 setzen	
,b2fc	d1 5f	cmp (5f),y	Vergleich der beiden HBs (Akku = HB des gewünschten Index; (\$5f/\$60)+Y enthält maximalen Index-Wert derselben Dimension)	
,b2fe	90 0e	bcc b30e	gewünschter Index < maximaler Index (C=0): Adresse der Arrayvariablen ermitteln	
,b300	d0 06	bne b308	gewünschter Index > maximaler Index (Z=0): BAD SUBSCRIPT auslösen, da Index zu hoch	
,b302	c8	iny	Offset auf "Ausdehnung" in nächster Dimension stellen	
,b303	8a	txa	LB des gewünschten Index (s. \$b2f5/\$b2f6) in Akku zwecks Vergleich holen	
,b304	d1 5f	cmp (5f),y	Vergleich der beiden LBs (Akku = LB des gewünschten Index; (\$5f/\$60)+Y enthält maximalen Index-Wert derselben Dimension)	
,b306	90 07	bcc b30f	gewünschter Index < maximaler Index (C=0): Adresse der Arrayvariablen ermitteln; in Berechnungsroutine wird 1 Byte später eingestiegen, um das bei \$b302 bereits erfolgte Erhöhen des Offset nicht ein weiteres Mal zu durchlaufen	
,b308	4c 45 b2	→jmp b245	BAD SUBSCRIPT ERROR auslösen, weil unerlaubt hoher Index verwendet wurde	

,b30b	4c 35 a4	jmp a435	OUT OF MEMORY ERROR auslösen, wenn Array dimensioniert werden sollte, aber kein Platz vorhanden war	

; Adresse einer Arrayvariablen ermitteln, deren Zulässigkeit bereits geprüft wurde, wobei wiederum die Array-Hilfsspeicher (\$0b, \$64/\$65, \$71/\$72, Y-Register usw.) auf die richtigen Werte gestellt wurden

,b30e	c8	→iny	Offset erhöhen (wird in Unterroutine ab \$b34c benötigt), um ihn auf maximale Größe der nächsten Dimension zu richten	
,b30f	a5 72	→lda 72	HB des Hilfszeigers \$71/\$72 holen	} OR-Verknüpfung von \$71 und \$72
,b311	05 71	ora 71	mit LB des Hilfszeigers \$71/\$72 verknüpfen	} zwecks Test, ob \$71/\$72=\$0000
,b313	18	clc	Carry vor Addition (evtl. bei \$b320) löschen	

,b314	f0 0a	beq b320	\$71/\$72 enthält \$0000 (Z=1): ersten Teil der Berechnungen überspringen
,b316	20 4c b3	jsr b34c	Multiplikationsroutine (enthält UMULT-Einsprung) zur Berechnung der Länge einer Dimension durch Multiplikation der maximalen Elementezahl mit der Länge eines Elements; Ergebnis kommt nach X/Y
,b319	8a	txa	LB des Produkts in Akku zwecks Addition holen
,b31a	65 64	adc 64	LB des gewünschten Index addieren; Carry ist seit \$b316 gelöscht
,b31c	aa	tax	Ergebnis als neues LB ins X-Register zurückschreiben
,b31d	98	tya	HB des Produkts zwecks Addition in Akku holen
,b31e	a4 22	ldy 22	Offset wieder ins Y-Register holen; wurde bei Multiplikation als Ergebnisspeicher verwendet und muß hier wiederhergestellt werden (s. \$b34c)
,b320	65 65	adc 65	HB des gewünschten Index addieren; Ergebnis der Addition liegt jetzt in X/A
,b322	86 71	stx 71	LB des Ergebnisses als LB des Hilfszeigers \$71/\$72 merken
,b324	c6 0b	dec 0b	Anzahl der Dimensionen im aktuellen Array herunterzählen
,b326	d0-ca	bne b2f2	noch nicht auf 0 heruntergezählt (Z=0): weiter mit Bearbeitung der nächsten Dimension
,b328	85 72	sta 72	HB in HB des Hilfszeigers \$71/\$72 schreiben
,b32a	a2 05	ldx #05	Ausgangswert für Länge einer Variablen in Bytes laden (5 = Fließkomma-Variable)
,b32c	a5 45	lda 45	Byte #1 des Variablennamen holen
,b32e	10 01	bpl b331	b7 gelöscht (N=0): Variablenlängenangabe nicht ändern, da keine String- oder Integervariable
,b330	ca	dex "ldx #04"	Variablenlänge von 5 auf 4 herunterzählen, da keine Fließkomma-Variable
,b331	a5 46	lda 46	Byte #2 des Variablennamen holen
,b333	10 02	bpl b337	b7 gelöscht (N=0): keine Integervariable, also X-Wert übernehmen
,b335	ca	dex "ldx #03"	Längenangabe=Längenangabe-1 } Längenangabe um 2 herunterzählen
,b336	ca	dex "ldx #02"	Längenangabe=Längenangabe-1 } (von 4 auf 2 bringen, da Integervariable)
,b337	86 28	stx 28	berechnete Variablenlänge (2, 3 oder 5) in Hilfsspeicher \$28 merken
,b339	a9 00	lda #00	HB des Faktors in X/A laden; LB wurde nach X berechnet
,b33b	20 55 b3	jsr b355	Multiplikationsroutine aufrufen, um ermittelte Variablenlänge mit Elementezahl zu multiplizieren; Ergebnis kommt nach X/A
,b33e	8a	txa	LB des Ergebnisses in Akku (zwecks Addition) bringen
,b33f	65 58	adc 58	dazu das LB der Adresse des 1. Elements (bei \$b264 berechnet) addieren; Carry ist nach \$b33b immer auf 0 gesetzt
,b341	85 47	sta 47	und Ergebnis als LB der Adresse der aktuellen Variablen setzen
,b343	98	tya	HB des Ergebnisses in Akku zwecks Addition bringen
,b344	65 59	adc 59	dazu das HB der Adresse des 1. Elements (bei \$b264 berechnet) addieren
,b346	85 48	sta 48	und Ergebnis als HB der Adresse der aktuellen Variablen setzen
,b348	a8	tay	zugleich ins Y-Register für Rückgabe des Ergebnisses in A/Y
,b349	a5 47	lda 47	LB (s. \$b341) in Akku holen, da Ergebnis in A/Y zurückgegeben wird } Ergebnis nach A/Y holen
,b34b	60	rts	Rücksprung von Routine

; Multiplikationsroutine zur Berechnung von maximaler Elementezahl in einer Dimension mal DIM-Wert;
 Aufruf von \$b29d und \$b316

,b34c	84 22	sty 22	Offset in \$22 merken; wird dann bei \$b31e wiederhergestellt
,b34e	b1 5f	lda (5f),y	LB der maximalen Elementezahl ("Ausdehnung") der Dimension holen
,b350	85 28	sta 28	und als LB des ersten Faktors setzen
,b352	88	dey	Offset herunterzählen; wird dadurch auf HB der maximalen Elementezahl gestellt, da an dieser Stelle das ungebräuchliche High-Low-Format (statt "Low-High") gilt
,b353	b1 5f	lda (5f),y	HB der maximalen Elementezahl ("Ausdehnung") in Akku holen

; hier: Aufruf von \$b33b

,b355	85 29	sta 29	Akku als HB des ersten Faktors setzen
-------	-------	--------	---------------------------------------

; UMULT-Einsprung: Multiplikation von (\$28/\$29) mit (\$71/\$72); Ergebnis kommt nach X/Y
 wird im ROM nur bei direkter Befehlsausführung hinter \$b355 genutzt, kann jedoch von eigenen Routinen gut verwendet werden

,b357	a9 10	lda #10 %00010000	Zähler für 16 Bit laden	
,b359	85 5d	sta 5d	und in Hilfsspeicher \$5d schreiben (dient als Schiebezahl)	
,b35b	a2 00	ldx #00	LB des Ergebnis-Ausgangswertes laden	} \$0000 als Ausgangswert des Ergebnisses laden
,b35d	a0 00	ldy #00	HB des Ergebnis-Ausgangswertes laden	
,b35f	8a	→txa	LB des aktuellen Wertes für das Ergebnis in Akku holen	} Verschiebung von X/Y um 1 Bit nach links
,b360	0a	asl	Akku mit 2 multiplizieren (Linksverschiebung um 1 Bit)	
,b361	aa	tax	Ergebnis wieder ins X-Register	
,b362	98	tya	HB des aktuellen Wertes für das Ergebnis in Akku holen	
,b363	2a	rol	Akku mit 2 multiplizieren (Linksverschiebung um 1 Bit)	} links
,b364	a8	tay	Ergebnis wieder ins Y-Register	
,b365	b0 a4	↖ bcs b30b	Übertrag bei Linksverschiebung (C=1): OUT OF MEMORY ERROR auslösen, da Ergebnis einen unerlaubten Wert darstellt	
,b367	06 71	asl 71	LB des Faktors in \$71/\$72 nach links verschieben	} Verschiebung des Faktors in \$71/\$72 nach links
,b369	26 72	rol 72	HB des Faktors in \$71/\$72 nach links verschieben	
,b36b	90 0b	bcc b378	oberstes Bit war nicht gesetzt, kein Übertrag (C=0): keine Ergebnisveränderung nötig, Bitzähler dekrementieren und weiter in Schleife	
,b36d	18	clc	Carry vor Addition bei \$b36f löschen	} aktuelles Ergebnis in X/Y um zweiten Faktor erhöhen
,b36e	8a	txa	LB des aktuellen Wertes für das Ergebnis in Akku zwecks Addition	
,b36f	65 28	adc 28	LB des zweiten Faktors addieren	
,b371	aa	tax	zurück in X-Register als LB des aktuellen Wertes für das Ergebnis	
,b372	98	tya	HB des aktuellen Wertes für das Ergebnis in Akku zwecks Addition	
,b373	65 29	adc 29	HB des zweiten Faktors addieren	
,b375	a8	tay	zurück in Y-Register als HB des aktuellen Wertes für das Ergebnis	


```
,b376 b0 93  ← bcs b30b      Additionsübertrag (C=1): OUT OF MEMORY ERROR auslösen
,b378 c6 5d  → dec 5d        Schiebezähler (Ausgangswert $10, s. $b357) dekrementieren
,b37a d0 e3  ← bne b35f      noch nicht heruntergezählt (Z=0): weiter in Multiplikationsschleife
,b37c 60      rts            Rücksprung von Routine, Ergebnis steht in X/Y, wobei der Akku den Y-Wert beinhaltet
```

; Routine zur Basic-Funktion FRE (Token: \$b8)

```
,b37d a5 0d    lda 0d        Datentyp-Flag zwecks Test auslesen
,b37f f0 03    beq b384      FRE-Parameter war numerisch (Z=1): String-Sonderbehandlung überspringen
,b381 20 a6 b6  jsr b6a6 "frestr" String auswerten und in Stringstapel übernehmen
,b384 20 26 b5  jsr b526 "garcol" Garbage Collection durchführen
,b387 38       sec          Carry vor Subtraktion bei $b38a setzen
,b388 a5 33    lda 33        LB der Obergrenze des Stringbereichs holen
,b38a e5 31    sbc 31        LB der Endadresse der Arrays (+1) abziehen
,b38c a8       tay          Ergebnis als LB des Ergebnisses ins Y-Register
,b38d a5 34    lda 34        HB der Obergrenze des Stringbereichs holen
,b38f e5 32    sbc 32        HB der Endadresse der Arrays (+1) abziehen
```

} Ermittlung des freien
Speicherplatzes durch
Subtraktion der Endadresse
des Array-Bereichs von der
Obergrenze des Stringbereichs

; INTFAC: Aufruf von \$aee0 (NOT), \$af6b (GETVAR), \$b013 (OR/AND), \$b3a4 (POS)

```
,b391 a2 00    → ldx #00     Datentyp-Flag für "numerisch" laden
,b393 86 0d    stx 0d        Datentyp-Flag (String/numerisch) auf "numerisch" setzen
,b395 85 62    sta 62        HB des Ergebnisses in $62 ablegen (s. $b38d/$b38f)
,b397 84 63    sty 63        LB des Ergebnisses in $63 ablegen (s. $b388-$b38c)
,b399 a2 90    ldx #90 %10010000 Exponent laden, daß Fließkomma-Zahl im richtigen Bereich (-32768 bis +32767) entsteht
,b39b 4c 44 bc  jmp bc44 "wrdfac" Integerzahl in FAC #1 als Fließkomma-Zahl bringen
```

} Ergebnis
nach \$62/\$63

; Routine zur Basic-Funktion POS (Token: \$b9)

```
,b39e 38       sec          Carry als Flag für "Position auslesen" setzen
,b39f 20 f0 ff  jsr fff0 "plot" Cursorposition nach X (Zeile) und Y (Spalte) holen
```

; BYTFAC: Aufruf von \$b77f (LEN), \$b795 (ASC), \$b821 (PEEK)

```
,b3a2 a9 00    lda #00        Highbyte von Y/A löschen, da Ergebnis ein Bytewert ist
,b3a4 f0 eb    ← beq b391 "jmp intf" Ergebnis (Cursorspalte) als Fließkomma-Zahl in FAC #1 zurückgeben
```

; CHKDIR-Routine: Ausgabe von ILLEGAL DIRECT ERROR bei Aufruf im Direktmodus (keine Wirkung im Programm-Modus);
 Aufruf von \$ab7b (READ/INPUT/GET) und \$b3b6 (DEF)

,b3a6	a6 3a	ldx 3a	HB der aktuellen Zeilennummer holen (gleichzeitig Flag für Direktmodus, wenn \$ff!)
,b3a8	e8	inx	zwecks Test auf \$ff erhöhen; bei X=\$ff wird X hier der Wert 0 gegeben
,b3a9	d0 a0	↙ bne b34b	kein Direktmodus (Z=0): RTS-Befehl anspringen, also ordnungsgemäßer Rücksprung
,b3ab	a2 15	ldx #15	Fehlernummer für ILLEGAL DIRECT laden
,b3ad	2c-a2-1b→"bit"	ldx #1b	Fehlernummer für UNDEF'D FUNCTION laden
,b3b0	4c 37 a4	jmp a437 "error"	Ausgabe der Fehlermeldung (ILLEGAL DIRECT bei Abarbeitung von \$b3ab, UNDEF'D FUNCTION bei Ausführung von \$b3ad)

 ; Routine zum Basic-Befehl DEF (Token: \$96)

,b3b3	20 e1 b3	jsr b3e1 "chkfns"	Prüfroutine für Syntax von benutzerdefinierten Funktionen aufrufen
,b3b6	20 a6 b3	jsr b3a6 "chkdir"	DEF nur im Programm-Modus zulassen
,b3b9	20 fa ae	jsr aefa "chkbro"	auf "(" als syntaktisches Erfordernis testen
,b3bc	a9 80	lda #80 %100000000	Flag für "Integervariablen gesperrt" laden
,b3be	85 10	sta 10	und in Integerflag schreiben
			} Integervariablen sperren
,b3c0	20 8b b0	jsr b08b "fndvar"	Variable holen und anlegen (falls nicht vorhanden)
,b3c3	20 8d ad	jsr ad8d "chknum"	nur numerische Variable zulassen
,b3c6	20 f7 ae	jsr aef7 "chkbcl"	auf ")" als syntaktisches Erfordernis testen
,b3c9	a9 b2	lda #b2	Token von "=" laden
,b3cb	20 ff ae	jsr aeff "chkbyt"	auf Akku-Inhalt ("="-Token) als syntaktisches Erfordernis testen
,b3ce	48	pha	Zeichen hinter "=" auf Stapel merken
,b3cf	a5 48	lda 48	HB der Adresse der FN-Variablen holen
,b3d1	48	pha	und auf den Stapel legen
,b3d2	a5 47	lda 47	LB der Adresse der FN-Variablen holen
,b3d4	48	pha	und auf den Stapel legen
,b3d5	a5 7b	lda 7b	HB des CHRGET-Zeigers holen
,b3d7	48	pha	und auf den Stapel legen
,b3d8	a5 7a	lda 7a	LB des CHRGET-Zeigers holen
,b3da	48	pha	und auf den Stapel legen
			} Adresse der FN-Variablen auf dem Stapel merken
,b3db	20 f8 a8	jsr a8f8 "ignorc"	Routine zum Basic-Befehl DATA ausführen, um CHRGET-Zeiger auf nächstem Befehl zu positionieren
,b3de	4c 4f b4	jmp b44f	an Ende von FN-Behandlung springen, um auf Stapel gelegte Werte zu verarbeiten

; CHKFNS-Routine zur Prüfung auf syntaktische Richtigkeit einer über FN benutzerdefinierten Funktion

,b3e1	a9 a5	lda #a5	Token von FN laden		} Angabe des Schlüsselwortes
,b3e3	20 ff ae	jsr aeff "chkbyt"	auf FN als syntaktisches Erfordernis testen		} FN verlangen
,b3e6	09 80	ora #80 %10000000	b7 im Akku setzen (Flag für "Integer gesperrt")		} Integervariablen
,b3e8	85 10	sta 10	und in Integerflag schreiben		} sperren
,b3ea	20 92 b0	jsr b092 "fndvar"	Variable (in diesem Fall: FN-Variable) suchen		
,b3ed	85 4e	sta 4e	LB der Adresse des Auffindens in LB von \$4e/\$4f merken		} Adresse der FN-Variablen
,b3ef	84 4f	sty 4f	HB der Adresse des Auffindens in HB von \$4e/\$4f merken		} nach \$4e/\$4f
,b3f1	4c 8d ad	jmp ad8d "chknum"	nur numerische Parameter zulassen		

; Routine zur Basic-Funktion FN (Token: \$a5)

,b3f4	20 e1 b3	jsr b3e1 "chkfns"	auf FN als syntaktisches Erfordernis testen	} FN verlangen
,b3f7	a5 4f	lda 4f	HB der Adresse des Auffindens (s. \$b3ea, \$b3ef) laden	} Adresse der FN-Variablen
,b3f9	48	pha	und auf den Stapel legen	
,b3fa	a5 4e	lda 4e	LB der Adresse des Auffindens (s. \$b3ea, \$b3ed) laden	} auf den Stapel legen
,b3fc	48	pha	und auf den Stapel legen	
,b3fd	20 f1 ae	jsr aef1 "brcevl"	in Klammern stehenden numerischen Ausdruck auswerten	} numerischen Parameter in Klammern auswerten
,b400	20 8d ad	jsr ad8d "chknum"	nur numerische Parameter zulassen	
,b403	68	pla	bei \$b3fa/\$b3fc gemerktes LB der Adresse der FN-Variablen	} Adresse der FN-Variablen
,b404	85 4e	sta 4e	vom Stapel in den Hilfszeiger \$4e/\$4f zurückholen	
,b406	68	pla	bei \$b3f7/\$b3f9 gemerktes HB der Adresse der FN-Variablen	} vom Stapel zurückholen
,b407	85 4f	sta 4f	vom Stapel in den Hilfszeiger \$4e/\$4f zurückholen	
,b409	a0 02	ldy #02	Offset mit 2 initialisieren (auf Byte #1 des Variableninhalts stellen)	
,b40b	b1 4e	lda (4e),y	LB der Adresse der FN-Variablen holen	
,b40d	85 47	sta 47	und als LB der aktuellen Variablenadresse setzen	
,b40f	aa	tax	zudem ins X-Register schreiben	
,b410	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf Byte #2 des Variableninhalts stellen)	
,b411	b1 4e	lda (4e),y	HB der Adresse der FN-Variablen holen	
,b413	f0-99	beq b3ae	HB=\$00, also FN-Funktion noch undefiniert (Z=1): UNDEF'D FUNCTION ERROR auslösen	
,b415	85 48	sta 48	als HB der aktuellen Variablenadresse setzen	
,b417	c8	iny "ldy #04"	Offset von 3 auf 4 erhöhen (auf Byte #3 des Variableninhalts stellen)	
,b418	b1 47	→ lda (47),y	FN-Variableninhalt auslesen (s. \$b40d, \$b415)	
,b41a	48	pha	und auf den Stapel legen	
,b41b	88	dey	Offset herunterzählen	
,b41c	10 fa	bpl b418	noch nicht auf \$ff heruntergezählt (N=0): weiter in Stapel-Ablage-Schleife	
,b41e	a4 48	ldy 48	HB der Adresse der FN-Variablen in Y-Register holen	
,b420	20 d4 bb	jsr bbd4 "movfm"	FAC #1 in Variablenspeicher übertragen; danach ist auch Y=0	

,b423	a5 7b	lda 7b	HB des CHRGET-Zeigers holen	} CHRGET-Zeiger auf den Stapel legen	
,b425	48	pha	und auf den Stapel legen		
,b426	a5 7a	lda 7a	LB des CHRGET-Zeigers holen		
,b428	48	pha	und auf den Stapel legen		
,b429	b1 4e	lda (4e),y	LB der Adresse des FN-Befehls im Basic-Text holen	} Adresse der FN-Variablen in CHRGET-Zeiger schreiben, um ihn hinter FN zu positionieren	
,b42b	85 7a	sta 7a	und in LB des CHRGET-Zeigers schreiben		
,b42d	c8	iny "ldy #01"	Offset erhöhen (auf HB stellen)		
,b42e	b1 4e	lda (4e),y	HB der Adresse des FN-Befehls im Basic-Text holen		
,b430	85 7b	sta 7b	und in HB des CHRGET-Zeigers schreiben	} Adresse der FN-Variablen auf den Stapel legen	
,b432	a5 48	lda 48	HB der Adresse der FN-Variablen in Akku holen		
,b434	48	pha	und auf den Stapel legen		
,b435	a5 47	lda 47	LB der Adresse der FN-Variablen in Akku holen		
,b437	48	pha	und auf den Stapel legen		
,b438	20 8a ad	jsr ad8a "frmnum"	numerischen Ausdruck auswerten; in diesem Fall ist es die benutzerdefinierte Funktion, da die CHRGET-Zeiger vorher auf sie gerichtet wurden		
,b43b	68	pla	LB der Adresse der FN-Variablen vom Stapel holen	} Adresse der FN-Variablen vom Stapel in Hilfsspeicher holen	
,b43c	85 4e	sta 4e	und als LB in Hilfsspeicher \$4e/\$4f schreiben		
,b43e	68	pla	HB der Adresse der FN-Variablen vom Stapel holen		
,b43f	85 4f	sta 4f	und als HB in Hilfsspeicher \$4e/\$4f schreiben		
,b441	20 79 00	jsr 0079 "chrnot"	letztes Zeichen hinter Funktionsdefinition zwecks Test einlesen		
,b444	f0 03	beq b449	Endmarkierung (Z=1): keinen SYNTAX ERROR auslösen		
,b446	4c 08 af	jmp af08 "synerr"	SYNTAX ERROR auslösen		

,b449	68	>pla	LB des CHRGET-Zeigers vom Stapel zurückholen	} CHRGET-Zeiger-Position vor Aufruf von FN vom Stapel zurück in den CHRGET-Zeiger holen	
,b44a	85 7a	sta 7a	und in LB des CHRGET-Zeigers schreiben		
,b44c	68	pla	HB des CHRGET-Zeigers vom Stapel zurückholen		
,b44d	85 7b	sta 7b	und in HB des CHRGET-Zeigers schreiben		
,b44f	a0 00	ldy #00	Offset mit 0 initialisieren	} Inhalt der FN-Variablen byteweise vom Stapel holen und in den Variablenspeicher übertragen	
,b451	68	pla	Byte #1 des FN-Variableninhalts vom Stapel holen		
,b452	91 4e	sta (4e),y	und in den Variablenspeicher schreiben		
,b454	68	pla	Byte #2 des FN-Variableninhalts vom Stapel holen		
,b455	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen		
,b456	91 4e	sta (4e),y	und Byte #2 in den Variablenspeicher schreiben		
,b458	68	pla	Byte #3 des FN-Variableninhalts vom Stapel holen		
,b459	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen		
,b45a	91 4e	sta (4e),y	und Byte #3 in den Variablenspeicher schreiben		
,b45c	68	pla	Byte #4 des FN-Variableninhalts vom Stapel holen		
,b45d	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen		
,b45e	91 4e	sta (4e),y	und Byte #4 in den Variablenspeicher schreiben		
,b460	68	pla	Byte #5 des FN-Variableninhalts vom Stapel holen		

,b461	c8	iny "ldy #04"	Offset von 3 auf 4 erhöhen	}
,b462	91 4e	sta (4e),y	und Byte #5 in den Variablenspeicher schreiben	
,b464	60	rts	Rücksprung von Routine	

; Routine zur Basic-Funktion STR\$ (Token: \$c4)

,b465	20 8d ad	jsr ad8d "chknum"	Prüfroutine, ob numerischer Parameter übergeben wurde, aufrufen	
,b468	a0 00	ldy #00	Vorbereitung für Aufruf der Umwandlungsroutine: Offset initialisieren	
,b46a	20 df bd	jsr bddf "flpstr"	FAC #1 in ASCII-String konvertieren (in FLPSTR-Routine mit Offset 0 einsteigen)	
,b46d	68	pla	LB der Rücksprungadresse vom Stapel löschen	} Rücksprungadresse am Stapel tilgen
,b46e	68	pla	HB der Rücksprungadresse vom Stapel löschen	
,b46f	a9 ff	lda #ff <(\$00ff)	LB der Adresse des bei \$b46a generierten ASCII-Strings laden	
,b471	a0 00	ldy #00 >(\$00ff)	HB der Adresse des bei \$b46a generierten ASCII-Strings laden	
,b473	f0-12	beq b487 "jmp strlit"	in STRLIT-Routine einsteigen (ermittelten String auf temporärem Stringstapel von Funktion zurückgeben)	

; Hilfsroutine zur Ermittlung der Stringparameter und Organisation von String-Speicherplatz

,b475	a6 64	ldx 64	LB der Adresse des aktuellen Variableneintrags holen	} Adresse der aktuellen Variablen übergeben
,b477	a4 65	ldy 65	HB der Adresse des aktuellen Variableneintrags holen	
,b479	86 50	stx 50	in LB von Übergabezeiger \$50/\$51 schreiben	
,b47b	84 51	sty 51	in HB von Übergabezeiger \$50/\$51 schreiben	
,b47d	20 f4 b4	jsr b4f4	Speicherplatz (Bytezahl im Akku) im Stringspeicher organisieren	
,b480	86 62	stx 62	LB der Stringadresse setzen	} Stringadresse und Stringlänge nach \$62/\$63 und \$61
,b482	84 63	sty 63	HB der Stringadresse setzen	
,b484	85 61	sta 61	Stringlänge setzen	
,b486	60	rts	Rücksprung von Routine	

; STRLIT-Routine:

Übergabe eines bei einer Stringfunktion ermittelten Strings (Adresse in A/Y) auf dem temporären Stringstapel;
Aufruf von \$aec6

,b487	a2-22	→ldx #22	ASCII-Code des Anführungszeichens laden
,b489	86 07	stx 07	als Suchbyte #1 und
,b48b	86 08	stx 08	als Suchbyte #2 setzen

; Einsprung: String in temporären Stringstapel übertragen

,b48d	85 6f	sta	6f	LB der Stringadresse als LB der Quelladresse setzen	} Stringadresse in temporären Descriptor schreiben
,b48f	84 70	sty	70	HB der Stringadresse als HB der Quelladresse setzen	
,b491	85 62	sta	62	LB der Stringadresse als LB des Hilfszeigers \$62/\$63 setzen	
,b493	84 63	sty	63	HB der Stringadresse als HB des Hilfszeigers \$62/\$63 setzen	
,b495	a0 ff	ldy	#ff	Offset für Stringverschiebung initialisieren	
,b497	c8	→iny		Offset um 1 erhöhen (auf nächstes Byte stellen)	
,b498	b1 6f	lda	(6f),y	Byte aus String holen (Zeiger \$6f/\$70 wurde bei \$b48d/\$b48f gesetzt)	
,b49a	f0 0c	beq	b4a8	String-Endmarkierung (Z=1): Schleife verlassen	
,b49c	c5 07	cmp	07	Vergleich mit Suchbyte #1	
,b49e	f0 04	beq	b4a4	Übereinstimmung (Z=1): Test auf Anführungszeichen, ggf. Sonderbehandlung	
,b4a0	c5 08	cmp	08	Vergleich mit Suchbyte #2	
,b4a2	d0 f3	bne	b497	keine Übereinstimmung (Z=0): weiter in Schleife	
,b4a4	c9 22	→cmp	#22	Vergleich des aktuellen Bytes mit ASCII-Code des Anführungszeichens	
,b4a6	f0 01	beq	b4a9	Übereinstimmung (Z=1): bei gesetztem Carry (s. \$b4a4!) Schleife verlassen	
,b4a8	18	→clc		Carry löschen, da kein Anführungszeichen	
,b4a9	84 61	→sty	61	letzten Offset als tatsächliche Stringlänge setzen	} Stringlänge (vorher ermittelt) auf Anfangsadresse addieren und als Endadresse in \$71/\$72 merken
,b4ab	98	tya		und zwecks Addition in Akku bringen	
,b4ac	65 6f	adc	6f	Stringlänge zu LB der Stringadresse addieren; ggf. wird Carry berücksichtigt, s. \$b4a6	
,b4ae	85 71	sta	71	und als LB der String-Endadresse in \$71 setzen	
,b4b0	a6 70	ldx	70	HB der Stringadresse holen (s. \$b48f)	
,b4b2	90 01	bcc	b4b5	kein Übertrag bei Addition bei \$b4ac (C=0): HB nicht erhöhen	
,b4b4	e8	inx		HB erhöhen, um Additionsübertrag zu berücksichtigen	
,b4b5	86 72	→stx	72	HB der String-Endadresse in \$72 setzen	
,b4b7	a5 70	lda	70	HB der String-Anfangsadresse holen (s. \$b48f)	
,b4b9	f0 04	beq	b4bf	HB = \$00 (Z=1): Sonderbehandlung für Strings auf Page 0 oder 2	
,b4bb	c9 02	cmp	#02 >(\$0200)	HB der Stringadresse mit 2 vergleichen	
,b4bd	d0 0b	bne	b4ca	keine Übereinstimmung (Z=0): keine Sonderbehandlung für Strings aus Page 0 oder 2	

; Sonderbehandlung für Strings aus Page 0 (Zeropage) oder Page 2 (Systemeingabepuffer)

,b4bf	98	→tya		tatsächliche Stringlänge (s. \$b4a9) in Akku holen	
,b4c0	20 75	b4 jsr	b475	Hilfsroutine zur Ermittlung der Stringparameter aufrufen, um Stringspeicherplatz zu organisieren	
,b4c3	a6 6f	ldx	6f	LB der Stringadresse holen	} String in neuen Speicherbereich kopieren
,b4c5	a4 70	ldy	70	HB der Stringadresse holen	
,b4c7	20 88	b6 jsr	b688	String in Variablenbereich übernehmen	
,b4ca	a6 16	→ldx	16	Zeiger für temporären Stringstapel holen	
,b4cc	e0 22	cpx	#22	mit äußerstem erlaubten Inhalt vergleichen	

,b4ce	d0 05	bne b4d5	keine Übereinstimmung (Z=0): keinen FORMULA TOO COMPLEX ERROR auslösen	
,b4d0	a2 19	ldx #19	Fehlernummer für FORMULA TOO COMPLEX laden	
,b4d2	4c 37 a4	jmp a437 "error"	Fehlereinsprung aufrufen	

,b4d5	a5 61	lda 61	ermittelte Stringlänge (s. \$b4c0) holen	} Länge und Adresse des Strings auf den temporären Stringstapel legen (Offset=Stringstapelzeiger)
,b4d7	95 00	sta 00,x	und in temporären Stringstapel schreiben	
,b4d9	a5 62	lda 62	LB der Stringadresse (s. \$b4c0) holen	
,b4db	95 01	sta 01,x	und in temporären Stringstapel schreiben	
,b4dd	a5 63	lda 63	HB der Stringadresse (s. \$b4c0) holen	} (Offset=Stringstapelzeiger)
,b4df	95 02	sta 02,x	und in temporären Stringstapel schreiben	
,b4e1	a0 00	ldy #00	Initialisierungswert für HB der Adresse des Stringstapels und Rundungsbyte laden	
,b4e3	86 64	stx 64	Offset (= Inhalt des temporären Stringstapelzeigers)	} Adresse des Eintrags im Stringstapel als Variablenadresse setzen
			(s. \$b4ca) als LB der aktuellen Variablenadresse setzen	
,b4e5	84 65	sty 65	HB der aktuellen Variablenadresse auf 0 setzen	} Rundungsbyte und Datentyp-Flag initialisieren
,b4e7	84 70	sty 70	Rundungsbyte von FAC #1 löschen	
,b4e9	88	dey "ldy #ff"	Y-Register mit \$ff laden (vorher \$00, s. \$b4e1)	
,b4ea	84 0d	sty 0d	Datentyp-Flag (String/numerisch) auf "String" stellen	
,b4ec	86 17	stx 17	Offset (= Inhalt des temporären Stringstapelzeigers)	
			(s. \$b4ca) als Adresse des letzten Strings setzen	
,b4ee	e8	inx	temporären Stringstapelzeiger um 1 erhöhen	} temporären Stringstapelzeiger um 3 erhöhen (somit für nächsten Stapeleintrag vorbereiten)
,b4ef	e8	inx	temporären Stringstapelzeiger um 1 erhöhen	
,b4f0	e8	inx	temporären Stringstapelzeiger um 1 erhöhen	
,b4f1	86 16	stx 16	und als neuen Zeiger für temporären Stringstapel setzen	
,b4f3	60	rts	Rücksprung von Routine	

; Stringeintrag von bestimmter Länge (Bytezahl im Akku) im Stringspeicher organisieren;

Aufruf nur von \$b47d;

s. Fließtext wegen simulierter Subtraktion bei \$b4f7-\$b500

,b4f4	46 0f	lsr 0f	Flag für Garbage Collection durch Rechtsverschiebung der Bits löschen	
,b4f6	48	pha	im Akku übergebene Stringlänge auf Stapel sichern	
,b4f7	49 ff	eor #ff %11111111	Stringlänge komplementieren	
,b4f9	38	sec	Carry setzen, da Addition bei \$b4fa aufgrund der Komplementierung von \$b4f7 als Subtraktionsersatz dient	
,b4fa	65 33	adc 33 "sbc"	LB der Anfangsadresse des Stringbereichs "addieren" (in Wirklichkeit: Subtraktion)	
,b4fc	a4 34	ldy 34	HB der Anfangsadresse des Stringbereichs laden	
,b4fe	b0 01	bcs b501	kein Subtraktionsübertrag bei \$b4fa (C=1): HB nicht dekrementieren	
,b500	88	dey	HB dekrementieren, um Subtraktionsübertrag zu berücksichtigen	
,b501	c4 32	cpy 32	Vergleich des neu errechneten HB mit HB des Zeigers auf die Endadresse der Arrays	
,b503	90 11	bcc b516	ermitteltes HB < Endadressen-HB für Arrays (C=0): GarbageCollection auslösen	

,b505	d0	04	bne b50b	ermitteltes HB > Endadressen-HB für Arrays (Z=0): LB-Vergleich überspringen	
,b507	c5	31	cmp 31	Vergleich des neu errechneten LB mit LB des Zeigers auf die Endadresse der Arrays	
,b509	90	0b	bcc b516	ermitteltes LB < Endadressen-LB für Arrays (C=0): Garbage Collection auslösen	
; keine Garbage Collection zur Organisation des Speicherplatzes erforderlich:					
,b50b	85	33	→sta 33	LB der neu errechneten Adresse als LB der Stringbereichs-Anfangsadresse setzen	
,b50d	84	34	sty 34	HB der neu errechneten Adresse als HB der Stringbereichs-Anfangsadresse setzen	
,b50f	85	35	sta 35	LB der neu errechneten Adresse als LB des Stringhilfszeigers setzen	
,b511	84	36	sty 36	HB der neu errechneten Adresse als HB der Stringhilfszeigers setzen	
,b513	aa		tax	LB zwecks Rückgabe der Adresse in X/Y in X-Register bringen	
,b514	68		pla	auf Stapel gemerkte Stringlänge (s. \$b4f6) zurück in den Akku holen	
,b515	60		rts	Rücksprung von Routine	

; Garbage Collection durchführen, da nicht ausreichend Platz im Stringspeicher vorhanden ist					
,b516	a2	10	→ldx #10	Fehlermeldung OUT OF MEMORY durch Laden der Fehlernummer	
,b518	a5	0f	lda 0f	Garbage-Collection-Flag zwecks Test auslesen	
,b51a	30	b6	↖bmi b4d2	Flag auf "Garbage Collection bereits durchgeführt" gestellt (N=1): OUT Of MEMORY ERROR erzeugen (s. \$b516; bei \$b4d2 steht "jmp error")	
,b51c	20	26	b5 jsr b526 "garcol"	Routine für Garbage Collection ausführen	
,b51f	a9	80	lda #80 %10000000	Flag für "Garbage Collection bereits durchgeführt" laden	
,b521	85	0f	sta 0f	und in Garbage-Collection-Flag schreiben	
,b523	68		pla	bei \$b4f6 auf Stapel gemerkte Stringlänge wieder holen	
,b524	d0	d0	↖bne b4f6 "jmp"	Rücksprung in Routine zur Organisation des Speicherplatzes in Stringbereich; dort wird, falls der durch die Garbage Collection gewonnene Speicherplatz nicht ausreicht, ein OUT OF MEMORY ERROR ausgelöst	

; GARCOL-Routine: Garbage Collection (Neuorganisation des Stringspeichers, indem nicht mehr benötigte Strings (Strings ohne Descriptor im gültigen Variableneintrag) gelöscht werden; Aufruf von \$b51c oder \$b384 (FRE-Funktion)					
,b526	a6	37	ldx 37	LB des Zeigers auf die Obergrenze des Basic-RAM laden	} Basic-RAM-Ende als Obergrenze fürStringbereich setzen
,b528	a5	38	lda 38	HB des Zeigers auf die Obergrenze des Basic-RAM laden	
,b52a	86	33	stx 33	LB in LB des Zeigers auf Obergrenze des Stringbereichs schreiben	
,b52c	85	34	sta 34	HB in HB des Zeigers auf Obergrenze des Stringbereichs schreiben	
,b52e	a0	00	ldy #00	Initialisierungswert für Hilfszeiger laden	} Hilfszeiger \$4e/\$4f mit \$0000 initialisieren
,b530	84	4f	sty 4f	HB des Hilfszeigers \$4e/\$4f löschen	
,b532	84	4e	sty 4e	LB des Hilfszeigers \$4e/\$4f löschen	

,b534	a5 31	lda 31	LB des Zeigers auf die Endadresse des Arraybereichs laden	Endadresse des
,b536	a6 32	ldx 32	HB des Zeigers auf die Endadresse des Arraybereichs laden	Arraybereichs in
,b538	85 5f	sta 5f	LB in LB des Hilfszeigers \$5f/\$60 schreiben	Hilfszeiger
,b53a	86 60	stx 60	HB in HB des Hilfszeigers \$5f/\$60 schreiben	\$5f/\$60 schreiben
,b53c	a9 19	lda #19 <(\$0019)	LB der Anfangsadresse des temporären Stringstapels laden	Anfangsadresse des
,b53e	a2 00	ldx #00 >(\$0019)	HB der Anfangsadresse des temporären Stringstapels laden	temporären String-
,b540	85 22	sta 22	LB in LB des Hilfszeigers \$22/\$23 schreiben	stapels in Hilfs-
,b542	86 23	stx 23	HB in HB des Hilfszeigers \$22/\$23 schreiben	zeiger \$5f/\$60
,b544	c5 16	→cmp 16	Zeiger auf temporären Stringstapel mit \$19 (s. \$b53c) vergleichen	
,b546	f0 05	→beq b54d	Übereinstimmung, temporärer Stringstapel ist voll (Z=1): Suche beenden	
,b548	20 c7	b5 jsr b5c7	Hilfsroutine, um Stringadresse für GARCOL zu ermitteln	
,b54b	f0 f7	→beq b544 "jmp"	weitere Suche, bis der Vergleich bei \$b544 eine Übereinstimmung ergibt	

,b54d	a9 07	→lda #07	Länge eines Stringeintrags laden	} 7 als Länge eines
,b54f	85 53	sta 53	und in Hilfsspeicher \$53 merken	
,b551	a5 2d	lda 2d	LB der Anfangsadresse des Variablenspeichers laden	Anfangsadresse des
,b553	a6 2e	ldx 2e	HB der Anfangsadresse des Variablenspeichers laden	Variablenspeichers in
,b555	85 22	sta 22	LB in LB des Hilfszeigers \$22/\$23 schreiben	Hilfszeiger \$22/\$23
,b557	86 23	stx 23	HB in HB des Hilfszeigers \$22/\$23 schreiben	übertragen
,b559	e4 30	→cpx 30	HB des Hilfszeigers \$22/\$23 mit HB der Anfangsadresse der Arrays vergleichen	
,b55b	d0 04	→bne b561	keine Übereinstimmung (Z=0): kein LB-Vergleich	
,b55d	c5 2f	cmp 2f	LB des Hilfszeigers \$22/\$23 mit LB der Anfangsadresse der Arrays vergleichen	
,b55f	f0 05	→beq b566	Übereinstimmung (Z=1): Strings aus Array bearbeiten (Sonderfall)	
; Strings außerhalb von Array bearbeiten				
,b561	20 bd	b5 jsr b5bd	Hilfsroutine zum Test des aktuellen Strings auf Gültigkeit aufrufen (gleichzeitig rückt Hilfszeiger \$22/\$23 vor)	
,b564	f0 f3	→beq b559 "jmp"	weiter, bis Suche fertig ist	

; Strings aus Array bearbeiten				
,b566	85 58	→sta 58	LB der Adresse in LB von Hilfszeiger \$58/\$59 schreiben	
,b568	86 59	stx 59	HB der Adresse in HB von Hilfszeiger \$58/\$59 schreiben	
,b56a	a9 03	lda #03	Länge eines Stringeintrags (Array-Verhältnisse) laden	
,b56c	85 53	sta 53	und in Hilfsspeicher \$53 schreiben	
,b56e	a5 58	lda 58	LB des Hilfszeigers \$58/\$59 auslesen	
,b570	a6 59	ldx 59	HB des Hilfszeigers \$58/\$59 auslesen	
,b572	e4 32	cpx 32	HB mit HB der Endadresse der Arrays +1 vergleichen	
,b574	d0 07	→bne b57d	keine Übereinstimmung (Z=0): Suche fortsetzen, keine Verschiebung des Stringbereichs	

,b576	c5 31	cmp 31	LB mit LB der Endadresse der Arrays +1 vergleichen
,b578	d0 03	bne b57d	keine Übereinstimmung (Z=0): Suche fortsetzen, keine Verschiebung des Stringbereichs
,b57a	4c 06 b6	jmp b606	ungültigen String gefunden, durch Stringbereichsverschiebung entfernen
,b57d	85 22	sta 22	LB des Hilfszeigers \$22/\$23 neu setzen } Hilfszeiger \$22/\$23
,b57f	86 23	stx 23	HB des Hilfszeigers \$22/\$23 neu setzen } aktualisieren
,b581	a0 00	ldy #00	Offset mit 0 initialisieren (auf Variablenamen stellen)
,b583	b1 22	lda (22),y	Byte #1 des Variablenamen holen
,b585	aa	tax	und ins X-Register bringen
,b586	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Byte #1 des Variablenamen stellen)
,b587	b1 22	lda (22),y	Byte #2 des Variablenamen holen
,b589	08	php	Prozessorstatus nach Auslesen von Byte #2 auf den Stapel retten
,b58a	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf LB der Array-Länge stellen) } Hilfszeiger
,b58b	b1 22	lda (22),y	LB der Array-Länge holen } \$58/\$59
,b58d	65 58	adc 58	zu LB des Hilfszeigers \$58/\$59 addieren } um Array-Länge
,b58f	85 58	sta 58	und Ergebnis in LB von \$58/\$59 schreiben } erhöhen,
,b591	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf HB der Array-Länge stellen) } damit er auf
,b592	b1 22	lda (22),y	HB der Array-Länge holen } nächstes Array
,b594	65 59	adc 59	zu HB des Hilfszeigers \$58/\$59 addieren } gestellt
,b596	85 59	sta 59	und Ergebnis in HB von \$58/\$59 schreiben } wird
,b598	28	plp	bei \$b589 gemerkten CPU-Status nach Auslesen von Byte #2 des Variablenamen holen
,b599	10 d3	bpl b56e	b7 war gelöscht (N=0): mit neuem Hilfszeiger \$58/\$59 weiterarbeiten, da es sich um kein String-Array handelte
,b59b	8a	txa	Byte #1 des Variablenamen vom X-Register (s. \$b583/\$b585) in Akku holen
,b59c	30 d0	bmi b56e	b7 war gesetzt (N=1): mit neuem Hilfszeiger \$58/\$59 weiterarbeiten, da es sich um kein String-Array handelte
,b59e	c8	iny "ldy #04"	Offset von 3 auf 4 erhöhen (auf Dimension stellen)
,b59f	b1 22	lda (22),y	Dimension des Arrays holen
,b5a1	a0 00	ldy #00	Offset mit 0 initialisieren
,b5a3	0a	asl	Dimension (s. \$b59f) mit 2 multiplizieren
,b5a4	69 05	adc #05	Konstante 5 (Anzahl der Kopfbytes außer Dimensionswerten) addieren
,b5a6	65 22	adc 22	dieses Ergebnis zu LB des Hilfszeiger \$22/\$23 addieren
,b5a8	85 22	sta 22	und in LB von \$22/\$23 schreiben
,b5aa	90 02	bcc b5ae	kein Additionsübertrag (C=0): HB nicht erhöhen
,b5ac	e6 23	inc 23	HB von \$22/\$23 erhöhen, um Additionsübertrag zu berücksichtigen
,b5ae	a6 23	ldx 23	HB des Hilfszeigers \$22/\$23 holen
,b5b0	e4 59	cpx 59	und mit HB des Hilfszeigers \$58/\$59 schreiben
,b5b2	d0 04	bne b5b8	keine Übereinstimmung (Z=0): kein LB-Vergleich
,b5b4	c5 58	cmp 58	LB des Hilfszeigers \$22/\$23 (s. \$b5a8) mit LB des Hilfszeigers \$58/\$59 vergleichen
,b5b6	f0 ba	beq b572	Übereinstimmung (Z=1): weiter mit neuem Hilfszeiger

```
,b5b8 20 c7 b5↳jsr b5c7    Adresse des Strings ermitteln
,b5bb f0 f3      beq b5b0 "jmp" weiter in Array-Durchsuchung
-----
```

; Unterroutine zum Test, ob aktueller String gültig ist; Aufruf nur bei \$b561 (GARCOL)

```
,b5bd bl 22      lda (22),y    Byte #1 des Variablenamen holen
,b5bf 30-35-----bmi b5f6      b7 war gesetzt (N=1): Hilfszeiger für Suche vorrücken, da kein String gefunden wurde
,b5c1 c8         iny "ldy #01" Offset erhöhen (auf Byte #2 des Variablenamen stellen)
,b5c2 bl 22      lda (22),y    Byte #2 des Variablenamen holen
,b5c4 10-30-----bpl b5f6      b7 war gelöscht (N=0): Hilfszeiger für Suche erhöhen, da kein String gefunden wurde
,b5c6 c8         iny "ldy #02" Offset erhöhen (auf Stringlänge stellen)
```

; Einsprung: Ermittlung der Stringadresse; Aufruf bei \$b548 und \$b5bb (beide Aufrufe aus GARCOL)

```
,b5c7 bl 22      lda (22),y    Byte (Stringlänge) aus aktuellem Stringeintrag holen
,b5c9 f0-2b-----beq b5f6      Stringlänge = 0 (Z=1): Hilfszeiger $22/$23 auf nächsten String stellen, also Ende
,b5cb c8         iny "ldy #03" Offset erhöhen (auf LB der Startadresse stellen)
,b5cc bl 22      lda (22),y    LB der Startadresse des Strings holen
,b5ce aa         tax           und ins X-Register bringen
,b5cf c8         iny "ldy #04" Offset erhöhen (auf HB der Startadresse stellen)
,b5d0 bl 22      lda (22),y    HB der Startadresse des Strings holen
,b5d2 c5 34      cmp 34        HB der Stringadresse mit HB der Anfangsadresse der Stringspeicherung vergleichen
,b5d4 90 06-----bcc b5dc      HB der Stringadresse < HB der Anfangsadresse der Stringspeicherung (C=0): kein
                                LB-Vergleich
,b5d6 d0-1e-----bne b5f6      HB der Stringadresse > HB der Anfangsadresse der Stringspeicherung (Z=0): Hilfszeiger
                                $22/$23 auf nächsten String stellen
,b5d8 e4 33      cpx 33        LB der Stringadresse mit LB der Anfangsadresse der Stringspeicherung vergleichen
,b5da b0-1a-----bcs b5f6      LB der Stringadresse >= LB der Anfangsadresse der Stringspeicherung (C=0):
                                Hilfszeiger $22/$23 auf nächsten String stellen
```

; Stringadresse < Anfangsadresse für Stringspeicherung:

```
,b5dc c5 60----->cmp 60      HB der Stringadresse mit HB des Hilfszeigers $5f/$60 vergleichen
,b5de 90 16      bcc b5f6      Stringadresse < Adresse in $5f/$60 (C=0): Hilfszeiger $22/$23 auf nächsten String
                                stellen
,b5e0 d0 04-----bne b5e6      Stringadresse > Adresse in $5f/$60 (Z=0): kein LB-Vergleich
,b5e2 e4 5f      cpx 5f        LB der Stringadresse mit LB des Hilfszeigers $5f/$60 vergleichen
,b5e4 90-10-----bcc b5f6      Stringadresse < Adresse in $5f/$60 (C=0): Hilfszeiger $22/$23 auf nächsten String
                                stellen
```

; Stringadresse > Adresse in \$5f/\$60:

,b5e6	86 5f	↳stx 5f	LB des Zeigers \$5f/\$60 neu mit LB der Stringadresse belegen	} Stringadresse kommt in Zeiger \$5f/\$60
,b5e8	85 60	sta 60	HB des Zeigers \$5f/\$60 neu mit HB der Stringadresse belegen	
,b5ea	a5 22	lda 22	LB des Hilfszeigers \$22/\$23 holen	} aktueller Inhalt des Hilfszeigers \$22/\$23 kommt nach \$4e/\$4f
,b5ec	a6 23	ldx 23	HB des Hilfszeigers \$22/\$23 holen	
,b5ee	85 4e	sta 4e	LB in LB des Hilfszeigers \$4e/\$4f schreiben	
,b5f0	86 4f	stx 4f	HB in HB des Hilfszeigers \$4e/\$4f schreiben	
,b5f2	a5 53	lda 53	Offset zum nächsten Stringeintrag holen	
,b5f4	85 55	sta 55	und in Hilfsspeicher \$55 merken	
,b5f6	a5-53	→lda 53	Offset zum nächsten Stringeintrag holen	
,b5f8	18	clc	Carry vor Addition löschen	
,b5f9	65 22	adc 22	Offset zum LB des Hilfszeigers \$22/\$23 addieren	
,b5fb	85 22	sta 22	und Ergebnis in LB des Hilfszeigers schreiben	
,b5fd	90 02	bcc b601	kein Additionsübertrag (C=0): HB nicht erhöhen	
,b5ff	e6 23	inc 23	HB des Hilfszeigers \$22/\$23 erhöhen	
,b601	a6 23	→ldx 23	HB des Hilfszeigers in X-Register holen	
,b603	a0 00	ldy #00	Offset wieder initialisieren	
,b605	60	rts	Rücksprung von Routine	

; Sonderfall: Verschiebung des Stringbereichs zum Entfernen des aktuellen, ungültigen Strings (Aufruf nur von \$b57a)

,b606	a5 4f	lda 4f	HB des Zeigers \$4e/\$4f holen	
,b608	05 4e	ora 4e	mit LB des Zeigers \$4e/\$4f verknüpfen	
,b60a	f0 f5	↳beq b601	\$4e/\$4f stand auf \$0000 (Z=1): Rücksprung, da Verschiebung noch nicht zulässig	
,b60c	a5 55	lda 55	Hilfsspeicher für Länge eines Variableneintrags holen	
,b60e	29 04	and #04 %00000100	alle Bits bis auf b2 ausblenden	
,b610	4a	lsr	\$00 im Akku bleibt unverändert, \$04 wird zu \$02	
,b611	a8	tay	Ergebnis der Verschiebung ins Y-Register als Offset holen	
,b612	85 55	sta 55	Ergebnis in Hilfsspeicher \$55 schreiben	
,b614	b1 4e	lda (4e),y	Stringlänge anhand von Hilfszeiger \$4e/\$4f ermitteln	} Endadresse des Originalbereichs für Speicherverschiebung nach \$5a/\$5b berechnen
,b616	65 5f	adc 5f	zu LB des Hilfszeigers \$5f/\$60 addieren	
,b618	85 5a	sta 5a	und Ergebnis in LB von Hilfszeiger \$5a/\$5b schreiben	
,b61a	a5 60	lda 60	HB des Hilfszeigers \$5f/\$60 holen	
,b61c	69 00	adc #00	eventuellen Additionsübertrag berücksichtigen (s. \$b616)	
,b61e	85 5b	sta 5b	und Ergebnis in HB von Hilfszeiger \$5a/\$5b schreiben	
,b620	a5 33	lda 33	LB der Obergrenze des Stringbereichs holen	} Obergrenze des Stringbereichs als Endadresse des Zielbereichs für Speicher- blockverschiebung angeben
,b622	a6 34	ldx 34	HB der Obergrenze des Stringbereichs holen	
,b624	85 58	sta 58	LB in LB des Hilfszeigers \$58/\$59 schreiben	
,b626	86 59	stx 59	HB in HB des Hilfszeigers \$58/\$59 schreiben	


```
,b628 20 bf a3 jsr a3bf "bltuc" Speicherblockverschiebung des Stringbereichs ($5f/$60: Anfang des Quellbereichs,
                                $5a/$5b: Ende des Quellbereichs, $58/$59: Ende des Zielbereichs)
,b62b a4 55 ldy 55 Hilfsppeicher für Länge eines Variableneintrags auslesen
,b62d c8 iny Offset um 1 erhöhen (auf LB des Descriptors stellen)
,b62e a5 58 lda 58 LB der Endadresse des Zielbereichs holen
,b630 91 4e sta (4e),y und in LB des Descriptors schreiben
,b632 aa tax außerdem ins X-Register bringen
,b633 e6 59 inc 59 HB der Endadresse des Zielbereichs erhöhen
,b635 a5 59 lda 59 und auslesen
,b637 c8 iny Offset um 1 erhöhen (auf HB des Descriptors stellen)
,b638 91 4e sta (4e),y und in HB des Descriptors schreiben
,b63a 4c 2a b5 jmp b52a Rücksprung an Anfang der Garbage-Collection-Schleife
```

; Stringverknüpfung (Operator: "+")

```
,b63d a5 65 lda 65 HB der aktuellen Variablenadresse holen } aktuelle
,b63f 48 pha und auf den Stapel legen } Variablenadresse
,b640 a5 64 lda 64 LB der aktuellen Variablenadresse holen } auf dem Stapel
,b642 48 pha und auf den Stapel legen } merken
,b643 20 83 ae jsr ae83 "eval" Auswertung eines Ausdrucks (Unterprogramm von FRMEVL)
,b646 20 8f ad jsr ad8f "chkstr" Prüfroutine, ob es sich um Stringausdruck handelte
,b649 68 pla LB der Adresse des ersten Strings vom Stapel holen } Adresse des
,b64a 85 6f sta 6f und in LB des Hilfszeigers $6f/$70 schreiben } ersten Strings
,b64c 68 pla HB der Adresse des ersten Strings vom Stapel holen } in Hilfszeiger
,b64d 85 70 sta 70 und in HB des Hilfszeigers $6f/$70 schreiben } $6f/$70 ablegen
,b64f a0 00 ldy #00 Offset mit 0 initialisieren (auf Stringlänge stellen)
,b651 b1 6f lda (6f),y Stringlänge des ersten Strings holen
,b653 18 clc Carry vor Addition löschen
,b654 71 64 adc (64),y Stringlänge des zweiten Strings addieren
,b656 90 05 bcc b65d kein Übertrag, also wird verknüpfter String nicht länger als 255 Zeichen (C=0): kein
                                Fehler
,b658 a2 17 ldx #17 Fehlernummer für STRING TOO LONG laden } STRING TOO LONG ERROR auslösen,
,b65a 4c 37 a4 jmp a437 "error" Fehlereinsprung aufrufen } da String größer als 255 Zeichen
```

```
,b65d 20 75 b4 jsr b475 Stringeintrag von im Akku enthaltener Länge organisieren
,b660 20 7a b6 jsr b67a "strvar" String (in diesem Fall: String #1) in Variablenbereich übertragen
,b663 a5 50 lda 50 LB der Adresse des zweiten Stringdescriptors laden } zweiten
,b665 a4 51 ldy 51 HB der Adresse des zweiten Stringdescriptors laden } String
,b667 20 aa b6 jsr b6aa String auswerten } auswerten
,b66a 20 8c b6 jsr b68c in Kopierroutine einsteigen, so daß zweiter String hinter ersten geschrieben wird
```

,b66d	a5 6f	lda 6f	LB der Adresse des ersten Strings holen	} ersten String auswerten
,b66f	a4 70	ldy 70	HB der Adresse des ersten Strings holen	
,b671	20 aa b6	jsr b6aa	String auswerten	
,b674	20 ca b4	jsr b4ca	Aufnahme des Ergebnisstrings in den temporären Stringstapel	
,b677	4c b8 ad	jmp adb8	Rücksprung in FRMEVL-Routine	

; STRVAR-Routine: Kopieren eines Strings in den Stringspeicher;
Verwendung bei \$aa61 (Zuweisung an Stringvariable) und \$b660 (Stringverknüpfung)

,b67a	a0 00	ldy #00	Offset mit 0 initialisieren (auf Stringlänge stellen)
,b67c	b1 6f	lda (6f),y	Stringlänge holen
,b67e	48	pha	und auf den Stapel legen
,b67f	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf LB der Stringadresse stellen)
,b680	b1 6f	lda (6f),y	LB der Stringadresse holen
,b682	aa	tax	und ins X-Register bringen
,b683	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf HB der Stringadresse stellen)
,b684	b1 6f	lda (6f),y	HB der Stringadresse holen
,b686	a8	tay	und ins Y-Register bringen
,b687	68	pla	bei \$b67e gemerkte Stringlänge wieder vom Stapel holen

; bei \$b4c7 genutzter Einsprung: String an Adresse in X/Y kopieren, Länge steht im Akku

,b688	86 22	stx 22	LB der Stringadresse (s. \$b682) in LB des Hilfszeigers \$22/\$23 schreiben
,b68a	84 23	sty 23	HB der Stringadresse (s. \$b686) in LB des Hilfszeigers \$22/\$23 schreiben

; Einsprung: String an Adresse in \$22/\$23 kopieren, Länge steht im Akku; Offset in Y
Nutzung des Einsprungs von \$b66a (Sonderbehandlung für Strings aus Page 0 oder 2)

,b68c	a8	tay	Stringlänge (s. \$b687) als Offset (und zwecks Test) in Y-Register bringen
,b68d	f0 0a	beq b699	Stringlänge 0 (Z=1): Ende, Additionen bei \$b699-\$b6a0 führen zu keiner Veränderung
,b68f	48	pha	Stringlänge auf den Stapel legen
,b690	88	>dey	Offset dekrementieren
,b691	b1 22	lda (22),y	Byte aus Quellstring holen
,b693	91 35	sta (35),y	und in Stringspeicher schreiben
,b695	98	tya	Offset in Akku (zwecks Test)
,b696	d0 f8	bne b690	Offset noch nicht auf 0 heruntergezählt (Z=0): weiter in Kopierschleife
,b698	68	pla	bei \$b68f gemerkte Stringlänge vom Stapel holen
,b699	18	>clc	Carry vor Addition löschen
,b69a	65 35	adc 35	Stringlänge zum LB der Zieladresse addieren (s. \$b693)
,b69c	85 35	sta 35	und Ergebnis in LB des String-Hilfszeigers \$35/\$36 schreiben

,b69e	90 02	bcc b6a2	kein Additionsübertrag (C=0): HB nicht erhöhen
,b6a0	e6 36	inc 36	HB des String-Hilfszeigers erhöhen, um Additionsübertrag zu berücksichtigen
,b6a2	60	→rts	Rücksprung von Routine

; Routine: String aus Basic-Text weiterverarbeiten; Aufruf aus STRPAR bei \$b782

,b6a3 20 8f ad jsr ad8f "chkstr" Prüfroutine, ob es sich um einen Stringausdruck handelt, prüfen

; FRESTR-Einsprung: Nutzung von \$a9e0 (TI\$-Zuweisung), \$ab21 (STROUT), \$b034 (Stringvergleich), \$b381 (FRE)

,b6a6	a5 64	lda 64	LB der Variablenadresse holen	} Variablenadresse in Hilfszeiger
,b6a8	a4 65	ldy 65	HB der Variablenadresse holen	
,b6aa	85 22	sta 22	LB in LB des Hilfszeigers \$22/\$23 schreiben	} \$22/\$23 schreiben
,b6ac	84 23	sty 23	HB in HB des Hilfszeigers \$22/\$23 schreiben	
,b6ae	20 db b6	jsr b6db	Eintrag in temporärem Stringstapel löschen	
,b6b1	08	php	Prozessorstatus merken	
,b6b2	a0 00	ldy #00	Offset mit 0 initialisieren (auf Stringlänge stellen)	
,b6b4	b1 22	lda (22),y	Stringlänge holen	
,b6b6	48	pha	und auf den Stapel legen	
,b6b7	c8	iny "ldy 01"	Offset von 0 auf 1 erhöhen (auf LB der Stringadresse stellen)	} Stringadresse aus Descriptor nach X/Y holen
,b6b8	b1 22	lda (22),y	LB der Stringadresse holen	
,b6ba	aa	tax	und ins X-Register bringen	
,b6bb	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf HB der Stringadresse stellen)	
,b6bc	b1 22	lda (22),y	HB der Stringadresse holen	
,b6be	a8	tay	und ins X-Register bringen	
,b6bf	68	pla	Stringlänge (s. \$b6b6) vom Stapel in Akku holen	
,b6c0	28	plp	bei \$b6b1 geretteten Prozessorstatus (Zustand nach Löschen des Eintrags im temporären Stringstapel) wiederherstellen	
,b6c1	d0 13	bne b6d6	keine Übereinstimmung zwischen den Strings (Z=0): Stringadresse nach \$22/\$23	
,b6c3	c4 34	cpy 34	HB der Stringadresse mit HB der Obergrenze des Stringbereichs vergleichen	
,b6c5	d0 0f	bne b6d6	keine Übereinstimmung (Z=0): Stringadresse nach \$22/\$23	
,b6c7	e4 33	cpx 33	LB der Stringadresse mit LB der Obergrenze des Stringbereichs vergleichen	
,b6c9	d0 0b	bne b6d6	keine Übereinstimmung (Z=0): Stringadresse nach \$22/\$23	
,b6cb	48	pha	Stringlänge (s. \$b6bf) wieder auf den Stapel retten	
,b6cc	18	clc	Carry vor Addition löschen	} Obergrenze des Stringbereichs um Stringlänge erhöhen
,b6cd	65 33	adc 33	Stringlänge zum LB der Obergrenze des Stringbereichs addieren	
,b6cf	85 33	sta 33	und Ergebnis in LB der Obergrenze des Stringbereichs schreiben	
,b6d1	90 02	bcc b6d5	kein Additionsübertrag (C=0): HB nicht erhöhen	
,b6d3	e6 34	inc 34	HB der Obergrenze des Stringbereichs addieren	
,b6d5	68	→pla	bei \$b6cb auf den Stapel gerettete Stringlänge holen	

,b6d6	86 22	→stx	22	LB der Stringadresse in LB von \$22/\$23 schreiben	} Stringadresse in \$22/\$23 zurückgeben
,b6d8	84 23	sty	23	HB der Stringadresse in HB von \$22/\$23 schreiben	
,b6da	60	rts		Rücksprung von Routine	

; Routine zum Löschen eines Eintrags im temporären Stringstapel;
Nutzung bei \$aa6c (String-Zuweisung) und \$b6ae (FRESTR)

,b6db	c4 18	cpy	18	HB mit HB der letzten Stringadresse vergleichen	} RTS auslösen, wenn Stringadresse mit Adresse des letzten Strings identisch ist
,b6dd	d0 0c	bne	b6eb	keine Übereinstimmung (Z=0): RTS anspringen	
,b6df	c5 17	cmp	17	LB mit LB der letzten Stringadresse vergleichen	
,b6e1	d0 08	bne	b6eb	keine Übereinstimmung (Z=0): RTS anspringen	
,b6e3	85 16	sta	16	LB in Zeiger für temporären Stringstapel schreiben	
,b6e5	e9 03	sbc	#03	Länge eines Eintrags im temporären Stringstapel abziehen	
,b6e7	85 17	sta	17	und Ergebnis als LB der letzten Stringadresse setzen	
,b6e9	a0 00	ldy	#00	Z-Flag setzen (keine Bedeutung)	
,b6eb	60	→rts		Rücksprung von Routine	

; Routine zur Basic-Funktion CHR\$ (Token: \$c7)

,b6ec	20 a1 b7	jsr	b7a1	in GETBYT-Routine einsteigen, damit CHR\$()-Parameter als Bytewert nach X kommt	
,b6ef	8a	txa		an CHR\$() übergebenen Bytewert in Akku holen	} CHR\$-Parameter auf Stapel legen
,b6f0	48	pha		und auf den Stapel legen	
,b6f1	a9 01	lda	#01	Stringlänge = 1 laden, denn CHR\$() liefert 1-Byte-Strings	} Ergebnisstring von 1 Byte Länge in Stringspeicher erzeugen
,b6f3	20 7d b4	jsr	b47d	entsprechende Menge Speicherplatz im Stringspeicher schaffen	
,b6f6	68	pla		bei \$b6f0 gemerkten Bytewert in Akku holen	
,b6f7	a0 00	ldy	#00	Offset mit 0 initialisieren	
,b6f9	91 62	sta	(62),y	ermittelten Bytewert in angelegten String schreiben	
,b6fb	68	pla		LB der Rücksprungadresse am Stapel entfernen	} Rücksprungadresse vom Stapel löschen
,b6fc	68	pla		HB der Rücksprungadresse am Stapel entfernen	
,b6fd	4c ca b4	jmp	b4ca	Ergebnisstring in temporären Stringstapel übernehmen	

; Routine zur Basic-Funktion LEFT\$ (Token: \$c8)

,b700	20 61 b7	jsr	b761 "pream"	an Stringfunktion übergebenen String vom Stapel holen
				Y-Register ist danach auf 0 gestellt, Akku und X-Register enthalten numerischen
				LEFT\$-Parameter, \$50/\$51 zeigt auf Stringdescriptor
,b703	d1 50	cmp	(50),y	numerischen LEFT\$-Parameter und Länge von LEFT\$-String vergleichen
,b705	98	tya	"lda #00"	Akkumit 0 laden (s. \$b700)

; ab hier: auch bei RIGHT\$ verwendet (s. \$b734)

,b706	90 04	bcc b70c	numerischer LEFT\$-Parameter < Stringlänge (C=0): Ergebnisstring hat als numerischer Parameter übergebene Länge
,b708	b1 50	lda (50),y	Stringlänge holen, da LEFT\$-Parameter >= Stringlänge
,b70a	aa	tax	als Länge des Ergebnisstrings in X-Register bringen
,b70b	98	tya "lda #00"	Akku mit 0 laden (s. \$b700 bzw. \$b72c), da linker Teilstring 0 Byte umfaßt
,b70c	48	→pha	Akku (Länge des linken Teilstrings) auf den Stapel retten

; ab hier: auch bei MID\$ verwendet (s. \$b755)

,b70d	8a	→txa	ermittelte Länge des Ergebnisstrings in Akku holen
,b70e	48	→pha	Länge des Ergebnisstrings auf den Stapel retten
,b70f	20 7d b4	jsr b47d	Stringeintrag der im Akku enthaltenen Länge anlegen
,b712	a5 50	lda 50	LB der Adresse des Stringdescriptors holen
,b714	a4 51	ldy 51	HB der Adresse des Stringdescriptors holen
,b716	20 aa b6	jsr b6aa	in FRESTR-Routine einsteigen (String aus Basic-Text weiterverarbeiten)
,b719	68	pla	bei \$b70e gemerkte Länge des Ergebnisstrings holen
,b71a	a8	tay	und bis \$b725 im Y-Register zwischenspeichern
,b71b	68	pla	Länge des linken Teilstrings vom Stapel holen
,b71c	18	clc	Carry vor Addition löschen
,b71d	65 22	adc 22	LB des Hilfszeigers \$22/\$23 dazu addieren
,b71f	85 22	sta 22	und als LB des Hilfszeigers \$22/\$23 setzen
,b721	90 02	bcc b725	kein Übertrag (C=0): HB nicht erhöhen
,b723	e6 23	inc 23	HB des Hilfszeigers \$22/\$23 erhöhen
,b725	98	→tya	bei \$b71a zwischengespeicherte Länge des Ergebnisstrings in Akku holen
,b726	20 8c b6	jsr b68c	in STRVAR-Routine einsteigen: Ergebnisstring an Adresse in \$22/\$23 kopieren
,b729	4c ca b4	jmp b4ca	Stringdescriptor an temporären Stringstapel übergeben

; Routine zur Basic-Funktion RIGHT\$ (Token: \$c9)

,b72c	20 61 b7	jsr b761 "pream"	an Stringfunktion übergebenen String vom Stapel holen
			Y-Register ist danach auf 0 gestellt, Akku und X-Register enthalten numerischen RIGHT\$-Parameter, \$50/\$51 zeigt auf Stringdescriptor
,b72f	18	clc	Carry löschen, damit bei Subtraktion zusätzlich 1 abgezogen wird
,b730	f1 50	sbc (50),y	von numerischem RIGHT\$-Parameter wird hier die Länge des RIGHT\$-Strings subtrahiert
,b732	49 ff	eor #ff %11111111	komplementieren, da RIGHT\$ genau von anderer Seite als LEFT\$ den Teilstring bildet
,b734	4c 06 b7	jmp b706	in Routine für LEFT\$ einsteigen

; Routine zur Basic-Funktion MID\$ (Token: \$ca)

,b737	a9 ff	lda #ff %11111111	Default-Wert für dritten MID\$-Parameter laden, falls dieser fehlt
,b739	85 65	sta 65	und Ersatzwert in \$65 ablegen
,b73b	20 79 00	jsr 0079 "chrgot"	nächstes Zeichen aus Basic-Text holen
,b73e	c9 29	cmp #29	folgt ")" (geschlossene Klammer)?
,b740	f0 06	beq b748	ja (Z=1): keine Auswertung eines weiteren numerischen Parameters
,b742	20 fd ae	jsr aefd "chkcom"	auf Komma als syntaktisches Erfordernis prüfen
,b745	20 9e b7	jsr b79e "getbyt"	Bytewert (0-255) aus Basic-Text in X-Register holen
,b748	20 61 b7	jsr b761 "pream"	an Stringfunktion übergebenen String vom Stapel holen
			Y-Register ist danach auf 0 gestellt, Akku und X-Register enthalten numerischen
			RIGHT\$-Parameter, \$50/\$51 zeigt auf Stringdescriptor
,b74b	f0 4b	beq b798	zweiter MID\$-Parameter (Startposition im String) = 0 (Z=1): ILLEGAL QUANTITY erzeugen
,b74d	ca	dex	bei \$b745 ausgewerteten Bytewert verkleinern
,b74e	8a	txa	und in Akku bringen
,b74f	48	pha	von dort auf den Stapel legen
,b750	18	clc	Carry löschen, damit bei Subtraktion (s. \$b753) zusätzlich 1 subtrahiert wird
,b751	a2 00	ldx #00	0 als Vorbelegungswert für Länge des Ergebnisstrings laden
,b753	f1 50	sbc (50),y	von drittem MID\$-Parameter die Länge des MID\$-Strings abziehen
,b755	b0 b6	bcs b70d	kein Subtraktionsübertrag (C=1): in RIGHT\$/LEFT\$-Routine einsteigen
,b757	49 ff	eor #ff %11111111	Subtraktionsergebnis komplementieren
,b759	c5 65	cmp 65	Vergleich mit drittem MID\$-Parameter
,b75b	90 b1	bcc b70e	komplementiertes Subtraktionsergebnis < dritter MID\$-Parameter (C=0): in RIGHT\$/LEFT\$-Routine einsteigen
,b75d	a5 65	lda 65	dritten MID\$-Parameter als Ergebnis-Stringlänge laden
,b75f	b0 ad	bcs b70e "jmp"	in RIGHT\$/LEFT\$-Routine einsteigen

; PREAM-Routine: an Stringfunktion übergebenen und in Funktionsverteiler ausgewerteten String vom Stapel holen;
 Rückgabespeicher: \$50/\$51 zeigen auf Stringdescriptor, Akku und X-Register enthalten Stringlänge;
 Verwendung bei \$b700, \$b72c und \$b737 (LEFT\$, RIGHT\$, MID\$)

,b761	20 f7 ae	jsr aef7 "chkbcl"	Prüfroutine, ob ")" folgt, aufrufen
,b764	68	pla	LB der Rücksprungadresse von PREAM aus holen
,b765	a8	tay	und im Y-Register merken
,b766	68	pla	HB der Rücksprungadresse von PREAM aus holen
,b767	85 55	sta 55	und in \$55 merken
,b769	68	pla	LB der Rücksprungadresse aus aufrufender Routine vom Stapel löschen
,b76a	68	pla	HB der Rücksprungadresse aus aufrufender Routine vom Stapel löschen
,b76b	68	pla	Wert des im Funktionsverteiler ausgewerteten numerischen Parameters holen

,b76c	aa	tax	und in X-Register bringen	
,b76d	68	pla	LB der Adresse des Stringdescriptors holen	} Hilfszeiger \$50/\$51
,b76e	85 50	sta 50	und in LB von Hilfszeiger \$50/\$51 schreiben	
,b770	68	pla	HB der Adresse des Stringdescriptors holen	} auf Stringdescriptor richten
,b771	85 51	sta 51	und in HB von Hilfszeiger \$50/\$51 schreiben	
,b773	a5 55	lda 55	HB der Rücksprungadresse von PREAM aus \$55 holen (s. \$b767)	} Rücksprungadresse wieder aus Y/\$55 auf den Stapel zurücklegen
,b775	48	pha	und wieder auf den Stapel legen	
,b776	98	tya	LB der Rücksprungadresse von PREAM aus Y holen (s. \$b765)	
,b777	48	pha	und wieder auf den Stapel legen	
,b778	a0 00	ldy #00	Offset-Register Y mit 0 initialisieren (auf Stringlänge stellen)	
,b77a	8a	txa	als LEFT\$-Parameter angegebene Stringlänge (s. \$b76c) in X-Register holen	
,b77b	60	rts	Rücksprung von Routine	

; Routine zur Basic-Funktion LEN (Token: \$c3)

,b77c	20 82 b7	jsr b782 "strpar"	Parameter eines Strings, der an Funktionen wie LEN übergeben wurde, auswerten
,b77f	4c a2 b3	jmp b3a2 "bytfac"	mit Ergebnis in Y-Register in POS-Routine einsteigen

; STRPAR-Routine: Auswertung der Parameter eines Strings, der an die Funktionen LEN, ASC oder VAL übergeben wurde
(Funktionen, die einen String als Parameter erwarten, und einen numerischen Wert zurückgeben);
Verwendung bei \$b77c (LEN), \$b78b (ASC) und \$b7ad (VAL)

,b782	20 a3 b6	jsr b6a3	Weiterverarbeitung eines von Basic übergebenen Strings
,b785	a2 00	ldx #00	Datentyp-Flag "numerisch" laden
,b787	86 0d	stx 0d	und in Datentyp-Flag (String/numerisch) schreiben
,b789	a8	tay	Stringlänge in Y-Register bringen; gleichzeitig werden CPU-Flags gemäß Stringlänge gesetzt
,b78a	60	rts	Rücksprung von STRPAR-Routine

; Routine zur Basic-Funktion ASC (Token: \$c6)

,b78b	20 82 b7	jsr b782 "strpar"	Parameter eines Strings, der an Funktionen wie ASC übergeben wurde, auswerten
,b78e	f0 08	beq b798	Stringlänge = 0 (Z=1): ILLEGAL QUANTITY ERROR auslösen
,b790	a0 00	ldy #00	Offset mit 0 initialisieren (auf erstes Byte in String richten)
,b792	b1 22	lda (22),y	erstes Byte des Strings auslesen (\$22/\$23 zeigt auf Stringinhalt, nicht den Descriptor!)
,b794	a8	tay	und dieses Byte #1 als Ergebnis verwenden
,b795	4c a2 b3	jmp b3a2 "bytfac"	mit Ergebnis in Y-Register in POS-Routine einsteigen

,b798 4c 48 b2 > jmp b248 "illqua" ILLEGAL QUANTITY ERROR erzeugen

,b79b 20 73 00 jsr 0073 "chrget" nächstes Zeichen aus Basic-Text holen

; GETBYT-Routine: Auswertung eines im Basic-Text stehenden Bytewertes (0-255), der im X-Register zurückgegeben wird
Verwendung bei \$a94b, \$aa86, \$ab85, \$aba5, \$afc7, \$b745 und \$b7f4

,b79e 20 8a ad jsr ad8a "frmnum" numerischen Parameter als FLPT-Zahl in FAC #1 holen
 ,b7a1 20 b8 b1 jsr blb8 in andere Routine so einsteigen, daß ausgewerteter Parameter in Integerformat nach \$65/\$64 kommt
 ,b7a4 a6 64 ldx 64 HB zwecks Test auslesen
 ,b7a6 d0 f0 bne b798 HB <> 0, also gesamter Wert >\$ff (Z=0): ILLEGAL QUANTITY ERROR
 ,b7a8 a6 65 ldx 65 LB der Fließkomma-Integer-Wandlung holen
 ,b7aa 4c 79 00 jmp 0079 "chrgot" Zeichen an aktueller CHRGET-Zeiger-Position holen und Rücksprung

; Routine zur Basic-Funktion VAL (Token: \$c5)

,b7ad 20 82 b7 jsr b782 "strpar" Parameter eines Strings, der an Funktionen wie VAL übergeben wurde, auswerten
 ,b7b0 d0 03 bne b7b5 Stringlänge <> 0 (Z=0): keine Sonderbehandlung für VAL("")
 ,b7b2 4c f7 b8 jmp b8f7 MFLPT-Konstante 0 als Ergebnis nehmen und Routine beenden

,b7b5 a6 7a	>ldx 7a	LB des CHRGET-Zeigers holen	} CHRGET-Zeiger in Hilfszeiger \$71/\$72 retten
,b7b7 a4 7b	ldy 7b	HB des CHRGET-Zeigers holen	
,b7b9 86 71	stx 71	LB in LB des Hilfszeigers \$71/\$72 schreiben	
,b7bb 84 72	sty 72	HB in HB des Hilfszeigers \$71/\$72 schreiben	} CHRGET-Zeiger auf VAL-String richten, damit dieser über herkömmliche Routinen aus- wertbar ist. \$24/\$25 zeigen dann auf String- endadresse. Byte hinter String mit 0 belegen; alten Wert merken
,b7bd a6 22	ldx 22	LB der Adresse des VAL-Strings laden	
,b7bf 86 7a	stx 7a	und in LB des CHRGET-Zeigers schreiben	
,b7c1 18	clc	Carry vor Addition löschen	
,b7c2 65 22	adc 22	Stringlänge (seit \$b7ad im Akku) zu dieser Adresse addieren	
,b7c4 85 24	sta 24	und Ergebnis als LB von Hilfszeiger \$24/\$25 setzen	
,b7c6 a6 23	ldx 23	HB der Adresse des VAL-Strings laden	
,b7c8 86 7b	stx 7b	und in HB des CHRGET-Zeigers schreiben	
,b7ca 90 01	bcc b7cd	kein Übertrag bei \$b7c2 (C=0): HB für \$24/\$25 nicht erhöhen	
,b7cc e8	inx	HB erhöhen, um Additionsübertrag zu berücksichtigen	
,b7cd 86 25	>stx 25	HB des Hilfszeigers \$24/\$25 setzen	
,b7cf a0 00	ldy #00	Offset mit 0 initialisieren	
,b7d1 b1 24	lda (24),y	Byte hinter String auslesen	
,b7d3 48	pha	und auf den Stapel merken	
,b7d4 98	tya "lda #00"	Akku mit 0 belegen (s. \$b7cf)	

,b7d5	91 24	sta (24),y	und 0 hinter String schreiben (Endmarkierung für Auswertungsroutinen!)	
,b7d7	20 79 00	jsr 0079 "chrget"	Zeichen an aktueller CHRGET-Zeiger-Position holen	
,b7da	20 f3 bc	jsr bcf3 "strflp"	String in Fließkomma-Zahl umwandeln	
,b7dd	68	pla	bei \$b7d3 gemerktes Byte hinter String vom Stapel holen	} Byte hinter String
,b7de	a0 00	ldy #00	Offset mit 0 initialisieren	
,b7e0	91 24	sta (24),y	Byte hinter String mit altem Wert belegen	} wiederherstellen alten
,b7e2	a6 71	ldx 71	LB des geretteten CHRGET-Zeigers (s. \$b7b9) holen	
,b7e4	a4 72	ldy 72	HB des geretteten CHRGET-Zeigers (s. \$b7bb) holen	} CHRGET-Zeiger aus Hilfszeiger wiederherstellen
,b7e6	86 7a	stx 7a	LB in LB des CHRGET-Zeigers schreiben	
,b7e8	84 7b	sty 7b	HB in HB des CHRGET-Zeigers schreiben	
,b7ea	60	rts	Rücksprung von Routine	

; GETWRB-Routine: 2-Byte-Wert und davon durch Komma getrennten 1-Byte-Wert holen;
Verwendung bei \$b824 (POKE) und \$b82d (WAIT)

,b7eb	20 8a ad	jsr ad8a "frmnum"	numerischen Ausdruck auswerten
,b7ee	20 f7 b7	jsr b7f7 "facwrdr"	FAC in 2-Byte-Integerzahl (Wort) nach \$14/\$15 umwandeln

; GETCBT-Einsprung: durch Komma getrennten 1-Byte-Wert holen;
Verwendung bei \$b839 (WAIT)

,b7f1	20 fd ae	jsr aefd "chkcom"	auf Komma als syntaktisches Erfordernis prüfen
,b7f4	4c 9e b7	jmp b79e "getbyt"	Bytewert (0-255) aus Basic-Text auswerten und Rücksprung

; FACWRD-Routine: FAC #1 als 2-Byte-Integerwert (Wort) nach \$14/\$15 bringen;
Verwendung bei \$b7ee (GETWRB), \$b813 (PEEK) und \$e12d (SYS)

,b7f7	a5 66	lda 66	Vorzeichenbyte von FAC #1 auslesen	} FAC#1 nur mit Werten von 0 bis 65535 zulassen, sonst ILLEGAL QUANTITY ERROR hervorrufen
,b7f9	30 9d	↖bmi b798	FAC #1 ist negativ (N=1): ILLEGAL QUANTITY ERROR auslösen	
,b7fb	a5 61	lda 61	Exponentenbyte von FAC #1 auslesen	
,b7fd	c9 91	cmp #91	mit Exponent von 65536 vergleichen	
,b7ff	b0 97	↖bcs b798	FAC-Inhalt >= 65536 (C=1): ILLEGAL QUANTITY ERROR auslösen	
,b801	20 9b bc	jsr bc9b "facint"	FAC in Integerformat nach \$65/\$64 umwandeln	} Ergebnis der FACINT-Konvertierung nach \$14/\$15 bringen
,b804	a5 64	lda 64	HB des Ergebnisses der FACINT-Konvertierung holen	
,b806	a4 65	ldy 65	LB des Ergebnisses der FACINT-Konvertierung holen	
,b808	84 14	sty 14	LB nach \$14	
,b80a	85 15	sta 15	HB nach \$15	
,b80c	60	rts	Rücksprung von Routine	

; Routine zur Basic-Funktion PEEK (Token: \$c2)

,b80d	a5 15	lda 15	HB des Hilfszeigers \$14/\$15 holen	} Hilfszeiger \$14/\$15 auf dem Stapel merken
,b80f	48	pha	und auf den Stapel legen	
,b810	a5 14	lda 14	LB des Hilfszeigers \$14/\$15 holen	
,b812	48	pha	und auf den Stapel legen	
,b813	20 f7 b7	jsr b7f7 "facwrđ"	FAC #1 als 2-Byte-Integerwert (Wort) nach \$14/\$15 bringen	
,b816	a0 00	ldy #00	Offset mit 0 belegen, da kein Offset gewünscht wird	
,b818	b1 14	lda (14),y	Inhalt der PEEK-Adresse in Akku holen	} Inhalt der PEEK-Adresse als Funktionsergebnis holen
,b81a	a8	tay	und von dort ins Y-Register transportieren	
,b81b	68	pla	LB des Hilfszeigers \$14/\$15 vom Stapel holen	} Hilfszeiger \$14/\$15 von Stapel wiederherstellen (s. \$b80d-\$b812)
,b81c	85 14	sta 14	und in den Speicher übernehmen	
,b81e	68	pla	HB des Hilfszeigers \$14/\$15 vom Stapel holen	
,b81f	85 15	sta 15	und in den Speicher übernehmen	
,b821	4c a2 b3	jmp b3a2 "bytfac"	in POS-Routine einsteigen, um im Y-Register (s. \$b81a) befindliches Ergebnis als Fließkomma-Zahl im FAC #1 weiterzugeben	

; Routine zum Basic-Befehl POKE (Token: \$97)

,b824	20 eb b7	jsr b7eb "getwrb"	2-Byte-Integerwert (Wort) und davon durch Komma getrennten Bytewert (0-255) holen
,b827	8a	txa	Bytewert (zu POKEnde Wert) in Akku
,b828	a0 00	ldy #00	Offset mit 0 belegen, da kein Offset gewünscht wird
,b82a	91 14	sta (14),y	Bytewert (s. \$b827) an POKE-Adresse schreiben
,b82c	60	rts	Rücksprung von Routine

; Routine zum Basic-Befehl WAIT (Token: \$92)

,b82d	20 eb b7	jsr b7eb "getwrb"	2-Byte-Integerwert (Wort) und davon durch Komma getrennten Bytewert (0-255) holen
,b830	86 49	stx 49	Bytewert #1 in Hilfsspeicher \$49 ablegen
,b832	a2 00	ldx #00	Bytewert #2 auf 0 setzen, falls er nicht angegeben wird
,b834	20 79 00	jsr 0079 "chrgot"	letztes Zeichen aus Basic-Text holen
,b837	f0 03	beq b83c	Endmarkierung des Befehls (Z=1): mit Vorbelegung von Byte #2 (s. \$b832) fortfahren
,b839	20 f1 b7	jsr b7f1 "getcbt"	durch Komma abgegrenzten Bytewert holen
,b83c	86 4a	→stx 4a	Bytewert #2 in Hilfsspeicher \$4a ablegen
,b83e	a0 00	ldy #00	Offset mit 0 belegen, da kein Offset gewünscht wird
,b840	b1 14	→lda (14),y	Byte an WAIT-Adresse auslesen
,b842	45 4a	eor 4a	EOR-Verknüpfung mit Bytewert #2; ist Byte #2 = 0 (z.B. weil es nicht angegeben wurde), so ist dieser Befehl wirkungslos
,b844	25 49	and 49	AND-Verknüpfung mit Bytewert #1

```
,b846 f0 f8    beq b840    kein Bit nach Verknüpfung gesetzt (Z=1): weiter in Warteschleife
                        kleine Schwäche: hier fehlt eine Abfrage der STOP-Taste innerhalb der Schleife!
,b848 60        rts        Rücksprung von Routine
```

```
; ADD0.5-Routine: FAC #1 um 0.5 erhöhen
```

```
,b849 a9 11    lda #11 <($bfl1) LB der Adresse der ROM-Konstanten 0.5 laden
,b84b a0 bf    ldy #bf >($bfl1) HB der Adresse der ROM-Konstanten 0.5 laden
,b84d 4c 67 b8 jmp b867 "addmem" MFLPT-Konstante (in diesem Fall 0.5) zu FAC #1 addieren } MFLPT-Konstante 0.5
                                                    } (ab $bfl1 im ROM)
                                                    } zu FAC #1 addieren
```

```
; SUBMEM-Routine: FAC := MFLPT-Konstante - FAC
```

```
,b850 20 8c ba jsr ba8c "movma" im Speicher befindliche Konstante in ARG holen
```

```
; SUBFAC-Routine: FAC := ARG - FAC
```

```
,b853 a5 66    lda 66      Vorzeichenbyte von FAC #1 holen
,b855 49 ff    eor #ff %11111111 und invertieren (entspricht Multiplikation mit -1)
,b857 85 66    sta 66      invertiertes Vorzeichen zurückschreiben
,b859 45 6e    eor 6e      Vorzeichenbyte des ARG wird invertiert, wenn FAC #1 positiv war
,b85b 85 6f    sta 6f      als Vorzeichenbyte des Ergebnisses setzen
,b85d a5 61    lda 61      Exponentenbyte von FAC #1 laden
,b85f 4c 6a b8 jmp b86a "addfac" FAC und ARG addieren (FAC := FAC + ARG)
```

```
; EQUEXP-Routine: FAC #1 und ARG auf gleichen Exponenten bringen, dann addieren
```

```
,b862 20 99 b9 jsr b999 "shiftr" FAC #1 nach rechts schieben, bis der Exponent 0 erreicht ist
,b865 90 3c    bcc b8a3 "jmp" in Additionsroutine einsteigen
```

```
; ADDMEM-Routine: FAC := FAC + MFLPT-Konstante
```

```
,b867 20 8c ba jsr ba8c "movma" im Speicher befindliche Konstante in ARG holen
```

```
; ADDFAC-Einsprung: FAC := FAC + ARG
```

```
,b86a d0 03    bne b86f    FAC <> 0 (Z=0): keine Sonderbehandlung für FAC = 0 ausführen
,b86c 4c fc bb jmp bbfc "movaf" ARG in FAC kopieren, da FAC vorher den Wert 0 hatte und somit additionsneutral war
```

,b86f	a6 70	>ldx 70	Rundungsbyte auslesen
,b871	86 56	stx 56	und in Hilfsspeicher für altes Rundungsbyte schreiben
,b873	a2 69	ldx #69 *(\$69)	Adresse des ARG laden (Zeropage-Adresse, benötigt daher nur 1 Byte)
,b875	a5 69	lda 69	Exponentenbyte des ARG holen
,b877	a8	tay	und in Y-Register bringen
,b878	f0 ce	beq b848	ARG hat den Wert 0 (Z=1): RTS-Befehl anspringen, da 0 nicht addiert werden muß
,b87a	38	sec	Carry vor Subtraktion löschen
,b87b	e5 61	sbc 61	Exponentenbyte des FAC subtrahieren
,b87d	f0 24	beq b8a3	Exponentenbytes von FAC und ARG sind gleich (Z=1): Addition bei gleichen Exponenten
,b87f	90 12	bcc b893	Exponent von ARG < Exponent von FAC (C=0): Sonderbehandlung für ARG < FAC
,b881	84 61	sty 61	bei \$b877 in Y-Register gerettetes Exponentenbyte des ARG als Exponentenbyte des FAC setzen
,b883	a4 6e	ldy 6e	Vorzeichenbyte des ARG holen
,b885	84 66	sty 66	und in Vorzeichenbyte des FAC schreiben
,b887	49 ff	eor #ff %11111111	Differenz der Exponentenbytes (s. \$b875, \$b87b/\$b87d) invertieren
,b889	69 00	adc #00	1 addieren, da hier das Carry-Flag gesetzt ist (s. \$b87f)
,b88b	a0 00	ldy #00	0 als altes Rundungsbyte laden
,b88d	84 56	sty 56	und in Hilfsspeicher für altes Rundungsbyte schreiben
,b88f	a2 61	ldx #61 *(\$61)	Adresse des FAC laden (Zeropage-Adresse, benötigt daher nur 1 Byte)
,b891	d0 04	bne b897 "jmp"	Löschen des Rundungsbyte überspringen

,b893	a0 00	ldy #00	Löschwert 0 für Rundungsbyte laden
,b895	84 70	sty 70	und in Rundungsbyte schreiben
,b897	c9 f9	cmp #f9	Vergleich mit \$ff-\$07, um festzustellen, ob die ermittelte Differenz der Exponenten größer als 7 war
,b899	30 c7	bmi b862 "equexp"	Differenz > 7 (N=1): FAC und ARG auf gleichen Exponenten bringen, dann addieren
,b89b	a8	tay	Differenz ins Y-Register bringen
,b89c	a5 70	lda 70	Rundungsbyte des FAC laden
,b89e	56 01	lsr 01,x	Exponentenbyte des entsprechenden Fließkomma-Akkus (FAC oder ARG) nach rechts verschieben, um Rundung miteinzubeziehen
,b8a0	20 b0 b9	jsr b9b0 "rolshf"	FAC bitweise nach rechts verschieben
,b8a3	24 6f	bit 6f	Ergebnis-Vorzeichen (s. \$b85b) testen
,b8a5	10 57	bpl b8fe	positiv, also gleiche Vorzeichen (N=0): Sonderbehandlung für ungleiche Vorzeichen überspringen, statt dessen Mantissen-Addition durchführen

; Sonderbehandlung: FAC und ARG hatten ungleiche Vorzeichen

,b8a7	a0 61	ldy #61 *(\$61)	Zeropage-Adresse des FAC als Offset laden
,b8a9	e0 69	cpx #69 *(\$69)	X-Offset mit Zeropage-Adresse des FAC vergleichen
,b8ab	f0 02	beq b8af	Übereinstimmung (Z=1): Y-Offset nicht auf ARG stellen
,b8ad	a0 69	ldy #69 *(\$69)	Y-Offset auf ARG stellen

; hier ist entweder X=\$61 und Y=\$69 oder X=\$69 und Y=\$61. X- und Y-Offset zeigen somit jeder auf einen anderen der beiden Fließkomma-Akkumulatoren FAC und ARG.

,b8af	38		↳sec	Carry vor bei \$b8b2 simulierter Subtraktion setzen	
,b8b0	49 ff		eor #ff %l1l1l1l1l1	bei \$b89c geladenes und bei \$b8a0 rechtsverschobenes Rundungsbyte des FAC zwecks Subtraktionssimulation invertieren	
,b8b2	65 56		adc 56 "sbc"	simulierte Subtraktion (Inhalt von \$56 - Akku); \$56 enthält das letzte Rundungsbyte (s. \$b871, \$b88d)	
,b8b4	85 70		sta 70	Ergebnis als neues FAC-Rundungsbyte setzen	
,b8b6	b9 04 00		lda 0004,y	4. Mantissenbyte des einen Fließkomma-Akku (FAC/ARG) laden	} Sub- traktion der Mantis- sen- bytes von FAC und ARG Ergebnis in Man- tisse des FAC
,b8b9	f5 04		sbc 04,x	davon 4. Mantissenbyte des anderen Fließkomma-Akku (FAC/ARG) subtrahieren	
,b8bb	85 65		sta 65	und Ergebnis in FAC #1 als 4. Mantissenbyte setzen	
,b8bd	b9 03 00		lda 0003,y	3. Mantissenbyte des einen Fließkomma-Akku (FAC/ARG) laden	
,b8c0	f5 03		sbc 03,x	davon 3. Mantissenbyte des anderen Fließkomma-Akku (FAC/ARG) subtrahieren	
,b8c2	85 64		sta 64	und Ergebnis in FAC #1 als 3. Mantissenbyte setzen	
,b8c4	b9 02 00		lda 0002,y	2. Mantissenbyte des einen Fließkomma-Akku (FAC/ARG) laden	
,b8c7	f5 02		sbc 02,x	davon 2. Mantissenbyte des anderen Fließkomma-Akku (FAC/ARG) subtrahieren	
,b8c9	85 63		sta 63	und Ergebnis in FAC #1 als 2. Mantissenbyte setzen	
,b8cb	b9 01 00		lda 0001,y	1. Mantissenbyte des einen Fließkomma-Akku (FAC/ARG) laden	
,b8ce	f5 01		sbc 01,x	davon 1. Mantissenbyte des anderen Fließkomma-Akku (FAC/ARG) subtrahieren	
,b8d0	85 62		sta 62	und Ergebnis in FAC #1 als 1. Mantissenbyte setzen	
,b8d2	b0 03		↳ bcs b8d7 "normal"	kein Subtraktionsübertrag (C=1): FAC normalisieren	
,b8d4	20 47 b9		↳ jsr b947	FAC einschließlich Vorzeichenbyte invertieren	

; NORMAL-Einsprung: Normalisierung des FAC

,b8d7	a0 00		↳ ldy #00	Y-Offset mit 0 initialisieren
,b8d9	98		tya "lda #00"	Zähler für Bitverschiebung mit 0 initialisieren
,b8da	18		clc	Carry vor Addition bei \$b8f1 oder \$b9ld löschen
,b8db	a6 62		↳ ldx 62	Mantisse #1 auslesen
,b8dd	d0 4a		↳ bne b929	Mantisse #1 <> 0 (Z=0): bitweise Linksverschiebung des FAC

; byteweise Linksverschiebung des FAC

,b8df	a6 63		ldx 63	Mantisse #2 auslesen	} Mantissenbytes an jeweils "linke" (= höherwertige) Position
,b8e1	86 62		stx 62	und in Mantisse #1 schreiben	
,b8e3	a6 64		ldx 64	Mantisse #3 auslesen	
,b8e5	86 63		stx 63	und in Mantisse #2 schreiben	
,b8e7	a6 65		ldx 65	Mantisse #4 auslesen	
,b8e9	86 64		stx 64	und in Mantisse #3 schreiben	

,b8eb	a6 70	ldx 70	Rundungsbyte des FAC holen	} schieben, Rundungsbyte einbinden
,b8ed	86 65	stx 65	und in Mantisse #4 schreiben	
,b8ef	84 70	sty 70	Rundungsbyte des FAC neu setzen (bei vorheriger Ausführung von \$b8d7: Löschen)	
,b8f1	69 08	adc #08	Anzahl der verschobenen Bits (1 Byte = 8 Bit) zu Bitzähler addieren	
,b8f3	c9 20	cmp #20	schon 32 Bit (= 4 Byte) verschoben?	
,b8f5	d0 e4	bne b8db	nein (Z=0): weiter mit neuem Bitzähler (im Akku)	

; hier: Aufruf von \$b7b2 und \$badc

,b8f7	a9 00	→ lda #00	gewünschten Inhalt für Exponenten- und Vorzeichenbyte des FAC laden
,b8f9	85 61	sta 61	Exponentenbyte des FAC nach erfolgter Normalisierung mit 0 belegen
,b8fb	85 66	sta 66	Vorzeichenbyte des FAC löschen
,b8fd	60	rts	Rücksprung von Routine

; Mantissen-Addition, wenn FAC und ARG gleiches Vorzeichen haben

Im Akku wird das Rundungsbyte des FAC erwartet

,b8fe	65 56	adc 56	altes Rundungsbyte (s. \$b871, \$b88d) addieren	} gleichwertige Mantissenbytes von FAC und ARG zueinander addieren
,b900	85 70	sta 70	und Ergebnis als neues Rundungsbyte des FAC setzen	
,b902	a5 65	lda 65	Mantisse #4 des FAC holen	
,b904	65 6d	adc 6d	dazu Mantisse #4 des ARG addieren	
,b906	85 65	sta 65	und Ergebnis in Mantisse #4 des FAC schreiben	
,b908	a5 64	lda 64	Mantisse #3 des FAC holen	
,b90a	65 6c	adc 6c	dazu Mantisse #3 des ARG addieren	
,b90c	85 64	sta 64	und Ergebnis in Mantisse #3 des FAC schreiben	
,b90e	a5 63	lda 63	Mantisse #2 des FAC holen	
,b910	65 6b	adc 6b	dazu Mantisse #2 des ARG addieren	
,b912	85 63	sta 63	und Ergebnis in Mantisse #2 des FAC schreiben	} zueinander addieren
,b914	a5 62	lda 62	Mantisse #1 des FAC holen	
,b916	65 6a	adc 6a	dazu Mantisse #1 des ARG addieren	
,b918	85 62	sta 62	und Ergebnis in Mantisse #1 des FAC schreiben	
,b91a	4c 36 b9	jmp b936 "squeeze"	Verschiebung der Überlaufstelle, falls Additionsübertrag entstand	

; Routine zur bitweisen Linksverschiebung des FAC

,b91d	69 01	→ adc #01	1 zum Vorzeichenbyte addieren; bei Übertrag gesetztes Carry wird bei \$b91f beachtet	
,b91f	06 70	asl 70	Vorzeichenbyte des FAC nach links verschieben	
,b921	26 65	rol 65	Mantisse #4 nach links verschieben	} alle 4 Mantissenbytes linksverschieben
,b923	26 64	rol 64	Mantisse #3 nach links verschieben	
,b925	26 63	rol 63	Mantisse #2 nach links verschieben	
,b927	26 62	rol 62	Mantisse #1 nach links verschieben	

```
,b929 10 f2  Lbpl b91d      b7 im Ergebnis gelöscht (N=0): weiter in Verschiebeschleife
,b92b 38      sec          Carry vor Subtraktion setzen
,b92c e5 61   sbc    61     Exponentenbyte des FAC #1 subtrahieren
,b92e b0 c7   bcs    b8f7   kein Subtraktionsübertrag (C=1): Exponenten- und Vorzeichenbyte des FAC löschen
,b930 49 ff   eor    #ff %11111111 Ergebnis invertieren
,b932 69 01   adc    #01    dann 1 addieren
,b934 85 61   sta    61     und Ergebnis in Exponentenbyte des FAC #1 schreiben
```

; SQUEEZ-Einsprung (von \$b91a aus genutzt): Verschiebung der Überlaufstelle, falls Additionsübertrag entstand

```
,b936 90 0e   bcc    b946   kein Additionsübertrag (C=0): RTS anspringen
,b938 e6 61   inc     61     Exponentenbyte des FAC um 1 erhöhen
,b93a f0 42   beq    b97e   Exponent wurde von $ff (maximaler Wert) noch erhöht (Z=1): OVERFLOW ERROR ausgeben
,b93c 66 62   ror     62     Mantisse #1 des FAC rechtsverschieben
,b93e 66 63   ror     63     Mantisse #2 des FAC rechtsverschieben
,b940 66 64   ror     64     Mantisse #3 des FAC rechtsverschieben
,b942 66 65   ror     65     Mantisse #4 des FAC rechtsverschieben
,b944 66 70   ror     70     Rundungsbyte des FAC rechtsverschieben
,b946 60      rts          Rücksprung von Routine
```

}

Mantissenbytes
und
Rundungsbyte
des FAC
rechtsverschieben

; Invertierung der FAC-Mantisse: FAC := - FAC

```
,b947 a5 66   lda     66     Vorzeichenbyte des FAC holen
,b949 49 ff   eor    #ff %11111111 Vorzeichen invertieren
,b94b 85 66   sta     66     und als neues Vorzeichenbyte des FAC setzen
,b94d a5 62   lda     62     Mantisse #1 des FAC holen
,b94f 49 ff   eor    #ff %11111111 Mantisse #1 invertieren
,b951 85 62   sta     62     und als Mantisse #1 des FAC setzen
,b953 a5 63   lda     63     Mantisse #2 des FAC holen
,b955 49 ff   eor    #ff %11111111 Mantisse #2 invertieren
,b957 85 63   sta     63     und als Mantisse #2 des FAC setzen
,b959 a5 64   lda     64     Mantisse #3 des FAC holen
,b95b 49 ff   eor    #ff %11111111 Mantisse #3 invertieren
,b95d 85 64   sta     64     und als Mantisse #3 des FAC setzen
,b95f a5 65   lda     65     Mantisse #4 des FAC holen
,b961 49 ff   eor    #ff %11111111 Mantisse #4 invertieren
,b963 85 65   sta     65     und als Mantisse #4 des FAC setzen
,b965 a5 70   lda     70     Rundungsbyte des FAC holen
,b967 49 ff   eor    #ff %11111111 Rundungsbyte invertieren
,b969 85 70   sta     70     und als Rundungsbyte des FAC setzen
```

}

Invertierung
des
Vorzeichenbytes
und
der 4
Mantissen-
bytes
des FAC
sowie
des
Rundungs-
bytes
durch
EOR-Verknüpfung
mit \$ff
(= %11111111)
zur Negation
der Bytes

,b96b	e6 70	inc 70	Rundungsbyte des FAC erhöhen	} Erhöhung des Rundungs- byte und der 4 Mantissen- bytes
,b96d	d0 0e	bne b97d	kein Erhöhungsübertrag (Z=0): RTS anspringen	
,b96f	e6 65	inc 65	Mantisse #4 des FAC erhöhen	
,b971	d0 0a	bne b97d	kein Erhöhungsübertrag (Z=0): RTS anspringen	
,b973	e6 64	inc 64	Mantisse #3 des FAC erhöhen	
,b975	d0 06	bne b97d	kein Erhöhungsübertrag (Z=0): RTS anspringen	
,b977	e6 63	inc 63	Mantisse #2 des FAC erhöhen	
,b979	d0 02	bne b97d	kein Erhöhungsübertrag (Z=0): RTS anspringen	
,b97b	e6 62	inc 62	Mantisse #1 des FAC erhöhen	
,b97d	60	→rts	Rücksprung von Routine	

; Einsprung für OVERFLOW ERROR

,b97e	a2 0f	ldx #0f	Fehlernummer für OVERFLOW laden
,b980	4c 37 a4	jmp a437 "error"	und Fehlereinsprung aufrufen

; byteweise Rechtsverschiebung eines Fließkomma-Akkumulators von \$0026 bis \$002a (bei Multiplikation verwendet)
durch Einsprung bei \$b985 mit anderem Offset im X-Register (Offset = Adresse - 1) universell einsetzbar
Im Akku wird Initialisierungswert für Bitzähler erwartet, wobei es sich um ein Faktor-Byte handelt.

,b983	a2 25	ldx #25 (\$0026-1)	Offset auf RES (Resultats-Fließkomma-Akku für Multiplikation) laden
-------	-------	--------------------	---

; hier: Einsprung mit beliebigem Offset möglich

,b985	b4 04	→ldy 04,x	Mantisse #4 holen	} Mantissenbytes an jeweils "rechte" (=niederwertige) Position schreiben (unterste Mantisse in Rundungsbyte)
,b987	84 70	sty 70	und in Rundungsbyte des FAC #1 schreiben	
,b989	b4 03	ldy 03,x	Mantisse #3 holen	
,b98b	94 04	sty 04,x	und in Mantisse #4 schreiben	
,b98d	b4 02	ldy 02,x	Mantisse #2 holen	
,b98f	94 03	sty 03,x	und in Mantisse #3 schreiben	
,b991	b4 01	ldy 01,x	Mantisse #1 holen	
,b993	94 02	sty 02,x	und in Mantisse #2 schreiben	
,b995	a4 68	ldy 68	Überlaufbyte des FAC #1 holen	
,b997	94 01	sty 01,x	und in Mantisse #1 schreiben	

; SHIFTR-Einsprung

,b999	69 08	adc #08	Bitzähler um Anzahl der verschobenen Bits (1 Byte = 8 Bit) erhöhen
,b99b	30 e8	bmi b985	Bitzähler liegt noch im negativen Bereich (N=1): zurück an Verschiebeschleifenbeginn

,b99d	f0 e6	└ beq b985	Bitzähler auf 0 erhöht (Z=1): zurück an Verschiebeschleifenbeginn
,b99f	e9 08	sbcb #08	Addition bei \$b999 wieder rückgängig machen
,b9a1	a8	tay	Bitzähler in Y-Register merken
,b9a2	a5 70	lda 70	Rundungsbyte des FAC holen
,b9a4	b0 14	└ bcs b9ba	kein Subtraktionsübertrag bei \$b99f (C=1): Carry löschen und RTS-Rücksprung
,b9a6	16 01	└> asl 01,x	Mantisse #1 des zu verschiebenden Fließkomma-Akkumulators verdoppeln
,b9a8	90 02	└ bcc b9ac	b7 war vor Verschieben gelöscht (C=0): Erhöhen von Mantisse #1 ist überflüssig
,b9aa	f6 01	└ inc 01,x	Mantisse #1 des zu verschiebenden Fließkomma-Akkumulators erhöhen
,b9ac	76 01	└> ror 01,x	Verdoppelung bei \$b9a6 rückgängig machen
,b9ae	76 01	ror 01,x	Rechtsverschiebung von Mantisse #1

; ROLSHF-Einsprung

,b9b0	76 02	ror 02,x	Rechtsverschiebung von Mantisse #2
,b9b2	76 03	ror 03,x	Rechtsverschiebung von Mantisse #3
,b9b4	76 04	ror 04,x	Rechtsverschiebung von Mantisse #4
,b9b6	6a	ror	Rundungsbyte des FAC (s. \$b9a2) rechtsverschieben
,b9b7	c8	iny	Bitzähler (s. \$b9a1) um 1 erhöhen (um 1 Bit wurde nach rechts geschoben)
,b9b8	d0 ec	└ bne b9a6	noch nicht auf 0 gezählt (Z=0): weiter in bitweiser Rechtsverschiebung
,b9ba	18	└> clc	Carry für aufrufende Routine löschen
,b9bb	60	rts	Rücksprung von Routine

; MFLPT-Konstante 1 für die Routine zur LOG-Funktion

,b9bc	81 00 00 00 00	MFLPT-Darstellung der Zahl 1
-------	----------------	------------------------------

; Polynom für die Routine zur LOG-Funktion

,b9c1	03	Polynomgrad 3 als Bytewert
,b9c2	7f 5e 56 cb 79	Koeffizient #3 im MFLPT-Format: a3 = 0.434255942
,b9c7	80 13 9b 0b 64	Koeffizient #2 im MFLPT-Format: a2 = 0.576584541
,b9cc	80 76 38 93 16	Koeffizient #1 im MFLPT-Format: a1 = 0.961800750
,b9d1	82 38 aa 3b 20	Koeffizient #0 im MFLPT-Format: a0 = 2.885390070

; MFLPT-Konstanten für die Routine zur LOG-Funktion

,b9d6	80 35 04 f3 34	MFLPT-Darstellung der Zahl 0.707106781: $\text{SQR}(2)^{\uparrow}(-1)$
,b9db	81 35 04 f3 34	MFLPT-Darstellung der Zahl 1.41421356 : $\text{SQR}(2)$

```

;b9e0 80 80 00 00 00      MFLPT-Darstellung der Zahl -.5      : (-2)↑(-1)
;b9e5 80 31 72 17 f8      MFLPT-Darstellung der Zahl 0.693147181: LOG(2)

```

```

; Routine zur Basic-Funktion LOG (Token: $bc)

```

```

;b9ea 20 2b bc  jsr bc2b "sign"  Vorzeichenbyte des FAC #1 in Akku holen und dann testen
;b9ed f0 02      beq b9f1        FAC enthält 0 (Z=1): ILLEGAL QUANTITY ERROR auslösen, da LOG(0) unerlaubt ist
;b9ef 10 03      bpl b9f4        FAC ist positiv (N=0): keinen ILLEGAL QUANTITY ERROR auslösen
;b9f1 4c 48      b2->jmp b248 "illqua" ILLEGAL QUANTITY ERROR auslösen
-----
;b9f4 a5 61      lda 61          Exponentenbyte des FAC holen
;b9f6 e9 7f      sbc #7f        $80-1 da das Carry-Flag seit $b9ea gesetzt ist, wird hier $80 subtrahiert; dies ist
                                   wiederum der im Exponent enthaltene Offset, der hier entfernt wird
;b9f8 48         pha           Subtraktionsergebnis auf dem Stapel merken
;b9f9 a9 80      lda #80 %100000000 Exponentenbyte für Bereich [0.5;1[ laden } FAC-Inhalt in Bereich
;b9fb 85 61      sta 61          und in FAC-Exponentenbyte schreiben } für Näherungspolynom bringen
;b9fd a9 d6      lda #d6 <($b9d6) LB der Adresse der ROM-Konstanten SQR(2)↑(-1) laden } FAC um
;b9ff a0 b9      ldy #b9 >($b9d6) HB der Adresse der ROM-Konstanten SQR(2)↑(-1) laden } SQR(2)↑(-1)
;ba01 20 67 b8  jsr b867 "addmem" MFLPT-Konstante addieren; in diesem Fall: SQR(2)↑(-1) } erhöhen
;ba04 a9 db      lda #db <($b9db) LB der Adresse der ROM-Konstanten SQR(2) laden } FAC mit
;ba06 a0 b9      ldy #b9 >($b9db) HB der Adresse der ROM-Konstanten SQR(2) laden } SQR(2)/FAC
;ba08 20 0f bb  jsr bb0f "divmem" Division von Konstante durch FAC } laden
;ba0b a9 bc      lda #bc <($b9bc) LB der Adresse der ROM-Konstanten 1 laden } FAC mit
;ba0d a0 b9      ldy #b9 >($b9bc) HB der Adresse der ROM-Konstanten 1 laden } 1-FAC
;ba0f 20 50 b8  jsr b850 "submem" FAC von MFLPT-Konstante subtrahieren } laden
;ba12 a9 c1      lda #c1 <($b9c1) LB der Adresse der Polynomtabelle laden } Auswertung der
;ba14 a0 b9      ldy #b9 >($b9c1) HB der Adresse der Polynomtabelle laden } Polynomtabelle
;ba16 20 43 e0  jsr e043 "polyx" Polynom auswerten } ab $b9c1
;ba19 a9 e0      lda #e0 <($b9e0) LB der MFLPT-Konstanten -.5 laden } FAC um -.5 erhöhen,
;balb a0 b9      ldy #b9 >($b9e0) HB der MFLPT-Konstanten -.5 laden } also wird .5 vom FAC
;bald 20 67 b8  jsr b867 "addmem" MFLPT-Konstante addieren; in diesem Fall: -.5 } abgezogen
;ba20 68         pla           bei $b9f8 gemerktes Subtraktionsergebnis vom Stapel holen
;ba21 20 7e bd  jsr bd7e "addafc" Akku als Mantisse #1 in FAC integrieren, dann FAC verdoppeln (FAC:=FAC+FAC)
;ba24 a9 e5      lda #e5 <($b9e5) LB der Adresse der ROM-Konstanten LOG(2) laden } LOG(2) als
;ba26 a0 b9      ldy #b9 >($b9e5) HB der Adresse der ROM-Konstanten LOG(2) laden; } Faktor laden
                                   im Speicher folgt der Einsprung zur MEMMULT-Routine; also: FAC := FAC * LOG(2)

```

```

; MEMMULT-Routine:          FAC := FAC * Konstante

```

```

;ba28 20 8c ba  jsr ba8c "movma" Konstante (2. Faktor) in ARG holen

```

```

; MULT-Routine:                FAC := FAC * ARG

,ba2b d0 03  | bne ba30      FAC <> 0 (Z=0): keine Sonderbehandlung für FAC:= 0 * ARG, also FAC := 0 überspringen
,ba2d 4c 8b ba | jmp ba8b      RTS anspringen, da Multiplikation von 0 mit anderem Wert nur 0 ergeben kann, und
                        dieser Wert steht ja dann bereits im FAC (s. $ba2b)
-----
,ba30 20 b7 ba |>jsr bab7      Exponenten addieren
,ba33 a9 00     | lda #00      Initialisierungswert für RES (Ergebnis-Akku) laden
,ba35 85 26     | sta 26      Mantisse #1 des RES löschen
,ba37 85 27     | sta 27      Mantisse #2 des RES löschen
,ba39 85 28     | sta 28      Mantisse #3 des RES löschen
,ba3b 85 29     | sta 29      Mantisse #4 des RES löschen
,ba3d a5 70     | lda 70      Rundungsbyte des FAC holen
,ba3f 20 59 ba | jsr ba59 "mltply" Multiplikation für Rundungsbyte, Ergebnis zu RES addieren
,ba42 a5 65     | lda 65      Mantisse #4 des FAC holen
,ba44 20 59 ba | jsr ba59 "mltply" Multiplikation für Mantisse #4, Ergebnis zu RES addieren
,ba47 a5 64     | lda 64      Mantisse #3 des FAC holen
,ba49 20 59 ba | jsr ba59 "mltply" Multiplikation für Mantisse #3, Ergebnis zu RES addieren
,ba4c a5 63     | lda 63      Mantisse #2 des FAC holen
,ba4e 20 59 ba | jsr ba59 "mltply" Multiplikation für Mantisse #2, Ergebnis zu RES addieren
,ba51 a5 62     | lda 62      Mantisse #1 des FAC holen
,ba53 20 5e ba | jsr ba5e      in MLTPLY-Routine einsteigen: Multiplikation für Mantisse #1, Ergebnis zu RES
,ba56 4c 8f bb | jmp bb8f "movrf" RES in FAC übertragen, dann FAC normalisieren
-----
; MLTPLY-Routine: Multiplikation des Akku mit dem ARG, Ergebnis in RES

,ba59 d0 03  | bne ba5e      ein zu multiplizierendes Byte <> 0 (Z=0): keine Rechtsverschiebung, sondern
,ba5b 4c 83 b9 | jmp b983      Multiplikation
                        byteweise Rechtsverschiebung des RES
-----
,ba5e 4a      |>lsr      b0 in Carry-Flag holen
,ba5f 09 80     | ora #80 %10000000 b7 im Akku setzen
,ba61 a8      |>tay      und Ergebnis im Y-Register merken
,ba62 90 19     | bcc ba7d      b0 ist nach $ba5e gelöscht (C=0): RES um 1 Bit nach rechts verschieben
,ba64 18      | clc      Carry vor Addition bei $ba67 löschen
,ba65 a5 29     | lda 29      Mantisse #4 des RES holen
,ba67 65 6d     | adc 6d      dazu Mantisse #4 des ARG addieren
,ba69 85 29     | sta 29      und Ergebnis in Mantisse #4 des RES schreiben
,ba6b a5 28     | lda 28      Mantisse #3 des RES holen
,ba6d 65 6c     | adc 6c      dazu Mantisse #3 des ARG addieren
,ba6f 85 28     | sta 28      und Ergebnis in Mantisse #3 des RES schreiben

```

```

; MLTPLY-Routine: Multiplikation des Akku mit dem ARG, Ergebnis in RES

```

```

,ba59 d0 03  | bne ba5e      ein zu multiplizierendes Byte <> 0 (Z=0): keine Rechtsverschiebung, sondern
,ba5b 4c 83 b9 | jmp b983      Multiplikation
                        byteweise Rechtsverschiebung des RES
-----
,ba5e 4a      |>lsr      b0 in Carry-Flag holen
,ba5f 09 80     | ora #80 %10000000 b7 im Akku setzen
,ba61 a8      |>tay      und Ergebnis im Y-Register merken
,ba62 90 19     | bcc ba7d      b0 ist nach $ba5e gelöscht (C=0): RES um 1 Bit nach rechts verschieben
,ba64 18      | clc      Carry vor Addition bei $ba67 löschen
,ba65 a5 29     | lda 29      Mantisse #4 des RES holen
,ba67 65 6d     | adc 6d      dazu Mantisse #4 des ARG addieren
,ba69 85 29     | sta 29      und Ergebnis in Mantisse #4 des RES schreiben
,ba6b a5 28     | lda 28      Mantisse #3 des RES holen
,ba6d 65 6c     | adc 6c      dazu Mantisse #3 des ARG addieren
,ba6f 85 28     | sta 28      und Ergebnis in Mantisse #3 des RES schreiben

```

,ba71	a5 27	lda	27	Mantisse #2 des RES holen	und Ergebnis
,ba73	65 6b	adc	6b	dazu Mantisse #2 des ARG addieren	jeweils in
,ba75	85 27	sta	27	und Ergebnis in Mantisse #2 des RES schreiben	entsprechende
,ba77	a5 26	lda	26	Mantisse #1 des RES holen	Mantisse des
,ba79	65 6a	adc	6a	dazu Mantisse #1 des ARG addieren	RES
,ba7b	85 26	sta	26	und Ergebnis in Mantisse #1 des RES schreiben	schreiben

; RES um 1 Bit nach rechts verschieben

,ba7d	66 26	ror	26	Mantisse #1 rechtsverschieben	Mantissenbytes
,ba7f	66 27	ror	27	Mantisse #2 rechtsverschieben	und Rundungsbyte
,ba81	66 28	ror	28	Mantisse #3 rechtsverschieben	des RES um 1Bit
,ba83	66 29	ror	29	Mantisse #4 rechtsverschieben	nach rechts verschieben
,ba85	66 70	ror	70	Rundungsbyte des FAC rechtsverschieben	
,ba87	98	tya		bei \$ba61 gemerktes Ergebnis holen	
,ba88	4a	lsr		nach rechts verschieben	
,ba89	d0 d6	bne	ba61	noch nicht fertig mit Verschiebung (Z=0): weiter mit nächstem Bit	
,ba8b	60	rts		Rücksprung von Routine	

; Konstante in ARG kopieren: ARG := Konstante

,ba8c	85 22	sta	22	LB der Adresse der Konstanten in LB von Hilfszeiger \$22/\$23 setzen	\$22/\$23 auf Kon-
,ba8e	84 23	sty	23	HB der Adresse der Konstanten in HB von Hilfszeiger \$22/\$23 setzen	stante richten
,ba90	a0 04	ldy	#04	Offset mit 4 initialisieren (auf Mantisse #4 richten)	
,ba92	b1 22	lda	(22),y	Mantisse #4 der MFLPT-Konstante holen	Bytes
,ba94	85 6d	sta	6d	und in Mantisse #4 des ARG schreiben	der
,ba96	88	dey	"ldy #03"	Offset von 4 auf 3 dekrementieren (auf Mantisse #3 richten)	Mantisse
,ba97	b1 22	lda	(22),y	Mantisse #3 der MFLPT-Konstante holen	der
,ba99	85 6c	sta	6c	und in Mantisse #3 des ARG schreiben	MFLPT-Konstanten
,ba9b	88	dey	"ldy #02"	Offset von 3 auf 2 dekrementieren (auf Mantisse #2 richten)	in
,ba9c	b1 22	lda	(22),y	Mantisse #2 der MFLPT-Konstante holen	Mantisse
,ba9e	85 6b	sta	6b	und in Mantisse #2 des ARG schreiben	des
,baa0	88	dey	"ldy #01"	Offset von 2 auf 1 dekrementieren (auf Mantisse #1 richten)	ARG
,baa1	b1 22	lda	(22),y	Mantisse #1 der MFLPT-Konstante holen	übertragen
,baa3	85 6e	sta	6e	und in Mantisse #1 des ARG schreiben	
,baa5	45 66	eor	66	EOR-Verknüpfung von Mantisse #1 mit Vorzeichenbyte des FAC	
,baa7	85 6f	sta	6f	Ergebnis in Speicher für Vorzeichenvergleich von FAC und ARG schreiben	
,baa9	a5 6e	lda	6e	Mantisse #1 des ARG holen	
,baab	09 80	ora	#80 %10000000	b7 setzen	b7 in ARG-Mantisse #1
,baad	85 6a	sta	6a	und in Mantisse #1 des ARG schreiben	setzen


```
,baaf 88      dey "ldy #00"      Offset von 1 auf 0 dekrementieren (auf Exponentenbyte richten)
,bab0 b1 22    lda (22),y         Exponentenbyte der Konstanten holen
,bab2 85 69    sta 69             und als Exponentenbyte des ARG setzen
,bab4 a5 61    lda 61             Exponentenbyte des FAC holen und CPU-Flags entsprechend setzen
,bab6 60      rts                 Rücksprung von Routine
```

; Hilfsroutine zum Addieren der Exponenten von FAC und ARG

```
,bab7 a5 69    lda 69             Exponentenbyte des ARG auslesen
,bab9 f0 1f    beq bada          Exponent von FAC = 0 (Z=1): an übergeordnete Routine mit FAC = 0 zurückkehren
,babb 18      clc                 Carry vor Addition löschen
,babc 65 61    adc 61             Exponentenbyte des FAC zu ARG-Exponent (s. $bab7) addieren
,babe 90 04    bcc bac4          kein Additionsübertrag (C=0): Sonderbehandlung überspringen
,bac0 30 1d    bmi badf          Additionsübertrag und Ergebnis >= $80 (N=1): OVERFLOW ERROR auslösen
,bac2 18      clc                 Carry vor Addition bei $bac6 löschen
,bac3 2c 10 14 =>"bit" bpl bada   kein Additionsübertrag, aber Ergebnis < $80 (N=1): 0 als Multiplikationsergebnis
                                zurückgeben
,bac6 69 80    adc #80 %10000000 Offset für Exponentenbyte addieren
,bac8 85 61    sta 61             Ergebnis als neuen FAC-Exponenten setzen
,baca d0 03    bne bacf          Exponent <> 0 (Z=0): Ergebnis des Vorzeichenvergleichs von FAC und ARG als
                                FAC-Vorzeichen setzen
,bacc 4c fb b8 jmp b8fb          0 als FAC-Vorzeichen zurückgeben, da Exponent schon auf 0 steht

,bacf a5 6f    lda 6f             Ergebnis des Vorzeichenvergleichs von FAC und ARG holen (b7: 0=gleiches
                                Vorzeichen/1=ungleiches Vorzeichen)
,bad1 85 66    sta 66             und als Vorzeichen des Multiplikationsergebnisses setzen
,bad3 60      rts                 Rücksprung von Routine

,bad4 a5 66    lda 66             Vorzeichenbyte des FAC holen
,bad6 49 ff    eor #ff %11111111 invertieren
,bad8 30 05    bmi badf          Vorzeichen ist jetzt negativ (N=1): OVERFLOW ERROR auslösen
,bada 68      pla                 LB der Rücksprungadresse von übergeordneter Routine vom Stapel entfernen
,badb 68      pla                 HB der Rücksprungadresse von übergeordneter Routine vom Stapel entfernen
,badc 4c f7 b8 jmp b8f7          0 als FAC-Exponent und FAC-Vorzeichen zurückgeben

,badf 4c 7e b9 jmp b97e "overfl" OVERFLOW ERROR auslösen
```

; FACM10-Routine (FAC mit Konstante 10 multiplizieren): $FAC := FAC * 10$

,bae2	20 0c bc	jsr bc0c "movfa"	FAC runden und in ARG übertragen
,bae5	aa	tax	Akku (zwecks Test des Exponentenbyte) in X-Register bringen
,bae6	f0 10	beq baf8	0 steht im FAC (Z=1): Rücksprung über RTS, da $0*10=0$ schon im FAC steht
,bae8	18	clc	Carry vor Addition löschen
,bae9	69 02	adc #02	Exponent um 2 erhöhen (entspricht Multiplikation mit 4!)
,baeb	b0 f2	bcs badf	Additionsübertrag (C=1): OVERFLOW ERROR auslösen
,baed	a2 00	ldx #00	Initialisierungswert für Vorzeichenvergleichsbyte laden
,baef	86 6f	stx 6f	und als Ergebnis des Vorzeichenvergleichs von FAC und ARG setzen
,baf1	20 77 b8	jsr b877	in ADDFAC-Routine einsteigen, damit ursprünglicher FAC-Inhalt, der seit \$bae2 im ARG steht, zum vervierfachen FAC addiert wird; danach ist FAC bereits verfünffacht
,baf4	e6 61	inc 61	Exponent erhöhen (entspricht Multiplikation mit 2); insgesamt wurde FAC jetzt verzehnfacht
,baf6	f0 e7	beq badf	Erhöhungsübertrag (Z=1): OVERFLOW ERROR auslösen
,baf8	60	↳rts	Rücksprung von Routine

; MFLPT-Konstante 10 für FACD10-Routine

:baf9	84 20 00 00 00	MFLPT-Darstellung von 10
-------	----------------	--------------------------

; FACD10-Routine: $FAC := FAC / 10$

,baf6	20 0c bc	jsr bc0c "movfa"	FAC runden und in ARG übertragen
,bb01	a9 f9	lda #f9 <(\$baf9)	LB der Adresse der ROM-Konstanten 10 laden
,bb03	a0 ba	ldy #ba >(\$baf9)	HB der Adresse der ROM-Konstanten 10 laden
,bb05	a2 00	ldx #00	Initialisierungswert für Vorzeichenvergleichsbyte laden
,bb07	86 6f	stx 6f	in Ergebnis für Vergleich der Vorzeichen von FAC und ARG schreiben
,bb09	20 a2 bb	jsr bba2 "movmf"	MFLPT-Konstante (in diesem Fall: 10) in den FAC holen
,bb0c	4c 12 bb	jmp bbl2 "divaf"	$FAC := ARG / FAC$; nach Vorbereitungen: $FAC := \text{vorheriger FAC} / 10$

; DIVMF-Routine: $FAC := \text{Konstante} / FAC$

,bb0f	20 8c ba	jsr ba8c "movma"	Konstante in ARG übertragen
-------	----------	------------------	-----------------------------

; DIVAF-Routine: $FAC := ARG / FAC$

,bbl2	f0 76	↳beq bb8a	FAC enthält 0 (Z=1): DIVISION BY ZERO hervorrufen
,bbl4	20 1b bc	jsr bclb "round"	FAC runden (Rundungsbyte berücksichtigen)

,bb17	a9 00	lda #00	Wert 0 laden, von dem bei \$bb1a das Exponentenbyte abgezogen wird
,bb19	38	sec	Carry vor Subtraktion setzen
,bb1a	e5 61	sbc 61	Exponentenbyte des FAC von 0 abziehen; dient Invertierung des Exponenten-Vorzeichens
,bb1c	85 61	sta 61	und Ergebnis als neues Exponentenbyte des FAC setzen
,bb1e	20 b7	ba jsr bab7	Exponenten von FAC und ARG addieren (wie bei Multiplikation)
,bb21	e6 61	inc 61	Exponent des FAC erhöhen (entspricht Multiplikation mit 2)
,bb23	f0 ba	beq badf	Erhöhungsübertrag (Z=1): OVERFLOW ERROR auslösen
,bb25	a2 fc	ldx #fc \$ff-4	komplementierten Offset für Mantisse #4 (4 Byte) laden
,bb27	a9 01	lda #01	Bytezähler laden (Zähler für 8 Bit = 1 Byte)
,bb29	a4-6a	→ldy 6a	Mantisse #1 des ARG holen
,bb2b	c4 62	cpy 62	mit Mantisse #1 des FAC vergleichen
,bb2d	d0 10	bne bb3f	keine Übereinstimmung (Z=0): Vergleich abbrechen
,bb2f	a4 6b	ldy 6b	Mantisse #2 des ARG holen
,bb31	c4 63	cpy 63	mit Mantisse #2 des FAC vergleichen
,bb33	d0 0a	bne bb3f	keine Übereinstimmung (Z=0): Vergleich abbrechen
,bb35	a4 6c	ldy 6c	Mantisse #3 des ARG holen
,bb37	c4 64	cpy 64	mit Mantisse #3 des FAC vergleichen
,bb39	d0 04	bne bb3f	keine Übereinstimmung (Z=0): Vergleich abbrechen
,bb3b	a4 6d	ldy 6d	Mantisse #4 des ARG holen
,bb3d	c4 65	cpy 65	mit Mantisse #4 des FAC vergleichen
,bb3f	08	→php	Vergleichsergebnis (Zero-Flag: 1=Übereinstimmung/0=keine Übereinstimmung) merken
,bb40	2a	rol	Bytezähler (s. \$bb27) verdoppeln, b7 ins Carry holen
,bb41	90 09	bcc bb4c	b7 war gelöscht (C=0): Sonderbehandlung abbrechen, Vergleichsergebnis auswerten
,bb43	e8	inx	Offset erhöhen (auf nächstes Mantissenbyte stellen)
,bb44	95 29	sta 29,x	und Bytezähler in RES-Mantissenbyte schreiben
,bb46	f0 32	↓beq bb7a	Offset wurde bei \$bb43 auf 0 erhöht (Z=1): Sonderbehandlung: mit \$40 im Akku bei \$bb4c weitermachen
,bb48	10 34	↓bpl bb7e	Offset ist nicht mehr negativ, da er vorher schon 0 war (N=0): Ende der Vergleichsschleife, weiter bei \$bb7e
,bb4a	a9 01	lda #01	Initialisierungswert für Bytezähler laden
,bb4c	28	→plp	bei \$bb3f gemerktes Vergleichsergebnis wieder vom Stapel holen
,bb4d	b0 0e	bcs bb5d	Mantisse des ARG >= Mantisse des FAC (C=1): FAC-Mantisse von ARG-Mantisse abziehen
,bb4f	06 6d	asl 6d	Mantisse #4 des ARG verdoppeln
,bb51	26 6c	rol 6c	Mantisse #3 des ARG verdoppeln
,bb53	26 6b	rol 6b	Mantisse #2 des ARG verdoppeln
,bb55	26 6a	rol 6a	Mantisse #1 des ARG verdoppeln
,bb57	b0 e6	bcs bb3f	Verdopplungsübertrag (C=1): weiter mit neu initialisiertem Bytezähler (s. \$bb4a)
,bb59	30 ce	bmi bb29	b7 im Ergebnis gesetzt (N=1): weiter mit neuem Vergleich von FAC und ARG
,bb5b	10 e2	bpl bb3f "jmp"	b7 gelöscht (N=0, wg. \$bb59: immer erfüllt): weiter mit neu initialisiertem Bytezähler (s. \$bb4a)

; Subtraktion der FAC-Mantisse von der ARG-Mantisse

,bb5d	a8	→tay	Inhalt des Akkumulators (Bytezähler) in Y-Register bis \$bb76 retten	
,bb5e	a5 6d	lda 6d	Mantisse #4 des ARG laden	} gleichwertige Mantissenbytes von ARG und FAC subtrahieren und Ergebnis in ARG schreiben
,bb60	e5 65	sbc 65	davon Mantisse #4 des FAC subtrahieren	
,bb62	85 6d	sta 6d	und Ergebnis in Mantisse #4 des ARG	
,bb64	a5 6c	lda 6c	Mantisse #3 des ARG laden	
,bb66	e5 64	sbc 64	davon Mantisse #3 des FAC subtrahieren	
,bb68	85 6c	sta 6c	und Ergebnis in Mantisse #3 des ARG	
,bb6a	a5 6b	lda 6b	Mantisse #2 des ARG laden	
,bb6c	e5 63	sbc 63	davon Mantisse #2 des FAC subtrahieren	
,bb6e	85 6b	sta 6b	und Ergebnis in Mantisse #2 des ARG	
,bb70	a5 6a	lda 6a	Mantisse #1 des ARG laden	
,bb72	e5 62	sbc 62	davon Mantisse #1 des FAC subtrahieren	
,bb74	85 6a	sta 6a	und Ergebnis in Mantisse #1 des ARG	
,bb76	98	tya	bei \$bb5d geretteten Akkumulator (Bytezähler) wiederherstellen	
,bb77	4c 4f bb	jmp bb4f	weiter mit Verdoppelung der ARG-Mantisse	

; Sonderbehandlung: \$40 in Akku (Bytezähler) laden

,bb7a	a9 40	lda #40 %01000000	neuen Bytezähler-Inhalt laden
,bb7c	d0 ce	bne bb4c "jmp"	weiter bei \$bb4c mit neuem Bytezähler

; Endbehandlung nach Vergleichsschleife:

Akku mit 64 multiplizieren, Ergebnis in Rundungsbyte und dann den RES in den FAC übertragen

,bb7e	0a	asl	Akku:=Akku*2	} Akku := Akku * 2 [↑] 6, also Akku := Akku * 64. Das Ergebnis ist die niederwertigste Stelle (Rundungsbyte).
,bb7f	0a	asl	Akku:=Akku*2	
,bb80	0a	asl	Akku:=Akku*2	
,bb81	0a	asl	Akku:=Akku*2	
,bb82	0a	asl	Akku:=Akku*2	
,bb83	0a	asl	Akku:=Akku*2	
,bb84	85 70	sta 70	Ergebnis als Rundungsbyte des FAC setzen	
,bb86	28	plp	bei \$bb3f gemerktes Vergleichsergebnis wieder vom Stapel holen (wird nicht mehr benötigt)	
,bb87	4c 8f bb	jmp bb8f "movrf"	RES in FAC übertragen und Rücksprung von Routine	

; DIVISION BY ZERO ERROR erzeugen

```
,bb8a a2 14    ldx #14      Fehlernummer für DIVISION BY ZERO laden
,bb8c 4c 37 a4  jmp a437 "error" Fehlermeldung erzeugen
```

; MOVRF-Routine: Übertragung des RES (Ergebnis-Fließkomma-Akkumulator) in den FAC

,bb8f	a5 26	lda 26	Mantisse #1 des RES auslesen	} Übertragung der 4 Mantissenbytes des RES (Resultats- Fließkomma- Akkumulator) in die FAC-Mantisse, dann FAC normalisieren
,bb91	85 62	sta 62	und in Mantisse #1 des FAC schreiben	
,bb93	a5 27	lda 27	Mantisse #2 des RES auslesen	
,bb95	85 63	sta 63	und in Mantisse #2 des FAC schreiben	
,bb97	a5 28	lda 28	Mantisse #3 des RES auslesen	
,bb99	85 64	sta 64	und in Mantisse #3 des FAC schreiben	
,bb9b	a5 29	lda 29	Mantisse #4 des RES auslesen	
,bb9d	85 65	sta 65	und in Mantisse #4 des FAC schreiben	
,bb9f	4c d7 b8	jmp b8d7 "normal"	Normalisierung des FAC (ggf. Linksverschiebung)	

; MOVMF-Routine: Übertragung einer MFLPT-Konstanten in den FAC

(Adresse der Konstanten wird in A/Y erwartet)

,bba2	85 22	sta 22	LB der Adresse in LB des Hilfsspeichers \$22/\$23	} Adresse der MFLPT-Konstanten in Hilfszeiger \$22/\$23 schreiben
,bba4	84 23	sty 23	HB der Adresse in HB des Hilfsspeichers \$22/\$23	
,bba6	a0 04	ldy #04	Offset mit 4 initialisieren (auf Mantisse #4 richten)	
,bba8	b1 22	lda (22),y	Mantisse #4 der MFLPT-Konstanten holen	
,bbaa	85 65	sta 65	und in Mantisse #4 des FAC übernehmen	
,bbac	88	dey "ldy #03"	Offset von 4 auf 3 dekrementieren (auf Mantisse #3 richten)	
,bbad	b1 22	lda (22),y	Mantisse #3 der MFLPT-Konstanten holen	
,bbaf	85 64	sta 64	und in Mantisse #3 des FAC übernehmen	
,bbb1	88	dey "ldy #02"	Offset von 3 auf 2 dekrementieren (auf Mantisse #2 richten)	
,bbb2	b1 22	lda (22),y	Mantisse #2 der MFLPT-Konstanten holen	
,bbb4	85 63	sta 63	und in Mantisse #2 des FAC übernehmen	
,bbb6	88	dey "ldy #01"	Offset von 2 auf 1 dekrementieren (auf Mantisse #1 richten)	
,bbb7	b1 22	lda (22),y	Mantisse #1 der MFLPT-Konstanten holen	
,bbb9	85 66	sta 66	und in Vorzeichenbyte des FAC schreiben, da b7 der MFLPT-Mantisse #1 gleichzeitig das Vorzeichen beinhaltet, um Speicherplatz zu sparen	
,bbbb	09 80	ora #80 %10000000	b7 setzen (bei Mantisse #1 im FLPT-Format des FAC ist dies Voraussetzung)	
,bbbd	85 62	sta 62	und korrigierte Mantisse #1 in FAC übernehmen	
,bbb f	88	dey "ldy #00"	Offset von 1 auf 0 dekrementieren (auf Exponent stellen)	
,bbc0	b1 22	lda (22),y	Exponentenbyte der MFLPT-Konstanten holen	

```
,bbc2 85 61    sta    61      und als Exponentenbyte des FAC setzen
,bbc4 84 70    sty    70      gleichzeitig Rundungsbyte des FAC löschen (seit $bbbf ist Y mit 0 belegt)
,bbc6 60      rts              Rücksprung von Routine; CPU-Flags sind entsprechend dem FAC-Exponentenbyte gesetzt
```

; MOVT4-Routine: FAC #1 in FAC #4 (FAC-Zwischenspeicher ab \$5c) übertragen

```
,bbc7 a2 5c    ldx    #5c      Zeropage-Adresse des FAC #4 laden (als Zieladresse der Übertragung)
```

; MOVT3-Routine: FAC #1 in FAC #3 (FAC-Zwischenspeicher ab \$57) übertragen

```
,bbc9 2c a2 57 "bit" ldx    #57 *$57 Zeropage-Adresse des FAC #3 laden (als Zieladresse der Übertragung)
```

; MOV TZ-Routine: FAC in Zeropage-Hilfsspeicher übertragen, dessen Zeropage-Adresse im X-Register steht

```
,bbcc a0 00    ldy    #00      HB der Adresse auf 0 setzen, da MOV TZ nur in Zeropage kopiert
,bbce f0 04    beq    bbd4 "jmp movfm" FAC an die in X/Y enthaltene Adresse ins MFLPT-Format übertragen
```

; FACVAR-Routine: FAC in aktuelle Variable (Adresse in \$49/\$4a enthalten) übertragen

```
,bbd0 a6 49    ldx    49      LB der Variablenadresse holen      } Variablenadresse
,bbd2 a4 4a    ldy    4a      HB der Variablenadresse holen      } nach X/Y holen
```

; MOV FM-Routine: FAC an beliebige Adresse (in X/Y enthalten) ins MFLPT-Format übertragen

```
,bbd4 20 1b bc->jsr    bclb "round" FAC runden (Rundungsbyte berücksichtigen)
,bbd7 86 22    stx    22      LB der Zieladresse in LB des Hilfszeigers $22/$23 schreiben } Zieladresse nach
,bbd9 84 23    sty    23      HB der Zieladresse in HB des Hilfszeigers $22/$23 schreiben } $22/$23 schreiben
,bbdb a0 04    ldy    #04      Offset mit 4 initialisieren (auf Mantisse #4 stellen)
,bbdd a5 65    lda    65      Mantisse #4 des FAC holen
,bbdf 91 22    sta    (22),y   und in Mantisse #4 der MFLPT-Zieladresse schreiben
,bbel 88      dey    "ldy #03" Offset von 4 auf 3 dekrementieren (auf Mantisse #3 stellen)
,bbe2 a5 64    lda    64      Mantisse #3 des FAC holen
,bbe4 91 22    sta    (22),y   und in Mantisse #3 der MFLPT-Zieladresse schreiben
,bbe6 88      dey    "ldy #02" Offset von 3 auf 2 dekrementieren (auf Mantisse #2 stellen)
,bbe7 a5 63    lda    63      Mantisse #2 des FAC holen
,bbe9 91 22    sta    (22),y   und in Mantisse #4 der MFLPT-Zieladresse schreiben
,bbec 88      dey    "ldy #01" Offset von 2 auf 1 dekrementieren (auf Mantisse #1 stellen)
,bbec a5 66    lda    66      Vorzeichenbyte des FAC holen      } b7 des Vorzeichenbyte
,bbef 09 7f    ora    #7f %01111111 alle Bits außer b7 setzen      } als b7 in FAC-Mantisse #1
,bbf0 25 62    and    62      und mit FAC-Mantisse #1 ANDen      } einblenden
```

```
,bbf2 91 22    sta (22),y    und in Mantisse #4 der MFLPT-Zieladresse schreiben
,bbf4 88       dey "ldy #00" Offset von 1 auf 0 dekrementieren (auf Exponentenbyte stellen)
,bbf5 a5 61    lda 61       Exponentenbyte des FAC holen
,bbf7 91 22    sta (22),y    und in Exponentenbyte der MLFPT-Zieladresse schreiben
,bbf9 84 70    sty 70       Rundungsbyte des FAC löschen (Y ist seit $bbf4 mit 0 belegt)
,bbfb 60       rts         Rücksprung von Routine; CPU-Flags sind entsprechend dem FAC-Exponentenbyte gesetzt
```

; MOVAF-Routine: ARG in FAC übertragen (FAC := ARG)

```
,bbfc a5 6e    lda 6e       Vorzeichenbyte des ARG holen          } Vorzeichen des ARG als
,bbfe 85 66    sta 66       und in Vorzeichenbyte des FAC schreiben } Vorzeichen des FAC setzen
,bc00 a2 05    ldx #05      Dekrementierzähler für Kopierschleife mit 5 initialisieren
,bc02 b5 68    →lda 68,x    Byte (Mantisse oder Exponent, je nach X) aus ARG lesen
,bc04 95 60    sta 60,x     und in entsprechendes Byte des FAC schreiben
,bc06 ca       dex         Dekrementierzähler verringern (auf nächstes Byte richten)
,bc07 d0 f9    bne bc02     noch nicht auf 0 heruntergezählt (Z=0): weiter in Kopierschleife
,bc09 86 70    stx 70       Rundungsbyte des FAC löschen (X ist wegen $bc06/$bc07 hier immer mit 0 belegt)
,bc0b 60       rts         Rücksprung von Routine
```

; MOVFA-Routine: FAC in ARG übertragen (ARG := FAC)

```
,bc0c 20 1b bc jsr bclb "round" FAC runden (Rundungsbyte berücksichtigen)
,bc0f a2 06    ldx #06      Dekrementierzähler für Kopierschleife mit 6 initialisieren
,bc11 b5 60    →lda 60,x    Byte aus FAC (Mantisse oder Exponent, je nach X) aus FAC holen
,bc13 95 68    sta 68,x     und in entsprechendes Byte des FAC schreiben
,bc15 ca       dex         Dekrementierzähler verringern (auf nächstes Byte richten)
,bc16 d0 f9    bne bc11     noch nicht auf 0 heruntergezählt (Z=0): weiter in Kopierschleife
,bc18 86 70    stx 70       Rundungsbyte des FAC löschen (X ist wegen $bcl5/$bcl6 hier immer mit 0 belegt)
,bc1a 60       →rts         Rücksprung von Routine
```

; ROUND-Routine: FAC runden (Rundungsbyte berücksichtigen)

```
,bclb a5 61    lda 61       FAC-Exponent zwecks Test auslesen
,bc1d f0 fb    beq bcla     FAC = 0 (Z=1): Rücksprung über RTS, da Rundung überflüssig
,bclf 06 70    asl 70       Rundungsbyte des FAC linksverschieben, damit b7 ins Carry kommt
,bc21 90 f7    bcc bcla     b7 des Rundungsbyte war gelöscht (C=0): Rücksprung über RTS, da Rundung überflüssig
,bc23 20 6f b9 jsr b96f     in andere Hilfsroutine an einer Stelle einsteigen, ab der die Mantisse des FAC um 1 erhöht wird
```

```

, bc26 d0 f2    l bne bc1a    kein Erhöhungsübertrag nach $bc23 (Z=0): Rücksprung über RTS, da Rundung bereits
                                korrekt ausgeführt ist
, bc28 4c 38 b9    jmp b938    in andere Hilfsroutine an einer Stelle einsteigen, ab der der Exponent um 1 erhöht
                                und die gesamte Mantisse um 1 Bit nach rechts verschoben wird
-----

; SIGN-Routine: Vorzeichen des FAC in Akku holen und testen (danach ist Z=1 und Akku=0, wenn FAC = 0; Z=0,
wenn FAC <> 0; N=1 und Akku=$ff, wenn FAC < 0; N=0 und Akku = $01, wenn FAC > 0)

, bc2b a5 61      lda 61      Exponent des FAC auslesen
, bc2d f0 09      l beq bc38    Exponent = 0, also FAC = 0 (Z=1): RTS bei gesetztem Z-Flag als Zeichen für "FAC = 0"
, bc2f a5 66      lda 66      Vorzeichenbyte des FAC holen
, bc31 2a         rol         und Linksverschiebung, damit b7 (Vorzeichenbit) ins Carry kommt
, bc32 a9 ff      lda #ff "sen" Ergebnis für "FAC < 0" vorbereiten, N-Flag wird hier gesetzt
, bc34 b0 02      l bcs bc38    b7 (Vorzeichenbit) war gesetzt (C=1): mit N=1 und Akku=$ff über RTS zurückspringen
, bc36 a9 01      lda #01 "cIn" Ergebnis für "FAC > 0" laden, N-Flag wird hier wieder gelöscht
, bc38 60         l rts        Rücksprung von Routine; CPU-Flags und Akku geben Auskunft über Ergebnis
-----

; Routine zur Basic-Funktion SGN (Token: $b4)

, bc39 20 2b bc    jsr bc2b "sign" Vorzeichen des FAC in Akku holen und testen
                                (danach ist Z=1 und Akku=0, wenn FAC = 0; Z=0, wenn FAC <> 0;
                                N=1 und Akku=$ff, wenn FAC < 0; N=0 und Akku = $01, wenn FAC > 0)
, bc3c 85 62      sta 62      Ergebnis in Speicher für Mantisse #1 schreiben
, bc3e a9 00      lda #00      Initialisierungswert für Mantisse #2 laden
, bc40 85 63      sta 63      und Mantisse #2 löschen
, bc42 a2 88      ldx #88      gewünschten Exponent für Fließkomma-Ergebnis laden (Ergebnis im Byte-Bereich [0;255])

; WRDFAC-Einsprung (von $b39b genutzt)

, bc44 a5 62      lda 62      bei $bc3c gemerktes Vergleichsergebnis holen
, bc46 49 ff      eor #ff %11111111 komplementieren (aus $ff wird $00, aus $00 wird $ff, aus $01 wird $fe)
, bc48 2a         rol         b7 durch Linksverschiebung ins Carry-Flag holen

; BINFAC-Einsprung: Umwandlung einer Integerzahl in $63/$62 in Fließkomma-Zahl, die in FAC geschrieben wird
(Nutzung von $bdd4)

, bc49 a9 00      lda #00      Initialisierungswert für weitere FAC-Speicherzellen laden
, bc4b 85 65      sta 65      Mantisse #4 löschen
, bc4d 85 64      sta 64      Mantisse #3 löschen

```


; SETFAC-Einsprung (von \$afbl genutzt)

,bc4f	86 61	stx 61	Exponentenbyte löschen
,bc51	85 70	sta 70	Rundungsbyte löschen
,bc53	85 66	sta 66	Vorzeichenbyte löschen
,bc55	4c d2 b8	jmp b8d2	in ADDFAC-Routine einsteigen, wo das Ergebnis in eine Fließkomma-Zahl umgewandelt wird

; Routine zur Basic-Funktion ABS (Token: \$b6)

,bc58	46 66	lsr 66	Rechtsverschiebung des Vorzeichenbyte, da in diesem nur b7 Bedeutung hat und somit durch Transport in b6 entfernt wird; Mantisse und Exponent bleiben unverändert
,bc5a	60	rts	Rücksprung von Routine

; CMPFAC-Routine: FAC mit MFLPT-Konstante vergleichen

,bc5b	85 24	sta 24	LB der Adresse der MFLPT-Konstanten in LB von Hilfszeiger \$24/\$25 schreiben
,bc5d	84 25	sty 25	HB der Adresse der MFLPT-Konstanten in HB von Hilfszeiger \$24/\$25 schreiben
,bc5f	a0 00	ldy #00	Offset mit 0 initialisieren (auf Exponentenbyte stellen)
,bc61	b1 24	lda (24),y	Exponentenbyte der MFLPT-Konstanten holen
,bc63	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf Mantisse #1 stellen)
,bc64	aa	tax	Exponent (s. \$bc61) in X-Register bringen; dabei wird Z-Flag entsprechend gesetzt
,bc65	f0 c4	beq bc2b "sign"	Exponent = 0, also FAC = 0 (Z=1): zur SIGN-Routine springen (ein Test des Vorzeichens kommt mathematisch gesehen einem Vergleich der Zahl mit 0 gleich!)
,bc67	b1 24	lda (24),y	Mantisse #1 der MFLPT-Konstanten holen
,bc69	45 66	eor 66	EOR-Verknüpfung mit Vorzeichenbyte des FAC zwecks Vergleich
,bc6b	30 c2	bmi bc2f	unterschiedliche Vorzeichen der beiden Werte (N=1): in SIGN-Routine einsteigen (s. Fließtext!)
,bc6d	e4 61	cpx 61	Vergleich des MFLPT-Exponenten (s. \$bc64) mit dem FAC-Exponentenbyte
,bc6f	d0 21	bne bc92	keine Übereinstimmung (Z=0): Vergleich abbrechen, da keine Übereinstimmung
,bc71	b1 24	lda (24),y	Mantisse #1 der MFLPT-Konstanten holen
,bc73	09 80	ora #80 %10000000	b7 setzen (Umwandlung in FLPT-Format)
,bc75	c5 62	cmp 62	Vergleich der konvertierten Mantisse #1 der MFLPT-Zahl mit Mantisse #1 des FAC
,bc77	d0 19	bne bc92	keine Übereinstimmung (Z=0): Vergleich abbrechen, da keine Übereinstimmung
,bc79	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf Mantisse #2 stellen)
,bc7a	b1 24	lda (24),y	Mantisse #2 der MFLPT-Konstanten holen
,bc7c	c5 63	cmp 63	Vergleich mit Mantisse #2 des FAC
,bc7e	d0 12	bne bc92	keine Übereinstimmung (Z=0): Vergleich abbrechen, da keine Übereinstimmung
,bc80	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf Mantisse #3 stellen)
,bc81	b1 24	lda (24),y	Mantisse #3 der MFLPT-Konstanten holen

,bc83	c5 64	cmp 64	Vergleich mit Mantisse #3 des FAC
,bc85	d0 0b	bne bc92	keine Übereinstimmung (Z=0): Vergleich abbrechen, da keine Übereinstimmung
,bc87	c8	iny "ldy #04"	Offset von 3 auf 4 erhöhen (auf Mantisse #4 stellen)
,bc88	a9 7f	lda #7f %01111111	Vergleichswert für Rundungsbyte laden (\$7f = höchster Wert, der keine Rundung erforderlich macht)
,bc8a	c5 70	cmp 70	Vergleich mit Rundungsbyte des FAC; wichtig ist dabei das Setzen/Löschen des C-Flags
,bc8c	b1 24	lda (24),y	Mantisse #4 der MFLPT-Konstanten holen
,bc8e	e5 65	sbc 65	Mantisse #4 des FAC abziehen; bei Rundung (s. \$bc88/\$bc8a) wird zusätzlich 1 subtrahiert, da das C-Flag von \$bc8a bei Rundungsbytes über \$7f gelöscht wird
,bc90	f0 28	beq bcba	Übereinstimmung (Z=1): mit Akku=\$00 und Z=1 erfolgt Rücksprung über RTS

; wenn FAC und MFLPT-Konstante ungleich: Ermittlung, welcher Wert größer ist

,bc92	a5 66	→lda 66	Vorzeichenbyte des FAC holen
,bc94	90 02	bcc bc98	MFLPT-Wert < FAC-Inhalt (C=0): in SIGN-Routine einsteigen, wo Vorzeichen des FAC als Vergleichsergebnis zurückgegeben wird
,bc96	49 ff	eor #ff %11111111	Vorzeichenbyte des FAC komplementieren, damit das umgekehrte Vorzeichen des FAC als Vergleichsergebnis verwendet wird
,bc98	4c 31 bc	→jmp bc31	in SIGN-Routine einsteigen; im Akku enthaltenes Vorzeichen wird als Vergleichsergebnis zurückgegeben

; FACINT-Routine: FAC in Integerzahl umwandeln

Die Integerzahl wird in \$62-\$65 (Mantissenbytes) zurückgegeben, wobei allerdings nicht das Low-High-Format, sondern das High-Low-Format gilt; \$62 ist also das MSB

,bc9b	a5 61	lda 61	Exponent des FAC holen
,bc9d	f0 4a	beq bce9	Exponentenbyte = 0, also FAC = 0 (Z=1): 0 als Ergebnis zurückgeben
,bc9f	38	sec	Carry vor Subtraktion setzen
,bca0	e9 a0	sbc #a0	Exponentenbyte des für Umwandlung in Frage kommenden Bereichs subtrahieren
,bca2	24 66	bit 66	Vorzeichenbyte des FAC testen
,bca4	10 09	bpl bcaf	positives Vorzeichen (N=0): Sonderbehandlung für negative Zahlen überspringen
,bca6	aa	tax	Subtraktionsergebnis in X-Register merken
,bca7	a9 ff	lda #ff %11111111	alle Bits gesetzt (= Überlauf)
,bca9	85 68	sta 68	Überlaufbyte des FAC setzen
,bcab	20 4d b9	jsr b94d	FAC invertieren (Vorzeichen umdrehen und Mantissenbytes komplementieren)
,bcae	8a	txa	bei \$bca6 gemerktes Subtraktionsergebnis wieder in Akku holen
,bcaf	a2 61	→ldx #61 *\$61	Zeropage-Adresse des FAC laden
,bcbl	c9 f9	cmp #f9	Vergleich des Subtraktionsergebnisses mit Ergebnis für FAC-Exponent \$99
,bcb3	10 06	bpl bcbb	Exponent-160 > -8 (N=0): Sonderbehandlung ab \$bcbb
,bcb5	20 99 b9	jsr b999 "shiftr"	FAC so lange rechtsverschieben, bis der Exponent auf 0 steht; dann steht das Ergebnis

					als Integerzahl in den FAC-Mantissenbytes \$62-\$65; wichtige Vorbereitung: FAC-Adresse seit \$bcaf im X-Register
,bcb8	84 68	sty 68			Überlaufbyte des FAC löschen, da Y nach "jsr \$b999" immer auf 0 steht
,bcba	60	→rts			Rücksprung von Routine

,bcbb	a8	→tay			Subtraktionsergebnis (Exponent-160) in Y-Register merken
,bcbc	a5 66	lda 66			Vorzeichenbyte des FAC holen
,bcbe	29 80	and #80 %10000000			alle Bits bis auf b7 löschen, also nur b7 weiterverwenden
,bcc0	46 62	lsr 62			Mantisse #1 des FAC rechtsverschieben (durch 2 dividieren)
,bcc2	05 62	ora 62			und b7 des Vorzeichenbyte in Mantisse #1 einblenden
,bcc4	85 62	sta 62			Ergebnis in Speicher für Mantisse #1 schreiben
,bcc6	20 b0 b9	jsr b9b0			in SHIFTR-Routine so einsteigen, daß andere Mantissenbytes rechtsverschoben werden
					wichtige Vorbereitung: FAC-Adresse seit \$bcaf im X-Register enthalten
,bcc9	84 68	sty 68			Überlaufbyte des FAC löschen, da Y nach "jsr \$b9b0" immer auf 0 steht
,bccb	60	→rts			Rücksprung von Routine

; Routine zur Basic-Funktion INT (Token: \$b5)

,bccc	a5 61	lda 61			Exponentenbyte des FAC holen
,bcce	c9 a0	cmp #a0			Vergleich mit maximal zulässigem Exponenten für Integerbereich
,bcd0	b0 20	bcs bcf2			Exponent des FAC > maximal zulässiger Exponent (C=1): Rücksprung über RTS
,bcd2	20 9b bc	jsr bc9b "facint"			FAC in Integerzahl umwandeln
,bcd5	84 70	sty 70			Rundungsbyte des FAC löschen, da Y nach FACINT-Routine immer gelöscht ist
,bcd7	a5 66	lda 66			Vorzeichenbyte des FAC in Akku holen,
,bcd9	84 66	sty 66			aber im Speicher löschen
,bcdb	49 80	eor #80 %10000000			b7 (Vorzeichenbit) umdrehen, andere Bits bleiben unverändert
,bcd d	2a	rol			b7 durch Linksverschiebung in Carry-Flag holen
,bcde	a9 a0	lda #a0			Exponent des Ergebnisses holen
,bce0	85 61	sta 61			und in Exponentenbyte des FAC schreiben
,bce2	a5 65	lda 65			Mantisse #4 des FAC auslesen
,bce4	85 07	sta 07			und in Hilfsspeicher \$07 ablegen
,bce6	4c d2 b8	jmp b8d2			in ADDFAC-Routine zwecks Komplementierung und Normalisierung des FAC

; Sonderfall der FACINT-Routine: Mantissenbytes mit Akku (immer \$00) belegen, Y-Register löschen, Rücksprung

,bce9	85 62	→sta 62	Mantisse #1 belegen	} Belegung aller 4 Mantissenbytes mit Akku-Inhalt
,bceb	85 63	sta 63	Mantisse #2 belegen	
,bcd e	85 64	sta 64	Mantisse #3 belegen	
,bcef	85 65	sta 65	Mantisse #4 belegen	

```
,bcf1 a8      |  tay "ldy #00"    Y-Register mit 0 laden (wird von aufrufenden Routinen erwartet)
,bcf2 60      |  >rts                    Rücksprung von Routine
```

; STRFLP-Routine: ASCII-String in Fließkomma-Format umwandeln

CHRGET-Zeiger muß auf String weisen, erstes Zeichen sollte vorher über CHRGET geholt werden

```
,bcf3 a0 00      ldy #00          Initialisierungswert für Bereich $5d-$66 laden
,bcf5 a2 0a      ldx #0a          Dekrementierzähler für Kopierschleife initialisieren
,bcf7 94 5d      >sty 5d,x        Speicherzelle löschen
,bcf9 ca         dex              Dekrementierzähler verringern
,bcfa 10 fb      |  bpl bcf7          noch nicht auf negativen Wert $ff heruntergezählt (N=0): weiter in Schleife (löschen)
,bcfc 90 0f      |  bcc bd0d         Ziffer als erstes Zeichen im String (C=0): Prüfung auf Vorzeichen überspringen
,bcfe c9 2d      |  cmp #2d          Vergleich des ersten Zeichens im String mit dem ASCII-Code von "-"
,bd00 d0 04      |  bne bd06         keine Übereinstimmung (Z=0): Sonderbehandlung für negatives Vorzeichen überspringen
,bd02 86 67      |  stx 67           Vorzeichenflag mit $ff (s. $bcf9/$bcfa) belegen, also auf "negativ" stellen
,bd04 f0 04      |  beq bd0a "jmp"   weiter mit nächstem Zeichen hinter dem Vorzeichen
```

```
,bd06 c9 2b      >cmp #2b          Vergleich des ersten Zeichens im String mit dem ASCII-Code von "+"
,bd08 d0 05      |  bne bd0f         keine Übereinstimmung (Z=0): Zeichen normal weiterverarbeiten; ansonsten folgt
                                   "jsr chrget", wodurch das "+" ignoriert wird
,bd0a 20 73 00   |  >jsr 0073 "chrget" nächstes Zeichen aus String holen
,bd0d 90 5b      |  >bcc bd6a         Ziffer (C=0): Behandlung ab $bd6a anspringen
,bd0f c9 2e      |  >cmp #2e          Vergleich mit ASCII-Code für Dezimalpunkt
,bd11 f0 2e      |  beq bd41         Übereinstimmung (Z=1): Sonderbehandlung für Dezimalpunkt anspringen
,bd13 c9 45      |  cmp #45          Vergleich mit ASCII-Code von "e"
,bd15 d0 30      |  >bne bd47         keine Übereinstimmung (Z=0): Sonderbehandlung für Exponentialzeichen überspringen
```

; Sonderbehandlung für Exponentialzeichen

```
,bd17 20 73 00   |  jsr 0073 "chrget" nächstes Zeichen aus ASCII-String holen
,bd1a 90 17      |  bcc bd33         Ziffer (C=0): Exponent hinter "E" auswerten
,bd1c c9 ab      |  cmp #ab          Vergleich des Zeichens mit Token von "-"
,bd1e f0 0e      |  beq bd2e         Übereinstimmung (Z=1): Sonderbehandlung für "Exponent negativ" anspringen
,bd20 c9 2d      |  cmp #2d          Vergleich des Zeichens mit ASCII-Code von "-"
,bd22 f0 0a      |  beq bd2e         Übereinstimmung (Z=1): Sonderbehandlung für "Exponent negativ" anspringen
,bd24 c9 aa      |  cmp #aa          Vergleich des Zeichens mit Token von "+"
,bd26 f0 08      |  beq bd30         Übereinstimmung (Z=1): weiter mit nächstem Zeichen aus ASCII-String, "+" ignorieren
,bd28 c9 2b      |  cmp #2b          Vergleich des Zeichens mit ASCII-Code von "+"
,bd2a f0 04      |  beq bd30         Übereinstimmung (Z=1): weiter mit nächstem Zeichen aus ASCII-String, "+" ignorieren
,bd2c d0 07      |  >bne bd35 "jmp"   ansonsten (Z=0, wg. $bd2a immer erfüllt): Exponentenauswertung fortsetzen
```


; Sonderbehandlung für "Exponent hinter E als negativ gekennzeichnet"

(Carry-Flag muß hier gesetzt sein, s. \$bdlc/\$bdle bzw. \$bd20/bd22)

```
,bd2e 66 60 |>ror 60      Carry-Flag in b7 des Hilfsspeichers $60 (Vorzeichen des Exponenten) einbinden
,bd30 20 73 00->jsr 0073 "chrget" nächstes Zeichen aus ASCII-String holen, dann weiter mit Exponentenauswertung
```

; Ziffer im Exponent hinter "E" auswerten

```
,bd33 90 5c |>bcc bd91      Zeichen ist Ziffer (C=0): Ziffer aus Exponent auswerten
,bd35 24 60 |>bit 60      Test des Vorzeichenbit für den Exponenten
,bd37 10 0e |bpl bd47      positiv (N=0): Behandlung für "weder Sonderzeichen noch Ziffer" anspringen
,bd39 a9 00 |lda #00      0 als Wert laden, von dem bei $bd3c die Exponenten-Position subtrahiert wird
,bd3b 38 |sec          Carry vor Subtraktion setzen
,bd3c e5 5e |sbc 5e      Position des Exponenten abziehen
,bd3e 4c 49 bd |jmp bd49     in Behandlung für "weder Sonderzeichen noch Ziffer" einsteigen
```

; Sonderbehandlung für Dezimalpunkt im ASCII-String

```
,bd41 66 5f |>ror 5f      Dezimalpunkt-Flag (b7) durch Rechtsverschiebung in b6 bringen
,bd43 24 5f |bit 5f      Dezimalpunkt-Flag testen
,bd45 50 c3 |bvc bd0a     noch kein Dezimalpunkt vorhanden (V=0): normal weiterarbeiten (Dezimalstellen auswerten)
```

; Behandlung für "weder Sonderzeichen noch Ziffer" (Schlußbehandlung der STRFLP-Wandlung)

```
,bd47 a5 5e |>lda 5e      Exponentenzähler holen
,bd49 38 |sec          Carry vor Subtraktion setzen
,bd4a e5 5d |sbc 5d      Dezimalstellenzähler abziehen
,bd4c 85 5e |sta 5e      Ergebnis in Exponentenzähler schreiben
,bd4e f0 12 |beq bd62     Ergebnis = 0 (Z=1): ggf. Negation durchführen
,bd50 10 09 |bpl bd5b     Ergebnis < $80 (N=0): 10er-Multiplikationen auslösen
```

; nachträgliches "Linksrücken" des vorher ignorierten Dezimalpunktes durch 10er-Divisionen

```
,bd52 20 fe ba->jsr bafe "facdl0" FAC durch 10 teilen, um Dezimalpunkt nach links zu rücken
,bd55 e6 5e |inc 5e      Hilfszähler $5e (Exponentenzähler) erhöhen
,bd57 d0 f9 |bne bd52     noch nicht auf 0 (Z=0): weiter mit Division durch 10, da Dezimalpunkt noch nicht an
,bd59 f0 07 |beq bd62     jetzt noch das Vorzeichen berücksichtigen
```

; nachträgliches "Rechtsrücken" des vorher ignorierten Dezimalpunktes durch 10er-Multiplikationen

```
,bd5b 20 1e 2ba->jsr bae2 "facml0" FAC mit 10 multiplizieren, um Dezimalpunkt nach rechts zu rücken
,bd5e c6 5e      dec 5e      Hilfszähler $5e (Exponentenzähler) dekrementieren
,bd60 d0 f9      bne bd5b     noch nicht auf 0 (Z=0): weiter mit 10er-Multiplikation, da Dezimalpunkt noch nicht
                                an richtige Position gerückt wurde
,bd62 a5 67      lda 67      Vorzeichenflag (s. $bd02) holen
,bd64 30 01      bmi bd67     negativ (N=1): FAC negieren, um negatives Vorzeichen zu berücksichtigen
,bd66 60         rts         Rücksprung von Routine
-----
,bd67 4c b4 bf->jmp bfb4 "negate" FAC negieren, damit negatives Vorzeichen entsteht
-----
```

; Ziffer aus ASCII-String verarbeiten

```
,bd6a 48         pha         Ziffer auf den Stapel retten (bis $bd74)
,bd6b 24 5f      bit 5f      Dezimalpunkt-Flag testen
,bd6d 10 02      bpl bd71     aktuelle Ziffer steht nicht hinter Dezimalpunkt (N=0): keine Erhöhung des
                                Dezimalstellenzählers
,bd6f e6 5d      inc 5d      Zähler für Anzahl der Stellen hinter Dezimalpunkt erhöhen
,bd71 20 e2 ba->jsr bae2 "facml0" FAC mit 10 multiplizieren, um an neue Stelle vorzurücken
,bd74 68         pla         bei $bd6a gemerkten ASCII-Code der Ziffer wieder vom Stapel holen
,bd75 38         sec         Carry vor Subtraktion setzen
,bd76 e9 30      sbc #30      ASCII-Code von "0" abziehen, um Wert der Ziffer im Akku zu erhalten (0—9)
,bd78 20 7e bd   jsr bd7e "addafc" Bytewert im Akku zu FAC addieren
,bd7b 4c 0a bd   jmp bd0a     weiter mit nächstem Zeichen aus String
-----
```

; Hilfsroutine ADDAFC: Bytewert (im Akku enthalten) zum FAC addieren

```
,bd7e 48         pha         Bytewert bis $bd82 auf Stapel retten
,bd7f 20 0c bc   jsr bc0c "movfa" FAC in ARG kopieren (ARG := FAC)
,bd82 68         pla         bei $bd7e geretteten Bytewert wieder vom Stapel holen
,bd83 20 3c bc   jsr bc3c     in SGN-Routine einsteigen, um Bytewert aus Akku in FAC zu schreiben
,bd86 a5 6e      lda 6e      Vorzeichenbyte des ARG holen
,bd88 45 66      eor 66      Vergleich mit Vorzeichenbyte des FAC
,bd8a 85 6f      sta 6f      Ergebnis als neuen Wert des Vorzeichenvergleichs setzen
,bd8c a6 61      ldx 61      Exponent des FAC holen
,bd8e 4c 6a b8   jmp b86a "addfac" FAC (enthält jetzt Bytewert im FLPT-Format) um ARG (vorheriger FAC-Inhalt) erhöhen
-----
```

; Ziffer aus Exponent verarbeiten

```
,bd91 a5 5e    lda    5e          Exponentenzähler holen
,bd93 c9 0a    cmp    #0a         Vergleich mit 10
,bd95 90 09    bcc    bda0       Exponentenzähler < 10 (C=0): obere Stelle des Exponenten verarbeiten
,bd97 a9 64    lda    #64         100 als Exponentialwert vorbereitenderweise laden
,bd99 24 60    bit     60         Test, ob Exponent negativ ist
,bd9b 30 11    bmi    bdae       ja (N=1): 100 als Exponentenzähler setzen
,bd9d 4c 7e b9 jmp    b97e "overfl" OVERFLOW ERROR auslösen

-----

,bda0 0a      ->asl             Akku verdoppeln  } Ziffer mit 4
,bda1 0a      asl             Akku verdoppeln  } multiplizieren
,bda2 18      clc             Carry vor Addition löschen
,bda3 65 5e   adc     5e       bisherigen Exponentenzähler addieren (also ursprünglichen Wert addieren)
,bda5 0a      asl             noch einmal verdoppeln; jetzt wurde Ziffer insgesamt mit 10 multipliziert
,bda6 18      clc             Carry vor Addition bei $bda9 löschen
,bda7 a0 00   ldy    #00       Offset mit 0 initialisieren (kein Offset)
,bda9 71 7a   adc     (7a),y    Zeichen an aktueller CHRGET-Zeiger-Position addieren
,bdab 38      sec             Carry vor Subtraktion setzen
,bdac e9 30   sbc     #30       ASCII-Code der Zahl 0 subtrahieren, um ASCII-Code in numerischen Wert umzuwandeln
,bdae 85 5e   ->sta     5e       Ergebnis als neuen Exponentenzähler setzen
,bdb0 4c 30 bd jmp    bd30       nächste Ziffer des Exponenten auswerten

-----
```

; MFLPT-Konstanten für FLPASC-Umwandlung (Fließkomma in ASCII-Code)

```
:bdb3 9b 3e bc 1f fd          MFLPT-Darstellung der Zahl  99999999.9
:bdb8 9e 6e 6b 27 fd          MFLPT-Darstellung der Zahl  999999999
:bdbd 9e 6e 6b 28 00          MFLPT-Darstellung der Zahl 10000000000

-----
```

; LINOUT-Routine: Ausgabe von "in" und der aktuellen Zeilennummer

```
,bdc2 a9 71    lda    #71 <($a371)  LB der Adresse des Textes "IN" laden
,bdc4 a0 a3    ldy    #a3 >($a371)  HB der Adresse des Textes "IN" laden
,bdc6 20 da bd  jsr    bdda         bei $bbda steht "jmp strout"; Text ausgeben
,bdc9 a5 3a    lda     3a           HB der aktuellen Zeilennummer holen
,bdcb a6 39    ldx     39           LB der aktuellen Zeilennummer holen
```

}

Ausgabe des
Textes "in"
über STROUT
aktuelle Zeilennummer
als auszugebende Zahl laden

; NUMOUT-Einsprung: Ausgabe einer Zahl in X (Lowbyte) und Akku (Highbyte);
Nutzung von \$a6ea (LIST) und \$e43a (MSGNEW)

```
,bddc 85 62    sta 62      HB in Mantisse #1 schreiben } Ausgabe-Wert in
,bdcf 86 63    stx 63      LB in Mantisse #2 schreiben } Mantissenbytes schreiben
,bddl a2 90    ldx #90 %10010000 gewünschten Exponent des Ergebnisses ($80 + 16, da 216=65536) laden
,bdd3 38      sec          Carry setzen (Vorbereitung für $bdd4)
,bdd4 20 49 bc jsr bc49 "binfac" Integerwert als Fließkomma-Zahl in FAC bringen
,bdd7 20 df bd jsr bddf "flpstr" FAC in ASCII-Code umwandeln
,bdda 4c le ab jmp able "strout" Ergebnis ausgeben
```

; Einsprung: FAC in String umwandeln; dabei mit Offset 1 beginnen, um Vorzeichen zu ignorieren

```
,bddd a0 01    ldy #01      Offset 1 als Anfangswert laden
```

; FLPSTR-Einsprung: FAC als ASCII-String ab \$0100 ablegen; davor das Vorzeichen oder ein führendes
Leerzeichen setzen

```
,bddf a9 20    lda #20      ASCII-Code des Leerzeichens als Vorbelegungswert für Vorzeichen laden (positives
                          Vorzeichen ergibt führendes Leerzeichen, negatives ergibt führendes Minus)
,bdel 24 66    bit 66      Vorzeichenbyte des FAC testen
,bde3 10 02    bpl bde7     positiv (N=0): bei $bddf vorbereitetes Leerzeichen an Anfang des ASCII-Strings
,bde5 a9 2d    lda #2d      ASCII-Code von "-" laden, da negativer FAC-Inhalt vorliegt
,bde7 99 ff 00 sta 00ff,y   Vorzeichen in FAC schreiben
,bdea 85 66    sta 66      dann Vorzeichenbyte löschen, da der Akku mit $20 oder $2d nie ein gesetztes b7 hat
,bdec 84 71    sty 71      Hilfsspeicher mit Offset belegen
,bdee c8      iny          Offset auf nächste Position im ASCII-String richten
,bdef a9 30    lda #30      ASCII-Code von 0 vorbereiten
,bdf1 a6 61    ldx #61      Exponentenbyte von FAC #1 testen
,bdf3 d0 03    bne bdf8     Exponent <> 0, also FAC <> 0 (Z=0): keine Sonderbehandlung für FAC = 0
,bdf5 4c 04 bf jmp bf04     ASCII-Code von 0 (s. $bdef) an Stringende schreiben, dahinter String-Endmarkierung
```

```
,bdf8 a9 00    lda #00      Vorbelegung für Zähler der Dezimalstellen laden
,bdfa e0 80    cpx #80 %100000000 Vergleich des Exponenten mit Exponent für [0,5;1[
,bdfc f0 02    beq be00     Übereinstimmung (Z=1): Multiplikation des FAC mit 1000000000
,bdfe b0 09    bcs be09     Exponent >= $80 (C=1): Dezimalstellenzähler 0 setzen, da FAC > 1
,be00 a9 bd    lda #bd <($bddb) LB der Adresse der ROM-Konstanten 1000000000 laden } FAC mit
,be02 a0 bd    ldy #bd >($bddb) HB der Adresse der ROM-Konstanten 1000000000 laden } 1 000 000 000
,be04 20 28 ba jsr ba28 "memmult" Multiplikation des FAC mit der ROM-Konstanten } multiplizieren
,be07 a9 f7    lda #f7 ($ff-9) $ff-9 (1000000000 = 109) als Wert für Dezimalstellenzähler laden
```


,be09	85 5d	→sta 5d	in Dezimalstellenzähler schreiben, um Ausgleich für "FAC := FAC * 10 ⁹ " zu schaffen
,be0b	a9 b8	→lda #b8 <(\$bdb8)	LB der Adresse der ROM-Konstanten 999999999 laden
,be0d	a0 bd	ldy #bd >(\$bdb8)	HB der Adresse der ROM-Konstanten 999999999 laden
,be0f	20 5b bc	jsr bc5b "cmpfac"	FAC mit Konstante vergleichen
,be12	f0-le	beq be32	FAC = 999999999 (Z=1): weiter mit Sonderbehandlung ab \$be32
,be14	10-l2	bpl be28	FAC > 999999999 (N=0): weiter mit Sonderbehandlung ab \$be28
,be16	a9 b3	→lda #b3 <(\$bdb3)	LB der Adresse der ROM-Konstanten 99999999.9 laden
,be18	a0 bd	ldy #bd >(\$bdb3)	HB der Adresse der ROM-Konstanten 99999999.9 laden
,be1a	20 5b bc	jsr bc5b "cmpfac"	FAC mit Konstante vergleichen
,be1d	f0 02	beq be21	FAC = 99999999.9 (Z=1): weiter mit 10er-Multiplikation
,be1f	10 0e	bpl be2f	FAC > 99999999.9 (N=0): weiter mit Addition von 0.5
,be21	20 e2	ba →jsr bae2 "facml0"	FAC mit 10 multiplizieren, um ihn über 99999999.9 zu bringen
,be24	c6 5d	dec 5d	Dezimalstellenzähler verringern, da 10er-Multiplikation Komma nach rechts verschiebt
,be26	d0 ee	bne be16	Dezimalstellenzähler noch nicht auf 0 heruntergezählt (Z=0): weiter bei Vergleich des FAC mit der ROM-Konstanten 99999999.9
,be28	20-fe	ba →jsr bafe "facdl0"	FAC durch 10 dividieren
,be2b	e6 5d	inc 5d	dafür den Dezimalstellenzähler erhöhen, da 10er-Division Komma nach links verschiebt
,be2d	d0 dc	bne be0b	Dezimalstellenzähler nicht von \$ff auf 0 heraufgezählt (Z=0): kein Erhöhungsübertrag, also weiter mit erneutem Vergleich des FAC mit 999999999
,be2f	20 49	b8 →jsr b849 "add0.5"	FAC um 0.5 erhöhen (dient Rundung!)
,be32	20-9b	bc →jsr bc9b "facint"	FAC in Integerformat umwandeln, Kommastellen bleiben unberücksichtigt
,be35	a2 01	ldx #01	Ausgangswert für Dezimalstellenzähler laden
,be37	a5 5d	lda 5d	Dezimalstellenzähler holen
,be39	18	clc	Carry vor Addition löschen
,be3a	69 0a	adc #0a	10 addieren, da am Anfang Multiplikation mit 10 ⁹ erfolgte
,be3c	30 09	bmi be47	Ergebnis > \$7f (N=1): weitere Behandlung ab \$be47, keine Tests auf Sonderfall
,be3e	c9 0b	cmp #0b	Vergleich mit 11 (steht dann im Akku, wenn Dezimalstellenzähler vorher 1 beinhaltete)
,be40	b0 06	bcs be48	Additionsergebnis >= 11 (C=1): weitere Behandlung ab \$be48, keine Sonderbehandlung
,be42	69 ff	adc #ff %11111111	entspricht Subtraktion von 1; Carry ist wg. \$be40 gelöscht
,be44	aa	tax	Ergebnis als Wert für Dezimalstellenzähler ins X-Register bringen
,be45	a9 02	lda #02	2 laden, damit nach \$be47/\$be48 der Akku-Inhalt 0 in den Exponentenzähler kommt
,be47	38	→sec	Carry vor Subtraktion setzen
,be48	e9 02	→sbc #02	2 vom Akku subtrahieren, um Exponentenzähler zu erhalten
,be4a	85 5e	sta 5e	Ergebnis als Exponentenzähler setzen
,be4c	86 5d	stx 5d	Dezimalstellenzähler setzen
,be4e	8a	txa	Dezimalstellenzähler zwecks Test in Akku holen
,be4f	f0 02	beq be53	noch 0 Dezimalstellen zu verarbeiten (Z=1): Dezimalpunkt in String schreiben
,be51	10-l3	bpl be66	Dezimalstellenzähler < \$80 (N=0): Ziffer für String bearbeiten

; Dezimalpunkt in ASCII-String schreiben

```
,be53 a4 71 >ldy 71      Hilfsspeicher (mit Offset belegt) auslesen
,be55 a9 2e lda #2e      ASCII-Code von "." laden
,be57 c8 iny             Offset erhöhen (auf nächste Position im String stellen)
,be58 99 ff 00 sta 00ff,y ASCII-Code des Zeichens in den String schreiben
,be5b 8a txa "ldy #00"    bei $be44 in X-Register gebrachten Dezimalstellenzähler holen, der laut $be4f den
                        Wert 0 haben muß
,be5c f0 06 beq be64 "jmp" weiter wie nach jedem in String übertragenen Zeichen
-----
,be5e a9 30 lda #30      ASCII-Code von 0 vorbereitenderweise in Akku laden
,be60 c8 iny             Offset erhöhen (auf nächste Position im String stellen)
,be61 99 ff 00 sta 00ff,y Ziffer in String schreiben
,be64 84 71 sty 71       und Offset in Hilfsspeicher $71 retten
```

; Ziffern des Zahlenstrings berechnen

```
,be66 a0 00 ldy #00      0 als Offset in Tabelle ab $bfl6 laden (ab $bfl6 steht -100000000 als Mantisse,
                        darauf folgen weitere Mantissen von Zehnerpotenzen)
,be68 a2 80 ldx #80 %100000000 Zähler initialisieren (s. $be87)
,be6a a5 65 lda 65       Mantisse #4 des FAC holen
,be6c 18 clc             Carry vor Addition löschen
,be6d 79 19 bf adc bfl9,y und Mantisse #4 der Zehnerpotenz addieren
,be70 85 65 sta 65       Ergebnis als neue Mantisse #4 des FAC setzen
,be72 a5 64 lda 64       Mantisse #3 des FAC holen
,be74 79 18 bf adc bfl8,y und Mantisse #3 der Zehnerpotenz addieren
,be77 85 64 sta 64       Ergebnis als neue Mantisse #3 des FAC setzen
,be79 a5 63 lda 63       Mantisse #2 des FAC holen
,be7b 79 17 bf adc bfl7,y und Mantisse #2 der Zehnerpotenz addieren
,be7e 85 63 sta 63       Ergebnis als neue Mantisse #2 des FAC setzen
,be80 a5 62 lda 62       Mantisse #1 des FAC holen
,be82 79 16 bf adc bfl6,y und Mantisse #1 der Zehnerpotenz addieren
,be85 85 62 sta 62       Ergebnis als neue Mantisse #1 des FAC setzen
,be87 e8 inx             Zähler erhöhen (gibt später Auskunft, wann der Übertrag entstand)
,be88 b0 04 bcs be8e      Additionsübertrag (C=1): Sonderbehandlung ab $be8e
,be8a 10 de bpl be6a      Zähler bei Übertrag < $80 (N=0): erneute Addition von Zehnerpotenz zur Mantisse
,be8c 30 02 bmi be90 "jmp" ansonsten (Zähler >=$80; N=1): in Sonderbehandlung für Additionsübertrag einsteigen
-----
,be8e 30 da bmi be6a      Zähler (s. $be87/$be88) >=$80 (N=1): erneute Addition der Zehnerpotenz zur Mantisse
```

; Sonderbehandlung: Additionsübertrag und positiver X-Zähler oder X-Zähler bereits negativ

,be90	8a	>txa	Zähler in Akku holen
,be91	90 04	bcc be97	kein Additionsübertrag (C=0): nicht komplementieren und 10 addieren
,be93	49 ff	eor #ff %11111111	Komplementierung des Zählers
,be95	69 0a	adc #0a	Addition von 10
,be97	69 2f	>adc #2f	ASCII-Code von 0 minus 1 addieren
,be99	c8	iny	Offset:=Offset+1
,be9a	c8	iny	Offset:=Offset+1
,be9b	c8	iny	Offset:=Offset+1
,be9c	c8	iny	Offset:=Offset+1
			Offset um 4 erhöhen, um ihn auf die nächste Mantisse gleichen Vorzeichens in der Zehnerpotenzentabelle zu richten
,be9d	84 47	sty 47	Offset in Hilfsspeicher \$47 (sonst LB der aktuellen Variablenadresse) merken
,be9f	a4 71	ldy 71	Offset in Zahlenstring holen
,bea1	c8	iny	auf nächste Position richten
,bea2	aa	tax	weiterverarbeiteten Zähler in X-Register merken
,bea3	29 7f	and #7f %01111111	b7 ausblenden, da es nur SHIFtung des Zeichens bewirkt
,bea5	99 ff 00	sta 00ff,y	Zeichen in Ziffernstring schreiben
,bea8	c6 5d	dec 5d	Zähler für Dezimalstellen verringern
,beaa	d0 06	bne beb2	noch nicht auf 0 heruntergezählt (Z=0): Schreiben des Dezimalpunktes überspringen
,beac	a9 2e	lda #2e	ASCII-Code des Dezimalpunktes "." laden
,beae	c8	iny	Offset in Zahlenstring auf nächste Position richten
,beaf	99 ff 00	sta 00ff,y	und Dezimalpunkt in Zahlenstring schreiben
,beb2	84 71	>sty 71	Offset in Zahlenstring wieder in Hilfsspeicher retten
,beb4	a4 47	ldy 47	bei \$be9d gemerkten Offset holen
,beb6	8a	txa	bei \$bea2 gemerkten ASCII-Code der Ziffer holen
,beb7	49 ff	eor #ff %11111111	Wert komplementieren
,beb9	29 80	and #80 %10000000	alle Bits bis auf b7 löschen
,bebb	aa	tax	Ergebnis in X-Register merken
,bebc	c0 24	cpy #24	Offset mit 30 (Länge der Zehnerpotenzen-Mantissentabelle) vergleichen
,bebe	f0 04	beq bec4	Übereinstimmung (Z=1): Abbruch der Zehnerpotenzen-Addition
,bec0	c0 3c	cpy #3c	Offset mit 60 (Länge der Mantissentabelle einschließlich Zeitkonstanten)
,bec2	d0 a6	bne be6a	keine Übereinstimmung (Z=0): nächste Zehnerpotenz/Zeitkonstante addieren
,bec4	a4 71	>ldy 71	gemerkten Offset in Zahlenstring holen
,bec6	b9 ff 00	>lda 00ff,y	aktuelle Ziffer aus Zahlenstring auslesen
,bec9	88	dey	Offset verringern (auf vorhergehendes Zeichen richten)
,beca	c9 30	cmp #30	Vergleich mit ASCII-Code von 0 .
,becc	f0 f8	beq bec6	Übereinstimmung (Z=1): weiterhin nach anderer Ziffer als 0 suchen
,bece	c9 2e	cmp #2e	Vergleich mit ASCII-Code von "."
,bed0	f0 01	beq bed3	Übereinstimmung (Z=1): Offset nicht erhöhen, Dezimalpunkt nicht erfassen
,bed2	c8	iny	Offset erhöhen, um letztes Zeichen zu erfassen
,bed3	a9 2b	>lda #2b	ASCII-Code von "+" als eventuell in String zu schreibenden Wert laden

,bed5	a6 5e	ldx 5e	Zähler für 10er-Exponent holen
,bed7	f0 2e	beq bf07	Zähler = 0 (Z=1): String-Endmarkierung setzen und Ende
,bed9	10 08	bpl bee3	Zähler < \$80 (N=0): Akku (seit \$bed3: "+") in Zahlenstring schreiben
,bedb	a9 00	lda #00	0 als Wert laden, von dem der 10er-Exponentenzähler subtrahiert wird
,bedd	38	sec	Carry vor Subtraktion laden
,bede	e5 5e	sbc 5e	10er-Exponentenzähler subtrahieren
,bee0	aa	tax	und Ergebnis in X-Register merken
,bee1	a9 2d	lda #2d	ASCII-Code von "-" laden
,bee3	99 01 01	sta 0101,y	Akku-Inhalt in Zahlenstring übernehmen
,bee6	a9 45	lda #45	ASCII-Code von "E" laden
,bee8	99 00 01	sta 0100,y	Akku-Inhalt eine Position davor in Zahlenstring schreiben
,beeb	8a	txa	10er-Exponentenzähler aus X-Register (s. \$bed5, \$bee0) in Akku holen
,beec	a2 2f	ldx #2f	ASCII-Code von 0 minus 1 laden
,beee	38	sec	Carry vor Subtraktion bei \$bef0 löschen
,beef	e8	inx	10er-Exponentenzähler um 1 erhöhen
,bef0	e9 0a	sbc #0a	im Gegenzug: 10 vom 10er-Exponentenzähler subtrahieren
,bef2	b0 fb	bcs beef	kein Subtraktionsübertrag (C=0): weitere Verringerung des Akkus um 10 bei gleichzeitiger Erhöhung des 10er-Exponentenzählers im X-Register
,bef4	69 3a	adc #3a	ASCII-Code von 9 plus 1 addieren
,bef6	99 03 01	sta 0103,y	und Ergebnis in Zahlenstring schreiben
,bef9	8a	txa	10er-Exponentenzähler aus X-Register in Akku holen
,befa	99 02 01	sta 0102,y	und eine Position davor in Zahlenstring schreiben
,befd	a9 00	lda #00	Endmarkierung des Zahlenstrings laden
,beff	99 04 01	sta 0104,y	und in Zahlenstring schreiben
,bf02	f0 08	beq bf0c "jmp"	Anfangsadresse des Zahlenstrings in A/Y zurückgeben und Ende

; Aufruf dieser Routine nur von \$bdf5 (FLPSTR)

,bf04	99 ff 00	sta 00ff,y	Akku-Inhalt in Zahlenstring schreiben
,bf07	a9 00	lda #00	Endmarkierung des Zahlenstrings laden
,bf09	99 00 01	sta 0100,y	und eine Position danach in Zahlenstring schreiben
,bf0c	a9 00	lda #00 <(\$0100)	LB der Adresse des Zahlenstrings in Akku laden
,bf0e	a0 01	ldy #01 >(\$0100)	HB der Adresse des Zahlenstrings in Y laden
,bf10	60	rts	Rücksprung von Routine

; MFLPT-Konstante für die SQR-Funktion

:bf11	80 00 00 00 00	MFLPT-Darstellung der Zahl 0.5 = 2 [↑] (-1)
-------	----------------	--

; Mantissenbytes (je 4) für FLPSTR-Umwandlung; Verwendung bei \$beb, \$be82, \$be74 und \$be7b

:bf16 fa 0a 1f 00	4-Byte-Integerzahl (vorzeichenbehaftet):	-100 000 000	= -10↑8
:bf1a 00 98 96 80	4-Byte-Integerzahl (vorzeichenbehaftet):	10 000 000	= 10↑7
:bf1e ff f0 bd c0	4-Byte-Integerzahl (vorzeichenbehaftet):	-1 000 000	= -100↑6
:bf22 00 01 86 a0	4-Byte-Integerzahl (vorzeichenbehaftet):	100 000	= 10↑5
:bf26 ff ff d8 f0	4-Byte-Integerzahl (vorzeichenbehaftet):	-10 000	= -10↑4
:bf2a 00 00 03 e8	4-Byte-Integerzahl (vorzeichenbehaftet):	1 000	= 10↑3
:bf2e ff ff ff 9c	4-Byte-Integerzahl (vorzeichenbehaftet):	-100	= -10↑2
:bf32 00 00 00 0a	4-Byte-Integerzahl (vorzeichenbehaftet):	10	= 0↑1
:bf36 ff ff ff ff	4-Byte-Integerzahl (vorzeichenbehaftet):	-1	= -10↑0

; weitere Mantissenbytes (je 4) für TISTR-Umwandlung (Zeit in String umwandeln)

:bf3a ff df 0a 80	4-Byte-Integerzahl (vorzeichenbehaftet):	-2 160 000	= -6↑4 * 10↑4 = 60↑3 * (-10)
:bf3e 00 03 4b c0	4-Byte-Integerzahl (vorzeichenbehaftet):	216 000	= 6↑4 * 10↑3 = 60↑3
:bf42 ff ff 73 60	4-Byte-Integerzahl (vorzeichenbehaftet):	-36 000	= -6↑3 * 10↑3 = 60↑2 * (-10)
:bf46 00 00 0e 10	4-Byte-Integerzahl (vorzeichenbehaftet):	3 600	= 6↑3 * 10↑2 = 60↑2
:bf4a ff ff fd a8	4-Byte-Integerzahl (vorzeichenbehaftet):	-600	= -6↑2 * 10↑2 = 60↑1 * (-10)
:bf4e 00 00 00 3c	4-Byte-Integerzahl (vorzeichenbehaftet):	60	= 6↑1 * 10↑1 = 60↑1

; 31 Füllbytes ohne jede Bedeutung

:bf52 ec aa aa aa aa aa aa aa	auch in anderen ROM-Versionen
:bf5a aa aa aa aa aa aa aa aa	stehen hier keine Programmteile,
:bf62 aa aa aa aa aa aa aa aa	möglicherweise jedoch andere
:bf6a aa aa aa aa aa aa aa aa	Füllbytes

; Routine zur Basic-Funktion SQR (Token: \$ba)

,bf71 20 0c bc	jsr bc0c "movfa"	FAC in ARG bringen, da die Potenzierungsroutine dort die Basis erwartet
,bf74 a9 11	lda #11 <(\$bf11)	LB der Adresse der Konstanten 0.5 laden
,bf76 a0 bf	ldy #bf >(\$bf11)	HB der Adresse der Konstanten 0.5 laden

} (SQR(X) = X ↑ 0.5!) für Exponent laden

; MEMPOT-Routine: FAC := ARG ↑ Konstante

,bf78 20 a2 bb	jsr bba2 "movmf"	Konstante in FAC holen
----------------	------------------	------------------------

; POTAF-C-Routine: FAC := ARG ↑ FAC

,bf7b f0 70	beq bfed	FAC (Exponent) = 0 (Z=1): Routine zur EXP-Funktion ausführen	
,bf7d a5 69	lda 69	Exponent des ARG holen	
,bf7f d0 03	bne bf84	Basis <> 0 (Z=0): Sonderbehandlung für "Basis = 0, Ergebnis = 0" überspringen	
,bf81 4c f9 b8	jmp b8f9	Exponent und Vorzeichen des Ergebnisses auf 0 setzen (ARG = 0 => 0 ↑ FAC ergibt auch 0)	

,bf84 a2 4e	ldx #4e <(\$004e)	LB der Adresse von FAC-Zwischenspeicher laden	
,bf86 a0 00	ldy #00 >(\$004e)	HB der Adresse von FAC-Zwischenspeicher laden	
,bf88 20 d4 bb	jsr bbd4 "movfm"	FAC an Adresse (in diesem Fall: FAC-Zwischenspeicher ab \$4e) kopieren	
,bf8b a5 6e	lda 6e	Vorzeichenbyte des ARG auslesen	
,bf8d 10 0f	bpl bf9e	ARG (Basis) ist positiv (N=0): Sonderbehandlung für negative Basis überspringen	
,bf8f 20 cc bc	jsr bccc	Routine zur Basic-Funktion INT aufrufen	
,bf92 a9 4e	lda #4e <(\$004e)	LB der Adresse von FAC-Zwischenspeicher laden	} Originalwert mit } INT-behandeltem } Wert vergleichen
,bf94 a0 00	ldy #00 >(\$004e)	HB der Adresse von FAC-Zwischenspeicher laden	
,bf96 20 5b bc	jsr bc5b "cmpfac"	FAC mit Konstante (hier: FAC-Zwischenspeicher) vergleichen	
,bf99 d0 03	bne bf9e	keine Übereinstimmung (Z=0): keine Sonderbehandlung für ganzzahligen Exponenten	
,bf9b 98	tya	Y-Inhalt seit \$bf96 in Akku holen	
,bf9c a4 07	ldy 07	Mantisse #1 (seit \$bf96 in \$07 enthalten!) holen	
,bf9e 20 fe bb	jsr bbfe	in MOVAF-Routine so einsteigen, daß Vorzeichen im Akku Gültigkeit hat	
,bfa1 98	tya	bei \$bf9e nicht zerstörtes Y-Register in Akku holen	
,bfa2 48	pha	und auf den Stapel legen	
,bfa3 20 ea b9	jsr b9ea	Routine zur Basic-Funktion LOG aufrufen	
,bfa6 a9 4e	lda #4e <(\$004e)	LB der Adresse von FAC-Zwischenspeicher laden	} Logarithmus } mit Zwischenwert } multiplizieren
,bfa8 a0 00	ldy #00 >(\$004e)	HB der Adresse von FAC-Zwischenspeicher laden	
,bfaa 20 28 ba	jsr ba28 "memmult"	FAC mit Konstante (hier: FAC-Zwischenspeicher) multiplizieren	
,bfad 20 ed bf	jsr bfed	Routine zur EXP-Funktion aufrufen	
,bfb0 68	pla	bei \$bfa2 gemerkten Wert (Mantisse #1 eines vorherigen FAC-Inhalts) holen	
,bfb1 4a	lsr	b0 durch Rechtsverschiebung zwecks Test ins Carry holen	
,bfb2 90 0a	bcc bfbe	b0 war gelöscht (C=0): Rücksprung über RTS, keine Sonderbehandlung für anderes Vorzeichen erforderlich	
,bfb4 a5 61	lda 61	Exponentenbyte des FAC zwecks Test auslesen	
,bfb6 f0 06	beq bfbe	FAC = 0 (Z=1): kein Vorzeichen zu setzen, Rücksprung über RTS	
,bfb8 a5 66	lda 66	Vorzeichenbyte des FAC auslesen	
,bfba 49 ff	eor #ff %11111111	komplementieren, um Vorzeichen zu wechseln	
,bfbc 85 66	sta 66	und in Vorzeichenbyte des FAC zurückschreiben	
,bfbe 60	rts	Rücksprung von Routine	

; MFLPT-Konstante für die Routine zur Funktion EXP

:bfbf 81 38 aa 3b 29 MFLPT-Darstellung von $1.44269504 = \log(2)^{\uparrow(-1)}$

; Polynomtabelle für die Routine zur Funktion EXP

:bfc4 07	Polynomgrad 7 als Bytewert
:bfc5 71 34 58 3e 56	MFLPT-Darstellung des Koeffizienten a7 = 2.14987637 e-05
:bfca 74 16 7e b3 1b	MFLPT-Darstellung des Koeffizienten a6 = 1.4352314 e-04
:bfcf 77 2f ee e3 85	MFLPT-Darstellung des Koeffizienten a5 = 1.34226348 e-03
:bfd4 7a 1d 84 1c 2a	MFLPT-Darstellung des Koeffizienten a4 = 9.61401701 e-03
:bfd9 7c 63 59 58 0a	MFLPT-Darstellung des Koeffizienten a3 = 0.555051269
:bfde 7e 75 fd e7 c6	MFLPT-Darstellung des Koeffizienten a2 = 0.240226385
:bfe3 80 31 72 18 10	MFLPT-Darstellung des Koeffizienten a1 = 0.693147186
:bfe8 81 00 00 00 00	MFLPT-Darstellung des Koeffizienten a0 = 1

; Routine zur Basic-Funktion EXP (Token: \$bd)

,bfed a9 bf	lda #bf <(\$bfbf)	LB der Adresse der MFLPT-Konstanten $1.44269504 = \log(2)^{\uparrow(-1)}$ laden	} FAC := FAC * $\log(2)^{\uparrow(-1)}$
,bfef a0 bf	ldy #bf >(\$bfbf)	HB der Adresse der MFLPT-Konstanten $1.44269504 = \log(2)^{\uparrow(-1)}$ laden	
,bff1 20 28 ba	jsr ba28 "memmult"	Multiplikation des FAC mit der Konstanten	
,bff4 a5 70	lda 70	Rundungsbyte des FAC holen	
,bff6 69 50	adc #50	80 addieren	
,bff8 90 03	bcc bffd	kein Additionsübertrag (C=0): Rundung ist nicht erforderlich	
,bff9 20 23 bc	jsr bc23	FAC-Mantisse um 1 erhöhen, um Rundung zu berücksichtigen	
,bffd 4c 00 e0	jmp e000	weiter bei \$e000 im zweiten Teil des C 64-ROM	

; Fortsetzung der Routine zur Basic-Funktion EXP (Token: \$bd), die bei \$bfed beginnt

,e000 85 56	sta 56	Speicher für HB der Adresse der Funktionsroutine als Zwischenspeicher für späteres Rundungsbyte zweckentfremden
,e002 20 0f bc	jsr bc0f "movfa"	FAC in ARG kopieren
,e005 a5 61	lda 61	Exponentenbyte des FAC #1 auslesen
,e007 c9 88	cmp #88 %10001000	Vergleich mit Exponent für maximalen erlaubten Wert
,e009 90 03	bcc e00e	FAC-Exponent < Exponent der Obergrenze (C=0): keine Sonderbehandlung
,e00b 20 d4 ba	jsr bad4	OVERFLOW ERROR bei positivem FAC-Vorzeichen, FAC=0 bei negativem FAC-Inhalt zurückgeben
,e00e 20 cc bc	jsr bccc	Routine zur Basic-Funktion INT ausführen

,e011	a5 07	lda 07	Mantisse #4 (seit \$e00e in \$07 enthalten) holen	
,e013	18	clc	Carry vor Addition löschen	
,e014	69 81	adc #81 %10000001	129 addieren	
,e016	f0 f3	beq e00b	Ergebnis = 0, also Akku vor Addition = 127 (Z=1): Sonderbehandlung	
,e018	38	sec	Carry vor Subtraktion setzen	
,e019	e9 01	sbc #01	1 abziehen (im Zusammenspiel mit \$e013/\$e014: Addition von \$80 = 128)	
,e01b	48	pha	Ergebnis auf dem Stapel merken	
,e01c	a2 05	ldx #05	Dekrementierzähler initialisieren (6 Byte werden übertragen)	
,e01e	b5 69	>lda 69,x	Byte aus ARG in Akku holen	Inhalte von
,e020	b4 61	ldy 61,x	Byte aus FAC in Y holen	FAC
,e022	95 61	sta 61,x	Byte aus FAC (s. \$e020) in ARG schreiben	und
,e024	94 69	sty 69,x	Byte aus ARG (s. \$e01e) in FAC schreiben	ARG
,e026	ca	dex	Dekrementierzähler herunterzählen (auf nächstes Byte stellen)	byteweise
,e027	10 f5	bpl e01e	noch nicht auf \$ff heruntergezählt (N=0): weiter in Austausch-Schleife	vertauschen
,e029	a5 56	lda 56	bei \$e000 in zweckentfremdetem Hilfsspeicher abgelegtes Rundungsbyte holen	
,e02b	85 70	sta 70	und als FAC-Rundungsbyte setzen	
,e02d	20 53 b8	jsr b853 "sub"	FAC := FAC - ARG	} Berechnung von
,e030	20 b4 bf	jsr bfb4 "negate"	FAC := FAC * (-1)	
				FAC := ARG - FAC
,e033	a9 c4	lda #c4 <(\$bfc4)	LB der Adresse des EXP-Polynoms laden	} Berechnung des Näherungspolynoms,
,e035	a0 bf	ldy #bf >(\$bfc4)	HB der Adresse des EXP-Polynoms laden	
				dessen Polynomtabelle ab \$bfc4
,e037	20 59 e0	jsr e059 "poly"	Polynom berechnen	im Speicher steht
,e03a	a9 00	lda #00	Initialisierungswert für Vorzeichenvergleichsbyte laden	
,e03c	85 6f	sta 6f	Flag für Ergebnis des FAC-ARG-Vorzeichenvergleichs löschen	
,e03e	68	pla	bei \$e01b gemerkten Wert (Exponent) holen	
,e03f	20 b9 ba	jsr bab9	Exponent des Ergebnisses ermitteln (Exponent im Akku zu FAC-Exponent addieren)	
,e042	60	rts	Rücksprung von Routine	

; POLYX-Routine: Berechnung eines Polynoms der Form $a_1x^1 + a_2x^3 + a_3x^5 + a_4x^7$

,e043	85 71	sta 71	LB der Adresse der Polynomtabelle in LB von \$71/\$72 schreiben	} Zeiger \$71/\$72 auf Polynomtabelle
,e045	84 72	sty 72	HB der Adresse der Polynomtabelle in HB von \$71/\$72 schreiben	
,e047	20 ca bb	jsr bbca "movt3"	FAC #1 in FAC #3 (ab \$57) kopieren	
,e04a	a9 57	lda #57 *\$57	Zeropage-Adresse des FAC #3 laden	} FAC #1 := FAC #1 * FAC #3 (da FAC #3 = FAC #1: Quadrierung des FAC)
,e04c	20 28 ba	jsr ba28 "memmult"	FAC := FAC * Konstante	
,e04f	20 5d e0	jsr e05d	Polynomberechnungsroutine für $a_0x^0 + a_1x^1 + a_2x^2 + \dots$ aufrufen	
,e052	a9 57	lda #57 <(\$0057)	LB der Adresse des FAC #3 laden	} FAC #1 := FAC #1 * FAC #3 (Ergebnis mit FAC #3 multiplizieren)
,e054	a0 00	ldy #00 >(\$0057)	HB der Adresse des FAC #3 laden	
,e056	4c 28 ba	jmp ba28 "memmult"	FAC := FAC * Konstante	

; POLY-Routine: Berechnung eines Polynoms der Form $a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + a_3 \cdot x^3 + \dots$

,e059	85 71	sta 71	LB der Adresse der Polynomtabelle in LB von \$71/\$72 schreiben	} Zeiger \$71/\$72 auf Polynomtabelle
,e05b	84 72	sty 72	HB der Adresse der Polynomtabelle in HB von \$71/\$72 schreiben	
,e05d	20 c7 bb	jsr bbc7 "movt4"	FAC #1 in FAC #4 kopieren (FAC #4 := FAC #1)	
,e060	b1 71	lda (71),y	da Y=0 (wg. \$e05d), wird hier der Polynomgrad ausgelesen	} Polynomgrad aus
,e062	85 67	sta 67	und in \$67 gemerkt (SGNFLG für Polynomauswertung)	
,e064	a4 71	ldy 71	LB der Adresse der Polynomtabelle holen	} Polynomtabelle entnehmen, in
,e066	c8	iny	und um 1 erhöhen	
,e067	98	tya	dann in Akku befördern	} \$67 merken und
,e068	d0 02	bne e06c	kein Erhöhungsübertrag bei \$e066 (Z=0):	
			HB des Polynomtabellezeigers nicht erhöhen	} Zeiger für Polynomtabelle
,e06a	e6 72	inc 72	HB der Adresse der Polynomtabelle erhöhen	
,e06c	85 71	sta 71	LB der Adresse der Polynomtabelle setzen	} auf ersten Koeffizienten
,e06e	a4 72	ldy 72	HB der Adresse der Polynomtabelle holen	
,e070	20 28	ba→jsr ba28 "memmult"	FAC := FAC * Konstante (in diesem Fall: Koeffizient aus Polynomtabelle)	} stellen
,e073	a5 71	lda 71	LB der Adresse der Polynomtabelle holen	
,e075	a4 72	ldy 72	HB der Adresse der Polynomtabelle holen	} Zeiger \$71/\$72 (weist auf
,e077	18	clc	Carry vor Addition löschen	
,e078	69 05	adc #05	Anzahl der Bytes eines Koeffizienten in der Polynomtabelle	} Position in Polynomtabelle)
			addieren (Hilfszeiger \$71/\$72 auf nächsten Koeffizienten stellen)	
,e07a	90 01	bcc e07d	kein Additionsübertrag (C=0): HB nicht erhöhen	} um 5 erhöhen,
,e07c	c8	iny	HB der Adresse der Polynomtabelle erhöhen	
,e07d	85 71	sta 71	LB des Zeigers auf die Polynomtabelle neu setzen	} nächsten Koeffizient zeigt
,e07f	84 72	sty 72	HB des Zeigers auf die Polynomtabelle neu setzen	
,e081	20 67 b8	jsr b867 "movt4"	FAC #1 in FAC #4 kopieren	
,e084	a9 5c	lda #5c <(\$005c)	LB der Adresse des FAC #4 laden	
,e086	a0 00	ldy #00 >(\$005c)	HB der Adresse des FAC #4 laden	
,e088	c6 67	dec 67	Zähler SGNFLG dekrementieren (enthält Anzahl der noch zu bearbeitenden Koeffizienten)	
,e08a	d0 e4	bne e070	noch nicht auf 0 heruntergezählt (Z=0): weiter in Polynomauswertungsschleife	
,e08c	60	rts	Rücksprung von Routine	

; MFLPT-Konstanten für die Routine zur Basic-Funktion RND

,e08d	98 35 44 7a 00	MFLPT-Darstellung der Zahl 11 879 546
,e092	68 28 b1 46 00	MFLPT-Darstellung der Zahl 3.927 677 74 e-08

; Routine zur Basic-Funktion RND (Token: \$bb)

,e097	20 2b bc	jsr bc2b "sign"	Vorzeichen des RND-Arguments holen
,e09a	30 37	bmi e0d3	negatives Argument (N=1): Sonderbehandlung ab \$e0d3 anspringen
,e09c	d0 20	bne e0be	positives Argument, aber nicht 0 (Z=0): Sonderbehandlung ab \$e0be

; Sonderbehandlung für RND(0): Ergebnis aus CIA-Timer entnehmen

,e09e	20 f3 ff	jsr fff3 "iobase"	Basis-Adresse des I/O-Bausteins (CIA) nach X/Y holen	} Hilfszeiger \$22/\$23 auf Basisadresse der CIA richten
,e0a1	86 22	stx 22	LB der Adresse in LB des Hilfszeigers \$22/\$23 schreiben	
,e0a3	84 23	sty 23	HB der Adresse in HB des Hilfszeigers \$22/\$23 schreiben	
,e0a5	a0 04	ldy #04	Offset mit 4 initialisieren (auf LB des Timers B in CIA 1 richten)	} Inhalt von Timer
,e0a7	b1 22	lda (22),y	LB des Timers B in CIA 1 holen	
,e0a9	85 62	sta 62	und als Mantisse #1 des FAC setzen	} A und B
,e0ab	c8	iny "ldy #05"	Offset von 4 auf 5 erhöhen (auf HB des Timers B in CIA 1 richten)	
,e0ac	b1 22	lda (22),y	HB des Timers B in CIA 1 holen	} aus CIA 1 in FAC-
,e0ae	85 64	sta 64	und als Mantisse #3 des FAC setzen	
,e0b0	a0 08	ldy #08	Offset mit 4 initialisieren (auf LB des Timers A in CIA 1 richten)	} Mantis- se holen
,e0b2	b1 22	lda (22),y	LB des Timers A in CIA 1 holen	
,e0b4	85 63	sta 63	und als Mantisse #2 des FAC setzen	}
,e0b6	c8	iny "ldy #09"	Offset von 8 auf 9 erhöhen (auf HB des Timers A in CIA 1 richten)	
,e0b7	b1 22	lda (22),y	HB des Timers A in CIA 1 holen	}
,e0b9	85 65	sta 65	und als Mantisse #4 des FAC setzen	
,e0bb	4c e3 e0	jmp e0e3 "strnex"	in FAC-Mantisse stehende Werte von RND zurückgeben	

; Sonderbehandlung für RND(positiv): Ergebnis aus "seed"-Wert (letztes RND-Ergebnis) berechnen

,e0be	a9 8b	lda #8b <(\$008b)	LB der Adresse des SEED-Speichers laden	} SEED-Wert (letztes Ergebnis der RND-Funktion) in den FAC übertragen
,e0c0	a0 00	ldy #00 >(\$008b)	HB der Adresse des SEED-Speichers laden	
,e0c2	20 a2 bb	jsr bba2 "movmf"	FAC := Konstante (Konstante in FAC kopieren)	} FAC (SEED-Wert) mit MFLPT-Konstante 11 879 546 multiplizieren
,e0c5	a9 8d	lda #8d <(\$e08d)	LB der Adresse der MFLPT-Konstanten 11 879 546 laden	
,e0c7	a0 e0	ldy #e0 >(\$e08d)	HB der Adresse der MFLPT-Konstanten 11 879 546 laden	} FAC um Konstante 3.927 677 74 e-08 erhöhen
,e0c9	20 28 ba	jsr ba28 "memmult"	FAC := FAC * Konstante	
,e0cc	a9 92	lda #92 <(\$e092)	LB der Adresse der MFLPT-Konstanten 3.927 677 74 e-08 laden	}
,e0ce	a0 e0	ldy #e0 >(\$e092)	HB der Adresse der MFLPT-Konstanten 3.927 677 74 e-08 laden	
,e0d0	20 67 b8	jsr b867 "addmem"	FAC := FAC + Konstante	

; Sonderbehandlung für RND(negativ): Mantissenbytes des Arguments vertauschen, danach Ergebnis als SEED-Wert setzen

,e0d3	a6 65	→ldx 65	Mantisse #4 des FAC in X holen	} Mantisse #1 mit Mantisse #4 vertauschen Mantisse #2 mit Mantisse #3 vertauschen	} Vertauschung der Mantissenbytes des FAC
,e0d5	a5 62	lda 62	Mantisse #1 des FAC in Akku holen		
,e0d7	85 65	sta 65	Mantisse #1 in Mantisse #4 des FAC schreiben		
,e0d9	86 62	stx 62	Mantisse #4 in Mantisse #1 des FAC schreiben		
,e0db	a6 63	ldx 63	Mantisse #2 des FAC in X holen		
,e0dd	a5 64	lda 64	Mantisse #3 des FAC in Akku holen		
,e0df	85 63	sta 63	Mantisse #3 in Mantisse #2 des FAC schreiben		
,e0e1	86 64	stx 64	Mantisse #2 in Mantisse #3 des FAC schreiben		

; STRNEX-Einsprung: Rückgabe der im FAC stehenden Mantisse als Funktionsergebnis

,e0e3	a9 00	lda #00	Initialisierungswert für Vorzeichenbyte des FAC laden	} FAC #1 bekommt positives Vorzeichen alten Exponent als Rundungsbyte setzen Exponent für]0;1[setzen und FAC normalisieren FAC-Inhalt als SEED-Wert merken
,e0e5	85 66	sta 66	und in FAC-Vorzeichenbyte schreiben	
,e0e7	a5 61	lda 61	Exponentenbyte des FAC holen	
,e0e9	85 70	sta 70	und in Rundungsbyte des FAC schreiben	
,e0eb	a9 80	lda #80 %10000000	Exponent für RND-Rückgabebereich]0;1[laden	
,e0ed	85 61	sta 61	und in Exponentenbyte des FAC schreiben	
,e0ef	20 d7 b8	jsr b8d7 "normal"	FAC normalisieren	
,e0f2	a2 8b	ldx #8b <(\$008b)	LB der Adresse des SEED-Speichers laden	
,e0f4	a0 00	ldy #00 >(\$008b)	HB der Adresse des SEED-Speichers laden	
,e0f6	4c d4 bb	jmp bbd4 "movfm"	FAC in Speicher als MFLPT-Zahl kopieren	

; EREXIT-Routine: Fehlerbehandlung nach I/O-Operationen des Basic-Interpreters

Im Akku steht die Fehlernummer. Hierher erfolgt nur von \$eldl ein Sprung.

,e0f9	c9 f0	→cmp #f0 %11110000	Vergleich mit Fehlercode bei OPEN oder CLOSE im RS232-Betrieb
,e0fb	d0 07	bne e104	keine Übereinstimmung (Z=0): Fehlermeldung erzeugen
,e0fd	84 38	sty 38	HB der aktuellen Adresse als HB der obersten Basic-Adresse setzen
,e0ff	86 37	stx 37	LB der aktuellen Adresse als LB der obersten Basic-Adresse setzen
,e101	4c 63	a6 jmp a663	in Routine zum Basic-Befehl CLR einsteigen

,e104	aa	→tax	Fehlernummer in X-Register laden
,e105	d0 02	bne e109	Fehlernummer <> 0 (Z=0): Fehlermeldung zur Fehlernummer im X-Register erzeugen
,e107	a2 1e	ldx #1e	Fehlernummer für BREAK ERROR laden
,e109	4c 37	a4 jmp a437 "error"	Fehlereinsprung aufrufen

; Basic-Einsprünge für Kernal-Routinen (Aufruf der entsprechenden Routine und Auswertung von Fehlern)

; Basic-BSOUT (Nutzung nur von \$ab47 aus)

```
,el0c 20 d2 ff jsr ffd2 "bsout" Kernal-Einsprung für BSOUT aufrufen
,el0f b0 e8 bcs e0f9 "erexit" I/O-Fehler (C=1): Fehlerbehandlung
,el11 60 rts Rücksprung von Routine, da kein I/O-Fehler vorlag
```

; Basic-BASIN (Nutzung nur von \$a562 aus)

```
,el12 20 cf ff jsr ffcf "basin" Kernal-Einsprung für BASIN aufrufen
,el15 b0 e2 bcs e0f9 "erexit" I/O-Fehler (C=1): Fehlerbehandlung
,el17 60 rts Rücksprung von Routine, da kein I/O-Fehler vorlag
```

; Basic-CKOUT (Nutzung nur von \$aa93 aus)

```
,el18 20 ad e4 jsr e4ad "bckout" spezielle Basic-CKOUT-Routine aufrufen
,el1b b0 dc bcs e0f9 "erexit" I/O-Fehler (C=1): Fehlerbehandlung
,el1d 60 rts Rücksprung von Routine, da kein I/O-Fehler vorlag
```

; Basic-CHKIN (Nutzung nur von \$ab8f und \$abaf aus)

```
,elle 20 c6 ff jsr ffc6 "chkin" Kernal-Einsprung für CHKIN aufrufen
,el21 b0 d6 bcs e0f9 "erexit" I/O-Fehler (C=1): Fehlerbehandlung
,el23 60 rts Rücksprung von Routine, da kein I/O-Fehler vorlag
```

; Basic-GETIN (Nutzung nur von \$ac35 aus)

```
,el24 20 e4 ff jsr ffe4 "getin" Kernal-Einsprung für GETIN aufrufen
,el27 b0 d0 bcs e0f9 "erexit" I/O-Fehler (C=1): Fehlerbehandlung
,el29 60 rts Rücksprung von Routine, da kein I/O-Fehler vorlag
```

; Routine zum Basic-Befehl SYS (Token: \$9e)

```
,el2a 20 8a ad jsr ad8a "frmevl" numerischen Ausdruck (SYS-Zieladresse) auswerten
,el2d 20 f7 b7 jsr b7f7 "facwrdr" und als 2-Byte-Integerzahl nach $14/$15 berechnen
```


,el30	a9 e1	lda #e1 >(\$el46)	HB der Rücksprungadresse in SYS-Routine	} \$el46 als Rücksprungadresse für RTS aus der über SYS aufgerufenen Maschinenroutine auf den Stapel legen
,el32	48	pha	auf den Stapel legen	
,el33	a9 46	lda #46 <(\$el46)	LB der Rücksprungadresse in SYS-Routine	
,el35	48	pha	auf den Stapel legen	
,el36	ad 0f 03	lda 030f	Speicher für 6502-Statusregister auslesen	} Prozessorregister einschließlich CPU-Status nach letzter SYS-Routine aus Hilfsspeicher \$030c-\$030f holen
,el39	48	pha	und bis \$el43 auf den Stapel legen	
,el3a	ad 0c 03	lda 030c	Speicher für 6502-Register A (Akku) auslesen	
,el3d	ae 0d 03	ldx 030d	Speicher für 6502-Register X auslesen	
,el40	ac 0e 03	ldy 030e	Speicher für 6502-Register Y auslesen	
,el43	28	plp	bei \$el39 auf den Stapel gelegten Prozessorstatus herstellen	
,el44	6c 14 00	jmp(0014)	an SYS-Routine springen (s. \$el2d)	

; Hierher erfolgt Rücksprung, wenn die bei \$el44 angesprungene SYS-Routine mit RTS endet (s. \$el30-\$el35)

,el47	08	php	Status auf den Stapel retten
,el48	8d 0c 03	sta 030c	Akku in dafür reservierten Speicher schreiben
,el4b	8e 0d 03	stx 030d	X-Register in dafür reservierten Speicher schreiben
,el4e	8c 0e 03	sty 030e	Y-Register in dafür reservierten Speicher schreiben
,el51	68	pla	Status vom Stapel (s. \$el47) in den Akku holen
,el52	8d 0f 03	sta 030f	und in dafür reservierten Speicher schreiben
,el55	60	rts	Rücksprung von Routine

; Routine zum Basic-Befehl SAVE (Token: \$94)

,el56	20 d4 e1	jsr eld4 "getlsv"	Parameter für LOAD, SAVE und VERIFY holen	
,el59	a6 2d	ldx 2d	LB der Endadresse des Basic-Programms im Speicher holen	} Endadresse für SAVE = Endadresse des Programms
,el5b	a4 2e	ldy 2e	HB der Endadresse des Basic-Programms im Speicher holen	
,el5d	a9 2b	lda #2b *\$2b	Zeropage-Adresse des Zeigers \$2b/\$2c (Anfangsadresse des Basic-Programms)	
,el5f	20 d8 ff	jsr ffd8 "save"	Kernal-Einsprung für SAVE aufrufen	
,el62	b0 95	bcs e0f9 "erexit"	I/O-Fehler (C=1): Fehlerbehandlung	
,el64	60	rts	Rücksprung von Routine	

; Routine zum Basic-Befehl VERIFY (Token: \$95)

,el65	a9 01	lda #01	Verify-Flag für LOAD/VERIFY-Kernal-Routine laden
-------	-------	---------	--

; Routine zum Basic-Befehl LOAD (Token: \$93)

```
,el67 2c a9 00 "bit" lda #00      Load-Flag für LOAD/VERIFY-Kernal-Routine laden
,el6a 85 0a     sta 0a          LOAD/VERFIY-Flag des Basic-Interpreters setzen
,el6c 20 d4 e1  jsr eld4 "getlsv" Parameter für LOAD, SAVE und VERIFY holen
,el6f a5 0a     lda 0a          LOAD/VERIFY-Flag des Basic-Interpreters holen
,el71 a6 2b     ldx 2b          LB der Anfangsadresse des Basic-Programms im Speicher holen
,el73 a4 2c     ldy 2c          HB der Anfangsadresse des Basic-Programms im Speicher holen
,el75 20 d5 ff  jsr ffd5 "load"  LOAD/VERIFY-Routine über Kernal-Einsprung aufrufen
,el78 b0 57     ↘ bcs eld1       I/O-Fehler (C=1): Fehlerbehandlung
,el7a a5 0a     ↗ lda 0a        LOAD/VERIFY-Flag des Basic-Interpreters holen
,el7c f0 17     ↗ beq el195     LOAD (Z=1): weitere Behandlung für LOAD ab $el95
```

; weitere Behandlung für VERIFY, nachdem Programm schon verifiziert ist

```
,el7e a2 1c     ldx #1c          Fehlernummer für VERIFY ERROR laden, um diesen Fehler vorzubereiten
,el80 20 b7 ff  jsr ffb7 "readst" Statusbyte des Betriebssystems auslesen
,el83 29 10     and #10 %00010000 alle Bits bis auf b4 löschen, also b4 testen
,el85 d0 17     ↗ bne el19e     b4 ist gesetzt (Z=0): VERIFY ERROR auslösen (s. $el7e)
,el87 a5 7a     lda 7a !lda 7b! LB des CHRGET-Zeigers auslesen; richtig wäre "lda $7b", da nur das HB darüber
                                informieren kann, ob es sich um eine Direktmodus-Ausführung handelt!
,el89 c9 02     cmp #02 >($02xx) Vergleich des LB (vermeintliches HB) mit HB des Systemeingabepuffers
,el8b f0 07     ↗ beq el194     Übereinstimmung (Z=1): RTS, da vermeintlich Direktmodus
,el8d a9 64     ↗ lda #64 <($a364) LB der Adresse des Textes "verifying ok" laden } VERIFYING OK
,el8f a0 a3     ↗ ldy #a3 >($a364) HB der Adresse des Textes "verifying ok" laden } als Systemmeldung
,el91 4c 1e ab  ↗ jmp abe "strout" Text ausgeben } ausgeben
-----
,el94 60       ↗ rts          Rücksprung von Routine
-----
```

; weitere Behandlung für LOAD, nachdem Programm schon geladen ist (hierher wird von \$el7c aus verzweigt)

```
,el95 20 b7 ff ↗ jsr ffb7 "readst" Statusbyte des Betriebssystems auslesen
,el98 29 bf     and #bf %10111111 b6 löschen
,el9a f0 05     ↗ beq el1a1     alle Bits (außer b6) gelöscht (Z=1): keine Fehlermeldung erzeugen
,el9c a2 1d     ↗ ldx #1d       Fehlernummer für LOAD ERROR laden } LOAD ERROR
,el9e 4c 37 a4 ↗ jmp a437 "error" Fehlereinsprung aufrufen } auslösen
-----
,el1a1 a5 7b     ↗ lda 7b       HB des CHRGET-Zeigers auslesen
,ela3 c9 02     cmp #02 >($02xx) mit HB des Systemeingabepuffers vergleichen
,ela5 d0 0e     ↗ bne elb5     keine Übereinstimmung (Z=0): Sonderbehandlung für "LOAD im Programm"
```

; Sonderbehandlung: LOAD im Direktmodus

,ela7	86 2d	stx 2d	LB der Endadresse von LOAD als LB der Programm-Endadresse setzen	
,ela9	84 2e	sty 2e	HB der Endadresse von LOAD als HB der Programm-Endadresse setzen	
,elab	a9 76	lda #76 <(\$a376)	LB der Adresse des Textes "READY." laden	} Ausgabe der Meldung
,elad	a0 a3	ldy #a3 >(\$a376)	HB der Adresse des Textes "READY." laden	
,elaf	20 1e ab	jsr able "strout"	Text ausgeben	} READY.
,elb2	4c 2a a5	jmp a52a	in CLR-Routine, Linkpointer-Neuberechnung und Warmstart einsteigen	

; Sonderbehandlung: LOAD im Programm

,elb5	20 8e a6	jsr a68e "stxtpt"	CHRGET-Zeiger initialisieren (auf Programmanfang stellen)
,elb8	20 33 a5	jsr a533 "lnkprg"	Linkpointer-Neuberechnung für gesamtes Programm (Programmzeilen binden)
,elbb	4c 77 a6	jmp a677	Routine zum Basic-Befehl RESTORE aufrufen, dann weitere Initialisierungen; daraufhin wird durch den automatischen Rücksprung zur Interpreterschleife das neu geladene Programm gestartet

; Routine zum Basic-Befehl OPEN (Token: \$9f)

,elbe	20 19 e2	jsr e219 "oclpdr"	OPEN/CLOSE-Parameter auswerten
,elcl	20 c0 ff	jsr ffc0 "open"	Kernal-Einsprung für OPEN aufrufen
,elc4	b0 0b	bcs eldl	I/O-Fehler (C=1): Fehlerbehandlung
,elc6	60	rts	Rücksprung von Routine

; Routine zum Basic-Befehl CLOSE (Token: \$a0)

,elc7	20 19 e2	jsr e219 "oclpdr"	OPEN/CLOSE-Parameter auswerten
,elca	a5 49	lda 49	logische Filenummer in Akku holen
,elcc	20 c3 ff	jsr ffc3 "close"	Kernal-Einsprung für CLOSE aufrufen
,elcf	90 c3	bcc el94	kein I/O-Fehler (C=1): RTS-Befehl anspringen

; Einsprung: Aufruf von EREXIT (I/O-Fehlerbehandlung)

,eldl	4c f9 e0	jmp e0f9 "erexit"	zur EREXIT-Routine springen
-------	----------	-------------------	-----------------------------

; GETLSV-Routine: Parameter für LOAD, SAVE und VERIFY auswerten

```
,eld4 a9 00    lda #00          0 als Länge des Filenamens (also kein Filename) laden    } "kein Filename vorhanden"
,eld6 20 bd ff  jsr ffbd "setnam" Filenamen setzen                                } einstellen
,eld9 a2 01    ldx #01          Geräteadresse 1 laden                          } 1 (Datasette) als Gerät und
,eldb a0 00    ldy #00          Sekundäradresse 0 laden                        } 0 als Sekundäradresse
,eldd 20 ba ff  jsr ffba "setpar" Fileparameter setzen                        } einstellen
,e200 20 06 e2  jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e201 20 57 e2  jsr e257 "namlsv" Filenamen für LOAD/SAVE/VERIFY auswerten
,e202 20 06 e2  jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e203 20 00 e2  jsr e200 "combyt" Komma und numerischen Parameter auswerten
,e204 a0 00    ldy #00          Sekundäradresse 0 vorbereiten
,e205 86 49    stx 49           bei $e205 ausgewerteten Bytewert als Geräteadresse setzen
,e206 20 ba ff  jsr ffba "setpar" Fileparameter setzen
,e207 20 06 e2  jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e208 20 00 e2  jsr e200 "combyt" Komma und numerischen Parameter auswerten
,e209 8a       txa             ausgewerteten numerischen Parameter in Akku als Sekundäradresse holen
,e20a a8       tay             Sekundäradresse ins Y-Register bringen
,e20b a6 49    ldx 49          bei $e20b gemerkte Geräteadresse in X-Register holen
,e20c 4c ba ff  jmp ffba "setpar" Fileparameter setzen und Rücksprung
```

; COMBYT-Routine: Komma und darauffolgenden numerischen Parameter auswerten

```
,e200 20 0e e2  jsr e20e "chkcpr" Prüfroutine, ob Komma und weitere Parameter folgen, aufrufen
,e201 4c 9e b7  jmp b79e "getbyt" Bytewert (hinter Komma) auswerten
```

; PARTST-Routine: Test, ob weitere Parameter folgen (wenn nicht: Rücksprung von GETLSV-Routine)

```
,e206 20 79 00  jsr 0079 "chrgot" Zeichen an CHRGET-Zeiger-Position in Akku holen
,e209 d0 02     bne e20d         keine Endmarkierung (Z=0): Rücksprung über RTS
,e20b 68       pla             LB der Rücksprungadresse von PARTST löschen    } Wirkung: später erfolgt Rücksprung
,e20c 68       pla             HB der Rücksprungadresse von PARTST löschen    } an die GETLSV übergeordnete Routine
,e20d 60       >rts            Rücksprung von PARTST- oder GETLSV-Routine (s. $e209)
```

; CHKCPR-Routine: Prüfroutine, ob Komma und weitere Parameter folgen (sonst SYNTAX ERROR)

```
,e20e 20 fd ae  jsr aefd "chkcom" Prüfroutine, ob Komma folgt, aufrufen
,e211 20 79 00  jsr 0079 "chrgot" Zeichen an CHRGET-Zeiger-Position in Akku holen
```



```
,e214 d0 f7    Lbne e20d    keine Endmarkierung (Z=0): Rücksprung über RTS
,e216 4c 08 af jmp af08 "synerr" SYNTAX ERROR, da hinter Komma ein Zeilen- oder Befehlsende folgt
-----
```

; OCLPAR-Routine: Parameter hinter OPEN oder CLOSE auswerten

```
,e219 a9 00    lda #00      0 als Länge des Filenamens laden (= "kein Filename")
,e21b 20 bd ff jsr ffbf "setnam" Filenamens setzen
,e21e 20 11 e2 jsr e211     in CHKCPR-Routine hinter "jsr chkcom" einsteigen, damit nur sichergestellt wird, daß
                               weitere Parameter folgen
,e221 20 9e b7 jsr b79e "getbyt" Bytewert (Filenummer) aus Basic-Text in X-Register holen
,e224 86 49    stx 49       Filenummer in Hilfsspeicher $49 (sonst FOR/NEXT-Variablenzeiger-LB) merken
,e226 8a       txa         und in Akku bringen (zwecks Übergabe an SETPAR)
,e227 a2 01    ldx #01     Vorbelegungswert 1 (Datasette) als Gerätenummer laden
,e229 a0 00    ldy #00     Vorbelegungswert 0 als Sekundäradresse laden
,e22b 20 ba ff jsr ffba "setpar" Fileparameter setzen
,e22e 20 06 e2 jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e231 20 00 e2 jsr e200 "combyt" Komma und numerischen Parameter auswerten
,e234 86 4a    stx 4a      ausgewerteten Parameter (Geräteadresse) in Hilfsspeicher $4a (sonst
                               FOR/NEXT-Variablenzeiger-HB) merken
,e236 a0 00    ldy #00     Sekundäradresse 0 als Vorbelegungswert laden
,e238 a5 49    lda 49       bei $e224 gemerkte Filenummer aus Hilfsspeicher holen
,e23a e0 03    cpx #03     Vergleich der Geräteadresse mit Gerätenummer für Bildschirm
,e23c 90 01    bcc e23f    kleinere Gerätenummer (C=0): $00 bleibt Vorbelegungswert für Sekundäradresse
,e23e 88       dey "ldy #ff" $ff als Vorbelegungswert für die Sekundäradresse laden
,e23f 20 ba ff jsr ffba "setpar" Fileparameter setzen
,e242 20 06 e2 jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e245 20 00 e2 jsr e200 "combyt" Komma und numerischen Parameter auswerten
,e248 8a       txa         Sekundäradresse (bei $e245 ausgewertet) holen
,e249 a8       tay         und ins Y-Register bringen
,e24a a6 4a    ldx 4a       Geräteadresse (s. $e234) laden
,e24c a5 49    lda 49       Filenummer (s. $e224) laden
,e24e 20 ba ff jsr ffba "setpar" Fileparameter setzen
,e251 20 06 e2 jsr e206 "partst" Test auf weitere Parameter; wenn nicht: Rücksprung von GETLSV
,e254 20 0e e2 jsr e20e "chkcpr" Prüfroutine, ob Komma und weitere Parameter folgen, aufrufen
,e257 20 9e ad jsr ad9e "frmevl" beliebigen Ausdruck aus Basic-Text auswerten
,e25a 20 a3 b6 jsr b6a3     String aus Basic-Text weiterverarbeiten
,e25d a6 22    ldx 22       LB der Stringadresse als LB der Adresse des Filenamens laden
,e25f a4 23    ldy 23       HB der Stringadresse als HB der Adresse des Filenamens laden
,e261 4c bd ff jmp ffbf "setnam" Filenamens setzen; Stringlänge steht seit $e25a im Akku
-----
```

} String als
Filename
übergeben

; Routine zur Basic-Funktion COS (Token: \$be)

,e264	a9 e0	lda #e0 <(\$e2e0)	LB der Adresse der Konstanten $\pi/2$ laden	} FAC um $\pi/2$ erhöhen (FAC := FAC + $\pi/2$), da $\cos(x) = \sin(x + \pi/2)$
,e266	a0 e2	ldy #e2 >(\$e2e0)	HB der Adresse der Konstanten $\pi/2$ laden	
,e268	20 67 b8	jsr b867 "addmem"	FAC um Konstante erhöhen	

; Routine zur Basic-Funktion SIN (Token: \$bf)

,e26b	20 0c bc	jsr bc0c "movfa"	FAC in ARG kopieren (ARG := FAC)	
,e26e	a9 e5	lda #e5 <(\$e2e5)	LB der Adresse der Konstanten 2π laden	
,e270	a0 e2	ldy #e2 >(\$e2e5)	HB der Adresse der Konstanten 2π laden	
,e272	a6 6e	ldx 6e	Vorzeichenbyte des ARG auslesen	
,e274	20 07 bb	jsr bb07	in DIVF10-Routine so einsteigen, daß FAC := FAC / 2π berechnet wird	
,e277	20 0c bc	jsr bc0c "movfa"	FAC in ARG kopieren	
,e27a	20 cc bc	jsr bccc	Routine zur Basic-Funktion INT aufrufen	
,e27d	a9 00	lda #00	Initialisierungswert für Vorzeichenvergleichsbyte laden	
,e27f	85 6f	sta 6f	Vorzeichenvergleichsbyte (ARG/FAC-Vorzeichen) initialisieren	
,e281	20 53 b8	jsr b853 "sub"	FAC := ARG - FAC	
,e284	a9 ea	lda #ea <(\$e2ea)	LB der Adresse der Konstanten 0.25 laden	} Berechnung von 0.25 - FAC (FAC := 0.25 - FAC)
,e286	a0 e2	ldy #e2 >(\$e2ea)	HB der Adresse der Konstanten 0.25 laden	
,e288	20 50 b8	jsr b850 "memsub"	FAC := Konstante - FAC	
,e28b	a5 66	lda 66	Vorzeichenbyte des FAC auslesen	
,e28d	48	pha	und auf den Stapel legen	
,e28e	10 0d	bpl e29d	positives Vorzeichen (N=0): Sonderbehandlung für negative Zahlen überspringen	
,e290	20 49 b8	jsr b849 "add0.5"	FAC := FAC + 0.5 (FAC um $\frac{1}{2}$ erhöhen)	
,e293	a5 66	lda 66	Vorzeichenbyte des FAC zwecks Test auslesen	
,e295	30 09	bmi e2a0	negativ (N=1): Sonderbehandlung verlassen	
,e297	a5 12	lda 12	Vorzeichenflag für TAN laden	} TAN-Vorzeichenflag auslesen, umdrehen und zurückschreiben
,e299	49 ff	eor #ff %l1l1l1l1	Vorzeichenflag umdrehen	
,e29b	85 12	sta 12	und zurückschreiben	
,e29d	20 b4 bf	jsr bfb4	Vorzeichen des FAC invertieren	
,e2a0	a9 ea	lda #ea <(\$e2ea)	LB der Adresse der Konstanten 0.25 laden	} Berechnung von FAC + 0.25 (FAC := FAC + 0.25)
,e2a2	a0 e2	ldy #e2 >(\$e2ea)	HB der Adresse der Konstanten 0.25 laden	
,e2a4	20 67 b8	jsr b867 "addmem"	Konstante zum FAC addieren	
,e2a7	68	pla	bei \$e28d gemerktes Vorzeichenbyte zwecks Test wieder vom Stapel holen	
,e2a8	10 03	bpl e2ad	positiv (N=0): Vorzeichenwechsel überspringen	
,e2aa	20 b4 bf	jsr bfb4	Vorzeichen des FAC invertieren	
,e2ad	a9 ef	lda #ef <(\$e2ef)	LB der Adresse der Polynomtabelle laden	} Auswertung des Näherungspolynoms
,e2af	a0 e2	ldy #e2 >(\$e2ef)	HB der Adresse der Polynomtabelle laden	
,e2b1	4c 43 e0	jmp e043 "polyx"	Polynom auswerten	

; Routine zur Basic-Funktion TAN (Token: \$c0)

```

,e2b4 20 ca bb jsr bbca "movt3" FAC #1 bis $e2c5-$e2c9 in FAC #3 retten (FAC #3 := FAC #1)
,e2b7 a9 00 lda #00 Initialisierungswert für TAN-Vorzeichenflag laden
,e2b9 85 12 sta 12 TAN-Vorzeichenflag initialisieren
,e2bb 20 6b e2 jsr e26b Routine zur Basic-Funktion SIN aufrufen
,e2be a2 4e ldx #4e <($004e) LB der Adresse eines FAC-Zwischenspeichers laden
,e2c0 a0 00 ldy #00 >($004e) HB der Adresse eines FAC-Zwischenspeichers laden
,e2c2 20 f6 e0 jsr e0f6"(movfm)" indirekt zur MOVFM-Routine springen ("e0f6 jmp movfm")
,e2c5 a9 57 lda #57 <($0057) LB der Adresse des FAC #3 laden
,e2c7 a0 00 ldy #00 >($0057) HB der Adresse des FAC #3 laden
,e2c9 20 a2 bb jsr bba2 "movmf" FAC #3 wieder in FAC #1 kopieren
,e2cc a9 00 lda #00 Initialisierungswert für Vorzeichenbyte laden
,e2ce 85 66 sta 66 Vorzeichenbyte des FAC auf "positiv" stellen
,e2d0 a5 12 lda 12 TAN-Vorzeichenflag (s. $e2b9) auslesen
,e2d2 20 dc e2 jsr e2dc FAC retten und Cosinus des Arguments ebenfalls berechnen
,e2d5 a9 4e lda #4e <($004e) LB der Adresse des FAC-Zwischenspeichers laden
,e2d7 a0 00 ldy #00 >($004e) HB der Adresse des FAC-Zwischenspeichers laden
,e2d9 4c 0f bb jmp bb0f "divmf" FAC := Konstante / FAC

-----

,e2dc 48 pha TAN-Vorzeichenflag (s. $e2d0/$e2d2) auf den Stapel legen
,e2dd 4c 9d e2 jmp e29d in SIN/COS-Routine einsteigen

```

; MFLPT-Konstanten für die SIN/COS-Routine

```

:e2e0 81 49 0f da a2 MFLPT-Darstellung von 1.57079633 =  $\pi/2$ 
:e2e5 83 49 0f da a2 MFLPT-Darstellung von 6.28318531 =  $\pi*2$ 
:e2ea 7f 00 00 00 00 MFLPT-Darstellung von 0.25 = 1/4

```

; Näherungspolynom für die SIN/COS-Routine

```

:e2ef 05 Polynomgrad 5 als Bytewert
:e2f0 84e6 1a 2d 1b MFLPT-Darstellung des Koeffizienten a5 = -14.3813907
:e2f5 86 28 07 fb f8 MFLPT-Darstellung des Koeffizienten a4 = 42.0077971
:e2fa 87 99 68 89 01 MFLPT-Darstellung des Koeffizienten a3 = -76.7041703
:e2ff 87 23 35 df e1 MFLPT-Darstellung des Koeffizienten a2 = 81.6052237
:e304 86 a5 5d e7 28 MFLPT-Darstellung des Koeffizienten a1 = -41.3417021
:e309 83 49 0f da a2 MFLPT-Darstellung des Koeffizienten a0 = 6.28318531 =  $\pi*2$ 

```

; Routine zur Basic-Funktion ATN (Token: \$c1)

```

,e30e a5 66    lda    66      Vorzeichenbyte des FAC auslesen
,e310 48      pha          und auf den Stapel retten
,e311 10 03    bpl    e316    positives Vorzeichen (N=0): keine Invertierung des Vorzeichens
,e313 20 b4 bf  jsr    bfb4    Vorzeichen des FAC umdrehen
,e316 a5 61    >lda    61      Exponentenbyte des FAC holen
,e318 48      pha          und auf den Stapel retten
,e319 c9 81    cmp    #81 %10000001 mit Exponent für Bereich [-1;1] vergleichen
,e31b 90 07    bcc    e324    kleinerer Exponent (C=0): Bildung des reziproken Wertes (Kehrwert) überspringen
,e31d a9 bc    lda    #bc <($b9bc) LB der Adresse der Konstanten 1 laden      } Bildung des reziproken Wertes
,e31f a0 b9    ldy    #b9 >($b9bc) HB der Adresse der Konstanten 1 laden      } (Kehrwertes) durch Berechnung von
,e321 20 0f bb  jsr    bb0f "divmf" FAC := Konstante / FAC                    } FAC := 1 / FAC
,e324 a9 3e    >lda    #3e <($e33e) LB der Adresse der Polynomtabelle laden    } Berechnung
,e326 a0 e3    ldy    #e3 >($e33e) HB der Adresse der Polynomtabelle laden    } des
,e328 20 43 e0  jsr    e043 "polyx" Näherungspolynom berechnen                } Näherungspolynoms
,e32b 68      pla          bei $e318 gemerkten Exponent vom Stapel holen
,e32c c9 81    cmp    #81 %10000001 mit Exponent für Bereich [-1;1] vergleichen
,e32e 90 07    bcc    e337    kleinerer Exponent (C=0): Bildung des reziproken Wertes (Kehrwert) überspringen
,e330 a9 e0    lda    #e0 <($e2e0) LB der Adresse der Konstanten  $\pi/2$  laden    } FAC von  $\pi/2$  abziehen
,e332 a0 e2    ldy    #e2 >($e2e0) HB der Adresse der Konstanten  $\pi/2$  laden    } (Berechnung von
,e334 20 50 b8  jsr    b850 "memsub" FAC := Konstante - FAC                    } FAC :=  $\pi/2$  - FAC)
,e337 68      >pla          bei $e310 gemerktes Vorzeichenbyte vom Stapel holen
,e338 10 03    bpl    e33d    positiv (N=0): Rücksprung über RTS, kein Vorzeichenwechsel
,e33a 4c b4 bf  jmp    bfb4    Vorzeichen des FAC invertieren

-----
,e33d 60      >rts          Rücksprung von Routine
-----

```

; Polynomtabelle für die Routine zur Basic-Funktion ATN

```

,e33e 0b      Polynomgrad 11 als Bytewert
,e33f 76 b3 83 bd d3 MFLPT-Darstellung des Koeffizienten a11 = -6.84793912 e-04
,e344 79 1e f4 a6 f5 MFLPT-Darstellung des Koeffizienten a10 = 4.85094216 e-03
,e349 7b 83 fc b0 10 MFLPT-Darstellung des Koeffizienten a9 = -0.0161117018
,e34e 7c 0c 1f 67 ca MFLPT-Darstellung des Koeffizienten a8 = 0.034209638
,e353 7c de 53 cb cl MFLPT-Darstellung des Koeffizienten a7 = -0.542791328
,e358 7d 14 64 70 4c MFLPT-Darstellung des Koeffizienten a6 = 0.0724571965
,e35d 7d b7 ea 51 7a MFLPT-Darstellung des Koeffizienten a5 = -0.0898023954
,e362 7d 63 30 88 7e MFLPT-Darstellung des Koeffizienten a4 = 0.110932413
,e367 7e 92 44 99 3a MFLPT-Darstellung des Koeffizienten a3 = -0.142839808

```



```

:e36c 7e 4c cc 91 c7      MFLPT-Darstellung des Koeffizienten a2 = 0.19999912
:e371 7f aa aa aa 13      MFLPT-Darstellung des Koeffizienten a1 = -0.333333316
:e376 81 00 00 00 00      MFLPT-Darstellung des Koeffizienten a0 = 1
-----

```

; NMI-Routine für Basic 2.0; an diese Stelle weist der ROM-Vektor \$a002/\$a003

```

,e37b 20 cc ff jsr ffcc "clrchn" Löschen der Filetabelle
,e37e a9 00 lda #00      Flag für "Standard-I/O" laden
,e380 85 13 sta 13      und in INPUT-Kommentar-Flag schreiben
,e382 20 7a a6 jsr a67a  Teil der CLR-Routine aufrufen, um Stringstapelzeiger, CPU-Stapelzeiger, CONT-Flag
                        und FN-Flag zu initialisieren
,e385 58 cli            Interrupts wieder zulassen

```

; Einsprung für Herstellung "READY."-Zustandes;
Nutzung von \$a714 (LIST), \$a854 (STOP/END) und \$e3a0 (Kaltstart)

```

,e386 a2 80 →ldx #80 %100000000 Fehlercode für "kein Fehler" laden
,e388 6c 00 03 jmp(0300) Sprung über Vektor IERROR in Fehlerbehandlungsroutine ab $e38b } Warmstart
                                                } auslösen
-----

```

; Fehlerbehandlungsroutine ERROR

Hierher weist normalerweise der Vektor IERROR \$0300/\$0301, über den bei \$a437 und bei \$e388 gesprungen wird.

```

,e38b 8a txa            Fehlercode aus X-Register zwecks Test in Akku bringen
,e38c 30 03 bmi e391    b7 gesetzt (N=1): Flag für "READY."-Zustand, da kein Fehler vorlag
,e38e 4c 3a a4 jmp a43a  Einstieg in ERROR-Routine, allerdings nicht über Vektor-Sprung wie bei $a437
-----
,e391 4c 74 a4 >jmp a474 "ready" Einsprung in ERROR-Routine, so daß nur die Ausgabe von "READY.", aber keine
                        Fehlermeldung erfolgt
-----

```

; Kaltstart-Routine für Basic 2.0; an diese Stelle weist der ROM-Vektor bei \$a000/\$a001

```

,e394 20 53 e4 jsr e453 "inivec" Initialisierung der Vektoren im Bereich $0300-$030b
,e397 20 bf e3 jsr e3bf "initmp" Initialisierung der RAM-Hilfsspeicher des Basic-Interpreters
,e39a 20 22 e4 jsr e422 "msgnew" Ausgabe der Einschaltmeldung und Ausführung des NEW-Befehls
,e39d a2 fb ldx #fb %11111011 Initialisierungswert des Stapelzeigers laden
,e39f 9a txs            und in den Stapelzeiger schreiben
,e3a0 d0 e4 bne e386 "jmp" Herstellung des "READY."-Zustands
-----

```

; von diesen Speicherplätzen wird die CHRGET-Routine nach \$0073-\$008a in der INITMP-Routine (\$e3bf) kopiert
 Anstelle eines Kommentars, der in höchster Ausführlichkeit bei der Routinenbeschreibung im Fließtext steht,
 wird rechts die CHRGET-Routine an ihrer richtigen Adresse (\$0073) disassembliert dargestellt und knapp kommentiert.

,e3a2 e6 7a	→inc 7a	,0073 e6 7a	→inc 7a	LB des CHRGET-Zeigers erhöhen (Selbstmodifik.)
,e3a4 d0 02	bne e3a8	,0075 d0 02	bne 0079	kein Erhöhungsübertrag (Z=0): nicht HB erhöhen
,e3a6 e6 7b	inc 7b	,0077 e6 7b	inc 7b	HB des CHRGET-Zeigers erhöhen (Selbstmodifik.)
; CHRGET-Einsprung: CHRGET-Zeiger nicht erhöhen, sondern letztes über CHRGET geholtes Zeichen erneut holen und testen				
,e3a8 ad 60	→lda ea60	,0079 ad ..	→lda	Basic-Byte auslesen (CHRGET-Zeiger \$7a/\$7b!)
,e3ab c9 3a	cmp #3a	,007c c9 3a	cmp #3a	Vergleich mit ASCII-Code des Doppelpunktes ":"
,e3ad b0 0a	bcs e3b9	,007e b0 0a	bcs 008a	Byte >= Doppelpunkt (C=1): Endemit C=1; Z ist hier = 1, wenn es ein Doppelpunkt war
,e3af c9 20	cmp #20	,0080 c9 20	cmp #20	Vergleich mit ASCII-Code des Leerzeichens
,e3b1 f0 ef	beq e3a2	,0082 f0 ef	beq 0073	Übereinstimmung (Z=1): überlesen, nächstes Byte
,e3b3 38	sec	,0084 38	sec	Carry vor Subtraktion setzen
,e3b4 e9 30	sbc #30	,0085 e9 30	sbc #30	ASCII-Code von "0" abziehen
,e3b6 38	sec	,0087 38	sec	Carry vor Subtraktion setzen
,e3b7 e9 d0	sbc #d0	,0088 e9 d0	sbc #d0	Subtraktion von \$0085 rückgängig machen, C-Flag entsprechend ASCII-Code setzen (C=0: Ziffer)
,e3b9 60	→rts	,008a 60	→rts	Rücksprung von Routine

; Initialisierungswert für SEED (RND-Ausgangsergebnis); Berücksichtigung bei \$e3bf-\$e421 (INITMP)

:e3ba 80 4f c7 52 58 MFLPT-Darstellung von 0.811635157 (Ausgangswert für RND)

; INITMP-Routine: Initialisierung der RAM-Arbeitsspeicher des Basic-Interpreters

,e3bf a9 4c	lda #4c	Opcode von "jmp" (absolute Adressierung) laden	} JMP-Befehl vor Hilfsvektor und USR-Vektor schreiben
,e3c1 85 54	sta 54	und in Hilfsspeicher für Funktionsausführung schreiben	
,e3c3 8d 10 03	sta 0310	auch vor USR-Vektor schreiben	
,e3c6 a9 48	lda #48 <(illqua)	LB der Adresse des "ILLEGAL QUANTITY"-Einsprungs laden	} USR-Vektor auf ILLEGAL QUANTITY richten
,e3c8 a0 b2	ldy #b2 >(illqua)	HB der Adresse des "ILLEGAL QUANTITY"-Einsprungs laden	
,e3ca 8d 11 03	sta 0311	LB des USR-Vektors setzen	
,e3cd 8c 12 03	sty 0312	HB des USR-Vektors setzen	
,e3d0 a9 91	lda #91 <(\$b391)	LB der Adresse zur Umwandlung eines Wortes ins FLPT-Format laden	} Umwandlungs- vektor "Y/A -> FAC" initialisieren
,e3d2 a0 b3	ldy #b3 >(\$b391)	HB der Adresse zur Umwandlung eines Wortes ins FLPT-Format laden	
,e3d4 85 05	sta 05	LB des Umwandlungsvektors "Wort in Fließkomma" setzen	
,e3d6 84 06	sty 06	HB des Umwandlungsvektors "Wort in Fließkomma" setzen	

,e3d8	a9 aa	lda #aa <(\$blaa)	LB der Adresse zur Umwandlung einer FLPT-Zahl in ein Wort laden	} Umwandlungs-vektor
,e3da	a0 b1	ldy #b1 >(\$blaa)	HB der Adresse zur Umwandlung einer FLPT-Zahl in ein Wort laden	
,e3dc	85 03	sta 03	LB des Umwandlungsvektors "Fließkomma in Wort" setzen	} "FAC -> A/Y" initialisieren
,e3de	84 04	sty 04	HB des Umwandlungsvektors "Fließkomma in Wort" setzen	
,e3e0	a2 1c	ldx #1c	Dekrementierzähler initialisieren (\$1c+1 = #29 Byte zu verschieben)	} Kopieren von CHRGET/CHRGOT und SEED-Wert (\$e3a2-\$e3b9) nach \$73-\$8a
,e3e2	bd a2 e3	→ lda e3a2,x	Byte aus ROM-Tabelle der CHRGET-Routine holen	
,e3e5	95 73	sta 73,x	und in RAM-Bereich der CHRGET-Routine schreiben	
,e3e7	ca	dex	Dekrementierzähler verringern (auf nächstes Byte richten)	
,e3e8	10 f8	bpl e3e2	noch nicht auf \$ff heruntergezählt (N=0): weiter in Kopierschleife	
,e3ea	a9 03	lda #03	Länge eines Stringvariableneintrags laden	
,e3ec	85 53	sta 53	und in Hilfsspeicher für GARCOL schreiben	
,e3ee	a9 00	lda #00	Initialisierungswert/Löschwert laden	
,e3f0	85 68	sta 68	Überlaufbyte des FAC löschen	
,e3f2	85 13	sta 13	INPUT-Kommentar-Flag löschen	
,e3f4	85 18	sta 18	HB der Adresse des letzten Strings auf temporärem Stringstapel löschen	
,e3f6	a2 01	ldx #01	Vorbelegungswert für Linkpointer vor Systemeingabepuffer laden	} Initialisierung des Linkpointer vor dem Systemeingabepuffer
,e3f8	8e fd 01	stx 01fd	HB des Linkpointers vor Systemeingabepuffer vorbelegen	
,e3fb	8e fc 01	stx 01fc	LB des Linkpointers vor Systemeingabepuffer vorbelegen	} temporären Stringstapelzeiger init.
,e3fe	a2 19	ldx #19 *\$19	Zeropage-Anfangsadresse des temporären Stringstapels laden	
,e400	86 16	stx 16	und in Zeiger für temporären Stringstapel schreiben	
,e402	38	sec	Carry als Flag für "MEMBOT-Lesevorgang" setzen	} Untergrenze des für Basic verfügbaren RAM als Basic-Anfang setzen
,e403	20 9c ff	jsr ff9c "membot"	Untergrenze des Basic-RAM nach X/Y laden	
,e406	86 2b	stx 2b	LB als LB der Programmspeicher-Anfangsadresse setzen	} RAM als Basic-Anfang setzen
,e408	84 2c	sty 2c	HB als HB der Programmspeicher-Anfangsadresse setzen	
,e40a	38	sec	Carry als Flag für "MEMTOP-Lesevorgang" setzen	} Obergrenze des für Basic verfügbaren RAM als oberste Basic-Adresse und Stringbereichs-obergrenze setzen
,e40b	20 99 ff	jsr ff99 "memtop"	Obergrenze des Basic-RAM nach X/Y laden	
,e40e	86 37	stx 37	LB als LB der obersten Basic-Adresse setzen	
,e410	84 38	sty 38	HB als HB der obersten Basic-Adresse setzen	
,e412	86 33	stx 33	LB als LB der Obergrenze des Stringbereichs setzen	
,e414	84 34	sty 34	HB als HB der Obergrenze des Stringbereichs setzen	
,e416	a0 00	ldy #00	Initialisierungswert für Offset laden	
,e418	98	tya "lda #00"	auch als Initialisierungswert für erstes Byte im Basic-Programm verwenden	
,e419	91 2b	sta (2b),y	erstes Byte des Basic-Speichers löschen (sozusagen Zeilenendmarkierung einer davor nicht vorhandenen Zeile)	
,e41b	e6 2b	inc 2b	LB des Zeigers auf den Programmspeicher-Anfang erhöhen	} Programmanfangs-zeiger \$2b/\$2c um 1 erhöhen
,e41d	d0 02	bne e421	kein Erhöhungsübertrag (Z=0): HB nicht erhöhen, sofort RTS	
,e41f	e6 2c	inc 2c	HB des Zeigers auf den Programmspeicher-Anfang erhöhen	
,e421	60	→rts	Rücksprung von Routine	

; MSGNEW-Routine: Ausgabe der Einschaltmeldung und Ausführung der NEW-Routine

,e422	a5 2b	lda 2b	LB der Programmspeicher-Anfangsadresse holen	
,e424	a4 2c	ldy 2c	HB der Programmspeicher-Anfangsadresse holen	
,e426	20 08 a4	jsr a408	Prüfroutine auf freien Speicherplatz im Variablenspeicher aufrufen	
,e429	a9 73	lda #73 <(\$e473)	LB der Adresse des Textes "**** COMMODORE 64 BASIC V2 ****[cr]64K RAM SYSTEM" laden	
,e42b	a0 e4	ldy #e4 >(\$e473)	HB der Adresse des Textes "**** COMMODORE 64 BASIC V2 ****[cr]64K RAM SYSTEM" laden	
,e42d	20 1e ab	jsr able "strout"	Text ausgeben	Anzahl
,e430	a5 37	lda 37	LB der obersten Basic-Adresse auslesen	der
,e432	38	sec	Carry vor Subtraktion setzen	freien
,e433	e5 2b	sbc 2b	LB der Programmspeicher-Anfangsadresse subtrahieren	Basic-Bytes
,e435	aa	tax	und Ergebnis als LB des Ergebnisses in X merken	aus
,e436	a5 38	lda 38	HB der obersten Basic-Adresse auslesen	Speichergrenzen
,e438	e5 2c	sbc 2c	HB der Programmspeicher-Anfangsadresse subtrahieren	errechnen und
,e43a	20 cd bd	jsr bdc d "numout"	Ergebnis (X/A) ausgeben	ausgeben
,e43d	a9 60	lda #60 <(\$e460)	LB der Adresse des Textes "BASIC BYTES FREE" laden	Text
,e43f	a0 e4	ldy #e4 >(\$e460)	HB der Adresse des Textes "BASIC BYTES FREE" laden	"BASIC BYTES FREE"
,e441	20 1e ab	jsr able "strout"	Text ausgeben	ausgeben
,e444	4c 44 a6	jmp a644	weiter mit Routine zum NEW-Befehl	

; Initialisierungswerte für die Basic-Vektoren im Bereich \$0300-\$030b

,e447	8b e3	\$e38b	für IERROR \$0300/\$0301	Fehlermeldung erzeugen
,e449	83 a4	\$a483	für IMAIN \$0302/\$0303	Basic-Warmstart auslösen
,e44b	7c a5	\$a57c	für ICRNCH \$0304/\$0305	Tokenisierung des Systemeingabepuffers
,e44d	1a a7	\$a71a	für IQPLOP \$0306/\$0307	Ent-Tokenisierung (LIST) des Akkumulators
,e44f	e4 a7	\$a7e4	für IGONE \$0308/\$0309	Basic-Befehl ausführen
,e451	86 ae	\$ae86	für IEVAL \$030a/\$030b	Auswertung eines Ausdrucks im Basic-Text

; INIVEC-Routine: Initialisierung der Basic-Vektoren im Bereich \$0300-\$030b

,e453	a2 0b	ldx #0b	Dekrementierzähler mit 11 (Anzahl der Bytes - 1) initialisieren	Bereich
,e455	bd 47 e4	lda e447,x	Byte aus ROM-Vorbelegungstabelle holen	\$e447-\$e452
,e458	9d 00 03	sta 0300,x	und in RAM-Vektorenbereich schreiben	nach
,e45b	ca	dex	Dekrementierzähler verringern	\$0300-\$030b
,e45c	10 f7	bpl e455	noch nicht auf \$ff heruntergezählt (N=0): weiter in Schleife	kopieren
,e45e	60	rts	Rücksprung von Routine	

; Füllbyte

:e45f 00 [nul] als Füllbyte ohne Bedeutung

; Texttabelle für STROUT-Ausgabe der Einschaltmeldung in MSGNEW-Routine

:e460 20 42 41 53 49 43 20 42 59 54 45 53 20 46 52 45 45 0d 00 [space]BASIC BYTES FREE[cr,nul]

:e473 93 0d [clr,cr]
 :e475 20 20 20 20 2a 2a 2a 2a 20 43 4f 4d 4d 4f 52 45 20 36 34 [4space]**** COMMODORE 64
 :e48a 20 42 41 53 49 43 20 56 32 20 2a 2a 2a 2a 0d 0d [space]BASIC V2 ****[2cr]
 :e49a 20 36 34 4b 20 52 41 4d 20 53 59 53 54 45 4d 20 20 00 [space]64K RAM SYSTEM[2space,nul]

; Füllbyte hinter Texttabellen

:e4ac 81 Füllbyte ohne Bedeutung; bei manchen C64-Versionen stehen hier andere Werte wie z.B. \$5c, die aber ebenfalls bedeutungslos sind

; BCKOUT-Routine: CKOUT-Behandlung des Basic-Interpreters

,e4ad	48	pha	an Akku übergebene Filenummer des Ausgabekanals auf Stapel legen
,e4ae	20 c9 ff	jsr ffc9 "ckout"	CKOUT-Routine über Kernal-Einsprung ausführen lassen
,e4b1	aa	tax	Akkuinhalt nach CKOUT-Ausführung in X-Register merken (s. \$e4b5)
,e4b2	68	pla	bei \$e4ad gemerkten Übergabeparameter vom Stapel holen
,e4b3	90 01	bcc e4b6	kein I/O-Fehler (C=0): Rücksprung über RTS
,e4b5	8a	txa	ansonsten Fehlernummer (s. \$e4ae/\$e4b1) wieder in Akku holen
,e4b6	60	rts	Rücksprung von Routine

; Füllbytes ohne Bedeutung

:e4b7 aa aa aa aa aa aa aa aa bei allen C64-Versionen
 :e4bf aa aa aa aa aa aa aa aa stehen an diesen Adressen
 :e4c7 aa aa aa aa aa aa aa aa Füllbytes, die keine
 :e4cf aa aa aa aa Bedeutung haben

; diese Routine ist nicht in allen C 64-Versionen enthalten, bei manchen stehen statt dessen \$aa-Füllbytes (!);
sofern vorhanden, wird sie bei \$ef94 aufgerufen

```
,e4d3 85 a9    sta    a9           Akku in RS232-Flag für Startbit-Prüfung schreiben
,e4d5 a9 01    lda    #01 %00000001 Belegung für RS232-Eingabeparität laden
,e4d7 85 ab    sta    ab           und in RS232-Eingabeparität bzw. Kassettenzähler schreiben
,e4d9 60       rts                Rücksprung von Routine
```

; diese Routine ist in den allerältesten C 64-Versionen nicht vorhanden (bei diesen stehen hier Füllbytes);
sofern vorhanden, wird sie bei \$ea07 aufgerufen

```
,e4da ad 86 02  lda    0286       aktuelle Zeichenfarbe holen
,e4dd 91 f3     sta    (f3),y      und an aktuelle Position im Farb-RAM schreiben
,e4df 60       rts                Rücksprung von Routine
```

; WATCBM-Hilfsroutine:

Warten auf Drücken der Commodore-Taste

Im Akku befindet sich der Inhalt von \$a1 (mittelwertiges Byte der Systemuhr)

```
,e4e0 69 02    adc    #02 %00000010 Zähler im Akku um 2 erhöhen
,e4e2 a4 91    >ldy    91          Flag für Commodore-Taste auslesen ($ff = Commodore-Taste gedrückt)
,e4e4 c8       iny              Flag erhöhen ($ff wird zu $00, wobei Z=1 wird; ansonsten Z=0)
,e4e5 d0 04    <| bne    e4eb       Commodore-Taste nicht gedrückt (Z=0): Rücksprung von Routine; Warten wird
                                     fortgesetzt
,e4e7 c5 a1    <| cmp     a1        mittelwertiges Byte der Systemuhr auslesen
,e4e9 d0 f7    <| bne    e4e2       Wartezeit noch nicht abgelaufen (Z=0): weiter in Warteschleife
,e4eb 60       >rts              Rücksprung von Routine, da Commodore-Taste gedrückt wurde oder Wartezeit ablief
```

; Baud-Raten für RS232 bei PAL-Version:

Da es sich um Timerkonstanten für CIA-Register handelt, stehen niedrigere Werte für geringere Verzögerungen und demzufolge höhere Baud-Raten.

```
:e4ec 19 26 44 19 1a 11 e8 0d    $2619 (50 Baud); $1944 (75 Baud); $111a (110 Baud); $0de8 (134.5 Baud)
:e4f4 70 0c 06 06 d1 02 37 01    $0c70 (150 Baud); $0606 (300 Baud); $02d1 (600 Baud); $0137 (1200 Baud)
:e4fc ae 00 69 00                $00ae (1800 Baud); $0069 (2400 Baud)
```

; IOBASE-Routine (wird von der Kernal-Sprungtabelle bei \$fff3 aufgerufen)

,e500	a2 00	ldx #00 <(\$dc00)	LB der CIA-Basisadresse in LB von X/Y laden	} CIA-Basisadresse \$dc00 in X/Y laden
,e502	a0 dc	ldy #dc >(\$dc00)	HB der CIA-Basisadresse in HB von X/Y laden	
,e504	60	rts	Rücksprung von Routine	

; SCREEN-Routine (wird von der Kernal-Sprungtabelle bei \$ffed aufgerufen)

,e505	a2 28	ldx #28	40 als Anzahl der Spalten am Bildschirm laden	} Bildschirmformat 40x25 in X/Y laden
,e507	a0 19	ldy #19	25 als Anzahl der Zeilen am Bildschirm laden	
,e509	60	rts	Rücksprung von Routine	

; PLOT-Routine (wird von der Kernal-Sprungtabelle bei \$fff0 aufgerufen)

,e50a	b0 07	[bcs e513 stx d6 sty d3 jsr e56c "stup"	Carry-Flag gesetzt (C=1): Cursorposition nach X/Y auslesen
,e50c	86 d6		Cursor auf Zeile X (\$00-\$27) richten
,e50e	84 d3		Cursor auf Spalte Y (\$00-\$18) richten
,e510	20 6c e5		Zeiger auf Bildschirm- und Farb-RAM gemäß \$d3/\$d6 aktualisieren
,e513	a6 d6	→ ldx d6	Cursorzeile (\$00-\$27) in X-Register holen
,e515	a4 d3	ldy d3	Cursorspalte (\$00-\$18) in Y-Register holen
,e517	60	rts	Rücksprung von Routine

; INTSCR-Routine: Bildschirm initialisieren

,e518	20 a0 e5	jsr e5a0 "intvic"	VIC-Register mit Vorbelegungswerten initialisieren	} Zeichensatz-Umschaltung über <SHIFT>+<CBM> zulassen
,e51b	a9 00	lda #00	Flag für "Shift/Commodore-Taste bewirkt Umschaltung" laden	
,e51d	8d 91 02	sta 0291	in Flag für Zeichensatz-Umschaltung schreiben	
,e520	85 cf	sta cf	Cursorblinkphasenflag löschen	
,e522	a9 48	lda #48 <(\$eb48)	LB der Adresse der Tastaturdekodierungsroutine laden	} KEYLOG-Vektor (zeigt auf Routine zur Tastaturdekodierung) initialisieren
,e524	8d 8f 02	sta 028f	und in LB des Vektors KEYLOG schreiben	
,e527	a9 eb	lda #eb >(\$eb48)	HB der Adresse der Tastaturdekodierungsroutine laden	
,e529	8d 90 02	sta 0290	und in HB des Vektors KEYLOG schreiben	
,e52c	a9 0a	lda #0a	Größe des Tastaturpuffers und Ausgangswert für Repeat-Zähler laden	
,e52e	8d 89 02	sta 0289	XMAX (maximale Größe des Tastaturpuffers) initialisieren	
,e531	8d 8c 02	sta 028c	DELAY (Zähler für Tastatur-Repeat-Verzögerung) initialisieren	
,e534	a9 0e	lda #0e	Farbcode für "hellblau" laden	
,e536	8d 86 02	sta 0286	und als COLOR (aktuelle Zeichenfarbe) setzen	
,e539	a9 04	lda #04	Initialisierungswert für Zählgeschwindigkeit bei Tastatur-Repeat laden	

.e53b	8d 8b 02	sta 028b	und als KOUNT (Zählgeschwindigkeit bei Tastatur-Repeat) setzen
.e53e	a9 0c	lda #0c	Initialisierungswert für Blinkzeitzähler und Cursormodus-Flag laden
.e540	85 cd	sta cd	BLNCT (Zähler für blinkenden Cursor) initialisieren
.e542	85 cc	sta cc	BLNSW (Flag für Cursor an/aus) setzen (\$0c <> 0; Bedeutung: Cursor aus)

Im Speicher folgt die Routine für das Steuerzeichen \$93 (CLR; Bildschirm löschen)

; CLEAR-Routine: Bildschirm löschen (Steuerzeichen \$93 = #147)

.e544	ad 88 02	lda 0288	HIBASE (HB der Bildschirmspeicher-Anfangsadresse) auslesen
.e547	09 80	ora #80 %10000000	b7 im Akku setzen (Flag für "keine Fortsetzung einer logischen Zeile")
.e549	a8	tay	und Ergebnis als Füllwert für LDTB1 (Bildschirmzeilen-Verknüpfungstabelle) verwenden
.e54a	a9 00	lda #00	Bildschirmspaltenzähler mit 0 initialisieren
.e54c	aa	tax "ldx #00"	Offset in LDTB1 (Bildschirmzeilen-Verknüpfungstabelle) mit 0 initialisieren
.e54d	94 d9	→sty d9,x	Füllwert in LDTB1 (Bildschirmzeilen-Verknüpfungstabelle) schreiben
.e54f	18	clc	Carry vor Addition löschen
.e550	69 28	adc #28	40 (Anzahl der Bildschirmspalten pro Zeile) addieren (auf nächste Zeile schalten)
.e552	90 01	bcc e555	kein Additionsübertrag (C=0): Füllwert nicht erhöhen
.e554	c8	iny	Füllwert für Bildschirmzeilen-Verknüpfungstabelle erhöhen
.e555	e8	→inx	Offset in LDTB1 (Bildschirmzeilen-Verknüpfungstabelle) erhöhen
.e556	e0 1a	cpx #1a	Offset schon in 26. Zeile (also außerhalb des zulässigen Bereichs)?
.e558	d0 f3	↖bne e54d	nein (Z=0): weiter in LDTB1-Initialisierungsschleife
.e55a	a9 ff	lda #ff %11111111	letztes Byte für LDTB1 (Bildschirmzeilen-Verknüpfungstabelle) laden
.e55c	95 d9	sta d9,x	und an letzte Position (Adresse \$f3 = \$d9+\$1a) schreiben
.e55e	a2 18	ldx #18	Nummer der untersten Bildschirmzeile laden
.e560	20 ff e9	→jsr e9ff "dellin"	Bildschirmzeile löschen
.e563	ca	dex	Nummer der zu löschenden Bildschirmzeile verringern
.e564	10 fa	↖bpl e560	noch nicht auf \$ff heruntergezählt (N=0): nächste Zeile (Nummer in X) löschen

; HOME-Routine: Cursor in Home-Position (Zeile 0/Spalte 0) bringen (Steuerzeichen \$13 = #19)

.e566	a0 00	ldy #00	Zeilen- und Spaltenwert der Home-Position laden
.e568	84 d3	sty d3	Nummer der aktuellen Cursorspalte auf 0 setzen
.e56a	84 d6	sty d6	Nummer der aktuellen Cursorzeile auf 0 setzen

; STUPT-Routine: Zeiger für Bildschirmspeicher und Farb-RAM gemäß \$d3/\$d6 aktualisieren

.e56c	a6 d6	ldx d6	Nummer der aktuellen Cursorzeile in X-Register holen
.e56e	a5 d3	lda d3	Nummer der aktuellen Cursorspalte in Akku holen
.e570	b4 d9	→ldy d9,x	HB der Adresse der Bildschirmzeile (enthält Flag für logische/echte Zeile) holen
.e572	30 08	↖bmi e57c	keine Fortsetzung von logischer Zeile (N=1): Suchschleife verlassen


```

,e574 18      |   clc          Carry vor Addition löschen
,e575 69 28    |   adc #28      40 (Anzahl der Bildschirmspalten pro Zeile) addieren (auf nächste Zeile schalten)
,e577 85 d3    |   sta  d3      Ergebnis als Cursorspalte innerhalb logischer Zeile setzen
,e579 ca      |   dex         Nummer der aktuellen Cursorzeile verringern
,e57a 10 f4    |   bpl e570     noch nicht auf $ff heruntergezählt (N=0): weiter in Suchschleife
,e57c 20 f0 e9->jsr e9f0 "linadr" Adresse der Bildschirmzeile berechnen (Nummer in X enthalten)
,e57f a9 27    |   lda #27      39 (höchste Spaltennummer in echter Zeile) laden
,e581 e8      |   inx         Offset in LDTBl (Bildschirmzeilen-Verknüpfungstabelle) erhöhen
,e582 b4 d9    |   ldy  d9,x    HB der Adresse der Zeile holen
,e584 30 06    |   bmi e58c     keine Fortsetzung von logischer Zeile (N=1): Suchschleife verlassen
,e586 18      |   clc          Carry vor Addition löschen
,e587 69 28    |   adc #28      40 (Anzahl der Bildschirmspalten pro Zeile) addieren (auf nächste Zeile schalten)
,e589 e8      |   inx         Offset in LDTBl (Bildschirmzeilen-Verknüpfungstabelle) erhöhen
,e58a 10 f6    |   bpl e582     noch nicht auf $ff heruntergezählt (N=0): weiter in Suchschleife
,e58c 85 d5    |   sta  d5      Additionsergebnis als Länge der aktuellen logischen Zeile setzen
,e58e 4c 24 ea |   jmp ea24 "colptr" Zeiger auf aktuelle Position im Farb-RAM aktualisieren
-----

```

; Unterroutine (von \$e621 aus aufgerufen)

Im X-Register wird die Nummer der aktuellen Cursorzeile erwartet.

```

,e591 e4 c9    |   cpx  c9      Vergleich mit Cursorzeile bei INPUT
,e593 f0 03    |   beq e598     Übereinstimmung (Z=1): Rücksprung über RTS
,e595 4c ed e6 |   jmp e6ed     sonst Neuberechnung der Hilfszeiger
-----
,e598 60      |   rts         Rücksprung von Routine
-----

```

; Füllbefehl, um Einsprungsadresse nach Betriebssystem-Modifikationen beizubehalten

```

,e599 ea      |   nop         keine Wirkung

```

; Einsprung: Ausführung von INTVIC und HOME

```

,e59a 20 a0 e5 |   jsr e5a0 "intvic" VIC-Register initialisieren
,e59d 4c 66 e5 |   jmp e566 "home"  Cursor in Home-Position bewegen
-----

```

; INTVIC-Routine:

VIC-Register initialisieren

,e5a0	a9 03	lda #03	Geräteadresse des Bildschirms laden	} Standard-I/O-Geräte (Tastatur/Bildschirm) für Betriebssystem setzen
,e5a2	85 9a	sta 9a	und als Nummer des aktuellen Ausgabegerätes setzen	
,e5a4	a9 00	lda #00	Geräteadresse der Tastatur laden	
,e5a6	85 99	sta 99	und als Nummer des aktuellen Eingabegerätes setzen	
,e5a8	a2 2f	ldx #2f	Anzahl der Register (47) als Dekrementierzähler laden	
,e5aa	bd b8 ec	→ lda ecb8,x	Initialisierungswert aus ROM-Tabelle \$ecb8-\$ece7 holen	
,e5ad	9d ff cf	sta cfff,x	und in VIC-Register (Bereich: \$d000—\$d02e) schreiben	
,e5b0	ca	dex	Dekrementierzähler verringern	
,e5b1	d0 f7	bne e5aa	noch nicht auf \$00 heruntergezählt (Z=0): weiter in Initialisierungsschleife	
,e5b3	60	rts	Rücksprung von Routine	

; NXTKEY-Routine:

nächstes Zeichen aus Tastaturpuffer in Akku holen

,e5b4	ac 77 02	ldy 0277	vorderstes Zeichen des Tastaturpuffers ins Y-Register holen (wird bei \$e5c6 im Akku an die aufrufende Routine zurückgegeben)	
,e5b7	a2 00	ldx #00	Offset für Verschiebeschleife initialisieren	} Bytes 2-x im Tastaturpuffer um 1 Byte (1 Tastencode) vorrücken lassen
,e5b9	bd 78 02	→ lda 0278,x	n+1.Byte aus Tastaturpuffer holen	
,e5bc	9d 77 02	sta 0277,x	und als n. Byte schreiben	
,e5bf	e8	inx	Offset erhöhen	
,e5c0	e4 c6	cpx c6	Vergleich mit NDX (Anzahl der Zeichen im Tastaturpuffer)	
,e5c2	d0 f5	bne e5b9	noch keine Übereinstimmung (Z=0): weiter in Verschiebeschleife	
,e5c4	c6 c6	dec c6	NDX (Anzahl der Zeichen im Tastaturpuffer) verringern, da 1 Byte ausgelesen wurde	
,e5c6	98	tya	bei \$e5b4 ausgelesenes 1. Zeichen des Tastaturpuffers in Akku bringen	
,e5c7	58	cli	Interrupt wieder zulassen	
,e5c8	18	clc	Carry löschen (wird von aufrufender Routine erwartet)	
,e5c9	60	rts	Rücksprung von Routine	

; Tastatur-Eingabeschleife

,e5ca	20 16-e7	→ jsr e716 "scrout"	Zeichen im Akku auf Bildschirm ausgeben
,e5cd	a5 c6	→ lda c6	NDX (Anzahl der Zeichen im Tastaturpuffer) holen
,e5cf	85 cc	sta cc	und BLNSW setzen (auf "Cursor aus" stellen), da Akku hier normalerweise <> 0 ist
,e5d1	8d 92 02	sta 0292	AUTODN (Flag für automatisches Scrolling) setzen, da Akku hier normalerweise <> 0 ist
,e5d4	f0 f7	beq e5cd	kein Zeichen im Tastaturpuffer (Z=1): warten, bis NDX<>0
,e5d6	78	sei	Interrupt verhindern, damit im Interrupt ablaufende Tastaturabfrage nicht stört
,e5d7	a5 cf	lda cf	BLNON auslesen (Flag für "Cursor in Blinkphase")
,e5d9	f0 0c	beq e5e7	Zeichen an Cursorposition z.Zt. revers (Z=1): Sonderbehandlung überspringen

```

,e5db a5 ce      lda  ce      GDBLN (Zeichen an Cursorposition) auslesen
,e5dd ae 87 02    ldx  0287    GDCOL (Farbe an Cursorposition) holen
,e5e0 a0 00      ldy  #00      Löschwert laden
,e5e2 84 cf      sty  cf      und in BLNON (Flag für "Cursor in Blinkphase") schreiben } Flag für "Zeichen
,e5e4 20 13 ea    jsr  eal3 "setchc" Zeichen und Farbe in Bildschirmspeicher und Farb-RAM übernehmen } revers" setzen
,e5e7 20 b4 e5    jsr  e5b4 "nxtkey" nächstes Zeichen aus Tastaturpuffer in Akku holen
,e5ea c9 83      cmp  #83 %10000011 Vergleich mit ASCII-Code von <SHIFT>+<RUN/STOP>
,e5ec d0 10      bne  e5fe      keine Übereinstimmung (Z=1): Überspringen der <SHIFT>+<RUN/STOP>-Sonderbehandlung

```

; Sonderbehandlung für <SHIFT>+<RUN/STOP>:

Simulation der Eingabe von LOAD[CR]RUN[CR]

```

,e5ee a2 09      ldx  #09      Dekrementierzähler (Anzahl der zu schreibenden Tasten-ASCII-Codes) laden
,e5f0 78         sei          Interrupt verhindern, damit im Interrupt ablaufende Tastaturabfrage nicht stört
,e5f1 86 c6      stx   c6      NDX (Anzahl der Zeichen im Tastaturpuffer) mit Anzahl der ASCII-Codes belegen
,e5f3 bd e6 ec    lda  ece6,x   ASCII-Code aus ROM-Tabelle von LOAD[CR]RUN[CR] holen
,e5f6 9d 76 02    sta  0276,x   und in Tastaturpuffer schreiben
,e5f9 ca         dex          Dekrementierzähler verringern
,e5fa d0 f7      bne  e5f3      noch nicht auf 0 heruntergezählt (Z=0): Fortsetzung der Kopierschleife
,e5fc f0 cf      beq  e5cd "jmp" zurück zum Anfang der Tastatur-Eingabeschleife

```

```

,e5fe c9 0d      cmp  #0d      Vergleich mit ASCII-Code von <RETURN>
,e600 d0 c8      bne  e5ca      keine Übereinstimmung (Z=0): zurück zum Anfang der Tastatur-Eingabeschleife

```

; Eingabe-Ende über <RETURN>: zunächst alle Leerzeichen am Zeilenende löschen

```

,e602 a4 d5      ldy  d5      LNMx (Länge der aktuellen logischen Bildschirmzeile) als Offset holen
,e604 84 d0      sty  d0      und gleichzeitig als Eingabe-Flag (INPUT/GET) setzen
,e606 b1 d1      lda  (d1),y    Zeichen aus derzeitiger Bildschirmzeile holen
,e608 c9 20      cmp  #20      Vergleich mit ASCII-Code für Leerzeichen
,e60a d0 03      bne  e60f      keine Übereinstimmung (Z=0): Schleife verlassen, da alle Schluß-Leerzeichen entfernt
,e60c 88         dey          Offset verringern (= Leerzeichen ignorieren)
,e60d d0 f7      bne  e606      noch nicht am Zeilenanfang (Z=0): weiter auf anderes Zeichen als Leerzeichen suchen
,e60f c8         iny          Offset zum letzten Zeichen := aktueller Offset + 1
,e610 84 c8      sty  c8      erhöhten Offset als INDx (Zeiger auf Ende der logischen Eingabezeile) setzen
,e612 a0 00      ldy  #00      Löschwert für AUTODN (Flag für automatisches Abwärts-Scrolling) laden
,e614 8c 92 02    sty  0292    und AUTODN (Flag für automatisches Abwärts-Scrolling) damit belegen
,e617 84 d3      sty  d3      aktuelle Cursorspalte mit 0 initialisieren
,e619 84 d4      sty  d4      QTSW (Quote-Mode-Flag für Bildschirmditor) löschen
,e61b a5 c9      lda  c9      Cursorspalte in aktueller Eingabezeile holen

```

,e61d	30 1b	bmi e63a	logische Zeile durch Abwärts-Scrolling entfernt (N=1): Zeichen aus
,e61f	a6 d6	ldx d6	Bildschirmspeicher holen, in ASCII-Code konvertieren, auswerten und Rücksprung
,e621	20 91 e5	jsr e591	Nummer der aktuellen Cursorzeile (TBLX) laden
,e624	e4 c9	cpx c9	Unterroutine bei \$e591 aufrufen, um Hilfszeiger auf diese Zeile zu richten
,e626	d0 12	bne e63a	Vergleich mit Cursorspalte innerhalb logischer Eingabezeile
,e628	a5 ca	lda ca	keine Übereinstimmung (Z=0): Zeichen aus Bildschirmspeicher holen und verarbeiten
,e62a	85 d3	sta d3	Cursorspalte für Eingabe holen
,e62c	c5 c8	cmp c8	und als Cursorspalte in aktueller Zeile setzen
,e62e	90 0a	bcc e63a	Vergleich mit INDX (Zeiger auf Ende der logischen Eingabezeile)
,e630	b0 2b	bcs e65d "jmp"	Eingabe-Cursorspalte < INDX (C=0): Zeichen aus Bildschirmspeicher holen/verarbeiten
			ansonsten Eingabe-Ende ohne Auslesen des Zeichens

; SCRGET-Routine: Zeichen von aktueller Bildschirmposition in Akku holen

,e632	98	tya	Y-Register in Akku bringen	} X-Register und
,e633	48	pha	und von dort auf den Stapel retten	
,e634	8a	txa	X-Register in Akku bringen	} Y-Register retten
,e635	48	pha	und von dort auf den Stapel retten	
,e636	a5 d0	lda d0	CRSW (Flag für INPUT oder GET von Tastatur) zwecks Test auslesen	
,e638	f0 93	beq e5cd	kein Aufruf wegen <RETURN> am Eingabe-Ende (Z=1): zurück in Eingabe-Warteschleife	

; Einsprung: Zeichen von aktueller Bildschirmposition in Akku holen und verarbeiten

,e63a	a4 d3	ldy d3	PNTR (Cursorspalte in aktueller Zeile) als Offset holen
,e63c	b1 d1	lda (d1),y	aktuelles Byte aus Bildschirmspeicher holen
,e63e	85 d7	sta d7	und in Hilfsspeicher \$d7 (dient hier als Speicher für aktuelles Zeichen) schreiben

; Zeichen in ASCII-Code umwandeln

,e640	29 3f	and #3f %00111111	b6 und b7 löschen
,e642	06 d7	asl d7	aktuellen Zeichencode verdoppeln
,e644	24 d7	bit d7	und testen
,e646	10 02	bpl e64a	b7=0, also war vorher b6=0 (N=0): b7 nicht setzen
,e648	09 80	ora #80 %10000000	b7 wieder setzen
,e64a	90 04	bcc e650	b7 im aktuellen Zeichencode war vor \$e642 gelöscht (C=0): kein Test auf Quote Mode
,e64c	a6 d4	ldx d4	QTSW zwecks Test auslesen (Flag für Quote Mode des Bildschirmeditors)
,e64e	d0 04	bne e654	Editor nicht im Quote Mode (Z=0): b6 nicht setzen
,e650	70 02	bvs e654	b6 war bei \$e644 gesetzt, also b5=1 vor \$e642 (V=1): b6 nicht setzen
,e652	09 40	ora #40 %01000000	b6 wieder setzen

; hier ist die Bildschirmcode-ASCII-Umwandlung abgeschlossen

```

,e654 e6 d3  →inc d3      PNTR (Cursorspalte in aktueller Zeile) erhöhen
,e656 20 84 e6 jsr e684 "chgqut" eventuelles Ändern des Quote-Mode-Flags QTSW ($d4)
,e659 c4 c8    cpy c8      Vergleich der Cursorspalte (s. $e63a) mit Nummer der letzten Spalte in logischer
                           Eingabezeile

,e65b d0 17    bne e674     keine Übereinstimmung (Z=0): Schlußbehandlung auslösen (Register vom Stapel usw.)
,e65d a9 00    lda #00      Löschwert für CRSW (Flag für INPUT oder GET über Tastatur) laden
,e65f 85 d0    sta d0       und CRSW (Flag für INPUT oder GET über Tastatur) damit belegen
,e661 a9 0d    lda #0d      ASCII-Code von <RETURN> laden
,e663 a6 99    ldx 99       DFLTn (Eingabegerät) zwecks Test auslesen
,e665 e0 03    cpx #03      Bildschirm?
,e667 f0 06    beq e66f     ja (Z=1): Zeichen auf Bildschirm ausgeben
,e669 a6 9a    ldx 9a       DFLT0 (Ausgabegerät) zwecks Test auslesen
,e66b e0 03    cpx #03      Bildschirm?
,e66d f0 03    beq e672     ja (Z=1): Bildschirmausgabe des aktuellen Zeichens überspringen
,e66f 20 16 e7 →jsr e716 "scrout" in ASCII-Code umgewandeltes Zeichen auf den Bildschirm ausgeben
,e672 a9 0d    →lda #0d      ASCII-Code von <RETURN> laden (falls $e66l bereits rückgängig gemacht wurde)
,e674 85 d7    →sta d7       und in Hilfsspeicher für aktuelles Zeichen schreiben
,e676 68      pla          bei $e634/$e635 gemerkten Wert holen      } X-Register
,e677 aa      tax          und wieder ins X-Register bringen      }
,e678 68      pla          bei $e632/$e633 gemerkten Wert holen      } Y-Register
,e679 a8      tay          und wieder ins Y-Register bringen      } wiederherstellen
,e67a a5 d7    lda d7       aktuellen ASCII-Code (s. $e672) auslesen
,e67c c9 de    cmp #de      Vergleich mit ASCII-Code für π
,e67e d0 02    bne e682     keine Übereinstimmung (Z=0): nicht π-Code laden
,e680 a9 ff    →lda #ff     für Basic üblichen ASCII-Code von π laden
,e682 18      →clc          Carry löschen (wird von aufrufenden Routinen erwartet)
,e683 60      rts          Rücksprung von Routine

```

; CHGQUT-Routine:

Quote-Mode-Flag invertieren, wenn Anführungszeichen-Code im Akku steht

```

,e684 c9 22    cmp #22      Vergleich des Akku mit dem ASCII-Code des Anführungszeichens
,e686 d0 08    bne e690     keine Übereinstimmung (Z=0): Rücksprung von Routine, da kein Anführungszeichen
,e688 a5 d4    →lda d4       QTSW (Quote-Mode-Flag des Editors) holen
,e68a 49 01    eor #01 %00000001 b0 invertieren (Quote-Mode-Flag invertieren)
,e68c 85 d4    sta d4       und in QTSW (Quote-Mode-Flag des Editors) zurückschreiben
,e68e a9 22    lda #22      wieder den ASCII-Code des Anführungszeichens laden, da Akku bei $e688 verändert wurde
,e690 60      →rts          Rücksprung von Routine

```

; CHRRAM-Routine: Zeichen ohne weitere Vorkehrungen in Bildschirmspeicher schreiben;
Steuerzeichen wurden vorher nicht bearbeitet.

```
,e691 09 40    ora #40 %010000000 b6 setzen
,e693 a6 c7    ldx  c7          RVS (Flag für reverse Zeichendarstellung) zwecks Test auslesen
,e695 f0 02    beq  e699        keine Reversdarstellung (Z=1): b7 (Revers-Bit im Bildschirmcode) nicht setzen
,e697 09 80    ora #80 %100000000 b7 setzen
,e699 a6 d8    >ldx  d8          INSRT (Anzahl der noch zu tätigenen <INST>-Einfügungen) auslesen
,e69b f0 02    beq  e69f        keine <INST>-Einfügungen mehr (Z=1): INSRT nicht dekrementieren
,e69d c6 d8    dec  d8          INSRT (Anzahl der noch zu tätigenen <INST>-Einfügungen) verringern
,e69f ae 86 02 >ldx 0286        COLOR (aktuelle Zeichenfarbe) auslesen
,e6a2 20 13 ea jsr  eal3 "setchc" Zeichen und Farbe in Bildschirmspeicher und Farb-RAM übernehmen
,e6a5 20 b6 e6 jsr  e6b6 "upldt" LDTB1 (Tabelle für HBs der Anfangsadressen der logischen Zeilen) aktualisieren
```

; wichtiger Einsprung: Endbehandlung der Bildschirmausgabe;
druckende Zeichen wurden hier bereits geschrieben bzw. als Steuerzeichen ausgeführt

```
,e6a8 68      pla              bei $e71b/$e71c gemerkten X-Inhalt vom Stapel holen
,e6a9 a8      tay              und ins X-Register bringen
,e6aa a5 d8    lda  d8          INSRT (Anzahl der noch zu tätigenen <INST>-Einfügungen) auslesen
,e6ac f0 02    beq  e6b0        keine <INST>-Einfügungen mehr (Z=1): X-Register und Akku holen, Ende
,e6ae 46 d4    lsr  d4          QTSW rechtsverschieben, dadurch Quote-Mode-Flag löschen
,e6b0 68      >pla              bei $e719/$e71a gemerkten X-Inhalt vom Stapel holen } X-Register
,e6b1 aa      tax              und ins X-Register bringen } und Akku
,e6b2 68      pla              bei $e716 gemerkten Akku-Inhalt vom Stapel holen } wiederherstellen
,e6b3 18      clc              Carry löschen (Flag für "kein I/O-Fehler")
,e6b4 58      cli              Interrupt wieder zulassen
,e6b5 60      rts              Rücksprung von Routine
```

; UPDTL-Routine: LDTB1 (Tabelle für HBs der Anfangsadressen der logischen Zeilen) aktualisieren

```
,e6b6 20 b3 e8 jsr  e8b3 "movrgh" Hilfsroutine für <CRSR RIGHT>-Bewegungaufrufen
,e6b9 e6 d3    inc  d3          Zeiger auf Cursorspalte in aktueller Zeile um 1 erhöhen
,e6bb a5 d5    lda  d5          LNMx (physikalische Bildschirmzeilenlänge) holen
,e6bd c5 d3    cmp  d3          mit neuem (s. $e6b9) Wert der Cursorspalte vergleichen
,e6bf b0 3f    bcs  e700        LNMx >= Cursorspalte (C=1): RTS anspringen
,e6c1 c9 4f    cmp  #4f         Vergleich der physikalischen Bildschirmzeilenlänge (s. $e6bb) mit 79 (Länge einer
                                logischen Zeile, die sich über 2 echte Bildschirmzeilen erstreckt)
,e6c3 f0 32    beq  e6f7        Übereinstimmung (Z=1): Sprung in nächste logische Zeile
,e6c5 ad 92 02 lda  0292        AUTODN (Flag für automatisches Scrolling) auslesen
```

```

,e6c8 f0 03  beq e6cd      automatisches Scrolling eingeschaltet (Z=1): LDTBl aktualisieren
,e6ca 4c 67 e9  jmp e967      in Routine zum Einfügen einer Leerzeile am unteren Bildschirmrand und
                                Aufwärts-Scrolling um 1 Zeile einspringen
-----
,e6cd a6 d6  >ldx d6      Nummer der aktuellen Cursorzeile (0-24) auslesen
,e6cf e0 19    cpx #19      Vergleich mit 25 (nur kleinere Werte sind zulässig)
,e6d1 90 07    bcc e6da      Cursorzeile < 25 (C=0): kein Zurückstellen auf Zeile 24
,e6d3 20 ea e8  jsr e8ea "scroll" Bildschirm-Scrolling um 1 Zeile nach oben
,e6d6 c6 d6    dec d6      Nummer der Cursorzeile verringern (von 25 auf 24 stellen)
,e6d8 a6 d6    ldx d6      Nummer der aktuellen Cursorzeile (0-24) auslesen
,e6da 16 d9  >asl d9,x      entsprechenden Eintrag in LDTBl linksverschieben, um b7 zu löschen
,e6dc 56 d9    lsr d9,x      Linksverschiebung wieder rückgängig machen, wobei b7 gelöscht bleibt
,e6de e8      inx          Offset in LDTBl auf Eintrag für nächste Bildschirmzeile richten
,e6df b5 d9    lda d9,x      LDTBl-Eintrag für folgende Zeile auslesen
,e6e1 09 80    ora #80 %100000000 b7 setzen (Flag für "Anfang einer logischen Zeile")
,e6e3 95 d9    sta d9,x      und in LDTBl-Eintrag zurückschreiben
,e6e5 ca      dex          Offset wieder auf vorherigen Eintrag stellen ($e6de rückgängig machen)
,e6e6 a5 d5    lda d5      LNMx (physikalische Bildschirmzeilenlänge) auslesen
,e6e8 18      clc          Carry vor Addition löschen
,e6e9 69 28    adc #28      Anzahl der Zeichen pro Zeile (40) addieren
,e6eb 85 d5    sta d5      und Wert als LNMx zurückschreiben
,e6ed b5 d9  >lda d9,x      LDTBl-Eintrag der aktuellen Zeile auslesen
,e6ef 30 03    bmi e6f4      Anfang einer logischen Zeile (N=1): Farb-RAM-Zeiger setzen und Rücksprung
,e6f1 ca      dex          Offset auf vorhergehende Zeile stellen
,e6f2 d0 f9    bne e6ed      noch nicht auf 0 heruntergezählt (Z=0): Fortsetzung der Suchschleife
,e6f4 4c f0 e9  >jmp e9f0 "linadr" Zeiger $d1/$d2 (PNT) auf aktuelle Bildschirmspeicheradresse stellen
-----
; Sprung in nächste logische Zeile

,e6f7 c6 d6  >dec d6      Nummer der Cursorzeile verringern
,e6f9 20 7c e8  jsr e87c "movcrs" Hilfsroutine für Cursorpositionierungen aufrufen
,e6fc a9 00    lda #00      0 (linker Rand einer Bildschirmzeile) laden
,e6fe 85 d3    sta d3      und als aktuelle Cursorspalte setzen
,e700 60      rts          Rücksprung von Routine
-----
; Cursor von linkem Zeilenrand (Spalte 0) an rechten Zeilenrand (Spalte 39) der vorhergehenden Zeile bewegen

,e701 a6 d6    ldx d6      Nummer der aktuellen Cursorzeile (0-24) auslesen
,e703 d0 06    bne e70b      nicht oberste Bildschirmzeile (Z=0): Sonderbehandlung überspringen

```

,e705	86 d3	stx d3	im X-Register ist wegen \$e701/\$e703 der Wert 0 enthalten, der hier als Nummer der aktuellen Cursorspalte gesetzt wird
,e707	68	pla	LB der Rücksprungadresse vom Stapel entfernen
,e708	68	pla	HB der Rücksprungadresse vom Stapel entfernen
,e709	d0 9d	bne e6a8 "jmp"	HB der Rücksprungadresse ist sicher mit \$00 (Zeropage!); hier erfolgt also immerder Sprung in die Endbehandlung für Bildschirmausgaben

,e70b	ca	>dex	Nummer der Cursorzeile verringern (= in darüberliegende Zeile bewegen)
,e70c	86 d6	stx d6	und als neuen Wert für die aktuelle Cursorzeile setzen
,e70e	20 6c e5	jsr e56c "stupt"	Zeiger für Bildschirmspeicher und Farb-RAM aktualisieren
,e711	a4 d5	ldy d5	LNMx (physikalische Bildschirmzeilenlänge) auslesen
,e713	84 d3	sty d3	und als aktuelle Cursorspalte setzen (Cursor auf letzte Spalte in logischer Zeile)
,e715	60	rts	Rücksprung von Routine

; SCROUT-Routine: BSOUT-Routine für Gerät #3 (Bildschirm);

,e716	48	pha	auszugebendes Zeichen auf den Stapel legen	} Akku, auszugebendes Zeichen, X- und Y-Register auf den Stapel legen
,e717	85 d7	sta d7	und in Hilfsspeicher als auszugebendes Zeichen merken	
,e719	8a	txa	X-Register in Akku bringen	
,e71a	48	pha	und auf den Stapel legen	
,e71b	98	tya	Y-Register in Akku bringen	
,e71c	48	pha	und auf den Stapel legen	
,e71d	a9 00	lda #00	Initialisierungswert für INPUT/GET-Flag laden	
,e71f	85 d0	sta d0	und in INPUT/GET-Flag für Tastatur schreiben	
,e721	a4 d3	ldy d3	Nummer der aktuellen Cursorzeile holen	
,e723	a5 d7	lda d7	auszugebendes Zeichen (s. \$e717) holen	
,e725	10 03	bpl e72a	b7 gelöscht (N=0): Sonderbehandlung für Zeichencodes \$00-\$7f	
,e727	4c d4 e7	jmp e7d4	Sonderbehandlung für Zeichencodes \$80-\$ff anspringen	

; Sonderbehandlung für die Bildschirmausgabe der Zeichencodes \$00-\$7f

,e72a	c9 0d	>cmp #0d	Vergleich mit ASCII-Code von [cr] (<RETURN>)
,e72c	d0 03	bne e731	keine Übereinstimmung (Z=0): keine CR-Sonderbehandlung
,e72e	4c 91 e8	jmp e891 "cr"	Ausführung des Zeichencodes \$0d

,e731	c9 20	>cmp #20	Vergleich mit ASCII-Code von [space]
,e733	90 10	bcc e745	kleinerer ASCII-Code, also Steuerzeichen (C=0): Steuerzeichen \$00-\$1f behandeln
,e735	c9 60	cmp #60	liegt ASCII-Code im Bereich \$20-\$5f?
,e737	90 04	bcc e73d	ja (C=0): b6 und b7 löschen, Quote-Mode-Flag evtl. invertieren, Zeichen in Bildschirmspeicher schreiben

; ASCII-Codes \$60-\$7f behandeln

```
,e739 29 df and #df %11011111 b5 löschen (zwecks Wandlung in den Bildschirmcode)
,e73b d0 02 bne e73f "jmp" weitere Behandlung: Quote-Mode-Flag ggf. invertieren, Zeichen in Bildschirmspeicher
```

; ASCII-Codes \$20-\$5f behandeln

```
,e73d 29 3f and #3f %00111111 b6 und b7 löschen (zwecks Wandlung in den Bildschirmcode)
,e73f 20 84 e6 jsr e684 "chgqut" Quote-Mode-Flag ggf. invertieren
,e742 4c 93 e6 jmp e693 in CHRRAM-Routine einsteigen (Zeichen in Bildschirmspeicher schreiben)
```

; ASCII-Codes \$00-\$1f (Steuerzeichen) behandeln

```
,e745 a6 d8 ldx d8 INSRT (Zähler für Einfügemodus) zwecks Test auslesen
,e747 f0 03 beq e74c keine Zeichen mehr einzufügen (Z=1): Sonderbehandlung für Steuerzeichen im
INSERT-Modus überspringen
,e749 4c 97 e6 jmp e697 in CHRRAM-Routine so einsteigen, daß b7 gesetzt wird (Reversdarstellung von
Steuerzeichen im INSERT-Modus)
```

```
,e74c c9 14 cmp #14 Vergleich mit ASCII-Code von <DEL>
,e74e d0 2e bne e77e keine Übereinstimmung (Z=0): Sonderbehandlung für <DEL> überspringen
```

; Ausführung des Steuerzeichens (\$14)

```
,e750 98 tya Cursorspalte (s. $e721) zwecks Test in Akku bringen
,e751 d0 06 bne e759 andere Cursorspalte als 0 (Z=0): Sonderbehandlung für <DEL> aus erster Spalte einer
Zeile überspringen
,e753 20 01 e7 jsr e701 Cursor vom linken Rand der einen Zeile an rechten Rand der vorhergehenden Zeile
,e756 4c 73 e7 jmp e773 Leerzeichen an aktuelle Cursorposition schreiben und Ende der Bildschirmausgabe
```

```
,e759 20 a1 e8 jsr e8a1 "movlft" Hilfsroutine für Cursorbewegungen nach links aufrufen
,e75c 88 dey Nummer der aktuellen Cursorspalte verringern (= Cursor nach links bewegen)
,e75d 84 d3 sty d3 und Ergebnis als aktuelle Cursorspalte setzen
,e75f 20 24 ea jsr ea24 "colptr" Zeiger auf aktuelle Position im Farb-RAM aktualisieren
,e762 c8 iny alten Wert der Cursorzeile (s. $e75c) wiederherstellen
,e763 b1 d1 lda (d1),y Zeichen an alter Position aus Bildschirmspeicher holen
,e765 88 dey Offset auf vorhergehendes Zeichen stellen
,e766 91 d1 sta (d1),y Zeichen dorthin schreiben
,e768 c8 iny Offset wieder auf alte Position stellen
```

```
,e769 b1 f3 lda (f3),y Farbe an alter Position aus Farb-RAM holen
,e76b 88 dey Offset auf vorhergehendes Zeichen stellen
,e76c 91 f3 sta (f3),y Farbe dorthin schreiben
,e76e c8 iny Offset wieder auf alte Position stellen
,e76f c4 d5 cpy d5 mit LNMX (physikalische Bildschirmzeilenlänge) vergleichen
,e771 d0 ef bne e762 keine Übereinstimmung (Z=0): Linksverschieben von Bildschirm- und Farb-RAM fortsetzen
```

; Einsprung: Leerzeichen an Löschposition schreiben

```
,e773 a9 20 lda #20 Bildschirmcode von [SPACE] laden
,e775 91 d1 sta (d1),y und an Position im Bildschirmspeicher schreiben
,e777 ad 86 02 lda 0286 COLOR (aktuelle Zeichenfarbe) auslesen
,e77a 91 f3 sta (f3),y und an Position im Farb-RAM schreiben
,e77c 10 4d bpl e7cb "jmp" indirekt an Schlußbehandlung für Bildschirmausgabe springen
```

; ASCII-Codes \$00-\$1f (außer \$14) behandeln

```
,e77e a6 d4 ldx d4 QTSW (Quote-Mode-Flag) zwecks Test auslesen
,e780 f0 03 beq e785 nicht im Quote Mode (Z=1): Steuerzeichen ausführen
,e782 4c 97 e6 jmp e697 Steuerzeichen als reverses Zeichen (b7 setzen) in Bildschirmspeicher schreiben
```

```
,e785 c9 12 cmp #12 Vergleich mit ASCII-Code von [rvs on]
,e787 d0 02 bne e78b keine Übereinstimmung (Z=0): RVS-ON-Sonderbehandlung überspringen
```

; Sonderbehandlung für <RVS ON> (\$12)

```
,e789 85 c7 sta c7 $12 (s. $e785/$e787) in RVS (Revers-Flag) schreiben, also RVS-Flag setzen
,e78b c9 13 cmp #13 Vergleich mit ASCII-Code von [home]
,e78d d0 03 bne e792 keine Übereinstimmung (Z=0): HOME-Sonderbehandlung überspringen
```

; Sonderbehandlung für <HOME> (\$13)

```
,e78f 20 66 e5 jsr e566 "home" HOME-Routine aufrufen; danach steht kein Steuerzeichen-Code im Akku (!)
,e792 c9 1d cmp #1d Vergleich mit ASCII-Code von [crsr right]
,e794 d0 17 bne e7ad keine Übereinstimmung (Z=0): CRSR-RIGHT-Sonderbehandlung überspringen
```

; Sonderbehandlung für <CRSR RIGHT> (\$1d)

,e796	c8		iny	Cursorspalte erhöhen (Cursor um 1 Spalte nach rechts bewegen)
,e797	20 b3	e8	jsr e8b3 "movrgh"	Hilfsroutine für <CRSR RIGHT>-Bewegung aufrufen
,e79a	84 d3		sty d3	neue Cursorspalte setzen
,e79c	88		dey	Wert der Cursorspalte um 1 verringern
,e79d	c4 d5		cpy d5	und mit LNMX (Nummer der letzten Cursorspalte in aktueller Zeile) vergleichen
,e79f	90 09		bcc e7aa	Cursorspalte < LNMX (C=0): Schlußbehandlung für Bildschirmausgabe
,e7a1	c6 d6		dec d6	Nummer der aktuellen Cursorzeile verringern (Cursor nach oben bewegen)
,e7a3	20 7c	e8	jsr e87c "movcrs"	Hilfsroutine für Cursorpositionierungen aufrufen
,e7a6	a0 00		ldy #00	Spaltenwert der am weitesten links stehenden Spalteladen
,e7a8	84 d3		sty d3	und als aktuelle Cursorspalte setzen
,e7aa	4c a8	e6	jmp e6a8	Schlußbehandlung für Bildschirmausgabe

; weitere Behandlung von Steuercodes < \$20

,e7ad	c9 11		cmp #11	Vergleich mit ASCII-Code von [CRSR DOWN]
,e7af	d0 1d		bne e7ce	keine Übereinstimmung (Z=0): CRSR-DOWN-Sonderbehandlung überspringen

; Behandlung des Steuerzeichens [CRSR DOWN] (\$11)

,e7b1	18		clc	Carry vor Addition bei \$e7b3 löschen
,e7b2	98		tya	aktuelle Spaltennummer zwecks Addition in Akku bringen
,e7b3	69 28		adc #28	40 (Anzahl der Spalten pro Zeile) addieren
,e7b5	a8		tay	und Ergebnis als neue Spaltennummer merken
,e7b6	e6 d6		inc d6	Nummer der aktuellen Cursorzeile erhöhen (Abwärtsbewegung)
,e7b8	c5 d5		cmp d5	Vergleich der Spalte (s. \$e7b3) mit LNMX (physikalische Bildschirmzeilenlänge)
,e7ba	90 ec		bcc e7a8	Spalte < LNMX (C=0): Cursorspalte gemäß Y-Register setzen und Schlußbehandlung
,e7bc	f0 ea		beq e7a8	Spalte = LNMX (Z=1): Cursorspalte gemäß Y-Register setzen und Schlußbehandlung
,e7be	c6 d6		dec d6	Nummer der aktuellen Cursorzeile wieder verringern (s. \$e7b6)
,e7c0	e9 28		sbc #28	40 (Anzahl der Spalten pro Zeile) subtrahieren; C=1 wg. \$e7ba
,e7c2	90 04		bcc e7c8	Subtraktionsübertrag (C=0): Setzen der Cursorspalte überspringen
,e7c4	85 d3		sta d3	Subtraktionsergebnis als neue Cursorspalte setzen
,e7c6	d0 f8		bne e7c0	Cursorspalte <> 0 (Z=0): weitere Subtraktion von 40
,e7c8	20 7c	e8	jsr e87c "setnwl"	neue Zeile einrichten
,e7cb	4c a8	e6	jmp e6a8	Schlußbehandlung für Bildschirmausgabe anspringen

; weitere Behandlung von Steuercodes < \$20

```
,e7ce 20 cb e8 → jsr e8cb "colcod" Farbststeuerzeichen bearbeiten (sofern vorliegend)
,e7dl 4c 44 ec jmp ec44 weitere Behandlung von Steuercodes < $20 anspringen
```

; Bildschirmausgabe der ASCII-Codes > \$7f

```
,e7d4 29 7f and #7f %01111111 b7 löschen
,e7d6 c9 7f cmp #7f %01111111 Test, ob vorher $ff (%11111111, ASCII-Code von  $\pi$ ) vorlag
,e7d8 d0 02 bne e7dc nein (Z=0):  $\pi$ -Sonderbehandlung überspringen
```

; Sonderbehandlung für π (\$ff)

```
,e7da a9 5e lda #5e Bildschirmcode von  $\pi$  laden
,e7dc c9 20 → cmp #20 Vergleich mit ASCII-Code von [space]
,e7de 90 03 bcc e7e3 kleinerer Bildschirmcode (C=0): Sonderbehandlung von Steuerzeichen $80-$9f
,e7e0 4c 91 e6 jmp e691 "chrram" Zeichen in Bildschirmspeicher übernehmen
```

; Sonderbehandlung für Steuerzeichen \$80-\$9f

Im Akku steht der Zeichencode, wobei allerdings Bit 7 gelöscht ist.

```
,e7e3 c9 0d → cmp #0d Vergleich mit [SHIFT CR] (b7 ist hier gelöscht)
,e7e5 d0 03 bne e7ea keine Übereinstimmung (Z=0): SHIFT-CR-Sonderbehandlung überspringen
```

; Sonderbehandlung für Steuerzeichen [SHIFT CR] (\$8d)

```
,e7e7 4c 91 e8 jmp e891 "cr" dieselbe Routine wie für [CR] ($0d) aufrufen
```

```
,e7ea a6 d4 → ldx d4 QTSW (Quote-Mode-Flag) zwecks Test auslesen
,e7ec d0 3f bne e82d im Quote Mode (Z=0): b6 setzen und Zeichen revers in Bildschirmspeicher schreiben
```

; Ausführung der Steuercodes \$80-\$9f (außer \$8d)

```
,e7ee c9 14 cmp #14 Vergleich mit [INST] (b7 ist hier gelöscht)
,e7f0 d0 37 bne e829 keine Übereinstimmung (Z=0): INST-Sonderbehandlung überspringen
```


; Sonderbehandlung für Steuerzeichen [INST] (\$94)

,e7f2	a4 d5	ldy d5	LNMX (Spaltennummer der letzten Spalte der aktuellen logischen Zeile) holen	
,e7f4	b1 d1	lda (d1),y	letztes Zeichen der aktuellen logischen Zeile auslesen	
,e7f6	c9 20	cmp #20	und mit ASCII-Code für [space] vergleichen	
,e7f8	d0 04	bne e7fe	keine Übereinstimmung (Z=0): Vergleich mit aktueller Cursorspalte überspringen	
,e7fa	c4 d3	cpy d3	befindet sich Cursor in letzter Spalte?	
,e7fc	d0 07	bne e805	nein (Z=0): zeilenübergreifende Einfügung überspringen	
,e7fe	c0 4f	→ cpy #4f	Cursor in letztmöglicher Spalte?	
,e800	f0 24	beq e826	ja (Z=1): sofortiger Sprung zur Schlußbehandlung, da kein Einfügen möglich	
,e802	20 65	e9 jsr e965 "inslin"	Zeile einfügen	
,e805	a4 d5	→ ldy d5	LNMX (Spaltennummer der letzten Spalte der aktuellen logischen Zeile) holen	
,e807	20 24	ea jsr ea24 "colptr"	Zeiger auf aktuelle Adresse im Farb-RAM stellen	
,e80a	88	→ dey	Offset verringern (auf vorhergehende Spalte stellen)	} Bildschirmcode und Farbcode um 1 Position nach vorne verschieben
,e80b	b1 d1	lda (d1),y	vorhergehendes Zeichen aus Bildschirmspeicher auslesen	
,e80d	c8	iny	Offset wieder erhöhen (s. \$e80a)	
,e80e	91 d1	sta (d1),y	Zeichen an darauffolgende Position schreiben	
,e810	88	dey	Offset verringern (auf vorhergehende Spalte stellen)	
,e811	b1 f3	lda (f3),y	Farbcode aus vorhergehender Position im Farb-RAM holen	
,e813	c8	iny	Offset wieder erhöhen (s. \$e810)	
,e814	91 f3	sta (f3),y	Farbcode an darauffolgende Position schreiben	
,e816	88	dey	Offset verringern	
,e817	c4 d3	cpy d3	und mit aktueller Cursorspalte vergleichen	
,e819	d0 ef	bne e80a	Offset noch nicht auf Cursorspalte heruntergezählt (Z=0): weiter in Schleife	
,e81b	a9 20	lda #20	ASCII-Code für [space] laden	
,e81d	91 d1	sta (d1),y	und an aktuelle Position im Bildschirmspeicher schreiben	
,e81f	ad 86 02	lda 0286	COLOR (aktueller Farbcode) laden	
,e822	91 f3	sta (f3),y	und an aktuelle Position im Farb-RAM schreiben	
,e824	e6 d8	inc d8	INST (Zähler für Anzahl der noch zu tätigen INSERT-Einfügungen) erhöhen	
,e826	4c a8	e6 → jmp e6a8	Schlußbehandlung für Bildschirmausgabe anspringen	

; Behandlung weiterer Steuerzeichen im ASCII-Code-Bereich \$80—\$9f

,e829	a6 d8	ldx d8	INST (Zähler für Anzahl der noch zu tätigen INSERT-Einfügungen) testweise auslesen
,e82b	f0 05	beq e832	keine <INST>-Einfügungen mehr (Z=1): Steuerzeichen ausführen
,e82d	09 40	ora #40 %01000000	b6 setzen
,e82f	4c 97 e6	jmp e697	in CHRRAM-Routine einsteigen, damit Zeichen revers in Bildschirmspeicher kommt

; weitere Steuerzeichen im ASCII-Code-Bereich \$80-\$9f ausführen.

```
,e832 c9 11    ↳cmp #11          Vergleich mit Wert für [CRSR UP] (b7 gelöscht)
,e834 d0 16    ↳bne e84c          keine Übereinstimmung (Z=0): CRSR-UP-Sonderbehandlung überspringen
```

; Behandlung des Steuerzeichens <CRSR UP> (\$91)

```
,e836 a6 d6    ldx d6          Nummer der aktuellen Cursorzeile holen
,e838 f0 37    ↳beq e871          oberste Bildschirmzeile (Z=1): effektiv zur Schlußbehandlung der Bildschirmausgabe
,e83a c6 d6    dec d6          Nummer der aktuellen Cursorzeile verringern (= Cursor nach oben bewegen)
,e83c a5 d3    lda d3          Nummer der aktuellen Cursorspalte zwecks Auswertung holen
,e83e 38       sec            Carry vor Subtraktion setzen
,e83f e9 28    sbc #28         40 (Anzahl der Spalten pro Zeile) subtrahieren
,e841 90 04    ↳bcc e847          Subtraktionsübertrag (C=0): Setzen der Cursorspalte überspringen
,e843 85 d3    sta d3          Subtraktionsergebnis ("modulo 40") als Cursorspalte setzen
,e845 10 2a    ↳bpl e871 "jmp"   effektiv zur Schlußbehandlung für Bildschirmausgabe springen
```

```
-----
,e847 20 6c e5 ↳jsr e56c "stupt" Hilfszeiger für Bildschirmspeicher und Farb-RAM berechnen
,e84a d0 25    ↳bne e871 "jmp"   effektiv zur Schlußbehandlung für Bildschirmausgabe springen
-----
```

; Behandlung weiterer Steuerzeichen im ASCII-Code-Bereich \$80-\$9f

```
,e84c c9 12    ↳cmp #12          Vergleich mit Wert für [RVS OFF] (b7 gelöscht)
,e84e d0 04    ↳bne e854          keine Übereinstimmung (Z=0): RVS-OFF-Sonderbehandlung überspringen
```

; Behandlung des Steuerzeichens [RVS ON] (\$92)

```
,e850 a9 00    lda #00          Flag für "nicht im Revers-Modus" laden      } RVS-Flag
,e852 85 c7    sta c7          und als RVS-Flag setzen                } löschen

,e854 c9 1d    ↳cmp #1d          Vergleich mit Wert für von [CRSR LEFT] (b7 gelöscht)
,e856 d0 12    ↳bne e86a          keine Übereinstimmung (Z=0): CRSR-LEFT-Sonderbehandlung überspringen
```

; Behandlung des Steuerzeichens [CRSR LEFT] (\$9d)

```
,e858 98       tya            Cursorspalte zwecks Test in Akku bringen
,e859 f0 09    ↳beq e864          Cursor in Spalte 0 (Z=1): in darüberliegende Zeile springen
,e85b 20 a1 e8 ↳jsr e8a1 "movlft" Hilfsroutine für Cursorbewegungen nach links
,e85e 88       dey            Spaltenzähler verringern
,e85f 84 d3    sty d3          und als aktuelle Cursorspalte setzen
```

```

,e861 4c a8 e6 jmp e6a8          Schlußbehandlung für Bildschirmausgabe
-----
,e864 20 01 e7 jsr e701          Cursor vom linken Zeilenrand in darüberliegende Zeile bewegen
,e867 4c a8 e6 jmp e6a8          Schlußbehandlung für Bildschirmausgabe
-----

; Behandlung weiterer Steuerzeichen im Bereich $80-$9f

,e86a c9 13    cmp #13           Vergleich mit Wert für [clr home] (b7 gelöscht)
,e86c d0 06    bne e874          keine Übereinstimmung (Z=0): CLR-HOME-Sonderbehandlung überspringen

; Behandlung des Steuerzeichens [CLR] ($93)

,e86e 20 44 e5 jsr e544 "clear"   Bildschirm löschen
,e871 4c a8 e6 jmp e6a8          Schlußbehandlung für Bildschirmausgabe
-----
,e874 09 80    ora #80 %10000000 b7 wieder setzen
,e876 20 cb e8 jsr e8cb "colcod"   möglicherweise vorliegende Farbsteuerzeichen ausführen
,e879 4c 4f ec jmp ec4f          ggf. $8e (Umschalten auf Großschrift) verarbeiten
-----

; SETNWL-Hilfsroutine: neue Zeile einrichten

,e87c 46 c9    lsr c9            Spalte des Cursors für INPUT rechtsverschieben (Flag für "Einrichtung einer neuen
,e87e a6 d6    ldx d6            Nummer der aktuellen Cursorzeile auslesen
,e880 e8      inx               um 1 erhöhen (= 1 Zeile nach unten bewegen)
,e881 e0 19    cpx #19           Vergleich mit 25 (erster unerlaubter Wert für Zeilenzahl)
,e883 d0 03    bne e888          keine Übereinstimmung (Z=0): kein Scrolling
,e885 20 ea e8 jsr e8ea "scroll"   Bildschirm-Scrolling um 1 Zeile nach oben
,e888 b5 d9    lda d9,x          Eintrag aus LDTB1 (Tabelle der HBS der Bildschirmzeilenanfänge) holen
,e88a 10 f4    bpl e880          Fortsetzung einer logischen Zeile (N=0): auch diese Zeile wegscrollen
,e88c 86 d6    stx d6            Ergebnis-Offset als aktuelle Cursorzeile setzen
,e88e 4c 6c e5 jmp e56c "stupt"   Hilfszeiger für Bildschirmspeicher und Farb-RAM auf aktuelle Position stellen
-----

; CR-Routine: Sprung an Anfangsspalte der folgenden Zeile

,e891 a2 00    ldx #00          Initialisierungswert für Flags und Cursor-Spalte laden
,e893 86 d8    stx d8            INSRT löschen (Anzahl der noch zu tätigen Einfügungen auf 0 setzen)
,e895 86 c7    stx c7            RVS-Flag löschen

```

```
,e897 86 d4    stx  d4      QTSW (Quote-Mode-Flag) löschen
,e899 86 d3    stx  d3      Cursorspalte auf 0 setzen
,e89b 20 7c e8  jsr  e87c "setnwl" neue Zeile einrichten
,e89e 4c a8 e6  jmp  e6a8     Schlußbehandlung für Bildschirmausgabe anspringen
```

; MOVLFT-Hilfsroutine: bei Cursorbewegungen nach links

```
,e8a1 a2 02    ldx  #02      Dekrementierzähler initialisieren
,e8a3 a9 00    lda  #00      Ausgangswert 0 in Akku laden
,e8a5 c5 d3    >cmp  d3      Vergleich des Akku mit der aktuellen Cursorspalte
,e8a7 f0 07    >beq  e8b0     Übereinstimmung (Z=1): in darüberliegende Cursorzeile springen
,e8a9 18      clc           Carry vor Addition löschen
,e8aa 69 28    adc  #28      40 (Anzahl der Spalten pro Bildschirmzeile) addieren
,e8ac ca      dex          Dekrementierzähler verringern
,e8ad d0 f6    >bne  e8a5     noch nicht auf 0 heruntergezählt (Z=0): weiter in Additionsschleife
,e8af 60      rts          Rücksprung von Routine
```

```
,e8b0 c6 d6    >dec  d6      Nummer der aktuellen Cursorzeile verringern (= Bewegung um 1 Zeile nach oben)
,e8b2 60      rts          Rücksprung von Routine
```

; MOVRGH-Routine: bei Cursorbewegungen nach rechts

```
,e8b3 a2 02    ldx  #02      Dekrementierzähler initialisieren
,e8b5 a9 27    lda  #27      Ausgangswert 39 in Akku laden
,e8b7 c5 d3    >cmp  d3      Vergleich des Akku mit der aktuellen Cursorspalte
,e8b9 f0 07    >beq  e8c2     Übereinstimmung (Z=1): in darunterliegende Cursorzeile springen
,e8bb 18      clc           Carry vor Addition löschen
,e8bc 69 28    adc  #28      40 (Anzahl der Spalten pro Bildschirmzeile) addieren
,e8be ca      dex          Dekrementierzähler verringern
,e8bf d0 f6    >bne  e8b7     noch nicht auf 0 heruntergezählt (Z=0): weiter in Additionsschleife
,e8c1 60      rts          Rücksprung von Routine
```

```
,e8c2 a6 d6    >ldx  d6      Nummer der aktuellen Cursorzeile auslesen
,e8c4 e0 19    cpx  #19      Vergleich mit 25 (Nummer der letzten Zeile + 1)
,e8c6 f0 02    >beq  e8ca     Übereinstimmung (Z=1): Rücksprung über RTS
,e8c8 e6 d6    >inc  d6      Nummer der aktuellen Cursorzeile erhöhen (= Abwärtsbewegung)
,e8ca 60      >rts         Rücksprung von Routine
```

; COLCOD-Routine: Erkennung und Ausführung von Farbsteuerzeichen
Der zu überprüfende und ggf. auszuführende ASCII-Code ist im Akku zu übergeben.

,e8cb	a2 0f	ldx #0f	Dekrementierzähler für Suchschleife mit Anzahl der Codes -1 initialisieren
,e8cd	dd da e8	→cmp e8da,x	Vergleich des ASCII-Code mit Steuercode aus Tabelle
,e8d0	f0 04	beq e8d6	Übereinstimmung (Z=1): gefunden, Ausführung des Steuerzeichens
,e8d2	ca	dex	Dekrementierzähler verringern
,e8d3	10 f8	bpl e8cd	noch nicht auf \$ff heruntergezählt (N=0): Vergleich mit nächstem ASCII-Code
,e8d5	60	rts	Rücksprung von Routine, da kein Steuerzeichen im Akku vorlag

,e8d6	8e 86 02	→stx 0286	Offset in Tabelle als COLOR (aktuelle Zeichenfarbe) setzen
,e8d9	60	rts	Rücksprung von Routine

; Tabelle der ASCII-Codes aller Farbsteuerzeichen in Reihenfolge der Farbcodes des VIC

:e8da	90	[black]	(schwarz)	VIC-Farbcode \$00 = #00
:e8db	05	[white]	(weiß)	VIC-Farbcode \$01 = #01
:e8dc	1c	[red]	(rot)	VIC-Farbcode \$02 = #02
:e8dd	9f	[cyan]	(türkis)	VIC-Farbcode \$03 = #03
:e8de	9c	[purple]	(purpur)	VIC-Farbcode \$04 = #04
:e8df	1e	[green]	(grün)	VIC-Farbcode \$05 = #05
:e8e0	1f	[blue]	(blau)	VIC-Farbcode \$06 = #06
:e8e1	9e	[yellow]	(gelb)	VIC-Farbcode \$07 = #07
:e8e2	81	[orange]	(orange)	VIC-Farbcode \$08 = #08
:e8e3	95	[brown]	(braun)	VIC-Farbcode \$09 = #09
:e8e4	96	[light red]	(hellrot)	VIC-Farbcode \$0a = #10
:e8e5	97	[dark grey]	(dunkelgrau)	VIC-Farbcode \$0b = #11
:e8e6	98	[middle grey]	(mittelgrau)	VIC-Farbcode \$0c = #12
:e8e7	99	[light green]	(hellgrün)	VIC-Farbcode \$0d = #13
:e8e8	9a	[light blue]	(hellblau)	VIC-Farbcode \$0e = #14
:e8e9	9b	[light grey]	(hellgrau)	VIC-Farbcode \$0f = #15

; SCROLL-Routine: Aufwärts-Scrolling um 1 Bildschirmzeile

,e8ea	a5 ac	lda	ac	} Hilfszeiger \$ac-\$af für
,e8ec	48	pha		
,e8ed	a5 ad	lda	ad	} Bildschirmscrolling byteweise auf
,e8ef	48	pha		
,e8f0	a5 ae	lda	ae	

,e8f2	48	pha		den
,e8f3	a5 af	lda	af	} Stapel
,e8f5	48	pha		} retten
,e8f6	a2 ff	ldx	#ff	Inkrementierzähler initialisieren
,e8f8	c6 d6	dec	d6	Nummer der aktuellen Cursorzeile dekrementieren
,e8fa	c6 c9	dec	c9	Nummer der Cursorzeile bei INPUT dekrementieren
,e8fc	ce a5 02	dec	02a5	Zwischenspeicher \$02a5 dekrementieren
,e8ff	e8			Inkrementierzähler erhöhen
,e900	20 f0 e9	jsr	e9f0 "linadr"	Hilfszeiger \$dl/\$d2 auf Anfangsadresse der in X enthaltenen echten Zeile stellen
,e903	e0 18	cpx	#18	Nummer der zu bearbeitenden echten Zeile mit 24 (höchste Zeilennummer) vergleichen
,e905	b0 0c	bcs	e913	schon erreicht (C=1): Schleife verlassen
,e907	bd f1 ec	lda	ecf1,x	LB der Adresse der echten Zeile aus ROM-Tabelle auslesen
,e90a	85 ac	sta	ac	und in LB des Scrolling-Hilfszeigers \$ac/\$ad schreiben
,e90c	b5 da	lda	da,x	HB der Adresse aus LDTBl auslesen
,e90e	20 c8 e9	jsr	e9c8 "scrlin"	einzelne Zeile nach oben abrollen lassen
,e911	30 ec	bmi	e8ff "jmp"	nächste Zeile bearbeiten

,e913	20 ff e9	jsr	e9ff "dellin"	Zeile \$18 (unterste Zeile) löschen, da X=\$18 (s. \$e903/\$e905)
,e916	a2 00	ldx	#00	Offset mit 0 (Nummer der höchsten Bildschirmzeile) initialisieren
,e918	b5 d9	lda	d9,x	HB der Adresse der aktuellen Zeile aus LDTBl entnehmen
,e91a	29 7f	and	#7f %01111111	b7 (Flag für "Fortsetzung einer logischen Zeile") löschen
,e91c	b4 da	ldy	da,x	HB der Adresse der darauffolgenden Zeile aus LDTBl in Y-Register holen
,e91e	10 02	bpl	e922	darauffolgende Zeile ist Fortsetzung einer logischen Zeile (N=0): b7 nicht setzen
,e920	09 80	ora	#80 %10000000	b7 setzen (Zeile zur ersten/einzigen Zeile einer logischen Zeile ernennen)
,e922	95 d9	sta	d9,x	bei \$e920 möglicherweise (s. \$e91e) veränderten LDTBl-Eintrag der Zeile schreiben
,e924	e8	inx		Offset erhöhen
,e925	e0 18	cpx	#18	Vergleich mit Nummer der letzten Zeile
,e927	d0 ef	bne	e918	keine Übereinstimmung (Z=0): weiter mit Entfernen der logischen Zeilen, die mehr als eine echte Zeile umfassen
,e929	a5 f1	lda	f1	LDTBl-Eintrag der zweituntersten Bildschirmzeile holen
,e92b	09 80	ora	#80 %10000000	b7 setzen (zur ersten/einzigen Zeile einer logischen Zeile erklären)
,e92d	85 f1	sta	f1	und zurückschreiben
,e92f	a5 d9	lda	d9	LDTBl-Eintrag der obersten Bildschirmzeile holen
,e931	10 c3	bpl	e8f6	Fortsetzung einer logischen Zeile (N=0): an Anfang der Scroll-Routine springen, um Rest der oben befindlichen logischen Zeile auch noch wegzuscrollen
,e933	e6 d6	inc	d6	Nummer der aktuellen Cursorzeile erhöhen
,e935	ee a5 02	inc	02a5	Zwischenspeicher \$02a5 erhöhen
,e938	a9 7f	lda	#7f %01111111	OUTPUTS-Wert für Tastaturabfrage der Reihe von <CTRL> (auch <1>, <Linkspfeil>, <2>, <space>, <cbm>, <q>, <stop>)
,e93a	8d 00 dc	sta	dc00	und in CIA-Spaltenregister schreiben, damit entsprechende Abfrage erfolgt
,e93d	ad 01 dc	lda	dc01	CIA-Reihenregister auslesen

```
,e940 c9 fb      cmp #fb %l1l1l1011 Vergleich mit Wert für Tastaturreihe von<CTRL> (nur dieser Wert wird nach
                                     $e938/$e93a abgefragt)
,e942 08         php                                     Vergleichsergebnis auf den Stapel retten
,e943 a9 7f      lda #7f %01l1l1l1l OUTPUTS-Wert für Tastaturabfrage der Reihe von<CTRL> (auch <1>, <Linkspfeil>, <2>,
                                     <space>, <cbm>, <q>, <stop>)
,e945 8d 00 dc   sta dc00                                und in CIA-Spaltenregister schreiben, damit entsprechende Abfrage erfolgt
,e948 28         plp                                     bei $e942 gemerktes Vergleichsergebnis wieder vom Stapel holen
,e949 d0 0b      bne e956                                <CTRL> nicht gedrückt (Z=0): Warteschleifeüberspringen
```

; Warteschleife, da <CTRL> gedrückt wurde, um das Scrolling zu verlangsamen

```
,e94b a0 00      ldy #00                                Initialisierungswert für HB des Zählers
,e94d ea         nop                                    Verzögerung um 3 Taktzyklen
,e94e ca         dex                                    LB des Zählers dekrementieren
,e94f d0 fc      bne e94d                                noch nicht auf 0 heruntergezählt (Z=0): weiter verzögern
,e951 88         dey                                    HB des Zählers dekrementieren
,e952 d0 f9      bne e94d                                noch nicht auf 0 heruntergezählt (Z=0): weiter verzögern
,e954 84 c6      sty c6                                Anzahl der Zeichen im Tastaturpuffer mit 0 belegen, um Tastaturpuffer zu löschen;
                                                         Y=0 wegen $e951/$e952
,e956 a6 d6      rdx d6                                Nummer der aktuellen Cursorzeile holen
,e958 68         pla                                    bei $e8ea-$e8f5
,e959 85 af      sta af                                auf
,e95b 68         pla                                    den
,e95c 85 ae      sta ae                                Stapel
,e95e 68         pla                                    gerettete
,e95f 85 ad      sta ad                                Scrolling-Hilfszeiger
,e961 68         pla                                    wieder
,e962 85 ac      sta ac                                zurückholen
,e964 60         rts                                    Rücksprung
```

; INSLIN-Routine: an aktueller Cursorposition eine Zeile einfügen

```
,e965 a6 d6      ldx d6                                Nummer der aktuellen Cursorzeile holen
,e967 e8         inx                                    auf darauffolgende Zeile schalten
,e968 b5 d9      lda d9,x                                LDTBl-Eintrag der folgenden Zeile auslesen
,e96a 10 fb      bpl e967                                Fortsetzung einer logischen Zeile (N=0): noch eine Zeile weiter unten mit Einfügung
                                                         beginnen
,e96c 8e a5 02   stx 02a5                                Hilfsspeicher $02a5 mit Cursorzeile für Einfügung belegen
,e96f e0 18      cpx #18                                Einfügung in unterster Zeile?
,e971 f0 0e      beq e981                                ja (Z=1): Sonderbehandlung
,e973 90 0c      bcc e981                                Einfügung in Zeile #0 - Zeile #23 (C=0): Sonderbehandlung
```

; Sonderfall: Einfügung müßte außerhalb des erlaubten Bildschirmbereichs erfolgen

,e975	20 ea e8	jsr e8ea "scroll"	Aufwärts-Scrolling, um Platz für Einfügezeile zu schaffen
,e978	ae a5 02	ldx 02a5	in Hilfsspeicher gemerkte Zeilennummer für Einfügung holen
,e97b	ca	dex	auf vorhergehende Zeile schalten
,e97c	c6 d6	dec d6	Nummer der aktuellen Cursorzeile verringern
,e97e	4c da e6	jmp e6da	in UPTDL-Routine einsteigen

; Sonderfall: Einfügung in erlaubtem Bildschirmbereich

,e981	a5 ac	>lda ac	Hilfszeiger \$ac-\$af
,e983	48	pha	für
,e984	a5 ad	lda ad	Bildschirmscrolling
,e986	48	pha	byteweise
,e987	a5 ae	lda ae	auf
,e989	48	pha	den
,e98a	a5 af	lda af	Stapel
,e98c	48	pha	retten
,e98d	a2 19	ldx #19	Dekrementierzähler mit 25 (Nummer der untersten Zeile + 1) initialisieren
,e98f	ca	>dex	Dekrementierzähler verringern (auf darüberliegende Zeile stellen)
,e990	20 f0 e9	jsr e9f0 "linadr"	Hilfszeiger \$dl/\$d2 auf Anfangsadresse der in X enthaltenen echten Zeile stellen
,e993	ec a5 02	cpx 02a5	Dekrementierzähler mit Einfügezeile vergleichen
,e996	90 0e	bcc e9a6	Dekrementierzähler < Einfügezeile (C=0): noch nicht Platz schaffen
,e998	f0 0c	beq e9a6	Dekrementierzähler = Einfügezeile (Z=1): noch nicht Platz schaffen

; Dekrementierzähler > Einfügezeile (C=1 und Z=0)

,e99a	bd ef ec	lda ecef,x	LB der Adresse der Einfügezeile aus ROM-Tabelle entnehmen
,e99d	85 ac	sta ac	und als LB des Scrolling-Hilfszeigers \$ac/\$ad setzen
,e99f	b5 d8	lda d8,x	HB der Adresse der Einfügezeile aus LDtbl entnehmen
,e9a1	20 c8 e9	jsr e9c8 "scrlin"	einzelne Zeile nach oben abrollen lassen
,e9a4	30 e9	bmi e98f "jmp"	nächste Zeile bearbeiten

; Dekrementierzähler <= Einfügezeile (C=0 oder Z=1)

,e9a6	20 ff e9	>jsr e9ff "dellin"	aktuelle Zeile löschen
,e9a9	a2 17	ldx #17	Nummer der zweituntersten Bildschirmzeile laden
,e9ab	ec a5 02	>cpx 02a5	mit Einfügezeile vergleichen
,e9ae	90 0f	bcc e9bf	Einfügezeile >= 23 (C=0): Sonderbehandlung überspringen


```

,e9b0 b5 da lda da,x LDTB1-Eintrag von Zeile #24 auslesen
,e9b2 29 7f and #7f %01111111 b7 löschen
,e9b4 b4 d9 ldy d9,x LDTB1-Eintrag von Zeile #23 auslesen
,e9b6 10 02 bpl e9ba Fortsetzung einer logischen Zeile (N=0): b7 nicht setzen
,e9b8 09 80 ora #80 %10000000 b7 setzen (zur einzigen/ersten Zeile einer logischen Zeile erklären)
,e9ba 95 da >sta da,x LDTB1-Eintrag von Zeile #24 zurückschreiben
,e9bc ca dex "ldx #16" Zeilenzähler auf Zeile #22 stellen
,e9bd d0 ec bne e9ab "jmp" weiter in INSLIN-Schleife
-----
,e9bf ae a5 02->ldx 02a5 Nummer der Einfügezeile holen
,e9c2 20 da e6 jsr e6da in UPTDL-Routine einsteigen
,e9c5 4c 58 e9 jmp e958 bei $e981-$e98c gerettete Scrolling-Hilfszeiger wieder vom Stapel holen
-----

```

; SCRLIN-Routine: einzelne Zeile nach oben abrollen lassen

```

,e9c8 29 03 and #03 %00000011 b2-b7 löschen
,e9ca 0d 88 02 ora 0288 diese jedoch aus HIBASE (HB der Bildschirmspeicher-Anfangsadresse) einblenden
,e9cd 85 ad sta ad und Ergebnis als HB des Scrolling-Hilfszeigers $ac/$ad setzen
,e9cf 20 e0 e9 jsr e9e0 "coladr" Hilfszeiger $ae/$af auf korrespondierende Farb-RAM-Adresse stellen
,e9d2 a0 27 ldy #27 Dekrementierzähler mit Anzahl der Zeichen pro echter Zeile minus 1 initialisieren
,e9d4 b1 ac >lda (ac),y Zeichen aus Bildschirmspeicher holen
,e9d6 91 d1 sta (d1),y und an neue Adresse kopieren
,e9d8 b1 ae lda (ae),y Farbcode aus Farb-RAM holen
,e9da 91 f3 sta (f3),y und an neue Adresse kopieren
,e9dc 88 dey Dekrementierzähler verringern
,e9dd 10 f5 bpl e9d4 noch nicht auf $ff heruntergezählt (N=0): weiter in Verschiebeschleife
,e9df 60 rts Rücksprung von Routine
-----

```

; COLADR-Routine: Hilfszeiger \$f3/\$f3 und \$ae/\$af auf korrespondierende Farb-RAM-Adressen stellen

```

,e9e0 20 24 ea jsr ea24 "colptr" Zeiger auf aktuelle Position im Farb-RAM aktualisieren
,e9e3 a5 ac lda ac LB des Scrolling-Hilfszeigers $ac/$ad holen
,e9e5 85 ae sta ae und als LB des Scrolling-Farb-Hilfszeigers $ae/$af setzen
,e9e7 a5 ad lda ad LB des Scrolling-Hilfszeigers $ac/$ad holen
,e9e9 29 03 and #03 %00000011 b2-b7 löschen
,e9eb 09 d8 ora #d8 >($d800) HB der Anfangsadresse des Farb-RAM einblenden
,e9ed 85 af sta af und als HB des Scrolling-Farb-Hilfszeigers $ae/$af setzen
,e9ef 60 rts Rücksprung von Routine
-----

```

} Hilfszeiger
\$ae/\$af
auf
Farb-RAM-Adresse
zu \$ac/\$ad
stellen

; LINADR-Routine: Hilfszeiger \$d1/\$d2 auf in X enthaltene Zeile stellen

```
,e9f0 bd f0 ec lda ecf0,x      LB der Adresse der Zeile aus ROM-Tabelle entnehmen
,e9f3 85 d1 sta d1           und als LB des Ergebnis-Zeigers $d1/$d2 setzen
,e9f5 b5 d9 lda d9,x         HB der Adresse der Zeile aus RAM-Tabelle LDTB1 entnehmen
,e9f7 29 03 and #03 %00000011 b2-b7 löschen
,e9f9 0d 88 02 ora 0288      b0/b1 in HIBASE (HB der Basisadresse des Bildschirmspeichers) einblenden
,e9fc 85 d2 sta d2           und als HB des Ergebnis-Zeigers $d1/$d2 setzen
,e9fe 60 rts                 Rücksprung von Routine
```

; DELLIN-Routine: Bildschirmzeile löschen, deren Nummer im X-Register enthalten ist

```
,e9ff a0 27 ldy #27          Dekrementierzähler mit 40 (Anzahl der Zeichen pro Zeile -1) initialisieren
,ea01 20 f0 e9 jsr e9f0 "linadr" Hilfszeiger $d1/$d2 auf in X enthaltene Zeile stellen
,ea04 20 24 ea jsr ea24 "colptr" Zeiger auf aktuelle Position im Farb-RAM aktualisieren
,ea07 20 da e4 →jsr e4da      Hilfsroutine zum Setzen der Zeichenfarbe (nicht bei ältester C 64-Version!) aufrufen
,ea0a a9 20 lda #20          Bildschirmcode des Leerzeichens (Löschzeichen-Funktion) laden
,ea0c 91 d1 sta (d1),y       Leerzeichen in Bildschirmspeicher schreiben
,ea0e 88 dey                Dekrementierzähler verringern
,ea0f 10 f6 bpl ea07         noch nicht auf $ff heruntergezählt (N=0): weiter in Lösch-Schleife
,eall 60 rts                 Rücksprung von Routine
```

; Füllbefehl

```
,eal2 ea nop                keine Wirkung
```

; SETCHC-Routine: Zeichen- und Farbcode in Bildschirmspeicher und Farb-RAM übernehmen

```
,eal3 a8 tay                Zeichencode in Y-Register merken (bis $ealb)
,eal4 a9 02 lda #02         Initialisierungswert für Cursorzähler laden
,eal6 85 cd sta cd          und in BLNCT (Zähler für blinkenden Cursor) schreiben
,eal8 20 24 ea jsr ea24 "colptr" Zeiger auf Farb-RAM aktualisieren
,ealb 98 tya                bei $eal3 gemerkten Bildschirmcode wieder in Akku holen
,ealc a4 d3 ldy d3          aktuelle Cursorspalte als Offset von Basisadresse der Bildschirmzeile holen
,eale 91 d1 sta (d1),y      Bildschirmcode an aktuelle Position im Bildschirmspeicher schreiben
,ea20 8a txa                Farbcode in Akku holen
,ea21 91 f3 sta (f3),y      und an aktuelle Position im Farb-RAM schreiben
,ea23 60 rts                 Rücksprung von Routine
```

; COLPTR-Routine: Hilfszeiger \$f3/\$f4 gemäß \$d1/\$d2 aktualisieren

```
,ea24 a5 d1    lda    d1        LB des Bildschirmspeicher-Zeigers $d1/$d2 holen
,ea26 85 f3    sta    f3        und als LB von Farb-RAM-Zeiger $f3/$f4 übernehmen
,ea28 a5 d2    lda    d2        HB des Bildschirmspeicher-Zeigers $d1/$d2 holen
,ea2a 29 03    and    #03 %00000011 b2-b7 löschen
,ea2c 09 d8    ora    #d8 >($d800) b0/bl in HB der Basisadresse des Farb-RAM einblenden
,ea2e 85 f4    sta    f4        und Ergebnis als HB von Farb-RAM-Zeiger $f3/$f4 setzen
,ea30 60      rts                Rücksprung von Routine
```

; IRQ-Routine des Betriebssystems; hierher wird bei \$ff58 gesprungen

```
,ea31 20 ea ff jsr    ffea "udtim" Vorbereitung der Abfrage von <STOP>, Weiterzählen der Systemuhr TI/TI$
```

; Cursor-Behandlung in IRQ-Routine

```
,ea34 a5 cc    lda    cc        BLNSW (Flag für "Cursor an/aus") auslesen
,ea36 d0 29    bne    ea61       kein blinkender Cursor (Z=0): Cursorbehandlungsteil verlassen
,ea38 c6 cd    dec    cd        BLNCT (Zähler für blinkenden Cursor) verringern
,ea3a d0 25    bne    ea61       noch nicht auf 0 heruntergezählt (Z=0): Cursorbehandlungsteil verlassen
,ea3c a9 14    lda    #14       Initialisierungswert 20 für BLNCT (Zähler für blinkenden Cursor) laden
,ea3e 85 cd    sta    cd        und in BLNCT (Zähler für blinkenden Cursor) schreiben
,ea40 a4 d3    ldy    d3        Nummer der aktuellen Cursorspalte als Offset von Basisadresse der Cursorzeile holen
,ea42 46 cf    lsr    cf        BLNON (Flag für "Cursor in Blinkphase") durch Rechtsverschiebung löschen
,ea44 ae 87 02 ldx    0287      GDCOL (Hintergrundfarbe unter Cursor) holen
,ea47 b1 d1    lda    (d1),y     über Zeiger auf Basisadresse der aktuellen Cursorzeile: Zeichen unter Cursor holen
,ea49 b0 11    bcs    ea5c       BLNON war vor $ea42 gesetzt (C=1): Zeichen invertieren und zurückschreiben
,ea4b e6 cf    inc    cf        BLNON setzen (wird hier von 0 auf 1 erhöht)
,ea4d 85 ce    sta    ce        und Zeichen an Cursorposition als GDBLN (Zeichen unter Cursor) setzen
,ea4f 20 24 ea jsr    ea24 "colptr" Zeiger $f3/$f4 auf korrespondierende Farb-RAM-Adresse stellen
,ea52 b1 f3    lda    (f3),y     Farbcode an Cursorposition holen
,ea54 8d 87 02 sta    0287      und als GDCOL (Hintergrundfarbe unter Cursor) setzen
,ea57 ae 86 02 ldx    0286      aktuelle Zeichenfarbe (COLOR) als zu setzende Hintergrundfarbe holen
,ea5a a5 ce    lda    ce        GDBLN (Zeichen unter Cursor) holen
,ea5c 49 80    eor    #80 %10000000 b7 (RVS-Bit) invertieren = Zeichen invertieren
,ea5e 20 1c ea jsr    ealc       in SETCHC-Routine einsteigen, so daß Farbe und Zeichen gesetzt werden
```

; Kassettenmotor-Abfrage im System-IRQ

```
,ea61 a5 01 → lda 01      Prozessorport (P6502: 6502 Eingabe-Ausgabe-Register) auslesen
,ea63 29 10    and #10 %00010000 alle Bits bis auf b4 (Kassettenmotor-Anforderungsbit) löschen
,ea65 f0 0a    beq ea71      Taste an Datasette gedrückt, die Motor startet (Z=1): Motor anschalten
```

; Kassettenmotor ausschalten

```
,ea67 a0 00    ldy #00      Initialisierungswert für CAS1 laden (Kassettenmotor-Flag)
,ea69 84 c0    sty c0        und in CAS1 (Kassettenmotor-Flag) schreiben
,ea6b a5 01    lda 01        Prozessorport (P6502: 6502 Eingabe-Ausgabe-Register) auslesen
,ea6d 09 20    ora #20 %00100000 darin b5 setzen (= Kassettenmotor ausschalten)
,ea6f d0 08    bne ea79 "jmp" neuen Wert in Prozessorport schreiben, dann weiter in IRQ
```

; Kassettenmotor einschalten

```
,ea71 a5 c0 → lda c0        CAS1 (Kassettenmotor-Flag) zwecks Test auslesen
,ea73 d0 06    bne ea7b      Kassettenmotor bereits eingeschaltet (Z=0): weiter in System-IRQ
,ea75 a5 01    lda 01        Prozessorport (P6502: 6502 Eingabe-Ausgabe-Register) auslesen
,ea77 29 1f    and #1f %00011111 b5-b7 löschen (= Kassettenmotor einschalten)
,ea79 85 01 → sta 01        und neuen Wert in Prozessorport schreiben
```

; weiterer System-IRQ (nach Abfrage der Datensettentasten)

```
,ea7b 20 87 ea → jsr ea87 "scnkey" SCNKEY-Routine aufrufen

,ea7e ad 0d dc lda dc0d      IRQ-Flagregister auslesen (READ-Zugriff löscht IRQ-Flag in b7!)
,ea81 68      pla            geretteten Inhalt des Y-Registers (s. $ff4b/$ff4c) holen
,ea82 a8      tay            und in Y-Register bringen
,ea83 68      pla            geretteten Inhalt des X-Registers (s. $ff49/$ff4a) holen
,ea84 aa      tax            und in X-Register bringen
,ea85 68      pla            geretteten Inhalt des Akkumulators (s. $ff48) holen
,ea86 40      rti            Rücksprung von IRQ in Hauptprogramm
```

} Prozessorregister
Akku,
X und
Y vom
Stapel holen

; SCNKEY-Routine (hierher wird von Kernall-Aufruf bei \$ff9f gesprungen)

```
,ea87 a9 00    lda #00      Flag für "keine Taste <CTRL>, <CBM> oder <SHIFT> gedrückt" laden
,ea89 8d 8d 02 sta 028d      und in SHFLAG schreiben (Flag für <CTRL>, <CBM> und <SHIFT>)
,ea8c a0 40    ldy #40 %01000000 Tastencode für "keine Taste" laden
,ea8e 84 cb    sty cb        und in SFDX schreiben
```

} SHFLAG
und
SFDX
initialisieren

,ea90	8d 00 dc	sta dc00	alle Spalten (A=0 seit \$ea87) abfragen
,ea93	ae 01 dc	ldx dc01	Reihe der gedrückten Taste holen
,ea96	e0 ff	cpx #ff %11111111	Vergleich mit Rückgabewert für "keine Taste in durchsuchter Spalte"
,ea98	f0 61	beq eafb	Übereinstimmung (Z=1): Schlußbehandlung der Tastaturabfrage, da keine Taste gedrückt
,ea9a	a8	tay "ldy #00"	Offset für \$eab7 hier initialisieren (A=0 seit \$ea87)
,ea9b	a9 81	lda #81 <(\$eb81)	LB der Adresse der Tastaturlabelle für "normal" laden
,ea9d	85 f5	sta f5	und in LB des KEYTAB-Zeigers schreiben
,ea9f	a9 eb	lda #eb >(\$eb81)	HB der Adresse der Tastaturlabelle für "normal" laden
,eaa1	85 f6	sta f6	und in HB des KEYTAB-Zeigers schreiben
,eaa3	a9 fe	lda #fe %11111110	OUTPUTS-Wert für Abfrage der Spalte #0 laden
,eaa5	8d 00 dc	sta dc00	und in CIA-Register für Spaltenabfrage schreiben
,eaa8	a2-08	ldx #08	Anzahl der zu testenden Tastaturspalten als Dekrementierzähler laden
,eaaa	48	pha	Bitmuster für Abfrage der aktuellen Tastaturspalte auf Stapel retten
,eaab	ad 01 dc	lda dc01	Daten-Port B laden
,eaae	cd 01 dc	cmp dc01	und mit Inhalt von Daten-Port B nach (!) Lesezugriff vergleichen
,eabl	d0 f8	bne eaab	keine Übereinstimmung (Z=0): Tastaturabfrage noch nicht bereit
,eab3	4a	lsr	b0 aus Daten-Port B (enthält Nummer der Tastaturreihe) ins Carry holen
,eab4	b0-16	bcs eacc	b0 war gesetzt (C=1): weiter, da Reihe der gedrückten Taste noch nicht gefunden
,eab6	48	pha	bitverschobenen Port-B-Inhalt (Tastaturreihe) merken
,eab7	b1 f5	lda (f5),y	über KEYTAB-Zeiger das zu testende Byte der zu prüfenden Tastaturspalte aus der aktuellen Tastaturdekodierungstabelle holen
,eab9	c9 05	cmp #05	Vergleich mit <CTRL>+<2> (niedrigster Code, der durch <CTRL>+<Ziffer> erreicht wird)
,eabb	b0 0c	bcs eac9	erster Code der Reihe >=<CTRL>+<2> (C=1): keine <CTRL>-Tastaturlabelle
,eabd	c9 03	cmp #03	Vergleich mit <RUN/STOP> (niedrigster Code, der ohne Zusatztaste erreicht wird)
,eabf	f0 08	beq eac9	Übereinstimmung (Z=1): Position in Tabelle als Tastencode zurückgeben
,eac1	0d 8d 02	ora 028d	zunächst ermittelten Tastencode von Zusatztaste mit SHFLAG verknüpfen und in SHFLAG
,eac4	8d 8d 02	sta 028d	(Flag für Zusatztasten <CONTROL>, <CBM> und <SHIFT>) schreiben
,eac7	10 02	bpl eacb "jmp"	Rückgabe des Offset als Tastencode überspringen

,eac9	84 cb	sty cb	Offset in Tastaturlabelle als SFDX (gedrückte Taste als Tastencode) zurückgeben
,eacb	68	pla	bei \$eab6 gemerkten bitverschobenen Port-B-Inhalt wieder holen
,eacc	c8	iny	Offset erhöhen
,eacd	c0 41	cpy #41	schon komplette Tastaturlabelle durchsucht?
,eacf	b0 0b	bcs eadc	ja (C=1): Schleife verlassen, Taste auswerten
,ead1	ca	dex	Dekrementierzähler für Anzahl der noch zu durchsuchenden Tastaturspalten verringern
,ead2	d0-df	bne eab3	noch nicht auf 0 heruntergezählt (Z=0): weiter mit Bitverschiebung
,ead4	38	sec	Carry setzen, damit dieses bei \$ead6 als b0 eingebunden wird
,ead5	68	pla	bei \$eaaa gemerktes Bitmuster für Abfrage wieder vom Stapel holen
,ead6	2a	rol	und durch Linksverschiebung die Abfrage der nächsten Spalte vorbereiten
,ead7	8d 00 dc	sta dc00	OUTPUTS-Wert für neue Tastaturspalte in CIA-Register schreiben

,eada d0 cc — bne eaa8 "jmp" weiter mit Abfrage der Tastaturreihe

,eadc 68 —> pla bei \$eaaa gemerktes Bitmuster für Abfrage am Stapel löschen

; KEYLOG-Einsprung: Auswertung von Tastencode und SHFLAG zur Ermittlung des ASCII-Codes, der in den Tastaturpuffer geschrieben wird

,eadd 6c 8f 02 jmp(028f) Sprung über KEYLOG-Vektor (zeigt normalerweise nach \$eae0)

,eae0 a4 cb ldy cb SFDX (Tastaturcode der gedrückten Taste) als Offset holen
 ,eae2 b1 f5 lda (f5),y über KEYTAB-Zeiger den ASCII-Code holen
 ,eae4 aa tax und ins X-Register übernehmen
 ,eae5 c4 c5 cpy c5 Tastaturcode mit LSTX (Tastaturcode der letzten Taste) vergleichen
 ,eae7 f0 07 beq eaf0 Übereinstimmung (Z=1): Sonderbehandlung für Tastatur-Repeat
 ,eae9 a0 10 ldy #10 %00010000 Initialisierungswert für DELAY (Verzögerung bei Tastaturwiederholung) laden
 ,eaeB 8c 8c 02 sty 028c und in DELAY schreiben
 ,eaeE d0 36 — bne eb26 "jmp" Repeat-Sonderbehandlung überspringen

; Sonderbehandlung für Tastatur-Repeat

,eaf0 29 7f —> and #7f %01111111 b7 löschen
 ,eaf2 2c 8a 02 bit 028a RPTFLG (Flag für Tastatur-Wiederholungen) testen
 ,eaf5 30 16 bmi eb0d Repeat für alle Tasten (N=1): aktuelle Taste darf wiederholt werden
 ,eaf7 70 49 bvs eb42 kein Repeat (V=1): Tastendruck ignorieren

; Sonderfall: Repeat nur für , <SPACE>, <CRSR>

,eaf9 c9 7f cmp #7f %01111111 Vergleich mit Tastaturcode für "keine Taste"
 ,eafb f0 29 beq eb26 Übereinstimmung (Z=1): Repeat-Sonderbehandlung verlassen
 ,eafd c9 14 cmp #14 Vergleich mit ASCII-Code für
 ,eaff f0 0c beq eb0d Übereinstimmung (Z=1): aktuelle Taste darf wiederholt werden
 ,eb01 c9 20 cmp #20 Vergleich mit ASCII-Code für <SPACE>
 ,eb03 f0 08 beq eb0d Übereinstimmung (Z=1): aktuelle Taste darf wiederholt werden
 ,eb05 c9 1d cmp #1d Vergleich mit ASCII-Code für <CRSR RIGHT>
 ,eb07 f0 04 beq eb0d Übereinstimmung (Z=1): aktuelle Taste darf wiederholt werden
 ,eb09 c9 11 cmp #11 Vergleich mit ASCII-Code für <CRSR DOWN>
 ,eb0b d0 35 bne eb42 keine Übereinstimmung (Z=1): Tastendruck ignorieren, da Repeat unzulässig

} Vergleich mit
ASCII-Codes für
repeat-fähige
Tasten; bei
Übereinstimmung
wird Repeat
durchgeführt

; Durchführung eines Repeat, nachdem festgestellt wurde, daß die aktuelle Taste repeat-zulässig ist

,eb0d	ac	8c	02	→ldy 028c	DELAY (Zähler für Repeat-Verzögerung) zwecks Test auslesen
,eb10	f0	05		beq eb17	schon abgelaufen (Z=1): auch Zählgeschwindigkeit für Repeat testen
,eb12	ce	8c	02	dec 028c	DELAY (Zähler für Repeat-Verzögerung) herunterzählen
,eb15	d0	2b		bne eb42	noch nicht auf 0 heruntergezählt (Z=0): Tastendruck ignorieren
,eb17	ce	8b	02	→dec 028b	KOUNT (enthält Zählgeschwindigkeit für Repeat) verringern
,eb1a	d0	26		bne eb42	noch nicht auf 0 heruntergezählt (Z=0): Tastendruck ignorieren
,eb1c	a0	04		ldy #04 %00000100	Zählgeschwindigkeit für Repeat laden
,eb1e	8c	8b	02	sty 028b	und in KOUNT (enthält Zählgeschwindigkeit für Repeat) schreiben
,eb21	a4	c6		ldy c6	NDX (Anzahl der Zeichen im Tastaturpuffer) holen
,eb23	88			dey	zwecks Test verringern
,eb24	10	1c		bpl eb42	vor Verringern war NDX<>0 (N=0): Tastendruck ignorieren

; hier endet der Repeat-Sonderbehandlungsteil; wird diese Stelle durchlaufen, erfolgt eine Übernahme der Taste in den Tastaturpuffer (vorher Umwandlung in ASCII-Code!)

,eb26	a4	cb		→ldy cb	SFDX (Tastaturcode der gedrückten Taste) holen
,eb28	84	c5		sty c5	und als LSTX (Tastaturcode der letzten Taste) setzen
,eb2a	ac	8d	02	ldy 028d	SHFLAG (Flag für <SHIFT>, <CTRL> und <CBM>) auslesen
,eb2d	8c	8e	02	sty 028e	und als LSTSHF (letztes SHIFT/CTRL/CBM-Muster der Tastatur) setzen
,eb30	e0	ff		cpx #ff	bei \$eae4 ins X-Register geretteten ASCII-Code mit Code für "keine Taste" vergleichen
,eb32	f0	0e		beq eb42	Übereinstimmung (Z=1): Tastendruck ignorieren
,eb34	8a			txa	ASCII-Code in Akku holen
,eb35	a6	c6		ldx c6	NDX (Anzahl der Zeichen im Tastaturpuffer) als Offset holen
,eb37	ec	89	02	cpx 0289	mit XMAX (maximale Anzahl von Zeichen im Tastaturpuffer) vergleichen
,eb3a	b0	06		bcs eb42	NDX >= XMAX (C=1): Tastendruck ignorieren, da Tastaturpuffer sonst überläuft
,eb3c	9d	77	02	sta 0277,x	ASCII-Code in Tastaturpuffer an letzte Position schreiben
,eb3f	e8			inx	NDX (Anzahl der Zeichen im Tastaturpuffer) erhöhen
,eb40	86	c6		stx c6	und als neuen Inhalt von NDX setzen
,eb42	a9	7f		→lda #7f %01111111	OUTPUTS-Wert für "keine Abfrage" laden
,eb44	8d	00	dc	sta dc00	und in CIA-Register schreiben
,eb47	60			rts	Rücksprung von Routine

; Unterroutine zur Prüfung von <SHIFT>, <CBM> und <CTRL>

,eb48	ad	8d	02	lda 028d	SHFLAG (Flag für <CTRL>, <CBM> und <SHIFT>) auslesen
,eb4b	c9	03		cmp #03 %00000011	Vergleich mit Bitkombination für <SHIFT>+<CBM>
,eb4d	d0	15		bne eb64	keine Übereinstimmung (Z=0): keine Sonderbehandlung für Zeichensatzwechsel

; Sonderbehandlung: <SHIFT>+<CBM>

.eb4f	cd 8e 02	cmp 028e	Vergleich von SHFLAG mit LSTSHF (letztes SHFLAG)	
.eb52	f0 ee	beq eb42	Übereinstimmung (Z=1): Tastendruck ignorieren	
.eb54	ad 91 02	lda 0291	MODE (Flag, ob <SHIFT>+<CBM> berücksichtigt werden soll) zwecks Test auslesen	
.eb57	30 1d	bmi eb76	<SHIFT>+<CBM> gesperrt (N=1): Rücksprung ohne Sonderbehandlung	
.eb59	ad 18 d0	lda d018	VIC-Register #25 auslesen	} Zeichensatz- Umschaltung über VIC #25
.eb5c	49 02	eor #02 %00000010	bl (zuständig für Zeichensatz-Adresse) umblenden	
.eb5e	8d 18 d0	sta d018	und Ergebnis in VIC-Register #25 zurückschreiben	
.eb61	4c 76 eb	jmp eb76	Rücksprung in Tastaturbehandlung	

.eb64	0a	asl	SHFLAG verdoppeln	
.eb65	c9 08	cmp #08	Vergleich mit verdoppeltem Wert für <CTRL>	
.eb67	90 02	bcc eb6b	<CTRL> ist nicht gedrückt (C=0): Offset 8 in Tabelle der Basisadressen verwenden	
.eb69	a9 06	lda #06	Offset 6 in Tabelle der Basisadressen für Tastaturtabellen verwenden	
.eb6b	aa	tax	Offset in X-Register bringen	
.eb6c	bd 79 eb	lda eb79,x	LB der Adresse aus ROM-Tabelle entnehmen	} Adresse mittels Offset aus ROM-Tabelle der Basisadressen holen und in RAM-Zeiger schreiben
.eb6f	85 f5	sta f5	und als LB von KEYTAB (RAM-Zeiger) setzen	
.eb71	bd 7a eb	lda eb7a,x	HB der Adresse aus ROM-Tabelle entnehmen	
.eb74	85 f6	sta f6	und als HB von KEYTAB (RAM-Zeiger) setzen	
.eb76	4c e0 ea	jmp eae0	Rücksprung in Tastaturbehandlung	

; ROM-Tabelle der Basisadressen für die Tastaturtabellen

.eb79 81 eb	Adresse von Tastaturtabelle #0 (Tasten ohne <SHIFT>, <CBM> oder <CTRL>)
.eb7b c2 eb	Adresse von Tastaturtabelle #1 (Tasten mit <SHIFT>)
.eb7d 03 ec	Adresse von Tastaturtabelle #2 (Tasten mit <CBM>)
.eb7f 78 ec	Adresse von Tastaturtabelle #3 (Tasten mit <CONTROL>)

; Tastaturtabelle #0: Tasten ohne Zusatztaste wie <SHIFT>, <CBM> oder <CTRL>

.eb81 14	Spalte #0/Reihe #0:	
.eb82 0d	Spalte #0/Reihe #1:	<RETURN>
.eb83 1d	Spalte #0/Reihe #2:	<CRSR RIGHT>
.eb84 88	Spalte #0/Reihe #3:	<F7>
.eb85 85	Spalte #0/Reihe #4:	<F1>
.eb86 86	Spalte #0/Reihe #5:	<F3>
.eb87 87	Spalte #0/Reihe #6:	<F5>
.eb88 11	Spalte #0/Reihe #7:	<CRSR DOWN>

:eb89 33	Spalte #1/Reihe #0:	3
:eb8a 57	Spalte #1/Reihe #1:	W
:eb8b 41	Spalte #1/Reihe #2:	A
:eb8c 34	Spalte #1/Reihe #3:	4
:eb8d 5a	Spalte #1/Reihe #4:	Z
:eb8e 53	Spalte #1/Reihe #5:	S
:eb8f 45	Spalte #1/Reihe #6:	E
:eb90 01	Spalte #1/Reihe #7:	<linkes SHIFT>
:eb91 35	Spalte #2/Reihe #0:	5
:eb92 52	Spalte #2/Reihe #1:	R
:eb93 44	Spalte #2/Reihe #2:	D
:eb94 36	Spalte #2/Reihe #3:	6
:eb95 43	Spalte #2/Reihe #4:	C
:eb96 46	Spalte #2/Reihe #5:	F
:eb97 54	Spalte #2/Reihe #6:	T
:eb98 58	Spalte #2/Reihe #7:	X
:eb99 37	Spalte #3/Reihe #0:	7
:eb9a 59	Spalte #3/Reihe #1:	Y
:eb9b 47	Spalte #3/Reihe #2:	G
:eb9c 38	Spalte #3/Reihe #3:	8
:eb9d 42	Spalte #3/Reihe #4:	B
:eb9e 48	Spalte #3/Reihe #5:	H
:eb9f 55	Spalte #3/Reihe #6:	U
:eba0 56	Spalte #3/Reihe #7:	V
:eba1 39	Spalte #4/Reihe #0:	9
:eba2 49	Spalte #4/Reihe #1:	I
:eba3 4a	Spalte #4/Reihe #2:	J
:eba4 30	Spalte #4/Reihe #3:	0
:eba5 4d	Spalte #4/Reihe #4:	M
:eba6 4b	Spalte #4/Reihe #5:	K
:eba7 4f	Spalte #4/Reihe #6:	0
:eba8 4e	Spalte #4/Reihe #7:	N
:eba9 2b	Spalte #5/Reihe #0:	+
:ebaa 50	Spalte #5/Reihe #1:	P
:ebab 4c	Spalte #5/Reihe #2:	L
:ebac 2d	Spalte #5/Reihe #3:	-
:ebad 2e	Spalte #5/Reihe #4:	

:ebae 3a	Spalte #5/Reihe #5:	:
:ebaf 40	Spalte #5/Reihe #6:	@
:ebb0 2c	Spalte #5/Reihe #7:	,
:ebb1 5c	Spalte #6/Reihe #0:	£
:ebb2 2a	Spalte #6/Reihe #1:	*
:ebb3 3b	Spalte #6/Reihe #2:	;
:ebb4 13	Spalte #6/Reihe #3:	<HOME>
:ebb5 01	Spalte #6/Reihe #4:	<rechtes SHIFT>
:ebb6 3d	Spalte #6/Reihe #5:	=
:ebb7 5e	Spalte #6/Reihe #6:	↑
:ebb8 2f	Spalte #6/Reihe #7:	/
:ebb9 31	Spalte #7/Reihe #0:	1
:ebba 5f	Spalte #7/Reihe #1:	<Linkspfeil>
:ebbb 04	Spalte #7/Reihe #2:	<CTRL>
:ebbc 32	Spalte #7/Reihe #3:	2
:ebbd 20	Spalte #7/Reihe #4:	<SPACE>
:ebbe 02	Spalte #7/Reihe #5:	<CBM>
:ebbf 51	Spalte #7/Reihe #6:	Q
:ebc0 03	Spalte #7/Reihe #7:	<STOP>
:ebc1 ff	Code für "kein Tastendruck"	

; Tastaturtabelle #1: Tasten mit <SHIFT>

:ebc2 94	Spalte #0/Reihe #0:	<INST>
:ebc3 8d	Spalte #0/Reihe #1:	<SHIFT RETURN>
:ebc4 9d	Spalte #0/Reihe #2:	<CRSR LEFT>
:ebc5 8c	Spalte #0/Reihe #3:	<F8>
:ebc6 89	Spalte #0/Reihe #4:	<F2>
:ebc7 8a	Spalte #0/Reihe #5:	<F4>
:ebc8 8b	Spalte #0/Reihe #6:	<F6>
:ebc9 91	Spalte #0/Reihe #7:	<CRSR UP>
:ebca 23	Spalte #1/Reihe #0:	#
:ebcb d7	Spalte #1/Reihe #1:	<SHIFT>+<W>
:ebcc c1	Spalte #1/Reihe #2:	<SHIFT>+<A>
:ebcd 24	Spalte #1/Reihe #3:	\$

:ebce da	Spalte #1/Reihe #4:	<SHIFT>+<Z>
:ebcf d3	Spalte #1/Reihe #5:	<SHIFT>+<S>
:ebd0 c5	Spalte #1/Reihe #6:	<SHIFT>+<E>
:ebd1 01	Spalte #1/Reihe #7:	<linkes SHIFT>
:ebd2 25	Spalte #2/Reihe #0:	%
:ebd3 d2	Spalte #2/Reihe #1:	<SHIFT>+<R>
:ebd4 c4	Spalte #2/Reihe #2:	<SHIFT>+<D>
:ebd5 26	Spalte #2/Reihe #3:	&
:ebd6 c3	Spalte #2/Reihe #4:	<SHIFT>+<C>
:ebd7 c6	Spalte #2/Reihe #5:	<SHIFT>+<F>
:ebd8 d4	Spalte #2/Reihe #6:	<SHIFT>+<T>
:ebd9 d8	Spalte #2/Reihe #7:	<SHIFT>+<X>
:ebda 27	Spalte #3/Reihe #0:	'
:ebdb d9	Spalte #3/Reihe #1:	<SHIFT>+<Y>
:ebdc c7	Spalte #3/Reihe #2:	<SHIFT>+<G>
:ebdd 28	Spalte #3/Reihe #3:	(
:ebde c2	Spalte #3/Reihe #4:	<SHIFT>+
:ebdf c8	Spalte #3/Reihe #5:	<SHIFT>+<H>
:ebe0 d5	Spalte #3/Reihe #6:	<SHIFT>+<U>
:ebel d6	Spalte #3/Reihe #7:	<SHIFT>+<V>
:ebe2 29	Spalte #4/Reihe #0:	)
:ebe3 c9	Spalte #4/Reihe #1:	<SHIFT>+<I>
:ebe4 ca	Spalte #4/Reihe #2:	<SHIFT>+<J>
:ebe5 30	Spalte #4/Reihe #3:	<SHIFT>+<O>
:ebe6 cd	Spalte #4/Reihe #4:	<SHIFT>+<M>
:ebe7 cb	Spalte #4/Reihe #5:	<SHIFT>+<K>
:ebe8 cf	Spalte #4/Reihe #6:	<SHIFT>+<0>
:ebe9 ce	Spalte #4/Reihe #7:	<SHIFT>+<N>
:ebeb db	Spalte #5/Reihe #0:	<SHIFT>+<+>
:ebef d0	Spalte #5/Reihe #1:	<SHIFT>+<P>
:ebec cc	Spalte #5/Reihe #2:	<SHIFT>+<L>
:ebed dd	Spalte #5/Reihe #3:	<SHIFT>+<->
:ebee 3e	Spalte #5/Reihe #4:	>
:ebef 5b	Spalte #5/Reihe #5:	[
:ebf0 ba	Spalte #5/Reihe #6:	<SHIFT>+<@>
:ebf1 3c	Spalte #5/Reihe #7:	<

:ebf2 a9	Spalte #6/Reihe #0:	<SHIFT>+<f>
:ebf3 c0	Spalte #6/Reihe #1:	<SHIFT>+<*>
:ebf4 5d	Spalte #6/Reihe #2:	]
:ebf5 93	Spalte #6/Reihe #3:	<CLR>
:ebf6 01	Spalte #6/Reihe #4:	<rechtes SHIFT>
:ebf7 3d	Spalte #6/Reihe #5:	=
:ebf8 de	Spalte #6/Reihe #6:	π
:ebf9 3f	Spalte #6/Reihe #7:	?
:ebfa 21	Spalte #7/Reihe #0:	!
:ebfb 5f	Spalte #7/Reihe #1:	<Linkspfeil>
:ebfc 04	Spalte #7/Reihe #2:	<CTRL>
:ebfd 22	Spalte #7/Reihe #3:	"
:ebfe a0	Spalte #7/Reihe #4:	<SHIFT>+<SPACE>
:ebff 02	Spalte #7/Reihe #5:	<CBM>
:ec00 dl	Spalte #7/Reihe #6:	<SHIFT>+<Q>
:ec01 83	Spalte #7/Reihe #7:	<RUN>
:ec02 ff	Code für "kein Tastendruck"	

; Tastaturtabelle #2: Tasten mit <CBM>

:ec03 94	Spalte #0/Reihe #0:	<INST>
:ec04 8d	Spalte #0/Reihe #1:	<SHIFT RETURN>
:ec05 9d	Spalte #0/Reihe #2:	<CRSR LEFT>
:ec06 8c	Spalte #0/Reihe #3:	<F8>
:ec07 89	Spalte #0/Reihe #4:	<F2>
:ec08 8a	Spalte #0/Reihe #5:	<F4>
:ec09 8b	Spalte #0/Reihe #6:	<F6>
:ec0a 91	Spalte #0/Reihe #7:	<CRSR UP>
:ec0b 96	Spalte #1/Reihe #0:	<LIGHT RED>
:ec0c b3	Spalte #1/Reihe #1:	<CBM>+<W>
:ec0d b0	Spalte #1/Reihe #2:	<CBM>+<A>
:ec0e 97	Spalte #1/Reihe #3:	<DARK GREY>
:ec0f ad	Spalte #1/Reihe #4:	<CBM>+<Z>
:ec10 ae	Spalte #1/Reihe #5:	<CBM>+<S>
:ec11 b1	Spalte #1/Reihe #6:	<CBM>+<E>
:ec12 01	Spalte #1/Reihe #7:	<linkes SHIFT>

:ec13 98	Spalte #2/Reihe #0:	<MIDDLE GREY>
:ec14 b2	Spalte #2/Reihe #1:	<CBM>+<R>
:ec15 ac	Spalte #2/Reihe #2:	<CBM>+<D>
:ec16 99	Spalte #2/Reihe #3:	<LIGHT GREEN>
:ec17 bc	Spalte #2/Reihe #4:	<CBM>+<C>
:ec18 bb	Spalte #2/Reihe #5:	<CBM>+<F>
:ec19 a3	Spalte #2/Reihe #6:	<CBM>+<T>
:ec1a bd	Spalte #2/Reihe #7:	<CBM>+<X>
:ec1b 9a	Spalte #3/Reihe #0:	<LIGHT BLUE>
:ec1c b7	Spalte #3/Reihe #1:	<CBM>+<Y>
:ec1d a5	Spalte #3/Reihe #2:	<CBM>+<G>
:ec1e 9b	Spalte #3/Reihe #3:	<LIGHT GREY>
:ec1f bf	Spalte #3/Reihe #4:	<CBM>+
:ec20 b4	Spalte #3/Reihe #5:	<CBM>+<H>
:ec21 b8	Spalte #3/Reihe #6:	<CBM>+<U>
:ec22 be	Spalte #3/Reihe #7:	<CBM>+<V>
:ec23 29	Spalte #4/Reihe #0:	)
:ec24 a2	Spalte #4/Reihe #1:	<CBM>+<I>
:ec25 b5	Spalte #4/Reihe #2:	<CBM>+<J>
:ec26 30	Spalte #4/Reihe #3:	<CBM>+<O>
:ec27 a7	Spalte #4/Reihe #4:	<CBM>+<M>
:ec28 al	Spalte #4/Reihe #5:	<CBM>+<K>
:ec29 b9	Spalte #4/Reihe #6:	<CBM>+<0>
:ec2a aa	Spalte #4/Reihe #7:	<CBM>+<N>
:ec2b a6	Spalte #5/Reihe #0:	<CBM>+<+>
:ec2c af	Spalte #5/Reihe #1:	<CBM>+<P>
:ec2d b6	Spalte #5/Reihe #2:	<CBM>+<L>
:ec2e dc	Spalte #5/Reihe #3:	<CBM>+<->
:ec2f 3e	Spalte #5/Reihe #4:	>
:ec30 5b	Spalte #5/Reihe #5:	[
:ec31 a4	Spalte #5/Reihe #6:	<CBM>+<@>
:ec32 3c	Spalte #5/Reihe #7:	<
:ec33 a8	Spalte #6/Reihe #0:	<CBM>+<£>
:ec34 df	Spalte #6/Reihe #1:	<CBM>+<*>
:ec35 5d	Spalte #6/Reihe #2:	]
:ec36 93	Spalte #6/Reihe #3:	<CLR>
:ec37 01	Spalte #6/Reihe #4:	<rechtes SHIFT>

:ec38 3d	Spalte #6/Reihe #5:	=
:ec39 de	Spalte #6/Reihe #6:	π
:ec3a 3f	Spalte #6/Reihe #7:	?
:ec3b 81	Spalte #7/Reihe #0:	<ORANGE>
:ec3c 5f	Spalte #7/Reihe #1:	<Linkspfeil>
:ec3d 04	Spalte #7/Reihe #2:	<CTRL>
:ec3e 95	Spalte #7/Reihe #3:	<BROWN>
:ec3f a0	Spalte #7/Reihe #4:	<SHIFT>+<SPACE>
:ec40 02	Spalte #7/Reihe #5:	<CBM>
:ec41 ab	Spalte #7/Reihe #6:	<CBM>+<Q>
:ec42 83	Spalte #7/Reihe #7:	<RUN>
:ec43 ff	Code für "kein Tastendruck"	

; Steuerzeichen für Zeichensatzzeile bearbeiten
 Der ASCII-Code des Steuerzeichens wird im Akku übergeben.

:ec44 c9 0e	cmp #0e	Vergleich mit ASCII-Code von [business] (Klein/Groß-Schrift)
:ec46 d0 07	bne ec4f	keine Übereinstimmung (Z=0): Test auf [graphics]

; Ausführung von [business] (\$0e)

:ec48 ad 18 d0	lda d018	VIC-Register #24 auslesen	} auf "business" (Klein/Groß-Schrift) schalten
:ec4b 09 02	ora #02 %00000010	bl setzen	
:ec4d d0 09	bne ec58 "jmp"	und zurückschreiben	

:ec4f c9 8e	cmp #8e	Vergleich mit ASCII-Code von [graphics] (Groß/Grafik-Schrift)
:ec51 d0 0b	bne ec5e	keine Übereinstimmung (Z=0): Test auf [lock]

; Ausführung von [graphics] (\$8e)

:ec53 ad 18 d0	lda d018	VIC-Register #24 auslesen	} auf "graphics" (Groß/Grafik-Schrift) schalten
:ec56 29 fd	and #fd %11111101	bl löschen	
:ec58 8d 18 d0	sta d018	und zurückschreiben	
:ec5b 4c a8 e6	jmp e6a8	Schlußbehandlung für Bildschirmausgabe anspringen	

:ec5e c9 08	cmp #08	Vergleich mit ASCII-Code von [lock]
:ec60 d0 07	bne ec69	keine Übereinstimmung (Z=0): Test auf [unlock]

; Ausführung von [lock] (\$08)

```

,ec62 a9 80      lda #80 %10000000 b7 setzen
,ec64 0d 91 02   ora 0291          und in MODE (Flag für "<SHIFT>+<CBM> gesperrt") einblenden
,ec67 30 09      bmi ec72 "jmp"    und zurückschreiben
-----
,ec69 c9 09      >cmp #09          Vergleich mit ASCII-Code von [unlock]
,ec6b d0 ee      bne ec5b          keine Übereinstimmung (Z=0): Schlußbehandlung für Bildschirmausgabe

```

; Ausführung von [unlock] (\$09)

```

,ec6d a9 7f      lda #7f %01111111 b7 löschen
,ec6f 2d 91 02   and 0291          entsprechend in MODE (Flag für "<SHIFT>+<CBM> erlaubt") einblenden
,ec72 8d 91 02   >sta 0291         und zurückschreiben
,ec75 4c a8 e6   jmp e6a8          Schlußbehandlung für Bildschirmausgabe anspringen
-----

```

; Tastaturtabelle #3: Tasten mit <CONTROL>

,ec78 ff	Spalte #0/Reihe #0:	kein Tastendruck	
,ec79 ff	Spalte #0/Reihe #1:	kein Tastendruck	
,ec7a ff	Spalte #0/Reihe #2:	kein Tastendruck	
,ec7b ff	Spalte #0/Reihe #3:	kein Tastendruck	
,ec7c ff	Spalte #0/Reihe #4:	kein Tastendruck	
,ec7d ff	Spalte #0/Reihe #5:	kein Tastendruck	
,ec7e ff	Spalte #0/Reihe #6:	kein Tastendruck	
,ec7f ff	Spalte #0/Reihe #7:	kein Tastendruck	
,ec80 1c	Spalte #1/Reihe #0:	<RED>	= <CTRL>+<3>
,ec81 17	Spalte #1/Reihe #1:	<CTRL>+<W>	
,ec82 01	Spalte #1/Reihe #2:	<CTRL>+<A>	
,ec83 9f	Spalte #1/Reihe #3:	<CYAN>	= <CTRL>+<4>
,ec84 1a	Spalte #1/Reihe #4:	<CTRL>+<Z>	
,ec85 13	Spalte #1/Reihe #5:	<HOME>	= <CTRL>+<S>
,ec86 05	Spalte #1/Reihe #6:	<WHITE>	= <CTRL>+<E>
,ec87 ff	Spalte #1/Reihe #7:	<linkes SHIFT>	
,ec88 9c	Spalte #2/Reihe #0:	<PURPLE>	= <CTRL>+<5>
,ec89 12	Spalte #2/Reihe #1:	<RVS ON>	= <CTRL>+<R>
,ec8a 04	Spalte #2/Reihe #2:	<CTRL>+<D>	

:ec8b 1e	Spalte #2/Reihe #3:	<GREEN>	= <CTRL>+<6>
:ec8c 03	Spalte #2/Reihe #4:	<CTRL>+<C>	
:ec8d 06	Spalte #2/Reihe #5:	<CTRL>+<F>	
:ec8e 14	Spalte #2/Reihe #6:		= <CTRL>+<T>
:ec8f 18	Spalte #2/Reihe #7:	<CTRL>+<X>	
:ec90 1f	Spalte #3/Reihe #0:	<BLUE>	= <CTRL>+<7>
:ec91 19	Spalte #3/Reihe #1:	<CTRL>+<Y>	
:ec92 07	Spalte #3/Reihe #2:	<CTRL>+<G>	
:ec93 9e	Spalte #3/Reihe #3:	<YELLOW>	= <CTRL>+<8>
:ec94 02	Spalte #3/Reihe #4:	<CTRL>+	
:ec95 08	Spalte #3/Reihe #5:	<LOCK>	= <CTRL>+<H>
:ec96 15	Spalte #3/Reihe #6:	<CTRL>+<U>	
:ec97 16	Spalte #3/Reihe #7:	<CTRL>+<V>	
:ec98 12	Spalte #4/Reihe #0:	<RVS ON>	
:ec99 09	Spalte #4/Reihe #1:	<CTRL>+<I>	
:ec9a 0a	Spalte #4/Reihe #2:	<CTRL>+<J>	
:ec9b 92	Spalte #4/Reihe #3:	<RVS OFF>	= <CTRL>+<0>
:ec9c 0d	Spalte #4/Reihe #4:	<RETURN>	= <CTRL>+<M>
:ec9d 0b	Spalte #4/Reihe #5:	<CTRL>+<K>	
:ec9e 0f	Spalte #4/Reihe #6:	<CTRL>+<O>	
:ec9f 0e	Spalte #4/Reihe #7:	<BUSINESS>	= <CTRL>+<N>
:eca0 ff	Spalte #5/Reihe #0:	kein Tastendruck	
:eca1 10	Spalte #5/Reihe #1:	<CTRL>+<P>	
:eca2 0c	Spalte #5/Reihe #2:	<CTRL>+<L>	
:eca3 ff	Spalte #5/Reihe #3:	kein Tastendruck	
:eca4 ff	Spalte #5/Reihe #4:	kein Tastendruck	
:eca5 1b	Spalte #5/Reihe #5:	<CTRL>+<.:>	
:eca6 00	Spalte #5/Reihe #6:	<NUL>	
:eca7 ff	Spalte #5/Reihe #7:	kein Tastendruck	
:eca8 1c	Spalte #6/Reihe #0:	<RED>	
:eca9 ff	Spalte #6/Reihe #1:	kein Tastendruck	
:ecaa ld	Spalte #6/Reihe #2:	<CRSR RIGHT>	
:ecab ff	Spalte #6/Reihe #3:	kein Tastendruck	
:ecac ff	Spalte #6/Reihe #4:	kein Tastendruck	
:ecad 1f	Spalte #6/Reihe #5:	<synth.: \$1f>	
:ecae 1e	Spalte #6/Reihe #6:	<synth.: \$1e>	
:ecaf ff	Spalte #6/Reihe #7:	kein Tastendruck	


```

:ecb0 90      Spalte #7/Reihe #0:    <BLACK>
:ecb1 06      Spalte #7/Reihe #1:    <synth.: $06>
:ecb2 ff      Spalte #7/Reihe #2:    kein Tastendruck
:ecb3 05      Spalte #7/Reihe #3:    <WHITE>
:ecb4 ff      Spalte #7/Reihe #4:    kein Tastendruck
:ecb5 ff      Spalte #7/Reihe #5:    kein Tastendruck
:ecb6 11      Spalte #7/Reihe #6:    <CRSR DOWN>      = <CTRL>+<Q>
:ecb7 ff      Spalte #7/Reihe #7:    kein Tastendruck

:ecb8 ff      Code für "kein Tastendruck"
-----

```

; Initialisierungswerte für VIC-Register

```

:ecb9 00      Initialisierungswert für VIC-Register #00 (Sprite #0, LB der X-Koordinate)
:ecba 00      Initialisierungswert für VIC-Register #01 (Sprite #0, Y-Koordinate)
:ecbb 00      Initialisierungswert für VIC-Register #02 (Sprite #1, LB der X-Koordinate)
:ecbc 00      Initialisierungswert für VIC-Register #03 (Sprite #1, Y-Koordinate)
:ecbd 00      Initialisierungswert für VIC-Register #04 (Sprite #2, LB der X-Koordinate)
:ecbe 00      Initialisierungswert für VIC-Register #05 (Sprite #2, Y-Koordinate)
:ecbf 00      Initialisierungswert für VIC-Register #06 (Sprite #3, LB der X-Koordinate)
:ecc0 00      Initialisierungswert für VIC-Register #07 (Sprite #3, Y-Koordinate)
:ecc1 00      Initialisierungswert für VIC-Register #08 (Sprite #4, LB der X-Koordinate)
:ecc2 00      Initialisierungswert für VIC-Register #09 (Sprite #4, Y-Koordinate)
:ecc3 00      Initialisierungswert für VIC-Register #10 (Sprite #5, LB der X-Koordinate)
:ecc4 00      Initialisierungswert für VIC-Register #11 (Sprite #5, Y-Koordinate)
:ecc5 00      Initialisierungswert für VIC-Register #12 (Sprite #6, LB der X-Koordinate)
:ecc6 00      Initialisierungswert für VIC-Register #13 (Sprite #6, Y-Koordinate)
:ecc7 00      Initialisierungswert für VIC-Register #14 (Sprite #7, LB der X-Koordinate)
:ecc8 00      Initialisierungswert für VIC-Register #15 (Sprite #7, Y-Koordinate)

:ecc9 00      Initialisierungswert für VIC-Register #16 (höchste Bits der Sprite-X-Koordinaten)

:ecca 9b %10011011 Initialisierungswert für VIC-Register #17
                    b7: MSB des Rasterwertes aus VIC #18 = 1
                    b6: Extended Color Mode (ECM) aus    = 0
                    b5: Bit Map Mode (BMM) aus          = 0
                    b4: Blank Screen to Color Border an = 1
                    b3: 25-Zeilen-Bildschirm            = 1
                    b0-b2: Soft-Scroll vertikal über 3 Rasterzeilen = 011

```

```

:eccb 37      Initialisierungswert für VIC-Register #18 (LB der Rasterzeile)

:eccc 00      Initialisierungswert für VIC-Register #19 (X-Koordinate des Light-Pen)
:eccd 00      Initialisierungswert für VIC-Register #20 (Y-Koordinate des Light-Pen)
:ecce 00      Initialisierungswert für VIC-Register #21 (Sprites ausschalten)

:eccf 08 %00001000      Initialisierungswert für VIC-Register #22
                          b7: unbenutzt           = 0
                          b6: unbenutzt           = 0
                          b5: reserviert          = 0
                          b4: Priorität aus        = 0
                          b3: 40-Spalten-Bildschirm an = 1
                          b0-b2: Soft-Scroll horizontal aus = 000

:ecd0 00      Initialisierungswert für VIC-Register #23 (kein Sprite vertikal verdoppeln)

:ecd1 14 %00010100      Initialisierungswert für VIC-Register #24
                          b4-b7: b10-b13 der Adresse $0400 = 0001
                          b1-b3: b11-b13 der Adresse $d000 = 0100
                          b0:   unbenutzt           = 0

:ecd2 0f %00001111      Initialisierungswert für VIC-Register #25
                          b7: VIC-IRQ aus           = 0
                          b4-b6: unbenutzt          = 000
                          b3: Light-Pen-IRQ an      = 1
                          b2: Sprite/Sprite-Kollision an = 1
                          b1: Sprite/Hintergrund-Koll. an = 1
                          b0: Raster-Vergleich-IRQ an = 1

:ecd3 00      Initialisierungswert für VIC-Register #26 (kein IRQ "enabled")

:ecd4 00      Initialisierungswert für VIC-Register #27 (keine Sprite-Priorität)

:ecd5 00      Initialisierungswert für VIC-Register #28 (keine Multicolor-Sprites)

:ecd6 00      Initialisierungswert für VIC-Register #29 (kein Sprite horizontal verdoppeln)

:ecd7 00      Initialisierungswert für VIC-Register #30 (keine Sprite/Sprite-Kollision)

:ecd8 00      Initialisierungswert für VIC-Register #31 (keine Sprite/Hintergrund-Kollision)

```

```

:ecd9 0e      Initialisierungswert für VIC-Register #32 (Rahmenfarbe = 14)

:ecda 06      Initialisierungswert für VIC-Register #33 (Hintergrundfarbe #0 = 6)
:ecdb 01      Initialisierungswert für VIC-Register #34 (Hintergrundfarbe #1 = 1)
:ecdc 02      Initialisierungswert für VIC-Register #35 (Hintergrundfarbe #2 = 2)
:ecdd 03      Initialisierungswert für VIC-Register #36 (Hintergrundfarbe #3 = 3)
:ecde 04      Initialisierungswert für VIC-Register #37 (Sprite-MC-Farbe #0 = 4)
:ecdf 00      Initialisierungswert für VIC-Register #38 (Sprite-MC-Farbe #1 = 0)

:ece0 01      Initialisierungswert für VIC-Register #39 (Farbe von Sprite #0 = 1)
:ecel 02      Initialisierungswert für VIC-Register #40 (Farbe von Sprite #1 = 2)
:ece2 03      Initialisierungswert für VIC-Register #41 (Farbe von Sprite #2 = 3)
:ece3 04      Initialisierungswert für VIC-Register #42 (Farbe von Sprite #3 = 4)
:ece4 05      Initialisierungswert für VIC-Register #43 (Farbe von Sprite #4 = 5)
:ece5 06      Initialisierungswert für VIC-Register #44 (Farbe von Sprite #5 = 6)
:ece6 07      Initialisierungswert für VIC-Register #45 (Farbe von Sprite #6 = 7)

```

; ROM-Tabelle des Textes für <SHIFT>+<RUN/STOP>

```

:ec  e7 4c 4f 41 44 0d 52 55 4e 0d  load[cr]run[cr]

```

; ROM-Tabelle der LBs der Basisadressen der Bildschirmzeilen im Bildschirmspeicher

```

:ecf0 00 28 50 78 a0 c8 f0 18
:ecf8 40 68 90 b8 e0 08 30 58
:ed00 80 a8 d0 f8 20 48 70 98
:ed08 c0

```

; TALK-Routine (hierher wird von \$ffb4 verzweigt)

```

,ed09 09 40      ora #40 %01000000 b6 (TALK-Bit) im Akku setzen

```

; LISTEN-Routine (nach \$ed0c wird von \$ffb1 verzweigt)

```

,ed0b 2c 09 20  "bit" ora #20 %00100000 b5 (LISTEN-Bit) im Akku setzen
,ed0e 20 a4 f0  jsr f0a4 "rsp232"      Warten auf Ende von RS232-I/O

```

; Ausgabe eines Bytewertes im Akku mit ATN-Signal (Einsprung wird von \$ee00 genutzt)

,ed11	48	pha	Bytewert auf den Stapel legen	
,ed12	24 94	bit 94	C3P0 (Flag: "Zeichen im Puffer für IEC-Bus") testen	
,ed14	10 0a	bpl ed20	Flag gelöscht (N=0): Sonderbehandlung überspringen	
,ed16	38	sec	Carry setzen, damit dieses über ROR in b7 von Hilfsspeicher \$a3 kommt	
,ed17	66 a3	ror a3	b7 im Hilfsspeicher \$a3 setzen	
,ed19	20 40 ed	jsr ed40 "iecbyt"	Byte über IEC-Bus senden	
,ed1c	46 94	lsr 94	C3P0 (Flag: "Zeichen im Puffer für IEC-Bus") durch Rechtsverschiebung löschen	
,ed1e	46 a3	lsr a3	Hilfsspeicher \$a3 nach rechts verschieben	
,ed20	68	pla	bei \$ed11 gemerkten Bytewert vom Stapel holen	
,ed21	85 95	sta 95	und als BSOUR (Zeichen im Puffer für seriellen Bus) setzen	
,ed23	78	sei	Interrupt verhindern	
,ed24	20 97 ee	jsr ee97 "datahi"	DATA auf HIGH setzen	
,ed27	c9 3f	cmp #3f %00111111	CIA-Datenport A mit Wert vergleichen, der nie auftritt	
,ed29	d0 03	bne ed2e "jmp"	CLCKHI-Aufruf überspringen	

,ed2b	20 85 ee	jsr ee85 "clckhi"	CLOCK auf HIGH setzen	
,ed2e	ad 00 dd	lda dd00	CIA-Datenport A	} ATN auf LOW
,ed31	09 08	ora #08 %00001000	b3 (Bit für ATN-Signalausgabe) setzen, also auf LOW stellen	
,ed33	8d 00 dd	sta dd00	und zurückschreiben	
,ed36	78	sei	Interrupt verhindern	
,ed37	20 8e ee	jsr ee8e "clcklo"	CLOCK auf LOW setzen	
,ed3a	20 97 ee	jsr ee97 "datahi"	DATA auf HIGH setzen	
,ed3d	20 b3 ee	jsr eeb3 "wait.1"	10 ⁺ -3 Sekunden warten	
,ed40	78	sei	Interrupt verhindern	
,ed41	20 97 ee	jsr ee97 "datahi"	DATA auf HIGH setzen	
,ed44	20 a9 ee	jsr eea9 "debpia"	Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7	
,ed47	b0 64	bcs edad	DATA IN auf LOW (C=1): DEVICE NOT PRESENT als Kernal-Fehler auslösen	
,ed49	20 85 ee	jsr ee85 "clckhi"	CLOCK auf HIGH setzen	
,ed4c	24 a3	bit a3	Hilfsspeicher \$a3 testen	
,ed4e	10 0a	bpl ed5a	b7 gelöscht (N=0): nur auf 1 DATA-IN-LOW-Signal warten	
,ed50	20 a9 ee	jsr eea9 "debpia"	Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7	} auf DATA IN low warten
,ed53	90 fb	bcc ed50	DATA IN auf HIGH (C=0): warten, bis DATA IN auf LOW	
,ed55	20 a9 ee	jsr eea9 "debpia"	Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7	} auf DATA IN high warten
,ed58	b0 fb	bcs ed55	DATA IN auf LOW (C=1): warten, bis DATA IN auf HIGH	
,ed5a	20 a9 ee	jsr eea9 "debpia"	Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7	} auf DATA IN low warten
,ed5d	90 fb	bcc ed5a	DATA IN auf HIGH (C=0): warten, bis DATA IN auf LOW	


```

,ed5f 20 8e ee jsr ee8e "clcklo" CLOCK auf LOW setzen
,ed62 a9 08 lda #08 %00001000 Anzahl der Bits in 1 Byte laden
,ed64 85 a5 sta a5 und in Bitzähler CNTDN schreiben
,ed66 ad 00 dd → lda dd00 CIA-Datenport A auslesen
,ed69 cd 00 dd cmp dd00 Veränderung durch Lesezugriff?
,ed6c d0 f8 bne ed66 nein (Z=0): warten, bis Veränderung eintritt
,ed6e 0a asl b7 aus CIA-Datenport A ins Carry holen
,ed6f 90 3f bcc edb0 DATA IN auf HIGH (C=0): TIME OUT als Kernal-Fehler auslösen
,ed71 66 95 ror 95 nächstes Bit aus zu übertragendem Byte holen
,ed73 b0 05 bcs ed7a gesetztes Bit (C=1): DATA HIGH senden
,ed75 20 a0 ee jsr eea0 "datalo" DATA LOW senden (gelöschtes Bit übertragen)
,ed78 d0 03 bne ed7d "jmp" Senden von DATA HIGH überspringen
-----
,ed7a 20 97 ee → jsr ee97 "datahi" DATA HIGH senden (gesetztes Bit übertragen)
,ed7d 20 85 ee → jsr ee85 "clckhi" CLOCK auf HIGH setzen
,ed80 ea nop Verzögerung um 2 Taktzyklen
,ed81 ea nop Verzögerung um 2 Taktzyklen
,ed82 ea nop Verzögerung um 2 Taktzyklen
,ed83 ea nop Verzögerung um 2 Taktzyklen } Verzögerung um 8 Taktzyklen
,ed84 ad 00 dd lda dd00 CIA-Datenport A auslesen
,ed87 29 df and #df %11011111 b5 löschen (DATA auf HIGH)
,ed89 09 10 ora #10 %00010000 b4 setzen (CLOCK auf LOW)
,ed8b 8d 00 dd sta dd00 und zurückschreiben
,ed8e c6 a5 dec a5 Bitzähler CNTDN verringern, da 1 Bit übertragen wurde
,ed90 d0 d4 bne ed66 noch nicht alle Bits übertragen (Z=0): weiter mit nächstem Bit
,ed92 a9 04 lda #04 %00000100 Wartewert (ca. 10-3 sec) für Timer B HIGH laden
,ed94 8d 07 dc sta dc07 und in HB von Timer B schreiben
,ed97 a9 19 lda #19 %00010011 Force Load, Timer-B-Ausgabe, Timer-B-Start
,ed99 8d 0f dc sta dc0f in CRB (Control Register B) schreiben
,ed9c ad 0d dc lda dc0d ICR (Interrupt Control Register) auslesen, um b7 zu löschen
,ed9f ad 0d dc → lda dc0d ICR (Interrupt Control Register) auslesen
,eda2 29 02 and #02 %00000010 alle Bits bis auf bl (Timer-B-Interrupt) löschen
,eda4 d0 0a bne edb0 bl = 1 (Z=0): Kernal-Fehler TIME OUT auslösen
,eda6 20 a9 ee jsr eea9 "debpia" Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7
,eda9 b0 f4 bcs ed9f DATA IN auf LOW (C=1): weiter in Warteschleife
,edab 58 cli Interrupts wieder zulassen
,edac 60 rts Rücksprung von Routine
-----

```

; Fehler bei IEC-Bus verarbeiten

,edad	a9 80	lda #80	Fehlernummer für DEVICE NOT PRESENT laden
,edaf	2c a9	03→bit "lda #03"	Fehlernummer für TIME OUT laden
,edb2	20 1c fe	jsr felc "erstat"	Fehler in Statusbyte übernehmen
,edb5	58	cli	Interrupts wieder zulassen
,edb6	18	clc	Carry löschen
,edb7	90 4a	bcc ee03 "jmp"	ATN HIGH, Verzögerung, CLOCK HIGH, DATA HIGH und Ende

; SECOND-Routine: Sekundäradresse für LISTEN senden (hierher wird von \$ff93 verzweigt)

,edb9	85 95	sta 95	Sekundäradresse als BSOUR (Zeichen im Puffer für seriellen Bus) setzen
,edbb	20 36 ed	jsr ed36	und über IEC-Bus senden
,edbe	ad 00 dd	lda dd00	CIA-Datenport A auslesen
,edc1	29 f7	and #f7 %11110111	b3 (ATN-Bit) löschen, also ATN auf HIGH
,edc3	8d 00 dd	sta dd00	und zurückschreiben
,edc6	60	rts	Rücksprung von Routine

; TKSA-Routine: Sekundäradresse für TALK senden (hierher wird von \$ff96 verzweigt)

,edc7	85 95	sta 95	Sekundäradresse als BSOUR (Zeichen im Puffer für seriellen Bus) setzen
,edc9	20 36 ed	jsr ed36	und über IEC-Bus senden
,edcc	78	sei	Interrupt verhindern
,edcd	20 a0 ee	jsr eea0 "datalo"	DATA LOW senden (gelöschtes Bit übertragen)
,edd0	20 be ed	jsr edbe "atnhi"	CIA-Datenport A auslesen, ATN auf HIGH setzen
,edd3	20 85 ee	jsr ee85 "clckhi"	CLOCK auf HIGH setzen
,edd6	20 a9 ee	jsr eea9 "debpia"	Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7
,edd9	30 fb	bmi edd6	CLOCK IN auf LOW (N=1): warten auf CLOCK IN high
,eddb	58	cli	Interrupts wieder zulassen
,eddc	60	rts	Rücksprung von Routine

; CIOUT-Routine: Byte auf IEC-Bus ausgeben (hierher wird von \$ffa8 verzweigt)

,eddd	24 94	bit 94	C3P0 (Flag für "Zeichen im Puffer für seriellen Bus") testen
,eddf	30 05	bmi ede6	C3P0 gesetzt (N=1): zuerst altes Byte aus Puffer senden, dann neues
,edel	38	sec	Carry setzen, damit dieses bei Rechtsverschiebung in b7 von C3P0 kommt
,ede2	66 94	ror 94	C3P0 (Flag für "Zeichen im Puffer für seriellen Bus") setzen
,ede4	d0 05	bne edeb "jmp"	Byte in Puffer für seriellen Bus schreiben und Rücksprung

; Sonderfall: bereits Zeichen im Puffer für seriellen Bus vorhanden

```
,ede6 48      pha          neu zu übertragendes Byte merken
,ede7 20 40 ed  jsr ed40 "iecbyt" altes Byte aus Puffer über IEC-Bus senden
,edea 68      pla          bei $ede6 gemerktes, neu zu übertragendes Byte wieder holen
,edeb 85 95      sta 95      und in BSOUR (1-Byte-Puffer für seriellen Bus) schreiben
,eded 18      clc          Carry löschen (Flag für "kein I/O-Fehler")
,edee 60      rts          Rücksprung von Routine
```

; UNTALK-Routine: UNTALK-Signal über IEC-Bus senden (hierher wird von \$ffab verzweigt)

```
,edef 78      sei          Interrupt verhindern
,edf0 20 8e ee  jsr ee8e "clcklo" CLOCK auf LOW setzen
,edf3 ad 00 dd  lda dd00      CIA-Datenport A
,edf6 09 08      ora #08 %00001000 b3 (Bit für ATN-Signalausgabe) setzen, also auf LOW stellen } ATN
,edf8 8d 00 dd  sta dd00      und zurückschreiben } auf
,edfb a9 5f      lda #5f %01011111 Bitmuster für UNTALK laden } LOW
```

; UNLSN-Routine: UNLISTEN-Signal über IEC-Bus senden (nach \$edfe wird von \$ffae verzweigt)

```
,edfd 2c a9 3f  "bit" lda #3f %00111111 Bitmuster für UNLISTEN laden
,ee00 20 11 ed  jsr ed11      Ausgabe eines Bytewertes mit ATN-Signal
,ee03 20 be ed  jsr edbe "atnhi" CIA-Datenport A auslesen, ATN auf HIGH setzen
,ee06 8a      txa          Inhalt des X-Registers (soll unverändert bleiben) in Akku retten
,ee07 a2 0a      ldx #0a      Dekrementierzähler für Verzögerungsschleife initialisieren } Verzögerung
,ee09 ca      dex          Dekrementierzähler verringern } um 42
,ee0a d0 fd      bne ee09      noch nicht auf 0 heruntergezählt (Z=0): weiter verzögern } Taktzyklen
,ee0c aa      tax          X-Register (s. $ee06) wiederherstellen
,ee0d 20 85 ee  jsr ee85 "clckhi" CLOCK auf HIGH setzen
,ee10 4c 97 ee  jmp ee97 "datahi" DATA auf HIGH setzen
```

; IECIN-Routine: Byte vom IEC-Bus einlesen (hierher wird von \$ffa5 gesprungen)

```
,eel3 78      sei          Interrupts verhindern
,eel4 a9 00      lda #00      Löschwert für CNTDN (Bitzähler) laden } Bitzähler für
,eel6 85 a5      sta a5      und in CNTDN (Bitzähler für Übertragung) schreiben } Übertragung löschen
,eel8 20 85 ee  jsr ee85 "clckhi" CLOCK auf HIGH setzen
,eelb 20 a9 ee  jsr eea9 "debpia" Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7 } auf CLOCK IN low
,eelc 10 fb      bpl eelb      CLOCK IN auf HIGH (N=0): warten auf CLOCK IN low } warten
```

```

,ee20 a9 01 → lda #01 %00000001 Wartewert für Timer B HIGH laden
,ee22 8d 07 dc sta dc07 und in HB von Timer B schreiben
,ee25 a9 19 lda #19 %00011001 Force Load, Timer-B-Ausgabe, Timer-B-Start
,ee27 8d 0f dc sta dc0f in CRB (Control Register B) schreiben
,ee2a 20 97 ee jsr ee97 "datahi" DATA auf HIGH setzen
,ee2d ad 0d dc lda dc0d ICR (Interrupt Control Register) auslesen, um b7 zu löschen
,ee30 ad 0d dc → lda dc0d ICR (Interrupt Control Register) auslesen
,ee33 29 02 and #02 %00000010 alle Bits bis auf bl löschen; also nur bl (Timer-B-Interrupt) testen
,ee35 d0 07 bne ee3e bl = 1 (Z=0): ggf. Kernal-Fehler TIME OUT auslösen
,ee37 20 a9 ee jsr eea9 "debpia" Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7
,ee3a 30 f4 bmi ee30 CLOCK IN auf LOW (N=1): timerbezogene Warteschleife fortsetzen
,ee3c 10 18 bpl ee56 "jmp" bitweise Übertragung

```

; Fehlerbehandlung, falls Timer übergelaufen ist (TIME OUT)

```

,ee3e a5 a5 → lda a5 Bitzähler CNTDN auslesen
,ee40 f0 05 beq ee47 kein Bit mehr zu übertragen (Z=1): nicht TIME OUT
,ee42 a9 02 lda #02 Fehlernummer für TIME OUT laden
,ee44 4c b2 ed jmp edb2 Kernal-Fehler auslösen

```

```

,ee47 20 a0 ee → jsr eea0 "datalo" DATA LOW senden
,ee4a 20 85 ee jsr ee85 "clckhi" CLOCK auf HIGH setzen
,ee4d a9 40 lda #40 %01000000 b6 (END OF FILE) setzen
,ee4f 20 1c fe jsr felc "erstat" und in Statusbyte des Betriebssystems übernehmen
,ee52 e6 a5 inc a5 Bitzähler CNTDN erhöhen
,ee54 d0 ca bne ee20 "jmp" erneute Warteschleife

```

```

,ee56 a9 08 → lda #08 Anzahl der Bits in 1 Byte laden
,ee58 85 a5 sta a5 und in Bitzähler für Übertragung (CNTDN) schreiben
,ee5a ad 00 dd → lda dd00 CIA-Datenport A auslesen
,ee5d cd 00 dd cmp dd00 Veränderung durch Lesezugriff?
,ee60 d0 f8 bne ee5a ja (Z=0): warten, bis keine Veränderung mehr erfolgt
,ee62 0a asl b6 (serieller Bus: Impulseingabe) in b7 bringen
,ee63 10 f5 bpl ee5a b6 war 0 (N=0): warten, bis Impulseingabe möglich ist
,ee65 66 a4 ror a4 geholtes Bit (Carry) in eingelesenes Byte einbinden
,ee67 ad 00 dd → lda dd00 CIA-Datenport A auslesen
,ee6a cd 00 dd cmp dd00 Veränderung durch Lesezugriff?
,ee6d d0 f8 bne ee67 ja (Z=0): warten, bis keine Veränderung mehr erfolgt

```


,ee6f	0a	asl	b6 (serieller Bus: Impulseingabe) in b7 bringen
,ee70	30 f5	bmi ee67	b6 war 1 (N=0): warten, bis Impulseingabe gesperrt ist
,ee72	c6 a5	dec a5	Bitzähler CNTDN verringern
,ee74	d0 e4	bne ee5a	noch nicht alle Bits eingelesen (Z=0): weiter in Leseschleife
,ee76	20 a0 ee	jsr eea0 "datalo"	DATA LOW senden
,ee79	24 90	bit 90	Statusbyte des Betriebssystems testen
,ee7b	50 03	bvc ee80	kein EOF (V=0): Sonderbehandlung für END OF FILE überspringen
,ee7d	20 06 ee	jsr ee06	Warteschleife, CLOCK HIGH, DATA HIGH
,ee80	a5 a4	lda a4	eingelesenes Byte (s. \$ee65, wo die Bits eingebunden werden) in Akku laden
,ee82	58	cli	Interrupts wieder zulassen
,ee83	18	clc	Carry löschen (Flag für "kein Fehler")
,ee84	60	rts	Rücksprung von Routine

; CLCKHI-Routine: CLOCK auf HIGH setzen

,ee85	ad 00 dd	lda dd00	CIA-Datenport A auslesen
,ee88	29 ef	and #ef %11101111	b4 löschen (CLOCK high)
,ee8a	8d 00 dd	sta dd00	und zurückschreiben
,ee8d	60	rts	Rücksprung von Routine

; CLCKLO-Routine: CLOCK auf LOW setzen

,ee8e	ad 00 dd	lda dd00	CIA-Datenport A auslesen
,ee91	09 10	ora #10 %00010000	b4 setzen (CLOCK low)
,ee93	8d 00 dd	sta dd00	und zurückschreiben
,ee96	60	rts	Rücksprung von Routine

; DATAHI-Routine: DATA auf HIGH setzen

,ee97	ad 00 dd	lda dd00	CIA-Datenport A auslesen
,ee9a	29 df	and #df %11011111	b5 löschen (DATA high)
,ee9c	8d 00 dd	sta dd00	und zurückschreiben
,ee9f	60	rts	Rücksprung von Routine

; DATALO-Routine: DATA auf LOW setzen

,eea0	ad 00 dd	lda dd00	CIA-Datenport A auslesen
-------	----------	----------	--------------------------

```
,eea3 09 20    ora #20 %00100000 b5 löschen (DATA low)
,eea5 8d 00 dd  sta dd00          und zurückschreiben
,eea8 60        rts              Rücksprung von Routine
```

; DEBPiA-Routine: Datenport A auslesen: DATA IN in Carry, CLOCK IN in b7

```
,eea9 ad 00 dd  >lda dd00          CIA-Datenport A auslesen
,eeac cd 00 dd  <cmp dd00          Veränderung durch Lesezugriff?
,eeaf d0 f8      bne eea9          ja (Z=0): warten, bis keine Veränderung erfolgt
,eeb1 0a        asl              Linksverschiebung: b7=DATA IN in Carry; b6=CLOCK IN in b7 und N-Flag
,eeb2 60        rts              Rücksprung von Routine
```

; WAIT.1-Routine: 10⁺-3 Sekunden warten

```
,eeb3 8a        txa              X-Register bis $eeb9 in Akku retten
,eeb4 a2 b8      ldx #b8          Dekrementierzähler für Verzögerungsschleife initialisieren
,eeb6 ca        >dex              Dekrementierzähler verringern
,eeb7 d0 fd      < bne eeb6       noch nicht auf 0 heruntergezählt (Z=0): weiter in Verzögerungsschleife
,eeb9 aa        tax              X-Register (s. $eeb3) wiederherstellen
,eeba 60        rts              Rücksprung von Routine
```

; Ausgabe-Teilroutine des NMI bei RS232-Betrieb; Fortsetzung von \$fe9d

```
,eebb a5 b4      lda  b4          BITTS (Bitzähler für RS232) zwecks Test auslesen
,eebd f0 47      <beq ef06 "rstbgn" BITTS schon auf 0 heruntergezählt (Z=1): nächstes Byte übertragen
,eebf 30 3f      < bmi ef00       Stop-Bit (N=1): Sonderbehandlung
```

; Übertragung des nächsten Bit

```
,eec1 46 b6      lsr  b6          RODATA (RS232-Bytepuffer) rechtsverschieben, um nächstes Bit in Carry zu holen
,eec3 a2 00      ldx #00          SPACE-Bitmuster als Vorbelegung laden
,eec5 90 01      <bcc eec8        0-Bit zu übertragen (C=0): SPACE-Bitmuster verwenden
,eec7 ca        < dex "ldx #ff"   MARK-Bitmuster laden (für gesetztes Bit)
,eec8 8a        >txa              Bitmuster (SPACE oder MARK) in Akku
,eec9 45 bd      eor  bd          Verknüpfung mit Paritätsbyte ROPRTY      } Neuberechnung
,eeeb 85 bd      sta  bd          Ergebnis als neues Paritätsbyte setzen    } der Parität
,eeed c6 b4      dec  b4          BITTS (Bitzähler für RS232) verringern
,eeef f0 06      <beq eed7       schon auf 0 heruntergezählt (Z=1): Byte-Ende behandeln
,eed1 8a        <txa              Bitmuster (SPACE oder MARK) in Akku
```

```

,eed2 29 04    and #04 %00000100  alle Bits bis auf b2 löschen
,eed4 85 b5    sta  b5             und als NXTBIT (nächstes zu übertragendes Bit) setzen
,eed6 60       rts                Rücksprung von Routine
-----

; Byte-Ende (Sonderbehandlung)

,eed7 a9 20    >lda #20 %00100000 gerade Parität ("odd parity") laden
,eed9 2c 94 02 bit 0294           Vergleich mit 6551-Befehlsregister für RS232
,eedc f0 14    beq eef2           Übereinstimmung (Z=1): "no parity"
,eede 30 1c    bmi eefc           b7 gesetzt (N=1): feste Parität
,eee0 70 14    bvs eef6           b6 gesetzt (V=1): "even parity"
,eee2 a5 bd    lda  bd            ROPRTY (Paritätsbyte) zwecks Test auslesen
,eee4 d0 01    bne eee7           ungerade Parität (Z=0): Bitmuster (SPACE oder MARK) nicht verringern
,eee6 ca       >dex              Bitmuster verringern (SPACE wird zu MARK)
,eee7 c6 b4    >dec  b4           BITTS (Bitzähler für RS232) verringern
,eee9 ad 93 02 lda 0293           M51CTR (6551-Kontrollregister) zwecks Test auslesen
,eeec 10 e3    bpl eed1           b7=0 (N=0): Bitmuster als nächstes zu übertragendes Bit merken
,eeee c6 b4    dec  b4           BITTS (Bitzähler für RS232) verringern
,eef0 d0 df    bne eed1           noch nicht auf 0 heruntergezählt (Z=0): Bitmuster als nächstes zu übertragendes Bit
,eef2 e6 b4    >inc  b4           BITTS (Bitzähler für RS232) erhöhen
,eef4 d0 f0    bne eee6 "jmp"     weiter in Bit-Bearbeitungsschleife
-----

; Behandlung für "even parity"

,eef6 a5 bd    >lda  bd            ROPRTY (Paritätsbyte) zwecks Test auslesen
,eef8 f0 ed    beq eee7           gerade Parität (Z=0): Bitmuster (SPACE oder MARK) nicht verringern
,eefa d0 ea    bne eee6 "jmp"     Bitmuster verringern
-----

; Behandlung für feste Parität

,eefc 70 e9    bvs eee7           feste Parität (V=1): Bitmuster (SPACE oder MARK) nicht verringern
,eefe 50 e6    bvc eee6 "jmp"     Bitmuster verringern
-----

; Behandlung für Stop-Bit

,ef00 e6 b4    inc  b4            BITTS (Bitzähler für RS232) erhöhen
,ef02 a2 ff    ldx #ff %11111111 Bitmuster für MARK laden
,ef04 d0 cb    <bne eed1 "jmp"     Bitmuster als nächstes zu sendendes Bit verwenden
-----

```

; RSTBGN: nächstes Byte übertragen, da BITTS = 0

```
,ef06 ad 94 02 lda 0294      6551-Befehlsregister für RS232 auslesen
,ef09 4a      lsr            b0 durch Rechtsverschiebung ins Carry holen
,ef0a 90 07    bcc ef13      b0 (DTR) war 0 (C=0): Sonderbehandlung für DTR HIGH überspringen
,ef0c 2c 01 dd bit dd01      CIA-Datenregister B testen
,ef0f 10 1d    bpl ef2e      DSR-Bit = 0 (N=0): Fehler "MISSING DSR" auslösen
,ef11 50 1e    bvc ef31      CTS-Bit = 0 (V=0): Fehler "MISSING CTS" auslösen
,ef13 a9 00    >lda #00      Initialisierungswert für ROPRTY (Paritätsbyte) und NXTBIT (nächstes Bit) laden
,ef15 85 bd    sta bd        in ROPRTY (Paritätsbyte) schreiben
,ef17 85 b5    sta b5        und in NXTBIT (nächstes Bit) schreiben
,ef19 ae 98 02 ldx 0298      BITNUM (Anzahl der noch zu übertragenden Bits) laden
,ef1c 86 b4    stx b4        und als RS232-Bitzähler (BITTS) setzen
,ef1e ac 9d 02 ldy 029d      RODBS (HB der Anfangsadresse des RS232-Ausgabepuffers) holen
,ef21 cc 9e 02 cpy 029e      Vergleich mit Endadresse des RS232-Ausgabepuffers
,ef24 f0 13    beq ef39      Übereinstimmung (Z=1): Sonderbehandlung für "Puffer leer"
,ef26 b1 f9    lda (f9),y    nächstes Byte aus RS232-Puffer über ROBUF-Zeiger für Ausgabepuffer holen
,ef28 85 b6    sta b6        und in RODATA (RS232-Bytepuffer) für Ausgabe schreiben
,ef2a ee 9d 02 inc 029d      RODBS (HB der Anfangsadresse des RS232-Ausgabepuffers) erhöhen
,ef2d 60      rts           Rücksprung von Routine
```

; RS232-Fehler auslösen

```
,ef2e a9 40    >lda #40 %01000000      Fehlernummer für "MISSING DSR" laden
,ef30 2c a9 10 >bit "lda #10" %00001000 Fehlernummer für "MISSING CTS" laden
,ef33 0d 97 02 ora 0297      Fehlerbit in RSSTAT (6551-Statusregister) einbinden
,ef36 8d 97 02 sta 0297      und in RSSTAT (6551-Statusregister) schreiben
,ef39 a9 01    >lda #01 %00000001      Wert für "Timer-A-Interrupt an" laden
,ef3b 8d 0d dd sta dd0d      und in ICR (Interrupt Control Register) schreiben
,ef3e 4d a1 02 eor 02a1      b0 (s. $ef39) in ENABL (RS232-Register) invertieren
,ef41 09 80    ora #80 %10000000      b7 setzen
,ef43 8d a1 02 sta 02a1      und als ENABL (RS232-Register) setzen
,ef46 8d 0d dd sta dd0d      ebenso in ICR schreiben
,ef49 60      rts           Rücksprung von Routine
```

; CALCBT-Hilfsroutine: Berechnung der Anzahl der über RS232 zu sendenden Bits pro Byte

```
,ef4a a2 09    ldx #09      Default-Wert (8 Bit Daten + 1 Bit Parität)
```



```

,ef4c a9 20    lda #20 %00100000 b5 testen
,ef4e 2c 93 02 bit 0293          Test von b5 in M51CTR (6551-Kontrollregister)
,ef51 f0 01    beq ef54          Wortlänge 6 Bit oder 8 Bit (Z=1): Dekrementierung überspringen
,ef53 ca      dex "ldx #08"      Wortlänge verringern, da sie 5 oder 7 Bit beträgt
,ef54 50 02    >bvc ef58        Wortlänge 7 oder 8 Bit (V=0): nicht um 2 dekrementieren, sondern übernehmen
,ef56 ca      dex              Wortlänge verringern (wird hier zu 7 oder 8)
,ef57 ca      dex              Wortlänge verringern (wird hier zu 6 oder 7)
,ef58 60      >rts             Rücksprung von Hilfsroutine, Ergebnis steht im X-Register
-----

```

; RSRCLR-Routine: Auswertung eines über RS232 einzulesenden Bit im NMI

```

,ef59 a6 a9    ldx a9          RINONE (Flag für Startbitprüfung bei RS232) testen
,ef5b d0 33    ↓ bne ef90 "rsrtrt" Startbitprüfung gewünscht (Z=0): auf Startbit warten
,ef5d c6 a8    dec a8          BITCI (RS232-Bitzähler) verringern
,ef5f f0 36    ↓ beq ef97      schon auf 0 heruntergezählt (Z=1): alle Bits empfangen, jetzt Byte in Puffer
,ef61 30 0d    bmi ef70       schon auf $ff heruntergezählt (N=1): Stop-Bit prüfen

```

; Berechnung der Eingabe-Parität

```

,ef63 a5 a7    lda a7          aktuelles Datenbit (INBIT) holen
,ef65 45 ab    eor ab          EOR-Verknüpfung mit RIPRTY (Eingabe-Parität)
,ef67 85 ab    sta ab          und Ergebnis als RIPRTY (Eingabe-Parität) setzen
,ef69 46 a7    lsr a7          aktuelles Datenbit (INBIT) in Carry holen
,ef6b 66 aa    ror aa          und in RIDATA (RS232-Eingabebyte) einbinden
,ef6d 60      >rts             Rücksprung von Routine
-----

```

; Sonderbehandlung "keine Parität" bei Übernahme eines Datenbyte in den RS232-Eingabepuffer

```

,ef6e c6 a8    dec a8          BITCI (RS232-Bitzähler) verringern
,ef70 a5 a7    >lda a7         INBIT (Stop-Bit-Zähler) zwecks Test auslesen
,ef72 f0 67    ↓ beq efdb     FRAME-Fehler oder BREAK (Z=1): RS232-Status gemäß Fehler setzen
,ef74 ad 93 02 lda 0293       M51CTR (6551-Kontrollregister) auslesen
,ef77 0a      asl             b7 (Flag für Anzahl der Stop-Bits) in Carry holen (s. $ef7a)
,ef78 a9 01    lda #01        1 als Mindestzahl für Stop-Bits laden
,ef7a 65 a8    adc a8          und zu BITCI (RS232-Bitzähler) addieren; bei C=1 (s. $ef77) wird 2 addiert
,ef7c d0 ef    bne ef6d       Ergebnis <> 0 (Z=0): Rücksprung über RTS

```

; RSRABL: RS232-Empfang initialisieren

```
,ef7e a9 90  →lda #90 %10010000 IRQ ENABLE, FLAG IRQ: RS232 RECEIVED DATA (also IRQ bei RS232-Empfang)
,ef80 8d 0d dd sta dd0d in ICR (Interrupt Control Register) von CIA 2 schreiben
,ef83 0d a1 02 ora 02a1 entsprechende Bits in ENABL (RS232-NMI-Flag) setzen
,ef86 8d a1 02 sta 02a1 und Ergebnis in ENABL (RS232-NMI-Flag) schreiben
,ef89 85 a9 sta a9 %10010000 in RINONE (Flag für Startbitprüfung bei RS232) schreiben
,ef8b a9 02 lda #02 %00000010 TIMER B UNDERFLOW IRQ (RS in) enable (Timer-B-Unterlauf aktiviert RS232-Eingabe-NMI)
,ef8d 4c 3b ef jmp ef3b "oenabl" Bit in ICR von CIA 2 schreiben
```

; RSRTRT: Startbitprüfung

```
,ef90 a5 a7 lda a7 INBIT (eingelesenes Bit) zwecks Test auslesen
,ef92 d0 ea bne ef7e "rsrabl" kein Startbit (Z=0): erneute Initialisierung
,ef94 4c d3 e4 jmp e4d3 Akku in RINONE (Flag für Startbitprüfung) schreiben und RIPRTY (Eingabe-Parität für RS232) mit 1 belegen
```

; alle Bits empfangen, Byte in RS232-Eingabepuffer schreiben

```
,ef97 ac 9b 02 ldy 029b RIDBE (HB der Endadresse des RS232-Eingabepuffers) auslesen
,ef9a c8 iny erhöhen
,ef9b cc 9c 02 cpy 029c Vergleich mit RIDBS (Offset auf Anfang des RS232-Eingabepuffers)
,ef9e f0 2a beq efca Übereinstimmung (Z=1): Status für "Eingabepuffer voll" setzen
,efa0 8c 9b 02 sty 029b ansonsten RIDBE (Offset auf Ende des RS232-Eingabepuffers) neu setzen
,efa3 88 dey Offset aber wieder verringern ($ef9a rückgängig machen), da er bei $efbl nötig ist
,efa4 a5 aa lda aa RIDATA (RS232-Eingabebyte) auslesen
```

; Datenbyte mit geringerer Wortlänge als 8 Datenbits nach rechts verschieben

```
,efa6 ae 98 02 ldx 0298 BITNUM (Anzahl der Datenbits über RS232) auslesen
,efa9 e0 09 →cpx #09 schon auf Ergebnislänge 9 (8 Bit Daten + 1 Bit Parität)?
,efab f0 04 beq efbl ja (Z=1): Rechtsverschiebungen nicht mehr erforderlich
,efad 4a lsr Rechtsverschiebung um 1 Bit
,efae e8 inx neue Anzahl der Datenbits setzen
,efaf d0 f8 bne efa9 "jmp" weiter in Verschiebeschleife, bis 9 Bit Länge erreicht sind
```

```
,efbl 91 f7 →sta (f7),y Datenbyte über RIBUF (Zeiger auf RS232-Eingabepuffer) schreiben
,efb3 a9 20 lda #20 %00100000 b5 (entscheidet über Paritätstest) testen
```

```
,efb5 2c 94 02 bit 0294      Test von b5 in M51CDR (6551-Befehlsregister)
,efb8 f0 b4  ↰_beq ef6e      keine Parität (Z=1): Sonderbehandlung
,efba 30 b1  ↰_bmi ef6d      Parität 0 oder 1 (N=1): Rücksprung über RTS
```

; Prüfung von gerader/ungerader Parität

```
,efbc a5 a7      lda  a7      INBIT (empfangenes Bit) holen
,efbe 45 ab      eor  ab      in RIPRTY (RS232-Eingabeparität) einbinden
,efc0 f0 03      ↰_beq ef65    Übereinstimmung von INBIT und RIPRTY (Z=1): Prüfung auf "odd parity" (ungerade)
```

; "even parity" liegt vor

```
,efc2 70 a9      ↰_bvs ef6d    "even parity" erfüllt (V=1): Rücksprung über RTS, kein Fehler
```

; "odd parity" liegt vor (bei Ausführung von \$efc5)

```
,efc4 2c 50 a6 ↰_bit" bvc ef6d    "odd parity" auch erfüllt (V=0): Rücksprung über RTS, kein Fehler
```

```
,efc7 a9 01      lda #01 %00000001    Bit für "Fehler bei Paritätstest" laden
,efc9 2c a9 04    "bit" lda #04 %00000100 Bit für "Überlauf des Eingabepufferspeichers" laden
,efcc 2c a9 80 ↰_bit" lda #80 %10000000    Bit für "Übertragung ist unterbrochen" laden
,efcf 2c a9 02 ↰_bit" lda #02 %00000010    Bit für "Fehler in der Bit-Folge" laden
,efd2 0d 97 02    ora 0297            Fehlerbit in RSSTAT (RS232-Statusregister) einbinden
,efd5 8d 97 02    sta 0297            und als neues RSSTAT (RS232-Statusregister) setzen
,efd8 4c 7e ef    jmp ef7e "rsrabl"    RS232-Empfang initialisieren
```

; FRAME-Fehler und BREAK differenzieren

```
,efdb a5 aa      lda  aa      RIDATA (Eingabebyte von RS232) auslesen
,efdd d0 f1      ↰_bne efd0    Bytewert vorliegend (Z=1): FRAME-Fehler (Fehler in der Bit-Folge)
,efdf f0 ec      ↰_beq efcd "jmp" ansonsten BREAK (Übertragung ist unterbrochen)
```

; CKORS: CKOUT (Umlenken der Ausgabe) auf RS232

```
,efel 85 9a      sta  9a      File als Ausgabekanal (DFLT0) setzen
,efe3 ad 94 02    lda 0294      M51CDR (6551-Kommandoregister) auslesen
,efe6 4a         lsr          b0 (zuständig für Handshake) durch Rechtsverschiebung in Carry holen
,efe7 90-29      bcc f012      3-Draht-Handshake (C=0): Ende der Routine (CLC und RTS)
,efe9 a9 02      lda #02 %00000010 bl als Testbit setzen
```

,efeb	2c 01 dd	bit dd01	bl (zuständig für RTS-Signal) in Datenport B von CIA 2 testen	
,efee	10 ld	bpl f00d	DSR (RS232-Datensatz bereit) nicht aktiviert (N=0): "missing dsr"-Status setzen	
,eff0	d0 20	bne f012	RTS-Signal vorhanden (Z=0): Ende der Routine (CLC und RTS)	
,eff2	ad a1 02	lda 02a1	ENABL (RS232-NMI-Flag) auslesen	
,eff5	29 02	and #02 %00000010	alle Bits bis auf bl löschen, also nur bl testen	
,eff7	d0 f9	bne eff2	Daten werden empfangen (Z=0): warten, bis RS232-Übertragung fertig ist	
,eff9	2c 01 dd	bit dd01	Datenport B von CIA 2 testen	
,effc	70 fb	bvs eff9	CTS low (V=1): weiter auf CTS warten	
,effe	ad 01 dd	lda dd01	Datenport B von CIA 2 auslesen	
,f001	09 02	ora #02 %00000010	bl (Daten werden empfangen) setzen	
,f003	8d 01 dd	sta dd01	und in Datenport B zurückschreiben	
,f006	2c 01 dd	bit dd01	Datenport B von CIA 2 testen	
,f009	70 07	bvs f012	CTS low (V=1): Ende der Routine (CLC und RTS)	
,f00b	30 f9	bmi f006	DSR low (V=1): weiter auf DSR (Datensatz bereit) warten	
,f00d	a9 40	lda #40 %01000000	b6 (zuständig für "missing dsr") gesetzt	} "missing dsr"-Status herstellen
,f00f	8d 97 02	sta 0297	und in RSSTAT (Statusregister für RS232-Betrieb) schreiben	
,f012	18	clc	Carry löschen (Flag für "kein Fehler")	
,f013	60	rts	Rücksprung von Routine	

; BSORS: BSOUT in RS232-Ausgabepuffer

,f014	20 28 f0	jsr f028	RS232-Datenausgabe über Interrupt starten, sofern noch nicht geschehen
,f017	ac 9e 02	ldy 029e	RODBE (Offset für Ausgabepuffer) auslesen
,f01a	c8	iny	erhöhen, um ihn auf nächste Position im Ausgabepuffer zu stellen
,f01b	cc 9d 02	cpy 029d	schon RODBS (Anfangsadresse des Ausgabepuffers im Speicher) erreicht?
,f01e	f0 f4	beq f014	ja (Z=1): Datenausgabe über Interrupt, bis Puffer für weiteres Byte frei ist
,f020	8c 9e 02	sty 029e	neuen RODBE (Offset für Ausgabepuffer) setzen
,f023	88	dey	Offset in Y jedoch wieder dekrementieren (\$f01a rückgängig machen)
,f024	a5 9e	lda 9e	auszugebendes Zeichen aus PTR1-Zeichenpuffer entnehmen
,f026	91 f9	sta (f9),y	und in RS232-Ausgabepuffer schreiben

; RS232-Datenausgabe über Interrupt starten, sofern noch nicht erfolgt

,f028	ad a1 02	lda 02a1	ENABL (RS232-NMI-Flag für CIA 2) holen
,f02b	4a	lsr	b0 (Timer A UNDERFLOW IRQ "RS out") zwecks Test in Carry holen
,f02c	b0 1e	bcs f04c	RS232-Ausgabe ist bereits eingeschaltet (C=1): Rücksprung
,f02e	a9 10	lda #10 %00010000	b7=0 (CLEAR IRQ FLAG), b4=1 (RS232 RECEIVED DATA IRQ FLAG)
,f030	8d 0e dd	sta dd0e	in CRA (Control Register A) von CIA 2 schreiben

,f033	ad 99 02	lda 0299	LB des Timer-Verzögerungswertes für Baud-Rate (BAUDOF) holen	} Verzögerungswert für aktuelle Baud-Rate in Timer A von CIA
,f036	8d 04 dd	sta dd04	und in LB von Timer A in CIA 2 schreiben	
,f039	ad 9a 02	lda 029a	HB des Timer-Verzögerungswertes für Baud-Rate (BAUDOF) holen	
,f03c	8d 05 dd	sta dd05	und in HB von Timer A in CIA 2 schreiben	
,f03f	a9 81	lda #81 %10000001	b7=1 (SET IRQ ENABLE), b0=1 (Timer A UNDERFLOW bei Senden)	
,f041	20 3b ef	jsr ef3b "oenabl"	Bits in ICR (Interrupt Control Register) und ENABLE (RS232-NMI-Flag) übernehmen	
,f044	20 06 ef	jsr ef06 "rstbgn"	nächstes Byte übertragen	
,f047	a9 11	lda #11 %00010001	b4=1 (RS232 RECEIVED DATA IRQ FLAG), b0=1 (Timer A UNDERFLOW bei Senden)	
,f049	8d 0e dd	sta dd0e	in CRA (Control Register A) von CIA 2 schreiben	
,f04c	60	↳rts	Rücksprung von Routine	

; CKIRS: CHKIN für RS232

,f04d	85 99	sta 99	Filenummer als DFLTIN (vorgebene Nummer des aktuellen Eingabegerätes) setzen
,f04f	ad 94 02	lda 0294	M51CDR (6551-Kommandoregister) auslesen
,f052	4a	lsr	b0 (zuständig für Handshake) durch Rechtsverschiebung in Carry holen
,f053	90-28	bcc f07d	3-Draht-Handshake (C=0): Sonderbehandlung

; Sonderbehandlung: X-Draht-Handshake

,f055	29 08	and #08 %00001000	alle Bits bis auf b3 (vorher b4, s. \$f052) löschen
,f057	f0-24	beq f07d	FULL DUPLEX (Z=1): Behandlung wie bei 3-Draht-Handshake
,f059	a9 02	lda #02 %00000010	b1 (RTS = Request To Send) als Testbit laden
,f05b	2c 01 dd	bit dd01	Test von Datenport B von CIA 2
,f05e	10-ad	bpl f00d	kein DSR (Data Set Ready)-Signal (N=0): "missing dsr"-Fehler in Statusbyte setzen
,f060	f0-22	beq f084	kein RTS (Request To Send)-Signal (Z=1): Rücksprung, vorher Carry löschen

; Warten auf Ende der RS232-Datenausgabe

,f062	ad a1 02	↳lda 02a1	ENABL (RS232-NMI-Flag) auslesen	
,f065	4a	lsr	b0 (Timer A UNDERFLOW IRQ bei RS232-Senden) ins Carry holen	
,f066	b0 fa	↳bcs f062	noch nicht fertig (C=1): weiter auf Ende der Datenausgabe warten	
,f068	ad 01 dd	lda dd01	Datenport B von CIA 2 auslesen	} Request To Send auf 0
,f06b	29 fd	and #fd %11111101	b1 (b1=0: RTS=Request To Send) löschen	
,f06d	8d 01 dd	sta dd01	und in Datenport B von CIA 2 zurückschreiben	} setzen
,f070	ad 01 dd	↳lda dd01	Datenport B von CIA 2 auslesen	} auf DTR warten
,f073	29 04	and #04 %00000100	alle Bits bis auf b2 (DTR = Data Terminal Ready) löschen, also b2 testen	
,f075	f0 f9	beq f070	noch kein DTR-Signal (Z=1): weiter auf DTR warten	
,f077	a9 90	↳lda #90 %10010000	b7=1 (SET IRQ ENABLE BIT), b4=1 (RS232 RECEIVED DATA IRQ FLAG)	
,f079	18	clc	Carry löschen (Flag für "kein Fehler")	
,f07a	4c 3b ef	jmp ef3b "oenabl"	Bit in ICR von CIA 2 schreiben und Ende	

; Sonderbehandlung: 3-Draht-Handshake

```

,f07d  ad a1 02 → lda 02a1      ENABL (RS232-NMI-Flag für CIA 2) auslesen
,f080  29 12      and #12 %00010010 nur b1 (RS232 INPUT NMI FLAG) und b4 (Timer B UNDERFLOW) testen
,f082  f0 f3      beq f077      b1=0 und b4=0 (Z=1): NMI freigeben, da RS232 fertig ist
,f084  18      → clc          Carry löschen (Flag für "kein Fehler")
,f085  60      rts          Rücksprung von Routine

```

; GETRS: GETIN von RS232-Eingabepuffer

```

,f086  ad 97 02  lda 0297      RSSTAT (RS232-Statusregister) holen
,f089  ac 9c 02  ldy 029c      Offset für RS232-Eingabepuffer (RIDBS) laden
,f08c  cc 9b 02  cpy 029b      schon mit Anfang des RS232-Eingabepuffers im Speicher identisch?
,f08f  f0 0b      beq f09c      ja (Z=1): Sonderbehandlung für GETIN bei leerem RS232-Eingabepuffer
,f091  29 f7      and #f7 %11110111 b3 (Fehler-Flag für "Puffer leer") in RSSTAT (s. $f086) löschen
,f093  8d 97 02  sta 0297      und in RSSTAT (RS232-Statusregister) schreiben
,f096  b1 f7      lda (f7),y    aktuelles Byte aus RS232-Eingabepuffer auslesen
,f098  ee 9c 02  inc 029c      RIDBS (Offset für RS232-Eingabepuffer) auslesen
,f09b  60      rts          Rücksprung von Routine

```

; Sonderbehandlung: GETIN bei leerem RS232-Eingabepuffer

```

,f09c  09 08      ora #08 %00001000 b3 (Fehler-Flag für "Puffer leer") in RSSTAT (s. $f086) setzen
,f09e  8d 97 02  sta 0297      und in RSSTAT (RS232-Statusregister) schreiben
,f0a1  a9 00      lda #00      Nullbyte als Rückgabewert laden
,f0a3  60      rts          Rücksprung von Routine

```

; RSP232-Hilfsroutine: Warten, bis RS232 fertig ist

```

,f0a4  48      pha          Akku-Inhalt bei Eintritt in Routine bis $f0bb auf Stapel merken
,f0a5  ad a1 02  lda 02a1      ENABL (RS232-NMI-Flag) auslesen
,f0a8  f0 11      beq f0bb      alle Bits gelöscht (Z=1): Rücksprung, da keine Übertragung aktiv
,f0aa  ad a1 02 → lda 02a1      ENABL (RS232-NMI-Flag) auslesen
,f0ad  29 03      and #03 %00000011 alle Bits bis auf b0 (Timer A UNDERFLOW IRQ bei RS232-Ausgabe) und b1 (Timer B
UNDERFLOW IRQ bei RS232-Eingabe) löschen, also nur b0 und b1 testen
,f0af  d0 f9      bne f0aa      b0 oder b1 gesetzt (Z=0): auf Ende von Timer A UNDERFLOW IRQ warten
,f0b1  a9 10      lda #10 %00010000 b4 (RS232 RECEIVED DATA IRQ FLAG) gesetzt
,f0b3  8d 0d dd  sta dd0d      und in ICR (Interrupt Control Register) von CIA 2 schreiben

```

```
,f0b6 a9 00 | lda #00 %00000000 RS232 "disabled" (nicht aktiviert) laden
,f0b8 8d a1 02 | sta 02a1           und in ENABL (RS232-NMI-Flag) schreiben
,f0bb 68      | >pla             bei $f0a4 gemerkten Akku-Inhalt wieder vom Stapel holen
,f0bc 60      | rts             Rücksprung von Routine
```

```
; Systemmeldungen (Steuer- und Fehlermeldungen) im ASCII-Format; b7 dient im jeweils letzten Byte als Endmarkierung

:f0bd 0d 49 2f 4f 20 45 52 52 4f 52 20 a3 [cr]i/o error # ($00)
:f0c9 0d 53 45 41 52 43 48 49 4e 47 a0 [cr]searching[space] ($0c)
:f0d4 46 4f 52 a0 for[space] ($17)
:f0d8 0d 50 52 45 53 53 20 50 4c 41 59 20 4f 4e 20 54 41 50 c5 [cr]press play on tape ($1b)
:f0eb 50 52 45 53 53 20 52 45 43 4f 52 44 20 26 20 50 4c 41 59 20 4f 4e 20 54 41 50 c5 press record & play on tape ($2e)
:f106 0d 4c 4f 41 44 49 4e c7 [cr]loading ($49)
:f10e 0d 53 41 56 49 4e 47 a0 [cr]saving ($51)
:f116 0d 56 45 52 49 46 59 49 4e c7 [cr]verifying ($59)
:f120 0d 46 4f 55 4e 44 a0 [cr]found[space] ($63)
:f127 0d 4f 4b 8d [cr]ok[cr] ($6a)
```

; Routine zur Ausgabe der Systemmeldung, deren Offset in Y übergeben wird

```
,f12b 24 9d bit 9d MSGFLG (Flag für Programm- oder Direktmodus) testen
,f12d 10 0d | bpl f13c Programm-Modus (N=0): keine Ausgabe der Systemmeldung
,f12f b9 bd f0 | >lda f0bd,y auszugebendes Byte aus ASCII-Tabelle entnehmen
,f132 08 | php CPU-Status (v.a. N-Flag wg. eventueller Text-Endmarkierung in b7) retten
,f133 29 7f | and #7f %01111111 b7 löschen (evtl. Endmarkierung ausmaskieren)
,f135 20 d2 ff | jsr ffd2 "bsout" Zeichen ausgeben
,f138 c8 | iny Offset auf nächstes Byte richten
,f139 28 | plp bei $f132 gemerkten CPU-Status (v.a. N-Flag) wiederherstellen
,f13a 10 f3 | bpl f12f keine Endmarkierung (N=0): weiter mit Textausgabe
,f13c 18 | >clc Carry löschen (Flag für "kein Fehler")
,f13d 60 | rts Rücksprung von Routine
```

; GETIN-Routine (hierher wird vom Kernal-Einsprung \$ffe4 verzweigt)

```
,f13e a5 99    lda  99          DFLTn (Standard-Eingabegerät) holen
,f140 d0 08    bne  f14a        keine Eingabe von Tastatur (Z=0): Sonderbehandlung überspringen
```

; GETKB: GETIN von Tastatur

```
,f142 a5 c6    lda  c6          NDX (Anzahl der Zeichen im Tastaturpuffer) holen
,f144 f0 0f    beq  f155        Tastaturpuffer leer (Z=1): $00 (im Akku befindlich) zurückgeben
,f146 78       sei              Interrupt verhindern, damit Tastaturabfrage im IRQ nicht in die Quere kommt
,f147 4c b4 e5  jmp  e5b4 "nxtkey" nächstes Zeichen aus Tastaturpuffer in Akku holen
```

```
-----
,f14a c9 02    >cmp  #02        Vergleich von DFLTn (s. $f13e) mit Gerätenummer für RS232
,f14c d0 18    bne  f166        keine Übereinstimmung (Z=0): Prüfung auf weitere Geräteummern durch Einsprung in
                                BASIN-Routine
```

; Sonderbehandlung: GETIN von RS232

```
,f14e 84 97    sty  97          Y-Register in Zwischenspeicher $97 retten (Y wird bei $f150 zerstört)
,f150 20 86 f0  jsr  f086 "getrs" GETIN von RS232 ausführen
,f153 a4 97    ldy  97          Y-Register wiederherstellen (s. $f14e)
,f155 18       >clc            Carry löschen (Flag für "kein I/O-Fehler")
,f156 60       rts             Rücksprung von Routine
-----
```

; BASIN-Routine (hierher wird vom Kernal-Einsprung bei \$ffcf verzweigt)

```
,f157 a5 99    lda  99          DFLTn (Standard-Eingabegerät) holen
,f159 d0 0b    bne  f166        keine Eingabe von Tastatur (Z=0): Sonderbehandlung überspringen
```

; BINKB: BASIN von Tastatur

```
,f15b a5 d3    lda  d3          aktuelle Cursorspalte holen
,f15d 85 ca    sta  ca          und als Cursorspalte für Eingabe setzen
,f15f a5 d6    lda  d6          aktuelle Cursorzeile holen
,f161 85 c9    sta  c9          und als Cursorzeile für Eingabe setzen
,f163 4c 32 e6  jmp  e632 "scrget" Zeichen von aktueller Bildschirmposition holen
```


; BASIN bzw. GETIN von weiteren Geräten (z.B. IEC-Bus)

```
,f166 c9 03 → cmp #03      Vergleich der aktuellen Gerätenummer mit Bildschirm-Gerätenummer
,f168 d0 09 → bne f173      keine Übereinstimmung (Z=0): Sonderbehandlung überspringen
```

; BGISC: BASIN bzw. GETIN vom Bildschirm

```
,f16a 85 d0  sta  d0      CRSW (Flag für INPUT/GET von Tastatur) mit 3 belegen (Flag setzen)
,f16c a5 d5  lda  d5      LNMX (physikalische Bildschirmzeilenlänge) auslesen
,f16e 85 c8  sta  c8      und als INDX (Zeiger auf Ende der logischen Eingabezeile) setzen
,f170 4c 32 e6 jmp e632 "scrget" Zeichen von aktueller Bildschirmposition holen
```

; BASIN bzw. GETIN von weiteren Geräten (z.B. IEC-Bus)

```
,f173 b0 38 → bcs flad      BASIN bzw. GETIN von Geräteadresse > 3 (C=1): Sonderbehandlung für IEC-Bus
```

; diese Stelle kann nur bei BASIN mit \$02 im Akku durchlaufen werden (s. \$f14a bei GETIN-Behandlung)!

```
,f175 c9 02  cmp #02      RS232-BASIN?
,f177 f0 3f → beq flb8 "bsirs" ja (Z=1): Sonderbehandlung für BASIN von RS232 anspringen
```

; BASIN bzw. GETIN von Gerät #1 (Datasette)

```
,f179 86 97  stx  97      X-Register bis $f18f in Zwischenspeicher $97 retten
,f17b 20 99 f1 jsr f199 "jtget" nächstes Byte aus Kassettenpuffer in Akkumulator holen
,f17e b0 16 → bcs f196      I/O-Fehler (C=1): X-Register wiederherstellen und RTS bei gesetztem Carry
,f180 48      pha        eingelestes Byte bis $f191 merken
,f181 20 99 f1 jsr f199 "jtget" nächstes Byte aus Kassettenpuffer in Akkumulator holen
,f184 b0 0d → bcs f193      I/O-Fehler (C=1): X-Register und Akku wiederherstellen und RTS bei gesetztem Carry
,f186 d0 05 → bne f18d      kein Nullbyte (Z=0): BUPNT (Zeiger für Kassettenpuffer) verringern, Byte zurückgeben

,f188 a9 40  lda #40 %01000000 b6 (End Of File) gesetzt, da Puffer-Endmarkierung $00 erreicht
,f18a 20 1c fe jsr felc "erstat" Bit in Statusbyte des Kernals übernehmen

,f18d c6 a6 → dec  a6      BUPNT (Zeiger für Kassettenpuffer) verringern, da 1 Byte zuviel ausgelesen wurde
                        (s. $f181)
,f18f a6 97  ldx  97      bei $f179 gemerkten Inhalt des X-Registers wiederherstellen
,f191 68      pla        bei $f180 gemerktes Byte wieder in Akku holen
,f192 60      rts        Rücksprung von Routine
```

; Fehlerbehandlung bei Eingabe von Datensette

,fl93	aa	→tax	Fehlernummer bis \$fl95 zwischenspeichern
,fl94	68	pla	eingelassenen Bytewert (bei \$fl80 gemerkt) vom Stapel löschen
,fl95	8a	txa	Fehlernummer (s. \$fl93) wieder in Akku holen
,fl96	a6 97	→ldx 97	bei \$fl79 gemerkten Inhalt des X-Registers wiederherstellen
,fl98	60	rts	Rücksprung von Routine (X=unverändert gegenüber Routineneinsprung, A=Fehlernummer, C=1 als Flag für I/O-Fehler)

; JTGET: nächstes Byte aus Kassettenpuffer in Akkumulator holen

,fl99	20 0d f8	→jsr f80d "tbful"	Test, ob der Kassettenpuffer voll ist; gleichzeitig Zeiger für Kassettenpuffer erhöhen; Offset BUFPNT in Y-Register holen
,fl9c	d0 0b	→bne fla9	Kassettenpuffer voll (Z=0): Byte auslesen, Ende
,fl9e	20 41 f8	→jsr f841 "rblk"	nächsten Block von Kassette in Puffer einlesen
,fla1	b0 11	→bcs flb4	I/O-Fehler (C=1): RTS bei gesetztem Carry
,fla3	a9 00	→lda #00	Initialisierungswert für BUFPNT (Zeiger für Kassettenpuffer) laden
,fla5	85 a6	→sta a6	und in BUFPNT (Zeiger auf aktuelles Byte in Kassettenpuffer) schreiben
,fla7	f0 f0	→beq fl99 "jmp jtget"	zurück an Anfang der JTGET-Routine, da Block ordnungsgemäß eingelesen wurde

,fla9	b1 b2	→lda (b2),y	Byte mittels TAPE1 (Zeiger auf Kassettenpuffer) und BUFPNT-Offset in Y holen
,flab	18	→clc	Carry löschen (Flag für "kein Fehler")
,flac	60	→rts	Rücksprung von Routine

; Sonderbehandlung: BASIN bzw. GETIN von IEC-Bus angefordert

,flad	a5 90	→lda 90	Statusbyte des Kernals auslesen
,flaf	f0 04	→beq flb5	kein Fehlerbit gesetzt (Z=1): zu IEC-Eingabe-Routine springen
,flb1	a9 0d	→lda #0d	ASCII-Code von [cr] laden, da Eingabe-Ende erreicht
,flb3	18	→clc	Carry löschen (Flag für "kein Fehler")
,flb4	60	→rts	Rücksprung von Routine

,flb5	4c 13 ee	→jmp eel3 "acptr"	Byte vom seriellen Bus (IEC-Bus) in Akku holen
-------	----------	-------------------	--

; BSIRS: BASIN von RS232

,flb8	20 4e f1	→jsr fl4e "getrs"	GETIN von RS232 ausführen
,flbb	b0 f7	→bcs flb4	I/O-Fehler (C=1): RTS bei gesetztem Carry

```
,flbd c9 00      cmp #00      Nullbyte als Endmarkierung?
,flbf d0 f2      bne flb3      nein (Z=0): Byte zurückgeben

; Sonderbehandlung: $00 von IEC-Bus eingeholt

,flcl ad 97 02   lda 0297      RS232-Statusregister (RSSTAT) holen
,flc4 29 60      and #60 %01100000 b5 (nicht belegt) und b6 (DSR = Data Set Ready) testen (alle anderen Bits löschen)
,flc6 d0-e9      bne flb1      b5 oder b6 gesetzt (Z=0): [cr] zurückgeben
,flc8 f0 ee      beq flb8 "bsirs" zurück an Anfang der BSIRS-Routine, da Datensatz eingelesen
```

; BSOUT-Routine (hierher wird normalerweise vom Kernall-Einsprung bei \$ffd2 verzweigt)

```
,flca 48      pha      auszugebendes Zeichen auf den Stapel legen
,flcb a5 9a    lda 9a    DFLTO (vorgegebenes Ausgabegerät) holen
,flcd c9 03    cmp #03   Ausgabe auf Bildschirm (Gerät #3)?
,flcf d0 04    bne fld5   nein (Z=0): Sonderbehandlung überspringen
```

; BSOUT-Sonderbehandlung: Ausgabe auf Bildschirm

```
,fldl 68      pla      bei $flca gemerktes Ausgabezeichen wieder in Akku holen
,fld2 4c 16 e7 jmp e716 "scout" und auf den Bildschirm ausgeben
```

```
,fld5 90 04    >bcc fldb   kleinere Geräteadresse als 3, also 0 (Tastatur), 1 (Datasette) oder 2 (RS232)
                        (C=0): Sonderbehandlung für IEC-Bus (Geräteadresse >= 4) überspringen
```

; BSOUT-Sonderbehandlung: Ausgabe auf IEC-Bus

```
,fld7 68      pla      bei $flca gemerktes Ausgabezeichen wieder in Akku holen
,fld8 4c dd ed jmp eddd "iecout" und auf den IEC-Bus ausgeben
```

; BSOUT-Behandlung für Tastatur (A=0), Datasette (A=1) oder RS232 (A=2)

```
,fldb 4a      >lsr      b0 ins Carry schieben (danach: C=0 bei Tastatur oder RS232; C=1 bei Datasette)
,fldc 68      pla      bei $flca gemerktes Ausgabezeichen wieder in Akku holen
,f added 85 9e sta 9e    und in 1-Byte-Puffer für Datasette (PTR1) schreiben
,f added 8a      txa      X-Register in Akku
,f added 48      pha      und dann auf den Stapel      X- und Y-Register
,f added 98      ty a     Y-Register in Akku          auf Stapel merken
,f added 48      pha      und dann auf den Stapel
,f added 90-f23 bcc f208   keine Ausgabe auf Datasette (s. $fldb) (C=0): Sonderbehandlung überspringen
```

; BSOUT-Sonderbehandlung: Ausgabe auf Datasette

,fle5	20	0d f8	jsr f80d "tbful"	Test, ob der Kassettenpuffer voll ist; gleichzeitig Zeiger für Kassettenpuffer erhöhen; Offset BUPNT in Y-Register holen
,fle8	d0	0e	bne flf8	Kassettenpuffer noch nicht voll (Z=0): Zeichen in Puffer schreiben
,flea	20	64 f8	jsr f864 "wblk"	Datenblock (= Kassettenpuffer-Inhalt) auf Kassette schreiben
,fled	b0	0e	bcs flfd	I/O-Fehler (C=1): Byte nicht in Kassettenpuffer schreiben, sondern Schlußbehandlung
,flef	a9	02	lda #02	Headermarke für Folge-Datenblock laden
,flf1	a0	00	ldy #00	Offset auf erstes Byte in Kassettenpuffer stellen
,flf3	91	b2	sta (b2),y	Headermarke an erste Position im Puffer setzen
,flf5	c8		iny "ldy #01"	Offset auf zweites Byte in Kassettenpuffer stellen
,flf6	84	a6	sty a6	und BUPNT (Offset für Kassettenpuffer) auf diese Position stellen, um Headermarke zu übergehen

; neues Ausgabe-Zeichen in Kassettenpuffer schreiben

,flf8	a5	9e	lda 9e	auszugebendes Zeichen laden	
,flfa	91	b2	sta (b2),y	und an aktuelle Position im Kassettenpuffer schreiben	
,flfc	18		clc	Carry löschen (Flag für "kein Fehler", da Schreiben in Kassettenpuffer keinen Fehler ausgelöst hat)	
,flfd	68		pla	bei \$fle1/\$fle2 gemerkten Y-Registerinhalt holen	} Inhalt von X- und Y-Register (bei Eintritt in Routine) wiederherstellen
,flfe	a8		tay	und ins Y-Register bringen	
,flff	68		pla	bei \$fldf/\$fle0 gemerkten X-Registerinhalt holen	
,f200	aa		tax	und ins X-Register bringen	
,f201	a5	9e	lda 9e	auszugebendes Zeichen wieder in Akku laden	
,f203	90	02	bcc f207	vorher kein I/O-Fehler (C=0): RTS bei gelöschtem Carry mit Ausgabe-Zeichen im Akku	
,f205	a9	00	lda #00	Fehlernummer für "BREAK: Abbruch durch STOP" laden	
,f207	60		rts	Rücksprung von Routine	

; BSOUT auf weitere Geräte (Tastatur oder RS232)

Da ein Ausgabekanal auf Tastatur von anderen Kernall-Routinen nicht aktiviert wird, erfolgt hier logischerweise nur RS232-Ausgabe.

,f208	20	17-f0	jsr f017	in BSORS-Routine einsteigen (das Starten der RS232-Übertragung im Interrupt wird übersprungen)
,f20b	4c	fc fl	jmp flfc	Carry löschen (Flag für "kein Fehler"), Register A/X/Y wiederherstellen

; CHKIN-Routine (hierher wird normalerweise vom Kernal-Einsprung bei \$ffc6 gesprungen)

```
,f20e 20 0f f3 jsr f30f "lookup" logische Filenummer (X-Register) in File-Tabelle suchen
,f211 f0 03     beq f216          File ist vorhanden (Z=1): keinen I/O ERROR #3 (FILE NOT OPEN) auslösen
,f213 4c 01 f7     jmp f701 "ioerr3" I/O ERROR #3 ("file not open") auslösen, da Eingabe von nicht vorhandenem File
                                   gewünscht wurde
-----
,f216 20 1f f3     jsr f31f "getfls" logische Filenummer, Geräteadresse und Sekundäradresse aus Filetabelle des Kernal in
                                   Hilfszeiger $b8-$ba holen
,f219 a5 ba       lda  ba          aktuelle Gerätenummer (FA) holen
,f21b f0 16       beq f233 "sdfltn" Eingabe von Gerät #0 (Tastatur) gewünscht (Z=1): Tastatur als Eingabegerät setzen
,f21d c9 03       cmp  #03         Vergleich der aktuellen Gerätenummer (FA) mit Geräteadresse für Bildschirm (#3)
,f21f f0 12       beq f233 "sdfltn" Übereinstimmung (Z=1): Bildschirm als Eingabegerät setzen
,f221 b0 14       bcs f237 "ckiser" Eingabe von IEC-Bus-Gerät (C=1): Sonderbehandlung "ckiser" anspringen
,f223 c9 02       cmp  #02         Eingabe von RS232 gewünscht?
,f225 d0 03       bne f22a "ckitap" nein (Z=0): Eingabe von Kassette (Gerät #1)
,f227 4c 4d f0     jmp f04d "ckirs" RS232 zum aktuellen Eingabegerät erklären
-----
```

; CKITAP: Datasette zum aktuellen Eingabegerät erklären

```
,f22a a6 b9       >ldx  b9          aktuelle Sekundäradresse (SA) holen
,f22c e0 60       cpx  #60 %01100000 Vergleich mit Eingabe-Sekundäradresse
,f22e f0 03       beq f233 "sdfltn" Übereinstimmung (Z=1): Gerätenummer aus Akku in DFLTN schreiben
,f230 4c 0a f7     jmp f70a "ioerr6" I/O ERROR #6 ("not input file") auslösen
-----
```

; SDFLTN: Geräteadresse aus Akku in DFLTN (Hilfsspeicher für Standard-Eingabegerät) schreiben;
dann Rücksprung von CHKIN-Routine

```
,f233 85 99       >sta  99          Akku in DFLTN schreiben
,f235 18         clc              Carry löschen (Flag für "kein Fehler", da Eingabe erfolgreich umgelenkt wurde)
,f236 60         rts              Rücksprung von Routine
-----
```

; CKISER: IEC-Bus-Gerät zum Eingabegerät erklären

```
,f237 aa         >tax              Gerätenummer zunächst in X-Register merken (bis $f248)
,f238 20 09 ed     jsr ed09 "talk" TALK-Routine aufrufen (nicht über Kernal-Einsprung)
,f23b a5 b9       lda  b9          aktuelle Sekundäradresse (SA) holen
,f23d 10 06       bpl f245         keine Geräteadresse > $80 (ab $80: "AUTO LINE FEED"-Aufforderung) (N=0): weiter
```

; Sonderbehandlung: Eingabe von Gerät mit Adresse >= \$80 (ab \$80: "AUTO LINEFEED"-Aufforderung)

,f23f	20 cc	ed	jsr edcc	ATN high senden (Flag für Sekundäradresse >= \$80)
,f242	4c 48	f2	jmp f248	Senden der Sekundäradresse überspringen

,f245	20 c7	ed	jsr edc7 "tksa"	TKSA-Routine (nicht über Kernal-Einsprung) aufrufen; sendet Sekundäradresse
,f248	8a		txa	bei \$f237 gemerkte Gerätenummer wieder in Akku holen
,f249	24 90		bit 90	Test des Statusbyte
,f24b	10 e6		bpl f233 "sdfltn"	DEVICE-NOT-PRESENT-Bit gelöscht (N=0): Akku als Eingabegerätenummer setzen
,f24d	4c 07	f7	jmp f707 "ioerr5"	I/O ERROR #5 ("device not present") auslösen

; CKOUT-Routine (hierher wird normalerweise vom Kernal-Einsprung bei \$ffc9 gesprungen)

,f250	20 0f	f3	jsr f30f "lookup"	logische Filenummer (X-Register) in File-Tabelle suchen
,f253	f0 03		beq f258	File ist vorhanden (Z=1): keinen I/O ERROR #3 (FILE NOT OPEN) auslösen
,f255	4c 01	f7	jmp f701 "ioerr3"	I/O ERROR #3 ("file not open") auslösen, da Ausgabe auf nicht vorhandenes File gewünscht wurde

,f258	20 1f	f3	jsr f31f "getfls"	logische Filenummer, Geräteadresse und Sekundäradresse aus Filetabelle des Kernal in Hilfszeiger \$b8-\$ba holen
,f25b	a5 ba		lda ba	aktuelle Geräteadresse (FA) holen
,f25d	d0 03		bne f262	andere Geräteadresse als Tastatur (Z=0): kein I/O ERROR #7 (NOT OUTPUT FILE)
,f25f	4c 0d	f7	jmp f70d "ioerr7"	I/O ERROR #7 ("not output file") auslösen, da Ausgabe von Tastatur erwünscht

,f262	c9 03		cmp #03	Vergleich der aktuellen Gerätenummer mit der Geräteadresse des Bildschirms
,f264	f0 0f		beq f275 "sdflto"	Übereinstimmung (Z=1): Bildschirm als Ausgabegerät setzen
,f266	b0 11		bcs f279 "ckoser"	Gerätenummer für IEC-Bus (C=1): Sonderbehandlung für IEC-Bus
,f268	c9 02		cmp #02	Vergleich der aktuellen Gerätenummer mit der Geräteadresse von RS232
,f26a	d0 03		bne f26f "ckotap"	keine Übereinstimmung (Z=0): CKOUT auf Datasette (Gerät #1)

; CHKOUT auf RS232 (Gerät #2)

,f26c	4c	el	ef	jmp efel "ckors"	Hilfsroutine für CKOUT auf RS232 aufrufen
-------	----	----	----	------------------	---

; CHKOUT auf Datasette (Gerät #1); Tastatur (Gerät #0) wurde schon vorher abgefangen (s. \$f25b-\$f25f)

,f26f	a6 b9		ldx b9	aktuelle Sekundäradresse (SA) holen
,f271	e0 60		cpx #60 %01100000	Vergleich mit Eingabe-Sekundäradresse
,f273	f0 ea		beq f25f	Übereinstimmung (Z=1): I/O ERROR #7 (NOT OUTPUT FILE) auslösen

; SDFLT0: Gerätenummer aus Akku in DFLT0 (Hilfsspeicher für Ausgabe-Gerätenummer) schreiben;
danach erfolgt Rücksprung aus CHKOUT.

,f275	85	9a	→sta	9a	Geräteadresse in DFLT0 schreiben
,f277	18		clc		Carry löschen (Flag für "kein Fehler", da Ausgabe erfolgreich umgelenkt wurde)
,f278	60		rts		Rücksprung von Routine

; CKOSER: CHKOUT auf IEC-Bus (serieller Bus)

,f279	aa		→tax		Geräteadresse bis \$f289 in X-Register retten
,f27a	20	0c ed	jsr ed0c	"listen"	LISTEN-Routine (nicht über Kernal-Einsprung) aufrufen
,f27d	a5	b9	lda	b9	aktuelle Sekundäradresse (SA) laden
,f27f	10	05	→bpl	f286	b7 in Sekundäradresse gelöscht (N=0): keine Sonderbehandlung
,f281	20	be ed	jsr edbe		ATN low senden, um Sekundäradresse > \$7f mitzuteilen
,f284	d0	03	→bne	f289 "jmp"	Senden der Sekundäradresse überspringen

,f286	20	b9 ed	→jsr	edb9 "second"	SECOND-Routine (nicht über Kernal-Einsprung) aufrufen; sendet Sekundäradresse
,f289	8a		→txa		bei \$f279 gemerkte Geräteadresse in Akku holen
,f28a	24	90	bit	90	Statusbyte des Kernal testen
,f28c	10	e7	→bpl	f275 "sdfldto"	Gerät war verfügbar (N=0): Gerät als Ausgabegerät setzen und Rücksprung
,f28e	4c	07 f7	jmp	f707 "ioerr5"	I/O ERROR #5 ("device not present") auslösen

; CLOSE-Routine (hierher wird normalerweise vom Kernal-Einsprung bei \$ffc3 verzweigt)

,f291	20	14 f3	jsr	f314 "jltlk"	logische Filenummer (X) in Filetabelle suchen
,f294	f0	02	→beq	f298	gefunden (Z=1): CLOSE ausführen, da File noch existiert
,f296	18		clc		Carry löschen (kein Fehler, da nicht offenes File als geschlossen gilt)
,f297	60		rts		Rücksprung von Routine

,f298	20	1f f3	→jsr	f31f "getfls"	logische Filenummer, Geräteadresse und Sekundäradresse aus Filetabelle des Kernal in Hilfszeiger \$b8-\$ba holen
,f29b	8a		txa		Zeiger in Filetabelle über Akku
,f29c	48		pha		auf den Stapel legen
,f29d	a5	ba	lda	ba	aktuelle Gerätenummer (FA) holen
,f29f	f0-50		→beq	f2f1 "delfle"	Tastatur-File schließen (Z=1): Eintrag aus Filetabelle löschen
,f2a1	c9	03	cmp	#03	Vergleich der aktuellen Gerätenummer mit der Nummer des Bildschirms
,f2a3	f0-4c		→beq	f2f1 "delfle"	Übereinstimmung (Z=1): Eintrag aus Filetabelle entfernen
,f2a5	b0-47		→bcs	f2ee	File von IEC-Bus schließen (C=1): IEC-File schließen und Fileeintrag löschen
,f2a7	c9	02	cmp	#02	Vergleich der aktuellen Gerätenummer mit der Nummer von RS232
,f2a9	d0	ld	→bne	f2c8	keine Übereinstimmung (Z=0): Datasetten-File schließen und Fileeintrag löschen

; RS232-File schließen und Fileeintrag löschen

,f2ab	68		pla	bei \$f29c gemerkten Zeiger innerhalb der Filetabelle in Akku holen
,f2ac	20	f2	jsr f2f2	in DELFLE-Routine einsteigen, damit Eintrag aus Filetabelle entfernt wird
,f2af	20	83	f4 jsr f483 "iciars"	CIA-Register nach RS232-Betrieb initialisieren
,f2b2	20	27	fe jsr fe27	in MEMTOP-Routine so einsteigen, daß oberste Basic-RAM-Adresse nach X/Y geholt wird
,f2b5	a5	f8	lda f8	HB der Adresse des RS232-Eingabepuffers holen
,f2b7	f0	01	beq f2ba	kein Eingabepuffer angelegt (Z=1): keine Erhöhung des HB der obersten Basic-Adresse
,f2b9	c8		iny	HB (s. \$f2b2) der obersten Adresse des für Basic verfügbaren RAM erhöhen, um
				Eingabepuffer (\$0100 Byte groß!) wieder für Basic-Zwecke freizugeben
,f2ba	a5	fa	→lda fa	HB der Adresse des RS232-Ausgabepuffers holen
,f2bc	f0	01	beq f2bf	kein Ausgabepuffer angelegt (Z=1): keine Erhöhung des HB der obersten Basic-Adresse
,f2be	c8		iny	HB (s. \$f2b2) der obersten Adresse des für Basic verfügbaren RAM erhöhen, um
				Ausgabepuffer (\$0100 Byte groß!) wieder für Basic-Zwecke freizugeben
,f2bf	a9	00	→lda #00	Flag für "kein Puffer angelegt" laden
,f2c1	85	f8	sta f8	in HB des Zeigers auf die Adresse des RS232-Eingabepuffers schreiben
,f2c3	85	fa	sta fa	in HB des Zeigers auf die Adresse des RS232-Ausgabepuffers schreiben
,f2c5	4c	7d	f4 jmp f47d	in MEMTOP-Routine so einsteigen, daß Obergrenze für Basic aus X/Y entnommen wird

; Datasetten-File schließen und Fileeintrag löschen

,f2c8	a5	b9	→lda b9	aktuelle Sekundäradresse (SA) holen
,f2ca	29	0f	and #0f %00001111	oberes Nibble (b4-b7) löschen
,f2cc	f0	23	beq f2f1 "delfle"	Sekundäradresse 0 (oder \$60, s. \$f2ca!) (Z=1): Fileeintrag unbesehen löschen
,f2ce	20	d0	f7 jsr f7d0 "getbfa"	Anfangsadresse des Kassettenpuffers nach X/Y holen
,f2d1	a9	00	lda #00	rechtsverschobenen Wert der Geräteadresse 3 (Bildschirm) laden
,f2d3	38		sec	b0 der ursprünglichen Geräteadresse 3 "laden"
,f2d4	20	dd	f1 jsr fldd	BSOUT-Behandlung aufrufen (in diesem Fall für Bildschirmausgabe)
,f2d7	20	64	f8 jsr f864 "wblk"	Datenblock (= Kassettenpuffer-Inhalt) auf Kassette schreiben
,f2da	90	04	bcc f2e0	kein I/O-Fehler (C=0): Fehlerbehandlung überspringen
,f2dc	68		pla	bei \$f29c gemerkten Zeiger innerhalb der Filetabelle vom Stapel entfernen
,f2dd	a9	00	lda #00	Fehlernummer für "BREAK: Abbruch durch STOP" laden
,f2df	60		rts	Rücksprung von Routine

; Fortsetzung der Datasetten-Behandlung für CLOSE

,f2e0	a5	b9	→lda b9	aktuelle Sekundäradresse (SA) holen
,f2e2	c9	62	cmp #62 %01100010	Sekundäradresse für "EOT (End Of Tape) schreiben"?
,f2e4	d0	0b	bne f2f1 "delfle"	nein (Z=0): Fileeintrag aus Filetabelle löschen
,f2e6	a9	05	lda #05 %00000101	Steuerbyte für Datenblock-Header "End Of Tape" laden

,f2e8	20 6a f7	jsr f76a "wblk"	nächsten Block (= Kassettenpuffer-Inhalt) auf Kassette schreiben
,f2eb	4c f1 f2	jmp f2f1 "delfle"	Fileeintrag aus Filetabelle löschen

; Sonderbehandlung: CLOSE für IEC-Bus			
,f2ee	20 42 f6	jsr f642 "ieccls"	File auf IEC-Bus schließen
; DELFLE-Einsprung: aktuellen Fileeintrag aus Filetabelle des Kernals entfernen			
,f2f1	68	→ pla	bei \$f29c gemerkten Zeiger innerhalb der Filetabelle holen
,f2f2	aa	tax	und in Offset-Register X bringen
,f2f3	c6 98	dec 98	Anzahl der offenen Dateien (LDTND) verringern
,f2f5	e4 98	cpx 98	handelt es sich beim zu löschenden Eintrag um den letzten Fileeintrag?
,f2f7	f0 14	beq f30d	ja (Z=1): Rücksprung, Dekrementieren von LDTND genügt zum Löschen
,f2f9	a4 98	ldy 98	Nummer des bei \$f2f3 ausgesonderten Fileeintrags in Offset-Register Y holen, damit er wieder wiederhergestellt werden kann
,f2fb	b9 59 02	lda 0259,y	logische Filenummer des letzten offenen Fileeintrags
,f2fe	9d 59 02	sta 0259,x	an Position des zu löschenden Eintrags schreiben
,f301	b9 63 02	lda 0263,y	Gerätenummer des letzten offenen Fileeintrags
,f304	9d 63 02	sta 0263,x	an Position des zu löschenden Eintrags schreiben
,f307	b9 6d 02	lda 026d,y	Sekundäradresse des letzten offenen Fileeintrags
,f30a	9d 6d 02	sta 026d,x	an Position des zu löschenden Eintrags schreiben
,f30d	18	→ clc	Carry löschen (Flag für "kein I/O-Fehler")
,f30e	60	rts	Rücksprung von Routine

; LOOKUP-Hilfsroutine: File (logische Filenummer in X zu übergeben) in Filetabelle suchen
Im X-Register wird der Offset in der Tabelle zurückgegeben.
Ist nach "jsr lookup" Z=0, wurde das File nicht gefunden.

,f30f	a9 00	lda #00	Initialisierungswert für "kein I/O-Fehler" laden	} Statusbyte
,f311	85 90	sta 90	und in Statusbyte des Kernals schreiben	
,f313	8a	txa	an LOOKUP übergebene Filenummer in Akku holen	} löschen

; JLTLK-Hilfsroutine: Position eines Files (Nummer im Akku) in Filetabelle des Kernals ermitteln

,f314	a6 98	ldx	98	Anzahl der offenen Dateien (LDTND) als Initialisierungswert für Dekrementier-Suchzähler laden
,f316	ca	→dex		Suchzähler verringern
,f317	30 15	bmi	f32e	schon auf \$ff heruntergezählt (N=1): File nicht gefunden, RTS bei N=1 und Z=0 (da X=\$ff das Z-Flag löscht!)

,f319	dd 59 02	cmp 0259,x	Vergleich der zu suchenden Filenummer mit Filenummer in LAT (Kernal-Tabelle für Filenummern)
,f31c	d0 f8	bne f316	keine Übereinstimmung (Z=0): Suche fortsetzen
,f31e	60	rts	Rücksprung, da Eintrag gefunden; X=Offset in Tabellen LAT, FAT und SAT

; GETFLS-Hilfsroutine: Fileparameter (logische Filenummer, Geräteadresse, Sekundäradresse) zum File mit dem in X enthaltenen Offset innerhalb der Filetabelle ermitteln

,f31f	bd 59 02	lda 0259,x	logische Filenummer aus Kernal-Tabelle LAT holen	} logische Filenummer (LA) ermitteln
,f322	85 b8	sta b8	und in Hilfsspeicher LA schreiben	
,f324	bd 63 02	lda 0263,x	Geräteadresse aus Kernal-Tabelle FAT holen	} Gerätenummer (FA) ermitteln
,f327	85 ba	sta ba	und in Hilfsspeicher FA schreiben	
,f329	bd 6d 02	lda 026d,x	Sekundäradresse aus Kernal-Tabelle SAT holen	} Sekundäradresse (SA) ermitteln
,f32c	85 b9	sta b9	und in Hilfsspeicher SA schreiben	
,f32e	60	→rts	Rücksprung von Routine	

; CLALL-Routine (hierher wird normalerweise vom Kernal-Einsprung bei \$ffe7 gesprungen)

,f32f	a9 00	lda #00	0 als Anzahl der offenen Files laden	} Anzahl der offenen Files mit 0 initialisieren
,f331	85 98	sta 98	und als solche in LDTND setzen	
,f333	a2 03	ldx #03	Geräteadresse für "Bildschirm" laden	
,f335	e4 9a	cpx 9a	Vergleich mit Nummer des aktuellen Ausgabegerätes (DFLT0)	
,f337	b0 03	bcs f33c	Datasette, RS232 oder Bildschirm als aktuelles Ausgabegerät (C=1): IEC-Sonderbehandlung überspringen	
,f339	20 fe ed	jsr edfe "unlsn"	UNLISTEN-Signal über IEC-Bus senden	
,f33c	e4 99	→cpx 99	Test des aktuellen Eingabegerätes (DFLTN)	
,f33e	b0 03	bcs f343	Datasette, RS232 oder Bildschirm als aktuelles Eingabegerät (C=1): IEC-Sonderbehandlung überspringen	
,f340	20 ef ed	jsr edef "untalk"	UNTALK-Signal über IEC-Bus senden	
,f343	86 9a	→stx 9a	Bildschirm (s. \$f333) als aktuelles Ausgabegerät (DFLT0) setzen	} Bildschirm/ Tastatur als I/O-Geräte
,f345	a9 00	lda #00	Geräteadresse für "Tastatur" laden	
,f347	85 99	sta 99	und als aktuelles Eingabegerät (DLFTN) setzen	
,f349	60	rts	Rücksprung von Routine	

; OPEN-Routine (hierher wird vom Kernal-Einsprung bei \$ffc0 verzweigt)

,f34a	a6 b8	ldx b8	aktuelle Filenummer (LA) holen
-------	-------	--------	--------------------------------

```

,f34c d0 03   bne f351   andere Filenummer als 0 (Z=0): keinen Fehler auslösen
,f34e 4c 0a f7 jmp f70a "ioerr6" I/O ERROR #6 ("not input file") melden, da Filenummer 0 angegeben
-----
,f351 20 0f f3 >jsr f30f "lookup" Fileeintrag in Filetabelle suchen (Test, ob File schon vorhanden ist)
,f354 d0 03   bne f359   File noch nicht vorhanden (Z=0): keinen I/O ERROR #2 (FILE OPEN) auslösen
,f356 4c fe f6 jmp f6fe "ioerr2" I/O ERROR #2 ("file open") melden
-----
,f359 a6 98   >ldx 98     Anzahl der offenen Dateien (LDTND) als Vergleichswert und gleichzeitig als Offset
                                für den nächsten hinzuzufügenden Fileeintrag holen
,f35b e0 0a   cpx #0a     schon 10 Files (Maximalzahl) geöffnet?
,f35d 90 03   bcc f362     nein (C=0): keinen I/O ERROR #1 (TOO MANY FILES) auslösen
,f35f 4c fb f6 jmp f6fb     I/O ERROR #1 ("too many files") melden
-----
,f362 e6 98   >inc 98     Anzahl der offenen Dateien erhöhen, da neuer Fileeintrag entsteht
,f364 a5 b8   lda b8      aktuelle Filenummer (LA) holen
,f366 9d 59 02 sta 0259,x  und in LAT (Filenummertabelle) schreiben
,f369 a5 b9   lda b9      aktuelle Sekundäradresse (SA) holen
,f36b 09 60   ora #60 %01100000 Offset für Sekundäradressen bei offenen Files einblenden
,f36d 85 b9   sta b9      Ergebnis als aktuelle Sekundäradresse (SA) setzen
,f36f 9d 6d 02 sta 026d,x  und in SAT (Sekundäradressentabelle) schreiben
,f372 a5 ba   lda ba      aktuelle Geräteadresse (FA) holen
,f374 9d 63 02 sta 0263,x  und in FAT (Gerätenummerntabelle) schreiben
,f377 f0 5a   <beq f3d3     Gerät #0 (Tastatur) aktiv (Z=1): Carry löschen ("kein Fehler") und Ende
,f379 c9 03   <cmp #03     Gerät #3 (Bildschirm) aktiv?
,f37b f0 56   <beq f3d3     ja (Z=1): Carry löschen ("kein I/O-Fehler"), Rücksprung
,f37d 90 05   <bcc f384     kein Gerät am IEC-Bus aktiv (C=0): IEC-Bus-Sonderbehandlung überspringen
; OPEN-Sonderbehandlung für IEC-Bus-Geräte
-----
,f37f 20 d5 f3 jsr f3d5 "iecopn" File auf IEC-Bus öffnen (angeschlossenes Gerät ebenfalls zu OPEN-Vorgang auffordern)
,f382 90 4f   <bcc f3d3 "jmp" Carry löschen ("kein I/O-Fehler"), Rücksprung
-----
,f384 c9 02   >cmp #02     Gerät #2 (RS232) aktiv?
,f386 d0 03   bne f38b     nein (Z=0): Sonderbehandlung für Datasette auslösen
; OPEN-Sonderbehandlung für RS232
-----
,f388 4c 09 f4 jmp f409 "rsopen" File auf RS232 öffnen
-----

```

Fileparameter
LA, SA und FA
aus
Hilfsspeichern
in
Filetabellen
LAT, SAT und FAT
übertragen

; OPEN-Sonderbehandlung für Datasette

```
f38b 20 d0 f7 >jsr f7d0 "getbfa" Anfangsadresse des Kassettenpuffers nach X/Y holen und auf zulässigen Bereich testen
f38e b0 03 bcs f393 zulässige Anfangsadresse (C=1): kein I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
f390 4c 13 f7 jmp f713 "ioerr9" I/O ERROR #9 ("illegal device number") auslösen
```

```
f393 a5 b9 >lda b9 aktuelle Sekundäradresse (SA) holen
f395 29 0f and #0f %00001111 nur unteres Nibble testen
f397 d0 1f bne f3b8 nicht Lese-Sekundäradresse (Z=0): Behandlung für Schreib-Sekundäradressen
```

; File für Lesevorgänge auf Datasette öffnen

```
f399 20 17 f8 jsr f817 "wtplay" warten, bis <PLAY> an Datasette ausgelöst wird
f39c b0 36 bcs f3d4 I/O-Fehler (Abbruch) (C=1): Rücksprung über RTS bei C=1 und A=0 (BREAK-Fehlercode)
f39e 20 af f5 jsr f5af "srcmsg" Meldung "SEARCHING [FOR ...]" erzeugen
f3a1 a5 b7 lda b7 Länge des aktuellen Filenamens (FNLEN) holen
f3a3 f0 0a beq f3af kein Filename (Z=1): Suche nach bestimmtem File überspringen, nächstes File nehmen
```

; bestimmtes File suchen

```
f3a5 20 ea f7 jsr f7ea "srctfl" vorgegebenes File auf Datenkassette suchen
f3a8 90 18 bcc f3c2 kein I/O-Fehler (C=0): Fehlerabfrage überspringen
f3aa f0 28 beq f3d4 BREAK-Fehler (Fehlercode #0) (Z=1): Rücksprung mit C=1, Z=1 und A=0
f3ac 4c 04 f7 jmp f704 "ioerr4" I/O ERROR #4 ("file not found") auslösen, da File nicht vor EOT-Markierung gefunden
```

; nächstes File öffnen

```
f3af 20 2c f7 >jsr f72c "getfhd" nächsten Header-Block auf Kassette öffnen und anzeigen
f3b2 f0 20 beq f3d4 BREAK-Fehler (Fehlercode #0) (Z=1): Rücksprung mit C=1, Z=1 und A=0
f3b4 90 0c bcc f3c2 kein anderer I/O-Fehler (C=0): nicht I/O ERROR #4 (FILE NOT FOUND) auslösen
f3b6 b0 f4 bcs f3ac "jmp" I/O ERROR #4 ("file not found") auslösen, da File nicht vor EOT-Markierung gefunden
```

; File für Schreibvorgänge auf Datasette öffnen

```
f3b8 20 38 f8 >jsr f838 "wtrecpl" File prüfen, Meldungen ausgeben, auf <RECORD>+<PLAY> warten
f3bb b0 17 bcs f3d4 I/O-Fehler (C=1): Rücksprung bei C=1 und A=Fehlercode
f3bd a9 04 lda #04 %00000100 Headermarke für 1. Block eines Datenfile laden
f3bf 20 6a f7 jsr f76a "wblk" Datenblock (= Kassettenpuffer-Inhalt) auf Kassette schreiben
f3c2 a9 bf >lda #bf Länge des Kassettenpuffers laden
f3c4 a4 b9 ldy b9 aktuelle Sekundäradresse (SA) holen
```



```

,f3c6 c0 60    cpy #60 %01100000 Vergleich mit Sekundäradresse $x0 ("x" = beliebiges Hi-Nibble)
,f3c8 f0 07    beq f3d1    Übereinstimmung (Z=1): Zeiger für Kassettenpuffer (BUFNT) initialisieren
,f3ca a0 00    ldy #00      Offset mit 0 initialisieren (auf erstes Byte im Kassettenpuffer richten)
,f3cc a9 02    lda #02 %00000010 Headermarke für Folgeblock eines Datenfile laden
,f3ce 91 b2    sta (b2),y   und an erste Position im Kassettenpuffer schreiben
,f3d0 98       tya "lda #00" 0 als Zeiger für Kassettenpuffer (BUFNT) laden
,f3d1 85 a6    >sta a6      gewünschten Wert (0 oder $bf) in BUFNT (Zeiger für Kassettenpuffer) schreiben
,f3d3 18       >clc        Carry löschen (Flag für "kein Fehler")
,f3d4 60       rts         Rücksprung von Routine

```

; IECOPN-Routine: File-OPEN-Vorgang auf am IEC-Bus angeschlossenem Gerät veranlassen
(z.B. die Floppy veranlassen, daß ein File auf Diskette geöffnet wird)

```

,f3d5 a5 b9    lda b9      aktuelle Sekundäradresse (SA) holen
,f3d7 30 fa    bmi f3d3    Sekundäradresse >= $80 (N=1): Carry löschen und Ende
,f3d9 a4 b7    ldy b7      Länge des aktuellen Filenamens (FNLEN) auslesen
,f3db f0 f6    beq f3d3    kein Filename (Z=1): Carry löschen und Ende
,f3dd a9 00    lda #00     Fehlerstatus für "kein Fehler" laden
,f3df 85 90    sta 90      und in Statusbyte des Kernals schreiben
,f3e1 a5 ba    lda ba      aktuelle Gerätenummer (FA) auslesen
,f3e3 20 0c ed jsr ed0c "listen" LISTEN-Signal auf IEC-Bus ausgeben
,f3e6 a5 b9    lda b9      aktuelle Sekundäradresse (SA) holen
,f3e8 09 f0    ora #f0 %11110000 oberes Nibble (High-Nibble) setzen
,f3ea 20 b9 ed jsr edb9 "second" Sekundäradresse nach LISTEN auf IEC-Bus ausgeben
,f3ed a5 90    lda 90      Statusbyte des Kernals zwecks Test auslesen
,f3ef 10 05    bpl f3f6    Gerät ist ansprechbar (N=0): kein I/O ERROR #5 (DEVICE NOT PRESENT)
,f3f1 68       pla         LB der Rücksprungadresse vom Stapel entfernen } Rücksprungadresse am Stapel
,f3f2 68       pla         HB der Rücksprungadresse vom Stapel entfernen } löschen
,f3f3 4c 07 f7 jmp f707 "ioerr5" I/O ERROR #5 ("device not present") auslösen

,f3f6 a5 b7    >lda b7      Länge des aktuellen Filenamens (FNLEN) auslesen
,f3f8 f0 0c    beq f406    kein Filename (Z=1): UNLISTEN senden und Ende
,f3fa a0 00    ldy #00     Offset mit 0 initialisieren (auf erstes Byte des Filenamens richten)
,f3fc b1 bb    >lda (bb),y  Byte aus Filenamens über Zeiger FNADR holen
,f3fe 20 dd ed jsr eddd "iecout" Byte über IEC-Bus ausgeben
,f401 c8       iny        Offset erhöhen (auf nächstes Byte des Filenamens richten)
,f402 c4 b7    cpy b7      Vergleich mit Länge des aktuellen Filenamens (FNLEN)
,f404 d0 f6    bne f3fc    noch keine Übereinstimmung (Z=0): nächstes Zeichen ausgeben
,f406 4c 54 f6 >jmp f654 "unlsn" UNLISTEN-Signal auf IEC-Bus ausgeben

```

Filenamens
 byteweise
 über
 IEC-Bus
 an Gerät
 übermitteln

; RSOPEN: File-OPEN-Vorgang auf über RS232 angeschlossenen Gerät veranlassen

```
,f4009 20 83 f4 jsr f483 "iciars" CIA-Register nach RS232-Betrieb initialisieren
,f400c 8c 97 02 sty 0297 da Y=0 seit $f409: RS232-Statusbyte (RSTAT) mit "kein Fehler" initialisieren
,f400f c4 b7 → cpy b7 Länge des aktuellen Filenamens (FNLEN) = 0?
,f4011 f0 0a → beq f41d ja (Z=1): Setzen der RS232-Parameter (im Filenamens enthalten) überspringen
,f4013 b1 bb → lda (bb),y Byte aus Filenamens (= Parameter) holen } RS232-Parameter
,f4015 99 93 02 sta 0293,y und in RS232-Hilfsregister schreiben } (Kontroll-, Befehls-,
,f4018 c8 → iny Offset erhöhen (auf nächstes Byte richten) } Zeit- und Status-
,f4019 c0 04 → cpy #04 schon 4 Byte (Maximalzahl der Parameter) übertragen? } Register) aus Filenamens
,f401b d0 f2 → bne f40f nein (Z=0): nächsten Parameter setzen } in Hilfsspeicher holen
,f41d 20 4a ef → jsr ef4a "calcbt" Berechnung der über RS232 zu sendenden Bits pro Datenbyte
,f420 8e 98 02 stx 0298 Ergebnis der Berechnung als Anzahl der noch zu sendenden Bits (BITNUM) setzen
,f423 ad 93 02 lda 0293 RS232-Kontrollregister (M51CTR) auslesen
,f426 29 0f and #0f %00001111 oberes Nibble (High-Nibble) löschen, um Übertragungsgeschwindigkeit auszusortieren
,f428 f0 1c → beq f446 benutzerdefinierte Übertragungsgeschwindigkeit (Z=1): frei wählbare Geschwindigkeit
; verwenden; diese Möglichkeit ist nicht realisiert worden, es erfolgt aber keine
; Fehlermeldung, sondern statt dessen wird "normal" weitergearbeitet
,f42a 0a asl Index für Übertragungsgeschwindigkeit verdoppeln
,f42b aa tax und in Offset-Register X schreiben
,f42c ad a6 02 lda 02a6 Flag für PAL/NTSC auslesen
,f42f d0 09 → bne f43a PAL-Version (Z=0): Timer-Verzögerungswert für PAL-Version ermitteln
```

; NTSC-Version: Timer-Verzögerungswert auslesen

```
,f431 bc c1 fe ldy fecl,x HB des Timer-Verzögerungswertes aus NTSC-Tabelle entnehmen } NTSC-Wert für
,f434 bd c0 fe lda fec0,x LB des Timer-Verzögerungswertes aus NTSC-Tabelle entnehmen } Verzögerung holen
,f437 4c 40 f4 jmp f440 Auslesen der PAL-Parameter überspringen
```

; PAL-Version: Timer-Verzögerungswert auslesen

```
,f43a bc eb e4 → ldy e4eb,x HB des Timer-Verzögerungswertes aus PAL-Tabelle entnehmen } PAL-Wert für
,f43d bd ea e4 lda e4ea,x LB des Timer-Verzögerungswertes aus PAL-Tabelle entnehmen } Verzögerung holen
,f440 8c 96 02 sty 0296 HB des Timer-Verzögerungswertes für die richtige Baud-Rate setzen } Ergebnis in
,f443 8d 95 02 sta 0295 LB des Timer-Verzögerungswertes für die richtige Baud-Rate setzen } Hilfsspeicher
,f446 ad 95 02 lda 0295 LB des Timer-Verzögerungswertes laden
,f449 0a asl verdoppeln
,f44a 20 2e ff jsr ff2e weitere Behandlung in Unterprogramm ab $ff2e (wird nur von hier aufgerufen!)
,f44d ad 94 02 lda 0294 RS232-Befehlsregister (M51CDR) holen
```

```

,f450 4a      lsr      b0 (zuständig für Handshake) in Carry schieben
,f451 90 09    bcc f45c 3-Draht-Handshake (C=0): Sonderbehandlung für X-Draht-Handshake überspringen

; Sonderbehandlung für X-Draht-Handshake

,f453 ad 01 dd  lda dd01  Datenport B von CIA 2 auslesen
,f456 0a      asl      b7 (Data Set Ready) in Carry schieben
,f457 b0 03    bcs f45c  DSR-Bit gesetzt (C=1): Sonderbehandlung für X-Draht-Handshake verlassen
,f459 20 0d f0  jsr f00d  "missing dsr (Data Set Ready)"-Status herstellen

,f45c ad 9b 02  >lda 029b  Zeiger auf Ende des RS232-Eingabepuffers (RIDBE) holen      } Eingabepuffer
,f45f 8d 9c 02  sta 029c  und als Anfang des RS232-Eingabepuffers (RIDBS) setzen      } und
,f462 ad 9e 02  lda 029e  Zeiger auf Ende des RS232-Ausgabepuffers (RODBE) holen      } Ausgabepuffer
,f465 8d 9d 02  sta 029d  und als Anfang des RS232-Ausgabepuffers (RODBS) setzen      } initialisieren
,f468 20 27 fe  jsr fe27  MEMTOP-Routine aufrufen, damit Obergrenze für Basic nach X/Y geholt wird
,f46b a5 f8     lda  f8    HB des Zeigers auf den RS232-Eingabepuffer holen
,f46d d0 05     bne f474  RS232-Eingabepuffer vorhanden (Z=0): Sonderbehandlung überspringen
,f46f 88        dey      HB der Obergrenze des Basic-RAM verringern, um dort für $0100 Byte Platz zu schaffen
,f470 84 f8     sty  f8    HB des Zeigers auf den RS232-Eingabepuffer setzen      } RS232-Eingabepuffer
,f472 86 f7     stx  f7    LB des Zeigers auf den RS232-Eingabepuffer setzen      } einrichten
,f474 a5 fa     >lda  fa    HB des Zeigers auf den RS232-Ausgabepuffer holen
,f476 d0 05     bne f47d  RS232-Ausgabepuffer vorhanden (Z=0): Sonderbehandlung überspringen
,f478 88        dey      HB der Obergrenze des Basic-RAM verringern, um dort für $0100 Byte Platz zu schaffen
,f479 84 fa     sty  fa    HB des Zeigers auf den RS232-Ausgabepuffer setzen      } RS232-Ausgabepuffer
,f47b 86 f9     stx  f9    LB des Zeigers auf den RS232-Ausgabepuffer setzen      } einrichten
,f47d 38        >sec      HB des Zeigers auf den RS232-Ausgabepuffer holen
,f47e a9 f0     lda #f0 "sen"  Negativ-Flag setzen (Flag für "Speicherobergrenze setzen"; erübrigt sich aber, da
                             gleich bei $f480 der Einsprung für "Adresse setzen" gewählt wird!)
,f480 4c 2d fe  jmp fe2d  in MEMTOP so einsteigen, daß die Obergrenze für Basic gemäß X/Y gesetzt wird
-----

```

; ICIARS-Hilfsroutine: CIA-Register nach RS232-Betrieb initialisieren

```

,f483 a9 7f     lda #7f %01111111 CLEAR IRQ ENABLE BIT, aber alle IRQ-Anforderungsbits gesetzt
,f485 8d 0d dd  sta dd0d  in ICR (Interrupt Control Register) als Initialisierungswert schreiben
,f488 a9 06     lda #06 %00000110 DSR (Data Set Ready), CTS (Clear To Send), DCD (Data Carrier Detect), unbenutztes
                             Bit, RI (Ring Indicator), RXD (Received Data) auf "input";
                             DTR (Data Terminal Ready) und RTS (Request To Send) auf "output"
,f48a 8d 03 dd  sta dd03  in Datenrichtungsregister B (DDRB) von CIA 2 schreiben
,f48d 8d 01 dd  sta dd01  in Datenport B (DPRB) von CIA 2 schreiben

```

,f490	a9 04	lda #04 %00000100	TXD (Transmit Data) auf "high"	} TXD-Bit (Transmit Data) in Datenport A von CIA 2 auf "high" setzen
,f492	0d 00 dd	ora dd00	Bit in Datenport A von CIA 2 einblenden	
,f495	8d 00 dd	sta dd00	und Ergebnis in Datenport A zurückschreiben	
,f498	a0 00	ldy #00 %00000000	Flag für "kein RS232-IRQ" (all interrupt requests disabled) laden	
,f49a	8c a1 02	sty 02a1	und in ENABL (RS232-NMI-Flag) schreiben	
,f49d	60	rts	Rücksprung von Hilfsroutine	

; LOAD/VERIFY-Routine (hierher wird vom Kernall-Einsprung bei \$ffd5 gesprungen)

,f49e	86 c3	stx c3	LB der Ladeadresse in LB von Hilfszeiger MEMUSS schreiben	} Ladeadresse nach MEMUSS-Zeiger (\$c3/\$c4)
,f4a0	84 c4	sty c4	HB der Ladeadresse in HB von Hilfszeiger MEMUSS schreiben	
,f4a2	6c 30 03	jmp(0330)	Sprung über LOAD-Vektor; normalerweise nach \$f4a5	

; reguläre LOAD/VERIFY-Routine (Ladeadresse in \$c3/\$c4 vorausgesetzt!)

,f4a5	85 93	sta 93	übergebenes LOAD/VERIFY-Flag in Hilfsspeicher VERCK schreiben
,f4a7	a9 00	lda #00 %00000000	Initialisierungswert für "kein I/O-Fehler" laden
,f4a9	85 90	sta 90	und in Statusbyte des Kernall schreiben
,f4ab	a5 ba	lda ba	aktuelle Gerätenummer (FA) auslesen
,f4ad	d0 03	bne f4b2	anderes Gerät als Tastatur (Z=0): kein I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
,f4af	4c 13 f7	→ jmp f713 "ioerr9"	I/O ERROR #9 ("illegal device number") auslösen

,f4b2	c9 03	→ cmp #03	Vergleich der Gerätenummer mit der Geräteadresse des Bildschirms
,f4b4	f0 f9	→ beq f4af	Übereinstimmung (Z=1): I/O ERROR #9 (ILLEGAL DEVICE NUMBER) auslösen
,f4b6	90 7b	→ bcc f533 "ldtprs"	Gerätenummer #1 oder #2 (C=0): Sonderbehandlung für Datasette/RS232

; LOAD/VERIFY-Sonderbehandlung für Gerät am IEC-Bus

,f4b8	a4 b7	ldy b7	Länge des aktuellen Filenamens (FNLEN) holen
,f4ba	d0 03	bne f4bf	Filename vorhanden (Z=0): keinen I/O ERROR #8 (MISSING FILENAME) auslösen
,f4bc	4c 10 f7	→ jmp f710 "ioerr8"	I/O ERROR #8 ("missing filename") auslösen

,f4bf	a6 b9	→ ldx b9	aktuelle Sekundäradresse (SA) bis \$f4e5 merken (Flag für "relativ/absolut laden")	
,f4c1	20 af f5	jsr f5af "srcmsg"	Meldung "SEARCHING FOR [filename]" erzeugen	
,f4c4	a9 60	lda #60 %01100000	Initialisierungswert für Sekundäradresse laden	} Sekundäradresse 0 (+ Offset \$60) setzen
,f4c6	85 b9	sta b9	und als aktuelle Sekundäradresse (SA) setzen	
,f4c8	20 d5 f3	jsr f3d5 "iecopn"	File auf Gerät am IEC-Bus öffnen	
,f4cb	a5 ba	lda ba	aktuelle Gerätenummer (FA) laden	

,f4cd	20 09 ed	jsr ed09 "talk"	TALK-Signal auf IEC-Bus ausgeben
,f4d0	a5 b9	lda b9	aktuelle Sekundäradresse (SA) holen
,f4d2	20 c7 ed	jsr edc7 "tksa"	Sekundäradresse nach TALK senden
,f4d5	20 13 ee	jsr eel3 "acptr"	Byte von IEC-Bus in Akku einlesen
,f4d8	85 ae	sta ae	und als LB der absoluten Ladeadresse setzen
,f4da	a5 90	lda 90	Statusbyte des Kernals auslesen
,f4dc	4a	lsr	bl (TIMEOUT) durch zwei Rechtsverschiebungen
,f4dd	4a	lsr	ins Carry-Flag holen
,f4de	b0 50	↓ bcs f530	keine Reaktion vom angesprochenen Gerät (C=1): I/O ERROR #4 (FILE NOT FOUND)
,f4e0	20 13 ee	jsr eel3 "acptr"	nächstes Byte von IEC-Bus in Akku einlesen
,f4e3	85 af	sta af	und als HB der absoluten Ladeadresse setzen
,f4e5	8a	txa	bei \$f4bf gemerkte Sekundäradresse wieder in Akku holen
,f4e6	d0 08	↓ bne f4f0	absolutes Laden gewünscht (Z=0): absolute Ladeadresse (nach \$ae/\$af geholt) verwenden

; Sonderbehandlung: relative Ladeadresse verwenden (Hilfszeiger \$ae/\$af darauf richten)

,f4e8	a5 c3	lda c3	LB der an LOAD übergebenen Ladeadresse laden	} an LOAD/VERIFY-Routine übermittelte Ladeadresse anstatt der Adresse im File
,f4ea	85 ae	sta ae	und in LB des Hilfszeigers für die Ladeadresse schreiben	
,f4ec	a5 c4	lda c4	HB der an LOAD übergebenen Ladeadresse laden	
,f4ee	85 af	sta af	und in HB des Hilfszeigers für die Ladeadresse schreiben	

,f4f0	20 d2 f5	→ jsr f5d2 "loadng"	Ausgabe von "LOADING", sofern im Direktmodus befindlich	} TIMEOUT-BIT im Statusbyte löschen
,f4f3	a9 fd	→ lda #fd %11111101	TIMEOUT-Bit gelöscht	
,f4f5	25 90	and 90	gelöschtes TIMEOUT-Bit in Statusbyte des Kernals einblenden	
,f4f7	85 90	sta 90	und Ergebnis als neues Statusbyte setzen	
,f4f9	20 e1 ff	jsr ffe1 "stop"	STOP-Taste über Kernals-Aufruf abfragen	
,f4fc	d0 03	↓ bne f501	<STOP>-Taste nicht gedrückt (Z=0): keine Fehlerbehandlung	
,f4fe	4c 33 f6	jmp f633 "clsbrk"	File schließen und LOAD-Vorgang abbrechen	

,f501	20 13 ee	→ jsr eel3 "acptr"	nächstes Byte von IEC-Bus in Akku einlesen
,f504	aa	tax	und in X-Register merken (bis \$f50b)
,f505	a5 90	lda 90	Statusbyte des Kernals laden
,f507	4a	lsr	bl (TIMEOUT) durch zwei Rechtsverschiebungen
,f508	4a	lsr	ins Carry-Flag holen
,f509	b0 e8	↓ bcs f4f3	keine Reaktion vom angesprochenen Gerät (C=1): warten, bis Gerät ansprechbar ist oder <STOP> gedrückt wird
,f50b	8a	txa	bei \$f504 gemerktes Eingabebyte wieder in Akku bringen
,f50c	a4 93	ldy 93	bei \$f4a5 gemerktes LOAD/VERIFY-Flag VERCK zwecks Test laden
,f50e	f0 0c	↓ beq f51c	LOAD-Vorgang, nicht VERIFY (Z=1): mit 0 im Y-Register (s. \$f50c/\$f50e) die Verify-Sonderbehandlung überspringen

; VERIFY-Sonderbehandlung

```
,f510 a0 00 ldy #00      0 als Offset laden
,f512 d1 ae cmp (ae),y    Vergleich über Hilfszeiger mit aktuellem Byte im Speicher
,f514 f0 08 beq f51e      Übereinstimmung (Z=1): Hilfszeiger erhöhen, nächstes Byte verarbeiten, kein Fehler
,f516 a9 10 lda #10 %00010000 Fehlerbit für "VERIFY ERROR" laden
,f518 20 1c fe jsr felc "erstat" Fehlerbit in Statusbyte des Kernal übernehmen
```

; LOAD-Sonderbehandlung ab \$f51c

```
,f51b 2c 91 ae >bit "sta (ae),y" Byte in Speicher schreiben; Y=0 wegen $f50c/$f50e

,f51e e6 ae >inc ae      LB des Hilfszeigers auf die aktuelle Adresse im Speicher erhöhen
,f520 d0 02 >bne f524    kein Erhöhungsübertrag (Z=0): HB nicht erhöhen
,f522 e6 af >inc af      HB des Hilfszeigers auf die aktuelle Adresse im Speicher erhöhen
,f524 24 90 >bit 90      Statusbyte des Kernal testen
,f526 50 cb >bvc f4f3    EOF (End Of File)-Bit gelöscht (V=0): nächstes Byte bearbeiten
```

} Hilfszeiger
auf nächste
Adresse stellen

; File-Ende erreicht

```
,f528 20 ef ed jsr edef "untalk" UNTALK-Signal über IEC-Bus ausgeben
,f52b 20 42 f6 jsr f642 "ieccls" File auf Gerät am IEC-Bus schließen
,f52e 90 79 >bcc f5a9 "jmp" Endadresse (letzten Inhalt von $ae/$af-Hilfszeiger) nach X/Y holen und Ende
```

```
-----
,f530 4c 04 f7 jmp f704 "ioerr4" I/O ERROR #4 ("file not found") auslösen
-----
```

; LDTPRS: LOAD/VERIFY von Datasette (Gerät #1) oder RS232 (Gerät #2); Gerätenummer im Akku

```
,f533 4a lsr          b0 der Gerätenummer zwecks Test in Carry schieben
,f534 b0 03 >bcs f539  LOAD/VERIFY von Gerät #1 (Datasette) gewünscht (C=1): kein I/O ERROR #9
,f536 4c 13 f7 >jmp f713 "ioerr9" I/O ERROR #9 ("illegal device number") auslösen, da LOAD von RS232 unmöglich ist

,f539 20 d0 f7 >jsr f7d0 "getbfa" Adresse des Kassettenpuffers holen und auf zulässigen Bereich testen
,f53c b0 03 >bcs f541  gültiger Kassettenpuffer (C=1): kein I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
,f53e 4c 13 f7 >jmp f713 "ioerr9" I/O ERROR #9 ("illegal device number") auslösen, da Kassettenpuffer fehlt

,f541 20 17 f8 >jsr f817 "wtplay" Warten, bis <PLAY> an Datasette ausgelöst wird
,f544 b0 68 >bcs f5ae  I/O-Fehler (C=1): RTS bei C=1 (und A=0/Z=1, da nur BREAK-Fehler möglich)
```

```

,f546 20 af f5 jsr f5af "srcmsg" Meldung "SEARCHING [FOR ...]" erzeugen
,f549 a5 b7 → lda b7 Länge des aktuellen Filenamens (FNLEN) auslesen
,f54b f0 09 ← beq f556 kein Filename (Z=1): Suche nach richtigem Header auf Datasette überspringen

; vorgegebenes File laden

,f54d 20 ea f7 jsr f7ea "srcf1" Header zu vorgegebenem File auf Kassette suchen
,f550 90 0b ← bcc f55d kein I/O-Fehler (C=0): gefundenes File laden
,f552 f0 5a ↓ beq f5ae BREAK-Fehler (Fehlercode 0) (Z=1): RTS bei C=1/Z=1/A=0
,f554 b0 da ← bcs f530 "jmp" anderer Fehler kann nur I/O ERROR #4 (FILE NOT FOUND) gewesen sein: FILE NOT FOUND
-----

; nächstes File auf Kassette laden

,f556 20 2c f7 → jsr f72c "getfhd" nächsten Header-Block auf Kassette öffnen und anzeigen
,f559 f0 53 ↓ beq f5ae BREAK-Fehler (Fehlercode 0) (Z=1): RTS bei C=1/Z=1/A=0
,f55b b0 d3 ← bcs f530 anderer Fehler kann nur I/O ERROR #4 (FILE NOT FOUND) gewesen sein: FILE NOT FOUND

; gefundenes File (vorgegebenes oder nächstes File auf Kassette) laden

,f55d a5 90 → lda 90 Statusbyte des Kernals holen
,f55f 29 10 and #10 %00010000 alle Bits bis auf b4 (TAPE READ ERROR, Lesefehler auf Kassette) löschen
,f561 38 sec Carry als Flag für "I/O-Fehler" setzen
,f562 d0 4a ↓ bne f5ae b4 war gesetzt (Z=0): RTS bei C=1/Z=0/A=$10 (Fehlernummer für "OUT OF MEMORY")
,f564 e0 01 cpx #01 Headermarke (s. $f54d bzw. $f556) mit Wert für "Basic-Programm" vergleichen
,f566 f0 11 ← beq f579 Übereinstimmung (Z=1): relatives Laden an vorgegebene Ladeadresse
,f568 e0 03 cpx #03 Headermarke (s. $f54d bzw. $f556) mit Wert für "Maschinenprogramm" vergleichen
,f56a d0 dd ← bne f549 keine Übereinstimmung (Z=1): relatives Laden an vorgegebene Ladeadresse

; absolutes Laden an Startadresse des Programms auf Kassette

,f56c a0 01 → ldy #01 Offset mit 1 initialisieren (auf LB der absoluten Anfangsadresse stellen)
,f56e b1 b2 lda (b2),y LB der absoluten Anfangsadresse aus Kassettenpuffer entnehmen
,f570 85 c3 sta c3 und in LB des Hilfszeigers MEMUSS schreiben
,f572 c8 iny "ldy #02" Offset von 1 auf 2 erhöhen (auf HB der absoluten Anfangsadresse stellen)
,f573 b1 b2 lda (b2),y HB der absoluten Anfangsadresse aus Kassettenpuffer entnehmen
,f575 85 c4 sta c4 und in HB des Hilfszeigers MEMUSS schreiben
,f577 b0 04 ← bcs f57d "jmp" Sonderbehandlung für Basic-Programm überspringen
-----

```

; relatives Laden (Sonderfall "Basic-Programm")

,f579	a5 b9	→lda b9	aktuelle Sekundäradresse (SA) holen	
,f57b	d0 ef	→bne f56c	keine Sekundäradresse für "relatives Laden" (Z=0): trotz Headermarke für "Basic-Programm" muß absolutes Laden durchgeführt werden	
,f57d	a0 03	→ldy #03	Offset mit 3 initialisieren (auf LB der absoluten Endadresse stellen)	} Programm- länge aus
,f57f	b1 b2	lda (b2),y	LB der Endadresse aus Kassettenpuffer entnehmen	
,f581	a0 01	ldy #01	Offset mit 1 belegen (auf LB der absoluten Anfangsadresse stellen)	} Differenz von
,f583	f1 b2	sbc (b2),y	LB der absoluten Anfangsadresse (aus Kassettenpuffer) subtrahieren	
,f585	aa	tax	Ergebnis als LB der Programmlänge in X merken	} Anfangs- und
,f586	a0 04	ldy #04	Offset mit 4 initialisieren (auf HB der absoluten Endadresse stellen)	
,f588	b1 b2	lda (b2),y	HB der Endadresse aus Kassettenpuffer entnehmen	} Endadresse nach
,f58a	a0 02	ldy #02	Offset mit 1 belegen (auf HB der absoluten Anfangsadresse stellen)	
,f58c	f1 b2	sbc (b2),y	HB der absoluten Anfangsadresse (aus Kassettenpuffer) subtrahieren	} X/Y berechnen
,f58e	a8	tay	Ergebnis als HB der Programmlänge in Y merken	
,f58f	18	clc	Carry vor Addition bei \$f591 löschen	} Endadresse nach Laden
,f590	8a	txa	bei \$f585 gemerktes LB der Programmlänge zwecks Addition in Akku bringen	
,f591	65 c3	adc c3	dazu LB des Hilfszeigers MEMUSS (zeigt auf Lade-Anfang) addieren	} aus Summe von Lade- adresse
,f593	85 ae	sta ae	und als LB der Endadresse nach dem Ladevorgang merken	
,f595	98	tya	bei \$f58e gemerktes HB der Programmlänge zwecks Addition in Akku bringen	} und Länge berechnen
,f596	65 c4	adc c4	dazu HB des Hilfszeigers MEMUSS (zeigt auf Lade-Anfang) addieren	
,f598	85 af	sta af	und als HB der Endadresse nach dem Ladevorgang merken	
,f59a	a5 c3	lda c3	LB der Ladeadresse aus LB des Hilfszeigers MEMUSS entnehmen	
,f59c	85 c1	sta c1	und als LB der Startadresse setzen	
,f59e	a5 c4	lda c4	HB der Ladeadresse aus HB des Hilfszeigers MEMUSS entnehmen	
,f5a0	85 c2	sta c2	und als HB der Startadresse setzen	
,f5a2	20 d2 f5	jsr f5d2 "loadng"	Ausgabe von "LOADING/VERIFYING", sofern im Direktmodus befindlich	
,f5a5	20 4a f8	jsr f84a "tpread"	File von Kassette lesen	

; bei \$f5a9: Einsprung für "Carry löschen und Endadresse laden"

,f5a8	24 18	bit "clc"	Carry löschen (Flag für "kein I/O-Fehler")	
,f5aa	a6 ae	ldx ae	LB der Endadresse nach Ladevorgang laden	} Lade-Endadresse nach X/Y zwecks Rückgabe laden
,f5ac	a4 af	ldy af	HB der Endadresse nach Ladevorgang laden	
,f5ae	60	rts	Rücksprung von Routine	

; SRCMSG-Hilfsroutine: Ausgabe von "SEARCHING [FOR ...]", sofern im Direktmodus befindlich

,f5af	a5 9d	lda 9d	Flag für Direkt-/Programm-Modus (MSGFLG) zwecks Test auslesen
-------	-------	--------	---


```

,f5b1 10 1e      bpl f5d1      nicht im Direktmodus (N=0): Rücksprung über RTS
,f5b3 a0 0c      ldy #0c       Offset für Systemmeldung "SEARCHING" laden      } "SEARCHING"
,f5b5 20 2f fl   jsr fl2f      Routine zur Ausgabe der Systemmeldung aufrufen      } ausgeben
,f5b8 a5 b7      lda b7        Länge des aktuellen Filenamen holen      } Rücksprung, falls
,f5ba f0 15      beq f5d1      kein Filename (Z=1): Rücksprung über RTS      } Filename fehlt
,f5bc a0 17      ldy #17       Offset für Systemmeldung "FOR" laden      } "FOR"
,f5be 20 2f fl   jsr fl2f      Routine zur Ausgabe der Systemmeldung aufrufen      } ausgeben
,f5c1 a4 b7      ldy b7        Länge des aktuellen Filenamen auslesen      } Rücksprung, falls
,f5c3 f0 0c      beq f5d1      kein Filename (Z=1): Rücksprung über RTS      } Filename fehlt
,f5c5 a0 00      ldy #00       Offset mit 0 initialisieren (auf erstes Byte im Filenamen stellen)
,f5c7 b1 bb      →lda (bb),y    Byte aus Filenamen auslesen
,f5c9 20 d2 ff   jsr ffd2 "bsout" und ausgeben
,f5cc c8         iny          Offset erhöhen (auf nächstes Byte im Filenamen richten)
,f5cd c4 b7      cpy b7        Vergleich des Offset mit Länge des Filenamen (FNLEN)
,f5cf d0 f6      bne f5c7      noch keine Übereinstimmung (Z=0): nächstes Zeichen ausgeben
,f5d1 60         →rts          Rücksprung von Hilfsroutine

```

}; LOADNG-Hilfsroutine: "LOADING" oder "VERIFYING" ausgeben

```

,f5d2 a0 49      ldy #49       Offset für Systemmeldung "LOADING" vorbereiten
,f5d4 a5 93      lda 93        LOAD/VERIFY-Flag (VERCK) zwecks Test auslesen
,f5d6 f0 02      beq f5da      LOAD (Z=1): mit Offset seit $f5d2 arbeiten
,f5d8 a0 59      ldy #59       Offset für Systemmeldung "VERIFYING" laden
,f5da 4c 2b fl   →jmp fl2b     Routine zur Ausgabe der Systemmeldung aufrufen

```

; SAVE-Routine (hierher wird vom Kernal-Einsprung bei \$ffd8 gesprungen)

```

,f5dd 86 ae      stx ae        LB der Endadresse in LB des Hilfszeigers EAL/EAH schreiben      } Endadresse des SAVE-
,f5df 84 af      sty af        HB der Endadresse in HB des Hilfszeigers EAL/EAH schreiben      } Bereichs setzen
,f5e1 aa         tax          Zeropage-Adresse des Zeigers auf die Anfangsadresse in X-Register holen
,f5e2 b5 00      lda 00,x      LB des Zeigers auf die Anfangsadresse auslesen      } Anfangsadresse
,f5e4 85 c1      sta c1        und als LB des Hilfszeigers STAL/STAH setzen      } des SAVE-Bereichs
,f5e6 b5 01      lda 01,x      HB des Zeigers auf die Anfangsadresse auslesen      } in Hilfszeiger STAL
,f5e8 85 c2      sta c2        und als HB des Hilfszeigers STAL/STAH setzen      } setzen
,f5ea 6c 32 03   jmp(0332)     Sprung über SAVE-Vektor (normalerweise nach $f5ed)

```

; reguläre SAVE-Routine

,f5ed	a5 ba	lda ba	aktuelle Gerätenummer (FA) auslesen
,f5ef	d0 03	bne f5f4	nicht Tastatur (Z=0): kein I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
,f5f1	4c 13	f7→jmp f713 "ioerr9"	I/O ERROR #9 ("illegal device number") auslösen

,f5f4	c9 03	→cmp #03	Vergleich der aktuellen Gerätenummer mit der Geräteadresse des Bildschirms
,f5f6	f0 f9	beq f5f1	Übereinstimmung (Z=1): I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
,f5f8	90 5f	↓bcc f659 "svtprs"	SAVE auf Datasette oder RS232 (C=0): Sonderbehandlung anspringen

; SAVE auf Gerät am IEC-Bus

,f5fa	a9 61	lda #61 %01100001	Sekundäradresse für "Schreiben" laden
,f5fc	85 b9	sta b9	und als aktuelle Sekundäradresse (SA) setzen
,f5fe	a4 b7	ldy b7	Länge des aktuellen Filenamens (FNLEN) holen
,f600	d0 03	bne f605	Filename vorhanden (Z=0): kein I/O ERROR #8 (MISSING FILENAME)
,f602	4c 10 f7	jmp f710 "ioerr8"	I/O ERROR #8 ("missing filename") auslösen

,f605	20 d5 f3	→jsr f3d5 "iecopn"	OPEN-Vorgang auf Gerät am IEC-Bus einleiten
,f608	20 8f f6	jsr f68f "saving"	Ausgabe der Systemmeldung "SAVING [filename]", sofern im Direktmodus befindlich
,f60b	a5 ba	lda ba	aktuelle Gerätenummer (FA) auslesen
,f60d	20 0c ed	jsr ed0c "listen"	Ausgabe des LISTEN-Signals auf den IEC-Bus
,f610	a5 b9	lda b9	aktuelle Sekundäradresse (SA) holen
,f612	20 b9 ed	jsr edb9 "second"	Sekundäradresse für LISTEN senden
,f615	a0 00	ldy #00	Offset mit 0 initialisieren (wird erst bei \$f629 benötigt)
,f617	20 8e fb	jsr fb8e "stacur"	Hilfszeiger \$ae/\$af mit Startadresse aus \$c1/\$c2 initialisieren
,f61a	a5 ac	lda ac	LB der Startadresse des Speicherbereichs laden
,f61c	20 dd ed	jsr eddd "iecout"	und über IEC-Bus senden
,f61f	a5 ad	lda ad	HB der Startadresse des Speicherbereichs laden
,f621	20 dd ed	jsr eddd "iecout"	und über IEC-Bus senden
,f624	20-d1-fc	→jsr fcd1 "cmpste"	Vergleich der aktuellen Speicheradresse mit der Endadresse des Bereichs
,f627	b0 16	bcf f63f	SAVE-Bereich schon überschritten (C=1): SAVE-Vorgang abbrechen
,f629	b1 ac	lda (ac),y	aktuelles Byte holen
,f62b	20 dd ed	jsr eddd "iecout"	und auf den IEC-Bus ausgeben
,f62e	20 e1 ff	jsr ffe1 "stop"	STOP-Taste über Kernal-Einsprung abfragen
,f631	d0 07	bne f63a	STOP-Taste nicht gedrückt (Z=0): nächstes Byte bearbeiten, Zeiger erhöhen
,f633	20 42 f6	jsr f642 "ieccls"	File auf IEC-Bus schließen
,f636	a9 00	lda #00	Fehlernummer für BREAK laden
,f638	38	sec	Carry setzen (Flag für "I/O-Fehler")
,f639	60	rts	Rücksprung von Routine

```
,f63a 20 db fc->jsr fcdb "incsal" Zeiger auf aktuelle Adresse ($ac/$ad, s. $f629) erhöhen
,f63d d0 e5-----bne f624 "jmp" und nächstes Byte speichern
```

```
-----
; IECCLS-Routine: File auf Gerät am IEC-Bus schließen
```

```
,f63f 20 fe ed->jsr edfe "unlsn" UNLISTEN-Signal auf IEC-Bus ausgeben
,f642 24 b9 bit b9 aktuelle Sekundäradresse (SA) testen
,f644 30 11 bmi f657 Sekundäradresse >= $80 (N=1): Carry löschen und Rücksprung über RTS
,f646 a5 ba lda ba aktuelle Gerätenummer (FA) laden
,f648 20 0c ed jsr ed0c "listen" LISTEN-Signal auf IEC-Bus ausgeben
,f64b a5 b9 lda b9 aktuelle Sekundäradresse (SA) holen
,f64d 29 ef and #ef %11101111 b4 löschen
,f64f 09 e0 ora #e0 %11100000 b5-b7 setzen
,f651 20 b9 ed jsr edb9 "second" Sekundäradresse nach LISTEN ausgeben
,f654 20 fe ed jsr edfe "unlsn" UNLISTEN-Signal auf IEC-Bus ausgeben
,f657 18 >clc Carry löschen (Flag für "kein Fehler")
,f658 60 rts Rücksprung von Routine
-----
```

```
; SVTPRS: SAVE auf Datasette (Gerät #1) oder RS232 (Gerät #2, ergibt aber ILLEGAL DEVICE ...)
```

```
,f659 4a lsr Gerätenummer rechtsverschieben, um b0 ins Carry-Flag zu bekommen und zu testen
,f65a b0 03 bcs f65f SAVE auf Datasette (Gerät #1) gewünscht (C=1): kein I/O ERROR #9 (ILLEGAL DEVICE ...)
,f65c 4c 13 f7 jmp f713 "ioerr9" I/O ERROR #9 ("illegal device number") auslösen (kein SAVE auf RS232 möglich!)
-----
```

```
,f65f 20 d0 f7->jsr f7d0 "getbfa" Anfangsadresse des Kassettenpuffers holen; Test auf Gültigkeit des Puffers
,f662 90 8d <bcc f5f1 kein Kassettenpuffer verfügbar (C=0): I/O ERROR #9 (ILLEGAL DEVICE NUMBER)
,f664 20 38 f8 jsr f838 "wtrecpl" auf <RECORD>+<PLAY> warten
,f667 b0 25 bcs f68e I/O-Fehler (C=1): RTS bei C=1 und A=Fehlernummer (0=BREAK)
,f669 20 8f f6 jsr f68f "saving" Ausgabe der Systemmeldung "SAVING [filename]", sofern im Direktmodus befindlich
,f66c a2 03 ldx #03 Headermarke für "Maschinenprogramm" laden
,f66e a5 b9 lda b9 Sekundäradresse zwecks Auswertung holen
,f670 29 01 and #01 %00000001 b0 aussondern (entscheidet über Programmtyp "Basic" oder "Maschinensprache")
,f672 d0 02 bne f676 Maschinenprogramm (Z=0): bei $f66c vorbereitete Headermarke verwenden
,f674 a2 01 ldx #01 Headermarke für "BASIC" laden
,f676 8a >txa Headermarke (vorher in X berechnet, s. $f66c-$f674) in Akku laden
,f677 20 6a f7 jsr f76a "tapehe" Programm-Header- oder Ende-Block auf Kassette schreiben
,f67a b0 12 bcs f68e I/O-Fehler (C=1): RTS bei C=1 und A=Fehlercode
,f67c 20 67 f8 jsr f867 "tpwrit" File auf Kassette schreiben
,f67f b0 0d bcs f68e I/O-Fehler (C=1): RTS bei C=1 und A=Fehlercode
```

```

,f681 a5 b9    lda  b9      aktuelle Sekundäradresse (SA) holen
,f683 29 02    and #02 %00000010 bl aussondern (entscheidet über End-Of-Tape-Markierung)
,f685 f0 06    beq f68d     keine EOT-Markierung angefordert (Z=1): CLC und RTS
,f687 a9 05    lda  #05     Headermarke "End Of Tape" (Band-Ende) laden
,f689 20 6a f7 jsr f76a "tapehe" Programm-Header- oder Ende-Block auf Kassette schreiben
,f68c 24 18    >bit "clc"    Carry löschen (Flag für "kein Fehler")
,f68e 60      >rts          Rücksprung von Routine

```

; SAVING-Hilfsroutine: Erzeugung der Systemmeldung "SAVING [filename]", sofern im Direktmodus befindlich

```

,f68f a5 9d    lda  9d      Flag für Systemmeldungen (MSGFLG) zwecks Test auslesen
,f691 10 fb    bpl f68e     Programm-Modus (N=0): Rücksprung über RTS, Meldung nicht ausgeben
,f693 a0 51    ldy #51     Offset für Systemmeldung "SAVING" laden
,f695 20 2f f1 jsr f12f "sysmsg" Routine zur Ausgabe der Systemmeldung aufrufen
,f698 4c cl f5 jmp f5cl     in SRCMSG-Routine so einsteigen, daß die Ausgabe des Filenamens erfolgt (sofern
                          vorhanden)

```

; UDTIM-Routine (hierher wird vom Kernal-Einsprung bei \$ffea gesprungen)

```

,f69b a2 00    ldx #00      Initialisierungswert für 00:00:00 laden
,f69d e6 a2    inc  a2      niederwertigstes Byte der Systemuhr TI/TI$ erhöhen
,f69f d0 06    bne f6a7     kein Erhöhungsübertrag (Z=0): Erhöhen fertig
,f6a1 e6 a1    inc  a1      mittelwertiges Byte der Systemuhr TI/TI$ erhöhen
,f6a3 d0 02    bne f6a7     kein Erhöhungsübertrag (Z=0): Erhöhen fertig
,f6a5 e6 a0    inc  a0      höchstwertiges Byte der Systemuhr TI/TI$ erhöhen
,f6a7 38      >sec          Carry vor Subtraktion bei $f6aa setzen
,f6a8 a5 a2    lda  a2      niederwertigstes Byte der Systemuhr TI/TI$ laden
,f6aa e9 01    sbc #01      1 (LSB für 24:00:00) zwecks Vergleich subtrahieren
,f6ac a5 a1    lda  a1      mittelwertiges Byte der Systemuhr TI/TI$ laden
,f6ae e9 1a    sbc #1a      26 (MiSB für 24:00:00) zwecks Vergleich subtrahieren
,f6b0 a5 a0    lda  a0      höchstwertiges Byte der Systemuhr TI/TI$ laden
,f6b2 e9 4f    sbc #4f      79 (MSB für 24:00:00) zwecks Vergleich subtrahieren
,f6b4 90 06    bcc f6bc     noch keine 24-Stunden-Überschreitung (C=0): Rücksetzen auf 00:00:00 überspringen

```

; Sonderbehandlung: von 24:00:00 auf 00:00:00 zurückschalten

```

,f6b6 86 a0    stx  a0      höchstwertiges Byte mit MSB für 00:00:00 initialisieren
,f6b8 86 a1    stx  a1      mittelwertiges Byte mit MiSB für 00:00:00 initialisieren
,f6ba 86 a2    stx  a2      niederwertigstes Byte mit LSB für 00:00:00 initialisieren

```


; Abfrage der STOP-Taste (ermöglicht Kernal-Routine STOP \$ffef)

```

,f6bc  ad 01 dc >lda dc01      Datenport B von CIA 1 laden (Reihe der gedrückten Taste)
,f6bf  cd 01 dc   cmp dc01      mit sich selbst vergleichen
,f6c2  d0 f8     bne f6bc      noch liegt keine Übereinstimmung vor (Z=0): warten, bis keine Veränderung mehr
,f6c4  aa        tax          Akku (Inhalt von Datenport B = Tastatur-Reihen-Register) zwecks Test nach X bringen
,f6c5  30 13     bmi f6da      b7 gesetzt, also keine Taste aus STOP-Reihe gedrückt (N=1): Ende
,f6c7  a2 bd     ldx #bd %10111101 Tastatur-Reihen 6 und 1 (enthalten <SHIFT>-Tasten) anwählen
,f6c9  8e 00 dc  stx dc00      Wert in Datenport A schreiben, damit Abfrage erfolgt
,f6cc  ae 01 dc >lda dc01      Datenport B von CIA 1 laden (Reihe der gedrückten Taste)
,f6cf  ec 01 dc   cmp dc01      mit sich selbst vergleichen
,f6d2  d0 f8     bne f6cc      noch liegt keine Übereinstimmung vor (Z=0): warten, bis keine Veränderung mehr
,f6d4  8d 00 dc  sta dc00      Wert mit gelöschtem b7 (s. $f6c4/$f6c5) in Datenport A schreiben, um Spalte 7
                                abzufragen
,f6d7  e8        inx          Wert erhöhen; bei $ff (Bedeutung: keine Taste gedrückt) wird Z=1
,f6d8  d0 02     bne f6dc      Taste in Spalte 7 (SHIFT-Spalte) gedrückt (Z=0): Rücksprung, STOP ignorieren
,f6da  85 91     >sta 91       STOP-Flag (STKEY) setzen
,f6dc  60        >rts         Rücksprung von Routine

```

; RDTIM-Routine (hierher wird vom Kernal-Einsprung bei \$ffde verzweigt)

```

,f6dd  78        sei          Interrupt-Flag setzen, damit die interruptgesteuerte Erhöhung der Systemuhr nicht in
                                die Quere kommt
,f6de  a5 a2     lda  a2      niederwertigstes Byte der Systemuhr in den Akku holen
,f6e0  a6 a1     ldx  a1      mittelwertiges Byte der Systemuhr in X-Register holen
,f6e2  a4 a0     ldY  a0      höchstwertiges Byte der Systemuhr in Y-Register holen

```

; SETTIM-Routine (hierher wird vom Kernal-Einsprung bei \$ffdb verzweigt)

```

,f6e4  78        sei          Interrupt-Flag setzen, damit die interruptgesteuerte Erhöhung der Systemuhr nicht in
                                die Quere kommt
,f6e5  85 a2     sta  a2      niederwertigstes Byte der Systemuhr gemäß Akku setzen
,f6e7  86 a1     stx  a1      mittelwertiges Byte der Systemuhr gemäß X-Register setzen
,f6e9  84 a0     sty  a0      höchstwertiges Byte der Systemuhr gemäß Y-Register setzen
,f6eb  58        cli          Interrupt-Flag wieder löschen
,f6ec  60        rts         Rücksprung von Routine

```

; STOP-Routine (hierher wird vom Kernal-Einsprung bei \$ffef verzweigt)
 Die interruptgesteuerte Abarbeitung von UDTIM (\$ffea) wird vorausgesetzt.

```
,f6ed a5 91    lda  91      STOP-Flag (STKEY) zwecks Test auslesen
,f6ef c9 7f    cmp  #7f %01111111 Vergleich mit Wert für "STOP gedrückt"
,f6f1 d0 07    bne  f6fa    keine Übereinstimmung (Z=0): Rücksprung von Routine
,f6f3 08      php          Prozessorstatus bis $f6f9 merken (Z-Flag!)
,f6f4 20 cc ff  jsr  ffcc "clrchn" Löschen aller I/O-Kanäle
,f6f7 85 c6    sta  c6      Anzahl der Zeichen im Tastaturpuffer auf 0 setzen (A=0 seit $f6f4)
,f6f9 28      plp          Prozessorstatus (Z=1!) von $f6f3 wiederherstellen
,f6fa 60      >rts        Rücksprung von Routine
```

; Einsprünge für I/O-ERROR-Meldungen

```
,f6fb a9 01    lda  #01      I/O ERROR #1: TOO MANY FILES
,f6fd 2c a9 02 bit  "lda #02" I/O ERROR #2: FILE OPEN
,f700 2c a9 03 bit  "lda #03" I/O ERROR #3: FILE NOT OPEN
,f703 2c a9 04 bit  "lda #04" I/O ERROR #4: FILE NOT FOUND
,f706 2c a9 05 bit  "lda #05" I/O ERROR #5: DEVICE NOT PRESENT
,f709 2c a9 06 bit  "lda #06" I/O ERROR #6: NOT INPUT FILE
,f70c 2c a9 07 bit  "lda #07" I/O ERROR #7: NOT OUTPUT FILE
,f70f 2c a9 08 bit  "lda #08" I/O ERROR #8: MISSING FILENAME
,f712 2c a9 09 bit  "lda #09" I/O ERROR #9: ILLEGAL DEVICE NUMBER
,f715 48      pha          Fehlernummer auf den Stapel merken (bis $f722)
,f716 20 cc ff  jsr  ffcc "clrchn" I/O auf Standardgeräte
,f719 a0 00    ldy  #00      Offset für Systemmeldung "I/O ERROR#" laden (wird bei $f71f benötigt)
,f71b 24 9d    bit  9d      Test des Flags für Systemmeldungen (MSGFLG)
,f71d 50 0a    bvc  f729    keine Ausgabe von I/O ERROR #x erwünscht (V=0): Ausgabe überspringen
,f71f 20 2f f1  jsr  f12f    Routine zur Ausgabe der Systemmeldung aufrufen
,f722 68      pla          bei $f715 gemerkte Fehlernummer wieder vom Stapel holen
,f723 48      pha          und bis $f729 (für Rückgabe an aufrufende Routine) merken
,f724 09 30    ora  #30 %00110000 ASCII-Code von "0" durch OR-Verknüpfung addieren
,f726 20 d2 ff  jsr  ffd2 "bsout" und Ergebnis als Fehlernummer ausgeben
,f729 68      >pla          auf Stapel gemerkte Fehlernummer wiederherstellen
,f72a 38      sec          Carry setzen (Flag für "I/O-Fehler aufgetreten")
,f72b 60      rts        Rücksprung (C=1 und A=Fehlernummer)
```

; GETFHD-Hilfsroutine: nächsten Header auf Datasette öffnen und mit FOUND-Meldung anzeigen

,f72c	a5 93	→lda 93	LOAD/VERIFY-Flag (VERCK) holen
,f72e	48	pha	und auf den Stapel merken
,f72f	20 41 f8	jsr f841 "rblk"	nächsten Block von Datasette einlesen
,f732	68	pla	bei \$f72c/\$f72e gerettetes LOAD/VERIFY-Flag holen
,f733	85 93	sta 93	und wiederherstellen
,f735	b0 32	bcs f769	I/O-Fehler bei Block-Lesen aufgetreten (C=1): RTS bei C=1 und A=Fehlercode
,f737	a0 00	ldy #00	Offset mit 0 initialisieren (auf Header-Byte im Kassettenpuffer richten)
,f739	b1 b2	lda (b2),y	Header-Byte (erstes Byte im Kassettenpuffer) auslesen
,f73b	c9 05	cmp #05	Vergleich mit Headermarke für "End Of Tape"
,f73d	f0 2a	beq f769	Übereinstimmung (Z=1): Rücksprung von Routine
,f73f	c9 01	cmp #01	Vergleich mit Headermarke für "Anfang: Basic-Programm"
,f741	f0 08	beq f74b	Übereinstimmung (Z=1): FOUND-Meldung ausgeben, Datenblock übernehmen
,f743	c9 03	cmp #03	Vergleich mit Headermarke für "Anfang: Maschinen-Programm"
,f745	f0 04	beq f74b	Übereinstimmung (Z=1): FOUND-Meldung ausgeben, Datenblock übernehmen
,f747	c9 04	cmp #04	Vergleich mit Headermarke für "Anfang: Datenfile"
,f749	d0 e1	bne f72c	keine Übereinstimmung (Z=0): nächsten Header-Block suchen, bis gültigen gefunden

; FOUND-Meldung ausgeben, sofern im Direktmodus befindlich

,f74b	aa	→tax	Headertyp in X-Register merken
,f74c	24 9d	bit 9d	Flag für Systemmeldungen (MSGFLG) testen
,f74e	10 17	bpl f767	nicht im Direktmodus (N=0): Carry löschen und Ende, keine FOUND-Meldung
,f750	a0 63	ldy #63	Offset für Systemmeldung "FOUND" laden
,f752	20 2f f1	jsr f12f	Routine zur Ausgabe der Systemmeldung aufrufen
,f755	a0 05	ldy #05	Offset auf Anfang des Filenamen im Kassettenpuffer richten
,f757	b1 b2	→lda (b2),y	Byte aus Filenamen holen
,f759	20 d2 ff	jsr ffd2 "bsout"	und ausgeben
,f75c	c8	iny	Offset erhöhen
,f75d	c0 15	cpy #15	Vergleich mit Offset des letzten Byte des Filenamen + 1
,f75f	d0 f6	bne f757	noch keine Übereinstimmung (Z=0): Ausgabe fortsetzen
,f761	a5 a1	lda a1	mittelwertiges Byte der Systemuhr für Warteschleife laden
,f763	20 e0 e4	jsr e4e0	Routine zum Warten auf <CBM>- oder <SPACE>-Taste im gegebenen Zeitraum warten
,f766	ea	nop	Füllbyte, um nach ROM-Änderungen die Einsprungadressen beizubehalten
,f767	18	→clc	Carry löschen (Flag für "kein Fehler")
,f768	88	dey	Offset verringern
,f769	60	→rts	Rücksprung von Routine

} Filenamen
bytwweise
ausgeben

; WBLK-Routine (Header in Kassettenpuffer; Kassettenpufferinhalt auf Kassette schreiben)

,f76a	85 9e	sta 9e	gewünschte Headermarke in Hilfsspeicher merken	
,f76c	20 d0 f7	jsr f7d0 "getbfa"	Adresse des Kassettenpuffers ermitteln und auf Gültigkeit testen	
,f76f	90 5e	← bcc f7cf	kein Kassettenpuffer vorhanden (C=0): RTS bei C=0	
,f771	a5 c2	lda c2	HB der Startadresse für Laden/Speichern (STAH) holen	} Start- und Endadresse für Laden/Speichern auf dem Stapel merken
,f773	48	pha	und auf den Stapel retten	
,f774	a5 c1	lda c1	LB der Startadresse für Laden/Speichern (STAL) holen	
,f776	48	pha	und auf den Stapel retten	
,f777	a5 af	lda af	HB der Endadresse für Laden/Speichern (EAH) holen	
,f779	48	pha	und auf den Stapel retten	
,f77a	a5 ae	lda ae	LB der Endadresse für Laden/Speichern (EAL) holen	
,f77c	48	pha	und auf den Stapel retten	
,f77d	a0 bf	ldy #bf	Offset auf letztes Datenbyte im Kassettenpuffer richten	} Kassettenpuffer mit \$20
,f77f	a9 20	lda #20	Initialisierungswert (ASCII-Code von [space]) laden	
,f781	91 b2	→ sta (b2),y	Füllwert in Kassettenpuffer schreiben	} (ASCII-Code von [space])
,f783	88	dey	Offset verringern	
,f784	d0 fb	← bne f781	noch nicht alle Bytes initialisiert (Z=0): weiter in Schleife	} füllen
,f786	a5 9e	lda 9e	bei \$f76a gemerkte Headermarke holen	
,f788	91 b2	sta (b2),y	und an erste Position im Kassettenpuffer schreiben (Y=0!)	} Headermarke schreiben
,f78a	c8	iny "ldy #01"	Offset von 0 auf 1 erhöhen (auf LB der Startadresse richten)	
,f78b	a5 c1	lda c1	LB der Startadresse für Laden/Speichern (STAL) holen	} Startadresse in
,f78d	91 b2	sta (b2),y	und als LB der Anfangsadresse in Header-Block schreiben	
,f78f	c8	iny "ldy #02"	Offset von 1 auf 2 erhöhen (auf HB der Startadresse richten)	} Header-Block als Anfang des File
,f790	a5 c2	lda c2	HB der Startadresse für Laden/Speichern (STAH) holen	
,f792	91 b2	sta (b2),y	und als HB der Anfangsadresse in Header-Block schreiben	} übertragen Endadresse
,f794	c8	iny "ldy #03"	Offset von 2 auf 3 erhöhen (auf LB der Endadresse richten)	
,f795	a5 ae	lda ae	LB der Endadresse für Laden/Speichern (EAL) holen	} in Header-Block
,f797	91 b2	sta (b2),y	und als LB der Anfangsadresse in Header-Block schreiben	
,f799	c8	iny "ldy #04"	Offset von 1 auf 2 erhöhen (auf HB der Endadresse richten)	} als Ende des File
,f79a	a5 af	lda af	HB der Endadresse für Laden/Speichern (EAH) holen	
,f79c	91 b2	sta (b2),y	und als HB der Anfangsadresse in Header-Block schreiben	} übertragen
,f79e	c8	iny "ldy #05"	Zeiger auf erstes Datenbyte in Kassettenpuffer ermitteln	
,f79f	84 9f	sty 9f	und in Hilfsspeicher PTR2 merken	
,f7a1	a0 00	ldy #00	0 als Ausgangswert für Länge des Filenamen laden	
,f7a3	84 9e	sty 9e	und in Hilfsspeicher PTR1 schreiben	
,f7a5	a4 9e	→ ldy 9e	aktuellen Wert für Länge des Filenamen holen	
,f7a7	c4 b7	cpy b7	Vergleich mit Länge des aktuellen Filenamen (FNLEN)	
,f7a9	f0 0c	← beq f7b7	bereits Übereinstimmung (Z=1): alle Bytes im Kassettenpuffer, Ende	
,f7ab	b1 bb	lda (bb),y	Byte aus Filenamen holen	
,f7ad	a4 9f	ldy 9f	Offset aus PTR2 (Zeiger auf erstes Datenbyte in Kassettenpuffer) holen	


```

,f7af 91 b2    sta (b2),y    und Byte in Kassettenpuffer schreiben
,f7b1 e6 9e    inc 9e       Offset innerhalb des Filenamen erhöhen
,f7b3 e6 9f    inc 9f       Offset innerhalb des Kassettenpuffers erhöhen
,f7b5 d0 ee    bne f7a5 "jmp" nächstes Byte schreiben, Test auf "alle Zeichen kopiert" durchführen
-----
,f7b7 20 d7 f7->jsr f7d7 "bfsaea" Start- und Endadresse für Kassettenpuffer setzen, damit dieser geschrieben werden
                                kann (s. $f7be!)
,f7ba a9 69    lda #69 %01101001 Initialisierungswert für RS232-Eingabeparität laden
,f7bc 85 ab    sta ab       und in Hilfsspeicher RIPRTY (RS232-Eingabeparität) schreiben
,f7be 20 6b f8 jsr f86b "tpwrit" Schreibvorgang auf Kassette
,f7c1 a8       tay         möglichen Fehlercode in Y-Register retten (bis $f7ce)
,f7c2 68       pla         LB der Endadresse für Laden/Speichern
,f7c3 85 ae    sta ae       vom Stapel wiederherstellen (s. $f77a/$f77c)
,f7c5 68       pla         HB der Endadresse für Laden/Speichern
,f7c6 85 af    sta af       vom Stapel wiederherstellen (s. $f777/$f779)
,f7c8 68       pla         LB der Startadresse für Laden/Speichern
,f7c9 85 c1    sta c1       vom Stapel wiederherstellen (s. $f774/$f776)
,f7cb 68       pla         HB der Startadresse für Laden/Speichern
,f7cc 85 c2    sta c2       vom Stapel wiederherstellen (s. $f771/$f773)
,f7ce 98       tya         bei $f7c1 gemerkten Fehlercode (möglicherweise aufgetreten) wiederherstellen
,f7cf 60       rts         Rücksprung von Routine
-----

```

; GETBFA-Hilfsroutine: Adresse des Kassettenpuffers ermitteln und auf Gültigkeit testen

```

,f7d0 a6 b2    ldx b2       LB der Adresse des Kassettenpuffers aus LB des Zeigers TAPE1 entnehmen
,f7d2 a4 b3    ldy b3       HB der Adresse des Kassettenpuffers aus HB des Zeigers TAPE1 entnehmen
,f7d4 c0 02    cpy #02 >($0200) liegt Adreßangabe unter $0200? (Ergebnis in C: C=0 für "Kassettenpuffer ungültig,
                                da im Bereich $0000-$01ff" ; C=1 für "Kassettenpuffer in gültigem Bereich")
,f7d6 60       rts         Rücksprung von Routine (X/Y = Adresse, C=Flag für Gültigkeit)
-----

```

; BFSAEA-Hilfsroutine: Zeiger für Anfang und Ende bei Laden/Speichern auf Grenzen des Kassettenpuffers stellen, damit dieser als Eingabe/Ausgabe-Speicherbereich gilt

```

,f7d7 20 d0 f7 jsr f7d0 "getbfa" Anfangsadresse des Kassettenpuffers holen
,f7da 8a       txa         LB der Anfangsadresse zwecks Addition in Akku
,f7db 85 c1    sta c1       als LB der Startadresse für Laden/Speichern setzen
,f7dd 18       clc         Carry vor Addition löschen
,f7de 69 c0    adc #c0      Größe des Kassettenpuffers addieren (ergibt Endadresse)
,f7e0 85 ae    sta ae       und als LB der Endadresse für Laden/Speichern setzen

```

,f7e2	98	tya	HB der Anfangsadresse zwecks Addition in Akku
,f7e3	85 c2	sta c2	als HB der Startadresse für Laden/Speichern setzen
,f7e5	69 00	adc #00	Carry von \$f7de bei HB der Endadresse einbinden
,f7e7	85 af	sta af	Ergebnis als HB der Endadresse für Laden/Speichern setzen
,f7e9	60	rts	Rücksprung von Routine

; SRCTFL: vorgegebenes File auf Datasette suchen

,f7ea	20 2c	f7→jsr f72c "getfhd"	nächsten Header auf Datasette öffnen und mit FOUND-Meldung anzeigen
,f7ed	b0 1d	→bcs f80c	I/O-Fehler (C=1): Rücksprung bei C=1 und A=Fehlercode
,f7ef	a0 05	ldy #05	Offset für Filenamen im Kassettenpuffer (bei Headerblock) laden
,f7f1	84 9f	sty 9f	und in Hilfsspeicher PTR2 merken
,f7f3	a0 00	ldy #00	Offset für aktuelles Byte im Filenamen mit 0 initialisieren
,f7f5	84 9e	sty 9e	und in Hilfsspeicher PTR1 merken
,f7f7	c4 b7	→cpy b7	Vergleich des Offset im Filenamen mit Länge des aktuellen Filenamen (FNLEN)
,f7f9	f0 10	→beq f80b	bereits Übereinstimmung (Z=1): Carry löschen (Flag für "kein Fehler") und Rücksprung
,f7fb	b1 bb	lda (bb),y	Byte aus Filenamen holen
,f7fd	a4 9f	ldy 9f	Offset für Filenamen im Kassettenpuffer holen
,f7ff	d1 b2	cmp (b2),y	Vergleich der beiden Bytes (gewünschter und gefundener Filename)
,f801	d0 e7	→bne f7ea "srctfl"	keine Übereinstimmung (Z=0): neuer Versuch mit nächstem Header
,f803	e6 9e	inc 9e	Offset innerhalb des Filenamen erhöhen
,f805	e6 9f	inc 9f	Offset innerhalb des Kassettenpuffers erhöhen
,f807	a4 9e	ldy 9e	Offset innerhalb des Filenamen holen
,f809	d0 ec	→bne f7f7 "jmp"	Vergleich der nächsten beiden Bytes

,f80b	18	→clc	Carry löschen (Flag für "kein Fehler")
,f80c	60	→rts	Rücksprung von Routine

; TBFUL-Hilfsroutine: Test, ob Kassettenpuffer voll ist; gleichzeitig Zeiger in Kassettenpuffer erhöhen und Offset
 BUFPNT in Y-Register holen

,f80d	20 d0 f7	jsr f7d0 "getbfa"	Adresse des Kassettenpuffers ermitteln und auf Gültigkeit testen
,f810	e6 a6	inc a6	Offset für aktuelles Byte im Kassettenpuffer (BUFPNT) erhöhen
,f812	a4 a6	ldy a6	und in Y-Register laden
,f814	c0 c0	cpy #c0	Vergleich mit kleinstem nicht mehr zulässigem Offset (Test auf "Puffer voll")
,f816	60	rts	Rücksprung von Routine

; WTPLAY-Hilfsroutine: Warten auf Auslösen von <PLAY> an Datasette

```
,f817 20 2e f8 jsr f82e "tsplay" <PLAY>-Taste an Datasette prüfen
,f81a f0 1a      beq f836      <PLAY> gedrückt (Z=1): Carry löschen (Flag für "kein Fehler") und Rücksprung
,f81c a0 1b      ldy #1b      Offset für Systemmeldung "PRESS PLAY ON TAPE" laden } "PRESS PLAY ON TAPE"
,f81e 20 2f f1→jsr f12f      Routine zur Ausgabe der Systemmeldung aufrufen } ausgeben
,f821 20 d0 f8 >jsr f8d0 "tsstop" <STOP>-Taste (Tastatur, nicht Datasette!) prüfen } auf
,f824 20 2e f8 jsr f82e "tsplay" <PLAY>-Taste an Datasette prüfen } <PLAY>
,f827 d0 f8      bne f821      <PLAY> noch nicht gedrückt (Z=0): weiter warten, bis <PLAY> gedrückt ist } warten
,f829 a0 6a      ldy #6a      Offset für Systemmeldung "OK" laden } "OK"
,f82b 4c 2f f1 jmp f12f      Routine zur Ausgabe der Systemmeldung aufrufen } ausgeben
```

; TSPLAY-Hilfsroutine: <PLAY>-Taste an Datasette prüfen

```
,f82e a9 10      lda #10 %00010000 b4 (verantwortlich für <PLAY>) als Testbit laden
,f830 24 01      bit 01      Test des Prozessorports
,f832 d0 02      bne f836      b4 gesetzt (Z=0): Carry löschen und Ende, da <PLAY> nicht ausgelöst
,f834 24 01      bit 01      Test des Prozessorports (Test, ob <PLAY> immernoch gedrückt ist)
,f836 18          →clc      Carry löschen (Flag für "kein Fehler")
,f837 60          rts      Rücksprung von Routine
```

; WTRCPL-Hilfsroutine: Warten auf Auslösen von <RECORD>+<PLAY> an Datasette

```
,f838 20 2e f8 jsr f82e "tsplay" <PLAY>-Taste an Datasette prüfen
,f83b f0 f9      beq f836      <PLAY>-Taste gedrückt (Z=1): Rücksprung mit C=0; <RECORD> wird nicht abgefragt, da
                        dies nicht möglich ist
,f8d a0 2e      ldy #2e      Offset für Systemmeldung "PRESS RECORD & PLAY ON TAPE" laden
,f83f d0 dd      bne f81e "jmp" indirekt: Routine zur Ausgabe der Meldung, weiter wie bei WTPLAY
```

; RBLK-Routine: Datenblock von Datasette in Kassettenpuffer einlesen

```
,f841 a9 00      lda #00      Initialisierungswert ("kein Fehler") laden
,f843 85 90      sta 90      und in Statusbyte des Kernals schreiben
,f845 85 93      sta 93      Hilfsspeicher VERCK (LOAD/VERIFY-Flag) auf LOAD stellen
,f847 20 d7 f7 jsr f7d7 "bfsaea" Start- und Endadresse für Kassettenpuffer setzen, damit dieser als Einlesebereich
                        dient
```

; TPREAD-Einsprung: Programm von Kassette einlesen (vorbereitet: Anfang/Ende-Zeiger \$ae/\$af und \$cl/\$c2)

```
,f84a 20 17 f8 jsr f817 "wtplay"   Warten auf Auslösen von <PLAY> an Datasette
,f84d b0 1f     bcs f86e           I/O-Fehler (C=1): indirekt RTS auslösen (vorher Hilfsspeicher $02a0 löschen)
,f84f 78       sei                Interrupt verhindern
,f850 a9 00     lda #00           Initialisierungswert für Hilfsspeicher laden
,f852 85 aa     sta aa            RIDATA (RS232-Eingabebyte) initialisieren
,f854 85 b4     sta b4            SNSW1 (Bitzähler für RS232-Ausgabe) initialisieren
,f856 85 b0     sta b0            CMPO (Timing-Wert für Datasette) initialisieren
,f858 85 9e     sta 9e            PTR1 (Fehler in Pass 1) initialisieren
,f85a 85 9f     sta 9f            PTR2 (Fehler in Pass 2) initialisieren
,f85c 85 9c     sta 9c            DPSW (Flag für Byte-Ausgabe auf Kassette) initialisieren
,f85e a9 90     lda #90 %10010000 Bitmuster für IRQ in CIA 1 laden
,f860 a2 0e     ldx #0e           Index für IRQ-Vektor "read" ($f92c: seriell von Kassette lesen)
,f862 d0 11     bne f875 "jmp tape" allgemeine Kassettenbehandlung (IRQ-Index in X)
```

; WBLK-Routine: Datenblock von Kassettenpuffer auf Kassette schreiben

```
,f864 20 d7 f7 jsr f7d7 "bfsaea" Start- und Endadresse für Kassettenpuffer setzen, damit dieser als Quellbereich
                        der zu speichernden Daten dient
,f867 a9 14     lda #14 %00010100 Initialisierungswert für Eingabeparität laden
,f869 85 ab     sta ab            und in Hilfsspeicher für RS232-Eingabeparität schreiben
,f86b 20 38 f8 jsr f838 "wtcrpl" File prüfen, Meldungen ausgeben, auf <RECORD>+<PLAY> an Datasette warten
,f86e b0 6c     >bcs f8dc          I/O-Fehler (C=1): RTS auslösen (vorher Hilfsspeicher $02a0 löschen)
```

; TPWRIT-Einsprung: Schreiben auf Kassette (vorbereitet: Anfang/Ende-Zeiger \$ae/\$af und \$cl/\$c2)

```
,f870 78       sei                Interrupt verhindern, da neuer IRQ angefordert wird
,f871 a9 82     lda #82 %10000010 Bitmuster für IRQ von CIA 1 laden
,f873 a2 08     ldx #08           Index für IRQ-Vektor "wrtz" ($fc6a: seriell auf Kassette schreiben) laden
```

; TAPE: Kassettenbehandlung (IRQ-Index in X)

```
,f875 a0 7f     >ldy #7f %01111111 Bitmuster für "no irq enabled" laden (kein IRQ)
,f877 8c 0d dc sty dc0d           und in ICR (Interrupt Control Register) von CIA 1 schreiben
,f87a 8d 0d dc sta dc0d           Bitmuster für IRQ in CIA 1 in ICR (Interrupt Control Register) schreiben
,f87d ad 0e dc lda dc0e           CRA (Control Register A) von CIA 1 holen
,f880 09 19     ora #19 %00011001 Timer A auf "start", "one shot" und "force load" setzen
,f882 8d 0f dc sta dc0f           Ergebnis in CRB (Control Register B) von CIA 1 schreiben
,f885 29 91     and #91 %10010001 alle Bits bis auf b0, b4 und b7 löschen (b0, b4 und b7 aussondern)
```



```

,f887 8d a2 02 sta 02a2      und Ergebnis in Zeitvergleich-Speicher für Kassetten-I/O schreiben
,f88a 20 a4 f0 jsr f0a4 "rsp232" warten, bis RS232 fertig ist
,f88d ad 11 d0 lda d011      VIC-Steueregister holen
,f890 29 ef and #ef %11101111 b4 löschen, um Bildschirm abzuschalten ("blanking")
,f892 8d 11 d0 sta d011      und Ergebnis in VIC-Steueregister schreiben
,f895 ad 14 03 lda 0314      LB der Adresse der aktuellen IRQ-Routine holen
,f898 8d 9f 02 sta 029f      und in LB des Hilfszeigers IRQTMP retten
,f89b ad 15 03 lda 0315      HB der Adresse der aktuellen IRQ-Routine holen
,f89e 8d a0 02 sta 02a0      und in HB des Hilfszeigers IRQTMP retten
,f8a1 20 bd fc jsr fcbd "bsiv" IRQ-Vektor nach Index in X setzen
,f8a4 a9 02 lda #02          Anzahl der Leseversuche laden
,f8a6 85 be sta be          und in Hilfsspeicher FSBLK (Blockzähler) schreiben
,f8a8 20 97 fb jsr fb97 "newch" Register für serielles Lesen/Schreiben initialisieren
,f8ab a5 01 lda 01          Datenregister des Prozessors holen
,f8ad 29 1f and #1f %00011111 b5-b7 löschen; b6/b7 sind ungenutzt; b5 = Kassettenmotor
,f8af 85 01 sta 01          und in Datenregister zurückschreiben
,f8b1 85 c0 sta c0          und CAS1 (Kassettenmotor-Flag) setzen, da A <> 0

```

; Verzögerungsschleife für 0.3 Sekunden (für Bandanlauf)

```

,f8b3 a2 ff ldx #ff          Dekrementierzähler für "äußere Schleife" initialisieren
,f8b5 a0 ff ldy #ff          Dekrementierzähler für "innere Schleife" initialisieren
,f8b7 88      dey            inneren Dekrementierzähler verringern
,f8b8 d0 fd bne f8b7          noch nicht abgelaufen (Z=0): weiter verzögern
,f8ba ca      dex            äußeren Dekrementierzähler verringern
,f8bb d0 f8 bne f8b5          noch nicht abgelaufen (Z=0): weiter verzögern

```

; I/O-Abschluß abwarten

```

,f8bd 58 cli                Interrupt zulassen (führt jetzt Lese- bzw. Schreibvorgang aus)
,f8be ad a0 02 lda 02a0      HB der Adresse des aktuellen IRQ holen
,f8c1 cd 15 03 cmp 0315      Vergleich mit HB des IRQ-Vektors
,f8c4 18 clc                Carry löschen (Flag für "kein Fehler")
,f8c5 f0 15 beq f8dc          Übereinstimmung (Z=1): wieder alter IRQ aktiv, also $02a0 (IRQ-HB) löschen und Ende
,f8c7 20 d0 f8 jsr f8d0 "tsstop" <STOP>-Taste der Tastatur (nicht Datasette!) prüfen
,f8ca 20 bc f6 jsr f6bc      in UDTIM-Routine einsteigen, um Tastaturabfrage von <STOP> zu ermöglichen
,f8cd 4c be f8 jmp f8be      Wartevorgang fortsetzen

```

; TSSTOP-Hilfsroutine: Prüfung der <STOP>-Taste an der Tastatur, nicht an der Datasette ...

```

,f8d0 20 e1 ff jsr ffel "stop" STOP-Abfrage über Kernal-Einsprung (Voraussetzung: Abarbeitung von UDTIM!)

```

```

,f8d3 18      clc          Carry löschen (Flag für "kein Fehler")
,f8d4 d0 0b    bne f8e1    <STOP>-Taste nicht gedrückt (Z=0): Rücksprungüber RTS

; gedrückte STOP-Taste (während Datasettenbehandlung) ausführen

,f8d6 20 93 fc  jsr fc93 "stptap"  Kassette stoppen (Motor ausschalten und IRQ zurücksetzen)
,f8d9 38      sec          Carry setzen (Flag für "Fehler (in diesem Fall: BREAK ERROR)")
,f8da 68      pla          LB der Rücksprungadresse vom Stapel löschen      } Rücksprungadresse auf
,f8db 68      pla          HB der Rücksprungadresse vom Stapel löschen      } Stapel löschen
,f8dc a9 00    >lda #00      Initialisierungswert für HB des IRQ-Zeigers IRQTMP und zugleich Fehlernummer für
                          "BREAK (Abbruch über STOP-Taste)" laden
,f8de 8d a0 02  sta 02a0     HB des Hilfszeigers IRQTMP (enthält normalen Inhalt des IRQ-Vektors $0314/$0315,
                          wenn dieser für Kassettenbetrieb verändert wird) zurücksetzen
,f8e1 60      >rts          Rücksprung von Routine; C=1 und A=0 (BREAK ERROR)

```

; SETPIN-Hilfsroutine: Datasette für Lesevorgang vorbereiten

```

,f8e2 86 b1    stx  b1      Zeitkonstante setzen
,f8e4 a5 b0    lda  b0      CPMO (Timing-Wert für Datasette) holen
,f8e6 0a      asl          Akku:=Akku*2      } Akku (Timing-Wert)
,f8e7 0a      asl          Akku:=Akku*2      } vervierfachen
,f8e8 18      clc          Carry vor Addition löschen
,f8e9 65 b0    adc  b0      Timing-Wert addieren
,f8eb 18      clc          Carry vor Addition löschen
,f8ec 65 b1    adc  b1      5-fachen Timing-Wert zu Zeitkonstante (s. $f8e2) addieren
,f8ee 85 b1    sta  b1      und Ergebnis als neue Zeitkonstante setzen
,f8f0 a9 00    lda  #00     Initialisierungswert für Schiebe-Wert laden
,f8f2 24 b0    bit  b0      CPMO (Timing-Wert für Datasette) testen
,f8f4 30 01    bmi f8f7     Zeitkorrektur negativ (N=1): Akku nicht verdoppeln
,f8f6 2a      rol          Akku verdoppeln (Carry einbinden)      } Akku
,f8f7 06 b1    >asl  b1     zugleich Zeitkonstante verdoppeln      } und
,f8f9 2a      rol          Akku verdoppeln                        } Zeitkonstante
,f8fa 06 b1    asl  b1     zugleich Zeitkonstante verdoppeln      } vervierfachen (Zeiteinheit = 4 Takte)
,f8fc 2a      rol          Akku verdoppeln
,f8fd aa      tax          Akku bis $f90a in X-Register merken
,f8fe ad 06 dc >lda dc06    LB von Timer B holen
,f901 c9 16    cmp  #16     mit 22 vergleichen
,f903 90 f9    bcc f8fe     kleinerer Wert (C=0): warten, bis Timer B unter 22 abgelaufen ist
,f905 65 b1    adc  b1      Zeitkonstante zum letzten Inhalt des LB von Timer B addieren
,f907 8d 04 dc  sta dc04    und Ergebnis als LB von Timer A setzen

```

} Timer A
 } gemäß
 } Zeitkonstanten

,f90a	8a	txa	bei \$f8fd gemerkten Akku wiederherstellen	(Akku; \$b1)
,f90b	6d 07 dc	adc dc07	und dazu das HB von Timer B addieren	und Timer B
,f90e	8d 05 dc	sta dc05	Ergebnis als HB von Timer A setzen	setzen
,f911	ad a2 02	lda 02a2	CASTON (Zeitvergleich für Kassettenoperationen) holen	
,f914	8d 0e dc	sta dc0e	und in CRA (Steuerregister A) von CIA 1 schreiben	
,f917	8d a4 02	sta 02a4	ebenfalls in temporäres Register für Kassettenbetrieb schreiben (Flag für "Timer läuft")	
,f91a	ad 0d dc	lda dc0d	ICR (Interrupt Control Register) von CIA 1 auslesen	
,f91d	29 10	and #10 %00010000	b4 aussondern (zuständig für IRQ bei Kassetten-Lesevorgang)	
,f91f	f0 09	beq f92a	Flag-IRQ wurde nicht veranlaßt (Z=1): Interrupt wieder zulassen und Rücksprung	
,f921	a9 f9	lda #f9 >(\$f92b-1)	LB der Adresse des RTS-Befehls laden	RTS bei \$f92b
,f923	48	pha	und als LB der Rücksprungadresse auf den Stapel legen	als Ziel für
,f924	a9 2a	lda #2a <(\$f92b-1)	HB der Adresse des RTS-Befehls laden	Rücksprung
,f926	48	pha	und als HB der Rücksprungadresse auf den Stapel legen	setzen
,f927	4c 43 ff	jmp ff43 "tpirq"	IRQ-Behandlung für Datasette auslösen	

,f92a	58	cli	Interrupt zulassen	
,f92b	60	rts	Rücksprung von Routine	

; IRQ-Routine "read": seriell von Kassette lesen

,f92c	ae 07 dc	ldx dc07	HB des Timer B in CIA 1 holen	Warten, bis
,f92f	a0 ff	ldy #ff %11111111	Bitmuster für maximalen Timer-Wert laden	Timer B
,f931	98	tya	und zwecks Subtraktion in Akku laden	nicht
,f932	ed 06 dc	sbc dc06	LB des Timer B von \$ff abziehen	mehr
,f935	ec 07 dc	cpx dc07	Ergebnis mit HB von Timer B vergleichen	verändert
,f938	d0 f2	bne f92c	keine Übereinstimmung (Z=0): warten, bis Übereinstimmung vorliegt	wird
,f93a	86 b1	stx b1	HB von Timer B als neue Zeitkonstante setzen	
,f93c	aa	tax	Subtraktionsergebnis in X-Register merken	
,f93d	8c 06 dc	sty dc06	maximalen Wert in LB von Timer B schreiben (Y=\$ff seit \$f92f)	Timer B auf
,f940	8c 07 dc	sty dc07	maximalen Wert in HB von Timer B schreiben (Y=\$ff seit \$f92f)	\$ffff setzen
,f943	a9 19	lda #19 %0011001	Bits für "one shot", "force load" und "start" gesetzt	und
,f945	8d 0f dc	sta dc0f	und in CRB (Control Register B) von Timer B schreiben	starten
,f948	ad 0d dc	lda dc0d	ICR (Interrupt Control Register) von CIA 1 auslesen	
,f94b	8d a3 02	sta 02a3	und in temporäres Register für Kassettenbetrieb schreiben	
,f94e	98	tya "lda #\$ff"	Akku zwecks Subtraktion mit \$ff laden	
,f94f	e5 b1	sbc b1	davon die Zeitkonstante (bei \$f93a gesetzt) subtrahieren	
,f951	86 b1	stx b1	bei \$f93c gemerktes Subtraktionsergebnis als Zeitkonstante setzen	
,f953	4a	lsr	Subtraktionsergebnis (s. \$f94e/\$f94f) halbieren	Subtraktions-
,f954	66 b1	ror b1	Zeitkonstante halbieren	ergebnis und

,f956	4a	lsr	Subtraktionsergebnis (s. \$f94e/\$f94f) halbieren	} Zeitkonstante } durch 4 teilen
,f957	66 b1	ror bl	Zeitkonstante halbieren	
,f959	a5 b0	lda b0	Zeitkorrektur-Byte laden	
,f95b	18	clc	Carry vor Addition löschen	
,f95c	69 3c	adc #3c	60 addieren, um Zeitpunkt #1 zu erhalten	
,f95e	c5 b1	cmp bl	Vergleich mit Zeitkonstante	
,f960	b0 4a	bcs f9ac	Kurzpuls (C=1): Sonderbehandlung	
,f962	a6 9c	ldx 9c	DPSW (Flag für "Byte empfangen") testen	
,f964	f0 03	beq f969	kein Startbit gefunden (Z=1): auf Byte warten	
,f966	4c 60 fa	jmp fa60 "tprbyt"	von Kassette eingelesenes Byte verarbeiten	

,f969	a6 a3	>ldx a3	Bitzähler zwecks Test auslesen	
,f96b	30 1b	bmi f988	Bitzähler ist abgelaufen (N=1): Startpuls behandeln	
,f96d	a2 00	ldx #00	Wert für Nullpuls laden	
,f96f	69 30	adc #30	48 addieren	
,f971	65 b0	adc b0	Zeitkorrektur-Wert addieren, um Zeitpunkt #2 zu erhalten	
,f973	c5 b1	cmp bl	Vergleich mit Zeitkonstante	
,f975	b0 1c	bcs f993	Zeitpunkt #1 < Zeitkonstante <= Zeitkonstante #2 (C=1): Nullpuls-Sonderbehandlung	
,f977	e8	inx "ldx #01"	Wert für Einspuls laden	
,f978	69 26	adc #26	38 addieren	
,f97a	65 b0	adc b0	Zeitkorrektur-Wert addieren, um Zeitpunkt #3 zu erhalten	
,f97c	c5 b1	cmp bl	Vergleich mit Zeitkonstante	
,f97e	b0 17	bcs f997	Zeitpunkt #2 < Zeitkonstante <= Zeitkonstante #3 (C=1): Einspuls-Sonderbehandlung	
,f980	69 2c	adc #2c	44 addieren	
,f982	65 b0	adc b0	Zeitkorrektur-Wert addieren, um Zeitpunkt #4 zu erhalten	
,f984	c5 b1	cmp bl	Vergleich mit Zeitkonstante	
,f986	90 03	bcc f98b	nicht Zeitpunkt #3 < Zeitkonstante <= Zeitkonstante #4 (C=0): Langpuls-Sonderbehandlung	
,f988	4c 10 fa	>jmp fa10	Startpuls-Sonderbehandlung	

; Langpuls-Sonderbehandlung				
,f98b	a5 b4	>lda b4	BITTS (Bitzähler) testen	
,f98d	f0 1d	beq f9ac	Timer-A-Interrupt noch nicht freigegeben (Z=1): Kurzpuls-Sonderbehandlung	
,f98f	85 a8	sta a8	ansonsten Flag für Kassettenfehler setzen	
,f991	d0 19	bne f9ac "jmp"	Kurzpuls-Sonderbehandlung	

; Nullpuls-Sonderbehandlung

```
,f993 e6 a9 | inc a9      Pulszähler erhöhen
,f995 b0 02 | bcs f999 "jmp" mit erhöhtem Pulszähler in Einspuls-Sonderbehandlung fortfahren
```

; Einspuls-Sonderbehandlung

```
,f997 c6 a9 |>dec a9      Pulszähler verringern
,f999 38    |>sec      Carry vor Subtraktion setzen
,f99a e9 13 | sbc #13     Subtraktion von 19, um Soll-Wert für Zeitkonstante zu erhalten
,f99c e5 b1 | sbc b1      tatsächliche Zeitkonstante abziehen, um Abweichung zu ermitteln
,f99e 65 92 | adc 92      letzte Abweichung addieren
,f9a0 85 92 | sta 92      und Ergebnis als Gesamtabweichung setzen
,f9a2 a5 a4 | lda a4      Wechselflag holen
,f9a4 49 01 | eor #01 %00000001 und invertieren (nur b0 entscheidet) } Wechselflag
,f9a6 85 a4 | sta a4      und als neues Wechselflag setzen } für Null/Eins-Puls
,f9a8 f0 2b | beq f9d5     Wechselflag wieder auf 0 (Z=1): Verzweigung zum Bit-Abschluß (bei jedem 2. Durchlauf)
,f9aa 86 d7 | stx d7      Pulswert (0 oder 1) in Hilfsspeicher merken } invertieren
```

; Kurzpuls-Sonderbehandlung

```
,f9ac a5 b4 |>lda b4      BITTS (Bitzähler) testen
,f9ae f0 22 | beq f9d2     Timer-A-Interrupt noch nicht freigegeben (Z=1): IRQ-Abschluß
,f9b0 ad a3 02 | lda 02a3     Interrupt-Flags holen
,f9b3 29 01 | and #01 %00000001 alle Bits bis auf b0 (zuständig für Timer-A-IRQ) löschen
,f9b5 d0 05 | bne f9bc     Timer-A-IRQ am Laufen (Z=0): IRQ noch nicht abschließen
,f9b7 ad a4 02 | lda 02a4     Flag für Timer A zwecks Test auslesen
,f9ba d0 16 | bne f9d2     Timer A läuft (Z=0): IRQ abschließen
,f9bc a9 00 |>lda #00     Initialisierungswert für Null/Eins-Pulswechselflag laden
,f9be 85 a4 | sta a4      und in Nullpuls/Einspuls-Wechselflag schreiben
,f9c0 8d a4 02 | sta 02a4     auch in Flag für "Timer A abgelaufen" schreiben
,f9c3 a5 a3 | lda a3      Bitzähler holen
,f9c5 10 30 | bpl f9f7     noch nicht auf negativen Wert heruntergezählt (N=0): in Bit-Abschluß einsteigen
,f9c7 30 bf | bmi f988 "jmp" Startpuls-Sonderbehandlung indirekt anspringen
```

; Byte-Abschluß

```
,f9c9 a2 a6 | ldx #a6      Zeitintervall laden
,f9cb 20 e2 f8 | jsr f8e2 "setpin" Datasette für Lesevorgang im gegebenen Zeitintervall vorbereiten
```

,f9ce	a5	9b	lda	9b	Zeichen-Parität laden	
,f9d0	d0	b9	↖bne	f98b	Parität <> 0 (Z=0): Kassettenfehler	
,f9d2	4c	bc	fe	jmp	febc	IRQ-Abschluß (Register wiederherstellen und RTI)

; Bit-Abschluß

,f9d5	a5	92	lda	92	Abweichung der Zeitkonstanten vom Soll-Wert	
,f9d7	f0	07	beq	f9e0	keine Abweichung (Z=1): keine Veränderung des Zeitkorrektur-Wertes	
,f9d9	30	03	bmi	f9de	Zeitkonstante > Soll-Wert (N=1): Zeitkorrektur-Wert erhöhen	
,f9db	c6	b0	dec	b0	Zeitkorrektur-Wert verringern	
,f9dd	2c	e6	b0	→bit	"inc b0"	Zeitkorrektur-Wert erhöhen
,f9e0	a9	00	↗lda	#00	Löschwert laden	
,f9e2	85	92	sta	92	und in Hilfsspeicher für Abweichung schreiben	
,f9e4	e4	d7	cpx	d7	aktuellen Pulstyp mit letztem Pulstyp vergleichen	
,f9e6	d0	0f	bne	f9f7	keine Übereinstimmung (Z=1): Bitwert weiterverarbeiten, Abschluß	
,f9e8	8a		txa		Pulstyp zwecks Test in Akku bringen	
,f9e9	d0	a0	↖bne	f98b	kein Nullpuls (Z=0): Kassettenfehler	
,f9eb	a5	a9	lda	a9	Pulszähler holen	
,f9ed	30	bd	↖bmi	f9ac	schon auf negativen Wert heruntergezählt (N=1): Kurzpuls-Sonderbehandlung	
,f9ef	c9	10	cmp	#10	Vergleich mit Maximalwert 16	
,f9f1	90	b9	↖bcc	f9ac	Pulszähler < 16 (C=0): Kurzpuls-Sonderbehandlung	
,f9f3	85	96	sta	96	Flag für "Synch-Pulsfolge erkannt" setzen	
,f9f5	b0	b5	↖bcs	f9ac	"jmp"	Kurzpuls-Sonderbehandlung anspringen

; Bitwert weiterverarbeiten

,f9f7	8a		→txa	Bitwert in Akku holen	} Parität } gemäß Bit } modifizieren	
,f9f8	45	9b	eor	9b		und mit Parität verknüpfen
,f9fa	85	9b	sta	9b		Ergebnis als neue Parität setzen
,f9fc	a5	b4	lda	b4	Flag für Timer-A-Interrupt zwecks Test auslesen	
,f9fe	f0	d2	↖beq	f9d2	Timer-A-Interrupt gesperrt (Z=1): IRQ-Abschluß	
,fa00	c6	a3	dec	a3	Bitzähler verringern	
,fa02	30	c5	↖bmi	f9c9	Bitzähler auf negativen Wert abgelaufen (N=1): Byte-Abschluß	
,fa04	46	d7	lsr	d7	Bitwert in Carry holen	
,fa06	66	bf	ror	bf	und in Empfangsregister einbinden	
,fa08	a2	da	ldx	#da	Zeitintervall laden	
,fa0a	20	e2	f8	jsr	f8e2 "setpin"	Datasette im gegebenen Zeitintervall für Lesen vorbereiten
,fa0d	4c	bc	fe	jmp	febc	IRQ-Abschluß

; Startbit-Sonderbehandlung

,fa10	a5 96	lda 96	Flag für "Synch-Pulsfolge erkannt" zwecks Test auslesen
,fa12	f0 04	beq fa18	noch keine Synch-Pulsfolge erkannt (Z=1): kein Test auf Timer-A-Interrupt
,fa14	a5 b4	lda b4	Flag für "Timer-A-Interrupt freigegeben" zwecks Test auslesen
,fa16	f0 07	beq fa1f	Timer-A-Interrupt gesperrt (Z=1): Zeitauswertung
,fa18	a5 a3	lda a3	Bitzähler zwecks Test auslesen
,fala	30 03	bmi fa1f	Bitzähler auf negativen Wert abgelaufen (N=1): Zeitauswertung
,falc	4c 97 f9	jmp f997	Einspuls-Sonderbehandlung ausführen

,fa1f	46 b1	lsr b1	Zeitkonstante halbieren
,fa21	a9 93	lda #93	Konstante 147 laden
,fa23	38	sec	Carry vor Subtraktion setzen
,fa24	e5 b1	sbc b1	halbierte Zeitkonstante von 147 abziehen
,fa26	65 b0	adc b0	dazu den Zeitkorrektur-Wert addieren
,fa28	0a	asl	Ergebnis verdoppeln
,fa29	aa	tax	und in X-Register bringen
,fa2a	20 e2 f8	jsr f8e2 "setpin"	Datasette im gegebenen Zeitraum für Lesevorgang vorbereiten
,fa2d	e6 9c	inc 9c	Startbit-Flag setzen (Erhöhung ergibt Wert <> 0)
,fa2f	a5 b4	lda b4	Flag für "Timer-A-Interrupt freigegeben" zwecks Test auslesen
,fa31	d0 11	bne fa44	Timer-A-Interrupt frei (Z=0): Sonderbehandlung überspringen
,fa33	a5 96	lda 96	Flag für "Synch-Pulsfolge erkannt" zwecks Test auslesen
,fa35	f0 26	beq fa5d	Synch-Pulsfolge noch nicht erkannt (Z=1): IRQ abschließen
,fa37	85 a8	sta a8	Flag für Kassettenfehler setzen (Akku <> 0 wegen \$fa35)
,fa39	a9 00	lda #00	Löschwert laden
,fa3b	85 96	sta 96	und in Flag für "Synch-Pulsfolge erkannt" schreiben
,fa3d	a9 81	lda #81 %10000001	Bitmuster für IRQ von Timer A laden
,fa3f	8d 0d dc	sta dc0d	und in ICR (Interrupt Control Register) schreiben
,fa42	85 b4	sta b4	Flag für "Timer-A-Interrupt freigegeben" setzen
,fa44	a5 96	lda 96	Flag für "Synch-Pulsfolge erkannt" holen
,fa46	85 b5	sta b5	und in Hilfsspeicher merken
,fa48	f0 09	beq fa53	Flag war gelöscht (Z=1): Timer-A-IRQ nicht sperren
,fa4a	a9 00	lda #00	Löschwert für Flag laden
,fa4c	85 b4	sta b4	und in Flag für "Timer-A-Interrupt gesperrt" schreiben
,fa4e	a9 01	lda #01 %00000001	Bitmuster für "IRQ von Timer A sperren" laden
,fa50	8d 0d dc	sta dc0d	und in ICR (Interrupt Control Register) schreiben
,fa53	a5 bf	lda bf	Empfangsregister auslesen
,fa55	85 bd	sta bd	und in Speicher für gelesenes Datenbyte schreiben
,fa57	a5 a8	lda a8	Flag für Kassettenfehler auslesen
,fa59	05 a9	ora a9	mit Pulszähler verknüpfen

```
,fa5b 85 b6 | sta b6      und Ergebnis als Zeichenparität setzen
,fa5d 4c bc -fe-> jmp febc  IRQ abschließen
```

; Byte lesen

```
,fa60 20 97 fb | jsr fb97 "intpbt"  Hilfsspeicher für Byte-Übertragung initialisieren
,fa63 85 9c | sta 9c      Startbit-Flag löschen (A=0 seit $fa60)
,fa65 a2 da | ldx #da     Zeitintervall laden
,fa67 20 e2 f8 | jsr f8e2 "setpin"  Datasette im gegebenen Zeitintervall für Lesen vorbereiten
,fa6a a5 be | lda be     Pass-Zähler auslesen
,fa6c f0 02 | beq fa70    Pass 1 und Pass 2 bereits durchgeführt (Z=1): Pass-Flag nicht setzen
,fa6e 85 a7 | sta a7     Pass-Flag setzen (Pass #1: 2; Pass #2: 1)
,fa70 a9 0f | >lda #0f %00001111 Countdown-Test
,fa72 24 aa | bit aa     Band-Leseflag testen
,fa74 10 17 | bpl fa8d    Datei-Lesevorgang noch nicht abgeschlossen (N=0): weitere Behandlung
,fa76 a5 b5 | lda b5     Flag für "Synch-Pulsfolge erkannt" zwecks Test auslesen
,fa78 d0 0c | bne fa86    Synch-Pulsfolge noch nicht fertig (Z=0): Flag für "Datei-Lesevorgang noch nicht
abgeschlossen" setzen und IRQ abschließen
,fa7a a6 be | ldx be     Pass-Zähler holen
,fa7c ca | dex        und verringern (also auf nächsten Pass schalten)
,fa7d d0 0b | bne fa8a    beide Passes durchlaufen (Z=0): IRQ abschließen
,fa7f a9 08 | lda #08 %00001000 Fehlerbit für "long block" laden
,fa81 20 1c fe | jsr felc "erstat" und in Statusbyte des Betriebssystems übernehmen
,fa84 d0 04 | bne fa8a    "jmp" IRQ abschließen

,fa86 a9 00 | >lda #00    Flag für "Datei-Lesevorgang noch nicht abgeschlossen" laden
,fa88 85 aa | sta aa     und in Leseflag für Datasette schreiben
,fa8a 4c bc fe-> jmp febc  IRQ abschließen

,fa8d 70 31 | >bvs fac0    Datei-Lesevorgang aktiv (V=1): Test auf "short block"
,fa8f d0 18 | bne faa9    Countdown-Vorspann wird gelesen (Z=0): Countdown-Zahl dekrementieren usw.
,fa91 a5 b5 | lda b5     Flag für "Synch-Pulsfolge erkannt" zwecks Test auslesen
,fa93 d0 f5 | bne fa8a    Synch-Pulsfolge nicht fertig (Z=0): IRQ-Abschluß
,fa95 a5 b6 | lda b6     Byte-Parität zwecks Test auslesen
,fa97 d0 f1 | bne fa8a    Parität <> 0 (Z=0): IRQ-Abschluß
,fa99 a5 a7 | lda a7     Pass-Nummer holen
,fa9b 4a | lsr        Wert 1 (steht für Pass 2) setzt Carry, anderer Wert löscht es
,fa9c a5 bd | lda bd     gelesenes Countdown-Byte holen
,fa9e 30 03 | bmi faa3    b7 gesetzt (N=1): nicht zur "Dateilesen beendet"-Behandlung springen
,faa0 90 18 | bcc faba    b7=0 im 1. Pass (C=0): "Dateilesen beendet" setzen
```


Adressen	Hex	Asm	Comment
,faa2	18	clc	Carry löschen, damit darauffolgender BCS-Befehl nicht verzweigt
,faa3	b0 15	→ bcs faba	Pass #2 (C=1): "Dateilesen beendet" setzen
,faa5	29 0f	and #0f %00001111	Hi-Nibble (vor allem b7) löschen
,faa7	85 aa	sta aa	und Band-Leseflag mit Countdown-Zahl belegen
,faa9	c6 aa	→ dec aa	Countdown-Zahl herunterzählen
,faab	d0 dd	→ bne fa8a	Countdown noch nicht abgelaufen (Z=0): IRQ abschließen
,faad	a9 40	lda #40 %01000000	Flag für "Datei wird gelesen" laden
,faaf	85 aa	sta aa	und in Band-Leseflag schreiben
,fab1	20 8e fb	jsr fb8e "stacur"	Anfangsadresse für I/O in Zeiger auf aktuelle I/O-Adresse schreiben
,fab4	a9 00	lda #00	Initialisierungswert für Lesezähler laden
,fab6	85 ab	sta ab	und in Datasetten-Lesezähler schreiben
,fab8	f0 d0	→ beq fa8a "jmp"	IRQ-Abschluß

,faba	a9 80	→ lda #80 %10000000	Flag für "Datei-Lesevorgang beendet" laden
,fabc	85 aa	sta aa	und in Datasetten-Leseflag schreiben
,fabe	d0 ca	→ bne fa8a "jmp"	IRQ-Abschluß

,fac0	a5 b5	lda b5	Flag für "Synch-Pulsfolge erkannt" zwecks Test auslesen
,fac2	f0 0a	→ beq face	Synch-Pulsfolge fertig (Z=1): keine "short block"-Fehler
,fac4	a9 04	lda #04 %00000100	Fehlerbit für "short block" laden
,fac6	20 1c fe	jsr felc "erstat"	und in Statusbyte des Kernals einbinden
,fac9	a9 00	lda #00	Flag für "Dateilesen noch nicht beendet" laden (da "short block"-Fehler!),
,facb	4c 4a fb	jmp fb4a	aber Abschluß des Dateilesens durchführen

,face	20 d1 fc	→ jsr fcd1 "cmpste"	Vergleich der aktuellen Adresse mit der Endadresse des Bereichs
,fad1	90 03	→ bcc fad6	Endadresse noch unerreicht (C=0): Korrektur von Lesefehlern
,fad3	4c 48 fb	jmp fb48	Abschluß des Datei-Lesevorgangs

; Korrektur von Lesefehlern			
,fad6	a6 a7	→ ldx a7	Pass-Zähler auslesen
,fad8	ca	dex	herunterzählen
,fad9	f0 2d	→ beq fb08	Pass #2 (Z=1): Vergleich der Fehlerindizes von Pass #1 und Pass #2
,fadb	a5 93	lda 93	LOAD/VERIFY-Flag VERCK holen
,fadd	f0 0c	→ beq faeb	LOAD (Z=1): VERIFY-Zusatzbehandlung überspringen

; VERIFY-Behandlung			
,fadf	a0 00	ldy #00	Offset mit 0 initialisieren
,fael	a5 bd	lda bd	eingelestes Byte holen

,fae3	d1	ac	cmp (ac),y	Vergleich mit Byte an aktueller Position im Speicher	
,fae5	f0	04	beq faeb	Übereinstimmung (Z=1): kein VERIFY-Fehler	
,fae7	a9	01	lda #01 %00000001	fehlerhafte Byte-Parität als VERIFY-ERROR-Flag laden	
,fae9	85	b6	sta b6	und in Byte-Parität schreiben	
; Ende der VERIFY-Behandlung					
,faeb	a5	b6	→ lda b6	Byte-Parität zwecks Test auslesen	
,faed	f0	4b	↙ beq fb3a	Parität in Ordnung (Z=1): Byte in Speicher übernehmen	
,faef	a2	3d	ldx #3d	6l als maximal zulässigen Fehlerindex laden (= 30 Fehler maximal)	
,faf1	e4	9e	cpx 9e	Vergleich mit PTR1 (Fehlerindex für Pass 1)	
,faf3	90	3e	↙ bcc fb33	Fehlerindex < PTR1 (C=0): "second pass error"	
,faf5	a6	9e	ldx 9e	PTR1 (Fehlerindex für Pass 1) laden	
,faf7	a5	ad	lda ad	HB der Adresse des fehlerhaften Byte laden	
,faf9	9d	01 01	sta 0101,x	und als HB auf Stapel schreiben	
,fafc	a5	ac	lda ac	LB der Adresse des fehlerhaften Byte laden	
,fafe	9d	00 01	sta 0100,x	und als LB auf Stapel schreiben	
,fb01	e8		inx	Fehlerindex um 1 erhöhen	} Fehlerindex um Anzahl der Bytes (LB und HB der Adresse) erhöhen
,fb02	e8		inx	Fehlerindex um 1 erhöhen	
,fb03	86	9e	stx 9e	und als neuen PTR1 (Fehlerindex für Pass 1) setzen	
,fb05	4c	3a fb	jmp fb3a	Byte in Speicher übernehmen	

; Korrektur von Lesefehlern am Ende von Pass #2					
,fb08	a6	9f	→ ldx 9f	PTR2 (Fehlerindex für Pass #2) holen	
,fb0a	e4	9e	cpx 9e	und mit PTR1 (Fehlerindex für Pass #1) vergleichen	
,fb0c	f0	35	↙ beq fb43	Übereinstimmung (Z=1): Hilfszeiger weiterzählen und IRQ-Abschluß	} keine weitere Behandlung, wenn Fehler- Adresse nicht in Fehlerliste
,fb0e	a5	ac	lda ac	LB des Hilfszeigers auf die Fehleradresse holen	
,fb10	dd	00 01	cmp 0100,x	und mit LB der Adresse in der Fehlerliste vergleichen	
,fb13	d0	2e	↙ bne fb43	keine Übereinstimmung (Z=0): Hilfszeiger weiterzählen und IRQ-Abschluß	
,fb15	a5	ad	lda ad	HB des Hilfszeigers auf die Fehleradresse holen	
,fb17	dd	01 01	cmp 0101,x	und mit HB der Adresse in der Fehlerliste vergleichen	
,fb1a	d0	27	↙ bne fb43	keine Übereinstimmung (Z=0): Hilfszeiger weiterzählen und IRQ-Abschluß	} Fehlerliste
,fb1c	e6	9f	inc 9f	PTR2 (Fehlerindex für Pass #2) erhöhen	
,fb1e	e6	9f	inc 9f	PTR2 (Fehlerindex für Pass #2) erhöhen	} Fehlerindex auf nächste Adresse stellen
,fb20	a5	93	lda 93	LOAD/VERIFY-Flag VERCK zwecks Test auslesen	
,fb22	f0	0b	↙ beq fb2f	LOAD (Z=1): VERIFY-Nachprüfung überspringen	

; Sonderfall: VERIFY

,fb24	a5 bd	lda bd	gelesenes Datenbyte holen
,fb26	a0 00	ldy #00	Offset mit 0 initialisieren
,fb28	d1 ac	cmp (ac),y	Vergleich von gelesenem Datenbyte und Speicherinhalt an aktueller Adresse
,fb2a	f0 17	beq fb43	Übereinstimmung (Z=1): Hilfszeiger weiterzählen und IRQ-Abschluß
,fb2c	c8	iny "ldy #01"	Zeichenparität 1 (= Fehler) laden
,fb2d	84 b6	sty b6	und als Byte-Parität setzen, um Fehler anzuzeigen
,fb2f	a5 b6	→lda b6	Byte-Parität holen
,fb31	f0 07	beq fb3a	Parität = 0 (Z=1): Sonderbehandlung für inkorrekte Parität überspringen

; Parität nicht korrekt

,fb33	a9 10	lda #10 %00010000	Fehlerbit für "second pass error" (Fehler in Pass #2) laden
,fb35	20 1c fe	jsr felc "erstat"	und in Statusbyte des Kernals einbinden
,fb38	d0 09	bne fb43 "jmp"	Hilfszeiger weiterzählen und IRQ-Abschluß

; Parität korrekt

,fb3a	a5 93	→lda 93	LOAD/VERIFY-Flag VERCK zwecks Test auslesen
,fb3c	d0 05	bne fb43	VERIFY (Z=1): LOAD-Behandlung überspringen; Hilfszeiger weiterzählen; IRQ-Abschluß

; LOAD-Behandlung; A=0 wg. \$fb3a/\$fb3c

,fb3e	a8	tay "ldy #00"	Offset mit 0 initialisieren	} geladenes Byte in Speicher übernehmen
,fb3f	a5 bd	lda bd	eingeladenes Datenbyte holen	
,fb41	91 ac	sta (ac),y	und an aktueller Position in den Speicher schreiben	

; Hilfszeiger weiterzählen und IRQ-Abschluß

,fb43	20 db fc	→jsr fcdb "incsal"	Hilfszeiger auf aktuelle Adresse (\$ac/\$ad, s. \$fb28 und \$fb41) erhöhen
,fb46	d0 43	↙bne fb8b "jmp"	IRQ-Abschluß

; Datei-Lesevorgang beenden

,fb48	a9 80	lda #80 %100000000	Flag für "Datei-Lesevorgang abgeschlossen" laden
,fb4a	85 aa	sta aa	und in Leseflag für Datasette schreiben
,fb4c	78	sei	Interrupt verhindern, um weitere Lesevorgänge zu unterbinden

,fb4d	a2 01	ldx #01 %00000001	b7=0 (kein IRQ aktiv); b0=1 (bezieht sich auf Timer-A-IRQ)	} Timer-A-IRQ stoppen
,fb4f	8e 0d dc	stx dc0d	in ICR (Interrupt Control Register) von CIA 1 schreiben	
,fb52	ae 0d dc	ldx dc0d	ICR (Interrupt Control Register) auslesen, wodurch IRQs freigegeben werden	
,fb55	a6 be	ldx be	Pass-Flag auslesen	
,fb57	ca	dex	verringern (war Pass-Flag = 0, so wird jetzt N=1)	
,fb58	30 02	bmi fb5c	beide Passes abgelaufen (N=1): Schreiben des verringerten Pass-Flags überspringen	
,fb5a	86 be	stx be	verringertes Pass-Flag (von 2 auf 1) schreiben, also auf Pass #2 schalten	
,fb5c	c6 a7	dec a7	Pass-Zähler verringern	
,fb5e	f0 08	beq fb68	Pass 2 (Z=1): Bearbeitung des Fehlerindex von Pass #1 überspringen	
,fb60	a5 9e	lda 9e	PTR1 (Fehler-Index für Pass #1) holen	
,fb62	d0-27	bne fb8b	Fehler hat vorgelegen (Z=0): IRQ abschließen	
,fb64	85 be	sta be	ansonsten Pass-Flag löschen (A=0 wegen \$fb60/\$fb62)	
,fb66	f0-23	beq fb8b "jmp"	und dann den IRQ abschließen	

,fb68	20 93 fc	jsr fc93 "stptap"	Kassettenbetrieb beenden (Motor ausschalten und IRQ stoppen)	} Prüf- summe des geladenen Bereichs über EOR- Schleife bilden
,fb6b	20 8e fb	jsr fb8e "stacur"	Hilfszeiger \$ac/\$ad (weist auf aktuelle Adresse) auf Anfang stellen	
,fb6e	a0 00	ldy #00	Offset mit 0 initialisieren	
,fb70	84 ab	sty ab	gleichzeitig Prüfbyte löschen	
,fb72	b1 ac	lda (ac),y	Byte von Startposition aus Speicher auslesen	
,fb74	45 ab	eor ab	mit Prüfbyte verknüpfen	
,fb76	85 ab	sta ab	und Ergebnis als neues Prüfbyte setzen	
,fb78	20 db fc	jsr fcdb "incsal"	Hilfszeiger auf aktuelle Adresse (\$ac/\$ad, s. \$fb28 und \$fb41) erhöhen	
,fb7b	20 dl fc	jsr fcdl "cmpste"	Vergleich der aktuellen Adresse mit der Endadresse des Bereichs	
,fb7e	90 f2	bcc fb72	Endadresse noch nicht erreicht (C=0): Prüfsummenbildung fortsetzen	
,fb80	a5 ab	lda ab	Prüfsumme auslesen	
,fb82	45 bd	eor bd	und zwecks Vergleich mit Paritätsbyte verknüpfen	
,fb84	f0-05	beq fb8b	Übereinstimmung (Z=1): kein Fehler, sondern IRQ-Abschluß	
,fb86	a9 20	lda #20 %00100000	Fehlerbit für "chksum error" laden	
,fb88	20 lc fe	jsr felc "erstat"	und in Status-Byte des Kernals einbinden	
,fb8b	4c-bc-fe	jmp febc	IRQ abschließen	

; STACUR-Hilfsroutine: Hilfszeiger \$ac/\$ad (weist auf aktuelle I/O-Adresse) mit Anfangsadresse initialisieren

,fb8e	a5 c2	lda c2	HB der Anfangsadresse für I/O laden	} Inhalt von \$c1/\$c2 nach \$ac/\$ad übertragen
,fb90	85 ad	sta ad	und in HB des Hilfszeigers auf die aktuelle I/O-Adresse schreiben	
,fb92	a5 c1	lda c1	LB der Anfangsadresse für I/O laden	
,fb94	85 ac	sta ac	und in LB des Hilfszeigers auf die aktuelle I/O-Adresse schreiben	
,fb96	60	rts	Rücksprung von Hilfsroutine	

; NEWCH-Hilfsroutine: Register für serielles Lesen/Schreiben initialisieren

```
,fb97 a9 08    lda #08      Anzahl der Bits pro Byte laden
,fb99 85 a3    sta  a3      und in Bitzähler schreiben
,fb9b a9 00    lda #00      Initialisierungswert 0 laden
,fb9d 85 a4    sta  a4      in Wechselflag schreiben
,fb9f 85 a8    sta  a8      in Flag für Kassettenfehler schreiben
,fbal 85 9b    sta  9b      als Byte-Parität setzen
,fba3 85 a9    sta  a9      in Pulszähler schreiben
,fba5 60      rts          Rücksprung von Hilfsroutine
```

; Flanke (0- oder 1-Bit) an Kassette senden

```
,fba6 a5 bd    lda  bd      Ausgaberegister holen
,fba8 4a      lsr          auszugebendes Bit ins Carry holen
,fba9 a9 60    lda #60      Zeitwert für Nullpuls laden
,fbab 90 02    bcc fbaf     Null-Bit (C=0): mit Wert für Nullpuls weiterarbeiten
,fbad a9 b0    lda #b0      Zeitwert für Einspuls laden
,fbaf a2 00    ldx #00      gelöscht HB laden
,fbbl 8d 06 dc sta dc06     LB der Zeitkonstante (s. $fba9-$fbad) setzen
,fb b4 8e 07 dc stx dc07     HB (0, s. $fbaf) setzen
,fb b7 ad 0d dc lda dc0d     ICR (Interrupt Control Register) auslesen, um IRQs freizugeben
,fbba a9 19    lda #19 %00011001 "force load", "one shot", "start" laden
,fb bc 8d 0f dc sta dc0f     und in CRB (Control Register B) schreiben
,fb bf a5 01    lda  01      Prozessorport auslesen
,fb c1 49 08    eor #08 %00001000 b3 (für Datenausgang auf Datensette) invertieren
,fb c3 85 01    sta  01      und schreiben
,fb c5 29 08    and #08 %00001000 alle Bits bis auf b3 löschen, um b3 zu testen
,fb c7 60      rts          Rücksprung von Routine
```

; Flag für Abschluß setzen

```
,fb c8 38      sec          Carry setzen, damit bei $fb c9 eine "1" einrotiert wird
,fb c9 66 b6    ror  b6      b7 in Ende-Flag setzen
,fb cb 30 3c    bmi fc09 "jmp" IRQ abschließen
```

; Bytestring senden (IRQ-Routine "wrtn")

```
,fb cd a5 a8    lda  a8      Startbit-Flag 1 holen
,fb cf d0 12    bne fbe3     Puls 1 des Startbit vorbei (Z=0): Flanke senden, Startbit-Flag auf 2
```

; Startbit senden (Puls 1)

,fbd1	a9 10	lda #10 <(\$0110)	LB des Timer-Wertes für eine Flanke laden	} Timer-Wert für Flanke laden
,fbd3	a2 01	ldx #01 >(\$0110)	HB des Timer-Wertes für eine Flanke laden	
,fbd5	20 b1	fb jsr fbb1	Flanke senden	
,fbd8	d0 2f	↘ bne fc09	Anstiegsflanke (Z=0): IRQ abschließen	
,fbda	e6 a8	inc a8	Startbit-Flag 1 setzen (von 0 auf 1 erhöhen), da Puls 1 abgelaufen	
,fbdc	a5 b6	lda b6	Ende-Flag zwecks Test auslesen	
,fbde	10 29	↘ bpl fc09	Ende-Flag gelöscht (N=0): IRQ abschließen	
,fbe0	4c 57	fc jmp fc57	Pass-Abschluß	

; Startbit senden (Puls 2)

,fbe3	a5 a9	→ lda a9	Startbit-Flag 2 auslesen
,fbe5	d0 09	↘ bne fbf0	nach 2. Puls des Startbit (Z=0): Datenbit senden
,fbe7	20 ad	fb jsr fbad	Flanke senden (Timer-Wert #176)
,fbea	d0 1d	↘ bne fc09	Anstiegsflanke (Z=0): IRQ abschließen
,fbec	e6 a9	inc a9	Startbit-Flag erhöhen (von 0 auf 1)
,fbee	d0 19	↘ bne fc09 "jmp"	IRQ abschließen

; Datenbit senden

,fbf0	20 a6	fb → jsr fba6	Flanke senden, Timer = 96 bei 0-Bit; Timer = 176 bei 1-Bit
,fbf3	d0 14	↘ bne fc09	Anstiegsflanke (Z=0): IRQ abschließen
,fbf5	a5 a4	lda a4	Wechselflag für Bitübertragung laden
,fbf7	49 01	eor #01 %00000001	invertieren
,fbf9	85 a4	sta a4	und zurückschreiben
,fbfb	f0 0f	beq fc0c	nach zweitem Datenbit-Puls (Z=1): auf nächstes Bit schalten

; Sonderbehandlung nach 1. Datenbit-Puls: Bit für 2. Puls invertieren und vorbereiten

,fbfd	a5 bd	lda bd	Datenbyte (b0 wird jeweils gesendet) holen
,fbff	49 01	eor #01 %00000001	b0 invertieren
,fc01	85 bd	sta bd	und für zweiten Datenbit-Puls speichern
,fc03	29 01	and #01 %00000001	alle Bits bis auf b0 löschen
,fc05	45 9b	eor 9b	b0 in Byte-Parität einbinden
,fc07	85 9b	sta 9b	und neue Byte-Parität setzen
,fc09	4c bc	fe → jmp febc	IRQ abschließen

; Sonderbehandlung nach 2. Datenbit-Puls

```
,fc0c 46 bd → lsr bd      nächstes Bit in b0 schieben, von wo aus es später übertragen wird
,fc0e c6 a3 dec a3      Bitzähler verringern, da 1 Bit übertragen wurde
,fc10 a5 a3 lda a3      neuen Bitzähler zwecks Test auslesen
,fc12 f0 3a ↙ beq fc4e   Bitzähler heruntergezählt (Z=1): Paritätsbit senden
,fc14 10 f3 ↘ bpl fc09 "jmp" IRQ abschließen
```

; Countdown-Vorspann senden

```
,fc16 20 97 fb jsr fb97 "newch" Register für serielles Lesen/Schreiben initialisieren
,fc19 58 cli          Interrupts wieder zulassen
,fc1a a5 a5 lda a5     Countdown-Zähler holen
,fc1c f0 12 ↙ beq fc30   Countdown abgelaufen (Z=1): Byte-Abschluß
,fc1e a2 00 ldx #00     Initialisierungswert für Parität laden
,fc20 86 d7 stx d7      und als Paritätsbyte setzen
,fc22 c6 a5 dec a5     Synch-Bytezähler dekrementieren
,fc24 a6 be ldx be      Pulszähler auslesen
,fc26 e0 02 cpx #02     Pass #1 (hat den Flag-Wert 2)?
,fc28 d0 02 ↙ bne fc2c   nein (Z=0): b7 nicht setzen
,fc2a 09 80 ↘ ora #80 %10000000 b7 setzen
,fc2c 85 bd ↗ sta bd     neues Countdown-Byte setzen
,fc2e d0 d9 ↘ bne fc09 "jmp" IRQ abschließen
```

; Byte-Abschluß

```
,fc30 20 d1 fc → jsr fcd1 "cmpste" Vergleich des Zeigers $ac/$ad auf die aktuelle Adresse mit der Endadresse für I/O
,fc33 90 0a ↙ bcc fc3f   Endadresse noch nicht erreicht (C=0): Datei-Ende noch nicht bearbeiten
,fc35 d0 91 ↗ bne fbc8   Endadresse und aktuelle Adresse nicht identisch (Z=0): Flag für Abschluß setzen
,fc37 e6 ad inc ad      HB des Zeigers auf die aktuelle Adresse erhöhen, damit beim nächsten CMPSTE-Aufruf
                        C=1 und Z=0 wird
,fc39 a5 d7 lda d7      Paritätsbyte (Prüfsumme des Bereichs) holen
,fc3b 85 bd sta bd      und in Ausgaberegister schreiben
,fc3d b0 ca ↙ bcs fc09 "jmp" IRQ abschließen
```

```
,fc3f a0 00 → ldy #00    Offset mit 0 initialisieren
,fc41 b1 ac lda (ac),y    nächstes Datenbyte aus Speicher holen
,fc43 85 bd sta bd      und in Ausgaberegister schreiben
,fc45 45 d7 eor d7      Verknüpfung mit Paritätsbyte (Prüfsumme)
```

```
,fc47 85 d7 sta d7 und als neues Paritätsbyte (Prüfsumme) setzen
,fc49 20 db fc jsr fcdb "incsal" Zeiger auf aktuelle Adresse erhöhen
,fc4c d0 bb bne fc09 "jmp" IRQ abschließen
```

; Paritätsbit senden

```
,fc4e a5 9b lda 9b Byte-Parität holen
,fc50 49 01 eor #01 %00000001 b0 (zu sendendes Byte) invertieren
,fc52 85 bd sta bd und Ergebnis in Ausgaberegister (wovon b0 gesendet wird) schreiben
,fc54 4c bc fe jmp febc IRQ abschließen
```

; Pass-Abschluß

```
,fc57 c6 be dec be Passzähler verringern (= auf nächsten Pass umschalten)
,fc59 d0 03 bne fc5e von Pass #1 auf Pass #2 geschaltet (Z=0): Motor nicht ausschalten
,fc5b 20 ca fc jsr fcca "tapmof" Motor der Datasette ausschalten
,fc5e a9 50 lda #50 Anzahl der Synch-Pulse zwischen Pass #1 und Pass #2 laden
,fc60 85 a7 sta a7 und in Pulszähler schreiben
,fc62 a2 08 ldx #08 Offset in Vektortabelle für IRQ-Vektor "wrtz" (Synch-Markierung schreiben) laden
,fc64 78 sei Interrupt verhindern, damit IRQ-Vektor verändert werden kann
,fc65 20 bd fc jsr fcdb "bsiv" IRQ-Vektor nach Index in X setzen
,fc68 d0 ea bne fc54 "jmp" IRQ abschließen
```

; Synch-Pulsfolge senden (IRQ-Routine "wrtz")

```
,fc6a a9 78 lda #78 120 als LB für Timer-Wert laden
,fc6c 20 af fb jsr fbaf und Flanke für Timer-LB im Akku (HB = 0) senden
,fc6f d0 e3 bne fc54 Anstiegsflanke (Z=0): IRQ abschließen
,fc71 c6 a7 dec a7 Pulszähler verringern
,fc73 d0 df bne fc54 noch nicht heruntergezählt (Z=0): IRQ abschließen
,fc75 20 97 fb jsr fb97 "newch" Register für seriellles Lesen/Schreiben initialisieren
,fc78 c6 ab dec ab Pass-Zähler verringern
,fc7a 10 d8 bpl fc54 vor dem ersten Pass (N=0): längere Synch-Folge, IRQ abschließen
,fc7c a2 0a ldx #0a Offset in Vektortabelle für IRQ-Vektor "wrtn" ($fbcd) laden
,fc7e 20 bd fc jsr fcdb "bsiv" IRQ-Vektor nach Index in X setzen
,fc81 58 cli Interrupt verhindern
,fc82 e6 ab inc ab Pass-Zähler wieder von $ff auf 0 erhöhen ($fc78 rückgängig machen)
,fc84 a5 be lda be Pass-Zähler für gesamtes Datenfile auslesen
```


,fc86	f0 30	beq fcb8	schon auf 0 heruntergezählt (Z=1): Kassettenbetrieb und IRQ abschließen
,fc88	20 8e fb	jsr fb8e "stacur"	Anfangsadresse in Zeiger auf aktuelle I/O-Adresse schreiben
,fc8b	a2 09	ldx #09	Initialisierungswert für Countdown laden
,fc8d	86 a5	stx a5	und in Countdown-Zähler schreiben
,fc8f	86 b6	stx b6	Ende-Flag (b7 in \$b6) löschen, da b7 in \$09 (s. \$fc8b) = 0
,fc91	d0 83	bne fcl6 "jmp"	Countdown-Vorspann senden

; STPTAP: Kassettenbetrieb beenden (Motor ausschalten und IRQ zurücksetzen)			
,fc93	08	php	Prozessorstatus bis \$fcb6 merken
,fc94	78	sei	Interrupt verhindern
,fc95	ad 11 d0	lda d011	VIC-Register #17 auslesen
,fc98	09 10	ora #10 %00010000	Bildschirm wieder einschalten
,fc9a	8d 11 d0	sta d011	und Wert zurückschreiben
,fc9d	20 ca fc	jsr fcca "tapmof"	Motor der Datasette ausschalten
,fca0	a9 7f	lda #7f %01111111	"no irq enabled" (kein IRQ aktiv) laden
,fca2	8d 0d dc	sta dc0d	und in ICR (Interrupt Control Register) schreiben
,fca5	20 dd fd	jsr fddd	Timer initialisieren und IRQ freigeben
,fca8	ad a0 02	lda 02a0	HB der IRQ-Adresse vor Kassettenbetrieb aus HB des Hilfszeigers IRQTMP holen
,fcab	f0 09	beq fcb6	HB = 0 (Z=1): IRQTMP-Inhalt nicht wiederherstellen
,fcad	8d 15 03	sta 0315	HB der IRQ-Adresse setzen
,fcb0	ad 9f 02	lda 029f	LB der IRQ-Adresse vor Kassettenbetrieb aus LB des Hilfszeigers IRQTMP holen
,fcb3	8d 14 03	sta 0314	LB der IRQ-Adresse setzen
,fcb6	28	plp	bei \$fc93 gemerkten Prozessorstatus wiederherstellen
,fcb7	60	rts	Rücksprung von Routine

; Kassettenbetrieb beenden und IRQ abschließen			
,fcb8	20 93 fc	jsr fc93 "stptap"	Kassettenbetrieb beenden (Motor ausschalten und IRQ zurücksetzen)
,fbbb	f0 97	bne fc54 "jmp"	IRQ-Abschluß

; BSIV: IRQ-Vektor setzen (gemäß Offset in X)			
,fcbd	bd 93 fd	lda fd93,x	LB aus ROM-Tabelle entnehmen
,fcc0	8d 14 03	sta 0314	und in LB des IRQ-Vektors schreiben
,fcc3	bd 94 fd	lda fd94,x	HB aus ROM-Tabelle entnehmen
,fcc6	8d 15 03	sta 0315	und in HB des IRQ-Vektors schreiben
,fcc9	60	rts	Rücksprung von Routine

; TAPMOF: Motor der Datasette ausschalten

```
,fcc a5 01    lda 01      Prozessorport auslesen
,fcc 09 20    ora #20 %00100000 b5 (zuständig für Kassettenmotor) setzen (= Kassette aus)
,fce 85 01    sta 01      und in Prozessorport schreiben
,fcd 60      rts          Rücksprung von Routine
```

; CMPSTE-Hilfsroutine

```
,fcd1 38      sec          Carry vor Subtraktion setzen
,fcd2 a5 ac    lda ac       LB des Hilfszeigers auf die aktuelle Adresse laden
,fcd4 e5 ae    sbc ae       LB des Zeigers auf das Programmende zwecks Vergleich subtrahieren
,fcd6 a5 ad    lda ad       HB des Hilfszeigers auf die aktuelle Adresse laden
,fcd8 e5 af    sbc af       HB des Zeigers auf das Programmende zwecks Vergleich subtrahieren
,fcd a 60      rts          Rücksprung von Hilfsroutine
```

} Subtraktion
der End- von
der aktuellen
Adresse

; INCSAL-Hilfsroutine

```
,fcd b e6 ac    inc ac       LB des Hilfszeigers auf die aktuelle Adresse erhöhen
,fcd d d0 02    bne fce1     kein Erhöhungsübertrag (Z=0): Rücksprung über RTS
,fcd f e6 ad    inc ad       HB des Hilfszeigers auf die aktuelle Adresse erhöhen
,fce 1 60      rts          Rücksprung von Hilfsroutine
```

; RESET-Routine

```
,fce2 a2 ff    ldx #ff      Initialisierungswert für Stapelzeiger laden
,fce4 78      sei          Interrupt verhindern
,fce5 9a      txs          Stapelzeiger mit $ff (s. $fce2) initialisieren, also ganzen Stapel freigeben
,fce6 d8      cld          Dezimalflag löschen, auf Hexadezimalmodus schalten
,fce7 20 02 fd  jsr fd02 "chkcbm" auf CBM80-Markierung testen
,fce a d0 03    bne fcef     nicht vorhanden (Z=0): weiter in RESET-Routine
,fce c 6c 00 80 jmp(8000)    neu definierte RESET-Routine anspringen
```

```
,fcef 8e 16 d0 >stx d016    VIC-Register #22 nach X setzen, wodurch das kurzfristige "Schrumpfen" des
                          Bildschirms während eines RESET entsteht
,fcf2 20 a3 fd  jsr fda3 "ioinit" Initialisierung der CIA-Register
,fcf5 20 50 fd  jsr fd50 "ramtas" RAM initialisieren, Kassettenpuffer einrichten, Bildschirm auf $0400 setzen
,fcf8 20 15 fd  jsr fd15 "restor" Standard-I/O-Vektoren zurücksetzen
```

```
,fcfb 20 5b ff jsr ff5b "cint"  Bildschirm-Editor-Initialisierung
,fcfe 58      cli               Interrupt wieder zulassen ($fce4 rückgängig machen)
,fcff 6c 00 a0 jmp(a000)        über Kaltstart-Vektor des Basic-Interpreters nach $e394 springen
```

```
-----
; CHKCBM-Hilfsroutine: auf CBM80-Markierung ab $8003 testen
```

```
,fd02 a2 05      ldx #05          Anzahl der zu testenden Bytes in Dekrementierzähler als Initialisierungswert laden
,fd04 bd 0f      fd→lda fd0f,x     Byte aus CBM80-Tabelle im ROM holen
,fd07 dd 03 80   cmp 8003,x       und mit RAM-Markierung vergleichen
,fd0a d0 03      bne fd0f         keine Übereinstimmung (Z=0): Rücksprung mit Z=0, da CBM80 nicht vorhanden
,fd0c ca        dex             Dekrementierzähler verringern (auf nächstes Byte stellen)
,fd0d d0 f5      bne fd04         noch nicht heruntergezählt (Z=0): nächstes Byte vergleichen
,fd0f 60        →rts            Rücksprung von Routine (Z=0: nicht gefunden; Z=1: gefunden)
```

```
-----
; CBM80-Tabelle für Vergleich (wird nur bei $fd04 verwendet)
```

```
,fd10 c3 c2 cd 38 30      ASCII-Codes von "CBM80"
      C B M 8 0
```

```
-----
; RESTOR: Standard-I/O-Vektoren zurücksetzen
```

```
,fd15 a2 30      ldx #30 <($fd30) LB der Adresse der ROM-Tabelle laden
,fd17 a0 fd      ldy #fd >($fd30) HB der Adresse der ROM-Tabelle laden
,fd19 18        clc             Carry löschen (Flag für "Vektoren nach Tabelle in X/Y setzen")
```

```
; VECTOR-Routine (hierher wird vom Kernal-Einsprung bei $ff8d gesprungen)
```

```
,fd1a 86 c3      stx  c3          LB der Adresse der Initialisierungstabelle setzen
,fd1c 84 c4      sty  c4          HB der Adresse der Initialisierungstabelle setzen
,fd1e a0 1f      ldy  #1f         Anzahl der Initialisierungsbytes in Dekrementierzähler schreiben
,fd20 b9 14 03  →lda 0314,y       Byte aus Inhalt des RAM-Vektors holen
,fd23 b0 02      bcs fd27         Flag für "Vektoren auslesen" (C=1): RAM-Vektor-Inhalt schreiben
,fd25 b1 c3      lda  (c3),y       Vektor-Inhalt aus Initialisierungstabelle entnehmen
,fd27 91 c3      →sta  (c3),y       Byte in Initialisierungstabelle setzen (entweder aus RAM-Vektor ausgelesener Wert
                                   oder Wert, der ohnehin an dieser Stelle in der Initialisierungstabelle steht)
,fd29 99 14 03   sta 0314,y       LB des Inhalts des RAM-Vektors holen
,fd2c 88        dey             Dekrementierzähler verringern
```

```
,fd2d 10 f1  bpl fd20    noch nicht auf $ff heruntergezählt (Z=0): weiter in Initialisierungsschleife
,fd2f 60      rts        Rücksprung von Routine
```

; Initialisierungstabelle für Vektoren \$0314-\$0333

,fd30 31 ea	\$ea31	für CINV	\$0314/\$0315	IRQ-Vektor
,fd32 66 fe	\$fe66	für CBINV	\$0316/\$0317	BREAK-Vektor
,fd34 47 fe	\$fe47	für NMINV	\$0318/\$0319	NMI-Vektor
,fd36 4a f3	\$f34a	für IOPEN	\$031a/\$031b	OPEN-Vektor
,fd38 91 f2	\$f291	für ICLOSE	\$031c/\$031d	CLOSE-Vektor
,fd3a 0e f2	\$f20e	für ICHKIN	\$031e/\$031f	CHKIN-Vektor
,fd3c 50 f2	\$f250	für ICKOUT	\$0320/\$0321	CKOUT-Vektor
,fd3e 33 f3	\$f333	für ICLRCH	\$0322/\$0323	CLRCH-Vektor
,fd40 57 f1	\$f157	für IBASIN	\$0324/\$0325	BASIN-Vektor
,fd42 ca f1	\$f1ca	für IBSOUT	\$0326/\$0327	BSOUT-Vektor
,fd44 ed f6	\$f6ed	für ISTOP	\$0328/\$0329	STOP-Vektor
,fd46 3e f1	\$f13e	für IGETIN	\$032a/\$032b	GETIN-Vektor
,fd48 2f f3	\$f32f	für ICLALL	\$032c/\$032d	CLALL-Vektor
,fd4a 66 fe	\$fe66	für USRCMD	\$032e/\$032f	Vektor für freie Benutzung
,fd4c a5 f4	\$f4a5	für ILOAD	\$0330/\$0331	LOAD-Vektor
,fd4e ed f5	\$f5ed	für ISAVE	\$0332/\$0333	SAVE-Vektor

; RAMTAS-Routine (hierher wird vom Kernall-Einsprung bei \$ff87 gesprungen)

,fd50 a9 00	lda #00	Löschwert für Bereich \$0002-\$03ff laden	
,fd52 a8	tay "ldy #00"	Initialisierungswert in Offset-Register laden	
,fd53 99 02 00	sta 0002,y	Byte im Bereich \$0002-\$0101 löschen	
,fd56 99 00 02	sta 0200,y	Byte im Bereich \$0200-\$02ff löschen	
,fd59 99 00 03	sta 0300,y	Byte im Bereich \$0300-\$03ff löschen	
,fd5c c8	iny	Offset erhöhen (auf nächstes Byte richten)	
,fd5d d0 f4	bne fd53	noch nicht auf \$00 heraufgezählt (Z=0): weiter löschen	
,fd5f a2 3c	ldx #3c <(\$033c)	LB der Adresse des Kassettenpuffers laden	} Kassettenpuffer ab Adresse \$033c positionieren
,fd61 a0 03	ldy #03 >(\$033c)	HB der Adresse des Kassettenpuffers laden	
,fd63 86 b2	stx b2	LB in LB des Hilfszeigers auf den Kassettenpuffer schreiben	
,fd65 84 b3	sty b3	HB in HB des Hilfszeigers auf den Kassettenpuffer schreiben	
,fd67 a8	tay "ldy #00"	Initialisierungswert in Offset-Register laden	
,fd68 a9 03	lda #03 >(\$0300)	HB der niedrigsten auf ROM zu prüfenden Adresse - 1 setzen	
,fd6a 85 c2	sta c2	und in HB des Hilfszeigers \$c1/\$c2 schreiben (LB = 0 seit \$fd50-\$fd5d)	


```

,fd6c e6 c2 → inc c2          HB erhöhen
,fd6e b1 c1 → lda (c1),y      Byte an zu prüfender Adresse holen
,fd70 aa      tax            und bis $fd80 in X-Register merken
,fd71 a9 55    lda #55 %01010101 Prüf-Bitmuster laden
,fd73 91 c1    sta (c1),y     und an Prüf-Adresse schreiben
,fd75 d1 c1    cmp (c1),y     wurde Byte wirklich geschrieben?
,fd77 d0 0f    bne fd88       nein (Z=0): ROM gefunden
,fd79 2a      rol            anderes Bitmuster hervorrufen
,fd7a 91 c1    sta (c1),y     und an Prüf-Adresse schreiben
,fd7c d1 c1    cmp (c1),y     wurde auch dieses Byte wirklich geschrieben?
,fd7e d0 08    bne fd88       nein (Z=0): ROM gefunden
,fd80 8a      txa            bei $fd6e/$fd70 gemerkten Ursprungsinhalt der Adresse in Akku bringen
,fd81 91 c1    sta (c1),y     und im Speicher wiederherstellen
,fd83 c8      iny            Offset erhöhen
,fd84 d0 e8    bne fd6e       noch nicht auf $000 heraufgezählt (Z=0): weiter mit unverändertem Zeiger, aber neuem
                                Offset in Y
,fd86 f0 e4    beq fd6c "jmp"  HB erhöhen, wieder mit Offset $000 starten

```

; ROM gefunden

```

,fd88 98      → tya          Offset als LB der ersten ROM-Adresse in Akku
,fd89 aa      tax            und ins X-Register
,fd8a a4 c2    ldy c2        HB des Hilfszeigers als HB der ersten ROM-Adresse laden
,fd8c 18      clc            Carry löschen (Flag für "Speichergrenze setzen")
,fd8d 20 2d fe jsr fe2d "memtop" erste ROM-Adresse als Speicher-Obergrenze setzen
,fd90 a9 08    lda #08 >($0800) HB der Anfangsadresse des Basic-RAM laden
,fd92 8d 82 02 sta 0282      und in HB des Zeigers MEMSTR schreiben
,fd95 a9 04    lda #04 >($0400) HB der Anfangsadresse des Bildschirmspeichers laden
,fd97 8d 88 02 sta 0288      und in Zeiger HIBASE schreiben
,fd9a 60      rts            Rücksprung von Routine

```

} Anfangsadresse des
Basic-RAM und des
Bildschirmspeichers
setzen

; ROM-Tabelle der IRQ-Routinen; Offset wird von \$fd9b-8 berechnet!

```

:fd9b 6a fc    $fc6a (Offset $08 = #08): WRTZ (Synchronisation auf Kassette schreiben)
:fd9d cd fb    $fbcd (Offset $0a = #10): WRTN (Datenfile auf Kassette schreiben)
:fd9f 31 ea    $ea31 (Offset $0c = #12): NIRQ (normaler IRQ, also ohne Kassettenbetrieb)
:fdal 2c f9    $f92c (Offset $0e = #14): READ (Datenfile von Kassette lesen)

```

; IOINIT-Routine (hierher wird vom Kernal-Einsprung bei \$ff84 gesprungen)

,fda3	a9 7f	lda #7f %01111111	Wert für "no irq enabled" laden	} Interrupt-Steueregister von CIA 1 und CIA 2 initialisieren
,fda5	8d 0d dc	sta dc0d	in ICR (Interrupt Control Register) von CIA 1 schreiben	
,fda8	8d 0d dd	sta dd0d	in ICR (Interrupt Control Register) von CIA 2 schreiben	
,fdab	8d 00 dc	sta dc00	Tastaturabfrage von Spalte #7 auslösen	
,fdae	a9 08	lda #08 %00001000	60 Hz an Time Of Day, Serial Port Input, Timer A zählt Phi2-Pulse, Timer "one shot", Timer "pulse", PB6 "normal operation", Timer "stop"	
,fdb0	8d 0e dc	sta dc0e	in CRA (Control Register A) von CIA 1 schreiben	
,fdb3	8d 0e dd	sta dd0e	in CRA (Control Register A) von CIA 2 schreiben	

; \$08 (%00001000) bedeutet für CRB: Write Time Of Day (Uhr stellen), Timer B zählt Phi2-Pulse, Timer "one shot", Timer "pulse", PB7 "normal operation", Timer B "stop"

,fdb6	8d 0f dc	sta dc0f	in CRB (Control Register B) von CIA 1 schreiben
,fdb9	8d 0f dd	sta dd0f	in CRB (Control Register B) von CIA 2 schreiben
,fdbc	a2 00	ldx #00 %00000000	Eingabe von allen Tastatur-Reihen
,fdbe	8e 03 dc	stx dc03	in Datenrichtungsregister B von CIA 1 schreiben
,fdc1	8e 03 dd	stx dd03	in Datenrichtungsregister B von CIA 2 schreiben
,fdc4	8e 18 d4	stx d418	SID-Register #24 (Lautstärke) auf 0 setzen
,fdc7	ca	dex "ldx #\$ff %11111111"	Eingabe von allen Tastatur-Spalten
,fdc8	8e 02 dc	stx dc02	in Datenrichtungsregister A von CIA 1 schreiben
,fdbb	a9 07	lda #07 %00000111	b7=0 (DATA Input Serial Device), b6=0 (CLOCK Input Serial Device), b5=0 (DATA Output Serial Device), b4=0 (CLOCK Output Serial Device), b3=0 (ATN Output Serial Device), b2=1 (TXD Transmit Data), b0=0 und b1=0 (Adreßbits 14/15 für VIC-RAM-Bereich)
,fddc	8d 00 dd	sta dd00	in Datenport A von CIA 2 schreiben
,fdd0	a9 3f	lda #3f %00111111	b7=0 (DATA Input Serial Device), b6=0 (CLOCK Input Serial Device), b5=1 (DATA Output Serial Device), b4=1 (CLOCK Output Serial Device), b3=1 (ATN Output Serial Device), b2=1 (TXD Output Transmit Data), b0=1 und b1=1 (Adreßbits 14/15 für VIC-RAM-Bereich)
,fdd2	8d 02 dd	sta dd02	in Datenrichtungsregister A von CIA 2 schreiben
,fdd5	a9 e7	lda #e7 %11100111	b7=1, b6=1 und b4=1 (unbenutzt), b5=1 (Kassettenmotor aus), b3=0 (Kassettenausgabe), b2=1 (I/O-ROM \$d000-\$dfff ein), b1=1 (Kernal-ROM \$e000-\$ffff ein), b0=1 (Basic-ROM \$a000-\$bfff ein)
,fdd7	85 01	sta 01	in Prozessorport schreiben
,fdd9	a9 2f	lda #2f %00101111	unbenutzte b6 und b7 sowie Rekorder-PLAY-Flag b4 auf "input"; b5 (Steuerung des Kassettenmotors), b3 (Kassettenausgabe), b0-b2 (ROM-Auswahl) auf "output"
,fddb	85 00	sta 00	und in Datenrichtungsregister für Prozessorport schreiben
,fddd	ad a6 02	lda 02a6	PAL/NTSC-Flag holen
,fde0	f0 0a	beq fdec	NTSC (Z=1): Timer für NTSC-Version des C 64 initialisieren

; Timer für PAL-Version des C 64 initialisieren

,fde2	a9 25	lda #25	Initialisierungswert für LB des Timers der PAL-Version laden
,fde4	8d 04 dc	sta dc04	und in LB von Timer A in CIA 1 schreiben
,fde7	a9 40	lda #40	Initialisierungswert für HB des Timers der PAL-Version laden
,fde9	4c f3 fd	jmp fdf3	HB setzen und Timer für IRQ initialisieren

; Timer für NTSC-Version des C 64 initialisieren

,fdec	a9 95	→ lda #95	Initialisierungswert für LB des Timers der NTSC-Version laden
,fdee	8d 04 dc	sta dc04	und in LB von Timer A in CIA 1 schreiben
,fdf1	a9 42	lda #42	Initialisierungswert für HB des Timers der NTSC-Version laden
,fdf3	8d 05 dc	sta dc05	und in HB von Timer A in CIA 1 schreiben
,fdf6	4c 6e ff	jmp ff6e	Timer für IRQ initialisieren

; SETNAM-Routine (hierher wird vom Kernall-Einsprung bei \$ffbd gesprungen)

,fdf9	85 b7	sta b7	Länge des Filenamens in FNLEN setzen
,fdfb	86 bb	stx bb	LB der Adresse des Filenamens in LB von FNADR-Zeiger setzen
,fdfd	84 bc	sty bc	HB der Adresse des Filenamens in HB von FNADR-Zeiger setzen
,fdff	60	rts	Rücksprung von Routine

; SETLFS-Routine (hierher wird vom Kernall-Einsprung bei \$ffbba gesprungen)

,fe00	85 b8	sta b8	Filennummer in LA-Hilfsspeicher setzen
,fe02	86 ba	stx ba	Geräteadresse in FA-Hilfsspeicher setzen
,fe04	84 b9	sty b9	Sekundäradresse in SA-Hilfsspeicher setzen
,fe06	60	rts	Rücksprung von Routine

; READST-Routine (hierher wird vom Kernall-Einsprung bei \$ffb7 gesprungen)

,fe07	a5 ba	lda ba	aktuelle Geräteadresse (FA) holen
,fe09	c9 02	cmp #02	Vergleich mit Geräteadresse von RS232
,fe0b	d0 0d	bne fela	keine Übereinstimmung (Z=0): RS232-Sonderbehandlung überspringen

; Sonderbehandlung: READST für RS232

```
,fe0d ad 97 02 lda 0297      RS232-Statusbyte (RSSTAT) auslesen
,fel0 48          pha        und bis $fel6 auf den Stapel retten
,fel1 a9 00      lda #00     Löschwert für Statusbyte laden
,fel3 8d 97 02 sta 0297     und in RS232-Statusbyte (RSSTAT) schreiben
,fel6 68          pla        bei $fel0 gemerkten RS232-Status wieder vom Stapel holen
,fel7 60          rts        Rücksprung von Routine
```

; SETMSG-Einsprung: Flag für Systemmeldungen (MSGFLG) setzen
(hierher wird vom Kernall-Einsprung bei \$ff90 gesprungen)

```
,fel8 85 9d      sta 9d      in Akku übergebenes Flag in MSGFLG schreiben
```

; Sonderbehandlung: READST für anderes Gerät als RS232

```
,fela a5 90 → lda 90      alten Inhalt des Statusbyte holen
```

; ERSTAT-Routine: Fehlerbits aus Akku in Statusbyte einblenden

```
,felc 05 90      ora 90      Akku-Bits in Statusbyte des Kernall einblenden
,fele 85 90      sta 90      und Ergebnis als neues Statusbyte setzen
,fe20 60          rts        Rücksprung von Routine
```

; SETTMO-Routine (hierher wird vom Kernall-Einsprung bei \$ffa2 verzweigt)

```
,fe21 8d 85 02 sta 0285     Akku in Time-Out-Flag TIMOUT (für IEC-Bus) schreiben
,fe24 60          rts        Rücksprung von Routine
```

; MEMTOP-Routine (hierher wird vom Kernall-Einsprung bei \$ff99 verzweigt)

```
,fe25 90 06      bcc fe2d     Flag für "Werte setzen" (C=0): Speicherobergrenze setzen
```

; MEMTOP-Behandlung: Speicherobergrenze auslesen (C=1)

```
,fe27 ae 83 02 ldx 0283     LB aus Zeiger MEMSIZ entnehmen
,fe2a ac 84 02 ldy 0284     HB aus Zeiger MEMSIZ entnehmen
```


; MEMTOP-Behandlung: Speicherobergrenze setzen (C=0)

```
,fe2d 8e 83 02 → stx 0283    LB in Zeiger MEMSIZ schreiben
,fe30 8c 84 02 sty 0284    HB in Zeiger MEMSIZ schreiben
,fe33 60      rts          Rücksprung von Routine
```

; MEMBOT-Routine (hierher wird vom Kernall-Einsprung bei \$ff9c verzweigt)

```
,fe34 90 06 — bcc fe3c      Flag für "Werte setzen" (C=0): Speicheruntergrenze setzen
```

; MEMBOT-Behandlung: Speicheruntergrenze auslesen (C=1)

```
,fe36 ae 81 02 ldx 0281    LB aus Zeiger MEMSTR entnehmen
,fe39 ac 82 02 ldy 0282    HB aus Zeiger MEMSTR entnehmen
```

; MEMBOT-Behandlung: Speicheruntergrenze setzen (C=0)

```
,fe3c 8e 81 02 → stx 0281    LB in Zeiger MEMSTR schreiben
,fe3f 8c 82 02 sty 0282    HB in Zeiger MEMSTR schreiben
,fe42 60      rts          Rücksprung von Routine
```

; NMI-Routine (hierher weist der ROM-Vektor \$fffa/\$fffb)

```
,fe43 78      sei          Interrupt verhindern
,fe44 6c 18 03 jmp(0318)    Sprung über RAM-Vektor für NMI; normalerweise nach $fe47
```

; reguläre NMI-Routine

```
,fe47 48      pha          Akku retten
,fe48 8a      txa          X-Register in Akku
,fe49 48      pha          und von dort aus retten
,fe4a 98      tya          Y-Register in Akku
,fe4b 48      pha          und von dort aus retten
,fe4c a9 7f   lda #7f %01111111 Wert für "no irq enabled" laden
,fe4e 8d 0d dd sta dd0d    und in ICR (Interrupt Control Register) von CIA 2 schreiben
,fe51 ac 0d dd ldy dd0d    ICR von CIA 2 auslesen, um IRQs zuzulassen
,fe54 30 1c — bmi fe72     b7=1, kann nur RS232-NMI sein (N=1): Sonderbehandlung für RS232 anspringen
,fe56 20 02 fd jsr fd02 "chkcbm" auf CBM80-Markierung ab $8003 testen
```

```
,fe59 d0 03 | bne fe5e      CBM80-Markierung nicht gefunden (Z=0): nicht benutzerdefinierten NMI anspringen
,fe5b 6c 02 80 | jmp(8002)     neudefinierte NMI-Routine anspringen
```

```
-----
,fe5e 20 bc f6 |>jsr f6bc      in UDTIM-Routine einsteigen, so daß STOP-Abfrage möglich wird
,fe61 20 e1 ff | jsr ffel "stop" STOP-Taste abfragen
,fe64 d0 0c | bne fe72      nicht gedrückt (Z=0): Sonderbehandlung für RS232-NMI
```

; Behandlung von <RUN/STOP>+<RESTORE>

```
,fe66 20 15 fd | jsr fd15 "restor" Standard-I/O-Vektoren zurücksetzen
,fe69 20 a3 fd | jsr fda3 "ioinit" CIA-Register initialisieren
,fe6c 20 18 e5 | jsr e518 "intscr" Bildschirm initialisieren
,fe6f 6c 02 a0 | jmp(a002)      Sprung über ROM-Warmstart-Vektor des Basic-Interpreters nach $e37b
```

; Sonderbehandlung für RS232-NMI

```
,fe72 98 |>tya          Inhalt des Interrupt Control Register von CIA 1 in Akku (s. $fe51)
,fe73 2d a1 02 | and 02a1      UND-Verknüpfung mit ENABL (RS232-NMI-Flag)
,fe76 aa | tax          Ergebnis bis $fe8b oder $fea3 in X-Register merken
,fe77 29 01 | and #01 %00000001 b0 (zuständig für Timer-A-Unterlauf; RS232-Senden) aussondern
,fe79 f0-28 | beq fea3      RS232-Eingabe (Z=1): zur Sonderbehandlung
```

; Sonderbehandlung: RS232-Ausgabe

```
,fe7b ad 00 dd | lda dd00      Datenport A von CIA 2 auslesen
,fe7e 29 fb | and #fb %11111011 b2 (zuständig für TXS Transmit Data) löschen
,fe80 05 b5 | ora b5        andere Bits mit nächstem Ausgabebit für RS232 (NXTBIT) verknüpfen
,fe82 8d 00 dd | sta dd00      und Datenport A neu beschreiben
,fe85 ad a1 02 | lda 02a1      ENABL (RS232-NMI-Flag) holen
,fe88 8d 0d dd | sta dd0d      und in ICR (Interrupt Control Register) von CIA 2 schreiben
,fe8b 8a | txa          bei $fe76 gemerktes Verknüpfungsergebnis wieder holen
,fe8c 29 12 | and #12 %00010010 alle Bits bis auf b4 (Flag für IRQ bei RS232-Eingabe) und b1 (Timer-B-Unterlauf; RS232-Eingabe) löschen
,fe8e f0 0d | beq fe9d      kein RS232-Empfang (Z=1): Sonderbehandlung überspringen
,fe90 29 02 | and #02 %00000010 alle Bits bis auf b1 (Timer-B-Unterlauf; RS232-Eingabe) löschen
,fe92 f0 06 | beq fe9a      kein Timer-B-Unterlauf (Z=1): Ausgabe-NMI-Routine und Ende
,fe94 20 d6 fe | jsr fed6      NMI-Routine für RS232-Eingabeflag aufrufen
,fe97 4c 9d fe | jmp fe9d      Ausgabe-Teilroutine des NMI aufrufen und Rücksprung von RS232-NMI
-----
,fe9a 20 07 ff |>jsr ff07      NMI-Routine für RS232-Ausgabe aufrufen
```

```

,fe9d 20 bb lee→jsr eebb      Ausgabe-Teilroutine des NMI bei RS232-Betrieb ausführen
,fea0 4c b6 fe jmp feb6      vom RS232-NMI zurückkehren
-----

; Sonderbehandlung: RS232-Eingabe

,fea3 8a →txa                bei $fe76 gemerktes Verknüpfungsergebnis wieder holen
,fea4 29 02 and #02 %00000010 alle Bits bis auf bl (Timer-B-Unterlauf; RS232-Eingabe) löschen
,fea6 f0 06 beq feae         kein Timer-B-Unterlauf (Z=1): Sonderbehandlung überspringen
,fea8 20 d6 fe jsr fed6      NMI-Routine für RS232-Eingabeflag aufrufen
,feab 4c b6 fe jmp feb6      vom RS232-NMI zurückkehren
-----

,feae 8a →txa                bei $fe76 gemerktes Verknüpfungsergebnis wieder holen
,feaf 29 10 and #10 %00010000 alle Bits bis auf b4 (Flag für IRQ bei RS232-Eingabe) löschen
,febl f0 03 beq feb6         kein RS232-IRQ ermöglicht (Z=1): vom RS232-NMI zurückkehren
,feb3 20 07 ff jsr ff07      NMI-Routine für RS232-Ausgabe aufrufen
-----

; Rückkehr vom RS232-NMI

,feb6 ad al 02 →lda 02a1     ENABL (RS232-NMI-Flag) auslesen
,feb9 8d 0d dd sta dd0d      und in ICR (Interrupt Control Register) von CIA 2 schreiben
,febc 68 pla                 bei $fe4a/$fe4b gemerkten Y-Wert holen      } bei $fe47-$fe4b
,febd a8 tay                 und wieder in Y-Register schreiben        } gemerkte
,febe 68 pla                 bei $fe48/$fe49 gemerkten X-Wert holen      } CPU-Register
,febf aa tax                 und wieder in X-Register schreiben        } (Akku, X und Y)
,fec0 68 pla                 bei $fe47 gemerkten Akku wiederherstellen } wiederherstellen
,fecl 40 rti                 Rücksprung von NMI
-----

```

; Baud-Raten für RS232 bei NTSC-Version:

Da es sich um Timerkonstanten für CIA-Register handelt, stehen niedrigere Werte für geringere Verzögerungen und demzufolge höhere Baud-Raten.

```

:fec2 c1 27 3e la c5 11 74 0e    $27c1 (50 Baud); $1a3e (75 Baud); $11c5 (110 Baud); $0e74 (134.5 Baud)
:feca ed 0c 45 06 f0 02 46 01    $0ced (150 Baud); $0645 (300 Baud); $02f0 (600 Baud); $0146 (1200 Baud)
:fed2 b8 00 71 00                $00b8 (1800 Baud); $0071 (2400 Baud)
-----

```

; NMI-Routine für gesetztes RS232-NMI-Eingabeflag

```

, fed6 ad 01 dd lda dd01      Datenport B von CIA 2 auslesen

```

```
,fed9 29 01 and #01 %00000001 alle Bits bis auf b0 (RD = Received Data) löschen
,fedb 85 a7 sta a7 und b0 als aktuelles Empfangsbit von RS232 (INBIT) setzen
,fedd ad 06 dd lda dd06 LB von Timer B für CIA 2 auslesen
,fee0 e9 lc sbc #lc Anzahl der Korrekturzyklen (28) abziehen
,fee2 6d 99 02 adc 0299 LB der Baud-Rate für RS232-NMI addieren
,fee5 8d 06 dd sta dd06 und LB von Timer B für CIA 2 neu beschreiben
,fee8 ad 07 dd lda dd07 HB von Timer B für CIA 2 auslesen
,feeb 6d 9a 02 adc 029a HB der Baud-Rate für RS232-NMI addieren
,feee 8d 07 dd sta dd07 und HB von Timer B für CIA 2 neu beschreiben
,fef1 a9 11 lda #11 %00010001 "force load" und "start"
,fef3 8d 0f dd sta dd0f in CRB (Control Register B) von CIA 2 schreiben
,fef6 ad a1 02 lda 02a1 ENABL (RS232-NMI-Flag) auslesen
,fef9 8d 0d dd sta dd0d und in ICR (Interrupt Control Register) von CIA 2 schreiben
,feff a9 ff lda #ff %11111111 maximalen Timer-Wert laden
,fefe 8d 06 dd sta dd06 in LB von Timer B für CIA 2 schreiben
,ff01 8d 07 dd sta dd07 in HB von Timer B für CIA 2 schreiben
,ff04 4c 59 ef jmp ef59 "rsrcvr" weiter mit Auswertung des eingelesenen Bit im NMI
```

; NMI-Routine für RS232-Ausgabe

```
,ff07 ad 95 02 lda 0295 LB der Baud-Rate für RS232-NMI holen } Timer-Wert für
,ff0a 8d 06 dd sta dd06 und in LB von Timer B für CIA 2 schreiben } RS232-Baud-Rate
,ff0d ad 96 02 lda 0296 HB der Baud-Rate für RS232-NMI holen } in Timer B von
,ff10 8d 07 dd sta dd07 und in HB von Timer B für CIA 2 schreiben } CIA 2 schreiben
,ff13 a9 11 lda #11 %00010001 "force load" und "start"
,ff15 8d 0f dd sta dd0f in CRB (Control Register B) von CIA 2 schreiben
,ff18 a9 12 lda #12 %00010010 b1 (Timer-B-Unterlauf) und b4 (RS232-Eingabe-IRQ-Flag) laden
,ff1a 4d a1 02 eor 02a1 Bits in ENABL (RS232-NMI-Flag) invertieren
,ff1d 8d a1 02 sta 02a1 und neuen Wert in ENABL (RS232-NMI-Flag) schreiben
,ff20 a9 ff lda #ff %11111111 maximalen Timer-Wert laden
,ff22 8d 06 dd sta dd06 in LB von Timer B für CIA 2 schreiben
,ff25 8d 07 dd sta dd07 in HB von Timer B für CIA 2 schreiben
,ff28 ae 98 02 ldx 0298 Anzahl der RS232-Datenbits (BITNUM) laden
,ff2b 86 a8 stx a8 und in Bitzähler für RS232 (BITCI) schreiben
,ff2d 60 rts Rücksprung von Routine
```

; Fortsetzung von \$f44a: Timer-Verzögerungswert auslesen

```
,ff2e aa tax verdoppeltes LB des Verzögerungswertes bis $ff34 in X-Register merken
```



```
,ff2f ad 96 02 lda 0296      HB des Timer-Verzögerungswertes holen
,ff32 2a      rol            verdoppeln
,ff33 a8      tay            und bis $ff3a in Y-Register merken
,ff34 8a      txa            bei $ff2e gemerkten Wert (doppeltes Verzögerungs-LB) wieder in Akku
,ff35 69 c8   adc #c8        200 addieren (Konstante)
,ff37 8d 99 02 sta 0299      und in LB der RS232-Baud-Rate "full bit time" in Mikro-Sekunden schreiben
,ff3a 98      tya            bei $ff33 gemerktes HB holen
,ff3b 69 00   adc #00        eventuellen Additionsübertrag von $ff35 berücksichtigen
,ff3d 8d 9a 02 sta 029a      und in HB der RS232-Baud-Rate "full bit time" in Mikro-Sekunden schreiben
,ff40 60      rts            Rücksprung von Routine, weiter bei $f44d
```

; Zwei Füllbefehle, die nie angesprungen werden

```
,ff41 ea      nop            Verzögerung um 2 Taktzyklen
,ff42 ea      nop            Verzögerung um 2 Taktzyklen
```

; TPIRQ: IRQ-Einsprung für Datasettenbehandlung (nur von \$f927 aus verwendet)

```
,ff43 08      php            Prozessorstatus zunächst auf Stapel legen
,ff44 68      pla            dann wieder holen
,ff45 29 ef   and #ef %11101111 b4 (BREAK-Flag) löschen
,ff47 48      pha            und zurückschreiben
```

; IRQ-Routine (hierher weist der ROM-Vektor bei \$fffe/\$ffff)

```
,ff48 48      pha            Akku merken
,ff49 8a      txa            X-Register in Akku
,ff4a 48      pha            und von dort aus auf den Stapel legen
,ff4b 98      tya            Y-Register in Akku
,ff4c 48      pha            und von dort aus auf den Stapel legen
,ff4d ba      tsx            Stapelzeiger als Index nach X holen
,ff4e bd 04 01 lda 0104,x     Prozessorstatus vom Stapel holen
,ff51 29 10   and #10 %00010000 alle Bits bis auf b4 löschen, um BREAK-Flag auszusondern
,ff53 f0 03   beq ff58        BREAK-Flag gelöscht (Z=1): System-IRQ auslösen
,ff55 6c 16 03 jmp(0316)      über BREAK-Vektor in BREAK-Routine (normalerweise bei $fe66) springen
-----
,ff58 6c 14 03 jmp(0314)      über IRQ-Vektor in IRQ-Routine (normalerweise bei $ea31) springen
-----
```

; CINT-Routine (hierher wird vom Kernal-Einsprung bei \$ff81 gesprungen)

```
,ff5b 20 18 e5 jsr e518 "intscr" Bildschirm initialisieren
,ff5e ad 12 d0  >lda d012      VIC-Register #18 (aktuelle Rasterzeile) auslesen
,ff61 d0 fb    | bne ff5e      keine Rasterzeile abgeschlossen (Z=0): warten, bis $d012 auf $00 steht
,ff63 ad 19 d0 lda d019      VIC-Register #25 auslesen
,ff66 29 01    and #01 %00000001 alle Bits bis auf b0 (zuständig für PAL/NTSC-Unterscheidung) löschen
,ff68 8d a6 02 sta 02a6      und in PAL/NTSC-Flag (0=NTSC/1=PAL) schreiben
,ff6b 4c dd fd jmp fddd      in IOINIT einsteigen: Timer für PAL- oder NTSC-Version initialisieren
```

; Fortsetzung von \$fdf6: Timer für IRQ initialisieren

```
,ff6e a9 81    lda #81 %10000001 b7=1 (IRQ enabled), b0=1 (Timer-A-Unterlauf als IRQ-Quelle)
,ff70 8d 0d dc sta dc0d      und in ICR (Interrupt Control Register) von CIA 1 schreiben
,ff73 ad 0e dc lda dc0e      CRA (Control Register A) von CIA 1 auslesen
,ff76 29 80    and #80 %10000000 alle Bits bis auf b7 (Uhr mit 50 Hz oder 60 Hz laden) löschen
,ff78 09 11    ora #11 %00010001 b0 ("start") und b4 ("force load") setzen
,ff7a 8d 0e dc sta dc0e      und in CRA (Control Register A) von CIA 1 zurückschreiben
,ff7d 4c 8e ee jmp ee8e "clcklo" CLOCK auf LOW setzen
```

; 1 Füllbyte (wird nie angesprungen)

```
,ff80 03      ***      kann auch anderes Byte wie z.B. $00 sein (hängt von C 64-Version ab)
```

; Kernal-Sprungtabelle

```
,ff81 4c 5b ff jmp ff5b "cint"  CINT:  Bildschirm-Editor-Initialisierung
,ff84 4c a3 fd jmp fda3 "ioinit" IOINIT: Eingabe/Ausgabe-Initialisierung (der CIA-Register)
,ff87 4c 50 fd jmp fd50 "ramtas" RAMTAS: RAM initialisieren, Kassettenpuffer einrichten, Bildschirm auf $0400 richten
,ff8a 4c 15 fd jmp fdl5 "restor" RESTOR: Standard-I/O-Vektoren zurücksetzen
,ff8d 4c 1a fd jmp fd1a "vector" VECTOR: RAM-Vektoren setzen/auslesen
,ff90 4c 18 fe jmp fel8 "setmsg" SETMSG: Flag für Systemmeldungen (MSGFLG) setzen
```

```

,ffa3 4c b9 ed  jmp edb9 "second" SECOND: Sekundäradresse nach LISTEN senden
-----
,ffa6 4c c7 ed  jmp edc7 "tksa"  TKSA:  Sekundäradresse nach TALK senden
-----
,ffa9 4c 25 fe  jmp fe25 "memtop" MEMTOP: Speicher-Obergrenze für Basic-RAM lesen/setzen
-----
,ffa2 4c 34 fe  jmp fe34 "membot" MEMBOT: Speicher-Untergrenze für Basic-RAM lesen/setzen
-----
,ffa5 4c 87 ea  jmp ea87 "scnkey" SCNKEY: Tastatur-Abfrage (im IRQ)
-----
,ffa8 4c 21 fe  jmp fe21 "settm0" SETTMO: Time-Out-Flag für IEC-Bus setzen
-----
,ffa5 4c 13 ee  jmp eel3 "acptr"  ACPTR:  Byte-Eingabe vom IEC-Bus
-----
,ffa8 4c dd ed  jmp eddd "ciout"  CIOUT:  Byte-Ausgabe auf IEC-Bus (auch IECOUT genannt)
-----
,ffab 4c ef ed  jmp edef "untalk" UNTALK: UNTALK-Signal auf IEC-Bus senden
-----
,ffae 4c fe ed  jmp edfe "unlsn" UNLSN:  UNLISTEN-Signal auf IEC-Bus senden
-----
,ffb1 4c 00c ed  jmp ed0c "listen" LISTEN: LISTEN-Signal auf IEC-Bus senden
-----
,ffb4 4c 09 ed  jmp ed09 "talk"  TALK:  TALK-Signal auf IEC-Bus senden
-----
,ffb7 4c 07 fe  jmp fe07 "readst" READST: Statusbyte des Kernals auslesen
-----
,ffba 4c 00 fe  jmp fe00 "setlfs" SETLFS: File-Spezifikationen (LA, FA und SA) setzen
-----
,ffbd 4c f9 fd  jmp fdf9 "setnam" SETNAM: Filenamen setzen
-----
,ffc0 6c 1a 03  jmp(031a)      OPEN:   File öffnen (normalerweise Sprung nach $f34a)
-----
,ffc3 6c 1c 03  jmp(031c)      CLOSE:  File schließen (normalerweise Sprung nach $f291)
-----
,ffc6 6c 1e 03  jmp(031e)      CHKIN:  Eingabefile setzen (normalerweise Sprung nach $f20e)
-----
,ffc9 6c 20 03  jmp(0320)      CKOUT:  Ausgabefile setzen (normalerweise Sprung nach $f250)
-----
,ffcc 6c 22 03  jmp(0322)      CLRCHN: Standard-I/O setzen (normalerweise Sprung nach $f333)
-----

```

```

,ffcf 6c 24 03 jmp(0324) BASIN: Eingabe vom aktuellen Gerät (normalerweise Sprung nach $f157)
-----
,ffd2 6c 26 03 jmp(0326) BSOUT: Ausgabe auf aktuelles Gerät (normalerweise Sprung nach $f1ca)
-----
,ffd5 4c 9e f4 jmp f49e "load" LOAD: Programm laden oder verifizieren
-----
,ffd8 4c dd f5 jmp f5dd "save" SAVE: Programm speichern
-----
,ffdb 4c e4 f6 jmp f6e4 "settim" SETTIM: Uhrzeit (Systemuhr) setzen
-----
,ffde 4c dd f6 jmp f6dd "rdtim" RDTIM: Uhrzeit (Systemuhr) auslesen
-----
,ffel 6c 28 03 jmp(0328) STOP: Abfrage der STOP-Taste (normalerweise Sprung nach $f6ed)
-----
,ffe4 6c 2a 03 jmp(032a) GETIN: Eingabe vom aktuellen Gerät (normalerweise Sprung nach $f13e)
-----
,ffe7 6c 2c 03 jmp(032c) CLALL: Filetabelle löschen (normalerweise Sprung nach $f32f)
-----
,ffea 4c 9b f6 jmp f69b "udtim" UDTIM: Systemuhr erhöhen, STOP-Tastenabfrage ermöglichen
-----
,ffed 4c 05 e5 jmp e505 "screen" SCREEN: Bildschirmformat ermitteln
-----
,fff0 4c 0a e5 jmp e50a "plot" PLOT: Cursorposition lesen/setzen
-----
,fff3 4c 00 e5 jmp e500 "iobase" IOBASE: CIA-Basisadresse ermitteln
-----

```

; 4 Füllbytes (werden nie angesprochen)

```

:fff6 52 52 42 59 ASCII-Darstellung von "RRBI"
-----

```

; ROM-Vektoren

```

:fffa 43 fe $fe43: NMI-Vektor
:fffc e2 fc $fce2: RESET-Vektor
:fffe 48 ff $ff48: IRQ-Vektor
-----

```


; Änderungen des SX-64-Betriebssystems gegenüber dem Kernal des C 64 "stand alone"
(hervorgehoben: veränderte Bytewerte)

; Einschaltmeldung für MSGNEW-Routine

```
:e473 93 0d                [clr,cr]
:e475 20 20 20 20 20 2a 2a 2a 2a 2a 20 20 53 58 2d 36 34    [5space]*****[2space]SX-64
:e486 20 42 41 53 49 43 20 56 32 2e 30 20 20              [space]BASIC V2.0[2space]
:e493 2a 2a 2a 2a 2a 0d 0d                                *****[2cr]
:e49a 20 36 34 4b 20 52 41 4d 20 53 59 53 54 45 4d 20 20    [space]64K RAM SYSTEM[2space]
:e4ab 00                                                    [nul]
```

; in INTSCR (Bildschirm initialisieren):

```
,e534 a9 06      lda #06          Farbcode für "blau (dunkelblau)" laden
,e536 8d 86 02   sta 0286         und als COLOR (aktuelle Zeichenfarbe setzen)
```

; in Tastatur-Eingabeschleife:

; Sonderbehandlung für <SHIFT>+<RUN/STOP>:

Simulation der Eingabe von LOAD":*",8[CR]RUN[CR]

```
,e5ee a2 0f      ldx #0f          Dekrementierzähler (Anzahl der zu schreibenden Tasten-ASCII-Codes) laden
,e5f0 78         sei             Interrupt verhindern, damit im Interrupt ablaufende Tastaturabfrage nicht stört
,e5f1 86 c6      stx  c6          NDX (Anzahl der Zeichen im Tastaturpuffer) mit Anzahl der ASCII-Codes belegen
,e5f3 bd d7 f0   →lda f0d7,x      ASCII-Code aus ROM-Tabelle von LOAD":*",8[CR]RUN[CR] holen
,e5f6 9d 76 02   sta 0276,x       und in Tastaturpuffer schreiben
,e5f9 ca         dex             Dekrementierzähler verringern
,e5fa d0 f7     bne e5f3          noch nicht auf 0 heruntergezählt (Z=0): Fortsetzung der Kopierschleife
,e5fc f0 cf     beq e5cd "jmp"     zurück zum Anfang der Tastatur-Eingabeschleife
```

; Initialisierungswerte für VIC-Register:

```
:ecd9 03                Initialisierungswert für VIC-Register #32 (Rahmenfarbe = 3)
```

```
:ecda 01                Initialisierungswert für VIC-Register #33 (Hintergrundfarbe #0 = 1)
```

; in der Tabelle der Systemmeldungen wurde anstelle von "press play on tape" (mangels SX-64-Kassettenanschluß nicht mehr benötigt) der Text nach Drücken von <SHIFT>+<RUN/STOP> abgelegt:

```
:f0d8 4c 4f 41 44 22 3a 2a 22 2c 38 0d 52 55 4e 0d      LOAD":*",8[CR]RUN[CR]
```

; in OPEN-Routine:

```
,f384 c9 02      cmp #02          Gerät #2 (RS232) aktiv?
,f386 d0 08      bne f390          nein (Z=0): I/O ERROR #9 (ILLEGAL DEVICE NUMBER), da Datasette angefordert wurde
...
,f390 4c 13 f7    jmp f713 "ioerr9" I/O ERROR #9 ("illegal device number") auslösen
```

; in LOAD/VERIFY-Routine:

```
,f4b2 c9 03      cmp #03          Vergleich der Gerätenummer mit der Geräteadresse des Bildschirms
,f4b4 f0 f9      beq f4af          Übereinstimmung (Z=1): I/O ERROR #9 (ILLEGAL DEVICE NUMBER) auslösen
,f4b6 90 f7      bcc f4af          Gerätenummer #1 oder #2 (C=0): RS232 oder Datasette, also ebenfalls I/O ERROR #9
```

; in SAVE-Routine:

```
,f5f4 c9 03      cmp #03          Vergleich der Gerätenummer mit der Geräteadresse des Bildschirms
,f5f6 f0 f9      beq f5f1          Übereinstimmung (Z=1): I/O ERROR #9 (ILLEGAL DEVICE NUMBER) auslösen
,f5f8 90 f7      bcc f5f1          Gerätenummer #1 oder #2 (C=0): RS232 oder Datasette, also ebenfalls I/O ERROR #9
```

Kapitel 2

So verwendet man das ROM-Listing

Lassen Sie mich dieses Kapitel mit einer recht amüsanten Frage beginnen:

»Was bewirkt ein ROM-Listing und wie gibt man es ein, oder ist es auch auf Diskette erhältlich?«

So ein Leser an eine bekannte Computerzeitschrift. Sicher lächeln Sie erhaben über derartige Einsteigerfragen, aber dennoch wird dieses Kapitel versuchen, so grundlegend wie möglich zu erklären, was ein ROM-Listing ist und wie man den größten Nutzen daraus zieht.

Zunächst zur Begriffsdefinition. Ein ROM-Listing ist die Darstellung aller Daten im ROM eines Computers, also der Firmware. Diese Begriffe (»ROM«, »Firmware« usw.) werden allesamt in Kapitel 3 erläutert. Durch folgende Eingabe haben Sie im weitesten Sinne auch ein ROM-Listing, das auf den Bildschirm ausgegeben wird:

```
FOR I=40960 TO 49151:PRINT PEEK(I),:NEXT
```

Diese Zeile gibt alle Bytes im Basic-ROM als dezimale Bytewerte aus. Mit diesen Zahlen läßt sich jedoch nicht viel anfangen. Ein ROM-Listing wird deshalb mit einem Disassembler (Programm, das Bytewerte als Maschinenbefehle auflistet) erstellt. Manche ROM-Listings jedoch sind Assembler-Quelltexte, was im Prinzip nur eine höhere Stufe des Disassemblerlistings ist.

Damit ist es jedoch noch nicht getan: Noch fehlen die Kommentare. In einem gedruckten ROM-Listing dienen Kommentare in menschlicher Sprache, die natürlich nicht vom Disassembler stammen, dem besseren Verständnis, denn Maschinenprogramme (wie zum Beispiel das C64-ROM), die sich selbst erklären, sind die noch nicht erfundenen »Wollmilchsäue«.

Wenn Sie einen kurzen Blick in Kapitel 1, das ROM-Listing in diesem Buch, werfen, können Sie die Kombination aus Disassemblerlisting und Kommentaren schnell erkennen: Die disassemblierten Stellen sind zur Abhebung von den dazugehörigen Kommentaren mit einem grauen Raster unterlegt.

Worin liegt nun der Nutzen eines ROM-Listings?

Dies hängt gewissermaßen auch vom Benutzer ab. Auf jeden Fall kann man ein ROM-Listing als Nachschlagewerk verwenden, wenn

man die genaue Funktionsweise irgendeiner ROM-Routine oder die Bedeutung irgendeines Hilfsspeichers ermitteln will. Dadurch hat man den exakten Überblick über alle Operationen, die der Computer aufgrund seiner im ROM gespeicherten Programme ausführt. Vom Blinken des Cursors über die Wirkungsweise eines Basic-Befehls bis zu einer fehleranfälligen Stelle im Interpreter lassen sich mit dem ROM-Listing die meisten Phänomene erklären.

Diese Form der Anwendung des ROM-Listings ist sicherlich die häufigste, und für einen »Vollprofi« genügt dies bereits. Als Semi-profi oder Fortgeschrittener, wie Sie sich eben bezeichnen wollen, sollten Sie sich jedoch nicht die einmalige Chance entgehen lassen, aus dem ROM-Listing vieles über das Programmieren an sich zu lernen. Dies wird oft übersehen, und die konventionellen ROM-Listings sehen dies auch nicht vor. Das vorliegende Buch jedoch möchte Ihnen auch anhand des C64-ROM demonstrieren, wie in Maschinensprache programmiert wird. Bekanntlich lassen sich Kenntnisse über das Programmieren im allgemeinen und die Maschinensprache im besonderen vor allem aus Beispielen gewinnen oder vertiefen. Ein praktischeres Lernobjekt als das ROM des Computers gibt es jedoch kaum, denn mit Betriebssystem und Basic-Interpreter hat man in der C64-Anwendung unbestritten den meisten Kontakt; daher ist es doch nur normal, sich auch einmal dafür zu interessieren, wie diese beiden Programme eigentlich realisiert wurden.

Als erwünschter Nebeneffekt tritt das bessere Verständnis des Systems auf, was Ihnen wiederum eine effizientere Programmierung des C64 ermöglicht.

Die beiden genannten Aufgaben erfüllt das ROM-Listing am besten in Verbindung mit Kapitel 4, wo jede ROM-Routine oder -Tabelle ein weiteres Mal erläutert wird; allerdings ist dabei der Bezug auf die einzelnen Befehle nicht so stark, während mehr auf die größeren Zusammenhänge, die Anwendung der Routinen oder auf Besonderheiten (wie die Programmierfehler im ROM) eingegangen wird.

Das ROM-Listing selbst ist nun eine wirkliche Besonderheit: Kein anderes ROM-Listing zum C64 oder irgendeinem anderen Computer ist auch nur annähernd so ausführlich! Daraus hat sich auch das ungewöhnlich breite Buchformat entwickelt (siehe Vor-

wort), um möglichst viele Kommentare geben zu können. Dasselbe wie über den Inhalt läßt sich guten Gewissens über die äußere Aufmachung sagen; schlagen Sie wahllos eine Seite auf, und vergleichen Sie diese mit einem anderen ROM-Listing – es ist, wie wenn man von einem Zeitalter ins andere gerät.

Nun genug der überschwenglichen Beschreibung; dieses Kapitel will Ihnen helfen, möglichst alle »special features« des ROM-Listings auszunutzen.

2.1 Symbole

Die Aufteilung in Disassemblerlisting und Kommentare wurde bereits erwähnt. Dieser Abschnitt beschreibt die Symbolik des ROM-Listings, die sich durch den logischen Aufbau und die praxisorientierte Aufmachung auszeichnet.

2.1.1 Geschweifte Klammern

Normalerweise bezieht sich, wie Sie schon wissen, ein Kommentar immer auf den vor ihm stehenden Befehl (»line by line«-Dokumentation). Dies erlaubt zwar größtmögliche Detailtreue, aber es geht unweigerlich die Übersichtlichkeit verloren, vor allem bei Programmschleifen und -blöcken. Deshalb wird im ROM-Listing wiederholt rechts von den Kommentaren zu einzelnen Befehlen hinter einer geschweiften Klammer eine Zusammenfassung angeboten. Eine geschweifte Klammer umfaßt somit alle Zeilen, die rechts von ihr erläutert werden.

An manchen Stellen treten auch zwei oder sogar drei geschweifte Klammern hintereinander auf, wobei sich dann eine weiter rechts stehende Klammer immer auf einen größeren Bereich bezieht und somit automatisch größer ist.

Dank »Splitting« (Aufteilen) des Kommentars durch geschweifte Klammern können Sie das ROM-Listing wahlweise von rechts nach links oder von links nach rechts lesen.

Von rechts nach links kommen Sie von der Adresse, den Op-codes (Bytewerten) über die disassemblierten Befehle und die ersten Kommentare bis zu demjenigen Kommentar, der eine Zusammenfassung mehrerer Befehle bildet.

Von links nach rechts ist es Ihnen möglich, wie mit einer Lupe schrittweise ins Detail vorzustoßen.

Wie gesagt: Die Vorgehensweise hängt vom Einzelfall ab und bleibt immer Ihrer freien Entscheidung überlassen.

2.1.2 Pfeile

Kein Maschinenprogramm kommt ohne Verzweigungen aus. Die meisten Verzweigungen laufen über BRANCH-Befehle (beq, bne,

bcc, bcs, bvc, bvs), wo Ausgangs- und Zielpunkt nicht zu weit auseinanderliegen dürfen (maximal 128 Bytes). Dennoch ist es eine mühselige Angelegenheit, bei jeder Verzweigung zuerst die entsprechende Adresse dem Disassemblerlisting zu entnehmen und sie dann ganz links zu suchen. Mit der größte Clou dieses Buches sind deshalb die Verzweigungspfeile, die von einem BRANCH-Befehl zur angesprungenen Adresse führen. Dadurch erhöht sich auch die Transparenz bestimmter Programmstrukturen.

2.1.3 Waagrechte Linien

In der Regel arbeitet der Prozessor ein Maschinenprogramm byteweise ab, er ackert sich also von einer Adresse im Speicher zur nächsthöheren vor. Bei manchen Befehlen wird aber auf jeden Fall aus dieser Ordnung ausgebrochen und die Programmausführung an anderer Stelle fortgesetzt: JMP, RTS, RTI.

Bei JSR wird die byteweise Ausführung nur zwischenzeitlich unterbrochen, bis das Unterprogramm abgelaufen ist. Die BRANCH-Befehle verzweigen nur bedingt; andernfalls sind sie »Pseudo-JMPs« (2.2) und werden wie JMP gehandhabt.

Unter solchen Befehlen (JMP, RTS, RTI, »jmp«) finden Sie deshalb im ROM-Listing waagrechte Linien, die aus Bindestrichen bestehen. Dies wissen die Anwender des Maschinensprachemonitors SMON (64'er-Magazin) bereits zu schätzen.

Der Vorteil dieser waagrecchten Linien liegt darin, daß man schon beim ersten Blick die klaren Abgrenzungen der einzelnen Programmteile erkennt.

2.2 Aufbau des Disassemblerlistings

Das Disassemblerlisting steht jeweils links in einer Zeile und ist mit einem grauen Raster unterlegt. Ist in einer Zeile kein Disassemblerlisting, sondern nur ein Kommentar vorhanden, so wurde auch kein Raster angebracht.

Die Symbolik zusätzlicher Anmerkungen im Disassemblerlisting ist dabei von den gängigen Assemblern übernommen (<, >, *, %, usw.).

2.2.1 Disassemblerformat

Ich habe das Disassemblerlisting mit dem Programm SMON aus dem 64'er-Magazin erstellt; das Format möchte ich an zwei repräsentativen Beispielen für die beiden grundlegenden Formen des Disassemblerlistings erklären:

Beispiel 1:

```
,aa04 20 ed ba jsr baed
```


Eine Befehlszeile wie diese wird mit einem Komma eingeleitet, auf welches die Basisadresse folgt, ab welcher die rechts davon stehenden Bytes im Speicher befindlich sind. Im Beispiel steht also \$20 in \$aa04, \$ed in \$aa05 und \$ba in \$aa06.

Hinter den Bytes wird erst das drei Zeichen lange Mnemonic, im Beispiel JSR für den Opcode \$20, angegeben. Dessen Parameter (hier: Adresse \$baed) werden nicht durch das Dollarzeichen eingeleitet, sind aber **ausnahmslos** hexadezimale Zahlen!

Beispiel 2:

```
:a004 43 42 4d 42 41 53 49 43
```

Diese Zeile ist ein sogenanntes Memory Dump, also eine Speicherauflistung in Form von Bytewerten. Solche Memory-Dump-Zeilen beginnen mit einem Doppelpunkt und der Basisadresse. Dahinter stehen die Bytewerte, die sich ab der vorangestellten Basisadresse im Speicher befinden, wie Sie es von Beispiel 1 kennen.

2.2.2 Weitere Informationen im Disassemblerlisting

Das Disassemblerformat ist zwar sehr schlüssig und wird deshalb strikt eingehalten; zusätzliche Anmerkungen erleichtern jedoch die Arbeit erheblich, zum Beispiel weil sie Ihnen das Umrechnen vom Hexadezimal- ins Binärsystem ersparen.

2.2.2.1 Andere Zahlenformate

Die Assemblersprache ist zwar byteorientiert, doch oft wird auch mit einzelnen Bits gerechnet. Dazu ist das Hexadezimalsystem nicht so geeignet wie die Binärdarstellung. An den entsprechenden Stellen wird diese Binärangebe deshalb zusätzlich hinter den hexadezimalen Zahlenwert geschrieben; ein Prozentzeichen (%) leitet die binäre Darstellung ein.

2.2.2.2 Low-High-Format

2-Byte-Werte (beispielsweise Adressen) stehen im Low-High-Format im Speicher. Eine Umrechnung in eine vierstellige Hexadezimal wird hinter einem Memory Dump durch ein Dollarzeichen (\$) eingeleitet. Ein Beispiel finden Sie gleich zu Beginn des ROM-Listings:

```
:a000 94 e3 $e394
```

Wird umgekehrterweise mit einem Bytewert operiert, der das Low- oder High-Byte einer vierstelligen Hexadezimalzahl ist, so bezeichnet dahinter

```
<($....)
```

das Low-Byte und

```
($....)
```

das High-Byte. Beispiel aus dem ROM-Listing:

```
,bb01 a9 f9 lda #f9 <($baf9)
,bb03 a0 ba ldy #ba >($baf9)
```

2.2.2.3 Zeropage-Adressen

Wird eine Zeropage-Adresse als Offset oder Adreßbezeichnung geladen, ist sie mit einem Stern (*) markiert:

```
$bbc7 a2 5c ldx #5c *$5c
```

2.2.2.4 ASCII-Codes

Bei vielen ASCII-Tabellen oder Befehle, die mit ASCII-Codes operieren, wird die ASCII-Darstellung hinter den Bytewerten in Anführungszeichen angegeben. In Ausnahmefällen entfällt das Anführungszeichen zugunsten der größeren Übersichtlichkeit und Einfachheit.

2.2.2.5 Anführungszeichen hinter Mnemonics

Wenn Anführungszeichen hinter Mnemonics stehen, schließen sie weitere Informationen ein. Hierfür gibt es mehrere Möglichkeiten, von denen die jeweils zutreffende im Zusammenhang klar erkennbar ist.

Bei Sprung- und Verzweigungsbefehlen liegen meist Label vor. Label sind Ihnen sicher von der Arbeit mit Makroassemblern bekannt. Da das Disassemblerlisting absolute Adressen angibt, werden bei den wichtigen Routinen und Einsprüngen zusätzlich die Label angegeben. Das ROM-Listing ist also in diesem Punkt kompromißlos: Sie genießen als Anwender die Vorzüge der Label (Verständlichkeit, Quelltextbezug) und der absoluten Adressen (leicht im ROM-Listing zu finden) gleichzeitig!

Die andere Bedeutung von Texten in Anführungszeichen hinter den Mnemonics läßt sich durch die Bezeichnung »simulierte Befehle« beschreiben. So sind manche BRANCH-Befehle aufgrund des Zusammenhangs, in welchem sie auftauchen, effektiv JMP-Sprünge, da sie immer verzweigen. Dann steht »jmp« hinter dem BRANCH, um somit einen Pseudo-JMP zu markieren.

Des weiteren gibt es simulierte Subtraktionen, die als Aditionsbefehle auftreten, oder Dekrementierbefehle, die immer dasselbe Ergebnis liefern; dann finden Sie Anmerkungen wie »sbc« oder »ldy #01« hinter »adc . . « oder »dey«.

2.2.2.6 Der Bit-Trick

Einer der ältesten Programmiertricks auf 65xx-Systemen ist der »Trick mit dem BIT«. Da der BIT-Befehl von einer 1 oder 2 Byte langen Adresse (Zeropage- oder absolute Adressierung) gefolgt wird, läßt sich anstelle dieser Adresse ein anderer Maschinenbefehl unterbringen, der durch den vorangestellten BIT-Opcode ignoriert wird, aber im Bedarfsfall von der CPU ausführbar ist.

Ein Beispiel finden Sie bei \$ab3f-\$ab43 im ROM-Listing:

\$ab3f lädt den ASCII-Code von <SPACE>. Bei der anschließenden Abarbeitung von »bit \$1da9« wird der Akkumulator nicht verändert, die Adresse \$1da9 auch nicht. Aber der durch \$a9/\$1d bezeichnete LDA-Befehl zum Laden des CRSR-RIGHT-Codes wird geflissentlich übergangen!

Springt hingegen die CPU gezielt bei \$ab42 ein, findet sie dort den Befehl »lda #1d« vor.

Solche Pseudo-BITs sind in Anführungszeichen gesetzt; hinter ihnen steht dann das disassemblierte Format des verdeckten Befehls.

2.3 Aufbau der Kommentare

Die Kommentare sind weder im Telegrammstil noch als Romane verfaßt. Im wesentlichen handelt es sich um Infinitive (»Flag setzen«) oder verkürzte Sätze, denen – für Grammatikfans – oft das Subjekt fehlt, für welches man »dieser Befehl« einsetzen könnte.

Bei BRANCH-Befehlen habe ich jedoch ein einheitliches Format eingehalten, das sich aus folgenden drei Komponenten zusammensetzt:

- Ganz links steht die Verzweigungsbedingung als Klartext; ist diese erfüllt, verzweigt der entsprechende Befehl.
- In der Mitte finden Sie die eingeklammerte Darstellung der Verzweigungsbedingung, ausgedrückt in Prozessorflag-Zuständen (z.B. C=0). Diese Verzweigungsbedingung geht auch aus dem BRANCH-Befehl selbst hervor:

```
BNE : Z=0
BEQ : Z=1
BCC : C=0
BCS : C=1
BVC : V=0
BVS : V=1
```

- Rechts von der eingeklammerten Verzweigungsbedingung befindet sich, abgegrenzt durch einen Doppelpunkt, eine Erklärung, was im Falle einer Verzweigung geschieht.

In Basic-Syntax ausgedrückt, würde also links vom Doppelpunkt die IF-Bedingung und rechts davon der THEN-Teil stehen.

Eine Besonderheit der Kommentare zum Kernall besteht darin, daß die Hilfsspeicher durch die Label aus Kapitel 6 bezeichnet werden. Dadurch kann statt »standardmäßig vorgesehenes Eingabegerät« schlicht »DFLTN« geschrieben werden, wenn diese Bezeichnung wiederholt auftritt.

Gleichzeitig gewöhnen Sie sich dadurch die Bezeichnung der Hilfsspeicher mit Labeln an, was Ihrer eigenen Programmierung zugute kommt.

2.4 Cross-Reference

Obwohl das ROM-Listing wiederholt darauf hinweist, von wo aus eine bestimmte Routine aufgerufen wird, sei hier eine **vollständige** Cross-Reference über den gesamten C64-Speicher (!) gegeben, die Sie als Ergänzung sowohl zu Kapitel 1 als auch zu Kapitel 6 hinzuziehen können.

In dieser Cross-Reference steht ganz links vor einem Doppelpunkt eine Adresse, die von den rechts angegebenen Adressen angesprochen wird.

Hinter diesen Adressen wiederum finden Sie das Mnemonic des jeweiligen Befehls und dessen Adressierungsart, wenn es sich nicht um einen absolut adressierten Befehl handelt:

```
,X    absolut X-indiziert
,Y    absolut Y-indiziert
ZP    Zeropage
ZP,X  Zeropage, X-indiziert
ZP,Y  Zeropage, Y-indiziert
( )   indirekt
(,X)  X-indiziert, indirekt
( ),Y indirekt, Y-indiziert
```

Zusätzlich gibt es zwei Besonderheiten:

```
.WO    Adreßtable (bitte eventuelle Dekrementierungen beachten)
»bit«  Pseudo-BITs sprechen im Grunde nicht die angegebene
        Adresse an, sondern verstecken einen Assemblerbefehl
```

Hier nun die umfangreichste Cross-Reference, die zum C64 gegeben werden kann:

```

0000:  b4d7 sta zp,x    f5e2 lda zp,x    fddb sta zp

0001:  b4db sta zp,x    b89e lsr zp,x    b8cb lda ,y      b8ce sbc zp,x
      b991 ldy zp,x    b997 sty zp,x    b9a6 asl zp,x    b9aa inc zp,x
      b9ac ror zp,x    b9ae ror zp,x    ea61 lda zp      ea6b lda zp
      ea75 lda zp      ea79 sta zp      f5e6 lda zp,x    f830 bit zp
      f834 bit zp      f8ab lda zp      f8af sta zp      fbbf lda zp
      fbc3 sta zp      fcca lda zp      fcce sta zp      fdd7 sta zp

0002:  b4df sta zp,x    b8c4 lda ,y      b8c7 sbc zp,x    b98d ldy zp,x
      b993 sty zp,x    b9b0 ror zp,x    fd53 sta ,y

0003:  b8bd lda ,y      b8c0 sbc zp,x    b989 ldy zp,x    b98f sty zp,x
      b9b2 ror zp,x    e3dc sta zp

0004:  b8b6 lda ,y      b8b9 sbc zp,x    b985 ldy zp,x    b98b sty zp,x
      b9b4 ror zp,x    e3de sty zp

0005:  e3d4 sta zp

0006:  e3d6 sty zp

0007:  a90b stx zp      a913 ldx zp      a915 sta zp      a975 sta zp
      a997 adc zp      ac61 sta zp      ac65 sta zp      ac6d sta zp
      aff4 sta zp      b00f and zp      b489 stx zp      b49c cmp zp
      bce4 sta zp      bf9c ldy zp      e011 lda zp

0008:  a592 sta zp      a5e3 sta zp      a5ea cmp zp      a90f sty zp
      a911 lda zp      a917 stx zp      a91d cmp zp      ac72 sta zp
      affa sta zp      b006 and zp      b48b stx zp      b4a0 cmp zp

0009:  aafd sty zp      ab0a sbc zp

000a:  e16a sty zp      e16f lda zp      e17a lda zp

000b:  a4a2 sty zp      a4fd adc zp      a51f ldy zp      a5b0 sty zp
      a5c5 ora zp      a5f7 inc zp

000c:  b090 stx zp      b1d1 lda zp      b216 sta zp      b24f lda zp
      b28a bit zp      b2e5 lda zp

000d:  a9b4 lda zp      aab8 bit zp      ac54 bit zp      ad90 bit zp
      ade1 adc zp      ae07 lsr zp      ae88 sta zp      af33 lda zp
      b030 sta zp      b0a1 stx zp      b0c0 sta zp      b1d6 lda zp
      b20f sta zp      b37d lda zp      b393 stx zp      b4ea sty zp
      b787 stx zp

```

000e:	a9b1 lda zp	ac8c lda zp	af5d bit zp	b0a3 stx zp
	b0ce sta zp	b1d3 ora zp	b212 sta zp	
000f:	a580 sty zp	a598 bit zp	a5dc sta zp	a6cb sty zp
	a6fa lda zp	a6fe sta zp	a720 bit zp	b4f4 lsr zp
	b518 lda zp	b521 sta zp		
0010:	a68b sta zp	a744 sta zp	b0c8 lda zp	b0de ora zp
	b0e9 sty zp	b3be sta zp	b3e8 sta zp	
0011:	ab4d lda zp	ac0f sta zp	ac31 bit zp	ac58 bit zp
	ace3 ldx zp			
0012:	ae68 sta zp	b075 and zp	e297 lda zp	e29b sta zp
	e2b9 sta zp	e2d0 lda zp		
0013:	a44c sta zp	aa91 stx zp	aad3 lda zp	aadc bit zp
	ab3b lda zp	ab62 lda zp	ab8d stx zp	aba0 ldx zp
	abad stx zp	abb5 lda zp	abbc stx zp	abd9 lda zp
	abef lda zp	abf9 lda zp	ac43 lda zp	acf0 lda zp
	e380 sta zp	e3f2 sta zp		
0014:	a50d lda zp	a62e lda zp	a6bd lda zp	a6c5 sta zp
	a6e2 cpx zp	a8a9 sbc zp	a96d stx zp	a97f lda zp
	a987 adc zp	a989 sta zp	a991 asl zp	a995 lda zp
	a999 sta zp	b808 sty zp	b810 lda zp	b818 lda (),y
	b81c sta zp	b82a sta (),y	b840 lda (),y	e144 jmp ()
0015:	a50f ldy zp	a623 lda zp	a6bf ora zp	a6c7 sta zp
	a6de cmp zp	a8ad sbc zp	a96f stx zp	a977 lda zp
	a98d adc zp	a98f sta zp	a993 rol zp	a99d inc zp
	b80a sta zp	b80d lda zp	b81f sta zp	
0016:	a67c stx zp	b4ca ldx zp	b4f1 stx zp	b544 cmp zp
	b6e3 sta zp	e400 stx zp		
0017:	b4ec stx zp	b6df cmp zp	b6e7 sta zp	
0018:	b6db cpy zp	e3f4 sta zp	f5a8 bit zp	f68c bit zp
0020:	baf9 sty zp			
0022:	a3c4 sta zp	a3d3 sbc zp	a3de sbc zp	a400 sta zp
	a403 cpx zp	a440 sta zp	a456 lda (),y	a4b1 sta zp
	a4d8 adc zp	a4df lda (),y	a537 sta zp	a53e lda (),y
	a545 lda (),y	a54b adc zp	a550 sta (),y	a557 sta (),y

	a559 stx zp	a784 sta zp	a979 sta zp	a982 rol zp
	a985 rol zp	a98b lda zp	aa1d lda (),y	ab2b lda (),y
	adea sta zp	aded adc zp	ae3a sta zp	ae3c inc zp
	ae55 jmp ()	b2a4 ldy zp	b31e ldy zp	b34c sty zp
	b540 sta zp	b555 sta zp	b57d sta zp	b583 lda (),y
	b587 lda (),y	b58b lda (),y	b592 lda (),y	b59f lda (),y
	b5a6 adc zp	b5a8 sta zp	b5bd lda (),y	b5c2 lda (),y
	b5c7 lda (),y	b5cc lda (),y	b5d0 lda (),y	b5ea lda zp
	b5f9 adc zp	b5fb sta zp	b688 stx zp	b691 lda (),y
	b6aa sta zp	b6b4 lda (),y	b6b8 lda (),y	b6bc lda (),y
	b6d6 stx zp	b71d adc zp	b71f sta zp	b792 lda (),y
	b7bd ldx zp	b7c2 adc zp	ba8c sta zp	ba92 lda (),y
	ba97 lda (),y	ba9c lda (),y	baa1 lda (),y	bab0 lda (),y
	bba2 sta zp	bba8 lda (),y	bbad lda (),y	bbb2 lda (),y
	bbb7 lda (),y	bbc0 lda (),y	bbd7 stx zp	bbdf sta (),y
	bbe4 sta (),y	bbe9 sta (),y	bbf2 sta (),y	bbf7 sta (),y
	e0a1 stx zp	e0a7 lda (),y	e0ac lda (),y	e0b2 lda (),y
	e0b7 lda (),y	e25d ldx zp		
0023:	a445 sta zp	a4ad sta zp	a4dc dec zp	a4e6 inc zp
	a539 sty zp	a552 lda zp	a55b sta zp	a786 sty zp
	ae3f sta zp	b542 stx zp	b557 stx zp	b57f stx zp
	b5ac inc zp	b5ae ldx zp	b5ec ldx zp	b5ff inc zp
	b601 ldx zp	b68a sty zp	b6ac sty zp	b6d8 sty zp
	b723 inc zp	b7c6 ldx zp	ba8e sty zp	bba4 sty zp
	bbd9 sty zp	e0a3 sty zp	e25f ldy zp	
0024:	a4c1 sta zp	a4e1 sta (),y	ad3d sta zp	b7c4 sta zp
	b7d1 lda (),y	b7d5 sta (),y	b7e0 sta (),y	bc5b sta zp
	bc61 lda (),y	bc67 lda (),y	bc71 lda (),y	bc7a lda (),y
	bc81 lda (),y	bc8c lda (),y		
0025:	a4b5 sta zp	a4d5 dec zp	a4e8 inc zp	b7cd stx zp
	bc5d sty zp			
0026:	ba35 sta zp	ba77 lda zp	ba6b sta zp	ba7d ror zp
	bb8f lda zp			
0027:	ba37 sta zp	ba71 lda zp	ba75 sta zp	ba7f ror zp
	bb93 lda zp			
0028:	b337 stx zp	b350 sta zp	b36f adc zp	ba39 sta zp
	ba6b lda zp	ba6f sta zp	ba81 ror zp	bb97 lda zp
0029:	b355 sta zp	b373 adc zp	ba3b sta zp	ba65 lda zp
	ba69 sta zp	ba83 ror zp	bb44 sta zp,x	bb9b lda zp

002b:	a533 lda zp	a613 lda zp	a647 sta (),y	a64a sta (),y
	a64c lda zp	a68f lda zp	a81e lda zp	a8bc lda zp
	e171 ldx zp	e406 stx zp	e419 sta (),y	e41b inc zp
	e422 lda zp	e433 sbc zp		
002c:	a535 ldy zp	a615 ldx zp	a653 lda zp	a695 lda zp
	a822 ldy zp	a8be ldx zp	e173 ldy zp	e408 sty zp
	e41f inc zp	e424 ldy zp	e438 sbc zp	
002d:	a4af lda zp	a4bd adc zp	a4bf sta zp	a4cf sbc zp
	a4f9 lda zp	a51b sta zp	a651 sta zp	a66b lda zp
	aa47 cmp zp	b0eb lda zp	b551 lda zp	e159 ldx zp
	ela7 stx zp			
002e:	a4c3 lda zp	a4c7 sta zp	a501 ldy zp	a51d sty zp
	a657 sta zp	a66d ldy zp	aa3f cpy zp	b0ed ldx zp
	b553 ldx zp	e15b ldy zp	ela9 sty zp	
002f:	a66f sta zp	b0f7 cmp zp	b143 lda zp	b165 sta zp
	b218 ldx zp	b55d cmp zp		
0030:	a671 sty zp	b0f3 cpx zp	b145 ldy zp	b167 sty zp
	b21a lda zp	b559 cpx zp		
0031:	a3bb sta zp	a517 lda zp	a673 sta zp	b14b lda zp
	b224 cpx zp	b2bc sta zp	b2d6 lda zp	b38a sbc zp
	b507 cmp zp	b534 lda zp	b576 cmp zp	
0032:	a3bd sty zp	a519 ldy zp	a675 sty zp	b14d ldy zp
	b220 cmp zp	b2be sty zp	b2de lda zp	b38f sbc zp
	b501 cpy zp	b536 ldy zp	b572 cpx zp	
0033:	a40e cmp zp	a430 cmp zp	a667 sta zp	aa39 cmp zp
	b388 lda zp	b4fa adc zp	b50b sta zp	b52a stx zp
	b5d8 cpx zp	b620 lda zp	b6c7 cpx zp	b6cd adc zp
	b6cf sta zp	e412 stx zp		
0034:	a408 cpy zp	a42a cpy zp	a669 sty zp	aa30 cmp zp
	b38d lda zp	b4fc ldy zp	b50d sty zp	b52c sta zp
	b5d2 cmp zp	b622 ldx zp	b6c3 cpy zp	b6d3 inc zp
	e414 sty zp			
0035:	b50f sta zp	b693 sta (),y	b69a adc zp	b69c sta zp
0036:	b511 sty zp	b6a0 inc zp		

0037:	a663 lda zp e430 lda zp	b526 ldx zp	e0ff stx zp	e40e stx zp
0038:	a665 ldy zp e410 sty zp	ad8e bit zp e436 lda zp	b528 lda zp	e0fd sty zp
0039:	a76a lda zp a891 lda zp ad64 sta zp	a7d1 sta zp a8a7 lda zp bdc9 ldx zp	a841 lda zp a8ed sta zp	a86c sta zp ab5b sta zp
003a:	a46c ldy zp a838 ldx zp a8ab lda zp b3a6 ldx zp	a492 stx zp a843 ldy zp a8f0 sta zp bdc9 lda zp	a767 lda zp a86e sty zp ab5d sty zp	a7d6 sta zp a88e lda zp ad69 sta zp
003b:	a845 sta zp	a868 lda zp		
003c:	a847 sty zp	a86a ldy zp		
003d:	a7ba sta zp	a83d sta zp	a862 lda zp	ab72 lda zp
003e:	a689 sta zp ab74 ldy zp	a7bc sty zp	a83f sty zp	a85b ldy zp
003f:	ab57 lda zp	acc9 sta zp		
0040:	ab59 ldy zp	accf sta zp		
0041:	a827 sta zp	ac06 ldx zp		
0042:	a829 sty zp	ac08 ldy zp		
0043:	ac11 stx zp acec lda (),y	ac24 ldx zp	acal sta zp	acdf lda zp
0044:	ac13 sty zp	ac26 ldy zp	aca3 sty zp	acel ldy zp
0045:	af2f ldx zp b0fb lda zp b1e7 sta zp	b092 sta zp b128 lda zp b22d cmp zp	b0d0 ora zp b16b lda zp b27d lda zp	b0d2 sta zp b1e0 lda zp b32c lda zp
0046:	af31 ldy zp b170 lda zp b275 lda zp	b0db stx zp b1dd lda zp b331 lda zp	b101 lda zp blea sta zp	b12a ldy zp b231 lda zp

0047:	b18f sta zp	b341 sta zp	b349 lda zp	b3d2 lda zp
	b40d sta zp	b418 lda (),y	b435 lda zp	be9d sty zp
	beb4 ldy zp			
0048:	ada8 bit zp	b191 sty zp	b346 sta zp	b3cf lda zp
	b415 sta zp	b41e ldy zp	b432 lda zp	
0049:	a39d sta zp	a3a9 lda zp	a6e8 sty zp	a6ef ldy zp
	a728 sty zp	a7a8 lda zp	a9a8 sta zp	a9ce sta (),y
	a9d3 sta (),y	aa73 sta (),y	aa78 sta (),y	aa7d sta (),y
	ac18 sta zp	ad27 sta zp	ad4b lda zp	b830 stx zp
	b844 and zp	bbd0 ldx zp	elca lda zp	elee stx zp
	elfb ldx zp	e224 stx zp	e238 lda zp	e24c lda zp
004a:	a396 lda zp	a3a2 sta zp	a7a5 lda zp	a8d6 sta zp
	a9aa sty zp	a9da ldy zp	ac1a sty zp	ad29 sty zp
	ad4d ldy zp	b83c stx zp	b842 eor zp	bbd2 ldy zp
	e234 stx zp	e24a ldx zp		
004b:	ac20 sta zp	aca5 lda zp	adfe ldy zp	ae64 sty zp
004c:	ac22 sty zp	aca7 ldy zp		
004d:	adb6 sta zp	adc9 eor zp	adcb cmp zp	adcf sta zp
	add7 ldx zp	ae15 sta zp	ae2b lda zp	b032 dec zp
004e:	b3ed sta zp	b3fa lda zp	b404 sta zp	b40b lda (),y
	b411 lda (),y	b429 lda (),y	b42e lda (),y	b43c sta zp
	b452 sta (),y	b456 sta (),y	b45a sta (),y	b45e sta (),y
	b462 sta (),y	b532 sta zp	b5ee sta zp	b608 ora zp
	b614 lda (),y	b630 sta (),y	b638 sta (),y	
004f:	b3ef sty zp	b3f7 lda zp	b407 sta zp	b43f sta zp
	b530 sty zp	b5f0 stx zp	b606 lda zp	
0050:	aa59 lda zp	aa68 sta zp	aa71 lda (),y	aa76 lda (),y
	aa7b lda (),y	b479 stx zp	b663 lda zp	b703 cmp (),y
	b708 lda (),y	b712 lda zp	b730 sbc (),y	b753 sbc (),y
	b76e sta zp			
0051:	aa5b ldy zp	aa6a sty zp	b47b sty zp	b665 ldy zp
	b714 ldy zp	b771 sta zp		
0053:	b54f sta zp	b56c sta zp	b5f2 lda zp	b5f6 lda zp
	e3ec sta zp			

0054:	afe0 jsr	e3c1 sta zp		
0055:	afd9 sta zp	b5f4 sta zp	b60c lda zp	b612 sta zp
	b62b ldy zp	b767 sta zp	b773 lda zp	
0056:	afde sta zp	b871 stx zp	b88d sty zp	b8b2 adc zp
	b8fe adc zp	e000 sta zp	e029 lda zp	
0057:	a417 lda zp,x			
0058:	a3dc lda zp	a3e0 sta zp	a3ea sta (),y	a3f1 sta (),y
	a4ff sta zp	b159 sta zp	b160 lda zp	b1a0 sta zp
	b2b2 adc zp	b2c9 sta (),y	b33f adc zp	b566 sta zp
	b56e lda zp	b58d adc zp	b58f sta zp	b5b4 cmp zp
	b624 sta zp	b62e lda zp		
0059:	a3e4 dec zp	a3f5 dec zp	a508 sty zp	b15b sty zp
	b162 ldy zp	b1a2 sty zp	b2aa adc zp	b2ae sta zp
	b2cd dec zp	b2d3 inc zp	b344 adc zp	b568 stx zp
	b570 ldx zp	b594 adc zp	b596 sta zp	b5b0 cpx zp
	b626 stx zp	b633 inc zp	b635 lda zp	
005a:	a3c0 lda zp	a3d0 lda zp	a3d5 sta zp	a3e8 lda (),y
	a3ef lda (),y	a4fb sta zp	b14f sta zp	b618 sta zp
005b:	a3c7 lda zp	a3d9 dec zp	a3f3 dec zp	a503 sty zp
	b151 sty zp	b61e sta zp		
005d:	af52 sty zp	b359 sta zp	b378 dec zp	bcf7 sty zp,x
	bd4a sbc zp	bd6f inc zp	be09 sta zp	be24 dec zp
	be2b inc zp	be37 lda zp	be4c stx zp	bea8 dec zp
005e:	af4b sty zp	bd3c sbc zp	bd47 lda zp	bd4c sta zp
	bd55 inc zp	bd5e dec zp	bd91 lda zp	bda3 adc zp
	bdae sta zp	be4a sta zp	bed5 ldx zp	bede sbc zp
005f:	a3c2 sbc zp	a4ab lda (),y	a4b7 lda zp	a4ba sbc (),y
	a4cd lda zp	a525 sta (),y	a619 sta zp	a61d lda (),y
	a625 cmp (),y	a631 cmp (),y	a638 lda (),y	a63c lda (),y
	a6cd lda (),y	a6d8 lda (),y	a6dc lda (),y	a703 lda (),y
	a708 lda (),y	a70c lda (),y	a70e stx zp	a8c5 lda zp
	b0f1 sta zp	b0fd cmp (),y	b104 cmp (),y	b10a lda zp
	b147 sta zp	b16d sta (),y	b17d sta (),y	b180 sta (),y
	b183 sta (),y	b185 lda zp	b199 adc zp	b21c stx zp
	b22a lda (),y	b233 cmp (),y	b238 lda (),y	b23b adc zp
	b23f lda (),y	b25a cmp (),y	b26f sta (),y	b277 sta (),y

b284 sta (),y	b297 sta (),y	b29b sta (),y	b2d8 sbc zp
b2dc sta (),y	b2e3 sta (),y	b2ea lda (),y	b2fc cmp (),y
b304 cmp (),y	b34e lda (),y	b353 lda (),y	b538 sta zp
b5e2 cpx zp	b5e6 stx zp	b616 adc zp	bd41 ror zp
bd43 bit zp	bd6b bit zp		

0060:	a3c9 sbc zp	a4b3 lda zp	a4c9 sbc zp	a61b stx zp
	a710 sta zp	a8cb lda zp	b0ef stx zp	b149 sty zp
	b18a ldy zp	b19b ldy zp	b21e sta zp	b241 adc zp
	b2e1 sbc zp	b53a stx zp	b5dc cmp zp	b5e8 sta zp
	b61a lda zp	bc04 sta zp,x	bc11 lda zp,x	bd2e ror zp
	bd35 bit zp	bd99 bit zp		

0061:	a422 sta zp,x	a937 lda zp	a9e9 sty zp	ae52 lda zp
	ae80 lda zp	b037 sta zp	b04a sbc zp	b052 ldx zp
	b1bf lda zp	b484 sta zp	b4a9 sty zp	b4d5 lda zp
	b7fb lda zp	b85d lda zp	b87b sbc zp	b881 sty zp
	b8f9 sta zp	b92c sbc zp	b934 sta zp	b938 inc zp
	b9f4 lda zp	b9fb sta zp	bab4 lda zp	babc adc zp
	bac8 sta zp	baf4 inc zp	bb1a sbc zp	bb1c sta zp
	bb21 inc zp	bbc2 sta zp	bbf5 lda zp	bc1b lda zp
	bc2b lda zp	bc4f stx zp	bc6d cpx zp	bc9b lda zp
	bccc lda zp	bce0 sta zp	bd8c ldx zp	bdf1 ldx zp
	bfb4 lda zp	e005 lda zp	e020 ldy zp,x	e022 sta zp,x
	e0e7 lda zp	e0ed sta zp	e316 lda zp	

0062:	a77c and zp	a77e sta zp	ae4f lda zp	af8f sty zp
	b039 stx zp	b068 cmp (),y	b395 sta zp	b480 stx zp
	b491 sta zp	b4d9 lda zp	b6f9 sta (),y	b8d0 sta zp
	b8db ldx zp	b8e1 stx zp	b914 lda zp	b918 sta zp
	b927 rol zp	b93c ror zp	b94d lda zp	b951 sta zp
	b97b inc zp	ba51 lda zp	bb2b cpy zp	bb72 sbc zp
	bb91 sta zp	bbbd sta zp	bbf0 and zp	bc3c sta zp
	bc44 lda zp	bc75 cmp zp	bcc0 lsr zp	bcc2 ora zp
	bcc4 sta zp	bce9 sta zp	bdcd sta zp	be80 lda zp
	be85 sta zp	e0a9 sta zp	e0d5 lda zp	e0d9 stx zp

0063:	aa16 ldy zp	ae4c lda zp	af89 sty zp	b03b sty zp
	b397 sty zp	b482 sty zp	b493 sty zp	b4dd lda zp
	b8c9 sta zp	b8df ldx zp	b8e5 stx zp	b90e lda zp
	b912 sta zp	b925 rol zp	b93e ror zp	b953 lda zp
	b957 sta zp	b977 inc zp	ba4c lda zp	bb31 cpy zp
	bb6c sbc zp	bb95 sta zp	bbb4 sta zp	bbe7 lda zp
	bc40 sta zp	bc7c cmp zp	bceb sta zp	bdcf stx zp
	be79 lda zp	be7e sta zp	e0b4 sta zp	e0db ldx zp
	e0df sta zp			

```

0064: a9cc lda zp    aa14 ldx zp    aa2e lda (),y    aa37 lda (),y
      aa45 lda zp    aa4b lda zp    aa54 lda (),y    ae49 lda zp
      aedc lda zp    af15 lda zp    af21 sbc zp    af2b sta zp
      af63 lda (),y  af67 lda (),y  af87 stx zp    afa0 lda zp
      afc2 lda zp    aff0 lda zp    b00b lda zp    blad lda zp
      b1f7 lda zp    b2f7 sta zp    b31a adc zp    b475 ldx zp
      b4e3 stx zp    b640 lda zp    b654 adc (),y    b6a6 lda zp
      b7a4 ldx zp    b804 lda zp    b8c2 sta zp    b8e3 ldx zp
      b8e9 stx zp    b908 lda zp    b90c sta zp    b923 rol zp
      b940 ror zp    b959 lda zp    b95d sta zp    b973 inc zp
      ba47 lda zp    bb37 cpy zp    bb66 sbc zp    bb99 sta zp
      bba7 sta zp    bbe2 lda zp    bc4d sta zp    bc83 cmp zp
      bced sta zp    be72 lda zp    be77 sta zp    e0ae sta zp
      e0dd lda zp    e0e1 stx zp

```

```

0065: a957 dec zp    a9d1 lda zp    aa18 lda zp    aa3d ldy zp
      aa4d ldy zp    ae46 lda zp    aed7 lda zp    af19 lda zp
      af25 sbc zp    af2d sty zp    af8b sta zp    afa2 ldy zp
      afbf lda zp    aff6 lda zp    b002 lda zp    blaf ldy zp
      b1fc lda zp    b2fa sta zp    b320 adc zp    b477 ldy zp
      b4e5 sty zp    b63d lda zp    b6a8 ldy zp    b739 sta zp
      b759 cmp zp    b75d lda zp    b7a8 ldx zp    b806 ldy zp
      b8bb sta zp    b8e7 ldx zp    b8ed stx zp    b902 lda zp
      b906 sta zp    b921 rol zp    b942 ror zp    b95f lda zp
      b963 sta zp    b96f inc zp    ba42 lda zp    bb3d cpy zp
      bb60 sbc zp    bb9d sta zp    bbaa sta zp    bbdd lda zp
      bc4b sta zp    bc8e sbc zp    bce2 lda zp    bcef sta zp
      be6a lda zp    be70 sta zp    e0b9 sta zp    e0d3 ldx zp
      e0d7 sta zp

```

```

0066: a778 lda zp    a9eb sty zp    ad49 sta zp    ae33 lda zp
      ae7c eor zp    b056 sta zp    b05f ldx zp    b1bb lda zp
      b7f7 lda zp    b853 lda zp    b857 sta zp    b885 sty zp
      b8fb sta zp    b947 lda zp    b94b sta zp    baa5 eor zp
      bad1 sta zp    bad4 lda zp    bbb9 sta zp    bbec lda zp
      bbfe sta zp    bc2f lda zp    bc53 sta zp    bc58 lsr zp
      bc69 eor zp    bc92 lda zp    bca2 bit zp    bcbc lda zp
      bcd7 lda zp    bcd9 sty zp    bd88 eor zp    bdel bit zp
      bdea sta zp    bfb8 lda zp    bfb9 sta zp    e0e5 sta zp
      e28b lda zp    e293 lda zp    e2ce sta zp    e30e lda zp

```

```

0067: bd02 stx zp    bd62 lda zp    e062 sta zp    e088 dec zp

```

```

0068: b995 ldy zp    bc02 lda zp,x  bc13 sta zp,x  bca9 sta zp
      bcb8 sty zp    bcc9 sty zp    e3f0 sta zp

```


0069:	ae6b sta zp	b875 lda zp	bab2 sta zp	bab7 lda zp
	bf7d lda zp	e01e lda zp,x	e024 sty zp,x	
006a:	ae6e sta zp	b01f and zp	b021 sta zp	b916 adc zp
	ba79 adc zp	baad sta zp	bb29 ldy zp	bb55 rol zp
	bb70 lda zp	bb74 sta zp		
006b:	ae71 sta zp	b910 adc zp	ba73 adc zp	ba9e sta zp
	bb2f ldy zp	bb53 rol zp	bb6a lda zp	bb6e sta zp
006c:	ae74 sta zp	b03d lda zp	b044 stx zp	b066 lda (),y
	b90a adc zp	ba6d adc zp	ba99 sta zp	bb35 ldy zp
	bb51 rol zp	bb64 lda zp	bb68 sta zp	
006d:	ae77 sta zp	b03f ldy zp	b046 sty zp	b904 adc zp
	ba67 adc zp	ba94 sta zp	bb3b ldy zp	bb4f asl zp
	bb5e lda zp	bb62 sta zp		
006e:	ae7a sta zp	b01b lda zp	b859 eor zp	b883 ldy zp
	baa3 sta zp	baa9 lda zp	bbfc lda zp	bd86 lda zp
	bf8b lda zp	e272 ldx zp		
006f:	aa5d sta zp	ae7e sta zp	b48d sta zp	b498 lda (),y
	b4ac adc zp	b4c3 ldx zp	b64a sta zp	b651 lda (),y
	b66d lda zp	b67c lda (),y	b680 lda (),y	b684 lda (),y
	b85b sta zp	b8a3 bit zp	baa7 sta zp	bacf lda zp
	baef stx zp	bb07 stx zp	bd8a sta zp	e03c sta zp
	e27f sta zp			
0070:	aa5f sty zp	af39 sta zp	b48f sty zp	b4b0 ldx zp
	b4b7 lda zp	b4c5 ldy zp	b4e7 sty zp	b64d sta zp
	b66f ldy zp	b86f ldx zp	b895 sty zp	b89c lda zp
	b8b4 sta zp	b8eb ldx zp	b8ef sty zp	b900 sta zp
	b91f asl zp	b944 ror zp	b965 lda zp	b969 sta zp
	b96b inc zp	b987 sty zp	b9a2 lda zp	ba3d lda zp
	ba85 ror zp	bb84 sta zp	bbc4 sty zp	bbf9 sty zp
	bc09 stx zp	bc18 stx zp	bclf asl zp	bc51 sta zp
	bc8a cmp zp	bcd5 sty zp	bff4 lda zp	e02b sta zp
	e0e9 sta zp			
0071:	a5ac sty zp	a5c7 ldy zp	a9ed sty zp	a9f5 inc zp
	a9f7 ldy zp	aa07 ldy zp	af4e sty zp	b27d stx zp
	b2a0 stx zp	b2c4 ldy zp	b2f0 sta zp	b311 ora zp
	b322 stx zp	b367 asl zp	b4ae sta zp	b7b9 stx zp
	b7e2 ldx zp	bdec sty zp	be53 ldy zp	be64 sty zp
	be9f ldy zp	beb2 sty zp	bec4 ldy zp	e043 sta zp

	e059 sta zp	e060 lda (),y	e064 ldy zp	e06c sta zp
	e073 lda zp	e07d sta zp		
0072:	b269 sty zp	b2a2 sta zp	b2c2 inc zp	b2cf dec zp
	b2f2 sta zp	b30f lda zp	b328 sta zp	b369 rol zp
	b4b5 stx zp	b7bb sty zp	b7e4 ldy zp	e045 sty zp
	e05b sty zp	e06a inc zp	e06e ldy zp	e075 ldy zp
	e07f sty zp			
0073:	a48a jsr	a6b3 jsr	a799 jsr	a7e4 jsr
	a801 jmp	a812 jsr	a95f jsr	a99f jsr
	ab13 jsr	ab82 jsr	ac51 jsr	ad84 jsr
	add1 jsr	ae8a jsr	aea5 jmp	af05 jmp
	afaa jsr	b0a5 jsr	b0b0 jsr	b0d8 jsr
	b1b2 jsr	b79b jsr	bd0a jsr	bd17 jsr
	bd30 jsr	e3e5 sta zp,x		
0079:	a6aa jsr	a792 jsr	a897 jsr	a92b jsr
	a940 jsr	aa9d jsr	ac2c jsr	ac91 jsr
	acad jsr	acd4 jsr	ad7d jsr	adb8 jsr
	b085 jsr	b08d jsr	b094 jsr	b202 jsr
	b441 jsr	b73b jsr	b7aa jmp	b7d7 jsr
	b834 jsr	e206 jsr	e211 jsr	
007a:	a486 stx zp	a57c ldx zp	a5b3 stx zp	a5f5 ldx zp
	a610 sta zp	a693 sta zp	a75f adc zp	a7b1 lda zp
	a7c0 lda (),y	a7c6 lda (),y	a7cf lda (),y	a7d4 lda (),y
	a7d9 adc zp	a7db sta zp	a834 lda zp	a864 sta zp
	a88b lda zp	a8b3 adc zp	a8c9 sta zp	a8f3 sta zp
	a8fd adc zp	a8ff sta zp	a919 lda (),y	ab76 sta zp
	ac1c lda zp	ac28 stx zp	ac4d stx zp	ac5d stx zp
	ac74 lda zp	ac9d lda zp	aca9 sta zp	acc2 lda (),y
	acc7 lda (),y	accc lda (),y	ad6e sta zp	ad9e ldx zp
	ada4 dec zp	ae0b ldx zp	ae11 dec zp	aebd lda zp
	af01 cmp (),y	b3d8 lda zp	b426 lda zp	b42b sta zp
	b44a sta zp	b7b5 ldx zp	b7bf stx zp	b7e6 stx zp
	bda9 adc (),y	e187 lda zp	e3a2 inc zp	
007b:	a488 sty zp	a60c dec zp	a699 sta zp	a762 lda zp
	a7b3 ldy zp	a7df inc zp	a836 ldy zp	a866 sty zp
	a888 lda zp	a8b5 ldx zp	a8cf sta zp	a8f6 sta zp
	a903 inc zp	ab78 sty zp	ac1e ldy zp	ac2a sty zp
	ac4f sty zp	ac76 ldy zp	ac9f ldy zp	acab sty zp
	ad73 sta zp	ada2 dec zp	ae0f dec zp	aebf ldy zp
	b3d5 lda zp	b423 lda zp	b430 sta zp	b44d sta zp

	b7b7 lda zp	b7c8 stx zp	b7e8 sty zp	ea11 lda zp
	e3a6 inc zp			
0080:	aalf jsr			
0090:	ee79 bit zp	f1ad lda zp	f249 bit zp	f28a bit zp
	f311 sta zp	f3df sta zp	f3ed lda zp	f4a9 sta zp
	f4da lda zp	f4f5 and zp	f4f7 sta zp	f505 lda zp
	f524 bit zp	f55d lda zp	f843 sta zp	fela lda zp
	felc ora zp	fele sta zp		
0091:	e4e2 ldy zp	f6da sta zp	f6ed lda zp	
0092:	f99e adc zp	f9a0 sta zp	f9d5 lda zp	f9e2 sta zp
0093:	f4a5 sta zp	f50c ldy zp	f5d4 lda zp	f72c lda zp
	f733 sta zp	f845 sta zp	fadb lda zp	fb20 lda zp
	fb3a lda zp			
0094:	ed12 bit zp	ed1c lsr zp	eddd bit zp	ede2 ror zp
0095:	ed21 sta zp	ed71 ror zp	edb9 sta zp	edc7 sta zp
	edeb sta zp			
0096:	f9f3 sta zp	fa10 lda zp	fa33 lda zp	fa3b sta zp
	fa44 lda zp			
0097:	f14e sty zp	f153 ldy zp	f179 stx zp	f18f ldx zp
	f196 ldx zp			
0098:	f2f3 dec zp	f2f5 cpx zp	f2f9 ldy zp	f314 ldx zp
	f331 sta zp	f359 ldx zp	f362 inc zp	
0099:	e5a6 sta zp	e663 ldx zp	f04d sta zp	f13e lda zp
	f157 lda zp	f233 sta zp	f33c cpx zp	f347 sta zp
009a:	e5a2 sta zp	e669 ldx zp	efel sta zp	f1cb lda zp
	f275 sta zp	f335 cpx zp	f343 stx zp	
009b:	f9ce lda zp	f9f8 eor zp	f9fa sta zp	fbal sta zp
	fc05 eor zp	fc07 sta zp	fc4e lda zp	
009c:	f85c sta zp	f962 ldx zp	fa2d inc zp	fa63 sta zp
009d:	f12b bit zp	f5af lda zp	f68f lda zp	f71b bit zp
	f74c bit zp	fel8 sta zp		

009e:	f024 lda zp	f1dd sta zp	f1f8 lda zp	f201 lda zp
	f76a sta zp	f786 lda zp	f7a3 sty zp	f7a5 ldy zp
	f7b1 inc zp	f7f5 sty zp	f803 inc zp	f807 ldy zp
	f858 sta zp	faf1 cpx zp	faf5 ldx zp	fb03 stx zp
	fb0a cpx zp	fb60 lda zp		
009f:	f79f sty zp	f7ad ldy zp	f7b3 inc zp	f7f1 sty zp
	f7fd ldy zp	f805 inc zp	f85a sta zp	fb08 ldx zp
	fb1c inc zp	fb1e inc zp		
00a0:	afe8 bit	f6a5 inc zp	f6b0 lda zp	f6b6 stx zp
	f6e2 ldy zp	f6e9 sty zp		
00a1:	e4e7 cmp zp	f6a1 inc zp	f6ac lda zp	f6b8 stx zp
	f6e0 ldx zp	f6e7 stx zp	f761 lda zp	
00a2:	f69d inc zp	f6a8 lda zp	f6ba stx zp	f6de lda zp
	f6e5 sta zp			
00a3:	ed17 ror zp	ed1e lsr zp	ed4c bit zp	f969 ldx zp
	f9c3 lda zp	fa00 dec zp	fa18 lda zp	fb99 sta zp
	fc0e dec zp	fc10 lda zp		
00a4:	ee65 ror zp	ee80 lda zp	f9a2 lda zp	f9a6 sta zp
	f9be sta zp	fb9d sta zp	fbf5 lda zp	fbf9 sta zp
00a5:	ed64 sta zp	ed8e dec zp	ee16 sta zp	ee3e lda zp
	ee52 inc zp	ee58 sta zp	ee72 dec zp	fc1a lda zp
	fc22 dec zp	fc8d stx zp		
00a6:	f18d dec zp	f1a5 sta zp	f1f6 sty zp	f3d1 sta zp
	f810 inc zp	f812 ldy zp		
00a7:	ef63 lda zp	ef69 lsr zp	ef70 lda zp	ef90 lda zp
	efbc lda zp	fa6e sta zp	fa99 lda zp	fad6 ldx zp
	fb5c dec zp	fc60 sta zp	fc71 dec zp	fedb sta zp
00a8:	ef5d dec zp	ef6e dec zp	ef7a adc zp	f98f sta zp
	fa37 sta zp	fa57 lda zp	fb9f sta zp	fbcd lda zp
	fbda inc zp	ff2b stx zp		
00a9:	e167 "bit"	e4d3 sta zp	ef59 ldx zp	ef89 sta zp
	f993 inc zp	f997 dec zp	f9eb lda zp	fa59 ora zp
	fba3 sta zp	fbe3 lda zp	fbec inc zp	

00aa:	ef6b ror zp	efa4 lda zp	efdb lda zp	f852 sta zp
	fa72 bit zp	fa88 sta zp	faa7 sta zp	faa9 dec zp
	faaf sta zp	fabc sta zp	fb4a sta zp	
00ab:	e4d7 sta zp	ef65 eor zp	ef67 sta zp	efbe eor zp
	f7bc sta zp	f869 sta zp	fab6 sta zp	fb70 sty zp
	fb74 eor zp	fb76 sta zp	fb80 lda zp	fc78 dec zp
	fc82 inc zp			
00ac:	e8ea lda zp	e90a sta zp	e962 sta zp	e981 lda zp
	e99d sta zp	e9d4 lda (),y	e9e3 lda zp	f61a lda zp
	f629 lda (),y	fae3 cmp (),y	fafc lda zp	fb0e lda zp
	fb28 cmp (),y	fb41 sta (),y	fb72 lda (),y	fb94 sta zp
	fc41 lda (),y	fcd2 lda zp	fcdb inc zp	
00ad:	e8ed lda zp	e95f sta zp	e984 lda zp	e9cd sta zp
	e9e7 lda zp	f61f lda zp	faf7 lda zp	fb15 lda zp
	fb90 sta zp	fc37 inc zp	fcd6 lda zp	fcdf inc zp
00ae:	e8f0 lda zp	e95c sta zp	e987 lda zp	e9d8 lda (),y
	e9e5 sta zp	f4d8 sta zp	f4ea sta zp	f512 cmp (),y
	f51c sta (),y	f51e inc zp	f593 sta zp	f5aa ldx zp
	f5dd stx zp	f77a lda zp	f795 lda zp	f7c3 sta zp
	f7e0 sta zp	fcd4 sbc zp		
00af:	e8f3 lda zp	e959 sta zp	e98a lda zp	e9ed sta zp
	f4e3 sta zp	f4ee sta zp	f522 inc zp	f598 sta zp
	f5ac ldy zp	f5df sty zp	f777 lda zp	f79a lda zp
	f7c6 sta zp	f7e7 sta zp	fcd8 sbc zp	
00b0:	f856 sta zp	f8e4 lda zp	f8e9 adc zp	f8f2 bit zp
	f959 lda zp	f971 adc zp	f97a adc zp	f982 adc zp
	f9db dec zp	f9de inc zp	fa26 adc zp	
00b1:	f8e2 stx zp	f8ec adc zp	f8ee sta zp	f8f7 asl zp
	f8fa asl zp	f905 adc zp	f93a stx zp	f94f sbc zp
	f951 stx zp	f954 ror zp	f957 ror zp	f95e cmp zp
	f973 cmp zp	f97c cmp zp	f984 cmp zp	f99c sbc zp
	falf lsr zp	fa24 sbc zp		
00b2:	f1a9 lda (),y	f1f3 sta (),y	f1fa sta (),y	f3ce sta (),y
	f56e lda (),y	f573 lda (),y	f57f lda (),y	f583 sbc (),y
	f588 lda (),y	f58c sbc (),y	f739 lda (),y	f757 lda (),y
	f781 sta (),y	f788 sta (),y	f78d sta (),y	f792 sta (),y
	f797 sta (),y	f79c sta (),y	f7af sta (),y	f7d0 ldx zp
	f7ff cmp (),y	fd63 stx zp		

00b3:	f7d2 ldy zp	fd65 sty zp		
00b4:	eebb lda zp	eedc dec zp	eee7 dec zp	eeee dec zp
	eef2 inc zp	ef00 inc zp	ef1c stx zp	f854 sta zp
	f98b lda zp	f9ac lda zp	f9fc lda zp	fa14 lda zp
	fa2f lda zp	fa42 sta zp	fa4c sta zp	
00b5:	eed4 sta zp	ef17 sta zp	fa46 sta zp	fa76 lda zp
	fa91 lda zp	fac0 lda zp	fe80 ora zp	
00b6:	eec1 lsr zp	ef28 sta zp	fa5b sta zp	fa95 lda zp
	fae9 sta zp	faeb lda zp	fb2d sty zp	fb2f lda zp
	fbcb ror zp	fbdc lda zp	fc8f stx zp	
00b7:	f3a1 lda zp	f3d9 ldy zp	f3f6 lda zp	f402 cpy zp
	f40f cpy zp	f4b8 ldy zp	f549 lda zp	f5b8 lda zp
	f5c1 ldy zp	f5cd cpy zp	f5fe ldy zp	f7a7 cpy zp
	f7f7 cpy zp	fdf9 sta zp		
00b8:	f322 sta zp	f34a ldx zp	f364 lda zp	fe00 sta zp
00b9:	f22a ldx zp	f23b lda zp	f26f ldx zp	f27d lda zp
	f2c8 lda zp	f2e0 lda zp	f32c sta zp	f369 lda zp
	f36d sta zp	f393 lda zp	f3c4 ldy zp	f3d5 lda zp
	f3e6 lda zp	f4bf ldx zp	f4c6 sta zp	f4d0 lda zp
	f579 lda zp	f5fc sta zp	f610 lda zp	f642 bit zp
	f64b lda zp	f66e lda zp	f681 lda zp	fe04 sty zp
00ba:	f219 lda zp	f25b lda zp	f29d lda zp	f327 sta zp
	f372 lda zp	f3e1 lda zp	f4ab lda zp	f4cb lda zp
	f5ed lda zp	f60b lda zp	f646 lda zp	fe02 stx zp
	fe07 lda zp			
00bb:	f3fc lda (),y	f413 lda (),y	f5c7 lda (),y	f7ab lda (),y
	f7fb lda (),y	fdfb stx zp		
00bc:	fdfd sty zp			
00bd:	eec9 eor zp	eeeb sta zp	eee2 lda zp	eef6 lda zp
	ef15 sta zp	fa55 sta zp	fa9c lda zp	fae1 lda zp
	fb24 lda zp	fb3f lda zp	fb82 eor zp	fba6 lda zp
	fbfd lda zp	fc01 sta zp	fc0c lsr zp	fc2c sta zp
	fc3b sta zp	fc43 sta zp	fc52 sta zp	
00be:	f8a6 sta zp	fa6a lda zp	fa7a ldx zp	fb55 ldx zp
	fb5a stx zp	fb64 sta zp	fc24 ldx zp	fc57 dec zp

```

fc84 lda zp

00bf: fa06 ror zp      fa53 lda zp

00c0: ea69 sty zp      ea71 lda zp      f8b1 sta zp

00c1: f59c sta zp      f5e4 sta zp      f774 lda zp      f78b lda zp
      f7c9 sta zp      f7db sta zp      fb92 lda zp      fd6e lda (),y
      fd73 sta (),y    fd75 cmp (),y    fd7a sta (),y    fd7c cmp (),y
      fd81 sta (),y

00c2: f5a0 sta zp      f5e8 sta zp      f771 lda zp      f790 lda zp
      f7cc sta zp      f7e3 sta zp      fb8e lda zp      fd6a sta zp
      fd6c inc zp      fd8a ldy zp

00c3: f49e stx zp      f4e8 lda zp      f570 sta zp      f591 adc zp
      f59a lda zp      fd1a stx zp      fd25 lda (),y    fd27 sta (),y

00c4: f4a0 sty zp      f4ec lda zp      f575 sta zp      f596 adc zp
      f59e lda zp      fd1c sty zp

00c5: eae5 cpy zp      eb28 sty zp

00c6: e5c0 cpx zp      e5c4 dec zp      e5cd lda zp      e5f1 stx zp
      e954 sty zp      eb21 ldy zp      eb35 ldx zp      eb40 stx zp
      f142 lda zp      f6f7 sta zp

00c7: e693 ldx zp      e789 sta zp      e852 sta zp      e895 stx zp

00c8: e610 sty zp      e62c cmp zp      e659 cpy zp      f16e sta zp

00c9: e591 cpx zp      e61b lda zp      e624 cpx zp      e87c lsr zp
      e8fa dec zp      f161 sta zp

00ca: e628 lda zp      f15d sta zp

00cb: ea8e sty zp      eac9 sty zp      eae0 ldy zp      eb26 ldy zp

00cc: e542 sta zp      e5cf sta zp      ea34 lda zp

00cd: e540 sta zp      ea16 sta zp      ea38 dec zp      ea3e sta zp

00ce: e5db lda zp      ea4d sta zp      ea5a lda zp

00cf: e520 sta zp      e5d7 lda zp      e5e2 sty zp      ea42 lsr zp
      ea4b inc zp

```

00d0:	e604 sty zp f16a sta zp	e636 lda zp	e65f sta zp	e71f sta zp
00d1:	e606 lda (),y e775 sta (),y e81d sta (),y eale sta (),y	e63c lda (),y e7f4 lda (),y e9d6 sta (),y ea24 lda zp	e763 lda (),y e80b lda (),y e9f3 sta zp ea47 lda (),y	e766 sta (),y e80e sta (),y ea0c sta (),y
00d2:	e9fc sta zp	ea28 lda zp		
00d3:	e50e sty zp e577 sta zp e654 inc zp e705 stx zp e79a sty zp e817 cpy zp e899 stx zp ea40 ldy zp	e515 ldy zp e617 sty zp e6b9 inc zp e713 sty zp e7a8 sty zp e83c lda zp e8a5 cmp zp f15b lda zp	e568 sty zp e62a sta zp e6bd cmp zp e721 ldy zp e7c4 sta zp e843 sta zp e8b7 cmp zp	e56e lda zp e63a ldy zp e6fe sta zp e75d sty zp e7fa cpy zp e85f sty zp ealc ldy zp
00d4:	e619 sty zp e6ae lsr zp	e64c ldx zp e77e ldx zp	e688 lda zp e7ea ldx zp	e68c sta zp e897 stx zp
00d5:	e58c sta zp e6eb sta zp e7b8 cmp zp	e602 ldy zp e711 ldy zp e7f2 ldy zp	e6bb lda zp e76f cpy zp e805 ldy zp	e6e6 lda zp e79d cpy zp f16c lda zp
00d6:	e50c stx zp e61f ldx zp e6f7 dec zp e7b6 inc zp e87e ldx zp e8c8 inc zp e965 ldx zp	e513 ldx zp e6cd ldx zp e701 ldx zp e7be dec zp e88c stx zp e8f8 dec zp e97c dec zp	e56a sty zp e6d6 dec zp e70c stx zp e836 ldx zp e8b0 dec zp e933 inc zp f15f lda zp	e56c ldx zp e6d8 ldx zp e7a1 dec zp e83a dec zp e8c2 ldx zp e956 ldx zp
00d7:	e63e sta zp e67a lda zp f9e4 cpx zp fc45 eor zp	e642 asl zp e717 sta zp fa04 lsr zp fc47 sta zp	e644 bit zp e723 lda zp fc20 stx zp	e674 sta zp f9aa stx zp fc39 lda zp
00d8:	e699 ldx zp e824 inc zp	e69d dec zp e829 ldx zp	e6aa lda zp e893 stx zp	e745 ldx zp e99f lda zp,x
00d9:	e54d sty zp,x e6da asl zp,x e6ed lda zp,x e92f lda zp	e55c sta zp,x e6dc lsr zp,x e888 lda zp,x e968 lda zp,x	e570 ldy zp,x e6df lda zp,x e918 lda zp,x e9b4 ldy zp,x	e582 ldy zp,x e6e3 sta zp,x e922 sta zp,x e9f5 lda zp,x


```

00da: e90c lda zp,x   e91c ldy zp,x   e9b0 lda zp,x   e9ba sta zp,x
00f1: e929 lda zp       e92d sta zp
00f3: e4dd sta (),y     e769 lda (),y     e76c sta (),y     e77a sta (),y
      e811 lda (),y     e814 sta (),y     e822 sta (),y     e9da sta (),y
      ea21 sta (),y     ea26 sta zp       ea52 lda (),y
00f4: ea2e sta zp
00f5: ea9d sta zp       eab7 lda (),y     eae2 lda (),y     eb6f sta zp
00f6: eaa1 sta zp       eb74 sta zp
00f7: efb1 sta (),y     f096 lda (),y     f472 stx zp
00f8: f2b5 lda zp       f2c1 sta zp       f46b lda zp       f470 sty zp
00f9: ef26 lda (),y     f026 sta (),y     f47b stx zp
00fa: f2ba lda zp       f2c3 sta zp       f474 lda zp       f479 sty zp
00ff: bde7 sta ,y       be58 sta ,y       be61 sta ,y       bea5 sta ,y
      beaf sta ,y     bec6 lda ,y       bf04 sta ,y
0100: bee8 sta ,y       bf09 sta ,y       fafe sta ,x       fb10 cmp ,x
0101: a38f lda ,x       blf3 lda ,x       blfe sta ,x       bee3 sta ,y
      faf9 sta ,x     bf17 cmp ,x
0102: a39a lda ,x       a3ab cmp ,x       blef lda ,x       blf9 sta ,x
      befa sta ,y
0103: a39f lda ,x       a3a4 cmp ,x       bef6 sta ,y
0104: beff sta ,y       ff4e lda ,x
0109: ad46 lda ,x       ad5c sbc ,x
010f: ad61 lda ,x
0110: ad66 lda ,x
0111: ad70 lda ,x
0112: ad6b lda ,x

```



```

01fb:  a5cb sta ,y      a5ce lda ,y      a5ef sta ,y

01fc:  a522 lda ,y      e3fb stx

01fd:  a609 sta ,y      e3f8 stx

01fe:  a511 sta

01ff:  a514 sty      abd3 sta

0200:  a4f3 lda      a569 sta ,x      a582 lda ,x      a5b8 lda ,x
      a5e5 lda ,x      a604 lda ,x      aacc sta ,x      abea lda
      ac38 sta      fd56 sta ,y

0201:  ab98 sta

0259:  f2fb lda ,y      f2fe sta ,x      f319 cmp ,x      f31f lda ,x
      f366 sta ,x

0263:  f301 lda ,y      f304 sta ,x      f324 lda ,x      f374 sta ,x

026d:  f307 lda ,y      f30a sta ,x      f329 lda ,x      f36f sta ,x

0276:  e5f6 sta ,x

0277:  e5b4 ldy      e5bc sta ,x      eb3c sta ,x

0278:  e5b9 lda ,x

0281:  fe36 ldx      fe3c stx

0282:  fd92 sta      fe39 ldy      fe3f sty

0283:  fe27 ldx      fe2d stx

0284:  fe2a ldy      fe30 sty

0285:  fe21 sta

0286:  e4da lda      e536 sta      e69f ldx      e777 lda
      e81f lda      e8d6 stx      ea57 ldx

0287:  e5dd ldx      ea44 ldx      ea54 sta

0288:  e544 lda      e9ca ora      e9f9 ora      fd97 sta

```

0289:	e52e sta ,y	eb37 cpx		
028a:	eaf2 bit			
028b:	e53b sta ,y	eb17 dec	eb1e sty	
028c:	e531 sta	eaeb sty	eb0d ldy	eb12 dec
028d:	ea89 sta eb48 lda	eac1 ora	eac4 sta	eb2a ldy
028e:	eb2d sty	eb4f cmp		
028f:	e524 sta	eadd jmp ()		
0290:	e529 sta			
0291:	e51d sta ec72 sta	eb54 lda	ec64 ora	ec6f and
0292:	e5d1 sta	e614 sty	e6c5 lda	
0293:	eee9 lda f423 lda	ef4e bit	ef74 lda	f415 sta ,y
0294:	eed9 bit f04f lda	ef06 lda f44d lda	efb5 bit	efe3 lda
0295:	f443 sta	f446 lda	ff07 lda	
0296:	f440 sty	ff0d lda	ff2f lda	
0297:	ef33 ora f00f sta f1c1 lda	ef36 sta f086 lda f40c sty	efd2 ora f093 sta fe0d lda	efd5 sta f09e sta fe13 sta
0298:	ef19 ldx	efa6 ldx	f420 stx	ff28 ldx
0299:	f033 lda	fee2 adc	ff37 sta	
029a:	f039 lda	feeb adc	ff3d sta	
029b:	ef97 ldy	efa0 sty	f08c cpy	f45c lda
029c:	ef9b cpy	f089 ldy	f098 inc	f45f sta

029d:	ef1e ldy	ef2a inc	f01b cpy	f465 sta
029e:	ef21 cpy	f017 ldy	f020 sty	f462 lda
029f:	f898 sta	fc00 lda		
02a0:	f89e sta	f8be lda	f8de sta	fca8 lda
02a1:	ef3e eor	ef43 sta	ef83 ora	ef86 sta
	eff2 lda	f028 lda	f062 lda	f07d lda
	f0a5 lda	f0aa lda	f0b8 sta	f49a sty
	fe73 and	fe85 lda	feb6 lda	fef6 lda
	ffa1 eor	ffa1 sta		
02a2:	f887 sta	f911 lda		
02a3:	f94b sta	f9b0 lda		
02a4:	f917 sta	f9b7 lda	f9c0 sta	
02a5:	e8fc dec	e935 inc	e96c stx	e978 ldx
	e993 cpx	e9ab cpx	e9bf ldx	
02a6:	f42c lda	fddd lda	ff68 sta	
02a9:	efcf "bit"			
0300:	a437 jmp ()	e388 jmp ()	e458 sta ,x	fd59 sta ,y
0302:	a480 jmp ()			
0304:	a579 jmp ()			
0306:	a717 jmp ()			
0308:	a7e1 jmp ()			
030a:	ae83 jmp ()			
030c:	e13a lda	e148 sta		
030d:	e13d ldx	e14b stx		
030e:	e140 ldy	e14e sty		
030f:	e136 lda	e152 sta		


```
0310: a058 .wo          e3c3 sta
0311: e3ca sta
0312: e3cd sty
0314: f895 lda          fcb3 sta          fcc0 sta          fd20 lda ,y
      fd29 sta ,y      ff58 jmp ()
0315: f89b lda          f8c1 cmp          fcad sta          fcc6 sta
0316: ff55 jmp ()
0318: fe44 jmp ()
031a: ffc0 jmp ()
031c: ffc3 jmp ()
031e: ffc6 jmp ()
0320: ffc9 jmp ()
0322: ffcc jmp ()
0324: ffcf jmp ()
0326: ffd2 jmp ()
0328: ffe1 jmp ()
032a: ffe4 jmp ()
032c: ffe7 jmp ()
0330: f4a2 jmp ()
0332: f5ea jmp ()
1ba2: b3ad "bit"
2009: ed0b "bit"
2ca9: aefc "bit"
3fa9: edfd "bit"
```



```
57a2: bbc9 "bit"
8000: fcec jmp ()
8002: fe5b jmp ()
8003: fd07 cmp ,x
80a9: efcc "bit"
9fea: afd6 lda ,y
9feb: afdb lda ,y
a000: fcff jmp ()
a002: fe6f jmp ()
a00c: a7fd lda ,y
a00d: a7f9 lda ,y
a080: adf1 cmp ,y      ae19 cmp ,y      ae35 ldx ,y
a081: ae24 lda ,y
a082: ae20 lda ,y
a09d: a5fa lda ,y
a09e: a5bc sbc ,y      a5ff lda ,y      a730 lda ,y      a738 lda ,y
a19e: a328 .wo
a1ac: a32a .wo
a1b5: a32c .wo
a1c2: a32e .wo
a1d0: a330 .wo
a1e2: a332 .wo
a1f0: a334 .wo
```

a1ff: a336 .wo

a210: a338 .wo

a225: a33a .wo

a235: a33c .wo

a23b: a33e .wo

a24f: a340 .wo

a25a: a342 .wo

a26a: a344 .wo

a272: a346 .wo

a27f: a348 .wo

a290: a34a .wo

a29d: a34c .wo

a2aa: a34e .wo

a2ba: a350 .wo

a2c8: a352 .wo

a2d5: a354 .wo

a2e4: a356 .wo

a2ed: a358 .wo

a300: a35a .wo

a30e: a35c .wo

a31e: a35e .wo

a324: a360 .wo

a326: a43d lda ,x

a327: a442 lda ,x

a383: a362 .wo

a38a: a749 jsr a8d8 jsr ad2b jsr

a38f: a3b5 bne

a3a4: a398 bne

a3b0: a3a7 bne

a3b7: a394 bne a3ae beq

a3b8: a50a jsr b15d jsr

a3bf: b628 jsr

a3dc: a3d7 bcs

a3e8: a3ed bne

a3ec: a3e2 bcs a3e6 "bcc" a3f8 bne

a3f3: a3ce beq

a3fb: a757 jsr a885 jsr adae jsr

a408: a3b8 jsr b264 jsr b2b9 jsr e426 jsr

a412: a40c bne

a416: a41a bpl

a421: a425 bmi

a434: a40a bcc a410 bcc a42c bcc

a435: a3fe bcs a405 bcc a42e bne a432 bcs
b30b jmpa437: a573 jmp a85f jmp a8e5 jmp ab68 jmp
ad32 jmp ad9b jmp af0a jmp b24a jmp
b3b0 jmp b4d2 jmp b65a jmp b980 jmp
bb8c jmp e109 jmp e19e jmp

```
a43a: e38e jmp
a456: a460 bpl
a469: a851 jmp
a474: a46f beq      e391 jmp
a480: a48e beq      a4f6 beq      a530 jmp
a483: e449 .wo
a49c: a494 bcc
a4d7: a4d2 bcs
a4df: a4da bcc      a4e4 bne      a4eb bne
a4ed: a4a7 bcc
a508: a505 bcc
a522: a528 bpl
a52a: e1b2 jmp
a533: a4f0 jsr      a52d jsr      e1b8 jsr
a53c: a55d "bcc"
a544: a547 bne
a55f: a540 beq
a560: a483 jsr      ac03 jmp
a562: a56f bcc
a576: a567 beq
a579: a496 jsr      a49f jsr
a57c: e44b .wo
a582: a58c "bne"    a5e1 bne
```


a58e:	a585	bpl		
a5a4:	a59e	bne		
a5ac:	a5a6	bcc		
a5b6:	a5bf	beq		
a5b8:	a602	bne		
a5c7:	a607	"bpl"		
a5c9:	a589	beq	a590 beq	a59a bvs
	a5aa	bcc	a5e8 beq	a5a2 "bne"
a5dc:	a5d6	beq		
a5de:	a5da	bne		
a5e5:	a5f3	"bne"		
a5ee:	a596	beq		
a5f5:	a5c3	bne		
a5f9:	a5fd	bpl		
a609:	a5d1	beq		
a613:	a4a4	jsr	a6a7 jsr	
a617:	a63e	"bcs"	a8c0 jsr	
a62e:	a629	beq		
a637:	a62c	"bne"		
a640:	a61f	beq		
a641:	a050	.wo	a627 bcc	a633 bcc
	a642	bne	a6b1 bne	a635 beq
a644:	e444	jmp		
a659:	a4ed	jsr	a52a jsr	a87a jmp

```
a65d: a044 .wo
a660: a87d jsr
a663: e101 jmp
a677: e1bb jmp
a67a: a462 jsr      e382 jsr
a68d: a65e bne      a6a2 bne
a68e: a659 jsr      e1b5 jsr
a69b: a042 .wo
a6a4: a69c bcc      a69e beq
a6bb: a6ad beq
a6c9: a6c1 bne      a712 bne
a6e6: a6e0 bne
a6e8: a6e4 beq
a6ef: a73b bmi
a6f3: a71a bpl      a71e beq      a722 bmi
a700: a6f8 bne
a714: a6cf beq      a6e6 bcs      a701 beq
a717: a705 bne
a71a: e44d .wo
a72c: a735 "bmi"
a72f: a733 bpl
a737: a72d beq      a740 "bne"
a741: a00e .wo
```

```
a753: a74c bne
a79f: a797 bne
a7ae: a7ea jmp      a89d jmp      ad75 jmp
a7be: a7b8 beq
a7ce: a7c9 bne
a7e1: a499 jmp      a7dd bcc      a809 beq
a7e4: e44f .wo
a7ed: a7e7 jsr      a948 jmp
a7ef: a95c jmp
a804: a7f1 bcc
a807: a7c2 bne
a80b: a810 bne
a80e: a7f5 bcs
a81c: a024 .wo
a81d: a677 jsr
a827: a824 bcs      ace7 jmp
a82b: a7ed beq
a82c: a6d1 jsr      a7ae jsr
a82e: a02c .wo
a830: a00c .wo
a832: a82f bcs
a849: a83b jmp
a84b: a7cb jmp
```

```
a854: a84f bcc
a856: a040 .wo
a862: a85d bne
a870: a020 .wo      a832 bne      a857 bne
a87d: a878 bne
a882: a026 .wo
a897: a880 jmp
a89f: a01e .wo
a8a0: a81a jmp      a89a jsr      a945 jmp
a8bc: a8af bcs
a8c0: a8b7 bcc      a8ba "bcs"
a8d1: a028 .wo      a8d2 bne
a8e3: a8c3 bcc
a8e8: a955 bne
a8eb: a8de beq
a8f7: a012 .wo
a8f8: abe7 jmp      b3db jsr
a8fb: a93e "beq"     abf6 jmp      acd1 jsr
a905: a901 bcc      a91b beq      a91f beq
a906: a75a jsr      a8f8 jsr      abf3 jsr      acb8 jsr
a909: a8a3 jsr      a93b jsr
a911: a926 "beq"
a919: a924 bne
```



```
a927: a022 .wo
a937: a930 beq
a93a: a02a .wo
a940: a939 bne
a948: a943 bcs
a94a: a02e .wo
a953: a97d bcs
a957: a951 beq      a967 beq
a95f: a959 bne
a96a: a971 bcs
a96b: a49c jsr      a6a4 jsr      a6b6 jsr      a8a0 jsr
      a962 jsr
a971: a9a2 jmp
a99f: a99b bcc
a9a4: a01c .wo
a9a5: a746 jsr      a804 jmp
a9c2: ac8e jsr
a9d6: a9c2 bpl
a9d9: a9bf bne
a9da: ac83 jsr
a9ed: aa0c bne
aa07: aa00 beq
aald: a9ef jsr      a9f9 jsr
aa24: a9e5 bne
```

```
aa27: aa22 bcc
aa2c: a9de bne
aa3d: aa34 bne
aa4b: aa32 bcc      aa3b bcc      aa41 bcc
aa52: aa43 bne      aa49 bcs
aa68: aa4f jmp
aa7f: a03c .wo
aa85: a046 .wo
aa86: aa80 jsr
aa90: aa89 beq
aa9a: aaba bmi
aa9d: aac8 "bne"
aa9f: a03e .wo
aaa0: aa97 jmp
aaa2: ab16 jmp
aaca: a576 jmp
aad7: a44e jsr      a6d4 jsr      aaa0 beq
aae5: aade bpl      ab35 jsr
aae7: aaa2 beq      aad5 bne      ab29 beq
aae8: aaaf beq
aaee: aaf0 bcs
aaf8: aaa6 beq      aaab beq
ab0e: aaf6 "bne"
```

```
ab0f:  ab07 bcc
ab10:  ab1c bne
ab13:  aab3 beq      ab0c bcc
ab19:  ab11 bne
able:  a469 jsr      a478 jsr      ab6f jsr      acf8 jmp
      bdda jmp      e191 jmp      elaf jsr      e42d jsr
      e441 jsr
ab21:  aa9a jsr      aac2 jsr      abcb jsr
ab28:  ab33 bne      ab38 jmp
ab3b:  aac5 jsr      ab19 jsr      ac00 jsr
ab42:  ab3d beq
ab45:  a451 jsr      abfd jsr      ac47 jsr
ab47:  a45b jsr      a6f3 jsr      a73d jsr      aad9 jsr
      aae2 jsr      ab2d jsr
ab4d:  ac9a jmp
ab57:  ab51 bmi
ab5b:  ab55 "bne"
ab5f:  ab04 bne
ab62:  ab4f beq
ab6b:  ab64 beq
ab7a:  a04e .wo
ab92:  ab80 bne
aba4:  a014 .wo
abb5:  aa83 jmp      abe4 jsr
abb7:  aba2 bne
```

```
abbe: a016 .wo
abce: abb2 jsr      abc1 bne
abd6: abf1 bne
abea: abdb beq      abe2 beq
abf9: abd6 jsr      ac4a jsr
ac03: abfb bne
ac05: a01a .wo
ac0d: abed bne
ac0f: ab9d jsr
ac15: acb5 jmp
ac41: ac33 bvc
ac4a: ac45 bne
ac4d: ac3f "bne"
ac51: ac2f jsr      acdc jmp
ac65: ac5a bvc
ac71: ac63 "beq"
ac72: ac69 beq
ac7d: ac7a bcc
ac89: ac56 bpl
ac91: ac86 jmp
ac9d: ac94 beq      ac98 beq
acb8: ac41 bmi      acda bne
acd1: acbd bne
```



```
acdf:  acb0 beq
acea:  ace5 bpl
acfb:  acee beq      afc2 bne
ad1d:  a010 .wo
ad24:  ad1e bne      ad87 jsr
ad27:  ad22 "beq"
ad32:  acc4 beq
ad35:  ad2e beq
ad75:  ad82 bne
ad78:  ad5f beq
ad8a:  a775 jsr      a79c jsr      b438 jsr      b79e jsr
      b7eb jsr      e12a jsr
ad8d:  a772 jsr      adf6 jsr      ae61 jsr      afe3 jmp
      b1b8 jsr      b3c3 jsr      b3f1 jmp      b400 jsr
      b465 jsr
ad8f:  afba jsr      b646 jsr      b6a3 jsr
ad90:  a9bc jsr      b016 jsr
ad96:  ad97 bcs
ad97:  ad92 bmi
ad99:  ad94 bcs
ad9e:  a928 jsr      a9b7 jsr      aab5 jsr      ad8a jsr
      aef4 jsr      afb4 jsr      blb5 jsr      e257 jsr
ada4:  ada0 bne
ada9:  ae2d jmp
adb8:  b677 jmp
```

```
adbb: add4 jmp
add7: adbe bcc      adc2 bcs
ade8: ade3 bne
adf0: ae17 "bne"
adf9: ae1e "bcc"
adfa: af11 jmp
ae07: add9 bne
ae11: ae0d bne
ae19: ae00 bpl
ae20: adfa jsr
ae30: adcd bcc
ae33: ae28 jsr
ae38: a7a2 jsr
ae43: a788 jmp
ae58: addb bcs      addf bcc
ae5b: ae03 beq
ae5d: adf4 bcs
ae64: ae5f beq
ae66: ae05 "bne"    aelc bcs
ae80: ae5b beq
ae83: adb1 jsr      b643 jsr
ae86: e451 .wo
ae8a: aeb7 beq
```

```
ae8f: aeaf beq
ae92: ae8d bcs
ae9a: ae95 bcc
aead: ae9c bne
aebd: abc3 jsr
aec6: aec3 bcc
aecc: aeab bne
aed3: a099 .wo
aee3: aece bne
aeea: aee5 bne
aef1: aeec bcc      afd1 jsr      b3fd jsr
aef7: b20b jsr      b3c6 jsr      b761 jsr
aefa: aef1 jsr      afb1 jsr      b3b9 jsr
aefd: acb2 jsr      afb7 jsr      b07e jsr      b742 jsr
      b7f1 jsr acb2  jsr e20e jsr
aeff: a76f jsr      a817 jsr      a934 jsr      a9ae jsr
      aa8d jsr      ab8a jsr      abaa jsr      abc8 jsr
      b3cb jsr      b3e3 jsr
af08: a80b jmp      a8e8 jmp      ab5f jmp      ae30 jmp
      af03 bne      b09c jmp      b138 jmp      b446 jmp
      e216 jmp
af0d: aeb3 beq
af0f: aed2 "bne"
af14: af3b jsr      af6e jsr
af27: afl d bcc
af28: ae97 jmp
```

```
af5c: af3e bcc          af42 bne          af46 bne
af5d: af35 beq
af6e: af5f bpl
af84: af48 jsr          af7b jsr
af92: af75 bne
afa0: af71 bcc          af79 bne          af94 bne          af98 bne
afa7: aeee jmp
afd1: afaf bcc
afd6: afce jmp
afe5: a093 .wo
afe8: a090 .wo
b015: a09c .wo
b02e: b019 bcs
b056: b04c beq          b050 bcc
b05b: b06a beq
b061: b02b jmp
b066: b05d bne
b072: b061 bmi          b064 "bcc"          b06e bcs
b07b: b077 beq
b07e: b088 bne
b080: a018 .wo
b08b: a9a5 jsr          ac15 jsr          ad24 jsr          af28 jsr
      b3c0 jsr
b090: b082 jsr
```



```
b092: b3ea jsr
b09c: b0ca bne
b09f: b09a bcs
b0af: b0a8 bcc
b0b0: b0b3 bcc      b0b8 bcs
b0ba: b0ad bcc
b0c4: b0bc bne
b0d4: b0c2 "bne"
b0db: b0c6 bne
b0e7: b0e2 bne
b0ef: b111 "bne"
b0f1: b10e bcc
b0fb: b0f5 bne
b109: b0ff bne
b113: ae92 jsr      b097 jsr      b0aa jsr      b0b5 jsr
b11c: b115 bcc
b11d: b0f9 beq
b123: b132 beq
b128: b121 bne
b138: b141 beq
b13b: b12e bne      b136 bne
b143: b13d bne
b159: b156 bcc
```

```
b185: b106 beq
b18f: b18c bcc
b194: b253 jsr      b261 jsr
b1a0: b19d bcc
b1b2: b1e3 jsr
b1b8: b7a1 jsr
b1bf: a9c7 jsr      aed4 jsr      afed jsr      afff jsr
      blaa jsr
b1cc: b1bd bmi
b1ce: b1c3 bcc
b1d1: b0e4 jmp
b1db: b207 beq
b21c: b243 bcc
b228: b222 bne
b237: b22f bne
b245: b25c bne      b308 jmp
b248: aa24 jmp      blcc bne      b798 jmp      b9f1 jmp
b24a: b251 bne
b24d: b235 beq
b261: b226 beq
b274: b271 bpl
b27d: b279 bpl
b286: b2a8 bne
b296: b28c bvc
```

```
b2b9: b2b4 bcc
b2c8: b2cb bne      b2d1 bne
b2cd: b2c6 beq
b2ea: b25e jmp
b2f2: b326 bne
b308: b300 bne
b30b: b2ac bcs      b2b7 beq      b365 bcs      b376 bcs
b30e: b2fe bcc
b30f: b306 bcc
b320: b314 beq
b331: b32e bpl
b337: b333 bpl
b34b: b2e7 bne      b3a9 bne
b34c: b29d jsr      b316 jsr
b355: b33b jsr
b35f: b37a bne
b378: b36b bcc
b37d: a05a .wo
b384: b37f beq
b391: aee0 jmp      af6b jmp      b013 jmp      b3a4 "beq"
b39e: a05c .wo
b3a2: b77f jmp      b795 jmp      b821 jmp
b3a6: ab7b jsr      abce jsr      b3b6 jsr
```

```
b3ae: b413 beq
b3b2: a038 .wo
b3e1: b3b3 jsr      b3f4 jsr
b3f4: aee7 jmp
b418: b41c bpl
b449: b444 beq
b44f: b3de jmp
b465: a072 .wo
b46f: af59 jmp
b475: aa56 jsr      b4c0 jsr      b65d jsr
b47d: b6f3 jsr      b70f jsr
b487: aabf jsr      able jsr      aec6 jsr      b473 "beq"
b48d: ac7d jsr
b497: b4a2 bne
b4a4: b49e beq
b4a8: b49a beq
b4a9: b4a6 beq
b4b5: b4b2 bcc
b4bf: b4b9 beq
b4ca: b4bd bne      b674 jsr      b6fd jmp      b729 jmp
b4d2: b51a bmi
b4d5: b4ce bne
b4f4: b47d jsr
```



```
b4f6: b524 "bne"
b501: b4fe bcs
b50b: b505 bne
b516: b503 bcc          b509 bcc
b526: a41c jsr          b384 jsr          b51c jsr
b52a: b63a jmp
b544: b54b "beq"
b54d: b546 beq
b559: b564 "beq"
b561: b55b bne
b566: b55f beq
b56e: b599 bpl          b59c bmi
b572: b5b6 beq
b57d: b574 bne          b578 bne
b5ae: b5aa bcc
b5b0: b5bb "beq"
b5b8: b5b2 bne
b5bd: b561 jsr
b5c7: b548 jsr          b5b8 jsr
b5dc: b5d4 bcc
b5e6: b5e0 bne
b5f6: b5bf bmi          b5c4 bpl          b5c9 beq          b5d6 bne
      b5da bcs          b5de bcc          b5e4 bcc
b601: b5fd bcc          b60a beq
```

b606: b57a jmp

b63d: ade5 jmp

b65d: b656 bcc

b67a: aa61 jsr b660 jsr

b688: b4c7 jsr

b68c: b66a jsr b726 jsr

b690: b696 bne

b699: b68d beq

b6a2: b69e bcc

b6a3: b782 jsr e25a jsr

b6a6: a9e0 jsr ab21 jsr b034 jsr b381 jsr

b6aa: b041 jsr b667 jsr b671 jsr b716 jsr

b6d5: b6d1 bcc

b6d6: b6c1 bne b6c5 bne b6c9 bne

b6db: aa6c jsr b6ae jsr

b6eb: b6dd bne b6e1 bne

b6ec: a078 .wo

b700: a07a .wo

b706: b734 jmp

b70c: b706 bcc

b70d: b755 bcs

b70e: b75b "bcc" b75f "bcs"

b725: b721 bcc

b72c: a07c .wo

b737: a07e .wo

b748: b740 beq

b761: b700 jsr b72c jsr b748 jsr

b77c: a070 .wo

b782: b77c jsr b78b jsr b7ad jsr

b78b: a076 .wo

b798: b74b beq b78e beq b7a6 beq b7f9 bmi
 b7ff bcs

b79b: aaff jsr

b79e: a94b jsr aa86 jsr ab85 jsr aba5 jsr
 afc7 jsr b745 jsr b7f4 jmp e203 jmp
 e221 jsr

b7a1: b6ec jsr

b7ad: a074 .wo

b7b5: b7b0 bne

b7cd: b7ca bcc

b7e2: ac80 jsr aec9 jmp

b7eb: b824 jsr b82d jsr

b7f1: b839 jsr

b7f7: b7ee jsr b813 jsr e12d jsr

b80d: a06e .wo

b823: a03a .wo

b82c: a030 .wo

b83c: b837 beq


```
b840: b846 beq
b848: b878 beq
b849: be2f jsr      e290 jsr
b850: ba0f jsr      e288 jsr      e334 jsr
b852: a084 .wo
b853: e02d jsr      e281 jsr
b862: b899 bmi
b867: ad4f jsr      b84d jmp      ba01 jsr      bald jsr
      e081 jsr      e0d0 jsr      e268 jsr      e2a4 jsr
b869: a081 .wo
b86a: b85f jmp      bd8e jmp
b86f: b86a bne
b877: baf1 jsr
b893: b87f bcc
b897: b891 "bne"
b8a3: b865 "bcc"      b87d beq
b8af: b8ab beq
b8d2: bc55 jmp      bce6 jmp
b8d7: b8d2 bcs      bb9f jmp      e0ef jsr
b8db: b8f5 bne
b8f7: b7b2 jmp      b92e bcs      badc jmp
b8f9: bf81 jmp
b8fb: bacc jmp
b8fe: b8a5 bpl
```



```
b91d: b929 bpl
b929: b8dd bne
b936: b91a jmp
b938: bc28 jmp
b946: b936 bcc
b947: b8d4 jsr
b94d: bcab jsr
b96f: bc23 jsr
b97d: b96d bne      b971 bne      b975 bne      b979 bne
b97e: b93a beq      badf jmp      bd9d jmp
b983: ba5b jmp
b985: b99b bmi      b99d beq
b999: b862 jsr      bcb5 jsr
b9a6: b9b8 bne
b9ac: b9a8 bcc
b9b0: b8a0 jsr      bcc6 jsr
b9ba: b9a4 bcs
b9ea: a062 .wo      bfa3 jsr
b9f1: b9ed beq
b9f4: b9ef bpl
ba28: be04 jsr      bfaa jsr      bff1 jsr      e04c jsr
      e056 jmp      e070 jsr      e0c9 jsr
ba2a: a087 .wo
ba30: ba2b bne
```

ba59:	ba3f jsr	ba44 jsr	ba49 jsr	ba4e jsr
ba5e:	ba53 jsr	ba59 bne		
ba61:	ba89 bne			
ba7d:	ba62 bcc			
ba8b:	ba2d jmp			
ba8c:	b850 jsr	b867 jsr	ba28 jsr	bb0f jsr
bab7:	ba30 jsr	bb1e jsr		
bab9:	e03f jsr			
bac4:	babe bcc			
bacf:	baca bne			
bad4:	e00b jsr			
bada:	bab9 beq	bac4 bpl		
badf:	bac0 bmi bb23 beq	bad8 bmi	baeb bcs	baf6 beq
bae2:	a9f2 jsr be21 jsr	aa0e jsr	bd5b jsr	bd71 jsr
baed:	aa04 jsr			
baf8:	bae6 beq			
bafe:	bd52 jsr	be28 jsr		
bb07:	e274 jsr			
bb0f:	ba08 jsr	e2d9 jmp	e321 jsr	
bb11:	a08a .wo			
bb12:	bb0c jmp			
bb29:	bb59 bmi			

```
bb3f: bb2d bne      bb33 bne      bb39 bne      bb57 bcs
      bb5b "bpl"

bb4c: bb41 bcc      bb7c "bne"

bb4f: bb77 jmp

bb5d: bb4d bcs

bb7a: bb46 beq

bb7e: bb48 bpl

bb8a: bb12 beq

bb8f: ba56 jmp      bb87 jmp

bba2: a78f jsr      ad42 jsr      aea2 jsr      afa4 jmp
      bb09 jsr      bf78 jsr      e0c2 jsr      e2c9 jsr

bbc7: e05d jsr

bbca: e047 jsr      e2b4 jsr

bbd0: a9d6 jmp      ad52 jsr

bbd4: b420 jsr      bbce beq      bf88 jsr      e0f6 jmp

bbfc: affc jsr      b86c jmp

bbfe: bf9e jsr

bc02: bc07 bne

bc0c: a9fc jsr      bae2 jsr      bafe jsr      bd7f jsr
      bf71 jsr      e26b jsr      e277 jsr

bc0f: e002 jsr

bcl1: bc16 bne

bcla: bcl d beq      bc21 bcc      bc26 bne

bclb: a9c4 jsr      ae43 jsr      bb14 jsr      bbd4 jsr
      bc0c jsr
```



```
bc23: bffa jsr
bc2b: a79f jsr      b9ea jsr      bc39 jsr      bc65 bne
      e097 jsr
bc2f: bc6b bmi
bc31: bc98 jmp
bc38: bc2d beq      bc34 bcs
bc39: a052 .wo
bc3c: af9d jmp      b07b jmp      bd83 jsr
bc44: b39b jmp
bc49: bdd4 jsr
bc4f: af81 jsr
bc58: a056 .wo
bc5b: b027 jsr      b1c9 jsr      be0f jsr      bela jsr
      bf96 jsr
bc5d: ad57 jsr
bc92: bc6f bne      bc77 bne      bc7e bne      bc85 bne
bc98: bc94 bcc
bc9b: aa11 jsr      b1ce jmp      b801 jsr      bcd2 jsr
      be32 jsr
bc9f: bca4 bpl
bcba: bc90 bne
bcbb: bcb3 bpl
bccc: a054 .wo      bf8f jsr      e00e jsr      e27a jsr
bce9: bc9d beq
bcf2: bcd0 bcs
```



```
bcf3:  ac89 jsr      ae8f jmp      b7da jsr
bcf7:  bcfa bpl
bd06:  bd00 bne
bd0a:  bd04 "beq"     bd45 bvc      bd7b jmp
bd0d:  bcfc bcc
bd0f:  bd08 bne
bd2e:  bd1e beq      bd22 beq
bd30:  bd26 beq      bd2a beq      bdb0 jmp
bd33:  bd1a bcc
bd35:  bd2c "bne"
bd41:  bd11 beq
bd47:  bd15 bne      bd37 bpl
bd49:  bd3e jmp
bd52:  bd57 bne
bd5b:  bd50 bpl      bd60 bne
bd62:  bd4e beq      bd59 "beq"
bd67:  bd64 bmi
bd6a:  bd0d bcc
bd71:  bd6d bpl
bd7e:  aa29 jmp      ba21 jsr      bd78 jsr
bd91:  bd33 bcc
bda0:  bd95 bcc
bdae:  bd9b bmi
```

```
bdc2:  a471 jsr
bdc4:  a6ea jsr      e43a jsr
bdda:  bdc6 jsr
bddd:  aabc jsr
bddf:  b46a jsr      bdd7 jsr
bde7:  bde3 bpl
bdf8:  bdf3 bne
be00:  bdfc beq
be09:  bdfe bcs
be0b:  be2d bne
be16:  be26 bne
be21:  be1d beq
be28:  be14 bpl
be2f:  be1f bpl
be32:  be12 beq
be47:  be3c bmi
be48:  be40 bcs
be53:  be4f beq
be64:  be5c "beq"
be66:  be51 bpl
be68:  af56 jsr
be6a:  be8a bpl      be8e bmi      bec2 bne
be8e:  be88 bcs
```

be90: be8c "bmi"

be97: be91 bcc

beb2: beaa bne

bec4: bebe beq

bec6: becc beq

bed3: bed0 beq

bee3: bed9 bpl

beef: bef2 bcs

bf04: bdf5 jmp

bf07: bed7 beq

bf0c: bf02 "beq"

bf16: be82 adc ,y

bf17: be7b adc ,y

bf18: be74 adc ,y

bf19: be6d adc ,y

bf71: a05e .wo

bf7a: a08d .wo

bf84: bf7f bne

bf9e: bf8d bpl bf99 bne

bfb3: a096 .wo

bfb4: bd67 jmp	e030 jsr	e29d jsr	e2aa jsr
e313 jsr	e33a jmp		

bfbe: bfb2 bcc bfb6 beq

bfed: a064 .wo bf7b beq bfad jsr

```

bffd: bff8 bcc

cfff: e5ad sta ,x

d011: f88d lda      f892 sta      fc95 lda      fc9a sta

d012: ff5e lda

d016: fcef stx

d018: eb59 lda      eb5e sta      ec48 lda      ec53 lda
      ec58 sta

d019: ff63 lda

d418: fdc4 stx

dc00: e93a sta      e945 sta      ea90 sta      eaa5 sta
      ead7 sta      eb44 sta      f6c9 stx      f6d4 sta
      fdab sta

dc01: e93d lda      ea93 ldx      eaab lda      eaae cmp
      f6bc lda      f6bf cmp      f6cc ldx      f6cf cpx

dc02: fdc8 stx

dc03: fdbe stx

dc04: f907 sta      fde4 sta      fdee sta

dc05: f90e sta      fdf3 sta

dc06: f8fe lda      f932 sbc      f93d sty      fbb1 sta

dc07: ed94 sta      ee22 sta      f90b adc      f92c ldx
      f935 cpx      f940 sty      fbb4 stx

dc0d: ea7e lda      ed9c lda      ed9f lda      ee2d lda
      ee30 lda      f877 sty      f87a sta      f91a lda
      f948 lda      fa3f sta      fa50 sta      fb4f stx
      fb52 ldx      fbb7 lda      fca2 sta      fda5 sta
      ff70 sta

dc0e: f87d lda      f914 sta      fdb0 sta      ff73 lda
      ff7a sta

```


dc0f:	ed99 sta	ee27 sta	f882 sta	f945 sta
	fbbc sta	fdb6 sta		
dd00:	ed2e lda	ed33 lda	ed66 lda	ed69 cmp
	ed84 lda	ed8b sta	edbe lda	edc3 sta
	edf3 lda	edf8 sta	ee5a lda	ee5d cmp
	ee67 lda	ee6a cmp	ee85 lda	ee8a sta
	ee8e lda	ee93 sta	ee97 lda	ee9c sta
	eea0 lda	eea5 sta	eea9 lda	eeac cmp
	f492 ora	f495 sta	fdcd sta	fe7b lda
	fe82 sta			
dd01:	ef0c bit	efeb bit	eff9 bit	effe lda
	f003 sta	f006 bit	f05b bit	f068 lda
	f06d sta	f070 lda	f453 lda	f48d sta
	fed6 lda			
dd02:	fdd2 sta			
dd03:	f48a sta	fdcl stx		
dd04:	f036 sta			
dd05:	f03c sta			
dd06:	fedd lda	fee5 sta	fefe sta	ff0a sta
	ff22 sta			
dd07:	fee8 lda	feee sta	ff01 sta	ff10 sta
	ff25 sta			
dd0d:	ef3b sta	ef46 sta	ef80 sta	f0b3 sta
	f485 sta	fda8 sta	fe4e sta	fe51 ldy
	fe88 sta	feb9 sta	fef9 sta	
dd0e:	f030 sta	f049 sta	fdb3 sta	
dd0f:	fdb9 sta	fef3 sta	ff15 sta	
e000:	bffd jmp			
e00b:	e016 beq			
e00e:	e009 bcc			
e01e:	e027 bpl			

```
e043: ba16 jsr      e2b1 jmp      e328 jsr

e059: e037 jsr

e05d: e04f jsr

e06c: e068 bne

e070: e08a bne

e07d: e07a bcc

e097: a060 .wo

e0be: e09c bne

e0d3: e09a bmi

e0e3: e0bb jmp

e0f6: e2c2 jsr

e0f9: e10f bcs      e115 bcs      e11b bcs      e121 bcs
      e127 bcs      e162 bcs      e1d1 jmp

e104: e0fb bne

e109: e105 bne

e10c: ab47 jsr

e112: a562 jsr

e118: aa93 jsr

e11e: ab8f jsr      abaf jsr

e124: ac35 jsr

e129: a048 .wo

e155: a034 .wo

e164: a036 .wo

e167: a032 .wo
```

```
e194: e18b beq      e1cf bcc
e195: e17c beq
e19e: e185 bne
e1a1: e19a beq
e1b5: e1a5 bne
e1bd: a04a .wo
e1c6: a04c .wo
e1d1: e178 bcs      e1c4 bcs
e1d4: e156 jsr      e16c jsr
e200: e1e9 jsr      e1f6 jsr      e231 jsr      e245 jsr
e206: e1e0 jsr      e1e6 jsr      e1f3 jsr      e22e jsr
      e242 jsr      e251 jsr
e20d: e209 bne      e214 bne
e20e: e200 jsr      e254 jsr
e211: e21e jsr
e219: e1be jsr      e1c7 jsr
e23f: e23c bcc
e257: e1e3 jsr
e264: a066 .wo
e26b: a068 .wo      e2bb jsr
e29d: e28e bpl      e2dd jmp
e2a0: e295 bmi
e2ad: e2a8 bpl
e2b4: a06a .wo
```

```
e2dc: e2d2 jsr
e30e: a06c .wo
e316: e311 bpl
e324: e31b bcc
e337: e32e bcc
e33d: e338 bpl
e37b: a002 .wo
e386: a714 jmp      a854 jmp      e3a0 "bne"
e38b: e447 .wo
e391: e38c bmi
e394: a000 .wo
e3a2: e3b1 beq      e3e2 lda ,x
e3a8: e3a4 bne
e3b9: e3ad bcs
e3bf: e397 jsr
e3e2: e3e8 bpl
e421: e41d bne
e422: e39a jsr
e447: e455 lda ,x
e453: e394 jsr
e455: e45c bpl
e4ad: e118 jsr
e4b6: e4b3 bcc
```



```
e4d3: ef94 jmp
e4da: ea07 jsr
e4e0: f763 jsr
e4e2: e4e9 bne
e4ea: f43d lda ,x
e4eb: e4e5 bne          f43a ldy ,x
e500: fff3 jmp
e505: ffed jmp
e50a: fff0 jmp
e513: e50a bcs
e518: fe6c jsr          ff5b jsr
e544: e86e jsr
e54d: e558 bne
e555: e552 bcc
e560: e564 bpl
e566: e59d jmp          e78f jsr
e56c: e510 jsr          e70e jsr          e847 jsr          e88e jmp
e570: e57a bpl
e57c: e572 bmi
e582: e58a bpl
e58c: e584 bmi
e591: e621 jsr
e598: e593 beq
```

```
e5a0: e518 jsr      e59a jsr
e5aa: e5b1 bne
e5b4: e5e7 jsr      f147 jmp
e5b9: e5c2 bne
e5ca: e600 bne
e5cd: e5d4 beq      e5fc "beq"      e638 beq
e5e7: e5d9 beq
e5f3: e5fa bne
e5fe: e5ec bne
e606: e60d bne
e60f: e60a bne
e632: f163 jmp      f163 jmp      f170 jmp
e63a: e61d bmi      e626 bne      e62e bcc
e64a: e646 bpl
e650: e64a bcc
e654: e64e bne      e650 bvs
e65d: e630 "bcs"
e66f: e667 beq
e672: e66d beq
e674: e65b bne
e682: e67e bne
e684: e656 jsr      e73f jsr
e690: e686 bne
```

```
e691: e7e0 jmp
e693: e742 jmp
e697: e749 jmp      e782 jmp      e82f jmp
e699: e695 beq
e69f: e69b beq
e6a8: e709 "bne"      e7aa jmp      e7cb jmp      e826 jmp
      e861 jmp      e867 jmp      e871 jmp      e89e jmp
      ec5b jmp      ec75 jmp
e6b0: e6ac beq
e6b6: e6a5 jsr
e6cd: e6c8 beq
e6da: e6d1 bcc      e97e jmp      e9c2 jsr
e6ed: e595 jmp      e6f2 bne
e6f4: e6ef bmi
e6f7: e6c3 beq
e700: e6bf bcs
e701: e753 jsr      e864 jsr
e70b: e703 bne
e716: e5ca jsr      e66f jsr      f1d2 jmp
e72a: e725 bpl
e731: e72c bne
e73d: e737 bcc
e73f: e73b "bne"
e745: e733 bcc
```

```
e74c: e747 beq
e759: e751 bne
e762: e771 bne
e773: e756 jmp
e77e: e74e bne
e785: e780 beq
e78b: e787 bne
e792: e78d bne
e7a8: e7ba bcc      e7bc beq
e7aa: e79f bcc
e7ad: e794 bne
e7c0: e7c6 bne
e7c8: e7c2 bcc
e7cb: e77c "bpl"
e7ce: e7af bne
e7d4: e727 jmp
e7dc: e7d8 bne
e7e3: e7de bcc
e7ea: e7e5 bne
e7fe: e7f8 bne
e805: e7fc bne
e80a: e819 bne
e826: e800 beq
```


e829: e7f0 bne

e82d: e7ec bne

e832: e82b beq

e847: e841 bcc

e84c: e834 bne

e854: e84e bne

e864: e859 beq

e86a: e856 bne

e871: e838 beq e845 "bpl" e84a "bne"

e874: e86c bne

e87c: e6f9 jsr e7a3 jsr e7c8 jsr e89b jsr

e880: e88a bpl

e888: e883 bne

e891: e72e jmp e7e7 jmp

e8a1: e759 jsr e85b jsr

e8a5: e8ad bne

e8b0: e8a7 beq

e8b3: e6b6 jsr e797 jsr

e8b7: e8bf bne

e8c2: e8b9 beq

e8ca: e8c6 beq

e8cb: e7ce jsr e876 jsr

e8cd: e8d3 bpl

```
e8d6: e8d0 beq
e8da: e8cd cmp ,x
e8ea: e6d3 jsr      e885 jsr      e975 jsr
e8f6: e931 bpl
e8ff: e911 "bmi"
e913: e905 bcs
e918: e927 bne
e922: e91e bpl
e94d: e94f bne      e952 bne
e956: e949 bne
e958: e9c5 jmp
e965: e802 jsr
e967: e6ca jmp      e96a bpl
e981: e971 beq      e973 bcc
e98f: e9a4 "bmi"
e9a6: e996 bcc      e998 beq
e9ab: e9bd "bne"
e9ba: e9b6 bpl
e9bf: e9ae bcc
e9c8: e90e jsr      e9a1 jsr
e9d4: e9dd bpl
e9e0: e9cf jsr
e9f0: e57c jsr      e6f4 jmp      e900 jsr      e990 jsr
      ea01 jsr
```

```
e9ff: e560 jsr      e913 jsr      e9a6 jsr

ea07: ea0f bpl

ea13: e5e4 jsr      e6a2 jsr

ea1c: ea5e jsr

ea24: e58e jmp      e75f jsr      e807 jsr      e9e0 jsr
      ea04 jsr      ea18 jsr      ea4f jsr

ea31: fd30 .wo      fd9f .wo

ea5c: ea49 bcs

ea60: e3a8 lda

ea61: ea36 bne      ea3a bne

ea71: ea65 beq

ea79: ea6f "bne"

ea7b: ea73 bne

ea87: ea7b jsr      ff9f jmp

eaa8: eada "bne"

eaab: eab1 bne

eab3: ead2 bne

eac9: eabb bcs      eabf beq

eacb: eac7 "bpl"

eacc: eab4 bcs

eadc: eacf bcs

eae0: eb76 jmp

eaf0: eae7 beq

eafb: ea98 beq
```

```
eb0d: eaf5 bmi          eaff beq          eb03 beq          eb07 beq
eb17: eb10 beq
eb26: eaee "bne"      eafb beq
eb42: eaf7 bvs          eb0b bne          eb15 bne          eb1a bne
      eb24 bpl          eb32 beq          eb3a bcs          eb52 beq
eb64: eb4d bne
eb6b: eb67 bcc
eb76: eb57 bmi          eb61 jmp
eb79: eb6c lda ,x
eb7a: eb71 lda ,x
eb81: eb79 .wo
ebc2: eb7b .wo
ec03: eb7d .wo
ec44: e7d1 jmp
ec4f: e879 jmp          ec46 bne
ec58: ec4d "bne"
ec5b: ec6b bne
ec5e: ec51 bne
ec69: ec60 bne
ec72: ec67 "bmi"
ec78: eb7f .wo
ecb8: e5aa lda ,x
ece6: e5f3 lda ,x
ecef: e99a lda ,x
```


ecf0: e9f0 lda ,x

ecf1: e907 lda ,x

ed09: f238 jsr f4cd jsr ffb4 jmp

ed0c: f27a jsr f3e3 jsr f60d jsr f648 jsr
ffb1 jmp

ed11: ee00 jsr

ed20: ed14 bpl

ed2e: ed29 "bne"

ed36: edbb jsr edc9 jsr

ed40: ed19 jsr ede7 jsr

ed50: ed53 bcc

ed55: ed58 bcs

ed5a: ed4e bpl ed5d bcc

ed66: ed6c bne ed90 bne

ed7a: ed73 bcs

ed7d: ed78 "bne"

ed9f: eda9 bcs

edad: ed47 bcs

edb0: ed6f bcc eda4 bne

edb2: ee44 jmp

edb9: f286 jsr f3ea jsr f612 jsr f651 jsr
ff93 jmp

edbe: edd0 jsr ee03 jsr f281 jsr

edc7: f245 jsr f4d2 jsr ff96 jmp

edcc: f23f jsr

edd6: edd9 bmi

eddd: f1d8 jmp f3fe jsr f61c jsr f621 jsr
 f62b jsr ffa8 jmp

ede6: eddf bmi

edeb: ede4 "bne"

edef: f340 jsr f528 jsr ffab jmp

edfe: f339 jsr f63f jsr f654 jsr ffae jmp

ee03: edb7 bcc

ee06: ee7d jsr

ee09: ee0a bne

ee13: f1b5 jmp f4d5 jsr f4e0 jsr f501 jsr
 ffa5 jmp

ee1b: ee1e bpl

ee20: ee54 "bne"

ee30: ee3a bmi

ee3e: ee35 bne

ee47: ee40 beq

ee56: ee3c "bpl"

ee5a: ee60 bne ee63 bpl ee74 bne

ee67: ee6d bne ee70 bmi

ee80: ee7b bvc

ee85: ed2b jsr ed49 jsr ed7d jsr edd3 jsr
 ee0d jsr ee18 jsr ee4a jsr

ee8e: ed37 jsr ed5f jsr edf0 jsr ff7d jmp

```
ee97:  ed24 jsr      ed3a jsr      ed41 jsr      ed7a jsr
        ee10 jmp      ee2a jsr

eea0:  ed75 jsr      edcd jsr      ee47 jsr      ee76 jsr

eea9:  ed44 jsr      ed50 jsr      ed55 jsr      ed5a jsr
        eda6 jsr      edd6 jsr      ee1b jsr      ee37 jsr
        eeaf bne

eeb3:  ed3d jsr

eeb6:  eeb7 bne

eebb:  fe9d jsr

eec8:  eec5 bcc

eed1:  eeec bpl      eef0 bne      ef04 "bne"

eed7:  eecf beq

eee6:  eef4 "bne"    ee fa "bne"    eefe bvc

eee7:  eee4 bne      eef8 beq      eefc bvs

eef2:  eedc beq

eef6:  eee0 bvs

eefc:  eede bmi

ef00:  eebf bmi

ef06:  eebd beq      f044 jsr

ef13:  ef0a bcc

ef2e:  ef0f bpl

ef31:  ef11 bvc

ef39:  ef24 beq

ef3b:  ef8d jmp      f041 jsr      f07a jmp

ef4a:  f41d jsr
```

```
ef54: ef51 beq
ef58: ef54 bvc
ef59: ff04 jmp
ef6d: ef7c bne      efba bmi      efc2 bvs      efc5 bvc
ef6e: efb8 beq
ef70: ef61 bmi
ef7e: ef92 bne      efd8 jmp
ef90: ef5b bne
ef97: ef5f beq
efa9: efaf "bne"
efb1: efab beq
efc5: efc0 beq
efca: ef9e beq
efcd: efdf "beq"
efd0: efdd bne
efdb: ef72 beq
efel: f26c jmp
eff2: eff7 bne
eff9: effc bvs
f006: f00b bmi
f00d: efec bpl      f05e bpl      f459 jsr
f012: efe7 bcc      eff0 bne      f009 bvs
f014: f01e beq
```



```
f017: f208 jsr
f028: f014 jsr
f04c: f02c bcs
f04d: f227 jmp
f062: f066 bcs
f070: f075 beq
f077: f082 beq
f07d: f053 bcc      f057 beq
f084: f060 beq
f086: f150 jsr
f09c: f08f beq
f0a4: ed0e jsr      f88a jsr
f0aa: f0af bne
f0bb: f0a8 beq
f0bd: f12f lda ,y
f12b: f5da jmp
f12f: f13a bpl      f5b5 jsr      f5be jsr      f695 jsr
      f71f jsr      f752 jsr      f81e jsr      f82b jmp
f13c: f12d bpl
f13e: fd46 .wo
f14a: f140 bne
f14e: f1b8 jsr
f155: f144 beq
f157: fd40 .wo
```

```
f166: f14c bne          f159 bne
f173: f168 bne
f18d: f186 bne
f193: f184 bcs
f196: f17e bcs
f199: f17b jsr          f181 jsr          f1a7 beq
f1a9: f19c bne
f1ad: f173 bcs
f1b1: f1c6 bne
f1b3: f1bf bne
f1b4: f1a1 bcs          f1bb bcs
f1b5: f1af beq
f1b8: f177 beq          f1c8 beq
f1ca: fd42 .wo
f1d5: f1cf bne
f1db: f1d5 bcc
f1dd: f2d4 jsr
f1f8: f1e8 bne
f1fc: f20b jmp
f1fd: f1ed bcs
f207: f203 bcc
f208: f1e3 bcc
f20e: fd3a .wo
```

```
f216: f211 beq
f22a: f225 bne
f233: f21b beq      f21f beq      f22e beq      f24b bpl
f237: f221 bcs
f245: f23d bpl
f248: f242 jmp
f250: fd3c .wo
f258: f253 beq
f25f: f273 beq
f262: f25d bne
f26f: f26a bne
f275: f264 beq      f28c bpl
f279: f266 bcs
f286: f27f bpl
f289: f284 "bne"
f291: fd38 .wo
f298: f294 beq
f2ba: f2b7 beq
f2bf: f2bc beq
f2c8: f2a9 bne
f2e0: f2da bcc
f2ee: f2a5 bcs
f2f1: f29f beq      f2a3 beq      f2cc beq      f2e4 bne
      f2eb jmp
```

```
f2f2: f2ac jsr
f30d: f2f7 beq
f30f: f20e jsr      f250 jsr      f351 jsr
f314: f291 jsr
f316: f31c bne
f31f: f216 jsr      f258 jsr      f298 jsr
f32e: f317 bmi
f32f: fd48 .wo
f333: fd3e .wo
f33c: f337 bcs
f343: f33e bcs
f34a: fd36 .wo
f351: f34c bne
f359: f354 bne
f362: f35d bcc
f384: f37d bcc
f38b: f386 bne
f393: f38e bcs
f3ac: f3b6 "bcs"
f3af: f3a3 beq
f3b8: f397 bne
f3c2: f3a8 bcc      f3b4 bcc
f3d1: f3c8 beq
```


f3d3:	f377 beq f3db beq	f37b beq	f382 "bcc"	f3d7 bmi
f3d4:	f39c bcs	f3aa beq	f3b2 beq	f3bb bcs
f3d5:	f37f jsr	f4c8 jsr	f605 jsr	
f3f6:	f3ef bpl			
f3fc:	f404 bne			
f406:	f3f8 beq			
f409:	f388 jmp			
f40f:	f41b bne			
f41d:	f411 beq			
f43a:	f42f bne			
f440:	f437 jmp			
f446:	f428 beq			
f45c:	f451 bcc	f457 bcs		
f474:	f46d bne			
f47d:	f2c5 jmp	f476 bne		
f483:	f2af jsr	f409 jsr		
f49e:	ffd5 jmp			
f4a5:	fd4c .wo			
f4af:	f4b4 beq			
f4b2:	f4ad bne			
f4bf:	f4ba bne			
f4f0:	f4e6 bne			
f4f3:	f509 bcs	f526 bvc		

f501: f4fc bne

f51c: f50e beq

f51e: f514 beq

f524: f520 bne

f530: f4de bcs f554 "bcs" f55b bcs

f533: f4b6 bcc

f539: f534 bcs

f541: f53c bcs

f549: f56a bne

f556: f54b beq

f55d: f550 bcc

f56c: f57b bne

f579: f566 beq

f57d: f577 "bcs"

f5a9: f52e "bcc"

f5ae: f544 bcs f552 beq f559 beq f562 bne

f5af: f39e jsr f4c1 jsr f546 jsr

f5c1: f698 jmp

f5c7: f5cf bne

f5d1: f5b1 bpl b5ba beq f5c3 beq

f5d2: f4f0 jsr f5a2 jsr

f5da: f5d6 beq

f5dd: ffd8 jmp

```
f5ed: fd4e .wo
f5f1: f5f6 beq      f662 bcc
f5f4: f5ef bne
f605: f600 bne
f624: f63d "bne"
f633: f4fe jmp
f63a: f631 bne
f63f: f627 bcs
f642: f2ee jsr      f52b jsr      f633 jsr
f654: f406 jmp
f657: f644 bmi
f659: f5f8 bcc
f65f: f65a bcs
f676: f672 bne
f68d: f685 beq
f68e: f667 bcs      f67a bcs      f67f bcs      f691 bpl
f68f: f608 jsr      f669 jsr
f69b: ffea jmp
f6a7: f69f bne      f6a3 bne
f6bc: f6b4 bcc      f6c2 bne      f8ca jsr      fe5e jsr
f6cc: f6d2 bne
f6da: f6c5 bmi
f6dc: f6d8 bne
```

```
f6dd: ffde jmp
f6e4: ffdb jmp
f6ed: fd44 .wo
f6fa: f6f1 bne
f6fb: f35f jmp
f6fe: f356 jmp
f701: f213 jmp      f255 jmp
f704: f3ac jmp      f530 jmp
f707: f24d jmp      f28e jmp      f3f3 jmp
f70a: f230 jmp      f34e jmp
f70d: f25f jmp
f710: f4bc jmp      f602 jmp
f713: f390 jmp      f4af jmp      f536 jmp      f53e jmp
      f5f1 jmp      f65c jmp
f729: f71d bvc
f72c: f3af jsr      f556 jsr      f749 bne      f7ea jsr
f74b: f741 beq      f745 beq
f757: f75f bne
f767: f74e bpl
f769: f735 bcs      f73d beq
f76a: f2e8 jsr      f3bf jsr      f677 jsr      f689 jsr
f781: f784 bne
f7a5: f7b5 "bne"
f7b7: f7a9 beq
```


f7cf: f76f bcc

f7d0: f2ce jsr f38b jsr f539 jsr f65f jsr
 f76c jsr f7d7 jsr f80d jsr

f7d7: f7b7 jsr f847 jsr f864 jsr

f7ea: f3a5 jsr f54d jsr f801 bne

f7f7: f809 "bne" .

f80b: f7f9 beq

f80c: f7ed bcs

f80d: f199 jsr fle5 jsr

f817: f399 jsr f541 jsr f84a jsr

f81e: f83f "bne"

f821: f827 bne

f82e: f817 jsr f824 jsr f838 jsr

f836: f81a jmp f832 bne f83b beq

f838: f3b8 jsr f664 jsr f86b jsr

f841: f19e jsr f72f jsr

f84a: f5a5 jsr

f864: fle5 jsr f2d7 jsr

f867: f67c jsr

f86b: f7be jsr

f86e: f84d bcs

f875: f862 bne

f8b5: f8bb bne

f8b7: f8b8 bne

```
f8be: f8cd jmp
f8d0: f821 jsr      f8c7 jsr
f8dc: f86e bcs      f8c5 beq
f8e1: f8d4 bne
f8e2: f9cb jsr      fa0a jsr      fa2a jsr      fa67 jsr
f8f7: f8f4 bmi
f8fe: f903 bcc
f92a: f91f beq
f92c: f938 bne      fdal .wo
f969: f964 beq
f988: f96b bmi
f98b: f986 bcc      f9d0 bne      f9e9 bne
f993: f975 bcs
f997: f97e bcs      falc jmp
f999: f995 "bcs"
f9ac: f960 bcs      f98d beq      f991 "bne"      f9ed bmi
      f9f1 bcc      f9f5 "bcs"
f9bc: f9b5 bne
f9c9: fa02 bmi
f9d2: f9ae beq      f9ba bne      f9fe beq
f9d5: f9a8 beq
f9de: f9d9 bmi
f9e0: f9d7 beq
f9f7: f9c5 bpl      f9e6 bne
```

```
fa10: f988 jmp
fa18: fa12 beq
fa1f: fa16 beq      fala bmi
fa44: fa31 bne
fa53: fa48 beq
fa5d: fa35 beq
fa60: f966 jmp
fa70: fa6c beq
fa86: fa78 bne
fa8a: fa7d bne      fa84 "bne"      fa93 bne      fa97 bne
      faab bne      fab8 "beq"      fabe "bne"
fa8d: fa74 bpl
faa3: fa9e bmi
faa9: fa8f bne
faba: faa0 bcc      faa3 bcs
fac0: fa8d bvs
face: fac2 beq
fad6: fad1 bcc
faeb: fadd beq      fae5 beq
fb08: fad9 beq
fb2f: fb22 beq
fb33: faf3 bcc
fb3a: faed beq      fb05 jmp      fb31 beq
```


fb43:	fb0c bne	fb13 bne	fb1a bne	fb2a beq
	fb38 "bne"	fb3c bne		
fb48:	fad3 jmp			
fb4a:	facb jmp			
fb5c:	fb58 bmi			
fb68:	fb5e beq			
fb72:	fb7e bcc			
fb8b:	fb46 bne	fb62 bne	fb66 "beq"	fb84 bne
fb8e:	f617 jsr	fab1 jsr	fb6b jsr	fc88 jsr
fb97:	f8a8 jsr	fa60 jsr	fc16 jsr	fc75 jsr
fba6:	fbf0 jsr			
fbad:	fbe7 jsr			
fbaf:	fbab bcc	fc6c jsr		
fbbl:	fbdb jsr			
fbcb:	fc35 bne			
fbcd:	fd9d .wo			
fbe3:	fbcf bne			
fbf0:	fbe5 bne			
fc09:	fbcb "bmi"	fbdb bne	fbde bpl	fbea bne
	fbee "bne"	fbf3 bne	fc14 "bpl"	fc2e "bne"
	fc3d "bcs"	fc4c "bne"		
fc0c:	fbfb beq			
fc16:	fc91 "bne"			
fc2c:	fc28 bne			
fc30:	fc1c beq			


```
fc3f: fc33 bcc
fc4e: fc12 beq
fc54: fc68 "bne"      fc6f bne      fc73 bne      fc7a bpl
      fcbb "beq"
fc57: fbe0 jmp
fc5e: fc59 bne
fc6a: fd9b .wo
fc93: f8d6 jsr      fb68 jsr      fcb8 jsr
fcb6: fcab beq
fcb8: fc86 beq
fcbd: f8a1 jsr      fc65 jsr      fc7e jsr
fcca: fc5b jsr      fc9d jsr
fcd1: f624 jsr      face jsr      fb7b jsr      fc30 jsr
fcdb: f63a jsr      fb43 jsr      fb78 jsr      fc49 jsr
fcel: fcdd bne
fce2: fffc .wo
fcef: fcea bne
fd02: fce7 jsr      fe56 jsr
fd04: fd0d bne
fd0f: fd04 lda ,x    fd0a bne
fd15: fcf8 jsr      fe66 jsr      ff8a jmp
fd1a: ff8d jmp
fd20: fd2d bpl
fd27: fd23 bcs
```

```
fd50: fcf5 jsr          ff87 jmp
fd53: fd5d bne
fd6c: fd86 "beq"
fd6e: fd84 bne
fd88: fd77 bne          fd7e bne
fd93: fcbd lda ,x
fd94: fcc3 lda ,x
fda3: fcf2 jsr          fe69 jsr          ff84 jmp
fddd: fca5 jsr          ff6b jmp
fdec: fde0 beq
fdf3: fde9 jmp
fdf9: ffbd jmp
fe00: ffba jmp
fe07: ffb7 jmp
fe18: ff90 jmp
fela: fe0b bne
felc: edb2 jsr          ee4f jsr          f18a jsr          f518 jsr
      fa81 jsr          fac6 jsr          fb35 jsr          fb88 jsr
fe21: ffa2 jmp
fe25: ff99 jmp
fe27: f2b2 jsr          f468 jsr
fe2d: f480 jmp          fd8d jsr          fe25 bcc
fe34: ff9c jmp
fe3c: fe34 bcc
```

```
fe43:  fffa .wo
fe47:  fd34 .wo
fe5e:  fe59 bne
fe66:  fd32 .wo      fd4a .wo
fe72:  fe54 bmi      fe64 bne
fe9a:  fe92 beq
fe9d:  fe8e beq      fe97 jmp
fea3:  fe79 beq
feae:  fea6 beq
feb6:  fea0 jmp      feab jmp      febl beq
febc:  f9d2 jmp      fa0d jmp      fa5d jmp      fa8a jmp
      fb8b jmp      fc09 jmp      fc54 jmp
fec0:  f434 lda ,x
fecl:  f431 ldy ,x
fed6:  fe94 jsr      fea8 jsr
ff07:  fe9a jsr      feb3 jsr
ff2e:  f44a jsr
ff43:  f927 jmp
ff48:  fffe .wo
ff58:  ff53 beq
ff5b:  fcfb jsr      ff81 jmp
ff5e:  ff61 bne
ff6e:  fdf6 jmp
ff90:  a47d jsr      a874 jsr
```


ff99: e40b jsr

ff9c: e403 jsr

ffb7: abdd jsr af9a jsr e180 jsr e195 jsr

ffba: e1dd jsr elf0 jsr elfd jmp e22b jsr
 e23f jsr e24e jsr

ffbd: e1d6 jsr e21b jsr e261 jmp

ffc0: e1c1 jsr

ffc3: e1cc jsr

ffc6: e11e jsr

ffc9: e4ae jsr

ffcc: a447 jsr abb7 jsr e37b jsr f6f4 jsr
 f716 jsr

ffcf: e112 jsr

ffd2: e10c jsr f135 jsr f5c9 jsr f726 jsr
 f759 jsr

ffd5: e175 jsr

ffd8: e15f jsr

ffdb: aala jmp

ffde: af84 jsr

ffe1: a82c jsr f4f9 jsr f62e jsr f8d0 jsr
 fe61 jsr

ffe4: e124 jsr

ffe7: a660 jsr

ffea: ea31 jsr

fff0: aae9 jsr aa fa jsr b39f jsr


```
fff3: e09e jsr
```

```
ffffa: jmp () durch CPU 6510 bei Auslösen eines NMI
```

```
ffffc: jmp () durch CPU 6510 bei Auslösen eines Reset
```

```
ffffe: jmp () durch CPU 6510 bei Auslösen eines IRQ
```


Kapitel 3

Die Firmware des C64

Dieses Grundlagenkapitel stellt die Funktionselemente der C64-Firmware vor. Die zahlreichen Begriffserklärungen und Definitionen erleichtern nicht nur das Verständnis des ROM-Listings, sondern bieten auch einen optimalen Einstieg in die Beschäftigung mit diesem.

Zuvor noch ein Rat an die sehr erfahrenen Leser: Auch wenn Ihnen manche Stellen in diesem Kapitel als »unter Ihrem Niveau« erscheinen, sollten Sie es zumindest »überfliegen«; später dient es auf jeden Fall zum Nachschlagen.

3.1 Grundbegriffe »Hardware«, »Software« und »Firmware«

Die drei »ware«-Begriffe (Hardware, Software, Firmware) sollen als erste vorgestellt werden.

Als »Hardware« bezeichnet man das gesamte »Material« eines Computersystems, also z.B. die Geräte (Computer, Drucker, Floppy, Monitor, auch Joystick, Maus, Lightpen, Datasette usw.). Die Faustregel »Alles an einem Computer, was man anfassen kann, ist Hardware« darf jedoch nicht mißverstanden werden: Eine Diskette selbst gilt zwar als Hardware (man kann sie ja mehr oder weniger sanft berühren), die auf ihr enthaltenen Daten und Programme hingegen sind »Software«. Mit diesem Begriff bezeichnet man also beispielsweise Programme, Dateien und ähnliches, was zwar am Bildschirm zu sehen oder auf Diskette abgespeichert ist, jedoch nur aufgrund der Hardware sichtbar wird; ohne ein Diskettenlaufwerk können Sie beispielsweise keine Programme von Diskette in den Computerspeicher oder auf den Bildschirm bringen.

»Software« ist also nicht aus Material, sondern gewissermaßen durch eine reine Denkleistung entstanden.

Tabelle 3.1 gibt weitere Beispiele zur Unterscheidung. Anhand der Tabelle können Sie leicht nachvollziehen, daß zwar keine Software ohne Hardware abläuft (mindestens ein Computer ist also erforderlich), aber umgekehrt auch die beste Hardware ohne die entsprechende Software nutzlos bleibt: Was hilft ein Joystick, wenn

Hardware	Software
Computer	Befehlseingabe wie LOAD"\$",8
Floppy	Kopierprogramm
Drucker	Textverarbeitungsprogramm
Joystick	Spielprogramm (z.B. Soccer)
Monitor bzw. Fernseher	Grafikprogramm (z.B. Hi-Eddi)
Floppy-Parallelkabel	Floppy-Speeder-Steuerprogramm

Tabelle 3.1: Beispiele für »Hardware« und »Software«

kein Spielprogramm vorhanden ist, das sich vom Joystick steuern läßt? Wofür hat man einen Drucker, wenn auf diesem kein Text ausgegeben werden kann?

Der nicht unbedingt bekannteste Begriff ist nun »Firmware«. Als Firmware bezeichnet man die hardwaremäßig eingebaute Software, die die Steuerung des Computers nach dessen Einschalten übernimmt. Wie wir soeben festgestellt haben, ist der Computer ohne entsprechende Software »tot«; deshalb muß er zumindest mit einem Programm versorgt sein, das den Aufruf weiterer Programme erlaubt.

Die Firmware des C64 ist in ROMs (Read Only Memory) eingebaut; diese nicht-löschbaren Speicherbereiche sind vom Augenblick der ersten Stromzufuhr an vorhanden und kommen zur Ausführung. Für Sie als Anwender macht sich dies dadurch bemerkbar, daß Einschaltmeldung und Cursor erscheinen.

Auf den Umfang der C64-Firmware geht Kapitel 3.2 ein; in Kapitel 1 finden Sie eine Dokumentation der Firmware als sogenanntes »ROM-Listing«.

3.2 Begriffe »Betriebssystem«, »Interpreter« und »Compiler«

Das zentrale Steuerprogramm, das für die Abwicklung der Ein- und Ausgabe von Daten letztlich zuständig ist, heißt »Betriebssystem«

(manchmal hört man auch den englischen Ausdruck »Operating System«).

Das Betriebssystem ermöglicht die Tastatureingabe, angefangen vom Anzeigen eines Cursors bis zur Sonderbehandlung von <SHIFT> + <RUN/STOP>. Genauso kann die Eingabe von externen Geräten (z.B. Floppy, Datasette) erfolgen; in diesem Fall spricht man davon, daß Programme bzw. Dateien »geladen« werden.

Die Datenausgabe erfolgt primär auf den Bildschirm, ist aber auch in Richtung Drucker, Diskette oder Datasette möglich.

Fast alle C64-Programme stützen sich auf die durch dieses Betriebssystem bereitgestellten Funktionen; nur wenige Ausnahmen verfügen über ein eigenes »Betriebssystem«.

Durch eine Veränderung des Betriebssystems, wie sie von manchen Floppy-Beschleunigern vorgenommen wird, ändert sich der Umgang mit dem Computer ganz beträchtlich. Im Verlaufe dieses Buches werden wir sogar das Betriebssystem erweitern.

Ganz allgemein gesehen ist das Betriebssystem immer die softwaremäßige Grundlage eines Computers; ohne ein Betriebssystem »läuft nichts«. Zu einem Betriebssystem gehören jedoch auch noch Programme, die seine Möglichkeiten ausnutzen. Für das C64-Betriebssystem gibt es ja Berge von Software . . .

Ein weiterer Begriff, der bei anderen Betriebssystemen (z.B. GEOS) eine große Rolle spielt, sei hier erwähnt: Programme, die von den Funktionen eines Betriebssystems Gebrauch machen, heißen »Applikationen« des betreffenden Betriebssystems. Die grundlegendste Applikation des C64-Betriebssystems ist der eingebaute Basic-Interpreter, der alle Ein-/Ausgabe-Angelegenheiten über das Betriebssystem abwickelt, selbst jedoch für die Ausführung und Auswertung der Ein-/Ausgaben verantwortlich zeichnet.

Der Begriff »Basic-Interpreter« (Basic-Dolmetscher) sagt aus, daß dieser Teil der Firmware Eingaben in der Programmiersprache Basic annimmt und insofern »übersetzt«, als er sie zur Ausführung bringt. Wie das Betriebssystem, ist auch der Basic-Interpreter ein Maschinenprogramm.

Als Alternative zu einem Interpreter gibt es noch »Compiler« (auch für das C64-Basic!). Diese übertragen die Basic-Anweisungen in äquivalente Maschinensprache-Befehle; zum Ablauf der »Compile« ist kein Interpreter mehr erforderlich, da es sich bereits um fertigen Maschinencode handelt, der sich auf das Betriebssystem stützt.

Der Vorteil eines Interpreters liegt darin, daß zum Austesten eines Programms kein Kompilationsvorgang erforderlich ist; kompilierte Programme laufen hingegen schneller ab, da keine zeitraubende Interpretation anfällt.

Kurz: Der C64 verfügt über einen Basic-Interpreter; zur Beschleunigung sind Compilerprogramme separat erhältlich (z.B. »Austro-Comp«).

Die C64-Firmware setzt sich somit aus dem Betriebssystem und dem Basic-Interpreter zusammen (Gesamtlänge: 16 Kbyte). In 3.3

zerpflücken wir das Betriebssystem und in 3.4 den Basic-Interpreter, während in 3.5 der Zusammenhang zwischen diesen beiden Firmware-Teilen unter die Lupe genommen wird.

3.3 Das Betriebssystem (Kernal)

Grob gesehen liegt das Kernal-ROM des C64 im Bereich \$e000-\$ffff. Deshalb wird oft auch »Kernal« anstelle von »Betriebssystem« gesagt. Eine genaue Unterscheidung erübrigt sich.

Die Schreibweise »Kernal« ist zwar mittlerweile quasi abgeschafft (jetzt schreibt man »Kernel«), doch sie hat sich in C64-Kreisen bereits so stark festgebissen, daß ich diesen Standard nicht brechen möchte. Zudem ist dieses Buch gewissermaßen als Ergänzung zu »Alles über den C64« zu sehen, und in letzterem Werk wird ebenfalls »Kernal« geschrieben.

In diesem Abschnitt 3.3, der in zahlreiche Unterkapitel gegliedert ist, werden verschiedene Funktionen des Betriebssystems erklärt.

3.3.1 Die Kernal-Sprungtabelle

Ab \$ff81 befindet sich im Speicher eine »Tabelle« von JMP-Anweisungen, die an alle möglichen Stellen im Betriebssystem verzweigen. Es handelt sich hierbei um die Kernal-Sprungtabelle, die der standardisierten Kommunikation zwischen Applikationssoftware und dem Betriebssystem dient. Diese Kernal-Sprungtabelle ist eine alte Tradition der Commodore-Heimcomputer.

Ein Maschinenprogramm, das nur die Kernal-Sprungtabelle und keine sonstigen Betriebssystem-Einsprünge aufruft, muß vom C64 auf den VC 20, C16/116/Plus 4 oder den C128 nicht angepaßt werden (höchstens ist es auf andere Speicher-Gegebenheiten einzurichten).

Natürlich stehen beim C16 oder C128 andere JMP-Befehle als in der Kernal-Sprungtabelle des C64, doch die Wirkung der angesprochenen Routinen ist immer dieselbe.

Andere Betriebssysteme lösen die zentrale Verteilung, indem jede Kernal-Routine eine Nummer bekommt, unter der sie aufgerufen wird; eine standardisierte Sprungtabelle ist allerdings erheblich besser zu handhaben.

Selbstverständlich ist es auch möglich, andere Einsprünge zu verwenden – dieses Buch stellt eine Vielzahl davon vor –, jedoch dürfen Sie dann nicht erwarten, daß diese Einsprünge

- auf jeder C64-Version gelten,
- auf anderen Commodore-Heimcomputern funktionieren,
- auf geänderten C64-Betriebssystemen (z.B. Floppy-Speeder wie Speed-Dos!) lauffähig sind.

Wenn Sie allerdings nur für Ihren eigenen Bedarf programmieren, ist die Sprungtabelle für Sie bedeutungslos, da Sie sich nur auf Ihre Betriebssystemkonfiguration einzurichten haben. Wollen Sie hingegen einen möglichst breiten Anwenderkreis ansprechen, so ist ein Kernalaufufr sicherlich eine wertvolle Hilfe, um diesem Zweck nachzukommen.

Da die Entwicklung des C64-Betriebssystems inzwischen abgeschlossen ist – zukünftige Versionen wird es nicht mehr geben – und Sie in der Regel zufrieden sind, wenn ein Programm auf jedem C64 läuft (für andere Commodore-Computer können Sie schließlich Anpassungen vornehmen), ist man also nicht immer auf die Kernalaufufr-Sprungtabelle festgelegt.

Es sei auch erwähnt, daß viele interessante Routinen nur über den direkten Aufruf und nicht über die Kernalaufufr-Sprungtabelle erreichbar sind. Dabei gelten die vorher zusammengefaßten drei Einschränkungen auf gleiche Weise.

Im ROM-Listing (Kapitel 1) können Sie ab \$ff81 die Sprungtabelle durchlesen, um eine Vorstellung zu bekommen, was für Einsprünge sich innerhalb dieser befinden.

3.3.2 Die IRQ-Routinen

Das Prinzip von Interrupts sollten Sie zwar schon kennen, es sei aber noch einmal kurz zusammengefaßt: 50mal pro Sekunde unterbricht der Prozessor das laufende Maschinenprogramm und führt eine sogenannte IRQ-Routine (ein weiteres Maschinenprogramm) aus, von dem wieder ins Hauptprogramm zurückgesprungen wird. Dadurch entsteht der Eindruck, beide Routinen (Hauptprogramm und IRQ-Routine) laufen gleichzeitig ab (»Pseudo-Multitasking«).

Im Interrupt erledigt das Betriebssystem beispielsweise die Abfrage der Tastatur, das Weiterzählen der Betriebssystem-Uhr, das Blinken des Cursors und manche Ein-/Ausgabe-Operationen.

Die Standard-IRQ-Routine beginnt bei \$ea31; zusätzlich gibt es noch die in Tabelle 3.2 aufgeführten IRQ-Routinen für bestimmte Sonderbehandlungen.

Adresse	Label	Funktion
\$fc6a	WRTZ	Synchronisationsmarke auf Kassette schreiben
\$fd9d	WRTN	Datenfile auf Kassette schreiben
\$f92c	READ	Datenfile von Kassette lesen

Tabelle 3.2: Spezielle IRQ-Routinen

Bei Bedarf schaltet das Betriebssystem also von der Standard-IRQ-Routine auf eine der Sonderbehandlungen im Interrupt. Sobald diese erledigt ist, wird wiederum die Standard-IRQ-Routine ab \$ea31 reaktiviert.

Auf diesen Standard-IRQ wollen wir, weil er den Normalzustand darstellt, etwas näher eingehen.

Folgende Tätigkeiten laufen im Standard-IRQ ab:

1. Abfrage der STOP-Taste

Die STOP-Taste dient dazu, ein Basic-Programm oder eine Ein-/Ausgabe-Operation abzubrechen (»BREAK«). Damit dies jederzeit möglich ist, muß die STOP-Taste zu jedem beliebigen Zeitpunkt überprüft werden; hierfür bietet sich der IRQ ja geradezu an.

Wenn im IRQ festgestellt wird, daß man die STOP-Taste betätigt hat, so wird ein Abbruch-Flag im Speicher gesetzt; die entsprechende Routine stellt dann außerhalb des IRQ fest, daß nun ein vorzeitiger Abbruch auszulösen ist.

Die IRQ-Routine jedoch greift nicht unmittelbar in das laufende Programm ein.

2. Weiterzählen der Betriebssystem-Uhr

In Basic wird die Betriebssystem-Uhr über die Variablen TI und TI\$ ausgelesen. Diese Betriebssystem-Uhr (»jiffy clock«) wird dadurch auf den neuesten Stand gebracht, daß sie bei jedem IRQ-Aufruf (also 50mal in der Sekunde) um 1 Einheit (»jiffy«) erhöht wird.

3. Cursor-Behandlung

Das regelmäßige Blinken des Cursors wird dadurch erreicht, daß in definierten Abständen (eingestellte Blinkzeit) das Zeichen an der Cursorposition invertiert wird. Sogesehen ist der Cursor kein selbständiges Zeichen, sondern nur durch das Umblenden des Zeichens an seiner Position erkennbar.

4. Kassettenmotor-Abfrage

Wird an der Datasette eine Taste ausgelöst, die den Motor startet, so geht dies nur über den Umweg des Betriebssystems. Erkennt dieses im IRQ die Aufforderung, den Kassettenmotor in Bewegung zu setzen, so wird über den Prozessorport (Adresse \$0001 = #1) dieser gestartet.

Zudem wird das entsprechende Flag gesetzt.

5. Tastaturabfrage

Nicht nur die STOP-Taste, sondern auch die restliche Tastatur wird im Interrupt abgefragt, damit prinzipiell zu jedem Zeitpunkt (außer bei Ein-/Ausgabe auf externe Geräte) ein Tastendruck berücksichtigt wird.

Dazu wird zunächst ermittelt, ob eine Taste betätigt wurde, und wenn ja, welchen ASCII-Code sie hat. Dieser wird dann in den sogenannten Tastaturpuffer geschrieben, einen Speicherbereich, der im IRQ mit allen erfolgten Tastendrücken »aufgefüllt« wird. Wenn das laufende Programm eine Betriebssystem-Routine zum Auslesen der Tastatur aufruft, so entnimmt diese lediglich dem

Tastaturpuffer den nächsten Tastendruck; die Hauptarbeit der Tastendekodierung hat bereits die IRQ-Routine geleistet.

3.3.3 Die Funktionsweise der universellen Routinen

Wenn Sie die Kernall-Sprungtabelle betrachten, fällt auf, daß manche Routinen schlicht Bezeichnungen wie »Eingabe eines Zeichens vom aktuellen Eingabegerät« tragen. Daraus läßt sich folgern, daß solche Routinen mit jedem Gerät zurechtkommen, das gerade als »aktuell« gilt.

Ein Vergleich mit dem PRINT-Befehl in Basic bietet sich an: Nach CMD gibt dieser seine Texte auf ein anderes Gerät aus.

Daher wollen wir im folgenden von »universellen Routinen« sprechen, wenn solche Routinen gemeint sind, die mit unterschiedlicher Peripherie zusammenarbeiten.

Dies sind im einzelnen:

```
OPEN ($ffc0), CLOSE ($ffc3), CHKIN ($ffc6),
CKOUT ($ffc9), BASIN ($ffc9), BSOUT ($ffd2),
LOAD ($ffd5), SAVE ($ffd8), GETIN ($ffe4)
```

Bei Aufruf dieser Routinen erfolgt eine Ausführung, die dem entsprechenden Gerät angemessen ist. Wird beispielsweise ein Kanal auf der Floppy eröffnet (OPEN \$ffc0), so existiert eine Floppy-Sonderbehandlung in der OPEN-Routine ab \$ffc0.

Der Vorteil dieser universellen Routinen ist somit die große Flexibilität: Nur durch die Vorbereitungen der Funktionsaufrufe wird festgelegt, welches Gerät anzusprechen ist.

Sieht man sich im ROM-Listing die Funktionsweise der universellen Routinen an, fällt auch sofort auf, daß diese mit einer Art »Verteiler« beginnen, der je nach aktuellem Gerät die entsprechende Routine zur Ausführung bringt.

Bei den einzelnen Routinenbeschreibungen in Kapitel 4 wird näher darauf eingegangen.

3.3.4 Die Initialisierung (Reset)

Nach dem Einschalten des Computers befinden sich in allen Speicherzellen mehr oder weniger zufällige Werte. Damit ein sinnvolles Ablaufen von Programmen überhaupt erst ermöglicht wird, erfolgt deshalb automatisch mit der ersten Stromzufuhr der Sprung in die sogenannte Reset-Routine (Rücksetzung). Dazu springt der Prozessor über den ROM-Vektor \$fffc/\$fffd, also bei unverändertem C64-ROM nach \$fce2. Die dortige Routine ist in der Routinenbeschreibung von Kapitel 4 genau zergliedert, hier soll uns ein Überblick über die Initialisierung genügen.

Zunächst wird der gesamte Prozessorstapel freigegeben. Daraufhin unterscheidet sich die weitere Behandlung dadurch, ob ein ROM-Modul vorliegt (Modulkennung ab \$8004 vorhanden) oder

nicht (Modulkennung nicht vorhanden). Im Falle eines ROM-Moduls wird dieses gestartet; ansonsten erfolgt die Initialisierung der Betriebssystemspeicher und der Register von VIC, SID und den CIAs. Anschließend wird der Basic-Interpreter angesprungen, der seinerseits die benötigten Arbeitsspeicher (wie die Basic-Programmezeiger, um nur ein Beispiel zu nennen) in einen Ausgangszustand versetzt.

Nach allen Initialisierungen erfolgt der Sprung in den Basic-Direktmodus (Eingabe von Programmzeilen oder Direktanweisungen).

Ein Reset kann auch durch »SYS 64738« über Software oder einen Resetschalter als Hardware-Erweiterung ausgelöst werden.

3.3.5 Die Fehlermeldungen und ihre Übermittlung

Bei den Ein-/Ausgabe-Operationen des C64-Kernal müssen zwangsläufig auch Fehler auftreten, z.B. weil ein Peripheriegerät nicht angeschlossen ist.

Die Routinen des C64-Kernal werden in der Regel nach den erforderlichen Vorbereitungen über JSR aufgerufen; nach dem Rücksprung aus der Routine gibt dann das Carry-Flag Auskunft darüber, ob ein Ein-/Ausgabe-Fehler aufgetreten ist (C=1) oder nicht (C=0).

Falls C=1 ist, so enthält der Akkumulator die entsprechende Fehlernummer, die aus Tabelle 3.3 zu entnehmen ist.

Fehler-nummer	Fehlertext	Beschreibung
\$00	BREAK	Abbruch durch <STOP>
\$01	TOO MANY FILES	Filetabelle voll
\$02	FILE OPEN	File schon offen
\$03	FILE NOT OPEN	File noch nicht offen
\$04	FILE NOT FOUND	File nicht vorhanden
\$05	DEVICE NOT PRESENT	Gerät nicht verfügbar
\$06	NOT INPUT FILE	kein Eingabefile
\$07	NOT OUTPUT FILE	kein Ausgabefile
\$08	MISSING FILENAME	fehlender Filename
\$09	ILLEGAL DEVICE NUMBER	unerlaubte Gerätenummer

Tabelle 3.3: Kernal-Fehlermeldungen (Fehlernummern werden im Akku übergeben)

3.3.6 Das Statusbyte (ST)

Bei Ein-/Ausgabe-Ereignissen, die lediglich die Funktionsweise einer angesprochenen Routine beeinträchtigen, jedoch nicht zu einem Abbruch mit Fehlermeldung (wie in 3.3.5 beschrieben)

führen müssen, wird im sogenannten Statusbyte das entsprechende Statusbit gesetzt.

In Basic wird das Statusbyte über die Systemvariable ST ausgelesen; sein Wert liegt in Adresse \$90 (siehe Beschreibung in Kapitel 6).

Eine wichtige Funktion ist dabei Bit 6 (End Of File = Datei-Ende); anhand dieses Bits wird das Ende eines Files rechtzeitig erkannt.

In Tabelle 3.4 finden Sie eine Zusammenstellung der einzelnen Statusbits, von denen auch mehrere zur selben Zeit gesetzt sein können.

Bit	Bedeutung, wenn Bit = 1	Peripheriegeräte
0	Fehler beim Schreiben	Drucker/Floppy
1	Fehler beim Lesen	Drucker/Floppy
2	Kurzer Block (»short block«)	Datasette
3	Langer Block (»long block«)	Datasette
4	unkorrigierbarer Lesefehler	Datasette
5	Prüfsummenfehler(»checksum«)	Datasette
6	Ende der Datei (»end of file«)	Datasette/Floppy
7	Band-Ende	Datasette
	Geräte nicht verfügbar	Drucker/Floppy

Tabelle 3.4: Fehlerbits im Statusbyte ST

Anhand dieser Tabelle sehen Sie auch, daß die RS-232-Schnittstelle nicht berücksichtigt wird. Diese hat ein eigenes Statusbyte (Adresse \$0297) mit den in Tabelle 3.5 aufgeführten Fehlerbits.

Bit	Bedeutung, wenn Bit = 1
0	Fehler bei Paritäts-Test
1	Fehler in der Bit-Folge
2	Überlauf des Eingabepuffers
3	leerer Eingabepuffer
4	kein CTS-Signal (Handshake)
5	unbenutzt
6	kein DSR-Signal (Handshake)
7	Übertragung ist unterbrochen

Tabelle 3.5: Fehlerbits im Statusbyte RSSTAT

3.3.7 Die Steuermeldungen

Bei vielen Ein-/Ausgabe-Operationen ist es unumgänglich, daß der Anwender zu einer bestimmten Handlung (z.B. Kassettenschalter betätigen) aufgefordert wird.

In solchen Fällen erscheinen am Bildschirm die entsprechenden Steuermeldungen des Betriebssystems. Folgende Steuermeldungen gibt es:

```
SEARCHING bzw. SEARCHING FOR [filename]
PRESS PLAY ON TAPE
PRESS RECORD & PLAY ON TAPE
LOADING
SAVING [filename]
VERIFYING bzw. VERIFYING OK
FOUND [filename]
```

3.3.8 Die Filetabelle

Wenn eine Datei geöffnet wird, so vollzieht sich dies in zwei Schritten:

- 1. Die Datei wird auf dem entsprechenden Peripheriegerät angelegt; z.B. öffnet die Floppy ein neues File.
- 2. Das Kernal merkt sich die Filespezifikationen in seiner Filetabelle.

Diese Filetabelle umfaßt maximal 10 Dateien; von diesen werden Filenummer, Geräte- und Sekundäradresse notiert. Später genügt es für alle Kernal-Routinen, die Filenummer anzugeben, damit das File ordnungsgemäß (richtiges Gerät, korrekte Sekundäradresse) ansprechbar ist.

Wird ein Eintrag in dieser Filetabelle gelöscht, so gilt das File für den Computer als geschlossen; auf dem externen Gerät existiert es jedoch nach wie vor. Deshalb ist die Kernal-Routine CLALL, die alle Fileeinträge aus der Filetabelle entfernt, kein Ersatz für das herkömmliche Schließen aller einzelnen Files über CLOSE.

Die Filetabelle befindet sich im Bereich \$0259-\$0276 und wird in Kapitel 6 detaillierter behandelt. Hier ging es nur darum, deren prinzipiellen Zweck zu verdeutlichen.

3.3.9 I/O-Beispiel: Drucker-Ausgabe

Zur Bildschirmausgabe (der Bildschirm ist das standardmäßige Ausgabegerät) werden die sogenannten »universellen Routinen« (siehe 3.3.3) aufgerufen. Um die Ausgabe umzulenken, muß zuerst ein Ausgabefile geöffnet und dann auf dieses die gesamte Ausgabe umgelenkt werden, damit bei weiteren Aufrufen der »universellen Routinen« (z.B. BSOUT zur Ausgabe eines einzelnen Zeichens) die Ausgabe auf das gewünschte Gerät erfolgt.

Um Ihnen einen Vorgeschmack zu geben, zeigt Listing 3.1 diese Vorgänge am Beispiel der Ausgabe auf den Drucker (Geräteadresse 4). Dieses Listing wurde mit dem Assembler Hypra-Ass erstellt.

hypra-ass assemblerlisting:

```

                100  -.li 1,8,2,"lst,s,w"
;
; drucker-ausgabe mit
; den kernal-routinen
;
                200  -.gl bsout = $ffd2 ; zeichen ausgeben
                210  -.gl setnam = $ffbd ; namen setzen
                220  -.gl setlfs = $ffba ; fileparameter setzen
                230  -.gl open  = $ffc0 ; file oeffnen
                240  -.gl ckout  = $ffc9 ; ausgabe umlenken
                250  -.gl clrchn = $ffcc ; standard-ein-/ausgabe
                260  -.gl close  = $ffc3 ; file schliessen
;
                500  -.ma print (text)
;
                1010 -.ba $c000 ; start: sys 49152
;
c000 a900      :1030 -      lda #0          ; keinen
c002 20bdff    :1040 -      jsr setnam       ; namen
;
c005 a904      :1060 -      lda #4          ; log. filenummer =4
c007 aa        :1070 -      tax              ; geraeteadresse 4
c008 a000      :1080 -      ldy #0          ; sekundaeradresse 0
c00a 20baff    :1090 -      jsr setlfs      ; parameter setzen
;
c00d 20c0ff    :1110 -      jsr open        ; file oeffnen
;
c010 a204      :1130 -      ldx #4          ; filenummer 4
c012 20c9ff    :1140 -      jsr ckout       ; ausgabe auf drucker lenken
;
                1160 -...print (text) ; text ausgeben
c015 a200      :510  -      ldx #0          ; offset initialisieren
c017 bd37c0    :520  -loop   lda text,x     ; byte aus text holen
c01a 20d2ff    :530  -      jsr bsout       ; ausgeben
c01d e8        :540  -      inx             ; offset erhoehen
c01e c900      :550  -      cmp #00        ; endmarkierung ?
c020 d0f5      :560  -      bne loop        ; nein (z=0): weiter
                570  -.rt
;
c022 20ccff    :1180 -      jsr clrchn      ; wieder bildschirmausgabe
;
                1200 -...print (text) ; jetzt auf bildschirm
c025 a200      :510  -      ldx #0          ; offset initialisieren
c027 bd37c0    :520  -loop   lda text,x     ; byte aus text holen

```



```

c02a 20d2ff :530 -      jsr bsout      ; ausgeben
c02d e8      :540 -      inx             ; offset erhoeihen
c02e c900    :550 -      cmp #00         ; endmarkierung ?
c030 d0f5    :560 -      bne loop        ; nein (z=0): weiter
                                570 -.rt
;
c032 a904    :1220 -     lda #4          ; log. filenummer 4
c034 4cc3ff :1230 -     jmp close        ; file schliessen
; & programm beenden
;
                                10000-text      .tx "beispieltext fuer die kernalroutinen
                                                zur textausgabe"
10010-.by 13,13,13,0 ; 3 * return, $00-endmarkierung
```

Listing 3.1: »drucker-ausg.src«

3.3.10 Steuerzeichen

Bei der Ausgabe von Texten ist die Unterscheidung zwischen »druckenden Zeichen« und »Steuerzeichen« zu treffen. Als »druckende Zeichen« gelten alle die Zeichencodes, deren Ausgabe über Routinen wie BSOUT (\$ffd2) am Bildschirm bzw. Drucker ein sichtbares Zeichen (Buchstabe, Ziffer, Symbol, Grafikzeichen) auslöst; »Steuerzeichen« hingegen lösen eine Steuerfunktion aus, z.B. löschen sie einzelne Zeichen oder den gesamten Bildschirm, verändern die Schriftfarbe, schalten zwischen Klein-/Groß-Zeichensatz und Groß-/Grafik-Zeichensatz um oder bewegen den Cursor an eine andere Position.

Trotz dieses grundsätzlichen Unterschiedes werden druckende Zeichen und Steuerzeichen auf gleiche Weise ausgelöst, in der Regel über »jsr bsout«. Nach diesem Aufruf prüft die BSOUT-Routine als erstes, ob ein Steuerzeichen vorliegt. Falls ja, so wird dieses ausgeführt; andernfalls wird das druckende Zeichen in den Bildschirmspeicher geschrieben, woraufhin es augenblicklich am Datensichtgerät (Monitor oder Fernseher) zu erkennen ist.

Hier seien kurz als Tabelle 3.6 diejenigen Steuerzeichen aufgeführt, die am Bildschirm Wirkung zeigen.

3.4 Der Basic-Interpreter

Nun wollen wir uns mit der Funktionsweise des Programms vertraut machen, das die Basic-Programmierung auf dem C64 ermöglicht.

Da Sie mit Sicherheit schon in Basic programmiert haben, dürfte Ihnen der Basic-Interpreter »von außen« nichts Neues sein, nun werden Sie zum »Insider«.

ASCII-Code	Bezeichnung	Wirkung
\$05	WHITE	Zeichenfarbe auf »weiß« stellen
\$08	LOCK	<SHIFT> + <CBM> blockieren
\$09	UNLOCK	<SHIFT> + <CBM> zulassen
\$0D	CR oder RETURN	an Anfang der nächsten Zeile springen
\$0E	BUSINESS	Klein-/Groß-Zeichensatz einschalten
\$11	DOWN	Cursor um 1 Zeile nach unten bewegen
\$12	RVS ON	Reversschrift einschalten
\$13	HOME	Cursor in Home-Position bewegen
\$14	DELETE	letztes Zeichen löschen
\$1C	RED	Zeichenfarbe auf »rot« stellen
\$1D	RIGHT	Cursor um 1 Spalte nach rechts bewegen
\$1E	GREEN	Zeichenfarbe auf »grün« stellen
\$1F	BLUE	Zeichenfarbe auf »blau« stellen
\$81	ORANGE	Zeichenfarbe auf »orange« stellen
\$8D	SHIFT CR	an Anfang der nächsten Zeile springen
\$8E	GRAPHICS	Groß-/Grafik-Zeichensatz einschalten
\$90	BLACK	Zeichenfarbe auf »schwarz« stellen
\$91	UP	Cursor um 1 Zeile nach oben bewegen
\$92	RVS OFF	Reversschrift ausschalten
\$93	CLR oder CLEAR	Bildschirm löschen

Tabelle 3.6: Die Steuerzeichen den C64 (Teil 1)

ASCII-Code	Bezeichnung	Wirkung
\$94	INSERT	1 Zeichen einfügen
\$95	BROWN	Zeichenfarbe auf »braun« stellen
\$96	LIGHT RED	Zeichenfarbe auf »hellrot« stellen
\$97	GREY 1	Zeichenfarbe auf »grau 1« stellen
\$98	GREY 2	Zeichenfarbe auf »grau 2« stellen
\$99	LIGHT GREEN	Zeichenfarbe auf »hellgrün« stellen
\$9A	LIGHT BLUE	Zeichenfarbe auf »hellblau« stellen
\$9B	GREY 3	Zeichenfarbe auf »grau 3« stellen
\$9C	PURPLE	Zeichenfarbe auf »purpur« stellen
\$9D	LEFT	Cursor um 1 Spalte nach links bewegen
\$9E	YELLOW	Zeichenfarbe auf »gelb« stellen
\$9F	CYAN	Zeichenfarbe auf »türkis« stellen

Tabelle 3.6: Die Steuerzeichen des C64 (Teil 2)

Wie beim Betriebssystem, wollen wir auch den Basic-Interpreter anhand einzelner Eigenschaften betrachten; den Überblick über die größeren Zusammenhänge haben Sie, wie gesagt, schon aus Ihrer Praxiserfahrung mit Basic 2.0 auf dem C64.

3.4.1 Die Initialisierung

Nach der Systeminitialisierung durch das Betriebssystem wird die Kaltstart-Routine des Basic 2.0 aufgerufen (ROM-Vektor \$a000/\$a001).

Diese initialisiert die Basic-Vektoren (\$0300-\$030b), alle RAM-Hilfsspeicher des Basic-Interpreters, gibt schließlich die Einschaltmeldung aus und führt den NEW-Befehl aus, wodurch der Basic-Programmierspeicher ausreichend initialisiert wird. Danach erfolgt der Warmstart (Eingabemodus).

Soweit die überblicksartige Beschreibung. Auf die Funktion »Vorbelegung der RAM-Hilfsspeicher« möchte ich noch etwas näher eingehen. Dazu zählt nämlich auch das Kopieren der CHRGET-Routine in den RAM-Bereich ab \$0073 sowie die Ermittlung der Größe des Basic-RAM, indem die erste ROM-Adresse ermittelt wird. Die Adresse darunter gilt dann als letzte RAM-Adresse. Diese Ermittlung der Grenze zwischen ROM und RAM ist programmtechnisch gesehen recht interessant und wird in Kapitel 4 dokumentiert.

3.4.2 Der Aufbau von Basic-Programmen im Speicher

Die Frage, wie sich der Basic-Interpreter ein eingegebenes Basic-Programm merkt, hat Sie sicherlich schon beschäftigt, und viele unter Ihnen werden sich damit bereits auseinandergesetzt haben. Betrachten Sie dann dieses Unterkapitel als günstige Gelegenheit, Ihr bereits vorhandenes Wissen auf systematische Weise aufzufrischen – ich hoffe, daß Ihnen meine gezielt knappen Erklärungen dabei entgegenkommen.

3.4.2.1 Ober- und Untergrenze des Basic-Speichers

Normalerweise beginnt der Basic-Speicher bei \$0800 und endet bei \$9fff. Nur durch die Kernal-Routinen MEMBOT und MEMTOP beziehungsweise durch direkte Manipulation der entsprechenden Hilfsspeicher kann diese Einstellung geändert werden.

In diesem Basic-Speicher liegen sowohl das Programm als auch die Variablen, wobei das Programm den Raum von \$0800 bis zu seiner Obergrenze (abhängig von der Programmlänge) beansprucht; daran schließen sich die Variablen an, auf die in 3.5 eingegangen wird.

Die Anfangsadresse des Basic-Programms wird in \$2b/\$2c abgelegt und ist normalerweise \$0801; die laufend veränderliche Endadresse ist aus \$2d/\$2e entnehmbar und stellt, wie soeben erwähnt, gleichzeitig die Anfangsadresse des Variablenspeichers dar.

3.4.2.2 Nullbyte vor dem Programmbeginn

Da der Basic-Speicher bei \$0800 beginnt, jedoch erst ab \$0801 das aktuelle Basic-Programm abgelegt wird, stellt sich die Frage, was es denn mit Adresse \$0800 auf sich hat. Dazu ist zu sagen, daß das Byte unmittelbar vor dem Programmbeginn immer ein Nullbyte ist; damit wird gewissermaßen das Ende einer Zeile markiert, die vor dem aktuellen Programmbeginn stehen könnte.

Wenn dieses Nullbyte mit einem anderen Wert überschrieben wird (z.B. POKE 2048,1), so funktionieren die Befehle »RUN« und »NEW« nicht mehr (probieren Sie's aus!).

3.4.2.3 Überblick über einen Zeileneintrag

Vereinfacht gesehen werden die Programmzeilen im Speicher in ihrer Reihenfolge abgelegt, wobei am Ende jeder Zeile ein Nullbyte als Zeilenendmarkierung steht.

Der Eintrag einer einzelnen Zeile ist wie in Tabelle 3.7 aufgebaut.

Bytes	Bedeutung
Bytes 0 und 1	Linkpointer
Bytes 2 und 3	Zeilennummer
Bytes 4 bis x	Zeileninhalt
Byte x plus 1	Zeilenendmarkierung
darauffolgend:	nächster Zeileneintrag

Tabelle 3.7: Aufbau eines Zeileneintrags im Basic-Programm

In dieser Tabelle befinden sich einige neue Begriffe wie »Linkpointer«. Die folgenden Unterabschnitte geben darüber ausführlichere Auskunft.

Als Grundlage für die dortigen Erklärungen sei hier jedoch ein Beispiel für ein Basic-Programm im Speicher gegeben (Listings 3.2/3.3).

```
100 rem *****
110 rem *** beispiel fuer den aufbau ***
120 rem ***   eines basic-programms   ***
130 rem ***       im speicher       ***
140 rem *****
150 :
160 print "beispieltext"
170 end
```

Listing 3.2: »basic-beispiel«

3.4.2.4 Linkpointer

Der Linkpointer einer Basic-Zeile – die beiden ersten Bytes eines Zeileneintrags – weist immer auf den Anfang der folgenden Zeile im Speicher, zeigt also auf das Byte nach der nächsten Endmarkierung.

Im Beispiel (Listing 3.3) besteht der Linkpointer von Zeile 100 aus den Bytes in \$0801/\$0802, also der Low-High-Darstellung von \$0828. Sehen wir nach \$0827 (= \$0828 – 1), so erkennen wir dort auch die nächste \$00-Zeilenendmarkierung und dahinter den Linkpointer von Zeile 110.

Die Linkpointer wären zwar von der Datenstruktur des Basic-Programms her nicht erforderlich, da die Endmarkierungen der Zeilen ebenso Auskunft über den jeweils nächsten Zeilenanfang geben; für eine schnellere Suche im Programm sind die Linkpointer jedoch sehr förderlich.

3.4.2.5 Zeilennummer

Unmittelbar hinter dem Linkpointer ist die Zeilennummer im Low-High-Format abgelegt. Im Beispiel aus Listing 3.3 enthalten die Bytes \$0803 und \$0804 im Low-High-Format den Wert \$0064, also die Zeilennummer 100 (= \$64).

3.4.2.6 Zeileninhalt

Hinter der Zeilennummer beginnt bekanntermaßen der Inhalt der Zeile. Im Fall von Zeile 100 in Listing 3.2/3.3 hat der Zeileninhalt folgendes Format:

```
8f 20 2a 2a 2a [weitere $2a-Bytes] 00
```

\$8f ist die Kodierung für den Befehl REM, \$20 das Leerzeichen hinter REM und die \$2a-Codes stehen für je ein Sternchen. \$00 ist die Zeilenendmarkierung.

```
:0800 00 28 08 64 00 8f 20 2a .(.... *
:0808 2a 2a 2a 2a 2a 2a 2a 2a  *****
:0810 2a 2a 2a 2a 2a 2a 2a 2a  *****
:0818 2a 2a 2a 2a 2a 2a 2a 2a  *****
:0820 2a 2a 2a 2a 2a 2a 2a 00  *****.
:0828 4f 08 6e 00 8f 20 2a 2a  O.... **
:0830 2a 20 42 45 49 53 50 49  * BEISPI
:0838 45 4c 20 46 55 45 52 20  EL FUER
:0840 44 45 4e 20 41 55 46 42  DEN AUFB
:0848 41 55 20 2a 2a 2a 00 76  AU ***..
:0850 08 78 00 8f 20 2a 2a 2a  ... ***
:0858 20 20 20 45 49 4e 45 53  EINES
:0860 20 42 41 53 49 43 2d 50  BASIC-P
:0868 52 4f 47 52 41 4d 4d 53  ROGRAMMS
:0870 20 20 2a 2a 2a 00 9d 08  ***...
:0878 82 00 8f 20 2a 2a 2a 20  ... ***
:0880 20 20 20 20 20 49 4d 20  IM
:0888 53 50 45 49 43 48 45 52  SPEICHER
:0890 20 20 20 20 20 20 20 20
:0898 20 2a 2a 2a 00 c4 08 8c  ***.....
:08a0 00 8f 20 2a 2a 2a 2a 2a  .. *****
:08a8 2a 2a 2a 2a 2a 2a 2a 2a  *****
:08b0 2a 2a 2a 2a 2a 2a 2a 2a  *****
:08b8 2a 2a 2a 2a 2a 2a 2a 2a  *****
:08c0 2a 2a 2a 00 ca 08 96 00  ***.\...
:08c8 3a 00 df 08 a0 00 99 20  :.....
:08d0 22 42 45 49 53 50 49 45  "BEISPIE
:08d8 4c 54 45 58 54 22 00 e5  LTEXT"..
:08e0 08 aa 00 80 00 00 00 20  .....
```

Listing 3.3: Monitor-Dump des Basic-Programms aus Listing 3.2

Das Leerzeichen zwischen Zeilennummer (100) und erstem Befehl (REM) wird im Speicher nicht vermerkt, sondern erst beim Listen automatisch ausgegeben, unabhängig davon, ob es vorher eingegeben wurde.

Wenn Sie sich noch die anderen Zeileninhalte ansehen, wird Ihnen auffallen, daß die Basic-Zeileninhalte überwiegend im ASCII-Code abgelegt werden. Eine Ausnahme bilden aber die Schlüsselwörter (REM, PRINT), die durch einen Bytewert zwischen \$80 und \$cb repräsentiert werden. Diese 1-Byte-Codes bezeichnet man als »Tokens«. \$8f ist also das Token für REM.

Eine Tabelle aller Tokens finden Sie als Tabelle 4.2 im nächsten Kapitel sowie im ROM-Listing (Kapitel 1) bei den Adressen \$a09e-\$a19d.

3.4.2.7 Programmende

Ein Sonderfall der Kombination von Zeilenendmarkierung und Linkpointer wurde bislang noch nicht besprochen: das Ende eines Basic-Programms. Dieses wird durch 3 Nullbytes markiert (Adressen \$08e4–\$08e6 im Beispiel). Dabei ist das erste Nullbyte die Endmarkierung der letzten Programmzeile, während die beiden darauffolgenden Nullen der Linkpointer der »Zeile hinter der letzten Zeile« sein müßten. Da es diese »allerletzte Zeile« allerdings nicht gibt, ist der Linkpointer mit der Endmarkierung \$0000 belegt.

3.4.3 Der Aufbau von Basic-Variablen im Speicher

Etwas schwieriger als die Struktur eines Basic-Programms im Speicher ist der Aufbau der Basic-Variablen. Der Grund für deren etwas kompliziertes Format liegt in den vielfältigen Variablentypen und der Array-Organisation. Folgende Variablentypen gibt es (in Klammern jeweils ein Beispiel für einen korrekten Variablennamen):

- 1. Fließkommavariablen (A)
- 2. Integervariablen (A%)
- 3. Stringvariablen (A\$)

Die Struktur der Variablen kennt zwei Erscheinungsformen:

- 1. einfache Variablen: A, A%, A\$
- 2. indizierte Variablen (Array-Variablen): A(0), A%(50), A\$(2)

Daraus ergeben sich folgende 6 Kombinationen:

- 1. einfache Fließkommavariablen: A, X, DD
- 2. einfache Integervariablen: A%, X%, DD%
- 3. einfache Stringvariablen: A\$, X\$, DD\$
- 4. indizierte Fließkommavariablen: A(5), X(3), DD(10)

- 5. indizierte Integervariablen: A%(5), X%(3), DD%(10)
- 6. indizierte Stringvariablen: A\$(5), X\$(5), DD\$(5)

Diese Variablen werden allesamt im Bereich vom Basic-Programmende bis zur Obergrenze des Basic-Speichers untergebracht. Die Reihenfolge der einzelnen Variablentypen entspricht der oben aufgeführten Einteilung: zuerst die einfachen, dann die indizierten Variablen.

Diese sequentielle Datenspeicherung beginnt unmittelbar nach dem Basic-Programmende und wird »nach oben« fortgesetzt; lediglich im Zusammenhang mit der Stringspeicherung wird sich eine kleine Ergänzung als notwendig erweisen.

3.4.3.1 Zeiger für den Variablenbereich

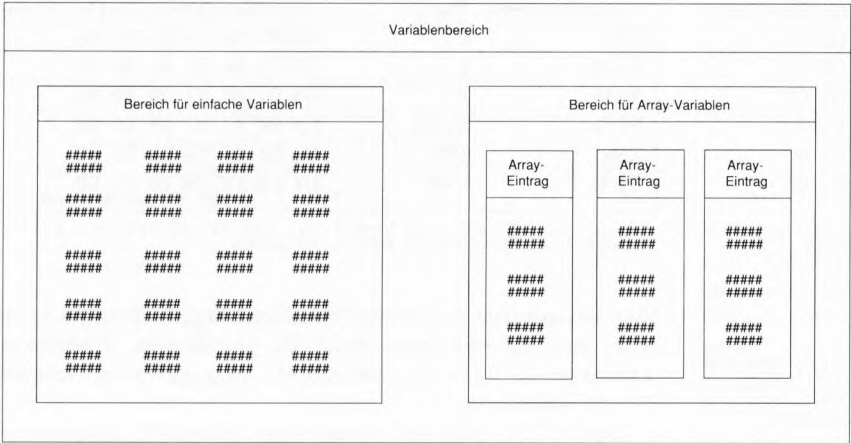
Folgende Hilfszeiger des Interpreters geben Auskunft, wo im Variablenspeicher welche Informationen zu finden sind (Tabelle 3.8):

Zeiger	Bedeutung der darin enthaltenen Adresse
\$2d/\$2e	Anfangsadresse des Variablenbereiches = Anfangsadresse der einfachen Variablen
\$2f/\$30	Anfangsadresse des Array-Bereiches = Endadresse der einfachen Variablen
\$31/\$32	Endadresse des Array-Bereiches + 1 = unterste freie Adresse für Stringinhaltsspeicherung
\$33/\$34	unterste Adresse für Stringinhaltsspeicherung (die Strings werden von hier an beginnend abgelegt)
\$37/\$38	oberste Adresse für Stringinhaltsspeicherung (gleichzeitig: oberste RAM-Adresse für Basic)

Tabelle 3.8: Hilfszeiger für den Variablenbereich

Diese Hilfszeiger stehen, wie aus der Tabelle ersichtlich ist, untereinander in bestimmten Abhängigkeiten. So kann man beispielsweise aus folgender Formel die Länge des Array-Bereiches entnehmen:
(Inhalt von \$31/\$32) – (Inhalt von \$2f/\$30)

Der Speicherbedarf der einfachen Variablen ergibt sich aus:
(Inhalt von \$2f/\$30) – (Inhalt von \$2d/\$2e)



Erläuterung:
= Eintrag einer einzelnen Variablen
(beliebiger Datentyp)

Abbildung 3.1: Aufbau des Variablenspeichers

3.4.3.2 Die Gliederung in Array- und Variableneinträge

Wie wir gesehen haben, ist der Variablenspeicher in diejenigen Bereiche, in denen bestimmte Variablentypen untergebracht sind, eingeteilt:

- a) Bereich für einfache Variablen
- b) Bereich für Array-Variablen

Im Fall a) handelt es sich um eine sequentielle Datenansammlung, in der jede neu angelegte Variable einen eigenen Variableneintrag zugewiesen bekommt. Ein solcher Variableneintrag ist somit die kleinste Dateneinheit: Er enthält Variablennamen und -inhalt von exakt einer Variablen.

Bei b) ist der Array-Bereich zunächst in Array-Einträge untergliedert; jeder Array-Eintrag informiert über ein Array und ist wiederum in Variableneinträge für die einzelnen Array-Elemente untergliedert.

Abbildung 3.1 verdeutlicht diese Datenstruktur noch einmal auf optische Weise. Im Grunde genommen ist diese Aufgliederung nicht besonders kompliziert, sondern nach einem klaren logischen Prinzip geordnet.

3.4.3.3 Eintrag einer Fließkomma-Variablen

Fließkomma-Variablen (»normale« Variablen, also ohne Prozent- oder Dollarzeichen im Variablennamen) können beliebige Werte

(auch mit Dezimalstellen und/oder negativem Vorzeichen) innerhalb des Bereiches [2.93873588 E-39;1.70141183 E+38] haben. Eine Bereichsüberschreitung nach oben führt zur Meldung OVERFLOW ERROR, Zahlen unterhalb des unteren Limits werden als 0 betrachtet.

Zur Darstellung einer Fließkommazahl benötigt der Computer 5 Byte, deren Format in 3.4.8 erläutert wird. An dieser Stelle aber genügt uns die Information, daß 5 Byte eine Fließkommazahl darstellen.

Ist nun eine einfache Variable oder ein Element eines Arrays eine Fließkommazahl, so liegt ein folgendermaßen aufgebauter Variableneintrag vor (Tabelle 3.9).

Vom Variablennamen werden also maximal zwei Zeichen berücksichtigt. Die Variable TEST ist demnach gleichbedeutend mit TEE.

Byte	Bedeutung	
Byte 1	1. Zeichen des Variablennamens im ASCII-Code	Name der Variablen
Byte 2	2. Zeichen des Variablennamens im ASCII-Code (\$00 = 1-Zeichen-Variablenname)	
Byte 3	Exponentenbyte der Fließkomma-Darstellung	Wert der Variablen
Byte 4	Mantissenbyte #1 der Fließkomma-Darstellung	
Byte 5	Mantissenbyte #2 der Fließkomma-Darstellung	
Byte 6	Mantissenbyte #3 der Fließkomma-Darstellung	
Byte 7	Mantissenbyte #4 der Fließkomma-Darstellung	

Tabelle 3.9: Eintrag einer Fließkommavariablen

Tabelle 3.10 gibt einige Beispiele anhand von exemplarischen Variablen:

Variablenname	Wert	Variableneintrag als Hex-Dump
X	1	58 00 81 00 00 00 00
XA	2	58 41 82 00 00 00 00
X1	3	58 31 82 40 00 00 00
PI	3.14159265	50 49 82 49 0F DAA1
Z	0.5	5A 00 80 00 00 00 00
TE	10	54 45 84 20 00 00 00

Tabelle 3.10: Beispiele für Fließkomma-Variableneinträge

Variablenname	Wert	Variableneintrag als Hex-Dump
X%	1	D8 80 00 01 00 00 00
XA%	2	D8 C1 00 02 00 00 00
X1%	3	D8 B1 00 03 00 00 00
PP%	31415	D0 D0 7A B7 00 00 00
Z%	500	DA 80 01 F4 00 00 00
TE%	- 40	D4 C5 FF D8 00 00 00

Tabelle 3.12: Beispiele für Intervervariablen

3.4.3.4 Eintrag einer Integer-Variablen

Intervervariablen werden durch ein nachgestelltes »%« (Prozentsymbol) gekennzeichnet und umfassen den Bereich von -32768 bis 32767 (nur ganzzahlige Werte, also keine Nachkommastellen).

Zur Darstellung eines solchen Wertes benötigt der C64 nur 2 Byte; da jedoch ein Fließkomma-Variableneintrag (siehe Tabelle 3.9) 7 Byte (2 Byte für den Variablennamen sowie 5 Byte für den Wert) Größe aufweist, sind die 3 restlichen Byte mit \$00 belegt. Daher ergibt sich bei Verwendung einer Intervervariablen kein geringerer Speicherbedarf, es sei denn, sie arbeiten mit Arrays (siehe 3.4.3.6).

Eine Besonderheit des Integer-Variableneintrags ist, daß in beiden Bytes des Variablennamens das 7. Bit gesetzt wird. Während der Variablenname »X1« also mit »58 31« dargestellt wird, steht »D8 B1« für »X1%«.

Tabelle 3.11 zeigt den Aufbau eines Integer-Variableneintrags, Tabelle 3.12 gibt einige Beispiele.

3.4.3.5 Eintrag einer String-Variablen

Da Strings sehr unterschiedliche Längen aufweisen können, wäre ein String-Variableneintrag, der außer dem Variablennamen auch

Byte	Bedeutung
Byte 1	1. Zeichen des Variablennamens im ASCII-Code + %10000000
Byte 2	2. Zeichen des Variablennamens im ASCII-Code + %10000000 (\$80=1-Zeichen-Variablenname)
Byte 3	HB des Integerwertes (nicht Low-High-, sondern High-Low-Format)
Byte 4	LB des Integerwertes (nicht Low-High-, sondern High-Low-Format)
Byte 5	\$00 als Füllbyte
Byte 6	\$00 als Füllbyte
Byte 7	\$00 als Füllbyte

Name der Variablen

Wert der Variablen

Tabelle 3.11: Eintrag einer Intervervariablen

Byte	Bedeutung
Byte 1	1. Zeichen des Variablennamens im ASCII-Code ohne Offset
Byte 2	2. Zeichen des Variablennamens im ASCII-Code + %10000000 (\$80=1-Zeichen-Variablenname)
Byte 3	Länge des Strings
Byte 4	LB der Adresse des String-Inhaltes
Byte 5	HB der Adresse des String-Inhaltes
Byte 6	\$00 als Füllbyte
Byte 7	\$00 als Füllbyte

Name der Variablen

Angaben zur Variablen

Tabelle 3.13: Eintrag einer Stringvariablen

den String-Inhalt angibt, hinsichtlich seiner Größe niemals konstant. Um deshalb die Stringverarbeitung möglichst effektiv zu gestalten, enthält der String-Variableneintrag nur die Adresse und Länge des String-Inhalts (und selbstverständlich den Variablennamen). Der String-Inhalt selbst ist in einem separaten Speicher untergebracht, auf den ich gleich zu sprechen komme.

Zunächst eine Zusammenstellung darüber, wie ein String-Variableneintrag aufgebaut ist, als Tabelle 3.13.

Gibt man unmittelbar nach dem Einschalten des C64 den Befehl

```
A1$ = "STRING"
```

ein, so ergibt dies folgenden Variableneintrag:

Byte 1/2:	\$41 \$B1	(Variablenname A1\$)
Byte 3:	\$06	(Stringlänge)
Byte 4/5:	\$9ffa	(Adresse des String-Inhalts)
Byte 6/7:	\$00 \$00	(ungenutzt)

In den Speicherzellen \$9ffa-\$9fff finden wir dann die ASCII-Darstellung der Buchstaben S, T, R, I, N und G. An dieser Lage im Speicher erkennt man sofort, daß die Strings vom oberen Basic-RAM-Ende (\$9fff) aus »nach unten« abgelegt werden. In diesem Stringinhaltspeicher befinden sich nur die Stringdarstellungen; die Deskriptoren (Stringlänge und Zeiger auf String-Adresse) findet man im herkömmlichen Variablenspeicher.

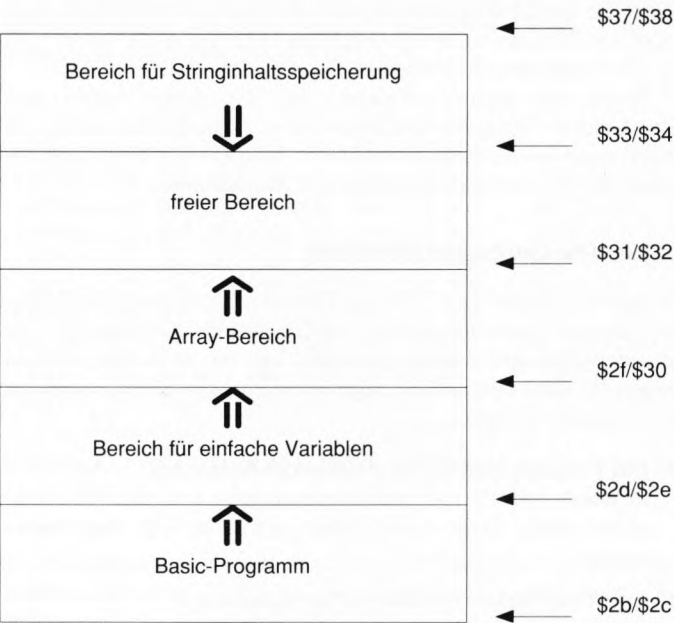


Abbildung 3.2: Aufteilung des Basic-RAM (\$0800-\$9fff)

Abbildung 3.2 zeigt nun eine modifizierte Speicheraufteilung für das Basic-RAM. Die Pfeile zeigen an, in welche Richtung ein Speicherblock vergrößert wird. Zudem können Sie aus Abbildung 3.2 noch einmal entnehmen, auf welche Speicherpositionen die Basic-Zeiger weisen.

3.4.3.6 Aufbau von indizierten Variablen (Arrays)

Während bei den einfachen Variablen der Einheitlichkeit halber jede Einzelvariable durch einen Eintrag von 5 Byte Länge (gegebenenfalls mit \$00 aufgefüllt) bestimmt wird, sind Arrays aus Gründen der Speicherwirtschaftlichkeit etwas »intelligenter« aufgebaut.

Jeder Array-Eintrag besteht zunächst aus 5 Kopfbytes, die einige allgemeine Aussagen über das Array treffen:

- Byte 1/2: Variablenname wie bei einfachen Variablen
- Byte 3/4: Länge des Array-Eintrags in Bytes im Low-High-Format
- Byte 5: Anzahl der Dimensionen im Array (1, 2 oder 3)

Darauf folgen dann im ungewöhnlichen High-Low-Format die Ausdehnungen des Arrays in den Dimensionen, wobei zuerst die letzte und zuletzt die erste Dimension angegeben wird. Die »Ausdehnung« in einer Dimension errechnet sich übrigens aus dem DIM-Parameter plus 1: Ein Array A(10, 5, 20) hat also in der 1. Dimension die Ausdehnung 11, in der 2. Dimension 6 und in der 3. Dimension 21.

Der entsprechende Array-Eintrag würde folgendermaßen aussehen:

Byte 1/2	(Variablenname A)	: \$41 \$00
Byte 3/4	(Länge des Array-Eintrags)	: \$1d \$1b
Byte 5	(Anzahl der Dimensionen)	: \$03
Byte 6/7	(Ausdehnung in 3. Dimension):	\$00 \$15
Byte 8/9	(Ausdehnung in 2. Dimension):	\$00 \$06
Byte 10/11	(Ausdehnung in 1. Dimension):	\$00 \$0b

Hinter einem solchen Array-Kopf folgen dann die Inhalte der einzelnen Elemente. Ein Integerelement benötigt 2 Byte, ein Stringelement 3 Byte (sowie einen Eintrag im Stringinhaltspeicher, versteht sich) und ein Fließkommaelement 5 Byte. Diese Byte-Angaben sind verbindlich; \$00-Auffüllungen kommen also im Zusammenhang mit Arrays nicht vor.

So ist auch der Array-Kopf möglichst speicherplatzextensiv: Liegen weniger als 3 Dimensionen vor, so wird nicht die Ausdehnung in den fehlenden Dimensionen mit »\$00 \$00« beziffert, sondern der Array-Kopf gekürzt. Zweidimensionale Arrays benötigen somit nur 9 Byte für den Kopfeintrag (Byte 6/7 = Ausdehnung in 2. Dimension, Byte 8/9 = Ausdehnung in 1. Dimension),

eindimensionale sogar nur 7 Byte (Byte 6/7 = Ausdehnung in 1. Dimension).

Als abschließendes Beispiel sehen Sie in Listing 3.4 ein Hex-Dump zum Array-Eintrag nach folgenden Direktmodus-Eingaben:

```
DIM TN% (2,3,2)
FOR F = 0 TO 2:FOR G=0 TO 3:FOR H=0 TO
2:TN%(F,G,H)=F*2000+G*300+H:NEXT H,G,F

:0818 D4 CE 53 00 03 00 03 00  I/S.....
:0820 04 00 03 00 00 07 D0 0F  .....7.
:0828 A0 01 2C 08 FC 10 CC 02  .....L.
:0830 58 0A 28 11 F8 03 84 0B  X.(.....
:0838 54 13 24 00 01 07 D1 0F  T.$...#.
:0840 A1 01 2D 08 FD 10 CD 02  ..-.../.
:0848 59 0A 29 11 F9 03 85 0B  Y.).....
:0850 55 13 25 00 02 07 D2 0F  U.%...-.
:0858 A2 01 2E 08 FE 10 CE 02  ...../.
:0860 5A 0A 2A 11 FA 03 86 0B  Z.*.....
:0868 56 13 26 0F 09 A5 D7 D0  V.&...0"
```

Listing 3.4: Hex-Dump zu einem Beispiel-Array

3.4.4 Der Editor

In diesem Unterkapitel möchte ich kurz die Funktionsweise der Eingabe eines Basic-Programms umreißen.

Die Eingabe der Zeile erfolgt durch Aufruf einer Kern-Routine; für die Bearbeitung der Tastatureingaben ist also zunächst das Betriebssystem verantwortlich. Dieses nimmt eine im Prinzip beliebige Zeichenfolge von bis zu 2 Zeilen Länge nach Bestätigung mit <RETURN> an.

Der Basic-Interpreter speichert diese Eingabe im Systemeingabepuffer ab \$0200 zwischen, um dann erst die Eingabe auszuwerten.

Dabei wird die Eingabe in das Token-Format gebracht (s. Kap. 3.4.2).

Zur Bearbeitung der Eingabe prüft der Interpreter zuerst, ob es sich um eine Basic-Zeile (Zeilennummer vorangestellt) oder eine Direktmodus-Anweisung (keine Zeilennummer vorangestellt) handelt.

Eingabe mit vorangestellter Zeilennummer

In diesem Fall wird ermittelt, ob auf die Zeilennummer eine Befehlsfolge – also der gewünschte Inhalt der bezifferten Zeile – folgt. Falls nein, so wird die entsprechende Zeile gelöscht, ansonsten wird die eingegebene Zeile ins Basic-Programm eingebunden.

Danach erfolgt ein erneuter Warmstart, d.h. die nächste Eingabe wird erwartet.

Eingabe ohne vorangestellte Zeilennummer

Solche Eingaben werden ausgeführt, indem die Interpreterschleife angesprochen wird. Vorher stellt der Basic-Editor die Zeiger auf die im Systemeingabepuffer befindliche Eingabe.

Nach der Ausführung der Eingabe erfolgt wieder ein Warmstart, um die nächste Anweisung entgegenzunehmen.

3.4.5 Die Interpreterschleife

Die Ausführung eines Basic-Programms wird von der »Interpreterschleife« übernommen, einem Teilprogramm des Interpreters, das gewissermaßen den Verteiler spielt: Alle Basic-Bytes werden von der Interpreterschleife an die richtigen Routinen weitergeleitet.

Befehle werden beispielsweise dadurch ausgeführt, daß aus der Adreßtabelle die Adresse der zum Befehl gehörenden Routine ermittelt und angesprochen wird.

Im Akku wird der Code des ersten Zeichens hinter dem Befehlstoken übermittelt, die Flags sind entsprechend diesem Code gesetzt. Die Routine zum entsprechenden Befehl wertet dann alle Parameter bis zum nächsten Befehl (erkennbar am Trennzeichen »:« oder am Zeilenende) aus und interpretiert diese. Alle Befehlsroutinen gelten als Unterrouinen der Interpreterschleife und springen über einen RTS-Befehl in die Interpreterschleife zurück.

Das Ende einer Zeile oder sogar des gesamten Programms erkennt die Interpreterschleife an den vereinbarten \$00-Markierungen. Auch ein Programmabbruch durch Auslösen der STOP-Taste wird in der Interpreterschleife erledigt.

Durch die zentrale Position und die große Anzahl von Durchläufen – nach der Ausführung jedes Basic-Befehls erfolgt ein Rücksprung in die Interpreterschleife – eignet sich diese Stelle im Interpreter besonders für tiefgreifende Manipulationen.

3.4.6 Die Garbage-Collection

Wenn ein in Basic geschriebenes Dateiverwaltungsprogramm (oder ein anderes Basic-Programm, das ungeheure Datenmengen im Hauptspeicher haben muß), plötzlich »tot« ist (es befindet sich in einem Wartezustand und ist auch nicht unterbrechbar), so gibt es drei mögliche Ursachen:

1. Das Programm ist in eine Endlosschleife geraten

In einem solchen Fall hilft die Betätigung von <RUN/STOP>, sofern diese Taste nicht vorher softwaremäßig abgeschaltet wurde.

2. Ein WAIT-Befehl ist fehlerhaft programmiert

Der WAIT-Befehl kann durch <RUN/STOP> nicht unterbrochen werden, gerät jedoch bei falscher beziehungsweise unvorsichtiger

Programmierung des öfteren in Endlosschleifen. Beispiel: WAIT 1,0.

3. Die »Garbage-Collection« hat zugeschlagen

Dieses Unterprogramm des Basic-Interpreters wird immer dann aktiv, wenn der Basic-Speicher voll ausgenutzt ist und Platz für eine weitere Variable geschaffen werden soll. Auf den ersten Blick erscheint es vielleicht verwunderlich, wieso es möglich ist, den Variablenspeicher zu optimieren. Beschäftigt man sich jedoch mit der internen Stringbehandlung, so erkennt man schnell, daß bei Stringoperationen jeder Teilergebnisstring im Stringinhaltsspeicher abgelegt wird. An einem Beispiel läßt sich dies gut zeigen. Geben Sie unmittelbar nach dem Einschalten die folgenden Befehle ein:

```
A$="STRING EINS"
B$="STRING ZWEI"
C$=A$+" PLUS "+B$
```

Man dürfte vermuten, daß jetzt folgende drei Strings im Stringinhaltsspeicher zu finden sind:

```
STRING EINS
STRING ZWEI
STRING EINS PLUS STRING ZWEI
```

Doch erinnern wir uns daran, daß auch Zwischenergebnisse aller Stringoperationen gespeichert werden, und sehen wir im Stringinhaltsspeicher nach (Abbildung 3.3).

:9FB0	00	00	00	00	00	00	00	53	S
:9FB8	54	52	49	4E	47	20	45	49	TRING EI
:9FC0	4E	53	20	50	4C	55	53	20	NS PLUS
:9FC8	53	54	52	49	4E	47	20	5A	STRING Z
:9FD0	57	45	49	53	54	52	49	4E	WEISTRIN
:9FD8	47	20	45	49	4E	53	20	50	G EINS P
:9FE0	4C	55	53	20	20	50	4C	55	LUS PLU
:9FE8	53	20	53	54	52	49	4E	47	S STRING
:9FF0	20	5A	57	45	49	53	54	52	ZWEISTR
:9FF8	49	4E	47	20	45	49	4E	53	ING EINS

Abbildung 3.3: Hex-Dump des Stringinhaltsspeichers nach Beispieloperationen

Im Stringinhaltsspeicher finden wir also folgende 5 Strings (in Anführungszeichen, dahinter in Klammern die Adressen im Stringspeicher):

```
String #1: "STRING EINS" ($9ff5-$9fff)
```

Inhalt von A\$. Der Stringdeskriptor von A\$ weist nach \$9ff5.

```
String #2: "STRING ZWEI" ($9fea-$9ff4)
```

Inhalt von B\$. Der Stringdeskriptor von B\$ weist nach \$9fea.

```
String #3: " PLUS " ($9fe4-$9fe9)
```

Dieser String wird bei der Verknüpfungsanweisung aus dem Basic-Text übergeben.

```
String #4: "STRING EINS PLUS " ($9fd3-$9fe3)
```

Bei Abarbeitung von C\$=A\$+" PLUS "+B\$ ist dies das erste Teilergebnis, wenn " PLUS " an A\$ angehängt wurde.

```
String #5: "STRING EINS PLUS STRING ZWEI"
($9fb7-$9fd2)
```

Hier erst finden wir das Ergebnis der Stringverknüpfung. Der Stringdeskriptor von C\$ weist nach \$9fb7.

In der abschließenden Wertung stellen wir fest, daß lediglich String #1, String #2 und String #5 benötigt werden: Auf diese Strings weisen Stringdeskriptoren, es handelt sich also um tatsächlich benötigte Variableninhalte.

String #3 und String #4 sind hingegen sogenannter String-Müll (engl. »garbage«), da sie als frühere Teilergebnisse nicht mehr von Belang sind.

Dennoch belegen Sie wertvollen Stringspeicherplatz. Dies wäre übrigens auch dann der Fall, wenn wir einen der Strings (A\$, B\$

:9FB0	00	00	00	00	00	00	21	53	!S
:9FB8	54	52	49	4E	47	20	45	49	TRING EI
:9FC0	4E	53	20	50	4C	55	53	20	NS PLUS
:9FC8	53	54	52	49	4E	47	53	54	STRINGST
:9FD0	52	49	4E	47	20	45	49	4E	RING EIN
:9FD8	53	20	50	4C	55	53	20	53	S PLUS S
:9FE0	54	52	49	4E	47	20	5A	57	TRING ZW
:9FE8	45	49	53	54	52	49	4E	47	EISTRING
:9FF0	20	5A	57	45	49	53	54	52	ZWEISTR
:9FF8	49	4E	47	20	45	49	4E	53	ING EINS

Abbildung 3.4: Stringinhaltsspeicher nach Garbage-Collection

oder C\$) mit einem neuen Wert belegen: Die alten Stringinhalte und Teilergebnisse bleiben im Stringinhaltsspeicher vorhanden, bis endlich die Garbage-Collection einsetzt. Über den Aufruf der FRE-Funktion mit einem String als Argument wird diese Garbage-Collection erzwungen. Sehen wir uns also das Ergebnis an, nachdem wir

```
PRINT FRE(" !")
```

eingegeben haben (Abbildung 3.4). Dann nämlich umfaßt der Stringinhaltspeicher nur noch die Adressen \$9fce–\$9fff; die Adressen davor sind mit denjenigen Werten belegt, die vor der Garbage-Collection an den entsprechenden Speicherstellen zu finden waren.

Im tatsächlich belegten Stringspeicher liegen jedoch nur noch die drei wirklich erforderlichen Strings "STRING EINS" (A\$), "STRING ZWEI" (B\$) und "STRING EINS PLUS STRING ZWEI" (C\$).

Zur Funktionsweise der Garbage-Collection ist nur zu sagen, daß sie alle diejenigen Stringinhalte aus dem Stringinhaltspeicher entfernt, auf die kein Stringdeskriptor aus einem Stringvariableneintrag weist. Dadurch reduziert sich der Speicherplatzbedarf auf die aktuellen Stringinhalte.

3.4.7 Die Parameterauswertung

Das Prinzip des Basic-Interpreters ist, daß er einen eingegebenen Befehl bei der Ausführung erkennt und ausführt. Zur Abarbeitung ist die Auswertung aller dem Befehl folgenden Parameter Voraussetzung.

Der Parameterauswertung ist daher ein sehr umfangreicher Teil des Basic-Interpreters gewidmet, den ich hier überblickartig vorstelle, damit bei Betrachtung der einzelnen Auswertungsroutinen der große Zusammenhang deutlich ist.

3.4.7.1 Die CHRGET/CHRGOT-Routine

Die elementarste Routine heißt CHRGET (»character get«, dt. »ein Zeichen holen«). Wie der Name schon aussagt, liest sie das nächste zu interpretierende Zeichen aus dem Basic-Programm ein. Davor wird der sogenannte CHRGET-Zeiger um eine Position nach vorne versetzt, damit auch das nächste Zeichen und nicht schon wieder dasselbe Byte gelesen wird.

CHRGOT liest das letzte über CHRGET eingeholte Zeichen ein; dabei wird der CHRGET-Zeiger unverändert gelassen.

3.4.7.2 Die FRMEVL-Routine

Fast so grundlegend wie CHRGET/CHRGOT ist die Auswertung eines beliebigen Ausdrucks (»FRMEVL« als Abkürzung für »Formula Evaluation«, deutsch »FormelAuswertung«).

Diese Routine liest den nächsten Parameter aus dem Basic-Text ein, unabhängig davon, um welchen Datentyp es sich handelt: Das Ergebnis der Auswertung kann sowohl ein String als auch ein numerischer Wert sein.

Wird der Datentyp eingeschränkt, so ist FRMNUM (numerische Ausdrücke auswerten) oder FRMSTR (Stringausdrücke auswerten) aufzurufen. Der Datentyp wird über CHKTYP geprüft (CHKNUM: Test auf »numerisch«; CHKSTR: Test auf »String«).

3.4.7.3 Sonderfall für numerische Parameter: Basic-Zeilenummer

Während die normale Parameterauswertung laut 3.4.7.2 auf FRMEVL gestützt ist, wurde für einen besonders wichtigen Ausnahmefall eine eigene Routine geschaffen. LINGET liest einen ganzzahligen numerischen Parameter im Bereich 0–63999 ein, der nur in Form von Ziffern angegeben werden darf. Variablen oder Rechenausdrücke sind also nicht zugelassen, wie Sie es ja als Basic-Programmierer von der Zeilennummernangabe kennen.

3.4.7.4 Auswertung numerischer Parameter innerhalb eingeschränkter Bereiche

Nur wenige Basic-Befehle erwarten numerische Parameter, die jeden Wert innerhalb des maximal zulässigen Wertebereiches für Fließkommazahlen haben dürfen. In den meisten Fällen sind Eingrenzungen unumgänglich, um Fehlfunktionen auszuschalten.

Die GETBYT-Routine beispielsweise liest nur Bytewerte (0–255) ein; Nachkommastellen werden ignoriert, Bereichsüber- oder unterschreitungen mit »ILLEGAL QUANTITY ERROR« quittiert.

GETWRB wertet zunächst ein Wort (0–65535), dann ein Komma als syntaktisch erforderliche Abgrenzung und zuletzt einen Bytewert (0–255) aus.

Wurde ein Fließkommawert über FRMEVL oder FRMNUM in den FAC gebracht, so wird er mit FACWRD in einen 2-Byte-Integerwert gewandelt.

3.4.7.5 Syntaktische Erfordernisse

Um die einzelnen Parameter eines Befehls voneinander abzugrenzen, werden herkömmlicherweise Kommas verwendet (Beispiel: WAIT 1,32,32).

Oft jedoch sind auch andere Trennzeichen erforderlich (Beispiel: PRINT A; 5*10; B).

Sehr häufig ist die Einklammerung von Ausdrücken, z.B. den Argumenten von Funktionen.

Insofern muß der Basic-Interpreter des öfteren nur feststellen, ob das nächste einzulesende Zeichen einen bestimmten Code hat. Dafür gibt es die CHKBYT-Routine, der im Akku der Vergleichscode übergeben wird, auf den hin das Zeichen an der aktuellen CHRGET-Position untersucht wird. Liegt der entsprechende Code vor, so er-

folgt ein ordnungsgemäßer Rücksprung, ansonsten die Meldung »SYNTAX ERROR«.

CHKCOM ruft CHKBYT mit dem Komma-Code im Akku auf und prüft demzufolge, ob ein Komma folgt. Weitere Routinen dieser Art sind CHKBRO (Klammer auf) und CHKBCL (Klammer zu).

3.4.8 Das Fließkommaformat

Fließkommazahlen (auch »Gleitkommazahlen« genannt) sind die flexibelste, aber gleichzeitig auch aufwendigste Form der Zahlendarstellung. Alle Rechnungen, die der Basic-Interpreter durchführt, werden im Fließkommaformat abgewickelt. Selbst wenn zwei Integervariablen miteinander verknüpft werden, geschieht dies über den Umweg der Fließkommadarstellung, da die entsprechenden Rechenroutinen nur auf dieses Zahlenformat ausgelegt sind.

Die Rechenroutinen für Fließkommazahlen sind in den meisten Maschinenprogrammen die einzigen Interpreter-Einsprünge, die aufgerufen werden. Eine Anwendung dieser Routinen bedingt nicht einmal die Kenntnis der mathematischen Grundlagen für Fließkomma-rechnungen, lediglich die verwendeten Speicherplätze sollte man einer Zeropagebelegung entnommen haben. Ebenso sollte man darüber informiert sein, daß im sogenannten MFLPT-Format eine Fließkommazahl in 5 Byte abgelegt wird.

In diesem Kapitel sollen jedoch die Einzelheiten des Fließkomma-Zahlenformates durchleuchtet werden, um ein Verständnis der entsprechenden ROM-Routinen zu erleichtern.

3.4.8.1 Zahlenformate

Das Fließkomma-Format ist nur eine Möglichkeit zur Darstellung einer Zahl. Weitaus häufiger findet man in der Maschinenprogrammierung die Verwendung von Integerwerten, also ganzen Zahlen, deren Bereich lediglich von der Anzahl der verfügbaren Bits abhängt. Der Prozessor selbst rechnet nämlich »integer«, und einen Fließkomma-Coprozessor hat der C64 bekannterweise nicht.

Wenn also andere Datentypen als »integer« verlangt sind, so kann sich ein Maschinenprogramm nur mit einer Simulation über den Umweg von Integerwerten behelfen. Die Struktur dieser Darstellung von Fließkommazahlen bezeichnet man dann als das »Fließkomma-Format«. In diesem wird beispielsweise der Wert 3.14159265 durch die Bytefolge »82 49 0F DA A1« repräsentiert.

Diese Bytefolge unterliegt also, vereinfacht gesprochen, einer festen Regel zur Speicherung mehrerer Integerwerte, die in ihrer Konstellation den Fließkomma-Zahlenwert bestimmen.

Blieben wir noch bei der Integerdarstellung. Diese ist in zwei grundsätzliche Typen zu unterteilen: vorzeichenlose (absolute) und vorzeichenbehaftete (»signed«) Werte. Vorzeichenlose Werte sind

sicher die häufigere Variante; ein vorzeichenloses Byte kann also alle Zahlen von 0 bis 255 (einschließlich) beinhalten. Ein vorzeichenbehaftetes Byte hingegen verwendet Bit 7 als Vorzeichenbit, wobei bei negativen Zahlen Bit 7 gesetzt ist und die Bits 0–6 komplementiert (invertiert) sind. Hier ein paar Beispiele für vorzeichenbehaftete Bytes:

```
%00000000 = $00 = 0
%00000001 = $01 = + 1
%01000001 = $41 = + 65
%01111111 = $7F = +127
%10000000 = $80 = -128
%11000000 = $C0 = - 64
%11101111 = $EF = - 17
%11111111 = $FF = - 1
```

Da mit einem vorzeichenbehafteten Byte nur Werte von –128 bis +127 möglich sind, ist auch hier eine Erweiterung auf mehrere Bytes möglich. –32768 ist z.B. %1000000000000000, +32767 z.B. %0111111111111111.

Durch die Erweiterung um zusätzliche Bytes ist zwar eine Bereichsvergrößerung zu erzielen, dennoch bleibt man auf ganze Zahlen beschränkt. Hier springt nun das Fließkommaformat ein, das Nachkommastellen zuläßt.

3.4.8.2 Mantisse und Exponent

Der erste Schritt zur »Zerlegung« eines Fließkomma-Wertes besteht darin, ihn in »Mantisse« und »Exponent« zur Basis 2 zu zergliedern. Dann kann mit der folgenden Formel der Wert wiederhergestellt werden:

$$\text{Mantisse} * 2^{\uparrow \text{Exponent}}$$

Folgendes Beispiel mag dies verdeutlichen:

```
Mantisse = 0.7
Exponent = 3
Wert      = 0.7 * 2↑3 = 0.7 * 8 = 5.6
```

Der Exponent ist eine vorzeichenbehaftete Integerzahl im Bytebereich, also ein Wert von –128 bis +127. Die Mantisse darf nach Definition nicht den Bereich von 0.5 bis 1 (bzw. –1 bis –0.5) verlassen, es handelt sich also um eine Dezimalzahl mit Nachkommastellen.

Bevor wir nun näher ins Detail gehen, sei eine Grobgliederung angegeben, welche Funktion die 5 Bytes einer Fließkommazahl ausüben:

```
Byte 1: Exponentenbyte
Byte 2–5: Mantissenbytes
```

3.4.8.3 Beispiel zur Berechnung der Mantissenbytes

Die Aufgliederung in Mantisse und Exponent war nur der rechnerische Anfangsschritt. Sie können sich jedoch mit Sicherheit vorstellen, daß das Hauptproblem darin besteht, Zahlen mit Nachkommastellen – also die Mantissen – ins Binärformat zu übertragen. »7« ist mit »%0111« leicht darstellbar, aber was ist mit »0.7«?

An einem praktischen Beispiel läßt sich das hierfür verwendete Rechenschema am verständlichsten erläutern. Nehmen wir einmal die Zahl »-13,2681«. Der Exponent ist 4, denn 2^4 ist 16 und eine Mantisse im Bereich 0,5–1,0 multipliziert mit 16 liegt im richtigen Zahlenbereich für unser Beispiel. Die Mantisse ergibt sich in direkter Abhängigkeit davon:

$$\text{Mantisse} * 2^{\text{Exponent}} = \text{Zahl}$$

$$\text{Mantisse} * 2^4 = -13,2681$$

$$\text{Mantisse} = -13,2681 / 2^4 = -13,2681 / 16 = -0,82925625$$

Diese Mantisse -0,82925625 ist nun in 4 Mantissenbytes aufzuteilen, denn das Exponentenbyte ist \$84 (4 ist der Exponent, doch bei positiven Exponenten ist Bit 7 im Exponentenbyte gesetzt).

Die Wandlung der Mantisse geschieht durch die sogenannte normalisierte Darstellung. Deren wichtigstes Kennzeichen ist die Tatsache, daß eine »linksbündige« Binärdarstellung entsteht, d.h. das höchstwertige Bit nach dem Komma muß gesetzt sein, während vor dem Komma nur Nullbits stehen dürfen.

Unsere Mantisse -0,82925625 wandeln wir nun dadurch in 4 Bytewerte um, daß wir die Nachkommastellen jeweils mit 16 multiplizieren und die Vorkommastelle als hexadezimale Ziffer verwenden:

0,82925625	x 16 =	13,26810000	Hex-Ziffer: D
0,26810000	x 16 =	4,28960002	Hex-Ziffer: 4
0,28960002	x 16 =	4,63360024	Hex-Ziffer: 4
0,63360024	x 16 =	0,13760380	Hex-Ziffer: A
0,13760380	x 16 =	2,20166016	Hex-Ziffer: 2
0,20166016	x 16 =	3,22656250	Hex-Ziffer: 3
0,22656250	x 16 =	3,62500000	Hex-Ziffer: 3
0,62500000	x 16 =	10,00000000	Hex-Ziffer: A

Hier wird die Rechnerei abgebrochen, da schon 8 Hex-Ziffern, also 4 Bytes, ermittelt wurden. Das negative Vorzeichen der Mantisse wurde in obiger Rechnung bewußt ignoriert, da es erst nach der Normalisierung berücksichtigt werden muß. Für die Mantisse erhalten wir also die normalisierte Darstellung »\$D4 \$4A \$23 \$3A«. Mit folgenden binären oder hexadezimalen Informationen ist demnach die Zahl »-13,2681« eindeutig beschrieben:

**Exponent: \$84 (Exponent 4; Vorzeichen des Exponenten: positiv
→ Offset \$80)**

**Mantisse: \$D4 \$4A \$23 \$3A (normalisierte Darstellung von
0,82925625)**

Vorzeichen der Mantisse: negativ

Exponent und Mantisse ergeben also exakt 5 Bytewerte. Stellt sich nun die Frage, wo das Vorzeichenbit der Mantisse untergebracht wird. Dafür gibt es zwei Möglichkeiten, die im nächsten Abschnitt vorgestellt werden.

3.4.8.4 FLPT- und MFLPT-Format

Bislang haben wir ganz allgemein vom »Fließkomma-Format« gesprochen, also von einer genau definierten Form der Zahlendarstellung. Dieses Fließkomma-Format ist jedoch noch unterteilt in FLPT- und MLFPT-Format. »FLPT« bedeutet dabei »Floating Point«, also nichts anderes als auch das deutsche Wort »Fließkomma«. »MLFPT« steht für »Memory Floating Point«, also »Speicher-Fließkomma«. Aus diesen Abkürzungen läßt sich somit schon der jeweilige Verwendungszweck ableiten: Das FLPT-Format dient zur Zahlenspeicherung bei Fließkomma-Rechnungen, d.h. die Fließkomma-Akkumulatoren sind entsprechend dem FLPT-Format strukturiert; Zahlen hingegen, die im Speicher gemerkt werden, wie z.B. Konstanten, unterliegen dem MFLPT-Format.

Am Beispiel aus 3.4.8.3 sehen wir die endgültige Zahlendarstellung in FLPT- und MFLPT-Format. Fassen wir vorher noch alle bislang gewonnenen Ergebnisse aus unserem Beispiel zusammen:

Zahl: -13,2681

Exponent: \$84 (Exponent 4, Vorzeichen des Exponenten ist positiv)

Mantisse: \$D4 \$4A \$23 \$3A (normalisierte Darstellung von
0,82925625)

Vorzeichen der Mantisse: negativ

FLPT-Format

Da mit dem FLPT-Format »gearbeitet« (sprich: gerechnet) wird, ist es nicht weiter verwunderlich, daß es auf schnelle Verarbeitbarkeit der Zahl ankommt und nicht auf speicherplatzökonomische Aufteilung.

Daher wird für das Vorzeichen der Zahl (= Vorzeichen der Mantisse) ein ganzes Byte aufgebracht (\$00 = positiv, \$80 bis \$ff = negativ). Insgesamt würde unsere Beispielzahl im FAC folgendermaßen untergebracht sein (Zahlen im FAC haben das FLPT-Format):

Adresse \$61 (Exponentenbyte) : \$84
 Adresse \$62 (Mantissenbyte 1) : \$D4
 Adresse \$63 (Mantissenbyte 2) : \$4A
 Adresse \$64 (Mantissenbyte 3) : \$23
 Adresse \$65 (Mantissenbyte 4) : \$3A
 Adresse \$66 (Vorzeichenbyte) : \$80

Im ARG (FAC #2) würden dieselben Werte stehen, allerdings in den Adressen \$69–\$6E.

Wir sehen somit, daß im FLPT-Format 6 Bytes für eine Zahl aufgewandt werden, wobei im sechsten Byte nur Bit 7 von Bedeutung ist.

MFLPT-Format

Den Luxus, für ein einziges Vorzeichenbit ein komplettes Byte aufzuwenden, erlaubt sich der Interpreter nur bei den Fließkomma-Akkumulatoren, da der erzielte Rechenzeitgewinn eindeutig gegenüber dem höheren Speicheraufwand überwiegt. Bei der Speicherung vieler Werte fällt ein um 20 Prozent erhöhter Speicherplatzbedarf jedoch schon ins Gewicht, und deshalb hat man sich einen Trick einfallen lassen, um das Vorzeichenbit anstatt in einem zusätzlichen Byte 6 innerhalb der fünf ersten Bytes unterzubringen. Sicherlich werden Sie sich fragen, wie dies denn möglich sein soll, wenn doch jedes Bit in den ersten fünf Bytes eine tragende Funktion hat.

Erinnern Sie sich daher bitte noch einmal an die Umwandlung der Mantisse. Wir sprachen von einer »Linksbündigkeit«, die durch »Normalisierung« erreicht werden sollte. Linksbündigkeit hieß dabei, daß unmittelbar hinter dem Komma ein gesetztes Bit stehen sollte. Daraus folgt wiederum, daß Bit 7 im ersten Mantissenbyte **immer** den Inhalt »1« hat. Insofern ist Bit 7 von Mantisse #1 kein Informationsträger, da wir dieses 1-Bit unabhängig vom numerischen Wert voraussetzen können. Der letzte Gedankenschritt ist vielleicht schon von Ihnen erkannt: Bit 7 in Mantisse #1 beinhaltet fortan das Vorzeichenbit. So kommt eine MFLPT-Zahl mit 5 Bytes Speicherbedarf aus.

Da beim negativen Vorzeichen des Beispiels Bit 7 wiederum zu setzen ist (0 = positiv, 1 = negativ), ist das MFLPT-Format von –13,2681 somit »\$84 \$D4 \$4A \$23 \$3A«. +13,2681 würde allerdings durch »\$84 \$54 \$4A \$23 \$3A« repräsentiert, da Bit 7 im ersten Mantissenbyte für positive Zahlen zu löschen ist.

3.4.9 Polynome und das Horner-Schema

Mit Hilfe des in 3.4.8 vorgestellten Fließkomma-Formates wickelt der Interpreter alle Rechenaufgaben ab. Er verfügt zunächst einmal über Routinen für die Grundrechenarten (Addition, Subtraktion, Multiplikation, Division); diese Routinen sind im Prinzip nur auf Fließkommazahlen ausgelegte Maschinenbefehle zur Verknüpfung der Mantissen- und Exponentenbytes, um die entsprechenden Rechenoperationen durchzuführen.

Bei Funktionen wie EXP, SIN und LOG jedoch ist es nicht mehr mit byteweisen Operationen möglich, die Ergebnisse zu berechnen. Hier setzen sogenannte Näherungspolynome ein. Ein Polynom ist in der Mathematik eine Reihenentwicklung von Potenzierungen, also z.B. » $x^5 + 7x^3 + 17$ «. Mit Hilfe solcher Formeln lassen sich

Näherungswerte für fast alle mathematischen Funktionen angeben, allerdings sind die entsprechenden Polynome erheblich komplizierter als das aufgeführte Beispiel.

Die allgemeine Form eines Polynoms lautet:

$$f(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + a_3 \cdot x^3 + \dots + a_N \cdot x^N$$

also nach Ausrechnen der beiden ersten Glieder:

$$f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + \dots + a_N \cdot x^N$$

$a_0 - a_N$ sind Koeffizienten (innerhalb des für Fließkommazahlen zulässigen Bereiches frei wählbar), N ist eine natürliche Zahl. Die Koeffizienten $a_0 - a_N$ und die Variable x (keine Basic-Variable) werden in dem bereits besprochenen MFLPT-Format (5 Bytes pro Zahlenwert) verarbeitet. Das sogenannte Horner-Schema beruht darauf, die Berechnung von Polynomen n -ten Grades (n bezieht sich dabei auf die höchste im Polynom auftretende Potenz) zu erleichtern und dadurch zu beschleunigen. Der obige allgemeine Ansatz läßt sich im Horner-Schema wie folgt formulieren:

$$f(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + \dots + x \cdot (a_{N-1} + a_N \cdot x) \dots)))$$

Durch dieses Ausklammern von x verringert sich die Anzahl der anfallenden Multiplikationen. Schon bei einem Polynom 4. Grades sind unter Anwendung des Horner-Schemas lediglich 4 statt 10 Multiplikationen erforderlich, um das Funktionsergebnis zu ermitteln.

Der Interpreter bzw. die Polynomberechnungsroutine des Interpreters setzen das Horner-Schema konsequent ein. Daher war es erforderlich, Ihnen diese Rechenmethode vorzustellen, wenn es sich auch um trockenen mathematischen Stoff handelt.

3.4.10 Fehler- und Steuermeldungen

Die Eingabe und Verarbeitung von Basic-Programmen ist im Grunde genommen eine Kommunikation zwischen dem Basic-Interpreter und dem Anwender oder Programmierer. Wenn bei diesem Dialog der Anwender/Programmierer etwas mitteilen möchte, so betätigt er die entsprechenden Tasten (oder führt Joystickbewegungen aus). Will der Interpreter sich zu Wort melden, so geschieht dies durch Meldungstexte, die am Bildschirm (oder einem anderen aktuellen Ausgabegerät) erscheinen. Vor allem bei fehlerhaften Eingaben ist dies vonnöten, damit der Anwender/Programmierer seinen Fehler bemerkt und nach Möglichkeit korrigieren kann. Andere Meldungen wiederum dienen als Signal, z.B. dafür, daß neue Eingaben getätigt werden dürfen.

Die gängigste Steuermeldung ist »READY.« (Eingabe-Prompt): Der Interpreter wartet auf neue Basic-Befehle oder Zeileneingaben. Ist eine INPUT-Eingabe nicht korrekt, erfolgt »REDO FROM START« (frei übersetzt: »nochmal von vorne eingeben«) oder

»EXTRA IGNORED«, wenn ein Teil der Eingabe aufgrund vorhandener Trennzeichen bewußt »vergessen« wurde.

Die meisten Interpretermeldungen jedoch sind Fehlertexte. Stellt die Routine zu einem bestimmten Befehl bei dessen Interpretation fest, daß Fehler vorliegen, so lädt sie die entsprechende Fehlernummer ins X-Register und ruft den allgemeinen Fehler einsprung auf. Dort wird der Text zur Fehlerbeschreibung ausgegeben und das Basic-Programm abgebrochen (Warmstart).

Folgende Fehlermeldungen kennt das Basic 2.0 des C64 (siehe Tabelle 3.14):

Fehlernummer	Fehlertext
1 (\$01)	TOO MANY FILES
2 (\$02)	FILE OPEN
3 (\$03)	FILE NOT OPEN
4 (\$04)	FILE NOT FOUND
5 (\$05)	DEVICE NOT PRESENT
6 (\$06)	NOT INPUT FILE
7 (\$07)	NOT OUTPUT FILE
8 (\$08)	MISSING FILENAME
9 (\$09)	ILLEGAL DEVICE NUMBER
10 (\$0A)	NEXT WITHOUT FOR
11 (\$0B)	SYNTAX!
12 (\$0C)	RETURN WITHOUT GOSUB
13 (\$0D)	OUT OF DATA
14 (\$0E)	ILLEGAL QUANTITY
15 (\$0F)	OVERFLOW
16 (\$10)	OUT OF MEMORY
17 (\$11)	UNDEF'D STATEMENT
18 (\$12)	BAD SUBSCRIPT
19 (\$13)	REDIM'D ARRAY
20 (\$14)	DIVISION BY ZERO
21 (\$15)	ILLEGAL DIRECT
22 (\$16)	TYPE MISMATCH
23 (\$17)	STRING TOO LONG
24 (\$18)	FILE DATA
25 (\$19)	FORMULA TOO COMPLEX
26 (\$1A)	CAN'T CONTINUE
27 (\$1B)	UNDEF'D FUNCTION
28 (\$1C)	VERIFY
29 (\$1D)	LOAD
30 (\$1E)	BREAK

Tabelle 3.14: Fehlermeldungen des Basic 2.0

3.4.11 Der Stapel als Hilfsmittel des Interpreters

In Ihren eigenen Maschinenprogrammen haben Sie den Stapel als bequemen Datenspeicher mit LIFO-Struktur (LIFO = »Last In, First Out«) schätzen gelernt.

Auch der Basic-Interpreter setzt diesen ausgiebig ein. Zum einen werden auf ihm zwischenspeichernde Werte (z.B. Registerinhalte) gemerkt, zum anderen dient er auch als »Basic-Stack«. Wie Sie wissen, speichert der Stapel bei JSR-Aufrufen in Maschinensprache die Rücksprungadressen; der Basic-Interpreter nun legt zusätzlich bei GOSUB-Aufrufen die entsprechenden Informationen (Zeilennummer und Programmstelle für Rücksprung) sowie bei FOR-Befehlen zum Öffnen einer FOR/NEXT-Schleife die diesbezüglichen Angaben auf dem Prozessorstapel ab. Daher spricht man auch vom »Basic-Stack«, der übrigens beim C128 einen eigenen Speicherbereich zugewiesen bekommen hat und somit nicht den Prozessorstapel belastet.

Halten wir also noch einmal fest, daß es zwei Arten von Basic-Einträgen im Stapel gibt: GOSUB/RETURN- und FOR/NEXT-Einträge. Diese werden jeweils bei Bedarf an der aktuellen Stapelposition untergebracht. Ist nicht mehr genügend Platz am Stapel vorhanden, erfolgt die Meldung »OUT OF MEMORY«, wie Sie nach der folgenden Eingabe, die ansonsten sinnlos ist, feststellen werden:

```
NEW
10 GOSUB 10
RUN
```

Das Prinzip dieser Eingabe ist klar: Ohne daß je ein RETURN erfolgt, das auch den Stapel vom GOSUB/RETURN-Eintrag befreien würde, wird solange ein weiterer GOSUB/RETURN-Eintrag angelegt, bis der Stapel überläuft. Die Anzahl der erfolgten Durchläufe wird mit folgendem Programm gemessen:

```
0 I=0
10 I=I+1:PRINT I:GOSUB 10
```

Als letzte Zahl sehen sie 24, also wurden 23 Durchläufe ohne Fehlermeldung ausgeführt.

Kommen wir nun zum Aufbau der beiden Arten von Basic-Einträgen im Stapel:

a) GOSUB/RETURN-Eintrag

Ein JSR/RTS-Eintrag des Prozessors belegt 2 Byte für die Rücksprungadresse. Etwas umfangreicher ist dieser Eintrag des Basic-Interpreters (Tabelle 3.15):

Byte 1	HB des CHRGET-Zeigers der Rücksprungadresse
Byte 2	LB des CHRGET-Zeigers der Rücksprungadresse
Byte 3	HB der Zeilennummer der Rücksprungstelle
Byte 4	LB der Zeilennummer der Rücksprungstelle
Byte 5	\$8D (GOSUB-Token) als Markierung des Interpreters

Tabelle 3.15: Aufbau eines GOSUB/RETURN-Eintrags

Ein GOSUB/RETURN-Eintrag verschwindet vom Stapel, sobald der RETURN-Befehl ausgeführt wird und der Rücksprung erfolgt ist. Ebenso entfernt der Prozessor einen JSR/RTS-Eintrag nach dem RTS-Rücksprung.

b) FOR/NEXT-Eintrag

Zur Bestimmung einer Schleife sind einige Informationen erforderlich (Tabelle 3.16):

Byte 1	HB des CHRGET-Zeigers des Befehls nach FOR
Byte 2	LB des CHRGET-Zeigers des Befehls nach FOR
Byte 3	HB der Zeilennummer der Fortsetzungsposition
Byte 4	LB der Zeilennummer der Fortsetzungsposition
Byte 5–9	MFLPT-Format des Schleifenendwertes
Byte 10	Vorzeichen der Schrittweite (= Schleifenrichtung)
Byte 11–15	MFLPT-Format der Schleifenschrittweite (STEP)
Byte 16	HB der Adresse der Schleifenvariablen
Byte 17	LB der Adresse der Schleifenvariablen
Byte 18	\$81 (FOR-Token) als Markierung des Interpreters

Tabelle 3.16: Aufbau eines FOR/NEXT-Eintrags

Ein FOR/NEXT-Eintrag wird erst dann vom Stapel gelöscht, wenn die Schleife komplett durchlaufen wurde (Überschreiten bzw. Erreichen des Schleifenendwertes).

Der Vollständigkeit halber sei auch erwähnt, daß die Bytes \$0100–\$013e bei Kassettenoperationen als Speicher für die Fehlerbeschreibungen dienen. Die Adressen \$00ff–\$010a werden zusätzlich noch bei der Umwandlung einer Fließkommazahl ins ASCII-Format verwendet, wodurch der Stapel im Bereich \$0100–\$010a tangiert ist.

Da die entsprechenden Adressen jedoch nur als herkömmlicher Arbeitsspeicher eingesetzt werden und die spezielle Speicherstruktur des Stapels darauf keinen Einfluß nimmt, finden Sie erst in Kapitel 5 (Speicherdokumentation) nähere Erläuterungen dazu.

3.5 Verknüpfungsstellen zwischen Betriebssystem und Basic-Interpreter

Wie Sie an der vorausgegangenen Beschreibung gesehen haben, arbeitet der Basic-Interpreter nicht für sich allein, sondern baut auf das Betriebssystem auf: **Jede Ein-/Ausgabe wickelt der Basic-2.0-Interpreter letztlich über Kernal-Aufrufe ab.**

Für die eigene Verwendung von Routinen des Basic-Interpreters (oder auch des Betriebssystems) sind daher die Verknüpfungsstellen von großem Interesse, um fehler- und absturzfrei programmieren zu können.

3.5.1 Speicherbereiche von Interpreter und Betriebssystem

Im allgemeinen nennt man den ROM-Bereich von \$a000 bis \$bfff »Interpreter-ROM« und denjenigen von \$e000 bis \$ffff »Betriebssystem-ROM« oder auch »Kernal-ROM«. Diese Grobgliederung ist jedoch sehr ungenau; es wäre ja auch erstaunlich, wenn zwei so komplexe Programme wie der Interpreter und das Kernal exakt dieselbe Länge an Programmcode belegen würden. Richtig ist zwar, daß beide »in etwa« 8 Kbyte lang sind, dabei wird aber zu oft vergessen, daß der Interpreter etwas größer als 8 Kbyte, das Kernal hingegen etwas kleiner ist. Somit kann die Aufteilung in zwei 8 Kbyte-Bereiche nicht funktionieren, wie Sie sicher einsehen werden.

Stattdessen belegt ein Teil des Interpreters Speicher im Kernal-ROM, nämlich von \$e000 bis \$e4b6. In diesem Bereich liegen folgende Programmteile:

- Fortsetzung der Routine zur Funktion EXP, die bei \$bfed beginnt
- Polynomroutinen POLYX und POLY
- Routinen zu den Basic-Anweisungen RND, SYS, SAVE, VERIFY, LOAD, OPEN, CLOSE, COS, SIN, TAN, ATN
- »Basic-Kernal-Aufrufe« (Beschreibung folgt in 3.5.2)
- Initialisierungsroutinen für Basic-Speicherbereiche (Vektoren, Arbeitsspeicher) inklusive Routine MSGNEW (Ausgabe der Einschaltmeldung)
- Basic-2.0-NMI-Einsprung

Diese ausgelagerten Interpreter-Routinen jedoch stützen sich ausgiebig auf Routinen im eigentlichen Interpreter-ROM. Daher ist es nicht möglich, bei ausgeschaltetem Basic-ROM (\$a000–\$bfff) bestimmte Routinen im Bereich \$e000–\$e4b6 mit Erfolg aufzurufen: Beim nächstbesten Aufruf eines Programmteils zwischen \$a000 und \$bfff ist ein Absturz vorprogrammiert.

Lediglich die Initialisierungsroutinen können verwendet werden, über den Nutzen läßt sich aber streiten.

3.5.2 Basic-Kernal-Aufrufe

Die Kernal-Sprungtabelle umfaßt bekanntlich Einsprünge für alle wichtigen Ein-/Ausgabe-Funktionen. Ist nun eine Interpreter-Routine (z.B. eine Befehlsroutine) auf die Verwendung eines Kernal-Einsprungs angewiesen, so wird dieser nur in den seltensten Fällen direkt angesprungen. Normalerweise erfolgt ein Aufruf eines »Basic-Kernal-Einsprungs« im Bereich \$e10c–\$e129. Dort stehen JSR-Befehle zu allen von Basic 2.0 benötigten Kernal-Routinen, bei denen I/O-Fehler auftreten könnten. Auf einen solchen JSR-Befehl folgt jeweils ein BCS-Befehl zum Testen des Fehlerbits C (C=Carry-Flag). Ist dieses gesetzt, so wird die spezielle Fehlerbehandlung bei \$e0f9 ausgelöst (EREXIT genannt), ansonsten erfolgt ein gewöhnlicher RTS-Rücksprung, da alles reibungslos abgelaufen ist. Eine Zusammenstellung der Basic-Kernal-Aufrufe ist Tabelle 3.17.

Basic-Kernal-Einsprung	Label	aufgerufene Kernal-Routine
\$e10c	BBSOUT	BSOUT (\$ffd2)
\$e112	BBASIN	BASIN (\$ffc6)
\$e118	BCKOUT	spezielle Routine bei \$e4ad
\$e11e	BCHKIN	CHKIN (\$ffc6)
\$e124	BGETIN	GETIN (\$ffe4)

Tabelle 3.17: Zusammenstellung der Basic-Kernal-Einsprünge

Außer diesen Einsprünge gibt es selbstverständlich noch eine Reihe weiterer Aufrufe von Betriebssystemroutinen, die der Inter-

preter ausführt. Diese Aufrufe können Sie der Cross-Referenz aus 2.4 entnehmen und dann gegebenenfalls im ROM-Listing (Kapitel 1) nach-schlagen.

3.5.3 Basic-ROM-Vektoren

Der Basic-Interpreter ist also ohne Betriebssystem nicht »lebens-fähig«. Stellt sich nun umgekehrt die Frage, ob sich das Betriebs-system auf Interpreter-Routinen stützt. Hierauf kann man mit einem klaren »Nein« antworten, muß jedoch die einzige Ausnahme ken-nen:

Die Routinen RESET (\$fce2) und NMI (\$fe43) lösen einen Kalt-(RESET) oder Warmstart (NMI) des Basic-Interpreters aus. Um den Interpreter zu aktivieren, müssen sie ihn logischerweise anspringen. Diese beiden Verzweigungen des Betriebssystems in den Interpreter liegen bei \$fcff und \$fe6f. Bei \$fcff springt die Reset-Routine über den Basic-Kaltstartvektor \$a000/\$a001, bei \$fe6f die NMI-Routine über den Basic-Warmstartvektor \$a002/\$a003. Diese beiden Vek-toren weisen dann ihrerseits in das Kernal-ROM \$e000–\$ffff, aller-dings in den von Interpreter-Routinen belegten Teil am Anfang des \$e000-Blockes.

Abgesehen davon nutzt das Betriebssystem keine einzige Inter-preter-Routine oder eine Datentabelle im Interpreter-ROM. Ledig-lich die »ausgelagerten« Interpreter-Routinen am Anfang des Kernal-ROM müssen zwangsläufig in den Bereich \$a000–\$bfff verzweigen, wie wir schon längst festgestellt haben. Auch hier gibt die Cross-Referenz (2.4) Auskunft.

Kapitel 4

Die ROM-Routinen im Detail

Vorbemerkung zu diesem Kapitel

Dieses Kapitel ist eine notwendige Erweiterung von Kapitel 1, dem eigentlichen ROM-Listing. Hier erfahren Sie noch einmal im Klartext,

1. welchen Zweck die einzelnen ROM-Routinen haben,
2. wie sie in programmtechnischer Hinsicht funktionieren,
3. welche Besonderheiten sie aufweisen,
4. wie sie angewendet werden.

Diese vier Punkte sprechen zwar das ROM-Listing auf seine Weise auch an, jedoch kann man nur in Verbindung mit diesem Kapitel von »vollständiger Information« sprechen. Bisherige Bücher dieser Art haben sich auf ROM-Listings beschränkt; hier soll jedoch ein neuer Standard für die Dokumentation von Basic-Interpreter und Betriebssystem des C64 gesetzt werden.

Damit die parallele Anwendung von Kapitel 1 und 4 möglichst unbelastet ist, geht auch Kapitel 4 »chronologisch«, also nach den Einsprungadressen der Routinen, vor. Die Eignung als Nachschlagewerk wird auch dadurch gefördert, daß an jeder Stelle in diesem Kapitel derselbe Kenntnisstand Voraussetzung ist. (Sie sollten Kapitel 3 gelesen haben.)

Bevor wir nun loslegen, sei noch eine kleine Bemerkung gestattet: Selbstverständlich werden hier auch die ROM-Tabellen sowie RAM-Programme (CHRGET \$0073) besprochen. Da jedoch diese ausschließlich im Zusammenhang mit ROM-Routinen Bedeutung tragen, ist die Kapitelüberschrift »Die ROM-Routinen im Detail« durchaus richtig.

CHRGET \$0073/CHRGOT \$0079:

Zeichen aus Basic-Text holen

Diese Routinen liegen zwar im RAM des C64, sind jedoch unverzichtbare Basis der ROM-Routinen und werden dazu in der Reset-Routine von der ROM-Quelle \$e3a2–\$e3b9 (siehe dort wegen Listing von CHRGET/CHRGOT) an diese Speicherplätze kopiert.

Nach »jsr chrget« befindet sich im Akku der Code an der aktuellen Position im Basic-Programm; die Prozessorflags enthalten erste

Informationen über diesen Code. Gleichzeitig zählt CHRGET (\$0073) den Zeiger auf die aktuelle Position (»CHRGET-Zeiger« genannt) um eine Position weiter.

CHRGOT (\$0079) hingegen liest den letzten über CHRGET (\$0073) eingeholten Wert ein, ohne den CHRGET-Zeiger zu erhöhen.

Nach Ausführung von CHRGET/CHRGOT haben die Prozessorflags bestimmte Werte, die einzig und allein vom eingelesenen Wert abhängen. Ist das Carry-Flag gelöscht, so handelt es sich um eine Ziffer (»0« – »9«, ASCII-Codes \$30–\$39); ein gesetztes Carry-Flag weist darauf hin, daß es sich um keine Ziffer handelt.

Falls das Zero-Flag den Wert »1« hat, liegt eine Befehls-/Zeilen-Endmarkierung vor (\$00 oder Doppelpunkt, ASCII-Code \$3a); bei Z=0 handelt es sich demnach um keine Endmarkierung.

Das N-Flag ist nur bei Werten, die kleiner als \$3a (Doppelpunkt) sind, gesetzt. Eine praktische Verwendung ergibt sich hierfür kaum.

Die Analyse der CHRGET-Routine ist mindestens so interessant wie ihre Anwendung. Wenn man nämlich CHRGET/CHRGOT näher untersucht, stellt man fest, daß als einziges Register der Akkumulator angesprochen wird. Die gerade auszulesende Adresse steht in der Routine selbst: Der LDA-Befehl bei \$0079 wird bei \$0073–\$0078 laufend modifiziert. Dies heißt »Selbstmodifikation«, weil sich das CHRGET-Programm selbst verändert. Diese Programmiertechnik habe ich im Artikel »Effektives Programmieren in Assembler« im 64'er-Sonderheft 8/85 (siehe Seite 74ff.) auch an anderen Beispielen vorgestellt; ich bin sicher, daß dieser Artikel eine Fundgrube für viele Insider ist, die noch ein paar Tricks zur Programmierung in Maschinsprache kennenlernen wollen.

Ein eigenes Anwendungsbeispiel zu CHRGET/CHRGOT ist noch Listing 4.1, das zwar mit Hypra-Ass assembliert wird, aber erst nach einem Reset gestartet werden kann. Die Syntax lautet:

```
SYS (49152) TEXT
```

Anstelle von **TEXT** können Sie beliebige Zeichen angeben, die danach auf den Bildschirm ausgegeben werden. Falls es sich um Ziffern handelt, erfolgt automatisch die Reversdarstellung.

```

READY.
100 -.BA $C000 : START: SYS (49152) TEXT
110 -;
120 -; NICHT VON HYPR-ASS AUS STARTEN.
130 -; SONDERN ERST RESET AUSLÖSEN UND
140 -; DANN STARTEN
150 -;
160 -;
170 -.GL CHRGET = $0073
180 -.GL CHRGOT = $0079
190 -;
200 -.GL BSOUT = $FFD2 : ZEICHEN AUSGEBEN
210 -;
220 -          LDA #13          : CARRIAGE RETURN
230 -          JSR BSOUT        : AUSGEBEN
240 -;
250 -          JSR CHRGOT       : ERSTES ZEICHEN HOLEN
260 -          JMP AUSGEBEN
270 -;
280 -START    LDA #13          : CARRIAGE RETURN
290 -          JSR BSOUT        : AUSGEBEN
300 -;
310 -          JSR CHRGET       : NAECHSTES ZEICHEN HOLEN
320 -;
330 -AUSGEBEN LDA #$92         : REVERS OFF
340 -          BCS PRINT        : C=1: KEINE ZIFFER
350 -          LDA #$12         : REVERS ON
360 -;
370 -PRINT    JSR BSOUT        : STEUERZEICHEN AUSGEBEN
380 -          JSR CHRGOT       : LETZTES ZEICHEN HOLEN
390 -          JSR BSOUT        : AUSGEBEN
400 -          JSR CHRGOT       : LETZTES ZEICHEN HOLEN
410 -          BNE START        : Z=0: KEINE ENDMARKIERUNG
420 -          RTS              : RUECKKEHR INS BASIC

```

READY.

Listing 4.1: Beispiel zu CHRGET/CHRGOT

ROM-Vektor für Basic-Kaltstart: \$a000/\$a001

Über diesen Vektor wird in der Reset-Routine bei \$fcff gesprungen (jmp (\$a000)); es erfolgt somit ein Sprung nach \$e394, wo sich die Kaltstart-Routine für Basic 2.0 befindet.

Voraussetzung für die Abarbeitung von \$fcff und folglich auch für die Verwendung dieses Vektors ist allerdings, daß keine Anti-Reset-Kennung vorliegt, weil sonst die Reset-Routine über den RAM-Vektor \$8000/\$8001 anstatt über diesen ROM-Vektor verzweigt.

Mit der Adresse \$a000 hat es eine besondere Bewandnis, wenn Sie von der recht beliebten Möglichkeit Gebrauch machen, das C64-ROM ins RAM an gleicher Adresse zu kopieren und auf RAM umzuschalten. Dann nämlich geht bei Auslösen eines Reset der Inhalt dieser Adresse \$a000 verloren, und wenn Sie nach dem Reset wieder mit »POKE 1,53« auf RAM-Betrieb schalten, kann es Probleme geben. Diese werden durch vorheriges »POKE 40960,148« (»POKE \$a000,\$94«) logischerweise umgangen, da durch diese Eingabe der richtige Inhalt von \$a000 wiederhergestellt wird.

Deshalb wird das Programm Hypra-Load (64'er, Ausgabe 10/84) nach einem Hardware-Reset nicht nur durch »POKE 1,53«, sondern durch »POKE 40960,148:POKE 1,53« reaktiviert.

Der Inhalt der Adresse \$a000 geht übrigens bei der Ermittlung der Grenze zwischen RAM und ROM verloren (siehe Dokumentation der Reset-Routine).

ROM-Vektor für Basic-NMI: \$a002/\$a003

Nach Betätigung von <RUN/STOP RESTORE> springt die NMI-Routine bei \$fe6f über diesen Vektor nach \$e37b.

Voraussetzung ist hier – wie beim ROM-Vektor für Basic-Kaltstart (\$a000/\$a001) – allerdings, daß keine Anti-Reset-Kennung im Speicher steht, damit nicht über \$8002/\$8003 (RAM-Vektor) anstatt über diesen ROM-Vektor verzweigt wird.

ROM-Kennung: \$a004–\$a00b

In diesen Adressen steht das ASCII-Format des Textes »CBMBASIC«. Es handelt sich hier um eine reine Copyright-Information ohne Einfluß auf die Funktionsweise des C64, denn diese Adressen werden von keiner Stelle im ROM angesprochen. Mir ist auch noch kein C64-Programm bekannt, daß von dieser ROM-Kennung irgendwelchen Gebrauch macht (außer zu Programmschutz-Zwecken, um den »Knacker« zu irritieren).

Sollten Sie also das ROM ins RAM übertragen, können Sie diese 8 Bytes ohne weiteres als Arbeitsspeicher für Ihr modifiziertes ROM einsetzen.

Adressen der Routinen zu den Basic-Befehlen: \$a00c–\$a051

Wie Sie aus Kapitel 3 wissen, werden die Basic-Befehle vom Interpreter als Tokens gespeichert. Dadurch ergibt sich eine feste Reihenfolge der Befehle:

```

END, FOR, NEXT, DATA, INPUT#, INPUT, DIM, READ,
LET, GOTO, RUN, IF, RESTORE, GOSUB, RETURN, REM,
STOP, ON, WAIT, LOAD, SAVE, VERIFY, DEF, POKE,
PRINT#, PRINT, CONT, LIST, CLR, CMD, SYS, OPEN,
CLOSE, GET, NEW

```

Als »Befehle« wollen wir in diesem Zusammenhang nur die soeben aufgeführten Schlüsselwörter bezeichnen; selbstverständlich haben auch Funktionen und Operatoren ihre Tokens, aber diese werden von der Adreßtabelle \$a00c–\$a051 nicht erfaßt. Definitionsgemäß ist ein »Befehl« also ein Basic-Schlüsselwort, das unmittelbar am Zeilenanfang oder hinter einem Doppelpunkt stehen kann.

Bei der Ausführung eines solchen Befehls liest die Interpreterschleife das Token des zu durchlaufenden Befehls ein. Dann wird anhand des Tokens die Position ermittelt, an welcher in dieser

Tabelle (\$a00c–\$a051) die Adresse der Befehlsroutine zu finden ist. Das Ergebnis ist ein Wert von 0 bis 34, da es ja 35 Befehle gibt. Dieser Wert wird dann bei \$a7f7 verdoppelt, weil jede Adresse 2 Bytes in der Tabelle beansprucht. Der dadurch erhaltene Offset kommt ins Y-Register, anhand dessen die Adresse aus der ROM-Tabelle auf den Stapel gelegt wird. Beim Aufruf der CHRGET-Routine über JMP erfolgt dann ein RTS aus CHRGET; um diesen auszuführen, holt der Prozessor die beiden obersten Bytes vom Stapel herunter und springt an die durch diese Bytes angegebene Adresse.

Dieser Stapelmanipulationstrick bedingt, daß in der ROM-Tabelle nicht die tatsächlichen Adressen der Befehlsroutinen stehen, sondern die um 1 verringerten Werte. Der Prozessor merkt sich nämlich die Rücksprungadressen für JSR/RTS in diesem Format (Rücksprungadresse – 1) am Stapel.

**Adressen der Routinen zu den Basic-Funktionen:
\$a052–\$a07f**

Nicht nur die Befehle, sondern auch die Funktionen tokenisiert der Basic-Interpreter. Auch die Funktionen haben somit eine durch die numerische Reihenfolge ihrer Tokens vorgegebene Reihenfolge:

```
SGN, INT, ABS, USR, FRE, POS, SQR, RND, LOG,  
EXP, COS, SIN, TAN, ATN, PEEK, LEN, STR$, VAL,  
ASC, CHR$, LEFT$, RIGHT$, MID$
```

Bei der Auswertung einer Funktion geht der Interpreter ähnlich wie bei Befehlen vor: Anhand des Token wird die Adresse der Routine aus der ROM-Tabelle entnommen und aufgerufen. Da jedoch bei der Funktionsauswertung kein Stapelmanipulationstrick wie bei der Befehlsinterpretation eingesetzt wird, stehen die Routinenadressen in dieser ROM-Tabelle im gewöhnlichen Low-High-Format, also ohne daß 1 subtrahiert wird.

Prioritätsflags und Routinenadressen zu den Basic-Operatoren: \$a080–\$a09d

Die Operatoren sind nach Befehlen und Funktionen nun endlich die dritte und letzte Gruppe von tokenisierten Basic-Schlüsselwörtern. Ihre Auswertung bereitet den größten Aufwand, da jeder Operator eine spezifische Priorität hat. Die bekannteste Prioritätsregel lautet »Punkt vor Strich« und besagt, daß zuerst alle »Punkt-Operationen« (Multiplikation, Division) und dann erst die »Strich-Operationen« (Addition, Subtraktion) auszurechnen sind. Als Beispiel eignet sich »3+10*5«. Hier würde es ohne die definierte Punkt-vor-Strich-Priorität zwei Lösungen geben:

- 1. $3+10*5 = 13*5 = 65$
- 2. $3+10*5 = 3+50 = 53$

Richtig ist natürlich, Sie wissen es schon, nur die zweite Lösung, da zuerst die Multiplikation »10*5« und dann die Addition zu berechnen ist.

Man spricht davon, daß die Multiplikation über eine höhere Priorität als die Addition verfügt. Die Division hingegen hat dieselbe Priorität wie die Multiplikation, und bei der Rechnung »3*10/5« ist es völlig belanglos, ob sie zuerst multiplizieren oder dividieren: In jedem Fall lautet das Ergebnis »6«.

In dieser Operatorentabelle besteht nun jeder Eintrag aus 3 Bytes: 1 Byte für die Priorität, 2 Bytes für die Adressen der Operatoren.

Die Adressen der Operatoren sind wie die Befehlsadressen um 1 dekrementiert, da zu deren Verwendung ein Stapelmanipulations-trick herangezogen wird, der dieses erfordert.

Die Priorität wird als 1-Byte-Wert gemerkt; je höher dieser Wert, desto höher ist auch die Priorität. Tabelle 4.1 gibt Auskunft über die Rangfolge der Prioritäten.

Rang	Operator	Prioritätsbyte
1	Potenzierung	\$7f
2	Vorzeichenwechsel	\$7d
3	Multiplikation und Division	\$7b
4	Addition und Subtraktion	\$79
5	Vergleichsoperatoren	\$64
6	NOT	\$5a
7	AND	\$50
8	OR	\$46

Tabelle 4.1: Die Prioritäten der Basic-2.0-Operatoren

**Tabelle der Basic-2.0-Schlüsselwörter für Tokenisierung:
\$a09e–\$a19d**

Damit nach der Eingabe einer Basic-Zeile die Tokenisierung (Umwandlung der Schlüsselwörter in Tokens) erfolgen kann, muß die Tokenisierungsroutine über den Klartext der Schlüsselwörter verfügen, welchen diese Tabelle \$a09e–\$a19d enthält. Die Tabellenstruktur ist äußerst effizient: In der Reihenfolge der Tokens sind die Klartexte enthalten, und das Ende jedes Klartextes wird dadurch markiert, daß im entsprechenden Byte das Bit 7 gesetzt ist.

Tabelle 4.2 enthält alle Tokens und die dazugehörigen Klartexte.

Aufbau der Tokens

Die Reihenfolge der Tokens ist sehr zweckgerichtet:

- Die Tokens \$80–\$a2 stehen für die Basic-2.0-Befehle, die in der Interpreterschleife ausgeführt werden, indem die Adresse der Befehlsroutine ausgelesen und angesprungen wird. Ist »X« das

Token	Klartext	Token	Klartext
\$80	END	\$a6	SPC(
\$81	FOR	\$a7	THEN
\$82	NEXT	\$a8	NOT
\$83	DATA	\$a9	STEP
\$84	INPUT#	\$aa	+
\$85	INPUT	\$ab	-
\$86	DIM	\$ac	*
\$87	READ	\$ad	/
\$88	LET	\$ae	↑
\$89	GOTO	\$af	AND
\$8a	RUN	\$b0	OR
\$8b	IF	\$b1	<
\$8c	RESTORE	\$b2	=
\$8d	GOSUB	\$b3	>
\$8e	RETURN	\$b4	SGN
\$8f	REM	\$b5	INT
\$90	STOP	\$b6	ABS
\$91	ON	\$b7	USR
\$92	WAIT	\$b8	FRE
\$93	LOAD	\$b9	POS
\$94	SAVE	\$ba	SQR
\$95	VERIFY	\$bb	RND
\$96	DEF	\$bc	LOG
\$97	POKE	\$bd	EXP
\$98	PRINT#	\$be	COS
\$99	PRINT	\$bf	SIN
\$9a	CONT	\$c0	TAN
\$9b	LIST	\$c1	ATN
\$9c	CLR	\$c2	PEEK
\$9d	CMD	\$c3	LEN
\$9e	SYS	\$c4	STR\$
\$9f	OPEN	\$c5	VAL
\$a0	CLOSE	\$c6	ASC
\$a1	GET	\$c7	CHR\$
\$a2	NEW	\$c8	LEFT\$
\$a3	TAB(	\$c9	RIGHT\$
\$a4	TO	\$ca	MID\$
\$a5	FN	\$cb	GO

Tabelle 4.2: Tokens des Basic 2.0 und dazugehörige Klartexte

Token eines Befehls, so ergibt folgende Formel die Adresse der Routine zu diesem Befehl:

```
PEEK($a00c+(X-$80)*2) +  
256*PEEK($a00c+(X-$80)*2+1) + 1
```

- Die Tokens \$a3–\$a7 sowie \$a9 repräsentieren Schlüsselwörter, die ausschließlich innerhalb bestimmter Befehlsroutinen interpretiert oder als syntaktische Erfordernisse aufgestellt werden: »TAB(« und »SPC(« bearbeitet die PRINT-Routine, TO/FN/THEN/STEP werden von den Routinen zu den Befehlen FOR/DEF/IF aus syntaktischen Gründen verlangt.
- Die Tokens \$a8 sowie \$aa–\$b3 ersetzen Basic-Operatoren.
- Die Tokens \$b4–\$ca entsprechen den Basic-Funktionen.
- Das Token \$cb schließlich ermöglicht es, das Schlüsselwort »GOTO« auch als »GO TO« zu schreiben, da »GO« als eigenes Token existiert und als solches von der Interpreterschleife erkannt wird.

Struktur der Tokenisierungstabelle

Um festzustellen, inwiefern die Reihenfolge der Tokens die Tokenisierung selbst erleichtert, müssen wir uns kurz über den Vorgang der Tokenisierung informieren.

Bei der Tokenisierung versucht der Interpreter, Schlüsselwörter im eingegebenen Basic-Text zu finden. Sobald er eine Zeichenfolge im Basic-Text entdeckt, die in derselben Reihenfolge auch als Eintrag in der Tokenisierungstabelle zu finden ist, ersetzt er die Bytefolge durch das Befehlstoken, also durch die Position des Klartextes innerhalb der Tokenisierungstabelle, wozu noch der Offset \$80 addiert wird. Daraufhin werden im Rest der Eingabe weitere Schlüsselwörter gesucht, bis die gesamte Basic-Eingabe durchforstet ist.

Als Beispiel wollen wir die Umwandlung der Eingabe »FOR I=1 TO 5« verfolgen:

Phase 1:

```
FOR I=1 TO 5
```

Phase 2:

```
[FOR-Token $81] I=1 TO 5
```

Phase 3:

```
[FOR-Token $81] I[""-Token $b2]1 TO 5
```

Phase 4:

```
FOR-Token $81] I[""-Token $b2]1 [TO-Token $a4] 5
```

In diesem Fall hat der Algorithmus zweifellos funktioniert. Schwierig wird es jedoch, wenn Schlüsselwörter in Variablenamen stehen. Diese werden dann nämlich, ob es sinnvoll ist oder nicht, ebenfalls tokenisiert:

```
TOM=7
```

ergibt folgendes:


```
[TO-Token $a4]M=7
```

Eine weitere Lücke des Tokenisierungsalgorithmus wird aber durch den Aufbau der Tokenisierungstabelle umgangen. Nehmen wir einmal den theoretischen Fall an, es gäbe die Schlüsselwörter AB und ABC. Wenn »AB« in der Klartext-Tabelle vor »ABC« steht, würde die Eingabe »ABC« also in

```
[AB-Token]C
```

umgewandelt. Die Folge ist absehbar: Das Schlüsselwort ABC könnte die Tokenisierungsroutine nie erkennen, da es bereits ein anderes Schlüsselwort – AB – enthält, das zuerst tokenisiert wird. Aber so ein theoretischer Fall liegt ja bei den Basic-Schlüsselwörtern gar nicht vor . . .

Doch! Die entsprechenden Paare lauten PRINT/PRINT#, INPUT/INPUT# und GO/GOTO. (GET und GET# sind hier nicht von Belang, da nur GET als Schlüsselwort gilt und das Doppelkreuz in der Praxis lediglich als GET-Parameter gilt.)

Dennoch läuft die Tokenisierung korrekt ab, da die Tokenisierungstabelle entsprechend geschickt aufgebaut ist. In ihr steht nämlich PRINT# vor PRINT, INPUT# vor INPUT und GOTO vor GO, so daß nichts »anbrennen« kann: Die längeren Schlüsselwörter werden zuerst tokenisiert. Für den Fall, daß das jeweils kürzere Befehlswort vorliegt, so erkennt es die Tokenisierungsschleife in einem späteren Durchlauf immer noch rechtzeitig.

Für den Aufbau eigener Tokenisierungstabellen von Basic-Erweiterungen müssen Sie sich also dieses Prinzip gut merken, weil sonst Fehlfunktionen in Ihren Programmen auftreten könnten, für die Sie keine (andere) logische Erklärung finden.

Der <SHIFT L>-Trick

Fast jede Zeitschrift hat schon darüber berichtet, daß man durch Eingabe von <SHIFT L> hinter einem REM-Befehl eine Art Listschutz einbaut, weil fortan beim Listen des entsprechenden REM-Befehls ein SYNTAX ERROR entsteht, obwohl die Zeile nach wie vor vom Interpreter ausgeführt werden kann.

Das Prinzip davon ist folgendes: <SHIFT L> wird bei der Tokenisierung unverändert gelassen und ergibt den Code \$cc im Basic-Speicher. Die LIST-Routine wiederum hat einen Programmierfehler, aufgrund dessen \$cc als Token aufgefaßt wird. Anstatt eines Schlüsselwort-Klartextes findet LIST jedoch die \$00-Endmarkierung der Tokenisierungstabelle, deren Ausgabe dann über Umwege zum SYNTAX ERROR führt. Bei der Beschreibung der LIST-Routine wird näher darauf eingegangen.

Tabelle der Basic-Fehlermeldungen im Klartext: \$a19e–\$a327

Diese Tabelle enthält die Klartexte der Basic-Fehlermeldungen im ASCII-Format. Im jeweils letzten Byte eines Befehlswortes ist Bit 7

als Endmarkierung gesetzt, wird aber anlässlich der Textausgabe ausgeblendet.

Die Texte der Fehlermeldungen sind in Reihenfolge der Fehlercodes angeordnet. Dies ist jedoch im Gegensatz zur Abfolge der Tokenisierungstabelle nicht einmal zwingend erforderlich, da sich ab \$a328 eine weitere Tabelle im Speicher befindet, die die Adressen aller Fehlermeldungen enthält und diese Zeigertabelle bereits richtig geordnet ist.

Die einzelnen Fehlertexte bestehen lediglich aus der Fehlerbeschreibung (z.B. »SYNTAX«). Der Zusatz »ERROR« oder »ERROR IN xxxx« wird in der Fehlerbehandlungsroutine erzeugt, da er für jede Fehlermeldung gleich ist.

Tabelle der Adressen der Basic-Fehlermeldungen im Klartext: \$a328–\$a363

Diese Tabelle enthält in Reihenfolge der Fehlercodes nicht nur die Adressen der Fehlermeldungstexte aus \$a19e–\$a327, sondern auch den Text zum Fehlercode #30 (BREAK ERROR), dessen Klartext bei \$a383 steht.

Die Adresse zum Text einer Basic-Fehlermeldung erhält man also über folgende Formel:

```
PEEK($a328+Fehlercode-1) +  
256*PEEK($a328+Fehlercode-1+1)
```

Vereinfacht:

```
PEEK($a327+Fehlercode) +  
256*PEEK($a328+Fehlercode)
```

Tabelle für Texte in Fehler- und Steuermeldungen: \$a364–\$a375

Die Texte »OK«, »ERROR« und »IN« werden für mehrere Fehler- und Steuermeldungen benötigt. Deshalb fressen Sie nicht bei jeder einzelnen solchen Meldung Speicher, sondern nur einmal an dieser Stelle im Speicher und werden in jedem Bedarfsfall von hier ausgegeben. Zur Ausgabe verwendet der Interpreter seine STROUT-Routine bei \$a364, welche die \$00-Endmarkierungen der einzelnen Texte verlangt.

Das Prompt »READY.« ist dahinter im Speicher ebenso im STROUT-Format untergebracht wie »BREAK«, der Text zur Fehlermeldung #30. Bei »BREAK« ist dies eine erhebliche Abweichung gegenüber dem Format der anderen Fehlermeldungen, in denen Bit 7 das letzte Byte markiert. Daran erkennt man die exponierte Stellung des BREAK ERROR innerhalb der 30 Fehlermeldungen.

SRCSTK \$a38a: Routine zur Suche von Stapeleinträgen des Basic-Interpreters

In 3.4.11 wurde genau erläutert, wie sich der Basic-Interpreter den Stapel im Zusammenhang mit den Befehlen GOSUB/RETURN und FOR/NEXT zunutze macht.

Dabei gibt es zwei verschiedene Ziele, die die Routine verfolgt.

1. Suche des letzten Stapeleintrags

Wenn im Akku nicht das FOR-Token \$81 steht, wird das Headerbyte des letzten Basic-Stapeleintrags in den Akku geholt und bei gelöschtem Z-Flag zurückgesprungen. Das X-Register enthält dann den Offset auf den ausgelesenen Stapeleintrag.

Wurde SRCSTK (\$a38a) von der RETURN-Routine aufgerufen, so stellt diese dann noch fest, ob der letzte Stapeleintrag von GOSUB stammt (Akku = \$8d) oder nicht (Akku \$8d).

2. Suche eines bestimmten FOR/NEXT-Eintrags

Befindet sich im FOR/NEXT-Variablenzeiger ein anderer Wert als \$0000, so durchsucht SRCSTK (\$a38a) den gesamten Stapel nach einem Basic-Stapeleintrag für FOR/NEXT mit demselben Variablenzeiger. X enthält bei dessen Auffinden den Stapelzeiger-Inhalt und das Zero-Flag wird gesetzt.

Bei »leerem« FOR/NEXT-Variablenzeiger wird lediglich festgestellt, ob sich noch ein FOR/NEXT-Eintrag am Stapel befindet, und dieser wird dann als Suchergebnis im FOR/NEXT-Variablenzeiger vorgelegt.

Programmtechnische Besonderheit

Bei \$a398 verzweigt SRCSTK (\$a38a) für den Fall, daß zwar ein FOR/NEXT-Eintrag am Stapel gefunden werden konnte, aber kein bestimmter FOR/NEXT-Variablenzeiger zu suchen ist. Das Raffinierte an »a398 bne a3a4« ist nun, daß bei \$a3a4 geprüft wird, ob der FOR/NEXT-Variablenzeiger am Stapel mit dem gewünschten FOR/NEXT-Variablenzeiger übereinstimmt.

Wurde jedoch bei \$a398 nicht verzweigt, so ist durch \$a39f garantiert, daß der Vergleich bei \$a3a4 positiv ausfällt:

```
$a39f lda 0103,x
...
$a3a4 cmp 0103,x
```

In dieser Reihenfolge wird immer das Zero-Flag gesetzt. Das Vergleichsergebnis ist sozusagen durch die logische Abfolge der Befehle vorher festgelegt.

Andernfalls jedoch trifft der Vergleich bei \$a3a4 nicht unbedingt zu, so daß zuerst ein Vergleich des gefundenen FOR/NEXT-Variablenzeigers mit dem gewünschten FOR/NEXT-Zeiger stattfindet.

\$a3b8:

Routine zur Bereitstellung von Variablen-Speicherplatz

Wenn der Interpreter ab einer bestimmten Adresse Variablen-Speicherplatz benötigt, übergibt er diese Adresse in Akku und Y-Register und ruft diese Routine auf. Dies geschieht beim Einfügen einer neuen Basic-Zeile und beim Anlegen einer Basic-Variablen.

Dazu muß der Variablenbereich und gegebenenfalls auch ein Teil des Programmbereiches verschoben werden, was die Routine BLTUC, ein Bestandteil dieses Programms, erledigt.

BLTUC \$a3bf: Speicherblockverschiebung

Der BLTUC-Einsprung ist für den Programmierer oft eine willkommene Hilfe, um sich die Entwicklung einer eigenen Speicher-verschiebungsroutine zu ersparen.

Die Nutzung von BLTUC ist denkbar einfach: In drei Zeropage-Zeigern befinden sich die Parameter, gemäß denen bei »jsr bltuc« eine Speicherverschiebung erfolgt. Die Parameter geben an, welcher Speicherblock an welche Stelle zu übertragen ist. Die Anfangsadresse des Originalbereiches hat sich dazu in \$5f/\$60, die dazugehörige Endadresse plus 1 in \$5a/\$5b und die Endadresse des Zielbereiches plus 1 in \$58/\$59 zu befinden. Die Anfangsadresse des Zielbereiches muß nicht angegeben werden, sie ergibt sich zwangsläufig aus der Endadresse des Zielbereiches und der Länge des zu verschiebenden Speicherblockes.

Zu den Endadressen wird deshalb 1 addiert, weil sie die jeweils erste nicht zu kopierende Speicherstelle bezeichnen. Die letzte Adresse im Übertragungsbereich liegt jedoch um 1 Byte darunter.

Kommen wir nun zur Funktionsweise von BLTUC. Zuerst wird die Länge des Kopierbereiches errechnet. Low- und High-Byte dieser Bereichslänge dienen später als Dekrementierzähler in den Verschiebeschleifen. Falls es sich um einen »ganz-seitigen« Bereich handelt (LB der Bereichslänge ist 0), also die Anzahl der zu verschiebenden Bytes ohne Rest durch 256 (\$0100) teilbar ist, wird eine Routine zum Verschieben vollständiger Speicherseiten angesprungen. Andernfalls ermittelt BLTUC die Anfangsadresse von Quell- und Ziel-Restbereich und springt eine Verschiebeschleife speziell für den Restbereich an, in welcher das Low-Byte der Bereichslänge als Dekrementierzähler dient. Nach Ablauf dieser Rest-Verschiebeschleife kommt die bereits erwähnte Kopierschleife für komplette Speicherseiten zur Ausführung, deren Dekrementierzähler das High-Byte der Bereichslänge ist. Danach springt BLTUC an die aufrufende Routine zurück.

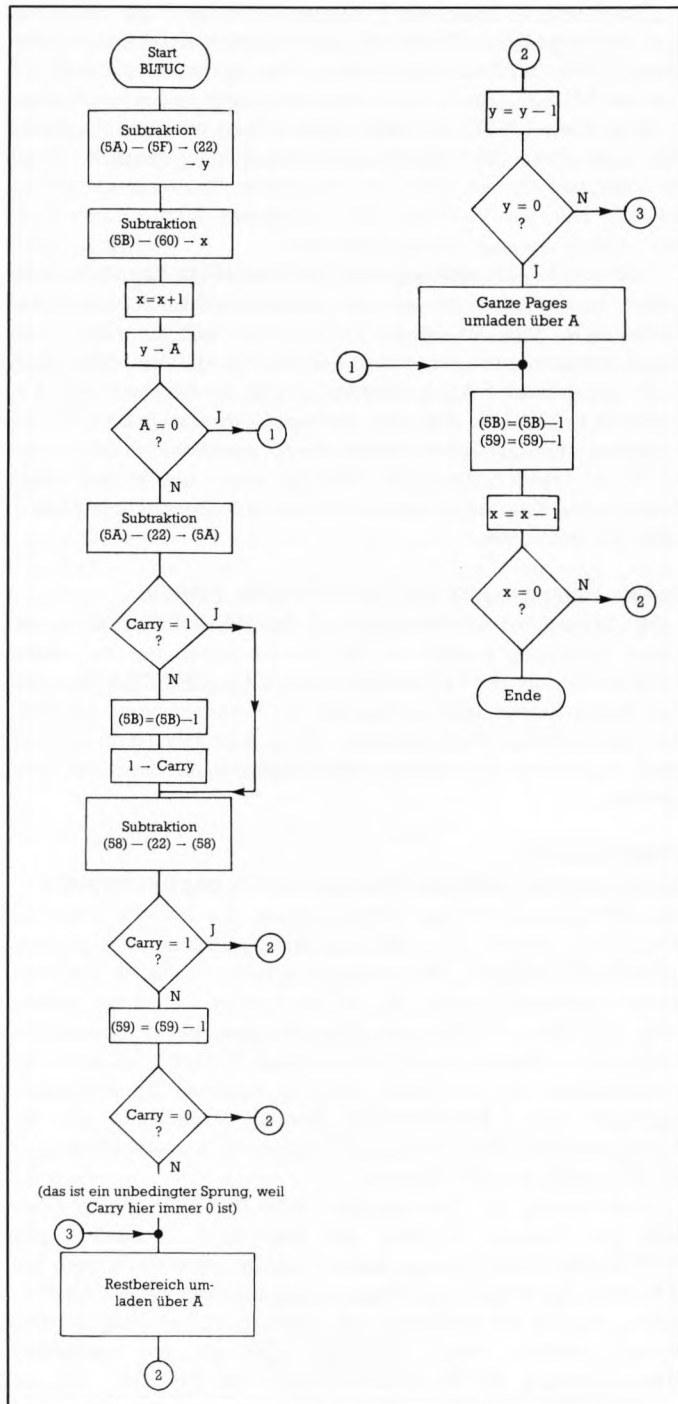
Interessant ist die Verschachtelung der beiden Kopierschleifen. Diese erwarten jeweils folgende Parameter:

Kopierschleife 1 (Rest-Verschiebeschleife)

- Anzahl der zu verschiebenden Bytes im Y-Register; bei Y=\$00 wird eine komplette Page verschoben
- Anfangsadresse des Quellbereiches in \$5a/\$5b
- Anfangsadresse des Zielbereiches in \$58/\$59

Kopierschleife 2 (Ganzseiten-Verschiebeschleife)

- Anzahl der zu verschiebenden Speicherseiten im X-Register
- Endadresse des Quellbereiches in \$5a/\$5b
- Endadresse des Zielbereiches in \$58/\$59



Dabei fällt sofort auf, daß beide Kopierschleifen die Zeiger \$58/\$59 (Zielbereich) und \$5a/\$5b (Quellbereich), jedoch zwei unterschiedliche Dekrementierzähler (X- und Y-Register) einsetzen. Dies ist deshalb von Bedeutung, weil in Kopierschleife 2 auf Kopierschleife 1 zurückgegriffen wird, um die jeweils zu übertragende Speicherseite zu kopieren. Im Y-Register steht dazu \$00, so daß die komplette Page von \$5a/\$5b nach \$58/\$59 verschoben wird. Kopierschleife 2 also bewirkt lediglich, daß in diesen beiden Hilfszeigern jeweils die richtigen Werte stehen, damit Kopierschleife 1 die tatsächliche Speicherübertragung vornimmt.

Abbildung 4.1 ist noch ein Flußdiagramm zur BLTUC-Routine. Bei der Besprechung von LNKPRG (\$a533) finden Sie ein sehr lehrreiches Anwendungsbeispiel für BLTUC (\$a3bf), das jedoch mehr der Demonstration von LNKPRG (\$a533) dient. Ein eigenes BLTUC-Beispiel ist Listing 4.2, das mit Hypra-Ass assembliert und nach einem Reset (SYS 64738) über SYS 49152 gestartet wird. Dann wird das Basic-ROM an das RAM mit gleicher Adresse kopiert und auf RAM-Betrieb umgeschaltet.

← Abbildung 4.1: Flußdiagramm zur Routine BLTUC

READY.

```

100 -.BA $C000 : START: SYS 49152
110 -;
120 -; NICHT VON HYPRA-ASS AUS STARTEN!
130 -;
140 -.GL BLTUC = $A3BF
150 -.GL ANFANGORIGINAL = $A000
160 -.GL ENDEORIGINAL = $BFFF
170 -.GL ENDEZIEL = ENDEORIGINAL
180 -;
190 -.GL MAIN = $A480
200 -;
210 - LDA #(<(ANFANGORIGINAL)
220 - STA $5F
230 - LDA #(>(ANFANGORIGINAL)
240 - STA $60
250 -;
260 - LDA #(<(ENDEORIGINAL+1)
270 - STA $5A
280 - LDA #(>(ENDEORIGINAL+1)
290 - STA $5B
300 -;
310 - LDA #(<(ENDEZIEL+1)
320 - STA $58
330 - LDA #(>(ENDEZIEL+1)
340 - STA $59
350 -;
360 - JSR BLTUC
370 -;
380 - LDA #54 ; BASIC-BEREICH
390 - STA 1 ; AUF RAM STELLEN
400 -;
410 - JMP MAIN ; WARMSTART
  
```

READY.

Listing 4.2: Beispiel zu BLTUC

GETSTK \$a3fb:**Routine zur Prüfung auf genügend freien Stapelplatz**

Wie Sie spätestens seit Kapitel 3.4.11 wissen, arbeitet der Interpreter sehr »stapel-intensiv«, wenn FOR/NEXT und GOSUB/RETURN auszuführen sind. Damit kein Stapelüberlauf auftritt, der einen Prozessorabsturz auslöst, prüfen die entsprechenden Routinen mit Hilfe von GETSTK (\$a3fb), ob der Stapel freien Platz in ausreichender Menge bietet. Dazu wird die Hälfte des erforderlichen Stapelspeichers (als Byte-Anzahl) in den Akku geladen und GETSTK (\$a3fb) aufgerufen. Ist der nötige Speicherplatz vorhanden, so erfolgt ein gewöhnlicher Rücksprung; andernfalls wird ein OUT OF MEMORY ERROR ausgelöst. In 3.4.11 wurde bereits angesprochen, daß der Text »OUT OF MEMORY« nicht unbedingt die beste Fehlerbeschreibung ist (bessere Alternativen wären wohl »OUT OF STACK«, »STACK OVERFLOW« oder einfach »STACK FULL«). Zudem finden Sie in 3.4.11 eine Beispieleingabe, um den Stapelüberlauf hervorzurufen.

Zur Funktionsweise von GETSTK (\$a3fb). Die Hälfte der benötigten Stapelbytes, die im Akku übergeben wurde, wird zuerst verdoppelt. Dann wird noch 62 addiert, da der Interpreter so viele Bytes am Anfang des Stapels als Hilfsspeicher bei Kassettenoperationen benötigt, um darin die aufgetretenen Fehler zu notieren (siehe Kap. 3.4.11). Nach der Addition von 62 steht also die Menge von Stapelplatz, die unter allen Umständen ungenutzt zu bleiben hat, damit der Interpreter arbeiten kann. Dieser Wert wird schließlich mit dem Stapelzeiger verglichen, und solange der Stapelzeiger groß genug ist (woraus die Verfügbarkeit von genügend freiem Stapelspeicherplatz hervorgeht), meldet GETSTK (\$a3fb) auch keine Bedenken in Form eines OUT OF MEMORY ERROR.

Als Hilfsspeicher für das Additionsergebnis (Parameter mal 2 plus 62) dient \$22; als weiterer Nebeneffekt befindet sich nach »jsr getstk« der Inhalt des Stapelzeigers zu diesem Zeitpunkt im X-Register.

Das Y-Register bleibt unverändert, und die Routine GETSTK (\$a3fb) erfordert ihrerseits keinen Stapelplatz.

GETFVM \$a408: Routine zur Prüfung und Bereitstellung von freiem Basic-Speicherplatz

Erinnern Sie sich noch an Kapitel 3.4.3, Abbildung 3.2? Dort steht beschrieben, wie der freie Speicher im Basic-RAM »aufgefressen« wird. »Von unten« drücken Basic-Programm und Basic-Variablen, »von oben« der Stringinhaltspeicher. \$31/\$32 ist die untere Grenze des freien Speicherbereiches und \$33/\$34 das obere Limit.

Wird »von unten« Speicherplatz benötigt, weil Programm oder Variablen zusätzliches RAM erfordern, so ist GETFVM (\$a408) dafür verantwortlich, den benötigten Speicherplatz aufzutreiben, wenn er nicht von sich aus vorhanden ist.

Dazu wird in Akku und Y-Register derjenige Wert übergeben, den der Zeiger \$31/\$32 nach der Nutzung des zusätzlich erforderlichen Basic-RAMs annehmen muß. Dieser Wert ist also der bisherige Inhalt von \$31/\$32, erhöht um die zusätzlich benötigte Speichermenge.

Dann kann GETFVM (\$a408) diesen Wert mit der Obergrenze des (noch) freien Basic-RAM vergleichen. Solange ausreichend viel Speicherplatz frei ist, wird der übergebene Parameter kleiner als \$33/\$34 sein, was zu einem RTS-Rücksprung führt: Grünes Licht zur Nutzung des angeforderten Speichers!

Andernfalls wird eine Garbage-Collection (siehe Kap. 3.4.6) ausgelöst, wozu vorher die von der Garbage-Collection veränderten Hilfsspeicher auf den Stapel gerettet und nachher wieder vom Stapel geholt werden. Ist nach der Garbage-Collection immer noch nicht genug Basic-RAM vorhanden, erfolgt die Meldung OUT OF MEMORY ERROR; war die Garbage-Collection jedoch so erfolgreich, daß genügend Speicherplatz gewonnen wurde, wird GETFVM (\$a408) über RTS verlassen, wobei für die aufrufende Routine kein Unterschied daraus entsteht, daß eine Garbage-Collection erforderlich war.

\$a435: Einsprung für OUT OF MEMORY ERROR

»JMP \$a435« löst die Meldung OUT OF MEMORY ERROR aus. Dieser Einsprung besteht im Grunde genommen nur aus einem LDX-Befehl, der die Fehlernummer von OUT OF MEMORY lädt. Auf diesen Befehl folgt im Speicher der Fehlereinsprung ERROR, der eine beliebige Fehlermeldung, deren Kennzahl im X-Register steht, ausgibt und das laufende Basic-Programm mit dieser Meldung abbricht.

ERROR \$a437:**Einsprung in Fehlerbehandlungsroutine des Interpreters**

Der allgemeinverbindliche Fehlereinsprung des Basic 2.0 liegt an dieser Stelle (\$a437). Dort steht ein Sprung über den Fehlervektor IERROR \$0300/\$0301, der – solange er nicht vorsätzlich verändert wurde – nach \$e38b weist. Die dortige Routine sortiert den Fehlercode \$80 (»kein Fehler«) aus; liegt allerdings ein herkömmlicher Fehler vor, springt sie nach \$a43a. Diese Routine verdoppelt die Fehlernummer, da die Tabelle mit den Adressen der Fehlermeldungen aus 2-Byte-Einträgen besteht. Ferner wird aus der Adreßtabelle \$a328 die Adresse des Fehlertextes in den Hilfszeiger \$22/\$23 geholt und dort gemerkt.

Dann werden die Ein-/Ausgabe-Geräte für Basic gesetzt (Eingabe von Tastatur, Ausgabe auf Bildschirm). Daraufhin gibt ERROR ein [CR] (Carriage Return) und gegebenenfalls noch ein [LF] (Line Feed) sowie ein Fragezeichen aus. Nun kommt der Fehlertext, der von der Fehlernummer abhängig ist, zur Ausgabe. Des weiteren werden einige Basic-Hilfszeiger für den temporären Stringstapel und den Benutzerfunktionsaufruf (FN, DEF FN) in-

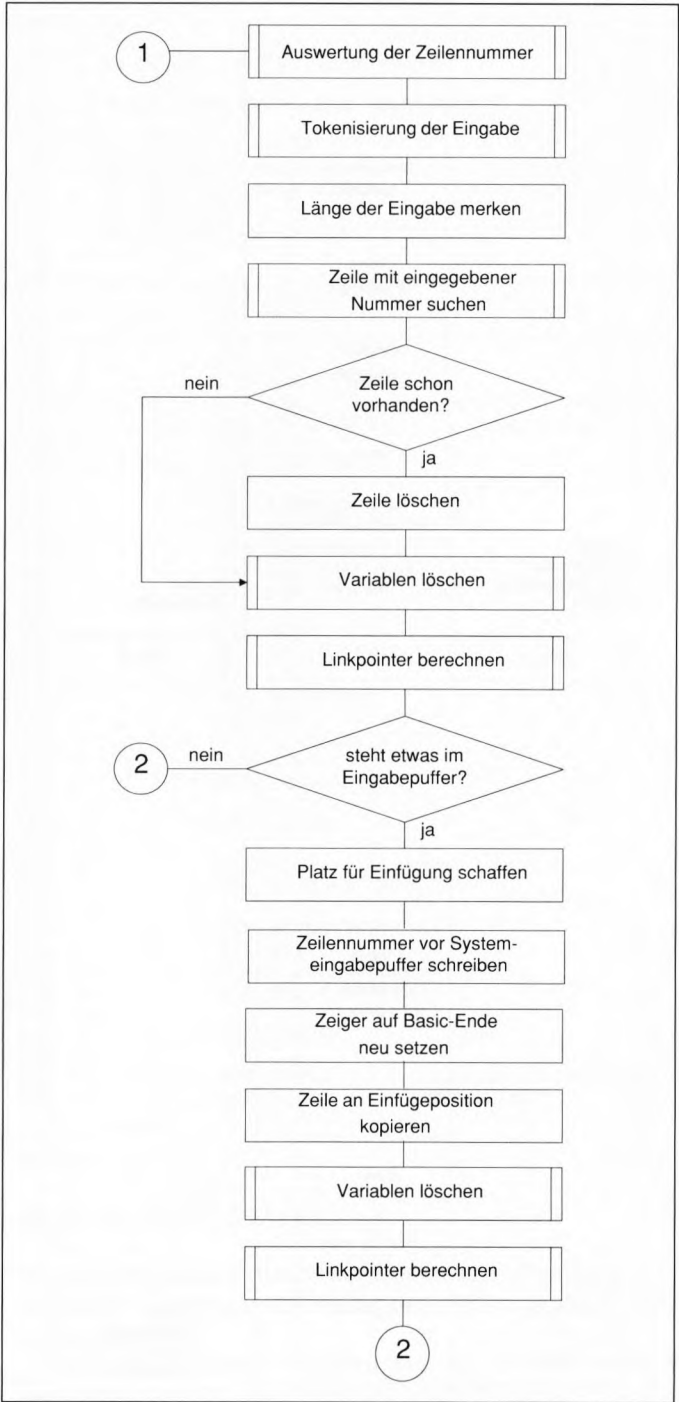
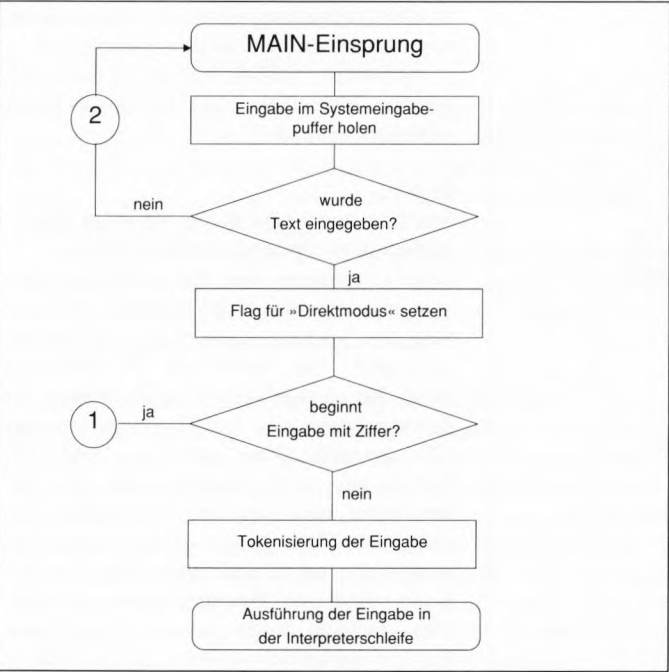
itialisiert sowie der CAN'T-CONTINUE-Zustand hergestellt (nach Fehlermeldungs-Programmabbruch darf CONT nicht ausgeführt werden). Schließlich erfolgt die Ausgabe von »ERROR« sowie gegebenenfalls »IN zeile«. Zu guter Letzt wird das Flag für den Basic-Direktmodus gesetzt und der Warmstart (MAIN \$a480) ausgelöst, dessen Einsprung unmittelbar hinter der ERROR-Routine im Speicher steht.

MAIN \$a480:
Basic-Warmstart (Betreten des Eingabemodus)

Bei MAIN (\$a480) liegt der Sprung über den Warmstart-Vektor IMAIN \$0302/\$0303, der in unverändertem Zustand nach \$a483 weist. Dort wird zuerst über eine Unteroutine eine Basic-Eingabe in den Systemeingabepuffer (ab \$0200) geholt, auf welchen auch der CHRGET-Zeiger gerichtet wird, damit die Auswertung der Eingabe über CHRGET und die auf CHRGET basierenden Parameterauswertungsroutinen (siehe Kap. 3.4.7) möglich ist.

Anhand des ersten Zeichens im Systemeingabepuffer wird dann festgestellt, ob es sich um eine Eingabe zur Direktausführung oder eine Basic-Programmzeile handelt: Basic-Zeilen werden an vorangestellten Zahlen erkannt, das erste Zeichen muß dann folglich eine Ziffer sein, während bei Direktmodus-Eingaben ein anderes Zeichen

Abbildung 4.2: Ablaufplan des Warmstarts →



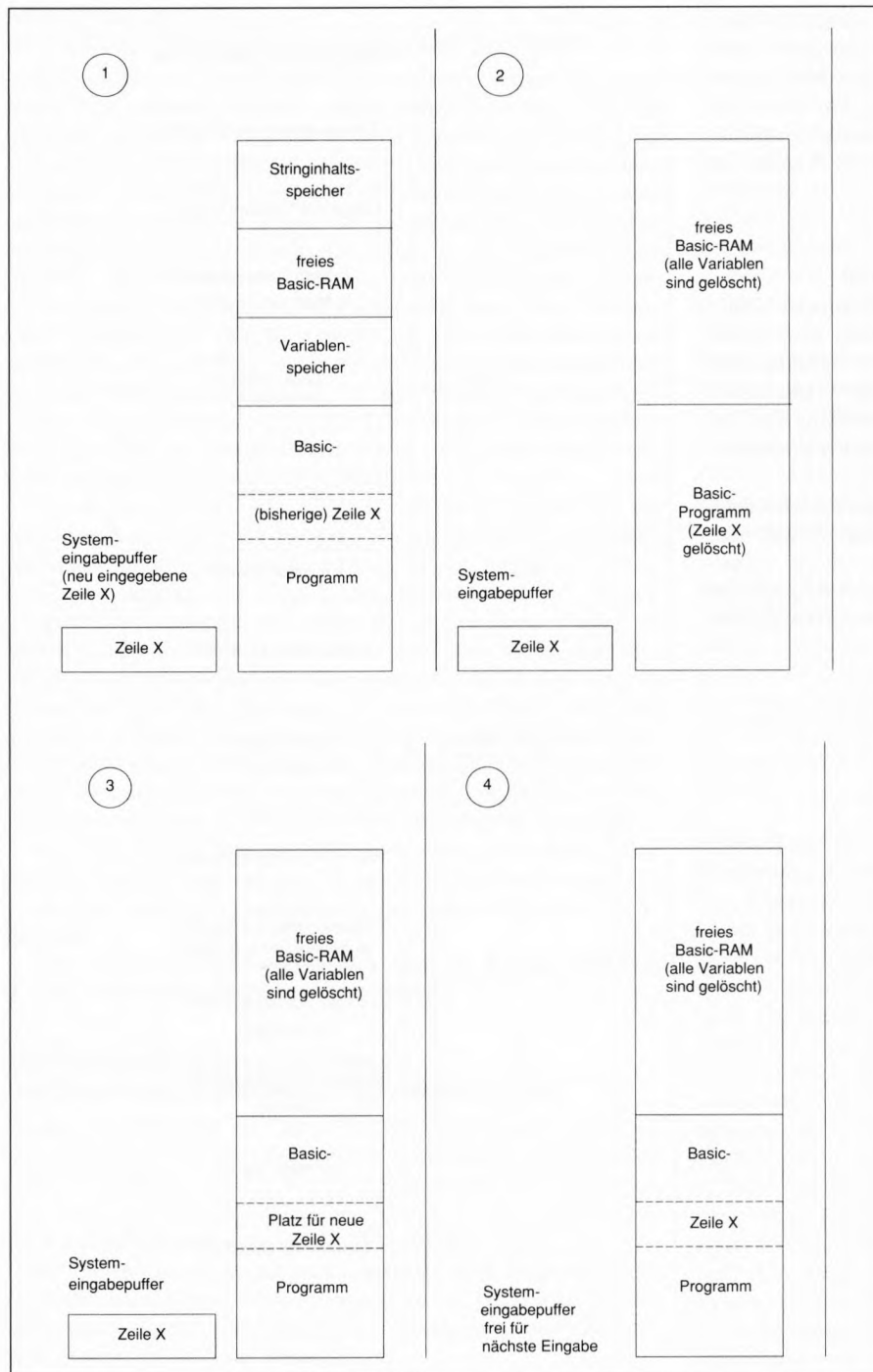


Abbildung 4.3: Die einzelnen Schritte zur Einfügung einer eingegebenen Basic-Zeile:

- (1) Eingabe der Zeile in den Systemeingabepuffer
- (2) Löschen des alten Inhalts der entsprechenden Zeile; entfällt, wenn eine Zeile mit dieser Nummer noch nicht existiert
- (3) Schaffen von Einfügeplatz an der Einfügeposition
- (4) Nun ist die Zeile endgültig ins Basic-Programm kopiert worden

am Anfang steht. Beginnt der Systemeingabepuffer mit der Endmarkierung \$00, so wird die Eingabe wiederholt. Der TAX-Befehl bei \$a48d ist deshalb erforderlich, weil das Z-Flag nach »jsr chrget« auch beim Code \$3a (Doppelpunkt) gesetzt wäre, der jedoch nicht auf einen leeren Eingabepuffer hinweist.

Zurück zur Unterscheidung zwischen Direktanweisung und Zeilenangabe. Bei Eingabe einer Direktmodus-Anweisung wird anhand des gelöschten Carry-Flags (von CHRGET beeinflusst) diese Situation erkannt, die Direktanweisung tokenisiert und an die Interpreterschleife zur Ausführung übergeben.

Andernfalls wird die Routine ab \$a49c aufgerufen (Einbindung einer im Eingabepuffer stehenden Basic-Zeile):

\$a49c:

Routine zur Einbindung einer im Eingabepuffer stehenden Basic-Zeile

Diese Unteroutine des Warmstartprogramms holt als erstes die Zeilennummer vom Anfang des Systemeingabepuffers und tokenisiert alles, was hinter der Zeilennummer steht. Die Länge der Eingabe hinter der Zeilennummer wird im Eingabepufferzeiger \$0b vermerkt. Dann sucht ein FNDLIN-Aufruf eine Zeilennummer mit der eingegebenen Nummer, um festzustellen, ob eine solche Zeile bereits existiert oder noch nicht. Falls schon vorhanden, wird die alte Zeile mit dieser Nummer zuerst gelöscht. Dies geschieht durch einfaches Verschieben des Speicherbereiches »über« der zu löschen-

den Zeile an die Anfangsadresse dieser Zeile, so daß diese überschrieben wird. Danach jedoch müssen noch die Linkpointer angepaßt und die Variablen gelöscht werden, da der Variablenbereich ebenfalls mitverschoben wurde und somit »verloren« ist.

Nach diesem Löschen des alten Zeileninhaltes prüft die Routine, ob sich im Eingabepuffer außer der Zeilennummer noch eine Eingabe befand; falls nicht, so wird erneut der Warmstart ausgelöst, da durch bloßes Eingeben einer Zeilennummer nur das Löschen der Zeile verlangt wird, und dies ist ja zum gegenwärtigen Zeitpunkt schon zu aller Zufriedenheit erfolgt.

Zur Einfügung der eingegebenen Zeile wird der Speicherbereich »über« der Einfügeposition »nach oben« verschoben, um Platz für die einzubindende Zeile zu schaffen. Dafür dient der Einsprung bei \$a3b8, der sich auf GETFVM (\$a408) und BLTUC (\$a3bf) stützt.

Nun ist also an der richtigen Position ein Bereich von ausreichender Länge vorhanden, in welchem die neu eingegebene Zeile Platz findet. Bevor sie dorthin kopiert werden kann, wird sie aber noch im Systemeingabepuffer hergerichtet, indem die Zeilennummer im Low-High-Format vor die tokenisierte Eingabe geschrieben wird. Im üblichen Format steht die Basic-Zeile also jetzt in folgenden Adressen:

\$01fc/\$01fd: Linkpointer (wird nicht mit besonderen Werten belegt, da nach Kopieren der Basic-Zeile ohnehin die Linkpointer-Aktualisierung erfolgt)

\$01fe/\$01ff: Zeilennummer

ab \$0200: tokenisierter Zeileninhalt

Nach Neusetzen des Zeigers auf das Basic-Programmende wird dann endlich die Zeile an die Einfügeposition übertragen, woraufhin die Variablen gelöscht und die Linkpointer berechnet werden. Dann springt der Interpreter an den Beginn der Warmstart-Routine zur Bearbeitung der nächsten Eingabe.

Abbildung 4.2 ist ein sehr schematischer Ablaufplan der Warmstartroutine, der zur Zusammenfassung der bisherigen Erklärungen dient. In Abbildung 4.3 sehen Sie grafisch die einzelnen Schritte zur Einfügung einer eingegebenen Basic-Zeile.

LNKPRG \$a533: Neuberechnung der Linkpointer

Bei einigen Operationen des Interpreters werden die Linkpointer zunächst nicht berücksichtigt, dann aber über »jsr lnkprg« auf den aktuellen Stand gebracht. Dies kann man sich auch zunutze machen, wenn man den Ladevorgang eines Basic-Programms oder des Disketten-Inhaltsverzeichnisses unterbrochen hat und auf Eingabe des LIST-Befehls nur chaotisches Zeichengewirr am Bildschirm erscheint: Nach »SYS 42291:LIST« blicken Sie sicher durch, was Sie gerade in den Speicher geholt haben. Auch ein OLD/RENEW-Trick beruht auf LNKPRG: »POKE 2050,8:SYS 42291« läßt das

READY.

```

100 -.BA $C000 ; START: SYS 49152
110 -;
120 -; NICHT MIT HYPRA-ASS VERWENDEN
130 -;
140 -.GL BLTUC = $A3BF
150 -.GL LNKPRG = $A533
160 -.GL BASICANFANG = $2B
170 -.GL BASICENDE = $2D
180 -.GL NEUBEREICH = $2001
190 -;
200 -;
210 - LDA BASICANFANG
220 - STA $5F
230 - LDA BASICANFANG+1
240 - STA $60
250 - LDA BASICENDE
260 - STA $5A
270 - LDA BASICENDE+1
280 - STA $5B
290 - LDA BASICENDE
300 - SEC
310 - SBC BASICANFANG
320 - STA $58
330 - LDA BASICENDE+1
340 - SBC BASICANFANG+1
350 - STA $59
360 - CLC
370 - LDA $58
380 - ADC #<(NEUBEREICH)
390 - STA $58
400 - STA BASICENDE
410 - LDA $59
420 - ADC #>(NEUBEREICH)
430 - STA $59
440 - STA BASICENDE+1
450 -;
460 - JSR BLTUC
470 -;
480 - LDA #0
490 - STA NEUBEREICH-1
500 -;
510 - LDA #<(NEUBEREICH)
520 - STA BASICANFANG
530 - LDA #>(NEUBEREICH)
540 - STA BASICANFANG+1
550 -;
560 - JSR LNKPRG
570 -;
580 - JSR $A659 ; VARIABLEN LOESCHEN
590 - JMP $A480 ; WARMSTART

```

READY.

Listing 4.3: Beispiel zu LNKPRG

versehentlich gelöschte Basic-Programm zumindest wieder bei LIST erscheinen; ordnungsgemäß wiederhergestellt ist es damit jedoch noch lange nicht!

Die LNKPRG-Routine orientiert sich bei der Berechnung der Linkpointer an den \$00-Markierungen im Programm: Da jedes

»\$00« ein Zeilenende markiert, folgt darauf der Linkpointer der nächsten Zeile. Drei aufeinanderfolgende »\$00« stehen für das Programmende.

Listing 4.3 verschiebt zuerst mit BLTUC (\$a3bf) das Basic-Programm von seiner aktuellen Anfangsadresse nach \$2001, stellt dann die Basic-Zeiger auf die neue Adresse und läßt die Linkpointer schließlich neu berechnen.

Abbildung 4.4 ist ein Flußdiagramm zur LNKPRG-Routine.

GETSYB \$a560: Eingabe in Systemeingabepuffer holen

GETSYB (\$a560) holt eine Eingabe von Tastatur in den Systemeingabepuffer, an dessen Ende GETSYB (\$a560) die Markierung »\$00« setzt.

Die Länge der Eingabe ist von GETSYB (\$a560) auf 88 Zeichen begrenzt, am Bildschirm sind jedoch maximal 80 Zeichen (2 Bild-

Bildschirmzeilen) vom Editor aus möglich. Sollte dennoch die 88-Zeichen-Obergrenze durchbrochen werden, erfolgt die Meldung STRING TOO LONG ERROR.

Abbildung 4.5 verdeutlicht den Ablauf der GETSYB-Routine.

Ein sehr trickreiches Anwendungsbeispiel ist eine zweizeilige Basic-Routine, die den normalen INPUT-Befehl ersetzt, aber zusätzlich Komma, Doppelpunkt und Semikolon (Strichpunkt) bei der Eingabe erlaubt, und außerdem nicht das bisweilen lästige Fragezeichen ausgibt. Alle sonstigen Eigenschaften von INPUT bleiben erhalten:

```
1 SYS 42336:XX$="" :II=512
2 AA=PEEK(II):IF AA THEN
  XX$=XX$+CHR$(AA) :II=II+1:GOTO 2
```

Die Wirkung dieses Zweizeilers ist, daß eine Eingabe in den String XX\$ geholt wird. Des weiteren verwendet die Routine die Variablen II zur Angabe der aktuellen Adresse im Systemeingabepuffer und AA für den aktuellen ASCII-Code der Eingabe.

Erfolgt keine Eingabe, ist XX\$=CHR\$(32); ansonsten enthält XX\$ alle sichtbaren, eingegebenen Zeichen, auch Komma, Doppelpunkt und Semikolon (Strichpunkt).

Eine ähnliche Routine für den C128 in Basic 7.0 habe ich in meinem Buch »Vom C64 zum C128 – Tips & Tricks« (Markt & Technik Verlag, ISBN 3-89090-402-5) auf den Seiten 131 und 132 vorgestellt.

CRUNCH \$a579: Tokenisierung des Systemeingabepuffers

Auch die Tokenisierungsroutine springt über einen Vektor (ICRUNCH \$0304/\$0305), der in unverändertem Zustand nach \$a57c weist. CRUNCH (\$a579) wandelt die im Systemeingabepuffer befindliche Eingabe soweit wie möglich in Tokens um.

Nach der Tokenisierung steht das Ergebnis ab \$0200 im Systemeingabepuffer; eine möglicherweise zuvor bei \$0200 befindliche Zeilennummer wird also überschrieben, da sie zuvor bereits ausgewertet und an anderer Adresse gespeichert wurde.

Die CRUNCH-Routine ist leicht zu verstehen, wenn man sich die Funktionen der einzelnen Prozessorregister innerhalb von CRUNCH (\$a579) vor Augen hält:

- Das X-Register ist der Offset zum jeweiligen nicht-tokenisierten Byte, von \$0200 (Anfang des Systemeingabepuffers) aus gesehen.
- Im Y-Register steht der Offset für die Adresse von \$0200 aus, in der das bei der Tokenisierung ermittelte Byte unterzubringen ist. Wird jedoch geprüft, ob an der aktuellen X-Position ein Schlüsselwort beginnt, so steht in Y jeweils vorübergehend ein Offset innerhalb der Schlüsselwörterterabelle.

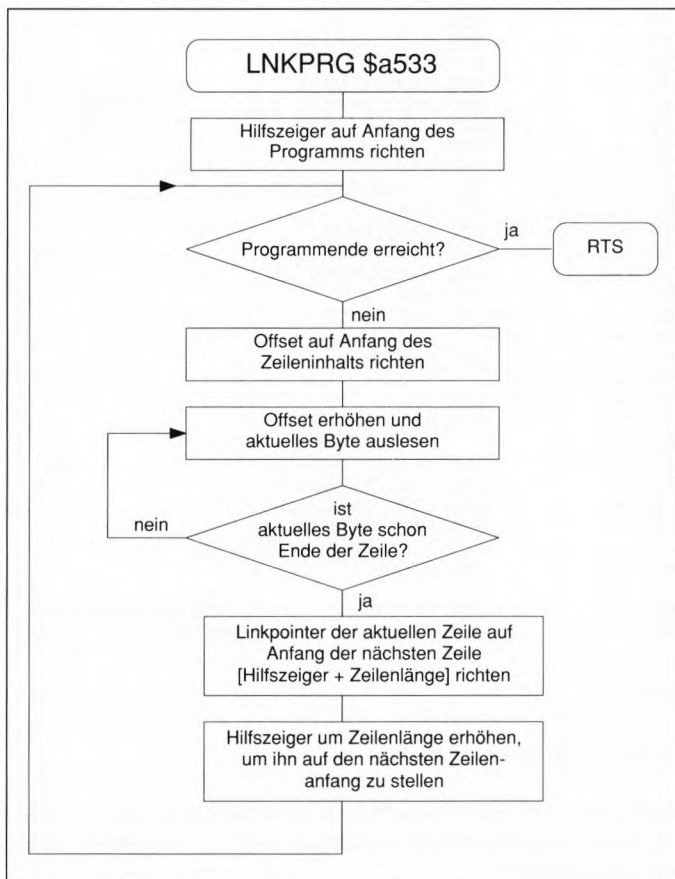


Abbildung 4.4: Ablauf der LNKPRG-Routine

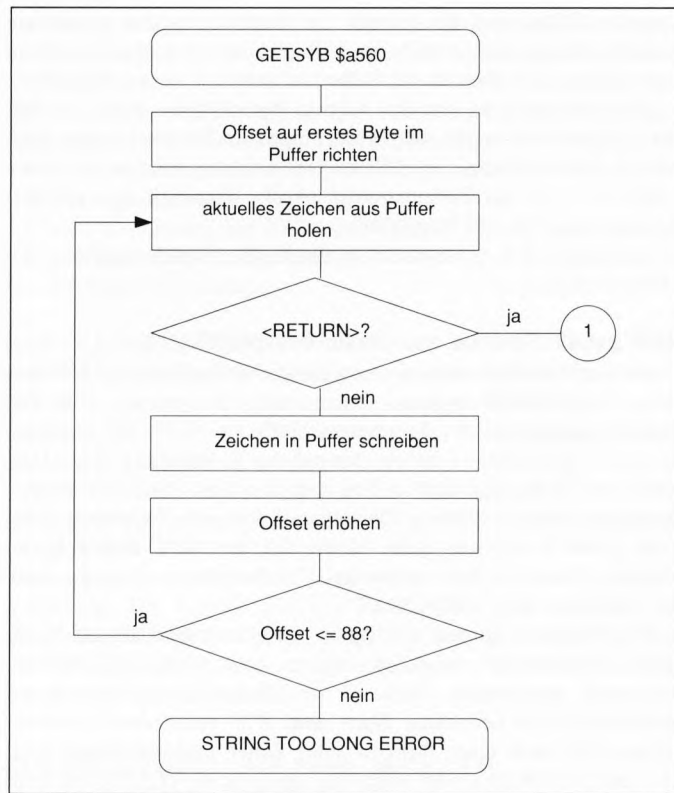


Abbildung 4.5: Ablauf der GETSYB-Routine (Teil 1)

- Der Akkumulator enthält jeweils den aktuellen Code beziehungsweise denjenigen Wert, der aus dem aktuellen Code erzeugt wurde.

Das prinzipielle Vorgehen von CRUNCH (\$a579) läßt sich anhand der Registerbeschreibung schon erkennen. Anhand des X-Offset wird ein nicht-tokenisiertes Byte in den Akku eingelesen, gegebenenfalls umgewandelt und mit Hilfe des Y-Offset in den Systemeingabepuffer als Ergebnis zurückgeschrieben. Da die tokenisierte Eingabe niemals länger als die untokenisierte Form ist, besteht zu keinem Zeitpunkt die Gefahr, daß durch Zurückschreiben des tokenisierten Wertes irgendwelche Informationen im Systemeingabepuffer überschrieben werden, bevor sie tokenisiert sind. Der Y-Offset ist also zu keinem Zeitpunkt größer als der X-Offset.

Der Ausgangswert des X-Offsets ist übrigens der Offset zum ersten Byte hinter der Zeilennummer. Dieser Wert wird durch Auslesen des CHRGET-Zeigers ermittelt, der ja jeweils auf das nächste

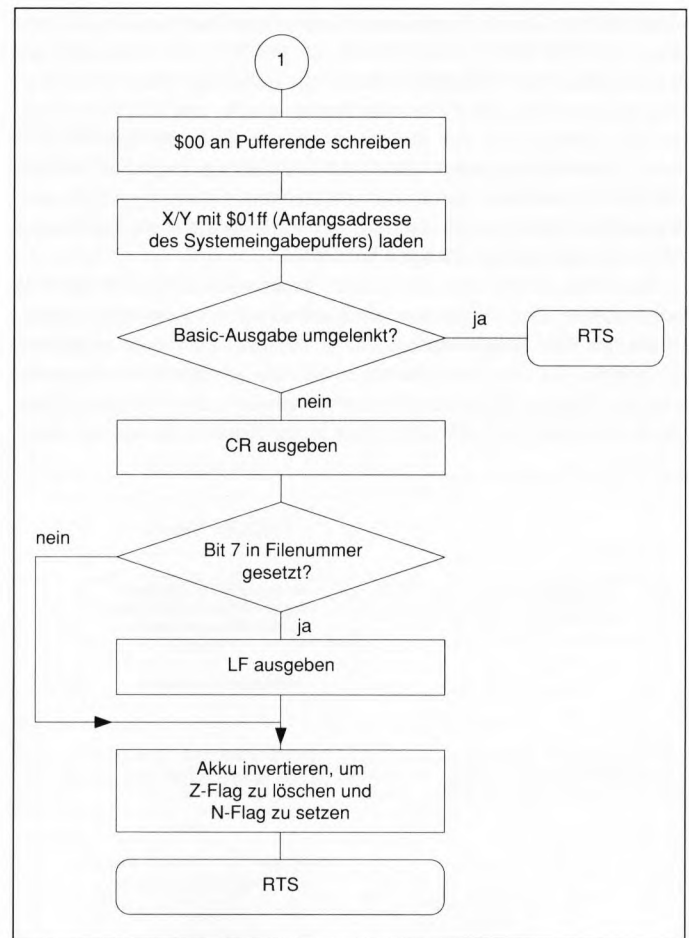


Abbildung 4.5: Ablauf der GETSYB-Routine (Teil 2)

Byte weist. Zur Erinnerung: Die Zeilennummer wurde über eine Routine eingelesen, die sich auf CHRGET (\$0073) stützt.

FNDLIN \$a613:

Adresse einer Zeile im Basic-Programmspeicher ermitteln

Kapitel 3 bespricht unter anderem den Aufbau des Basic-Programms im Speicher.

Wir haben dort festgestellt, daß es sich im Prinzip um eine sequentielle Datenstruktur handelt: In Reihenfolge der Zeilennummern folgt eine Basic-Zeile im Speicher unmittelbar auf die vorhergehende. Bei der sukzessiven Abarbeitung des Basic-Programms ist diese Struktur optimal, doch bei gezielten Zugriffen auf einzelne

»Datensätze« (sprich: Programmzeilen) ist eine Suchroutine erforderlich. An FNDLIN (§a613) wird in \$14/\$15 die Nummer der gewünschten Zeile übergeben. Nach »jsr fndlin« ist dann das Carry-Flag gesetzt, wenn die Zeile aufgefunden wurde, und \$5f/\$60 enthalten die Adresse, bei der die entsprechende Zeile im Speicher beginnt (Adresse des Linkpointers); ist das Carry gelöscht, so steht in \$5f/\$60 die Adresse, bei der die entsprechende Zeile eingefügt werden müßte, nämlich die Adresse der Zeile mit der nächsthöheren Zeilennummer als der erfolglos gesuchten.

Die FNDLIN-Routine ist in ihrer Struktur mit LNKPTR (§a533) vergleichbar: Das Programm wird anhand der Linkpointer durchforstet, bis eine Zeilennummer, die größer oder gleich der gesuchten ist, auftritt. Ist die Zeilennummer mit der gewünschten identisch, wird die Routine bei gesetztem Carry verlassen; der Übergabezeiger für die Zeilenadresse (§5f/\$60) dient in der Routine als laufend aktu-

alisierter Hilfszeiger, der jeweils die Basisadresse der gerade zu durchsuchenden Zeile enthält. Im Y-Register steht immer der Offset zum untersuchten Byte (0 = LB des Linkpointers wegen Aktualisierung des Hilfszeigers mit der Adresse der nächsten Zeile; 1 = HB des Linkpointers wegen möglicher Programm-Endmarkierung oder wegen Aktualisierung des Hilfszeigers mit der Adresse der Folgezeile; 2 = LB der Zeilennummer zwecks Vergleich; 3 = HB der Zeilennummer zwecks Vergleich).

Abbildung 4.6 verdeutlicht grafisch die Funktionsweise von FNDLIN (§a613).

NEW §a642: Routine zum Basic-Befehl NEW

Etwas ungewöhnlich mag es erscheinen, daß die Routine NEW mit einem BNE-Befehl beginnt. Dazu muß man wissen, daß die Befehlsroutinen von der Interpreterschleife das CHRGET-Ergebnis des ersten Bytewertes erhalten, der auf das Befehlstoken folgt. Das somit über BNE abgefragte Z-Flag sagt also aus, ob auf den NEW-Befehl eine Endmarkierung (Doppelpunkt \$3a oder Zeilenende \$00) folgt (dann Z=1) oder nicht (Z=0). Da der NEW-Befehl keine weiteren Parameter hat, verzweigt BNE bei einem Syntaxverstoß wie »NEW 1« oder »NEW TEXT«.

Gleichfalls erklärungsbedürftig ist, warum dieser BNE-Befehl zu einem RTS-Befehl verzweigt, anstatt den SYNTAX-ERROR-Einsprung anzusteuern. Dazu ist die Funktionsweise der Interpreterschleife zu bedenken: Nach dem RTS sucht diese nämlich automatisch nach einer Befehls- oder Zeilen-Endmarkierung, und »1« (aus »NEW 1«) oder »TEXT« (aus »NEW TEXT«) ergeben dann einen SYNTAX ERROR.

Unmittelbar auf den BNE-Befehl folgt schließlich die tatsächliche Routine. Wenn diese also von einer eigenen Maschinenroutine aus aufgerufen werden soll, ist bei §a644 einzuspringen:

NEWIN §a644:

Einsprung für die NEW-Routine für eigene Programme

Dabei ist noch zu beachten, daß innerhalb von NEW, wie wir gleich noch sehen werden, der Stapelzeiger manipuliert wird. Deshalb sollte NEWIN (§a644) nur über JSR, niemals aber über JMP angesprochen werden. Dies gilt auch für alle weiteren Einsprünge in die NEW/CLR-Routine; ein Beispiel gibt Listing 4.3 für den NEWCLR-Einsprung bei §a659.

Nun aber zu den tatsächlich von NEW bewirkten Operationen. Zunächst werden zwei Nullbytes an die Position des ersten Linkpointers im Programm, also in die beiden ersten Adressen des Basic-Programmspeichers, geschrieben. Dadurch wird der Programmfang gleichzeitig zum Programmende deklariert, das Programm ist also insofern »gelöscht«. Damit dieser grobe Eingriff in den Programmspeicher jedoch keine Fehlfunktionen auslöst, sind noch einige Initialisierungsarbeiten für andere Hilfsspeicher des Inter-

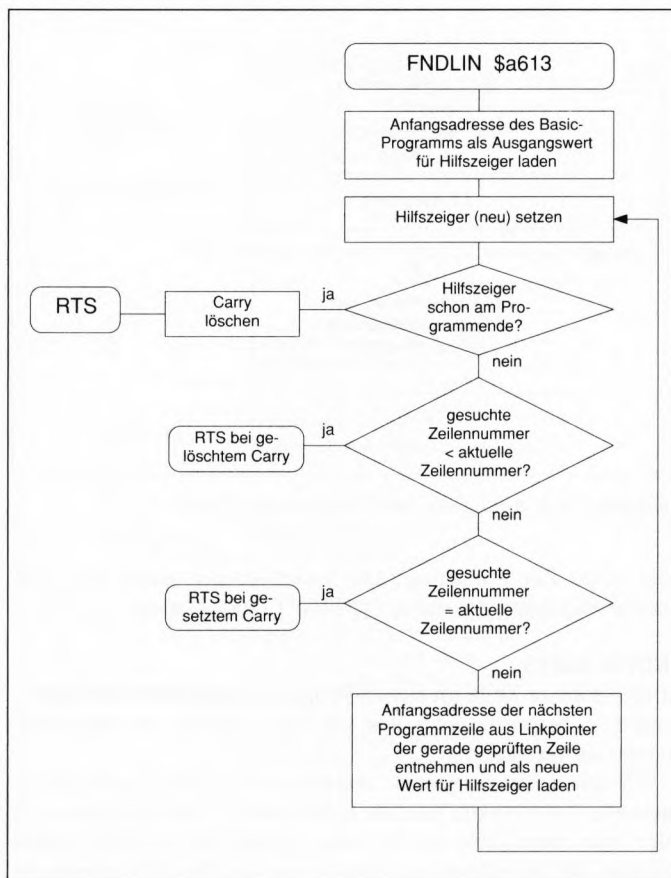


Abbildung 4.6: Ablauf von FNDLIN

preters zu erledigen. Als erstes wird der Zeiger auf das Basic-Programmende mit der Basic-Programm-Anfangsadresse plus 2 (2 Nullbytes sind auch zwei Programmbytes!) geladen. Damit ist gleichzeitig ein neuer Beginn des Variablenspeichers festgelegt. Auch dies ist ein sehr unkoordinierter Eingriff in den Speicher, der nur deshalb funktioniert, weil in der Folge auch der Basic-Befehl CLR durchgeführt wird.

Zuvor noch initialisiert NEW den CHRGET-Zeiger, indem es ihn auf den Basic-Programmanfang richtet. Dies geschieht bei \$a659, dem NEWCLR-Einsprung:

NEWCLR \$a659: CHRGET- und Variablen-Zeiger initialisieren und Variablen löschen

In Listing 4.3 wurde dieser Einsprung beispielshalber eingesetzt, um nach dem Verschieben des Basic-Programms die Variablenzeiger an den neuen Speicherbereich anzupassen.

In NEWCLR (\$a659) wird noch das Z-Flag durch »LDA #00« gesetzt, um vorzutäuschen, daß über CHRGET eine Endmarkierung (Doppelpunkt \$3a oder Zeilenende \$00) eingeholt wurde, denn danach folgt unmittelbar die CLR-Routine im Speicher. Der NEW-spezifische Teil besteht also nur aus \$a642–\$a65d, der Rest ist gleichzeitig die CLR-Routine, deren Beschreibung hier folgt. Die CLR-Beschreibung schließt übrigens mit einem Flußdiagramm der gesamten NEW/CLR-Routine.

CLR \$a65e: Routine zum Basic-Befehl CLR, gleichzeitig Bestandteil der Routine zu NEW

Auch CLR beginnt mit einem »bne -> rts«-Prüfvorgang, um sicherzustellen, daß kein Parameter auf CLR folgt; eine detaillierte Beschreibung dieses Prüfmechanismus finden Sie bei NEW (\$a642).

Die erste von CLR ausgeübte Tätigkeit besteht darin, die Filetabelle des Betriebssystems zu löschen. Daraufhin wird das gesamte Basic-RAM bis zur festgelegten Obergrenze freigegeben, das heißt, der Stringinhaltsspeicher beginnt jetzt an der Speicherobergrenze und ist somit gelöscht. Die Bereiche für einfache Variablen und Arrays werden dadurch gelöscht, daß ihre Anfangs- und Endadressen pauschal mit dem Ende des Basic-Programms festgesetzt werden. Die Variablenbereiche sind somit auf eine Länge von 0 Byte reduziert, was einem Löschen gleichkommt – wie es auch dem Stringinhaltsspeicher widerfahren ist.

Des weiteren ruft CLR die Routine zum RESTORE-Befehl auf, um den READ-DATA-Zeiger auf den Ausgangswert zu setzen. Auch der Zeiger auf den temporären Stringstapel wird so initialisiert, daß wieder der gesamte Stringstapel freigegeben ist.

Der Rest der CLR-Routine holt zunächst die Rücksprungadresse vom Stapel, initialisiert dann den Stapel, legt daraufhin die Rücksprungadresse wieder auf den Stapel zurück und initialisiert vor

dem RTS-Rücksprung noch schnell zwei Flags: CONT und FN werden gesperrt (CAN'T CONTINUE; UNDEF'D FUNCTION).

Abbildung 4.7 faßt den Ablauf von NEW/CLR zusammen.

STXTPT \$a68e: CHRGET-Zeiger auf Basic-Programmanfang stellen

Dieses Unterprogramm (von NEW und LOAD) zieht von der Basic-Anfangsadresse 1 ab und setzt das Ergebnis in den CHRGET-Zeiger \$007a/\$007b ein. Damit ist gewährleistet, daß die Interpreterschleife am Programmanfang beginnt.

Interessant ist, wie die Subtraktion von 1 realisiert wurde: Zum Low-Byte wird \$ff addiert, was einer Subtraktion von 1 gleichkommt (ein mathematischer Beweis erübrigt sich).

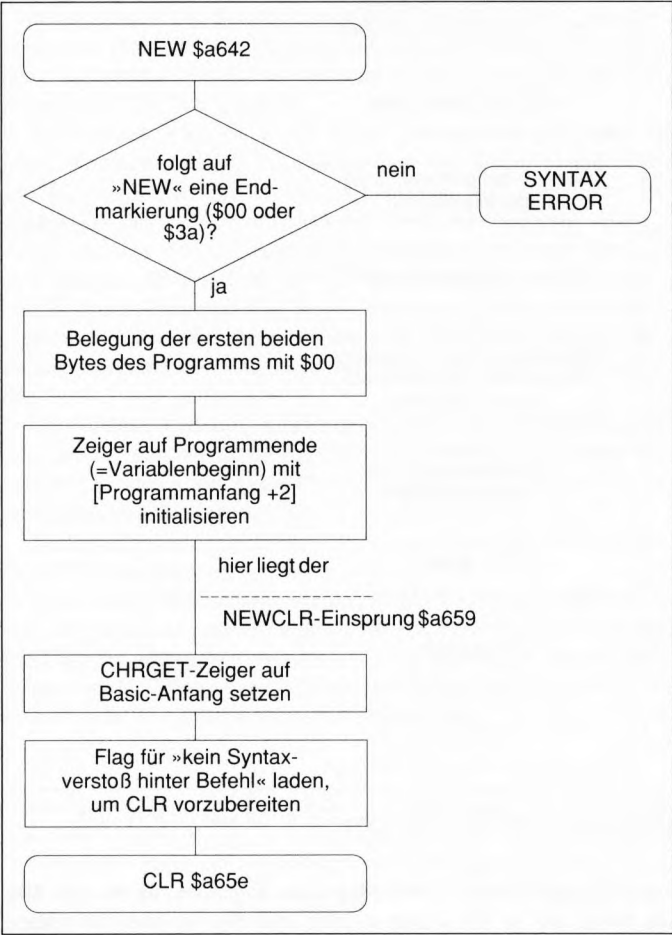


Abbildung 4.7: Ablauf der Routinen zu NEW und CLR (Teil 1)

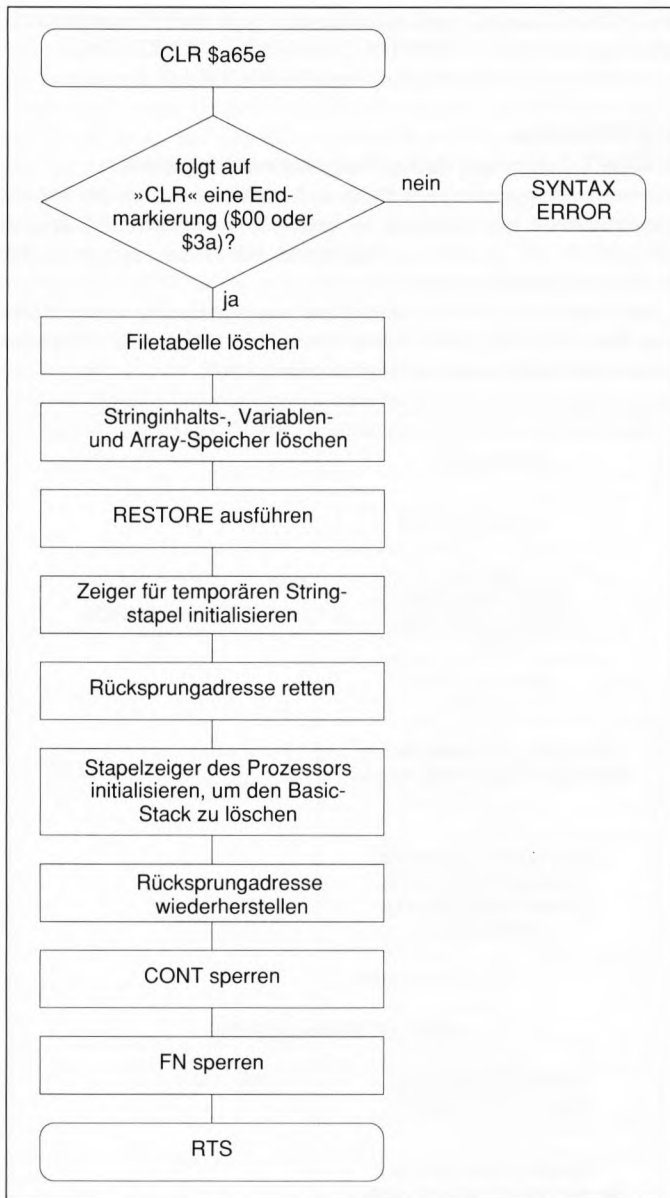


Abbildung 4.7: Ablauf der Routinen zu NEW und CLR (Teil 2)

Normalerweise ist das Carry-Flag danach gesetzt, da bis auf \$00 alle Werte, die zu \$ff addiert werden, eine Summe über \$ff bilden. In diesen Fällen wird bei \$a697 der Wert \$ff+1 (1 für das gesetzte Carry-Flag) addiert, also \$00. War das Low-Byte jedoch \$00, so ist

C=0 und durch Addition von \$ff+0 (0 für das gelöschte Carry-Flag) bei \$a697 wird letztlich auch das High-Byte um 1 verringert.

LIST \$a69c: Routine zum Basic-Befehl LIST

Die LIST-Routine läßt sich in folgende drei Routinenabschnitte unterteilen:

1. Parameterauswertung: \$a69c-\$a6c8

Nach diesem Teil steht in \$5f/\$60 die Adresse der ersten Zeile, die gelistet werden soll, und in \$14/\$15 die letzte Zeilennummer, bei der der LIST-Vorgang noch ohne Abbruch durchzuführen ist. Damit ist die normale Syntax »LIST zeile1–zeile2« vorgesehen.

Für die Sonderfälle »LIST zeile–«, »LIST –zeile«, »LIST zeile« und »LIST 0« entstehen Extremwerte in den Zeigern:

- »LIST zeile–«: \$14/\$15 enthalten \$ffff (höchste 2-Byte-Zahl)
- »LIST –zeile«: \$5f/\$60 enthalten zunächst die Adresse des Programmanfangs
- »LIST zeile«: \$14/\$15 und \$5f/\$60 beschreiben jeweils die einzelne Zeile
- »LIST 0«: \$14/\$15 enthalten \$ffff (höchste 2-Byte-Zahl), \$5f/\$60 enthalten zunächst die Adresse des Programmanfangs

2. Schleifengesteuerte Ausgabe der Zeilen: \$a6c9–\$a716

Das Listen der entsprechenden Zeilen erfolgt in einer Schleife, die sich Byte für Byte durch den Basic-Text hindurchwühlt und nur bei folgenden Bedingungen verlassen wird:

- Programmende (\$00 als Linkpointer) erreicht
- Nummer der aktuell zu listenden Zeile überschreitet die in \$14/\$15 angegebene Endzeilennummer

Zunächst gibt LIST für das Listen einer Zeile ein Carriage Return und gegebenenfalls auch einen Line Feed (in Abhängigkeit vom aktuellen Ausgabegerät) sowie die Zeilennummer und ein darauffolgendes Leerzeichen aus.

Alle weiteren Zeichen werden byteweise ausgelesen und gedruckt, Tokens jedoch in den Klartext der Basic-Schlüsselwörter umgewandelt (solange sie nicht innerhalb von Anführungszeichen stehen). Die Ausgabe eines im Akku befindlichen Bytewertes erledigt QPLOP, der dritte und letzte Bestandteil der gesamten LIST-Routine.

3. QPLOP \$a717:

Ausgabe eines einzelnen Bytewertes mit Ent-Tokenisierung

Alle Werte außer \$ff (PI-Symbol) werden als Tokens betrachtet, wenn sie größer als \$7f sind. Dann sucht QPLOP (\$a717) in der Schlüsselwörtertabelle, die bei der CRUNCH-Tokenisierung herangezogen wurde, nach dem dazugehörigen Klartext und gibt ihn aus.

Bei \$a717 wird übrigens über den Vektor IQPLOP gesprungen, der in unverändertem Zustand nach \$a71a deutet. Zum Verständnis von QPLOP (\$a717) muß man auch bedenken, daß kein RTS-Rücksprung erfolgt, sondern nach Ausgabe des Zeichens unmittelbar in die LIST-Schleife zurückgesprungen wird.

Das Verblüffendste an QPLOP (\$a717) ist aber sicherlich der kleine Programmierfehler, der den <SHIFT L>-Trick zuläßt. Wie auch in diesem Buch schon erwähnt wurde, ist folgende Programmzeile insofern gegen LIST geschützt, als beim Listen der Zeile anstelle des geschifteten »L« ein SYNTAX ERROR auftritt, der einen Abbruch des LIST-Vorgangs zur Folge hat:

```
10 REM <SHIFT L>
```

Dieser Listschutz ist einer der einfachsten und hat entsprechend oft Verwendung gefunden, vor allem bei Einsteigern. Um ein auf diese Weise geschütztes Programm listen zu können, genügt folgender Einzeiler:

```
POKE 95,0:POKE 96,160:POKE 90,0:POKE 91,192:POKE
88,0:POKE 89,192:SYS 41919:POKE 1,54:POKE
42816,144
```

Die Wirkungsweise ist schnell erklärt: Mit Hilfe von BLTUC (\$a3bf = #41919) wird der Basic-Interpreter ins RAM kopiert; der letzte POKE-Befehl schließlich behebt einen kleinen Programmierfehler, auf den wir nun eingehen wollen.

Bekanntlich enthält die Tabelle ab \$a09e alle Schlüsselwörter, wobei im jeweils letzten Byte das Bit 7 als Endmarkierung gesetzt ist. Ein Token ist nun letzten Endes die Positionsangabe innerhalb dieser Tabelle: Zieht man von einem Befehlstoken den Wert \$7f ab, erhält man die Position, ab welcher in der Tabelle ab \$a09e das jeweilige Schlüsselwort enthalten ist. Diese Tatsache macht sich QPLOP (\$a717) zunutze, um den Klartext zu einem Token zu ermitteln: Vom auszugebenden Token wird \$7f subtrahiert (siehe \$a725); das Ergebnis ist die Anzahl von Bit-7-Endmarkierungen, die vor dem gesuchten Klartext stehen, und wird als Dekrementierzähler verwendet. QPLOP (\$a717) sucht also solange nach gesetzten Bit-7-Markierungen, bis der Dekrementierzähler beim Wert 0 angelangt ist. Alle danach folgenden Zeichen aus der Schlüsselwörtertabelle werden nun ausgegeben (\$a73d). Bis dahin handelt es sich um einen fehlerfreien Algorithmus, an dem lediglich zu kritisieren wäre, daß er unerlaubte Tokens (Werte >\$cb) nicht rechtzeitig »herausfiltert«. Dann aber kommt die entscheidende Unzulänglichkeit.

Bei \$a740 soll der BNE-Befehl als Pseudo-JMP dienen, da nach \$a73d die Prozessorflags gemäß dem auszugebenden Byte gesetzt sind, und dieses ist ja keinesfalls \$00 ... oder doch?

Bei allen Werten bis auf \$cc (<SHIFT L>) funktioniert dies auch, und der BNE-Befehl verzweigt wie ein Pseudo-JMP. Doch wurde der Bytewert \$cc an QPLOP (\$a717) übergeben, so besteht

eine Ausnahme. Nach der Umwandlung des Tokens in den Offset für den Klartext (vom Beginn der Befehlswörtertabelle bei \$a09e aus gesehen) enthält Y den Wert \$ff (mit einem TRACE-Befehl Ihres Maschinensprachemonitors können Sie dies nachvollziehen, indem Sie in den Akku den Wert \$cc schreiben, von \$a717 bis \$a738 tracen lassen und dabei den Wert des Y-Registers beobachten), wodurch der Inhalt von \$a19d (\$a09e+\$ff) ausgelesen wird. Und dort steht ein unheilbringendes Nullbyte als Endmarkierung der Schlüsselwörtertabelle!

Die Folgen ergeben sich ganz logisch: Bei \$a740 ist das Z-Flag gesetzt, da der Akku den Wert 0 hatte, und es erfolgt keine Verzweigung, obwohl dies zur fehlerfreien Funktion unbedingt notwendig wäre. Statt dessen wird die im Speicher direkt hinter der LIST-Routine liegende FOR-Routine abgearbeitet, die ihrerseits die Parameter des FOR-Befehls einzuholen versucht (siehe Beschreibung von FOR \$a742). Da wir aber nach LIST keine FOR-Parameter eingeben, vermißt der Interpreter augenscheinlich erforderter Parameter: SYNTAX ERROR!

Nun wissen wir, wie es zu dieser Fehlermeldung kommt, die einen Abbruch des LIST-Vorgangs bewirkt. Beim Erstellen des ROM-Listings (Kapitel 1) fand ich heraus, wie man diese Ungenauigkeit der LIST-Routine behebt. Wenn das Interpreter-ROM in das an gleicher Adresse liegende RAM kopiert wird, kann man den »schuldigen« BNE-Befehl bei \$a740 in einen BCC umwandeln. Dieser Verzweigungsbefehl funktioniert einwandfrei als Pseudo-JMP, da bei der Bildschirmausgabe (»\$a73d jsr \$ab47«) kein I/O-Fehler vorkommt und von daher das Carry-Flag an der entsprechenden Stelle immer gelöscht ist.

Genau diese Änderung nimmt der eingangs vorgestellte Einzeiler vor, der im übrigen durch <RUN/STOP RESTORE> wieder aufgehoben werden kann. Dadurch wird nämlich das alte (und fehlerhafte) Basic-ROM eingeschaltet.

Bei geänderten C64-ROMs (z.B. Floppy-Speeder »64'er-DOS«) ist der Fehler jedoch oft schon behoben.

Eine weitere Möglichkeit, um den <SHIFT L>-Schutz auszutricksen, beruht darauf, daß beliebige FOR-NEXT-Parameter nach dem LIST-Befehl angegeben werden. Das ist allerdings nur möglich, wenn vor den Parametern der Bindestrich aus der Syntax »LIST zeile-« steht. Die Befehlsfolge sieht dann so aus:

```
LIST 10- A=1 TO 1
```

Nach 12maligem LISTen erscheint allerdings ein OUT OF MEMORY ERROR, da zu viele FOR-NEXT-Schleifen ineinander verschachtelt und nicht geschlossen wurden (Stapelüberlauf). Hängt man also noch ein NEXT an die genannte Befehlsfolge, so kann selbst dies vermieden werden.

Abbildung 4.8 beschreibt grafisch die drei Bestandteile der LIST-Routine.

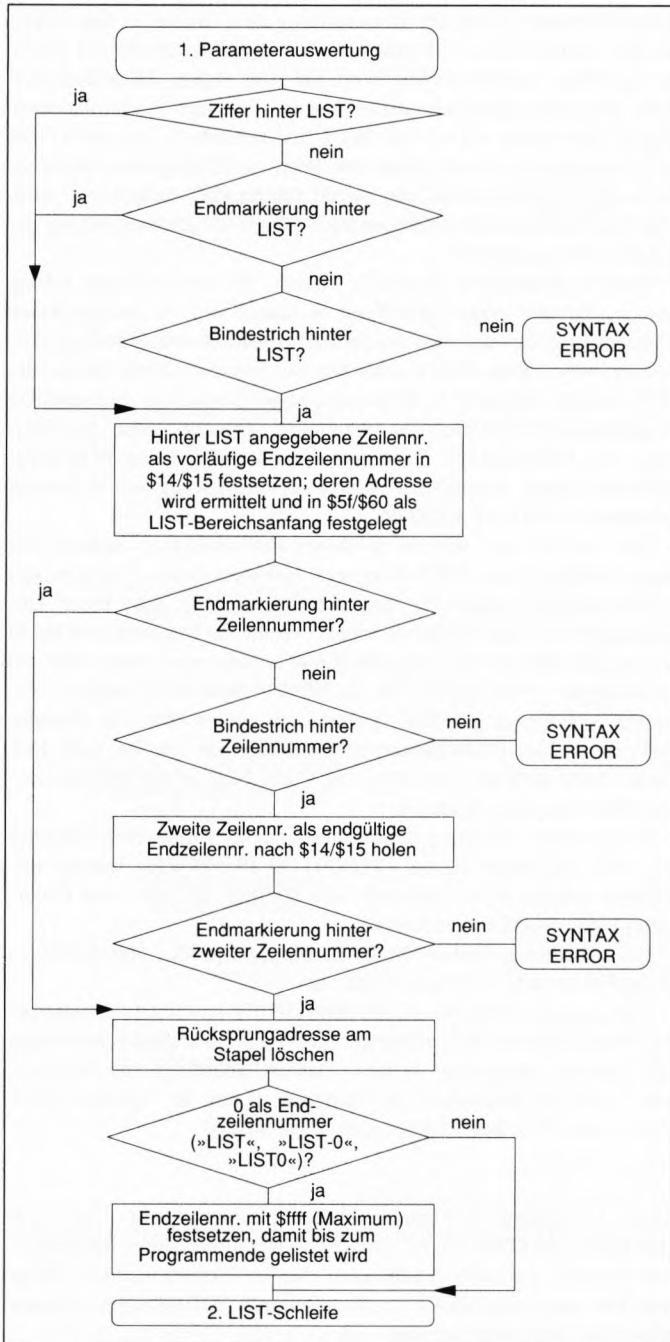


Abbildung 4.8: Die Aufgabenverteilung in der LIST-Routine (Teil 1)

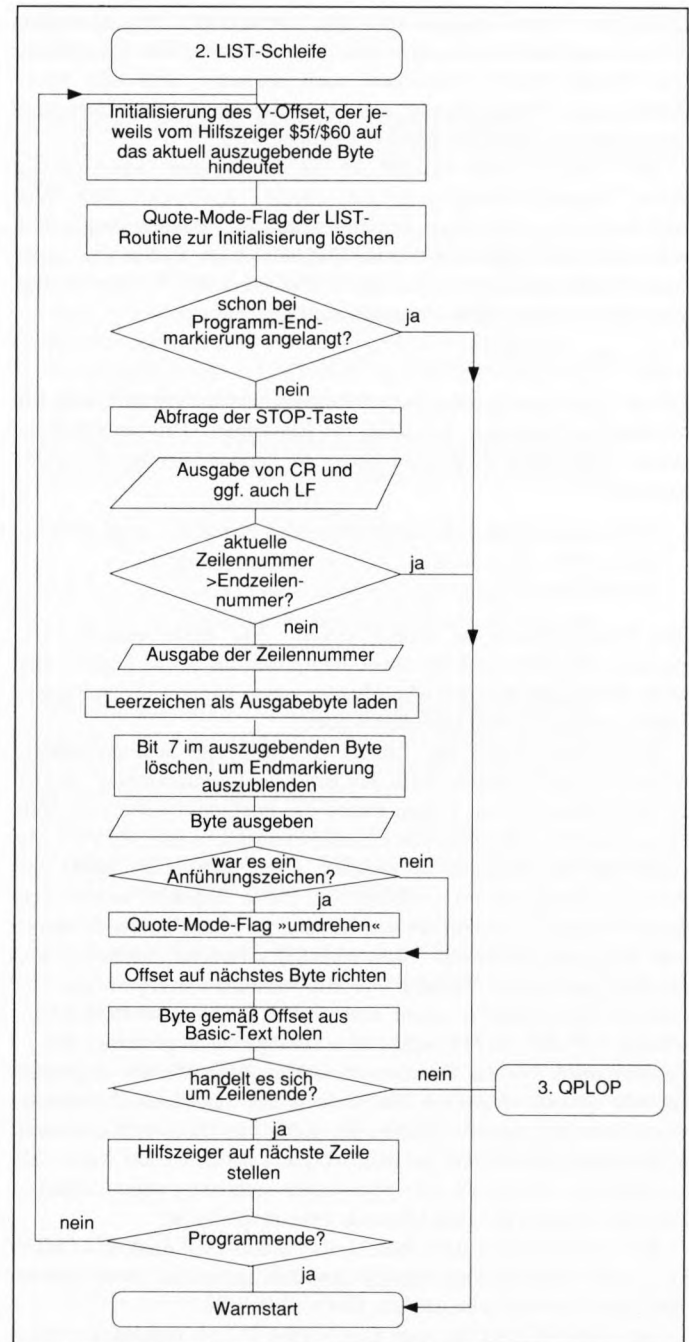


Abbildung 4.8: Die Aufgabenverteilung in der LIST-Routine (Teil 2)

FOR \$a742: Routine zum Basic-Befehl FOR

Aus 3.4.11 wissen wir, wie ein FOR-NEXT-Stapeleintrag des Basic-Interpreters aufgebaut ist und wie er entsteht: Die FOR-Routine legt ihn an. Die eigentliche Schleifenkontrolle läuft dann beim jeweiligen NEXT-Befehl ab, der den zugehörigen Stapeleintrag ausliest und auswertet.

Die Aufgabe der FOR-Routine besteht in der Auswertung der FOR-Parameter, die eine nicht gerade einfache Syntax haben, und deren Ablage auf dem Stapel in richtiger Reihenfolge. Nach dieser Tätigkeit springt FOR in die Interpreterschleife zurück, die unmittelbar hinter FOR (\$a742) im Speicher steht.

Abbildung 4.9 beschreibt die FOR-Routine. Zusätzlich sei aber noch einmal an dieser Stelle die FOR-Syntax aufgeführt:

```
FOR variable=anfang TO ende
[STEP schrittweite]
```

Die eckigen Klammern um »STEP schrittweite« bedeuten, daß dieser Parameter auch entfallen kann (dann wird ersatzweise »STEP 1« angenommen).

Bei genauerer Überlegung ist nun erkennbar, daß die Schleifenvariable »variable« und deren Ausgangswert »anfang« sehr einfach ausgewertet werden können: Die FOR-Routine läßt einfach den LET-Befehl ausführen, so als ob »LET« statt »FOR« dastünde. Nach »LET variable=anfang« ist sogar schon die Schleifenvariable auf den Ausgangswert eingestellt – sehr praktisch! Die LET-Routine wertet übrigens nur die Texte bis zum TO-Token, welches als Abgrenzung erkannt wird. FOR prüft selbstverständlich, ob auch dieses Token vorliegt, liest noch den Endwert ein und – sofern angegeben – auch die Schrittweite hinter einem STEP-Token.

INTPRT \$a7ae: Interpreterschleife

Schon in 3.4.5 haben Sie das Prinzip der Interpreterschleife kennengelernt. Hier soll noch eine weitere Differenzierung, nämlich in zwei Teile der Interpreterschleife, vorgenommen werden.

1. INTPRT (\$a7ae–\$a7e0 und \$a807–\$a80d): Schleifenkonstruktion

Dies ist die eigentliche Schleife, in der das Basic-Programm byteweise abgearbeitet wird. In ihr erfolgt auch die Abfrage der STOP-Taste, genauer gesagt: die Berücksichtigung der STOP-Taste mit eventuellem Programmabbruch, denn die Abfrage selbst geschieht im Interrupt. Somit ist auch zu erklären, daß während der Ausführung eines Basic-Befehls kein Abbruch möglich ist, sondern erst danach

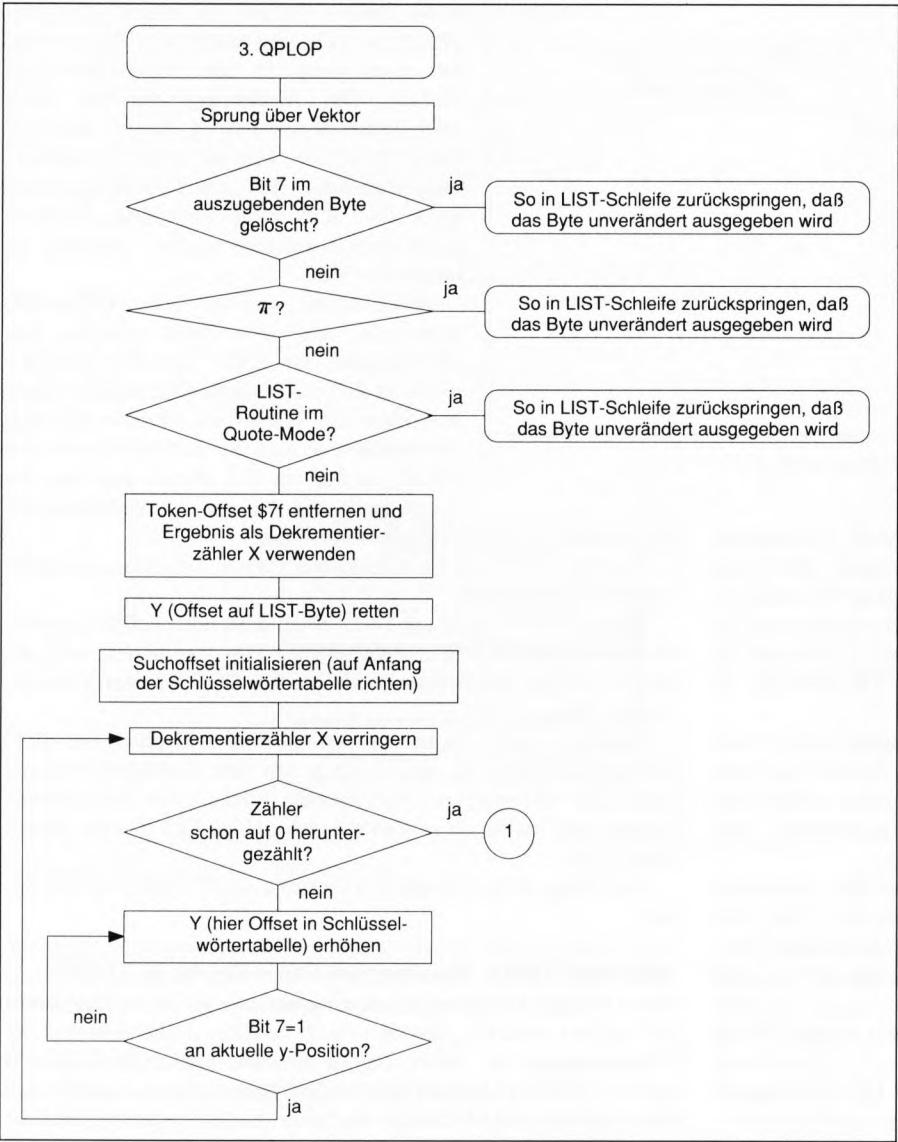


Abbildung 4.8: Die Aufgabenverteilung in der LIST-Routine (Teil 3)

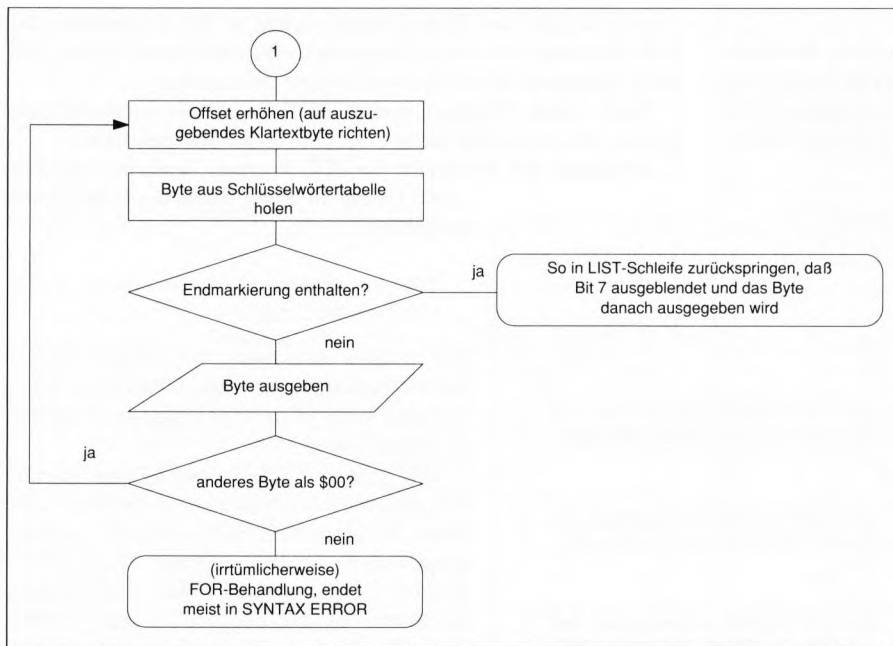


Abbildung 4.8: Die Aufgabenverteilung in der LIST-Routine (Teil 4)

oder davor, also dann, wenn die Interpreterschleife durchlaufen wird. Eine Ausnahme bilden lediglich LIST (siehe dort) und Operationen wie LOAD/SAVE/VERIFY, die sich ja auf Betriebssystemroutinen mit STOP-Tastenabfrage stützen. Dann spricht man allerdings nicht vom herkömmlichen STOP-Abbruch (»BREAK IN zeile«), sondern von der Fehlermeldung »BREAK ERROR IN zeile«.

INTPR (>\$a7ae) erwartet jeweils an der CHRGET-Zeiger-Position einen abzuarbeitenden Befehl. Vor seiner Ausführung wird seine Adresse als CONT-Fortsetzungsadresse gemerkt, sofern sich der C64 im Programm-Modus befindet; im Direktmodus sind CONTs nicht möglich.

Handelt es sich beim auszuführenden Byte um eine Zeilenendmarkierung, wird der CHRGET-Zeiger auf das erste Byte der nächsten Basic-Zeile gerichtet und dieses Byte dann ausgeführt. Eine Befehlsendmarkierung (Doppelpunkt \$3a) wird ignoriert und das unmittelbar darauffolgende Byte interpretiert.

Bei Erreichen des Programmendes (\$00 \$00 \$00) wird der END-Befehl simuliert.

Ein einzelnes Byte wird interpretiert, indem die INTPRT-Routine eine weitere Routine anspricht:

2. GONE (\$a7e1–\$a806 und \$a80e–\$a81c): einzelnes Byte ausführen

Solange der Vektor IGONE \$0308/\$0309 nicht verändert wird, liegt die GONE-Routine an den angegebenen Adressen. Ihre Tätigkeit läßt sich kurz beschreiben: Das Byte an der aktuellen CHRGET-Position wird in den Akku geholt (\$a7e4), in einem Unterprogramm ab \$a7ed ausgeführt und dann erfolgt ein Rücksprung in die Interpreterschleife (\$a7ea). Die Ausführung des im Akku befindlichen Bytewertes ist also der entscheidende Teil. Dort wird bei einer Endmarkierung (Zeilenende \$00 oder Doppelpunkt \$3a) sofort über RTS zurückgesprungen, da diese Bytes nicht ausgeführt werden, sondern zu ignorieren sind.

Anschließend wird der Token-Offset \$80 abgezogen, um einen Wert zwischen \$00 (Befehlstoken für END) und \$4b (Befehlstoken für GO) zu erhalten. Kommandos haben normalerweise Tokens im Bereich \$80–\$a2, also ergeben sie nach der Subtraktion von \$80 Werte von \$00 bis \$22, mittels derer aus der Adreßtabelle die Adresse der Befehlsroutine

entnommen und angesprungen werden kann.

Erkennt GONE ein Funktionstoken (Werte \$23–\$4a), so erfolgt ein SYNTAX ERROR.

Liegt das GO-Token vor (Wert \$4b), wird noch auf TO geprüft; ist also die gesperrte Schreibweise »GO TO« identifiziert, wird die GOTO-Routine angesprungen, als ob das GOTO-Token gefunden worden wäre.

Handelt es sich um kein Token, so wird schließlich die LET-Routine angesprungen, da es sich ja um eine Variablenzuweisung ohne LET (Beispiel: »A=5«) handeln könnte. Die LET-Routine erkennt dann selbst, ob es sich um eine gültige LET-Syntax handelt oder nicht.

Abbildung 4.10 stellt die Verwicklung von INTPRT und GONE dar.

RESTORE \$a81d: Routine zum Basic-Befehl RESTORE

Diese Routine berechnet die Anfangsadresse des Basic-Programms im Speicher minus 1, also theoretisch gesprochen die »Adresse der Endmarkierung der nullten Programmzeile«. Dieser Ausgangswert wird in den DATA-Zeiger geschrieben. Damit ist sichergestellt, daß beim nächsten READ-Befehl die Suche nach dem ersten DATA-

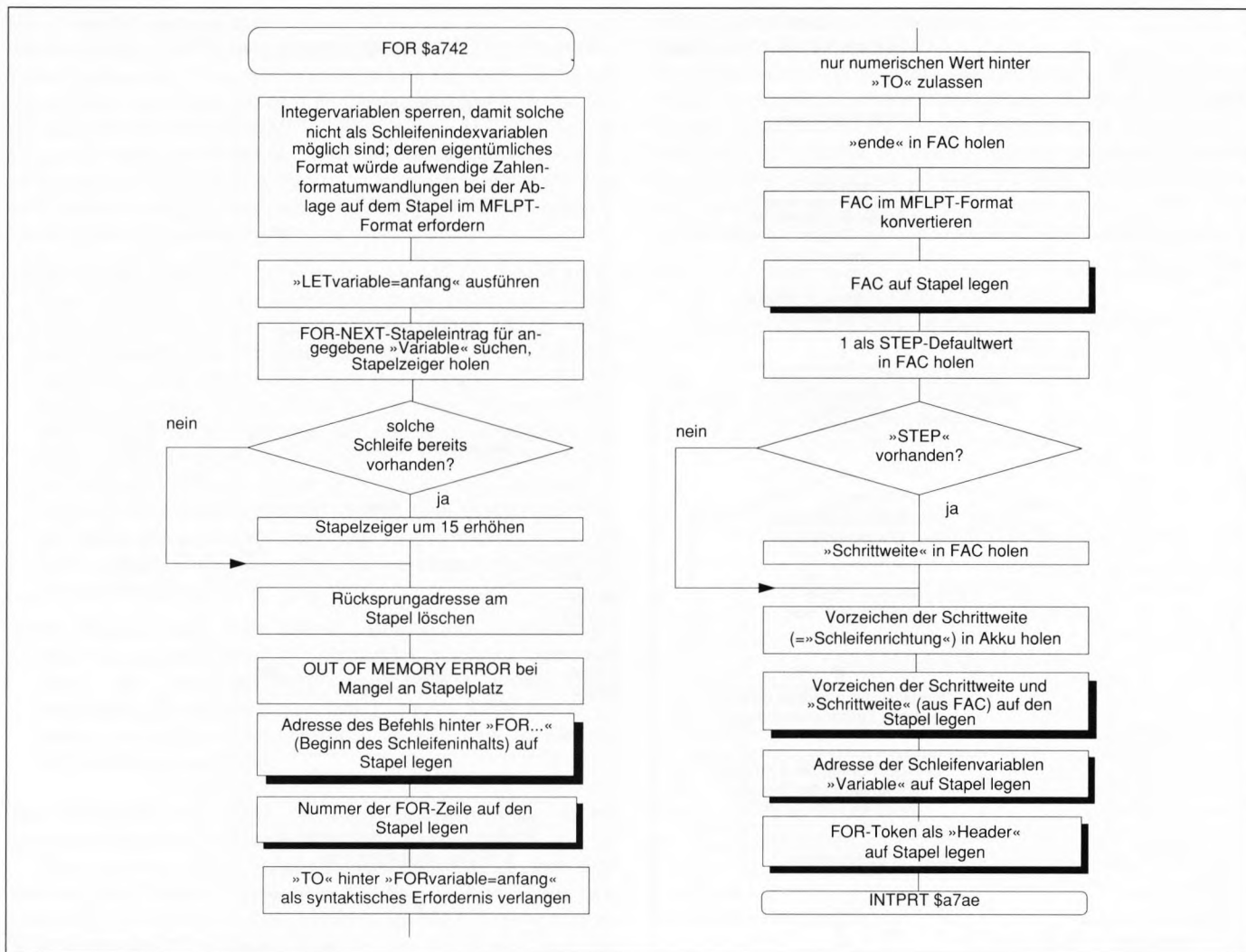


Abbildung 4.9: Ablauf der FOR-Routine

Statement von der ersten Programmzeile an durchgeführt wird. Somit erklärt sich auch die Subtraktion von 1: Der erste DATA-Befehl könnte ja schon in der ersten Programmzeile stehen.

BSTOP \$a82c: Berücksichtigung der möglicherweise gedrückten STOP-Taste

Die STOP-Taste wird bekanntlich im Interrupt abgefragt; ins laufende Basic-Programm eingreifen kann die IRQ-Routine jedoch nicht. Deshalb muß außerhalb des IRQ regelmäßig (vor und nach

jedem Befehl, also in der Interpreterschleife) geprüft werden, ob die Interrupt-Routine ein Auslösen von <STOP> entdeckt hat. Dazu wird die Routine STOP (\$ffe1) des Kernal herangezogen. Nach dieser Routine sind Zero- und Carry-Flag gelöscht, wenn <STOP> nicht betätigt wurde, bei gedrücktem <STOP> sind Zero- und Carry-Flag gesetzt.

Unmittelbar auf den Kernal-Aufruf folgt im Speicher die Routine zum Befehl STOP (Token: \$90), der in jedem Fall einen BREAK-Abbruch auslöst:

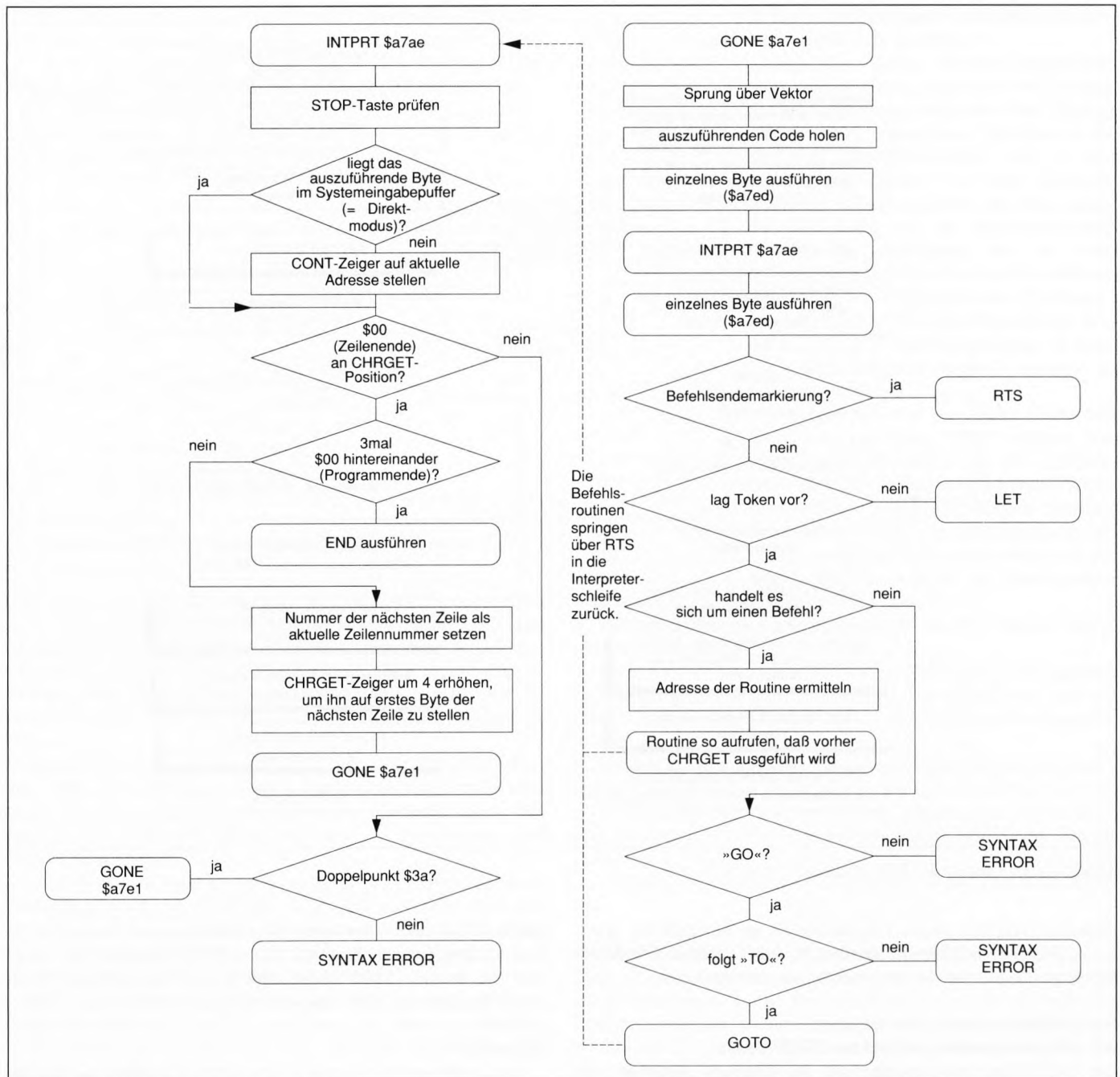


Abbildung 4.10: Die Interpreterschleife

STOP \$a82f: Routine zum Basic-Befehl STOP sowie Bestandteil der BSTOP-Routine

Diese Routine dient also zwei Herren gleichzeitig: Zum einen muß sie in jedem Fall einen Abbruch des Programms auslösen, da dies die Aufgabe des STOP-Befehls ist; zum anderen aber darf sie das Programm nicht unterbrechen, wenn BSTOP (\$a82c) feststellen mußte, daß <STOP> nicht gedrückt wurde. Betrachten wir also den BCS-Befehl bei \$a82f. Dieser verzweigt also bei zwei Bedingungen, von denen nur jeweils eine erfüllt sein kann:

1. Die Routine wurde als Befehlsroutine zum STOP-Befehl aufgerufen, und auf den STOP-Befehl folgt keine Ziffer. Dies ist ohnehin ein syntaktisches Erfordernis von STOP (STOP kennt keine Parameter), insofern kann man sagen, daß der BCS-Befehl immer dann verzweigt, wenn er als Befehlsroutine zum STOP-Befehl aufgerufen wurde. Dann hat der angesprungene BNE-Befehl bei \$a832 die Funktion, auf eine Befehlsendmarkierung hinter dem STOP-Kommando zu prüfen. Liegt diese nicht vor, so verzweigt BNE zu einem RTS-Befehl, der wiederum in diesem Fall zu einem SYNTAX ERROR führt. Andernfalls wird bei \$a834 mit gesetztem Carry-Flag weitergearbeitet. Das seit \$a82f gesetzte Carry-Flag dient dann im folgenden als Flag für die Ausführung von STOP.
2. Die Routine wurde über BSTOP angesprungen, und die STOP-Taste wurde gedrückt (C=1 seit \$a82c). Nachdem auf diese Weise die gedrückte STOP-Taste erkannt wurde, wird automatisch die weitere Behandlung wie beim STOP-Befehl erfolgen, das gesetzte Carry dient also im folgenden als Flag für die Ausführung von STOP.

Der BNE-Befehl bei \$a832 verzweigt dann keinesfalls, da bei gesetztem Carry seit \$a82c auch das Z-Flag gesetzt ist.

Wird in diesem Fall 2 bei \$a82f nicht verzweigt, so wird nach dem in dieser Situation irrelevanten Löschen des Carry-Flags (das Carry-Flag ist ohnehin schon gelöscht) bei \$a832 bestimmt zum RTS gesprungen, da dann seit \$a82c neben dem Carry auch das Zero-Flag gelöscht ist. Der RTS-Rücksprung wiederum führt in diejenige Routine zurück, die BSTOP (\$a82c) aufgerufen hat.

END \$a831: Routine zum Basic-Befehl END

Als Befehlsroutine zu END löscht diese Routine das Carry-Flag als Flag für die Ausführung des END-Befehls (ab \$a834 laufen END und STOP parallel ab, und nur das Carry-Flag gibt Auskunft, um welchen Befehl es sich handelt).

Der BNE-Befehl bei \$a832 löst dann mittels RTS einen SYNTAX ERROR aus, wenn keine Endmarkierung (Zeilenende \$00 oder Doppelpunkt \$3a) auf den END-Befehl folgt; ansonsten wird bei \$a834 fortgefahren:

\$a834: gemeinsame Behandlung der Befehle STOP und END

Das Beenden bzw. Abbrechen eines Programms über END bzw. STOP ist in vielen Punkten auf gleiche Weise durchzuführen. Deshalb existiert dafür im Grunde genommen nur diese eine und einzige Routine, die bei der einzigen unterschiedlichen Behandlung das Carry-Flag benötigt, um zu wissen, welcher Befehl auszuführen ist (C=0: END; C=1: STOP).

Den Ablauf dieser Routine ab \$a834 schildert Abbildung 4.11.

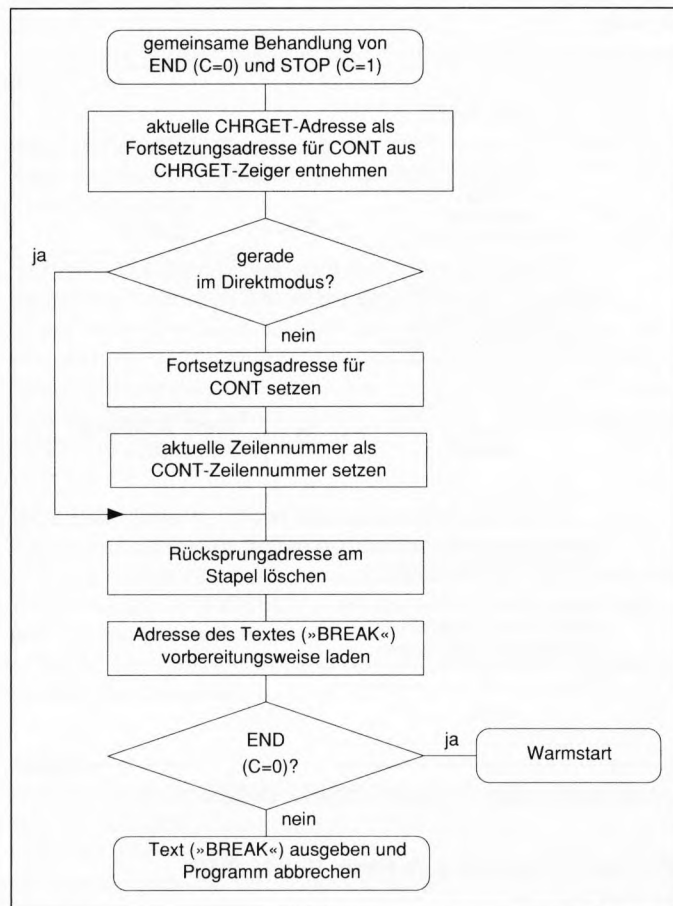


Abbildung 4.11: Die gemeinsame Behandlung von STOP (C=1) und END (C=0)

CONT \$a857: Routine zum Basic-Befehl CONT

Wie die Routinen zu RUN, NEW, CLR, END und allen anderen Befehlen, denen kein Parameter folgen soll, beginnt auch CONT

(\$a857) mit einer Syntax-Prüfung, die im Falle eines überflüssigerweise angegebenen Parameters einen SYNTAX ERROR auslöst.

Weiterhin wird geprüft, ob die CONT-Fortsetzung erlaubt ist; falls nicht, erfolgt ein CAN'T CONTINUE ERROR.

Sind jedoch alle Bedingungen erfüllt, wird die CONT-Fortsetzungsadresse in den CHRGET-Zeiger als aktuelle Interpretationsadresse und die CONT-Fortsetzungszeilennummer als aktuelle Basic-Zeile festgesetzt. Nach dem RTS-Rücksprung arbeitet dann die Interpreterschleife an der neu angegebenen Position, die durch CHRGET-Zeiger und CURLIN (Zeiger auf aktuelle Zeile, \$39/\$3a) bestimmt ist, weiter.

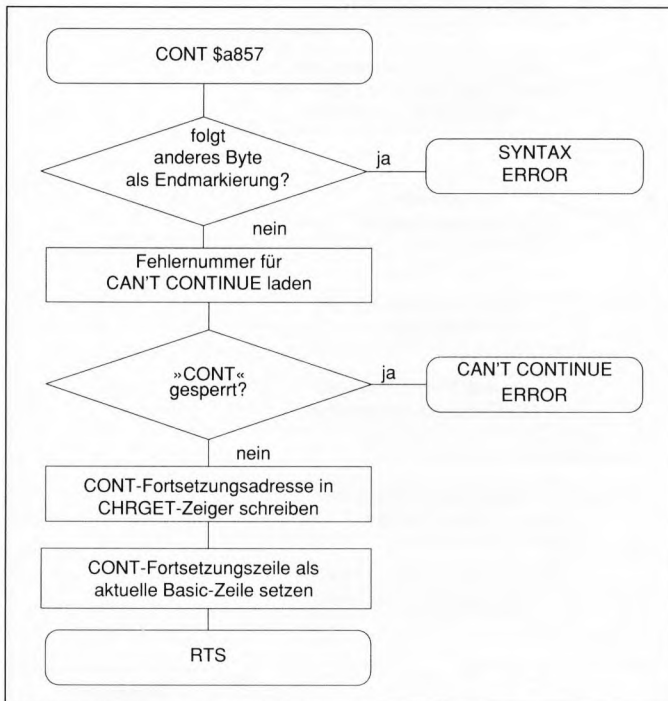


Abbildung 4.12: Ablauf des Basic-Befehls CONT

RUN \$a871: Routine zum Basic-Befehl RUN

Die Wirkung von

```
RUN
```

läßt sich auch durch zwei andere Befehle beschreiben:

```
CLR:GOTO programmanfang
```

(Korrekterweise ist noch hinzuzufügen, daß bei RUN auf den Programm-Modus umgeschaltet wird.)

Ebenso ist

```
RUN zeile
```

durch

```
CLR:GOTO zeile
```

umschrieben.

Somit ist also leicht einzusehen, daß RUN keine vollständig eigene Routine ist, sondern im wesentlichen die Verknüpfung zweier anderer Befehlsroutinen darstellt. Zuerst schaltet RUN also auf den Programm-Modus. Dann wird bei der Syntax »RUN« (ohne Zeilennummer) die NEWCLR-Routine (\$a659) ausgeführt, in der neben dem Löschen aller Variablen auch durch die Hilfsroutine STXTPT (\$a68e) die CHRGET-Zeiger auf den Programmanfang gerichtet werden; der RTS aus NEWCLR (\$a659) führt schließlich zurück in die Interpreterschleife, wo die Programmausführung automatisch am Programmanfang fortgesetzt bzw. begonnen wird.

Bei der Syntax »RUN zeile« wird zuerst die CLR-Routine durchlaufen und dann ein Teil der GOSUB-Routine aufgerufen, der sich wiederum auf die GOTO-Routine stützt.

GOSUB \$a883: Routine zum Basic-Befehl GOSUB

Der Befehl GOSUB löst nicht nur einen Sprung zur Zielzeile aus, zu dem die GOSUB-Routine übrigens die GOTO-Routine einsetzt, sondern muß auch einen GOSUB-RETURN-Stapeleintrag anlegen. Dessen Aufbau ist in 3.4.11 aufgegliedert worden.

Der Schlußteil von GOSUB (\$a883) – die Adressen \$a897–\$a89f – findet auch Verwendung von RUN (\$a871).

GOTO \$a8a0: Routine zum Basic-Befehl GOTO

Die GOTO-Routine liest zunächst die Nummer der anzuspringenden Zeile ein und berechnet den Offset vom GOTO-Befehl zur nächsten Programmzeile im Speicher.

Nun muß noch die Zielzeile im Speicher gesucht, der CHRGET-Zeiger darauf gerichtet und schließlich in die Interpreterschleife zurückgesprungen werden, damit die Interpretation an der Zielposition fortfährt. Man müßte jetzt also einen Aufruf von FNDLIN (\$a613) erwarten, doch GOTO (\$a8a0) versucht, die Suche zu optimieren. FNDLIN (\$a613) hat nämlich den Nachteil, daß diese Routine in jedem Fall vom Programmanfang aus die Suche der entsprechenden Zeile beginnt. GOTO hingegen unterscheidet zur Optimierung zwei verschiedene Fälle in wenigen Mikrosekunden Rechenzeit:

1. Sprung »nach vorne«

Wird beispielsweise von Zeile 100 nach Zeile 150 gesprungen, so wäre es unklug, die Suche am Programmanfang zu beginnen; schließlich läßt sich aus den Zeilennummern ersehen, daß die Zeile 150 auf keinen Fall vor der aktuellen Zeile (100) im

Speicher steht. Deshalb wird in dieser Situation die Suche erst ab der nächsten Zeile, zu welcher ja der Offset berechnet wurde, begonnen. Möglich ist dies durch einen späteren Einstieg in die FNDLIN-Routine, nämlich bei \$a617, der die Suche bei der durch Akkumulator und X-Register angegebenen Adresse beginnt.

2. Sprung »nach hinten«

Keine weitere Optimierung ergibt sich, wenn von Zeile 100 nach Zeile 50 gesprungen wird, da theoretisch die Zeile 50 sogar die erste Programmzeile sein kann; auf jeden Fall aber steht sie vor Zeile 100 im Speicher. Somit muß die Suche am Programmfang begonnen werden. Dazu wird die Anfangsadresse des Basic-Programms in Akku und X-Register geladen und aus Gründen der Einfachheit ebenfalls bei \$a617 eingesprungen. Rein theoretisch aber wäre ein FNDLIN-Einsprung bei \$a613 genauso korrekt.

Nach der Suche der Zeile muß noch sicherheitshalber festgestellt werden, ob sie auch vorhanden war; falls nein, wird die Meldung UNDEF'D STATEMENT ERROR ausgelöst. Andernfalls wird die Adresse der Zielzeile um 1 dekrementiert und der CHRGET-Zeiger auf diese Adresse gerichtet. Er zeigt somit auf das Nullbyte vor der Zielzeile, und beim über RTS bewirkten Rücksprung in die Interpreterschleife setzt diese den CHRGET-Zeiger auf das erste Byte der Zielzeile und stellt den Hilfszeiger CURLIN (\$39/\$3a), der jeweils die Nummer der aktuellen Basic-Zeile enthält, richtig.

Abbildung 4.13 erklärt noch einmal die Routinen zu RUN, GOSUB und GOTO, damit über deren Verknüpfungen keine Fragen mehr offen bleiben.

RETURN \$a8d2: Routine zum Basic-Befehl RETURN

Diese Routine sucht den letzten GOSUB-RETURN-Eintrag am Stapel; wird kein solcher gefunden, erfolgt die Meldung RETURN WITHOUT GOSUB ERROR.

Andernfalls werden Zeilennummer und CHRGET-Zeigerinhalt für die Rücksprungposition vom Stapel in die entsprechenden Hilfszeiger (CURLIN und CHRGET-Zeiger) übertragen; dann wird der darauffolgende Befehl gesucht und angesprungen, wofür die im Speicher unmittelbar folgende DATA-Routine verantwortlich ist.

DATA \$a8f8: Routine zum Basic-Befehl DATA

Der Befehl DATA dient nur zur Einleitung einer Folge von Daten, die über READ eingelesen werden können. Selbst hat er jedoch keine Befehlswirkung.

Der Unterschied zu REM ist allerdings, daß nach REM der Rest einer Basic-Zeile völlig ignoriert wird, während nach DATA auch ein Doppelpunkt und weitere Befehle folgen können. Deshalb muß DATA das nächste Trennzeichen (Doppelpunkt \$3a oder Zeilenende

\$00) suchen und bei diesem die Programmausführung fortsetzen, während REM lediglich auf das Zeilenende (\$00) zusteuert.

Dazu wird die Hilfsroutine GOSNXT (\$a906) eingesetzt; um deren Ergebnis – der Offset vom aktuellen CHRGET-Zeiger zur nächsten Befehlsendmarkierung – wird dann der CHRGET-Zeiger erhöht und an dieser neu ermittelten CHRGET-Position die Basic-Interpretation fortgesetzt.

Im ROM-Listing (siehe Kapitel 1) bezieht sich der durch die geschweifte Klammer gekennzeichnete Kommentar übrigens nur auf die Situation, daß DATA (\$a8f8) als Teil der RETURN-Routine abläuft.

Auch von anderen Routinen wird die DATA-Routine verwendet; dabei findet

ADCGPT (\$a8fb): Einsprung zur Addition des Y-Registers zum CHRGET-Zeiger Verwendung.

GOSNXT (\$a906): Offset vom aktuellen CHRGET-Zeiger zur nächsten Befehlsendmarkierung ermitteln

Dieser Einsprung sucht ab dem aktuellen CHRGET-Zeiger nach dem nächsten \$3a- oder \$00-Code, der also als Befehls- oder sogar Zeilen-Endmarkierung fungiert. Der Offset zu diesem Trennbyte wird im Y-Register zurückgegeben. GOSNXT (\$a906) stützt sich auf dieselbe Suchroutine wie

GOSEND (\$a909): Offset vom aktuellen CHRGET-Zeiger zur nächsten Zeilenendmarkierung ermitteln

Während GOSNXT (\$a906) sowohl nach \$00 als auch nach \$3a Ausschau hält, ist GOSEND nur auf das nächste durch \$00 bezeichnete Zeilenende abgerichtet.

Auch hier wird die von GOSNXT (\$a906) eingesetzte Suchroutine verwendet:

\$a90b: Suchroutine für GOSNXT (\$a906) und GOSEND (\$a909)

Wie Sie aus den vorausgegangenen Beschreibungen der Routinen GOSNXT (\$a906) und GOSEND (\$a909) wissen, handelt es sich um zwei Routinen mit fast derselben Aufgabe. Um Speicherplatz im ROM zu sparen, wird nur eine einzige Suchschleife verwendet, nämlich diese hier ab \$a90b.

Zum Verständnis der Routine ist zu beachten, daß sie zum einen nach \$00, zum anderen nach dem im X-Register übermittelten Code sucht, der natürlich ebenfalls \$00 sein kann. Bei \$a911 beginnt dann die eigentliche Suchschleife, die als Suchbytes folgende Werte geliefert bekommt:

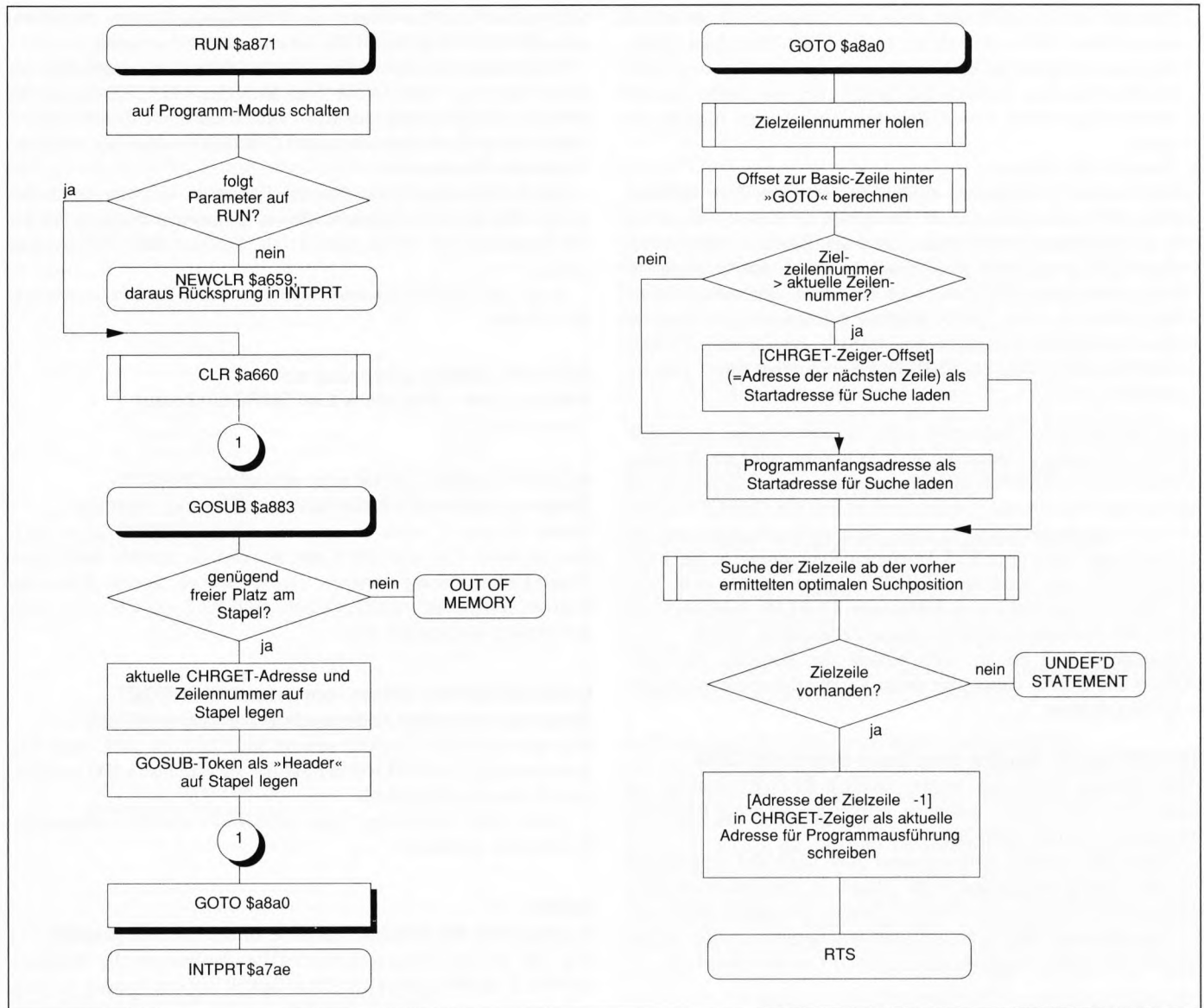


Abbildung 4.13: Verbindung von RUN, GOSUB und GOTO

GOSNXT: Suchbyte 1 = \$3a; Suchbyte 2 = \$00
 GOSEND: Suchbyte 1 = \$00; Suchbyte 2 = \$00

Die beiden Suchbytes stehen in den Zeropage-Adressen \$07 und \$08.

Kommen wir nun auf die Suchschleifenkonstruktion zu sprechen. Am Anfang der Suchschleife werden die beiden Suchbytes vertauscht. Dann wird das Byte an der aktuellen Position (CHRGET-Adresse + Y-Offset, wobei der Y-Offset mit dem Wert \$00 startet) ausgelesen. Handelt es sich um ein Zeilenende, so wird über RTS

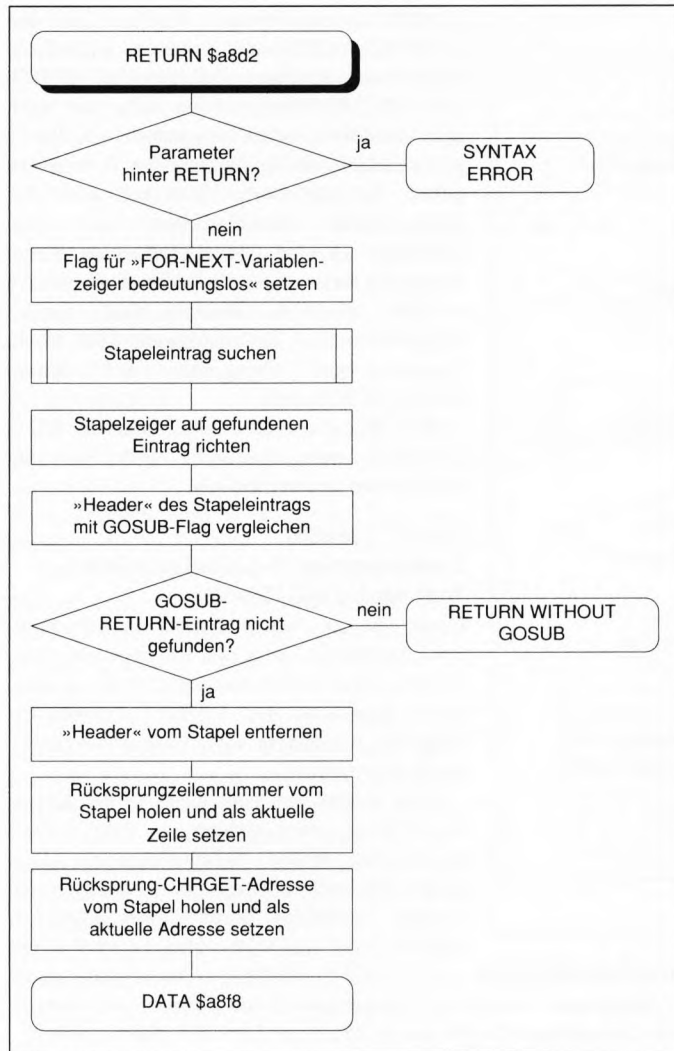


Abbildung 4.14 : Der Ablauf des RETURN-Befehls

an die aufrufende Routine zurückgesprungen. Andernfalls wird ein Vergleich mit Suchbyte 2 durchgeführt. Im Falle von GOSEND (\$a909) ist dies \$00, worauf ja schon der BEQ-Befehl bei \$a91b geprüft hat; dann ist der Vergleich mit Suchbyte 2 überflüssig. Andernfalls – bei GOSNXT (\$a906) – ist das Suchbyte 2 nur dann \$00, wenn das aktuelle Byte innerhalb von Anführungszeichen steht; außerhalb von Anführungszeichen ist Suchbyte 2 bei GOSNXT (\$a906) \$3a, also der Doppelpunkt, dessen Auffinden ebenfalls einen Schleifenabbruch veranlaßt.

Um den Quote Mode bei der GOSNXT-Suche nach \$3a zu berücksichtigen, werden also bei jedem aufgefundenen Anführungszeichen die beiden Suchbytes vertauscht; ansonsten wird die Suche ohne Vertauschung der Suchbytes beim nächsten Byte fortgesetzt, bis eine Markierung aufgefunden wurde.

Abbildung 4.15 beschreibt den Ablauf von GOSNXT (\$a906). Listing 4.4 ist eine verkürzte Form von GOSEND (\$a909); so würde die Routine also aussehen, wenn sie extra programmiert und nicht als GOSNXT-Sonderfall realisiert worden wäre. Ich meine, daß Listing 4.4 keine theoretische Spielerei ist, sondern ganz gut zeigt, worauf es bei GOSEND (\$a909) eigentlich ankommt.

```

LDY #0          ; Offset initialisieren
SCHLEIFE LDA ($7a),Y ; Byte aus Basic-Text holen
BEQ $a905       ; RTS-Befehl anspringen,
                ; wenn Zeilenende $00
                ; bei aktuellem Offset
                ; aufgefunden wurde
INY             ; Offset erhöhen
JMP SCHLEIFE    ; Schleife mit neuem
                ; Offset fortsetzen
  
```

Listing 4.4: Darauf läßt sich GOSEND reduzieren (dient nur zur Erklärung)

IF \$a928: Routine zum Basic-Befehl IF

Die Kürze der IF-Routine (35 Bytes) wird jene erstaunen, die eine aufwendige Routine zur Behandlung von logischen Verknüpfungen und Vergleichsoperatoren erwarten. Wenn man allerdings weiß, daß in den Routinen zur Auswertung von Parametern (FRMEVL & Co.) bereits alle derartigen Operationen regelrecht »ausgerechnet« werden, ist der prinzipielle Ablauf von IF klar: Über FRMEVL (\$a9e) wird der IF-Ausdruck wie ein gewöhnlicher Parameter ausgewertet. Das Ergebnis ist ein numerischer Wert: »0« steht für eine unerfüllte IF-Bedingung, »-1« für einen wahren IF-Ausdruck.

Die IF-Routine selbst muß also nur noch anhand dieses numerischen Wertes entweder die THEN-Behandlung anspringen (-1) oder übergehen (0).

Für »IF bedingung THEN GOTO zeile« ist dabei auch die verkürzte Syntax »IF bedingung GOTO zeile« oder »IF bedingung THEN zeile« zugelassen, die sogar in der Abarbeitung um einen rein theoretisch vorhandenen, praktisch allerdings nicht feststellbaren Zeitabschnitt schneller und aufgrund des weggelassenen THEN-Befehls zudem speicherplatzsparender ist.

Die Behandlung für eine nicht-erfüllte IF-Bedingung (Ignorieren der restlichen Zeile) ist übrigens gleichzeitig die Routine zum Basic-Befehl REM (\$a93b–\$a93f):

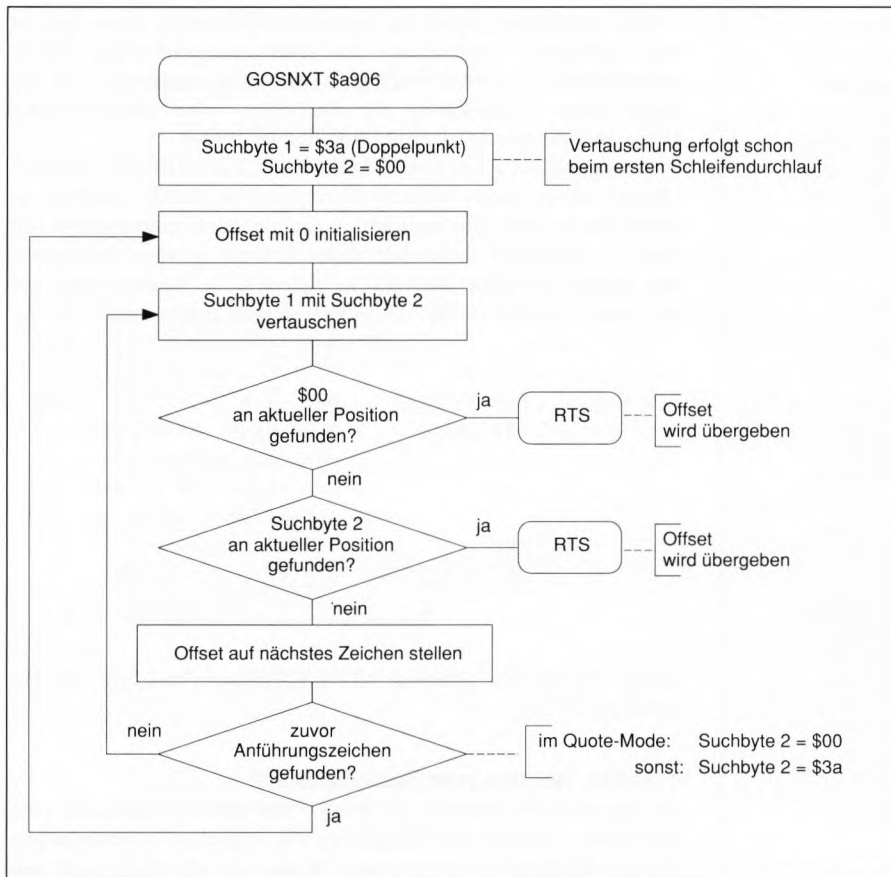


Abbildung 4.15: Der Ablauf von GOSNXT

REM \$a93b: Routine zum Basic-Befehl REM

Wie schon bei IF (\$a928) besprochen, wird hier der Offset zur Zeilenendmarkierung (Nullbyte) berechnet und zum CHRGET-Zeiger addiert, worauf dieser auf den Anfang der nächsten Zeile weist.

ON \$a94b: Routine zum Basic-Befehl ON

Die ON-Routine ist äußerst trickreich programmiert. Zunächst wird ein Bytewert, nämlich die Nummer des Sprungziels, ausgelesen. Das Zeichen hinter dem Bytewert wird dabei gemerkt, bis es bei \$a95b wieder Bedeutung trägt, doch soweit sind wir noch nicht. Zunächst wird nämlich sichergestellt, daß hinter »ON byte« kein anderes Kommando als GOSUB oder GOTO steht.

Dann wird mit Hilfe des Bytewertes als Dekrementierzähler und der CHRGET-Routine so lange nach Kommas gesucht, bis die tat-

sächlich anzuspringende Position an der CHRGET-Position steht. Dann schließlich kommt das gerettete Befehlstoken (GOTO oder GOSUB) wieder in den Akku und wird über einen trickreichen Einsprung an die Interpreterschleife als auszuführendes Byte übergeben; die Interpreterschleife ruft dann die entsprechende Befehlsroutine auf, die wiederum an der CHRGET-Position ihren Parameter sucht – und die vorher in der Suchschleife ermittelte Zeilennummer findet. Mögliche weitere Zeilennummern sind durch Kommas gut abgegrenzt und stören GOTO/GOSUB nicht.

Wie Sie also sehen, lohnt sich ein Blick ins ROM-Listing bestimmt, wenn Sie von ON (\$a94b) lernen möchten.

**LINGET (\$a96b):
Zeilennummer (0–63999) aus Basic-
Text nach \$14/\$15 holen**

Diese Routine wurde bereits in Abschnitt 3.4.7.3 erwähnt. Unter den Routinen zur Auswertung numerischer Parameter stellt sie eine große Ausnahme dar: LINGET (\$a96b) ist nicht im mindesten auf Fließkomma-Arithmetik angewiesen!

Dies äußert sich zum einen in der Kürze der Routine, zum anderen in ihrer außergewöhnlich hohen Geschwindigkeit. Der größte Nachteil soll aber nicht verschwiegen werden: Variablen werden von LINGET (\$a96b) nicht bearbeitet. Dies ist zwar beim

Hauptanwendungsfall von LINGET (\$a96b) – Auswertung einer Zeilennummer vor einer eingegebenen Basic-Zeile – von Vorteil, aber da sich auch GOTO und GOSUB auf LINGET (\$a96b) stützen, ist »GOTO variable« oder »GOSUB variable« nicht möglich, obwohl es durchaus seine Berechtigung hätte.

Eine weitere Einschränkung von LINGET (\$a96b) liegt darin, daß die Zeilennummer kleiner als 64000 zu sein hat, das interne Basic-Zeilenformat des Interpreters jedoch durchaus auch Zeilennummern im Bereich 64000–65535 problemlos verarbeiten könnte.

Und noch ein drittes Manko: LINGET ist nicht fehlerfrei! Durch gezielte Fehleingaben kann man diese Routine ganz schön durcheinanderbringen, auch wenn dies unabsichtlich kaum möglich ist. Geben Sie doch einmal folgende Zeilennummer gefolgt von <RETURN> ein:

350721

Die Wirkung ist normalerweise – zumindest unmittelbar nach dem Einschalten des C64 – daß die Teilinitialisierung mit Warmstart erfolgt, wie Sie es von <RUN/STOP RESTORE> kennen. Dies geschieht, da irrtümlicherweise der BRK-Befehl zur Ausführung gelangt. Wie es nun dazu kommt, das erkläre ich Ihnen gerne, wenn wir uns ein wenig mit LINGET (\$a96b) vertraut gemacht haben; dann ist es keine Schwierigkeit, die offensichtliche Fehlerquelle dafür zu finden, daß keine ordnungsgemäße Fehlermeldung erfolgt, sondern eine Art Systemabsturz.

In LINGET (\$a96b) enthält der Hilfszeiger \$14/\$15 den bis zum jeweiligen Zeitpunkt für die Zeilennummer ermittelten Wert. Deshalb wird er zunächst mit 0 initialisiert (\$a96b–\$a970).

Über CHRGET wird dann jeweils die nächste Ziffer geholt; vor dem Aufruf von LINGET (\$a96b) muß daher CHRGET ausgeführt sein, damit Akkumulator und Prozessorflags richtig gesetzt sind.

Wird keine Ziffer mehr im Basic-Text gefunden, endet die Routine mit dem augenblicklichen Stand von \$14/\$15; eine gefundene Ziffer wird hingegen zum Inhalt von \$14/\$15 addiert, wobei vorher der bisherige Inhalt von \$14/\$15 verzehnfacht wird, um Platz für die neue Stelle zu schaffen. Bei Auswertung der Zahl »42309« enthält \$14/\$15 also nach den jeweiligen Bearbeitungsschritten (1 Schritt = 1 Ziffer; daher 5 Schritte für 5 Ziffern) die folgenden Werte:

Initialisierung:	0
nach Schritt 1:	4 = 0*10 + 4
nach Schritt 2:	42 = 4*10 + 2
nach Schritt 3:	423 = 42*10 + 3
nach Schritt 4:	4230 = 423*10 + 0
nach Schritt 5:	42309 = 4230*10 + 9

Wird als High-Byte von \$14/\$15 vor einer solchen Zehnermultiplikation ein Wert ≥ 25 gefunden, bedeutet dies aufgrund mathematischer Gesetze, daß eine Zeilennummer über 63999 vorliegt ($64000 = 25 * 256 * 10$). Bei \$a97b/\$a97d unterläuft nun ein Fehler, der nur bei den Zahlen von 350720 ($\$89 * 256 * 10$) 353729 ($= \$8a * 256 * 10 - 1$) auftritt: Als High-Byte steht \$89 im Akku, und

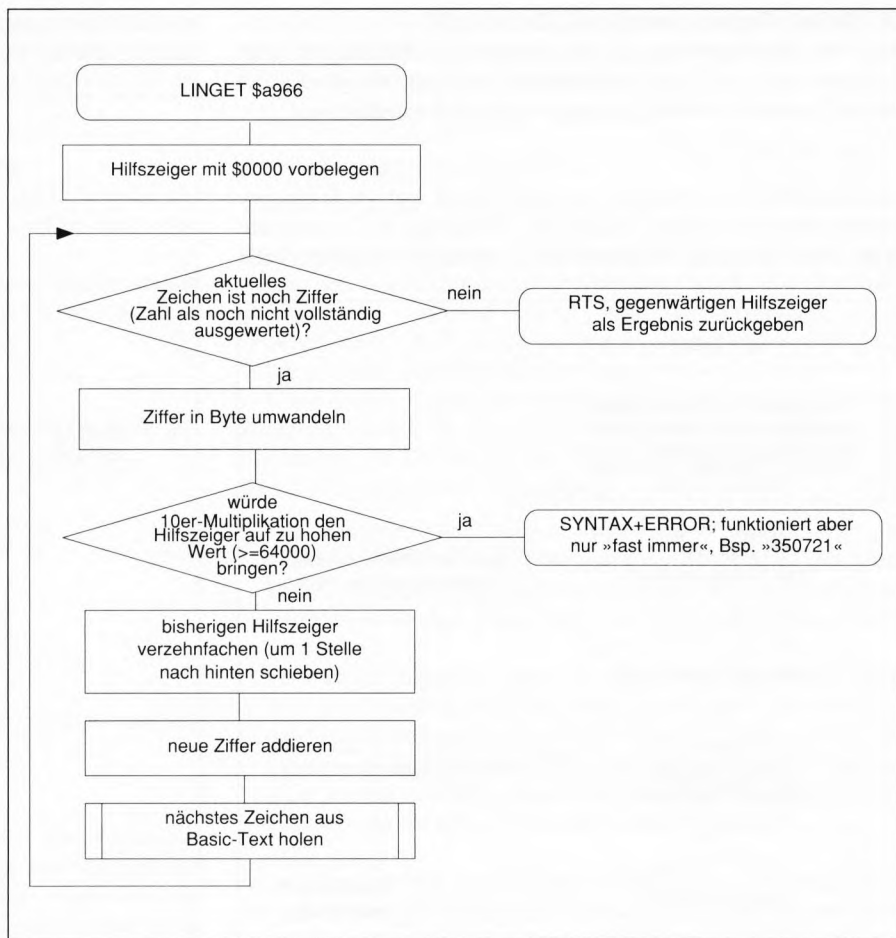


Abbildung 4.16: Der Ablauf der LINGET-Routine

bei \$a953, der angesprochenen Adresse, liegt ein positives Vergleichsergebnis vor (siehe \$a953), woraufhin nicht der eigentlich von LINGET (\$a96b) gewünschte SYNTAX ERROR ausgelöst wird, sondern andere Operationen durchlaufen werden; unter diesen fälschlicherweise ausgelösten Befehlen ist nun auch ein PLA bei \$a95b zu finden, der die am Stapel befindliche Rücksprungadresse verfälscht, so daß beim nächsten RTS der Prozessor ins Leere springt, genauer gesagt nach \$79a5 (#31141). Nach

POKE 31141,2

hat die Eingabe von »350721« sogar einen Systemabsturz zur Folge (2 ist ein undefinierter Opcode, der ein »Aufhängen« des Prozessors bewirkt).

LET \$a9a5: Routine zum Basic-Befehl LET

Diese Routine zum Anlegen beziehungsweise Neubelegen einer Variablen ist in mehrere Teilabschnitte unterteilt, die jeweils auf den vorliegenden Variablentyp eingehen. Auch Sonderfälle wie

```
LET TI$="123015"
```

zum Einstellen der Systemuhr auf »12 Uhr 30 und 15 Sekunden« werden von LET (\$a9a5) gemeistert. Abbildung 4.17 verschafft Ihnen einen schnellen Gesamtüberblick über alle möglichen Fälle

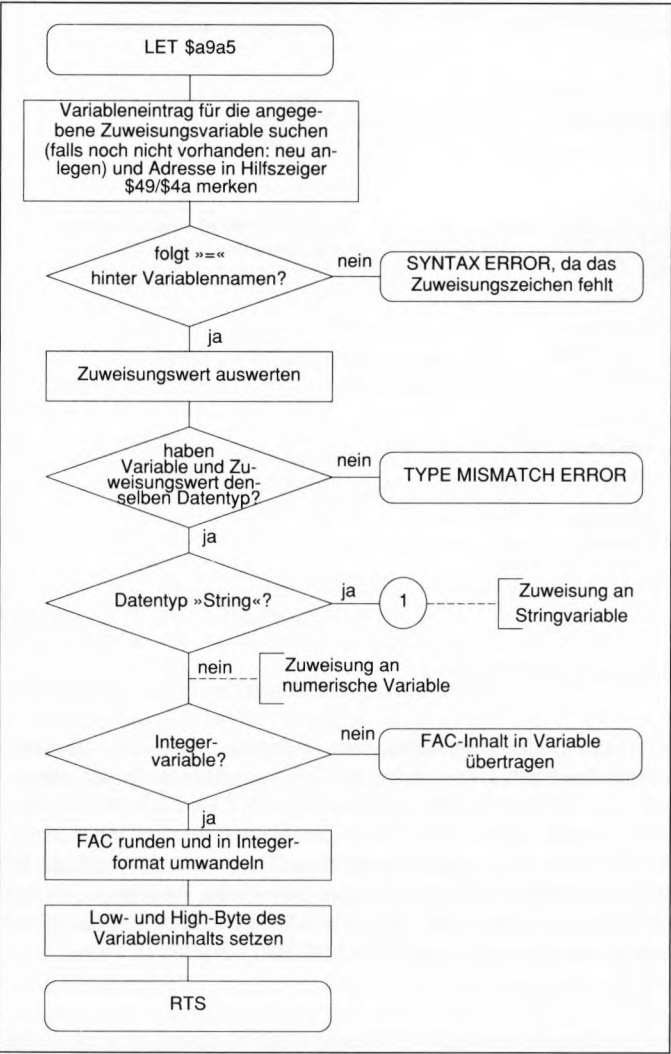


Abbildung 4.17: So geht eine LET-Zuweisung vor sich (Teil 1)

von LET-Zuweisungen und deren Behandlung, allerdings wird dabei nicht so sehr ins Detail gegangen wie bei bisherigen Flußdiagrammen

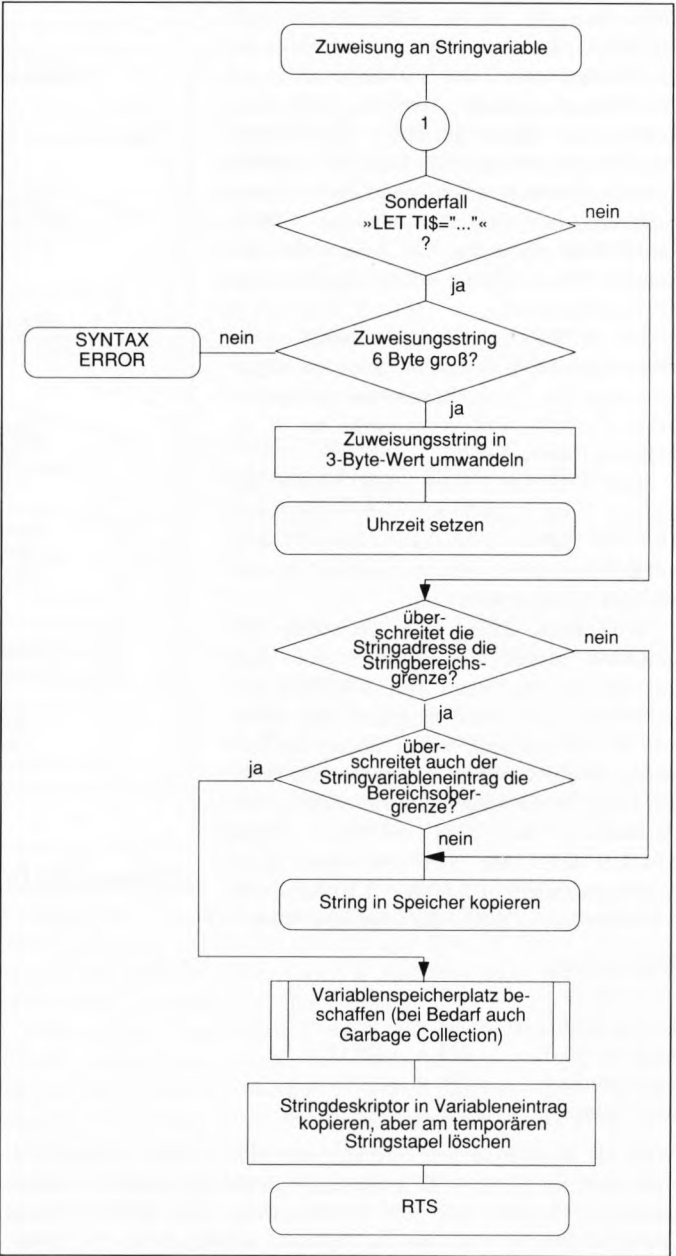


Abbildung 4.17: So geht eine LET-Zuweisung vor sich (Teil 2)

men; dafür gibt es aber Kapitel 1, das ROM-Listing, das mit seinem ausführlichen Kommentar kein einziges Byte unerklärt läßt.

Ein Teil der LET-Routine ist als allgemeines Unterprogramm interessant:

STRCGT (\$aa1d): Zeichen aus String holen

Liegen die Adresse eines Strings in \$22/\$23 und der Offset zum aktuell auszuwertenden Byte des Strings in Y, wird er durch »jsr strcgt« byteweise von links nach rechts gelesen.

Bei Auffinden einer Ziffer wird diese zum Akku addiert; andere Zeichen als Ziffern lösen einen ILLEGAL QUANTITY ERROR aus.

\$aa80: Routine zum Basic-Befehl PRINT#

Der Befehl PRINT# läßt sich durch zwei andere Befehle beschreiben und teilweise sogar ohne weiteres ersetzen: Bei

```
PRINT#file, "text"
```

wird nur für die Ausgabe von »text«, also den Befehl

```
PRINT "text"
```

die Ausgabe auf »file« umgelenkt:

```
CMD file
```

Dabei ist noch nicht einmal bedacht, daß der CMD-Befehl auch folgende Syntax zuläßt:

```
CMD file, "text"
```

Berücksichtigt man auch dies, kann man sich die Kürze der Befehlsroutine zu PRINT# (6 Bytes) erklären: Es wird lediglich auf den CMD-Befehl umgestiegen und danach sofort wieder auf die herkömmliche Ausgabe umgeschaltet (Basic-Kernal-Einsprung für CLRCHN).

CMD \$aa86: Routine zum Basic-Befehl CMD

Die mindestens erforderte Syntax ist »CMD filename«, um die Ausgabe auf ein gewünschtes File umzulenken. Dazu wird einfach die »filename« eingelesen und über einen Basic-Kernal-Einsprung die CKOUT-Routine aufgerufen. Stellt CMD (\$aa86) dann noch fest, daß ein zusätzlicher Parameter »text« folgt, wird dieser an die PRINT-Befehlsroutine weitergeleitet.

PRINT (\$aa9a–\$aac9 und \$aae8–\$ab1d;

Einsprung bei \$aaa0): Routine zum Basic-Befehl PRINT

Diese Routine ist, da dem PRINT-Befehl eine variable Zahl von Parametern gleichen Ranges, die nacheinander abzuarbeiten sind, folgt, als Schleife konstruiert; sie wird erst bei Auffinden einer Endmarkierung verlassen, davor erhält jeder PRINT-Parameter Ein-

zelbehandlung. Komma (»,«) und Semikolon (»;«) sind zur Abgrenzung der Parameter voneinander möglich, jedoch oftmals nicht syntaktisch erforderlich. Beispielsweise kann

```
PRINT TAB(10); "INHALT VON Z: "; Z
```

auch durch folgendes ersetzt werden:

```
PRINT TAB(10) "INHALT VON Z: " Z
```

Zwei aufeinanderfolgende Fließkommavariablen müssen jedoch wegen der Variablennamenunterscheidung durch ein Semikolon getrennt sein, während String- oder Integervariablen durch Prozentzeichen (bei Integervariablen) oder Dollarzeichen (bei Stringvariablen) eindeutig definiert sind.

Diese Syntax-Regelungen sind Ihnen sicher längst bekannt; eine Zusammenfassung in dieser Form erleichtert aber bestimmt das Verständnis der PRINT-Schleife. Diese beginnt normalerweise bei \$aaa2, doch es gibt zwei Ausnahmen:

- Bei Ausgabe eines vorher ausgewerteten Strings wird die Schleife schon bei \$aa9a, also sogar vor dem PRINT-Einsprung \$aaa0, begonnen, da \$aa9a/\$aa9d den String ausgeben und die Interpretation ab dem direkt nach dem String stehenden Zeichen veranlassen.
- Beim ersten Einsprung, also aus der Interpreterschleife, beginnt die PRINT-Schleife schon bei \$aaa0, wo im Falle einer vorliegenden Befehlsendmarkierung noch Carriage Return (CR) und gegebenenfalls Line Feed (LF) ausgegeben werden. Dadurch wird der Fall »PRINT« (ohne Parameter) berücksichtigt, der die automatische Ausgabe eines Zeilenvorschubes bewirkt.

Zu beachten ist auch noch, daß die »Funktionen« TAB und SPC in bezug auf ihre programmtechnische Abwicklung seitens des Basic-Interpreters gar keine Funktionen sind: In der PRINT-Routine werden sie statt dessen als Parameter (ähnlich Komma und Semikolon) erkannt und ausgeführt; somit erklärt sich, warum sie beispielsweise in Stringvariablenzuweisungen unzulässig sind.

Aus Gründen der einfacheren Syntax-Prüfung lautet der Klartext zu den TAB- und SPC-Tokens daher auch »TAB(« bzw. »SPC(«, so daß die Prüfung auf die offene Klammer gewissermaßen schon bei der Tokenisierung erfolgt. Nachteilhaft für den Basic-Programmierer ist dabei allerdings, daß er nicht die gesperrte Schreibweise »TAB (10)« verwenden darf, obwohl beispielsweise »SIN (2)« durchaus erlaubt ist.

Abbildung 4.18 stellt die Interpretation einer PRINT-Anweisung dar.

Fortsetzung von GETSYB: \$aaca–\$aae7

Hinsichtlich der Speicheraufteilung liegt übrigens mitten in der PRINT-Routine – bei \$aaca–\$aae7 – noch die Fortsetzung einer Un-

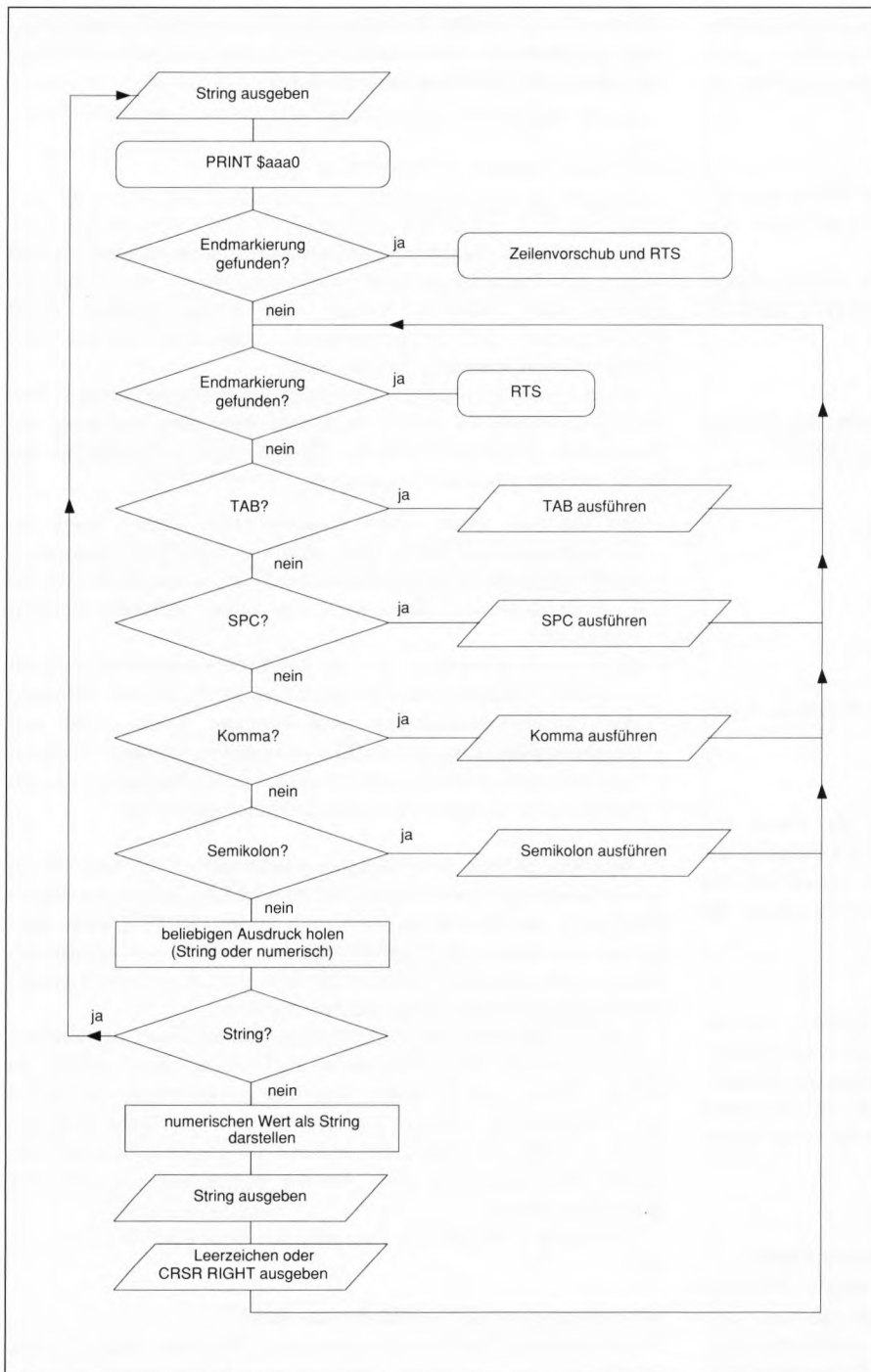


Abbildung 4.18: So wird PRINT interpretiert

teroutine des Warmstarts. Bei der Dokumentation der GETSYB-Routine (\$a560) wird auch dieser »Routinenschwanz«, der eine \$00-Markierung ans Pufferende schreibt, erläutert.

Der Grund, warum diese Routine im Speicher so nahe an PRINT (\$aaa0) liegt, ist dabei, daß der GETSYB-Abschluß als letztes das Carriage Return (CR, ASCII-Code \$0d) ausgibt sowie einen Line Feed (LF, \$0a) bei Ausgabe auf ein File mit einer größeren Filenummer als 127 (\$7f): Von \$aaa0 aus wird im Falle einer Verzweigung dorthin gesprungen.

STROUT (\$ab1e): String ausgeben

Zur Ausgabe eines Strings, der mit \$00 abgeschlossen ist und maximal 255 (\$ff) Byte umfassen darf, ruft der Basic-Interpreter die STROUT-Routine auf, wobei in Akku und Y-Register die String-Anfangsadresse übermittelt wird. Dazu wird zunächst der String ausgewertet (Berechnung der Stringlänge) und anschließend in einer BSOUT-Schleife auf das aktuelle Ausgabegerät gebracht.

Listing 4.5 ist ein Beispielprogramm, das nach seinem Start über SYS 49152 einen Text, der übrigens auch einige Steuerzeichen enthält, ausgibt.

PRTSTR (\$ab21): aktuellen String ausgeben

Ein späterer Einsprung in STROUT (\$ab1e) liegt bei PRTSTR (\$ab21); er setzt voraus, daß vorher über STRLIT (\$b487) die String-parameter geholt wurden. Bei Strings, die gerade über eine entsprechende Routine ausgewertet wurden und bei denen die String-hilfsspeicher auf den richtigen Werten stehen, ist PRTSTR (\$ab21) also der einfachere Einsprung, da er keine Parameter erfordert: Es gilt die Adresse des aktuellen Strings.

Ein dritter und letzter Einsprung, der vom Basic-Interpreter selbst nicht genutzt wird, soll hier auch noch erwähnt werden, weil er für eigene Programme aus einem einfachen Grund am empfehlenswertesten ist (er verwen-

READY.

```

100 -.BA $C000 : START: SYS 49152
110 -;
120 -; BEISPIEL ZU STROUT $AB1E
130 -;
140 -.GL STROUT = $AB1E
150 -;
160 -      LDA #<(TEXT)
170 -      LDY #>(TEXT)
180 -      JMP STROUT
190 -;
200 -TEXT      .BY 147      : CLEAR
210 -      .TX "C64 FUER INSIDER"
220 -      .BY 13      : CARRIAGE RETURN
230 -      .TX "-----"
240 -      .BY 13,13      : 2 * CR
250 -      .TX "NOW INSIDE STROUT!"
260 -      .BY 13,13,13,13; 4 * CR
270 -      .BY 0      : ENDMARKIERUNG

```

READY.

Listing 4.5: Beispiel zu STROUT (Fassung 1)

det keine weiteren Interpreter Routinen wie STRLIT (\$b487) oder FRESTR (\$b6a6) und besteht somit nur aus einer Ausgabeschleife):

READY.

```

100 -.BA $C000 : START: SYS 49152
110 -;
120 -; BEISPIEL ZU STROUT $AB24
130 -;
140 -.GL STROUT = $AB24 ; !!!!!
150 -;
160 -      LDA #<(TEXT)
170 -      LDY #>(TEXT)
180 -      STA $22
190 -      STY $23
200 -      LDA #TEXTENDE-TEXT
210 -      JMP STROUT
220 -;
230 -TEXT      .BY 147      : CLEAR
240 -      .TX "C64 FUER INSIDER"
250 -      .BY 13      : CARRIAGE RETURN
260 -      .TX "-----"
270 -      .BY 13,13      : 2 * CR
280 -      .TX "NOW INSIDE STROUT!"
290 -      .BY 13,13,13,13; 4 * CR
300 -TEXTENDE  .BA TEXTENDE
310 -      ; ^      LABEL DEFINIEREN/ ERZEUGT
320 -      ; ^      KEIN BYTE OBJEKT CODE!

```

READY.

Listing 4.6: Beispiel zu STROUT (Fassung 2)

\$ab24: Ausgabe eines Strings ohne Verwendung weiterer Stringroutinen

Diesen Vorteil der leichteren Verträglichkeit erkaufte man sich – äußerst preisgünstig, wie ich denke – mit der größeren Anzahl zu übergebender Parameter:

\$22/\$23 = Stringadresse
 Akkumulator = Stringlänge in Bytes

Listing 4.6 ist das Beispiel aus Listing 4.5, diesmal aber mit unserem neuen Einsprung bei \$ab24, der unkorrekterweise, aber der Einfachheit halber ebenfalls mit dem Label »STROUT« bezeichnet wird. Wie gesagt, diesen Einsprung sollten Sie auch in Ihren Programmen verwenden, selbst wenn er in anderen Werken bislang nicht angesprochen wurde und somit noch nicht so populär wie der \$able-Einstieg ist.

Die \$ab24-Lösung hat aber auch einen Nachteil, der nicht zu vergessen ist: Ist \$22/\$23 nicht ohnehin schon in einem Programm als Hilfszeiger auf den String gerichtet, ist die Parameterübergabe etwas umständlicher zu programmieren als beim Laden von A/Y mit der Stringadresse. Darüber tröstet auch nicht die Tatsache hinweg, daß die \$00-Endmarkierung am Textende weggelassen kann.

RGTSPP (\$ab3b): Leerzeichen oder CRSR RIGHT ausgeben

Will der Basic-Interpreter einen Text einrücken, so gibt er dazu am Bildschirm das Steuerzeichen CRSR RIGHT (\$1d), auf anderen Geräten (Floppy, Drucker) allerdings Leerzeichen (\$20) aus. Damit nicht jede einzelne Routine eine solche Prüfung auf das aktuelle Ausgabegerät vorzunehmen hat, existiert RGTSPP (\$ab3b). Bei »jsr rgtspc« wird das richtige Zeichen gedruckt. Die Ausgabe läuft über den Basic-Einsprung der Kernall-Routine BSOUT ab.

Die jeweiligen Teilroutinen für die beiden Fälle können auch direkt angesprochen werden:

SPCOUT (\$ab3f): Leerzeichen ausgeben RGTOUT (\$ab41): CRSR RIGHT ausgeben

Sogar zur Ausgabe eines Fragezeichens ist ein ähnlicher Einsprung vorhanden:

QUMOUT (\$ab44): Fragezeichen (>question mark<) ausgeben

Auch diese Routine besteht nur aus einem Zeichencode-Ladebefehl sowie der Basic-BSOUT-Routine:

BBSOUT (\$ab47): Basic-BSOUT-Behandlung

Diese Routine unterscheidet sich in der Anwendung von BSOUT (\$ffd2) nur darin, daß sie wegen zweier verschachtelter JSR-Aufrufe

vier Bytes mehr Stapelplatz benötigt und bei Ein-/Ausgabe-Fehlern automatisch die Basic-Fehlerbehandlung auslöst.

Fehlerbehandlung bei INPUT/READ/GET: \$ab4d

Tritt bei einem der Dateneingabebefehle (INPUT, READ oder GET) ein spezifischer Fehler auf, wird lediglich diese Fehlerbehandlungs-routine bei \$ab4d angesprungen, die dann anhand des Befehls, bei dem der Fehler auftrat, die richtige Fehlermeldung erzeugt. Ein einfaches Laden der Fehlernummer und darauffolgendes Anspringen des Fehlereinsprungs genügt dabei nicht, da noch weitere Behandlungen erforderlich sind:

GET: Bei GET soll keine Zeile ausgegeben werden, da lediglich der Benutzer aufgrund einer Eingabe des falschen Datentyps den Fehler veranlaßt hat. Deshalb wird vor der Bearbeitung des Fehlers auf Direktmodus umgeschaltet, also die Nummer der Fehlerzeile ignoriert.

READ: Verantwortlich ist nicht die Zeile, die den READ-Befehl enthält, sondern diejenige DATA-Zeile, in der das fehlerhafte Datum (Einzahl von »Daten«) stand. Deshalb muß bei möglicher-weise erfolgreicher Ausgabe der Fehlerzeile (»... ERROR IN zeile«) die DATA-Zeilenummer angegeben werden.

INPUT/INPUT#: Hier gibt es zwei Fälle. Entweder wurde anstelle einer Ziffer ein Buchstabe oder sonstiges unerlaubtes Zeichen ausgegeben, was die Meldung REDO FROM START und die erneute INPUT-Eingabe veranlassen muß, oder es handelt sich um Daten aus einem mit INPUT# angesprochenen File, die nicht verwertbar sind.

GET (\$ab7b):

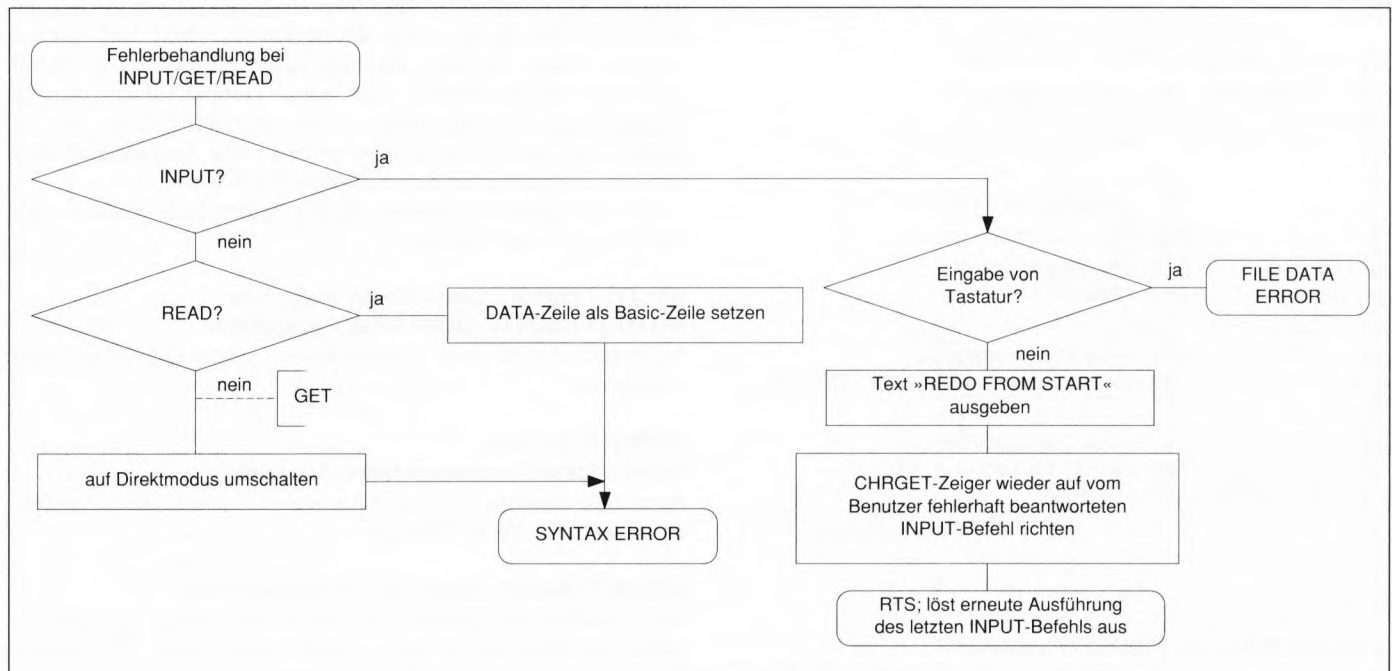
Routine zu den Basic-Befehlen GET und GET#

Wie bei jedem Eingabebefehl, erfolgt zunächst die Sicherstellung, daß sich der Computer auch im Programm-Modus befindet; im Direktmodus werden die Befehle INPUT, READ und GET mit der Fehlermeldung ILLEGAL DIRECT ERROR quittiert.

Als zweites werden die Befehle GET und GET# unterschieden; da GET# kein eigenes Token hat, muß geprüft werden, ob das Doppelkreuz (#) auf das GET-Token folgt. Da diese Prüfung mittels CHRGET durchgeführt wird und CHRGET Leerzeichen überliest, ist auch die gesperrte Schreibweise »GET #« anstelle der komprimierten Syntax »GET#« zulässig, obwohl »INPUT #« nicht erlaubt ist.

Die GET#-Routine unterscheidet sich von GET nur dadurch, daß zunächst die Eingabe auf das angegebene File umgelenkt wird;

Abbildung 4.19: Fehlerbehandlung für INPUT/READ/GET



danach wird die Routine für GET (ohne »#«) ausgeführt, als ob es sich um diesen Befehl handeln würde.

GET stellt im Grunde keine eigene Routine dar, sondern ruft lediglich eine allgemeine READ/INPUT/GET-Routine auf, wobei der Eingabepuffer auf 1 Byte begrenzt wird.

Nach dem Aufruf dieser generellen READ/INPUT/GET-Routine wird noch einmal festgestellt, ob sich GET oder GET# in der Ausführung befand; bei GET wird unverzüglich zurückgesprungen (RTS), bei GET# vorher noch auf das herkömmliche Eingabegerät, die Tastatur, geschaltet.

\$aba5: Routine zum Basic-Befehl INPUT#

Diese Routine lenkt die Eingabe auf das angegebene File um, ruft dann die INPUT-Routine auf und schaltet wieder auf Tastatureingabe zurück.

Es wird übrigens nicht die komplette INPUT-Routine durchlaufen, da diese die überflüssigen Fragezeichen ausgeben würde, die ja bei anderen Eingabegeräten als der Tastatur keinen Sinn haben, weil nicht der Benutzer zur Eingabe aufgefordert ist, sondern das entsprechende File auf Diskette oder Kassette, das wohl nichts mit einer derartigen Eingabeaufforderung anzufangen wüßte . . .

INPUT \$abbf: Routine zum Basic-Befehl INPUT

Die Syntax von INPUT ist bekanntlich recht vielfältig. So kann der Benutzer oft zwischen verschiedenen syntaktisch zulässigen Varianten entscheiden, z.B. ob ein Kommunikationstext in einem String übergeben werden soll oder nicht. Dieser muß als String in Anführungszeichen unmittelbar hinter dem INPUT-Befehl stehen; dann wird er vor dem Einholen der Eingabe am Bildschirm ausgedruckt. Die dazu erforderliche Behandlung steht am Anfang der INPUT-Routine. Der Kommunikationsstring wird dabei nur daran erkannt, daß ein Anführungszeichen hinter dem INPUT-Befehl vorhanden ist. Ebenso erforderlich ist ein Semikolon hinter dem Kommunikationsstring.

Eine INPUT-Eingabe wird dann über die allgemeine INPUT/READ/GET-Routine wie eine DATA-Anweisung behandelt, weshalb vor den Systemeingabepuffer ein Komma als DATA-Trennmарkierung kommt. Die weitere Behandlung besteht darin, daß eine Eingabe in den Systemeingabepuffer geholt und als DATA-Datum ausgewertet wird, wofür die, bereits genannte, generelle INPUT/READ/GET-Routine verantwortlich zeichnet, da INPUT (\$abbf) in diese einsteigt.

Da eine INPUT-Anweisung also mit einer DATA-Zeile gleichzusetzen ist, erklärt sich, warum die Anweisung

```
INPUT Z;A;T%
```

auch mit einmaligem Drücken von <RETURN> ausreichend beantwortbar ist:

```
5.156,100,3
```

Das Komma wird also wie bei READ/DATA als Trennmарkierung akzeptiert. Der Nachteil ist aber auch, daß ein über INPUT eingegebener String kein Komma beinhalten darf – es sei denn, er wird in Anführungszeichen gestellt.

READ \$ac06: Routine zum Basic-Befehl READ

Die READ-Routine selbst besteht nur aus den Befehlen zum Auslesen der aktuellen DATA-Adresse. Dann wird lediglich das READ-Flag geladen und die allgemeine INPUT/GET/READ-Routine durchlaufen:

\$ac0f: allgemeine Routine für INPUT/ READ/GET

Zur Ausführung dieser universellen Routine müssen die folgenden Daten bereitstehen:

- Befehlsflag im Akku (\$00 = INPUT; \$40 = GET; \$98 = READ)
- Adresse der Eingabe in X- und Y-Register

Das Befehlsflag wird dann in \$11, die Adresse der Eingabe in \$43/\$44 gemerkt. Nach dem Retten des CHRGET-Zeigers wird dieser auf die Eingabe gerichtet, damit diese über CHRGET (\$0073) auswertbar ist. Die Auswertung selbst schließlich behandelt zunächst jeden Befehl (INPUT, READ oder GET) gesondert und schreitet dann zur allgemeinen Auswertung, wobei numerische Parameter ins Fließkommaformat umgewandelt werden.

Abbildung 4.20 ist hierzu ein sehr großes Flußdiagramm, das die Befehlsroutinen zu INPUT, INPUT#, READ und GET (inklusive GET#, was als syntaktischer Sonderfall von GET gilt), im Zusammenhang zeigt. Damit dürften diese etwas unübersichtlichen Programmstrukturen noch transparenter werden, als es das ROM-Listing auf sich alleine gestellt ermöglichen könnte.

NEXT \$ad1e: Routine zum Basic-Befehl NEXT

Die FOR-Routine läßt gewissermaßen viel Arbeit für NEXT zu tun übrig.

Zunächst liest NEXT den richtigen Stapelbeitrag vom Stapel ein (bei »NEXT« ohne Parameter den nächstbesten); wird keiner gefunden, erfolgt die Meldung NEXT WITHOUT FOR ERROR.

Dann wird die Schleifenvariable um die Schrittweite erhöht und mit dem Schleifenendwert verglichen; solange dieser nicht überschritten ist, wird die Schleife bei der im Stapelbeitrag angegebenen Adresse wiederholt, ansonsten wird die Schleife abgebrochen und der Stapelbeitrag gelöscht, da abgeschlossene Schleifen keinen Stapelspeicher mehr belegen dürfen. Beim Verlassen einer Schleife muß auch noch der Sonderfall »NEXT variable1, variable2, . . .« behandelt werden.

Abbildung 4.21 stellt die NEXT-Routine dar; alle Operationen mit Stapelzugriff sind dabei hervorgehoben.

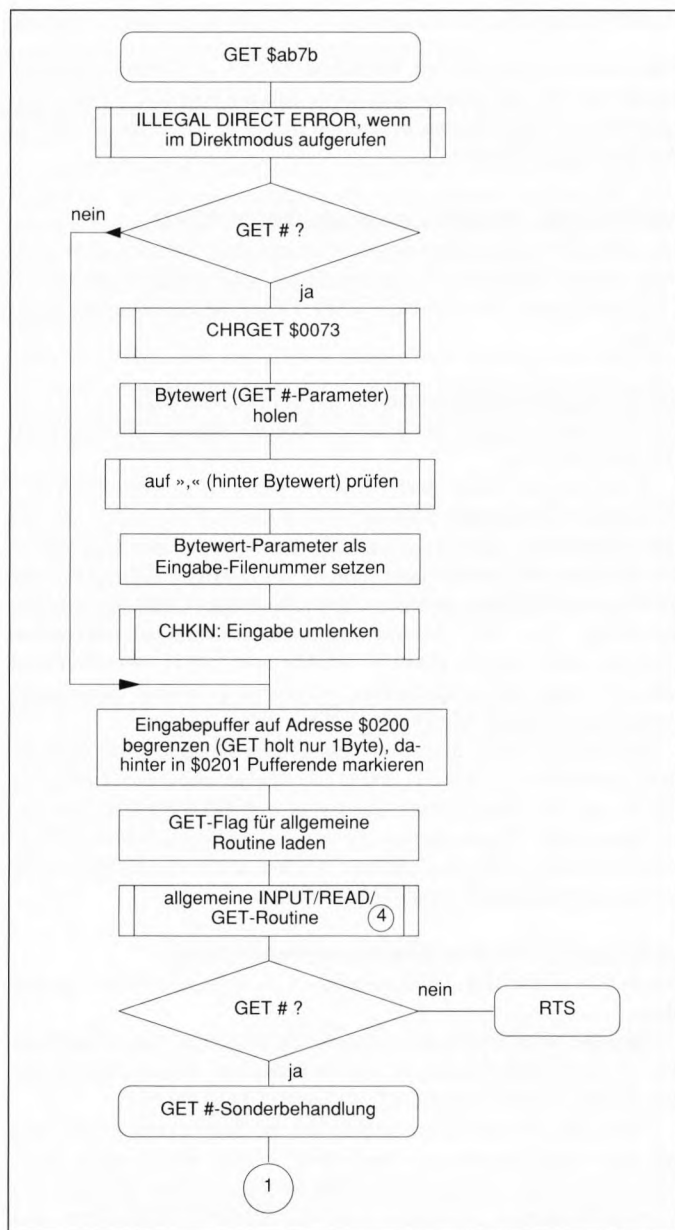


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 1)

FRMNUM (\$ad8a): numerischen Ausdruck auswerten

Einen numerischen Ausdruck, der ab der CHRGET-Zeiger-Position steht, wertet FRMNUM (\$ad8a) aus. Diese »Routine« besteht aus

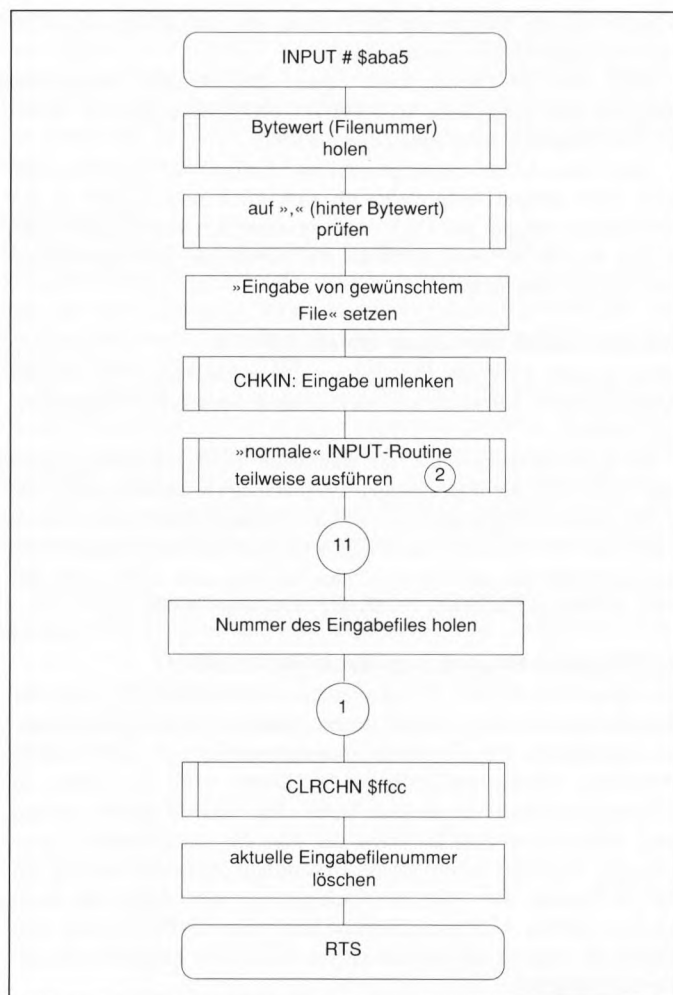


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 2)

dem Aufruf der Routine FRMEVL (\$ad9e), die beliebige Basic-Ausdrücke – also auch Strings – interpretiert; hinter diesem Routinenaufruf steht die CHKNUM-Routine im Speicher, die noch sicherstellt, daß es sich um einen numerischen Wert handelt (TYPE MISMATCH ERROR, wenn nicht).

CHKNUM (\$ad8d):**ausgewerteten Ausdruck auf »numerisch« prüfen**

Auch dies ist nur eine vorgeschaltete Routine. Sie lädt das Carry-Flag mit 0; da die CHKTYP-Routine auf einen BIT-Befehl zum Übergehen des CHKSTR-Einsprungs folgt, die den Datentyp des

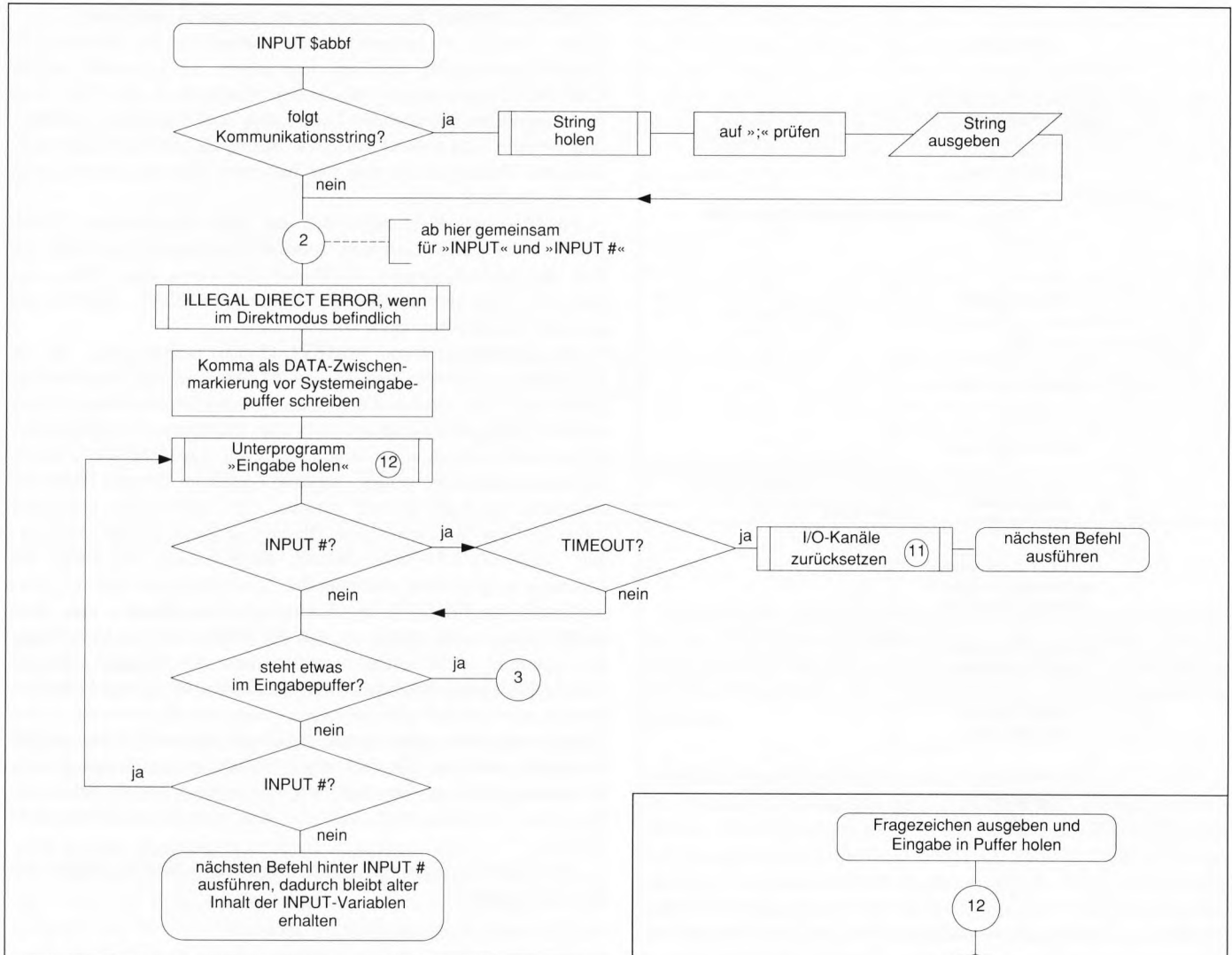


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 3)

letzten über FRMEVL (\$ad9e) eingeholten Ausdrucks auf das richtige Byte stellt, erfolgt somit eine Prüfung auf »numerisch«.

CHKSTR (\$ad8f):

ausgewerteten Ausdruck auf »String« prüfen

Dies ist das Gegenstück zu CHKNUM (\$ad8d). Es wird das Carry-Flag mit 1 geladen und die CHKTYP-Routine, die hier unmittelbar im Speicher folgt, ausgeführt.

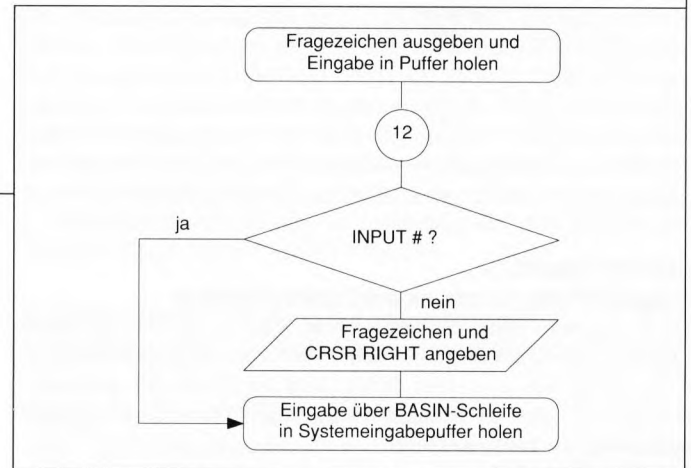


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 4)

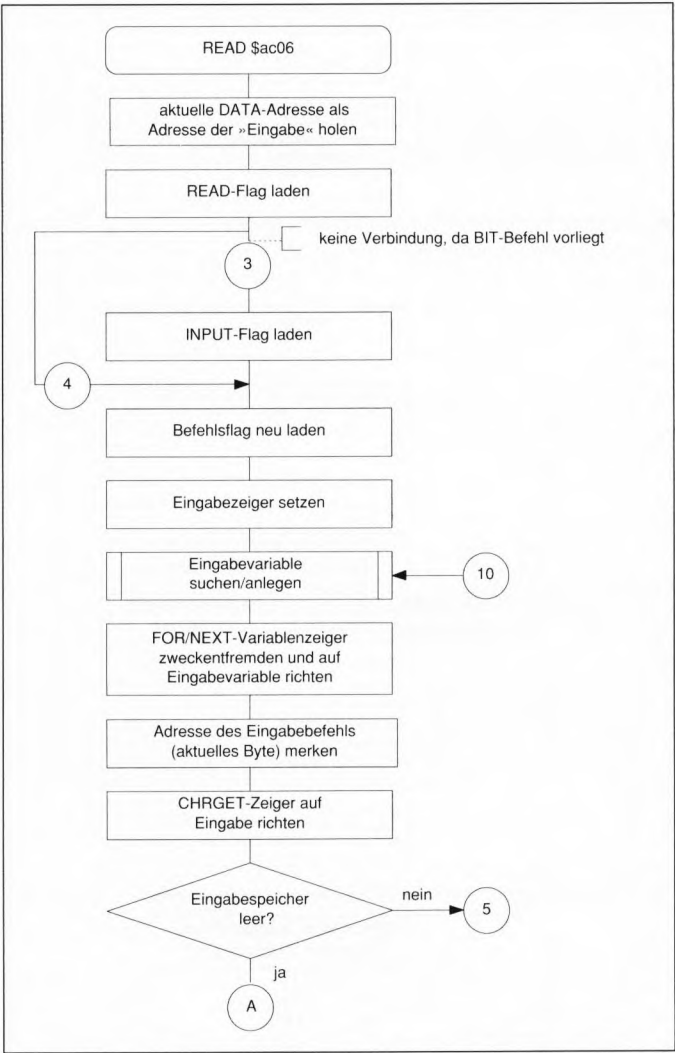


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 5)

**CHKTYP (\$ad90):
ausgewerteten Ausdruck auf Datentyp prüfen**

Diese Routine prüft bei gelöschtem Carry-Flag, ob der letzte FRMEVL-Ausdruck ein numerischer Wert war, bei gesetztem Carry-Flag, ob es sich um einen String handelte. Trifft der gefundene Datentyp zu, wird die Routine wieder über RTS verlassen, andernfalls erfolgt die Fehlermeldung
? TYPE MISMATCH ERROR
zur Beschreibung des falschen Datentyps.

FRMEVL (\$ad9e): Auswertung beliebiger Ausdrücke

Diese Routine ist bekanntlich die Grundlage für nahezu jede Parameterevaluation des Basic-Interpreters. Sie holt einen an der CHRGET-Zeigerposition befindlichen Ausdruck in den FAC oder den temporären Stringstapel. Dieser darf auch Variablen enthalten, und es kann sich sowohl um einen String als auch eine Zahl handeln; das Datentyp-Flag \$0d gibt hinterher Auskunft darüber, was für ein Ausdruck vorlag.

FRMEVL (\$ad9e) stützt sich auf den Programmteil EVAL (\$ae83), der einen einzelnen Ausdrucksbestandteil auswertet. Im Fall des Basic-Ausdrucks »(100+A)*3.5« wären also »100«, »A« und »3.5« die EVAL-Glieder, während FRMEVL (\$ad9e) den gesamten Ausdruck einholt.

Die Schwierigkeit von FRMEVL (\$ad9e) besteht darin, daß die Prioritäten zu beachten sind. Dazu wird zum einen die Prioritätsflag-tabelle und zum anderen die Klammersetzung berücksichtigt. Durch die Prioritäten wird allerdings eines klar: Die byteweise aufeinander-folgende Interpretation der Ausdrücke kann von FRMEVL nicht erfolgreich praktiziert werden. Deshalb wiederum arbeitet FRMEVL **rekursiv**, das heißt, es ruft teilweise sich selbst (oder zumindest einen Teil von sich selbst) auf. Bleiben wir beim Beispiel von vor-hin. »(100+A)*3.5« wird dadurch ausgerechnet, daß zuerst die Klammer ausgerechnet und dann mit 3.5 multipliziert wird. Um also innerhalb von FRMEVL die Klammer auszurechnen – nun, dazu bleibt nichts weiter übrig, als daß FRMEVL sich zur Berechnung der Klammer selbst aufruft, wodurch dann die Addition »100+A« durchgeführt wird. Nachdem dieses geschehen ist, springt FRMEVL in sich selbst zurück und multipliziert dann das Ergebnis mit »3.5«. Stünde statt »3.5« eine weitere Klammer da, müßte eine weitere Rekursion erfolgen. Da bei einer Rekursion am Stapel jeweils Rücksprungadressen abgelegt werden müssen, wird durch die begrenzte Stapelkapazität auch die Klammersverschachtelungstiefe limitiert.

Am folgenden Beispiel erklärt Abbildung 4.22 noch einmal das Rekursionsprinzip:

$$(1350-D) / (56 * (130 + \sin(47-E)))$$

D=1349, E=47 wird vorausgesetzt.

Natürlich ließe sich dieses Beispiel auch teilweise ausmultiplizieren, damit weniger Klammern vorliegen; in dieser Form aber ist die Rekursion noch stärker.

Im folgenden stelle ich Ihnen noch einzelne Teilroutinen von FRMEVL/EVAL vor, die einer Erklärung bedürfen.

FACSTK (\$ae33): FAC auf Stapel legen

Diese Routine legt nach ihrem Aufruf über »jsr facstk« den aktuel-len Inhalt des FAC auf den Stapel. Da die Rücksprungadresse von »jsr« ebenfalls am Stapel Platz beansprucht, wird sie unverzüglich

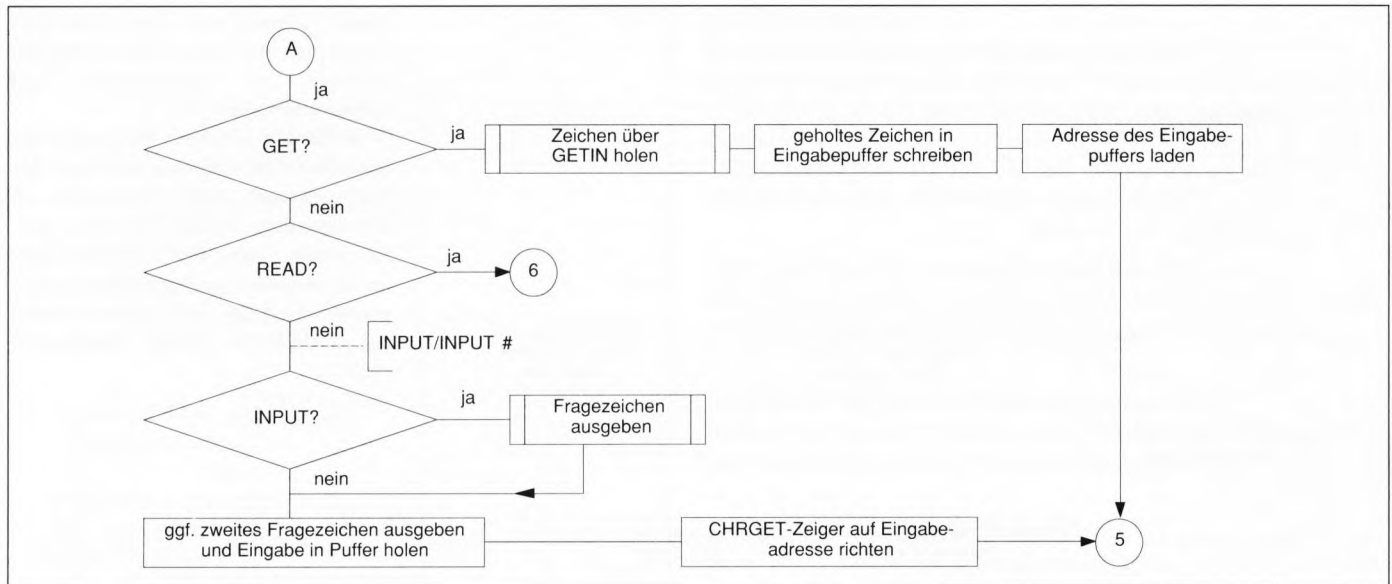


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 6)

von dort entfernt und die Adresse ersatzweise im Hilfszeiger \$22/\$23 gemerkt. Am Ende der Routine erfolgt über »jmp (\$0022)« der Rücksprung.

Vor dem FAC wird das Vorzeichen auf den Stapel gelegt.

EVAL (\$ae83): nächsten Ausdrucksbestandteil verwerten

Auf diesen FRMEVL-Teil, der den Vektor IEVAL \$030a/\$030b einsetzt, wurde bereits bei der FRMEVL-Besprechung eingegangen.

NOT \$aed4: Routine zum Basic-Operator NOT

Der Operator NOT (der Begriff »Funktion« wird zwar oft verwendet – auch im ROM-Listing –, ist aber nicht der treffendste) muß aufgrund des für diese Operation ungünstigen Fließkommaformates recht umständlich ausgeführt werden: Der FAC wird in eine 2-Byte-Zahl umgewandelt, dann in dieser Integerdarstellung mittels EOR-Verknüpfung invertiert und daraufhin schließlich wieder ins Fließkommaformat zurückgebracht.

BRCEVL (\$ae1f): beliebigen Ausdruck in Klammern auswerten

Durch »jsr brcevl« wird ein beliebiger Ausdruck (String oder Zahl) wie über FRMEVL (\$ad9e) behandelt, wobei zusätzlich die Syntaxprüfung erfolgt, ob er auch in Klammern eingefaßt ist. So werden beispielsweise Funktionsargumente ausgewertet, die bekanntlich in Klammern zu stehen haben.

Die BRCEVL-Routine besteht genaugenommen nur aus dem Aufruf der Prüfroutine CHKBRO (\$ae1a), die eine offene Klammer prüft, und dem FRMEVL-Einsatz; dahinter folgt im Speicher der CHKBCL-Einsprung zur Sicherstellung einer geschlossenen Klammer.

CHKBCL (\$ae17): Prüfung auf »Klammer zu«

Die Prüfung auf »« (geschlossene Klammer) erledigt nach »jsr chkbcl« diese Routine; steht an der aktuellen CHRGET-Position kein »«, gibt sie die Meldung SYNTAX ERROR aus, da eine syntaktische Vorschrift verletzt wurde, ansonsten erfolgt ein gewöhnlicher RTS-Rücksprung. Der CHRGET-Zeiger ist danach um dieses eine zu prüfende Byte weitergezählt – vorausgesetzt, das Prüfbyte konnte gefunden werden.

CHKBCL (\$ae17) ist nur der Ladebefehl des ASCII-Codes von »« als Prüfbyte für die CHKBYT-Routine.

CHKBRO (\$ae1a): Prüfung auf »Klammer auf«

Wie CHKBCL (\$ae17), aber für die geöffnete Klammer.

CHKCOM (\$ae1d): Prüfung auf Komma

Diese Prüfroutine ist statistisch gesehen die am häufigsten benötigte von seiten des Interpreters. Wie CHKBCL (\$ae17) und CHKBRO (\$ae1a), aber für das Komma »,«,.

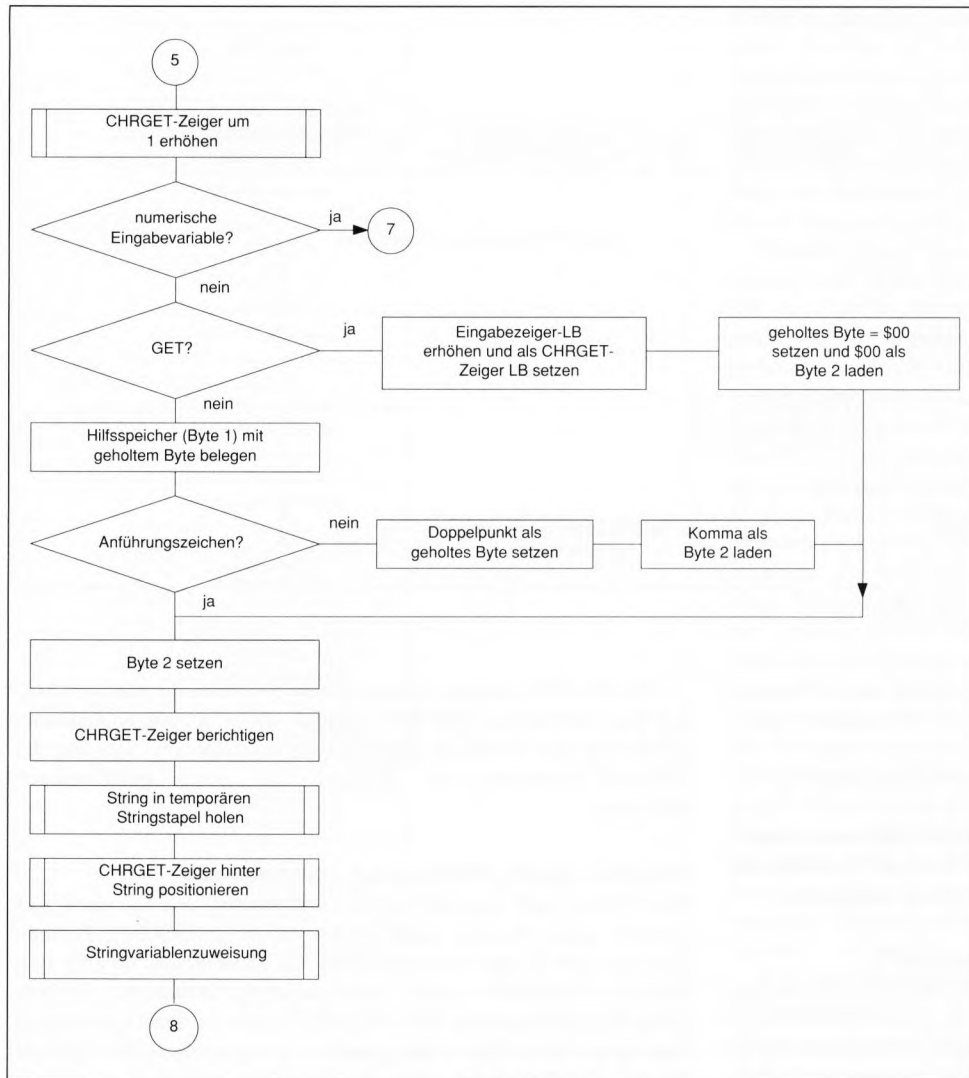


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 7)

CHKBYT (\$aeff):**Prüfung auf im Akkumulator enthaltenes Byte**

Diese Routine bildet letztlich die Grundlage für CHKBCL (\$aef7), CHKBRO (\$aefa) und CHKCOM (\$aefd). Sie führt einen Vergleich aus, der zunächst ohne CHRGET-Zeiger-Erhöhung abläuft. Dann wird bei Bedarf die Fehlermeldung SYNTAX ERROR über den Einsprung SYNERR (\$af08) ausgelöst; andernfalls erhöht CHKBYT

(\$aeff) schnell noch den CHRGET-Zeiger, um ihn hinter dem ermittelten Prüfbyte zu positionieren und zurückzuspringen.

Wenn Sie in einem Programm aus irgendwelchen Gründen zwar auf das Vorhandensein eines Bytewertes an der aktuellen CHRGET-Position prüfen wollen, ohne den CHRGET-Zähler zu verändern, genügen also die folgenden Befehle, die gewissermaßen aus CHKBYT (\$aeff) entnommen sind:

```

100 - lda #prüfbyte ;
      Vergleichswert
      laden
110 - ldy #0 ; Offset 0
120 - cmp ($7a),y ; Ver-
      gleich über CHRGET-
      Zeiger
130 - beq ja ;
      z=1: Übereinstimmung
      z=0: keine Überein-
      stimmung
  
```

Dadurch sind Sie also nicht auf die einseitige Behandlung »SYNTAX ERROR oder NICHT SYNTAX ERROR« angewiesen, sondern haben eigene Alternativen zur Verfügung.

SYNERR (\$af08):**Einsprung für die Fehlermeldung SYNTAX ERROR**

Die Anweisung »jmp synerr« löst einen SYNTAX ERROR aus. In Basic können Sie dies über den SYS-Einsprung bei \$af08 (#44808) zumindest demonstrieren bekommen:

```
SYS 44808
```

```
?SYNTAX ERROR
READY.
```

\$af14: Hilfsroutine zur Prüfung auf »Variable im ROM«

Wenn bestimmte Spezialvariablen auftreten, wird für diese eine Adresse im ROM-Bereich angegeben. Diese Routine stellt also

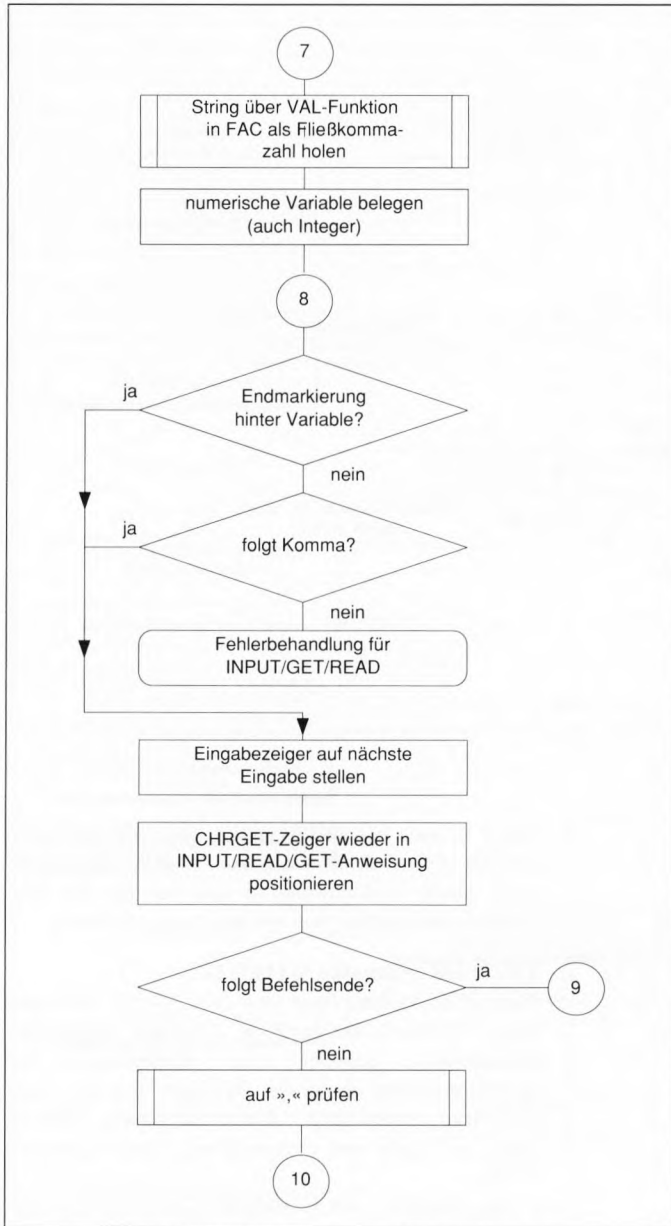


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 8)

effektiv fest, ob die aktuelle Variable (Adresse in \$64/\$65) im Bereich \$a000–\$e3a2 liegt und somit eine Sonderbehandlung erfordert (C=0) oder nicht (C=1).

GETVAR (\$af28):**Im Basic-Text stehende Variable auswerten**

Dieser EVAL-Bestandteil sucht eine Variable, deren Name ab der CHRGET-Zeiger-Position im Basic-Text steht, und ermittelt deren Inhalt.

Sonderfälle wie TI, TI\$ und ST werden ebenso berücksichtigt wie alle Datentypen (Fließkomma, Integer, String).

OR \$afe6: Routine zur Basic-Operation OR

Diese Routine besteht nur aus dem Laden des OR-Flags \$ff und der anschließenden Ausführung der allgemeinen AND/OR-Routine.

AND \$afe9: Routine zur Basic-Operation AND

Auch diese Routine beschränkt sich darauf, das AND-Flag \$00 zu laden und die allgemeine AND/OR-Behandlung auszulösen.

\$afeb: allgemeine AND/OR-Routine

Hier wird zunächst das AND/OR-Flag im Hilfsspeicher \$0b gesichert. Es handelt sich um eine Bitmaske je nach Operator: OR = %11111111 (alle Bits gesetzt), AND = %00000000 (alle Bits gelöscht). Daraufhin wird der erste Parameter ins 2-Byte-Integerformat konvertiert.

Von da an läuft die Behandlung für AND und OR parallel, es wird aber regelmäßig eine Verknüpfung mit der Bitmaske durchgeführt. Betrachten wir deshalb die einzelnen Fälle getrennt, und beginnen wir dabei mit dem leichteren:

Fall 1: AND (Bitmaske %00000000)

Da eine EOR-Verknüpfung mit 0 den verknüpften Wert nicht im geringsten ändert, wird also bei \$aff0–\$affa der erste AND-Parameter unverändert nach \$07/\$08 kopiert. Dadurch entsteht Platz für den zweiten Parameter bei \$64/\$65.

Streicht man nun die entfallenen EOR-Operationen mit %000000000, könnte man folgenden Quelltext für die Befehle bei \$b002–\$b012 schreiben, der alle tatsächlich wirksamen Anweisungen enthält, nach denen in Akku und Y-Register die Lösung steht, welche bei \$b013 in den FAC als Ergebnis kommt:

```

100 - lda highbyteparameter2 ; AND-Verknüpfung
110 - and highbyteparameter1 ; der High-Bytes
120 - tay                    ; Ergebnis als High-
                             ; Byte nach Y
130 - lda lowbyteparameter2 ; AND-Verknüpfung
140 - and lowbyteparameter1 ; der Low-Bytes
150 - ;
160 - ; Ergebnis steht hier in Akku (Low-Byte) und
      ; Y-Register (High-Byte)
  
```

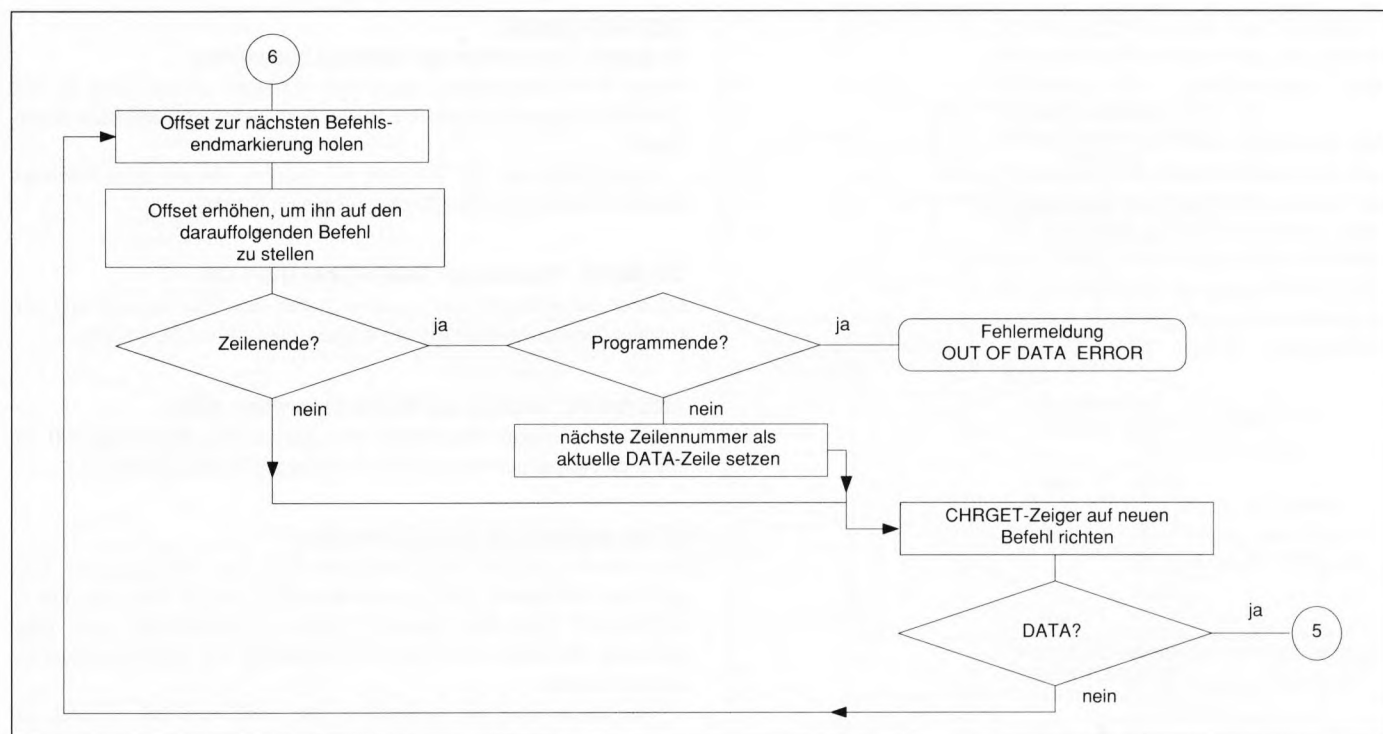
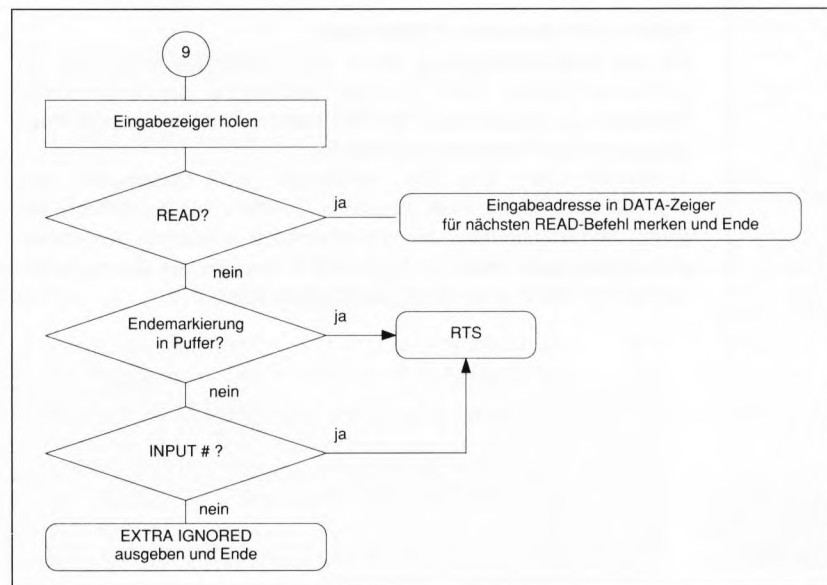


Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 9)



Die 9 Befehle bei \$b002–\$b012 reduzieren sich also auf die obigen 5, da viermal »eor \$0b« eingespart wird. Diese Verknüpfung ist also nur für die OR-Routine erforderlich, wie wir gleich sehen werden.

Fall 2: OR (Bitmaske %11111111)

Eine EOR-Verknüpfung mit %11111111 invertiert einen Bytewert, das heißt, es wird das sogenannte Komplement gebildet. Das Komplement ist definitionsgemäß derjenige Bytewert, der mit dem nicht-komplementierten Wert zusammen addiert genau \$ff ergibt und entspricht der Basic-Operation NOT.

Im OR-Fall wird also der erste Parameter zunächst komplementiert in \$07/\$08 gemerkt. Nach Auswertung des zweiten Parameters wird dann jeweils das High- oder Low-Byte des zweiten

Abbildung 4.20: INPUT/READ/GET im Überblick (Teil 10)

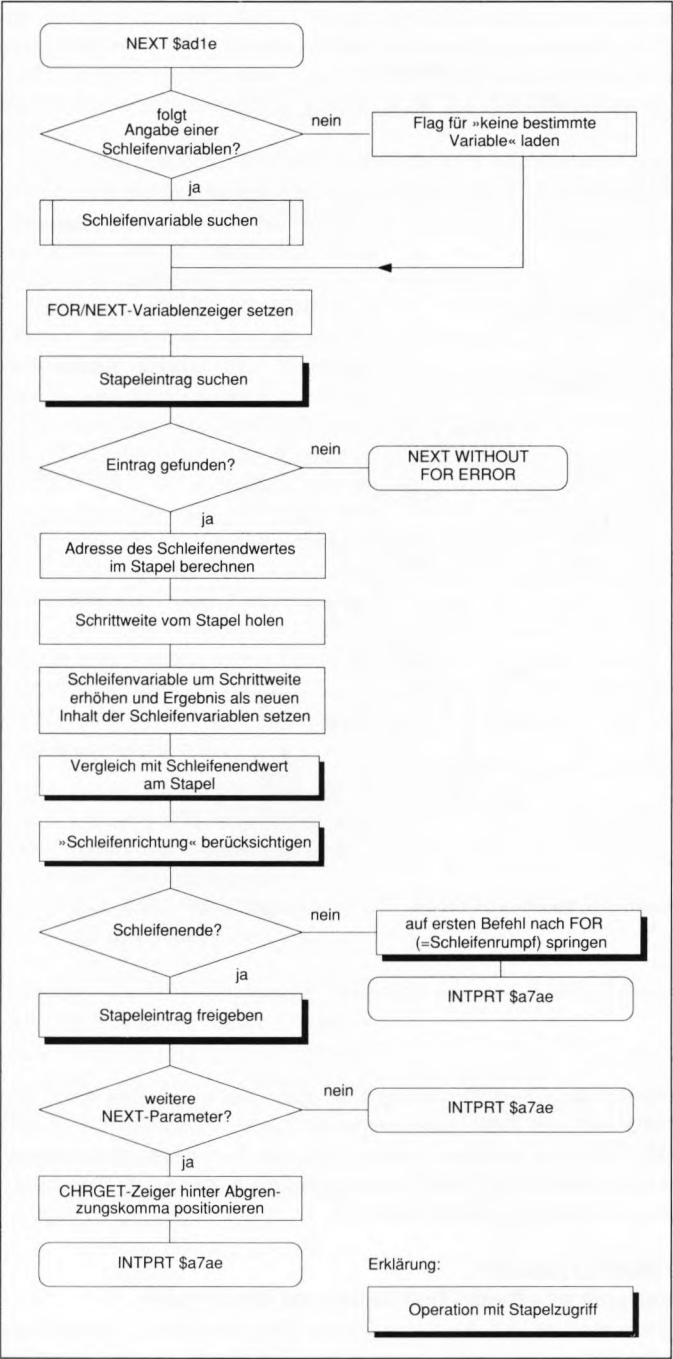


Abbildung 4.21: Die NEXT-Routine

Parameters komplementiert, mit dem bereits invertierten Byte des ersten Parameters UND-verknüpft und dieses Ergebnis wiederum komplementiert, bis schließlich das gesamte 2-Byte-Ergebnis in den FAC kommt. Die Komplementierung kann man, wie schon erwähnt, auch als NOT-Verknüpfung ansehen. Dann wurden der erste (para1) und zweite (para2) Parameter folgendermaßen miteinander verknüpft, um den Wert »para1 OR para2« zu erhalten:

```
(NOT para2) AND (NOT para1)
```

Um diese komplizierte Arbeitsweise, die sich auf logische Prinzipien stützt, möglichst praxisnah erklären zu können, weiche ich deshalb auf Wahrheitswerte aus. Mit AND, OR und NOT werden bekanntlich auch die Wahrheitswerte (»wahr« = 1; »falsch« = 0) von Aussagen verknüpft. Damit ist gedanklich viel leichter umzugehen als mit Zahlenwerten bei diesen sogenannten Booleschen Operationen.

Dann ist also die OR-Verknüpfung dann erfüllt (Wahrheitswert 1), wenn entweder »para1« oder »para2« oder beide zutreffen. Halten wir diese Fälle fest:

- 1. para1 = 1; para2 = 0 (eines, nämlich »para1«, trifft zu)
- 2. para1 = 0; para2 = 1 (eines, nämlich »para2«, trifft zu)
- 3. para1 = 1; para2 = 1 (beides trifft zu)

Bleibt also nur noch ein vierter Fall übrig, in welchem die OR-Operation nicht erfüllt ist (Wahrheitswert 0):

- 4. para1 = 0; para2 = 0 (beide unzutreffend)

Anstatt zu sagen: »Beide treffen nicht zu« ist folgendes unter logischen Gesichtspunkten genauso richtig:

Weder »para1« noch »para2« trifft zu.

Und das darf man auch folgendermaßen formulieren:

»para1« trifft nicht zu und gleichzeitig trifft »para2« nicht zu.

Und damit sind wir schon bei der mathematischen Schreibweise, nach der die ROM-Routine vorgeht:

```
( NOT para1 ) AND ( NOT para2 )
```

Verstanden? Herzlichen Glückwunsch, es war nicht ganz leicht, aber wenn Sie denselben logischen Gedankengang nachvollzogen haben, so ist es Ihnen gelungen, die AND/OR-Routine endgültig zu durchschauen, und Sie wissen jetzt, wie es dem Interpreter gelingt, den ORA-Befehl zu umgehen, da dieser eine Fallunterscheidung bzw. eigene Routinen für AND und OR erforderlich gemacht hätte. Falls Sie den Faden verloren haben sollten, gibt es prinzipiell zwei Möglichkeiten; entweder Sie unternehmen einen erneuten Leseanlauf (was der Erfahrung nach, die ich als Autor gewinnen konnte, den meisten Lesern hilft, auch wenn ich es selbst nicht für möglich

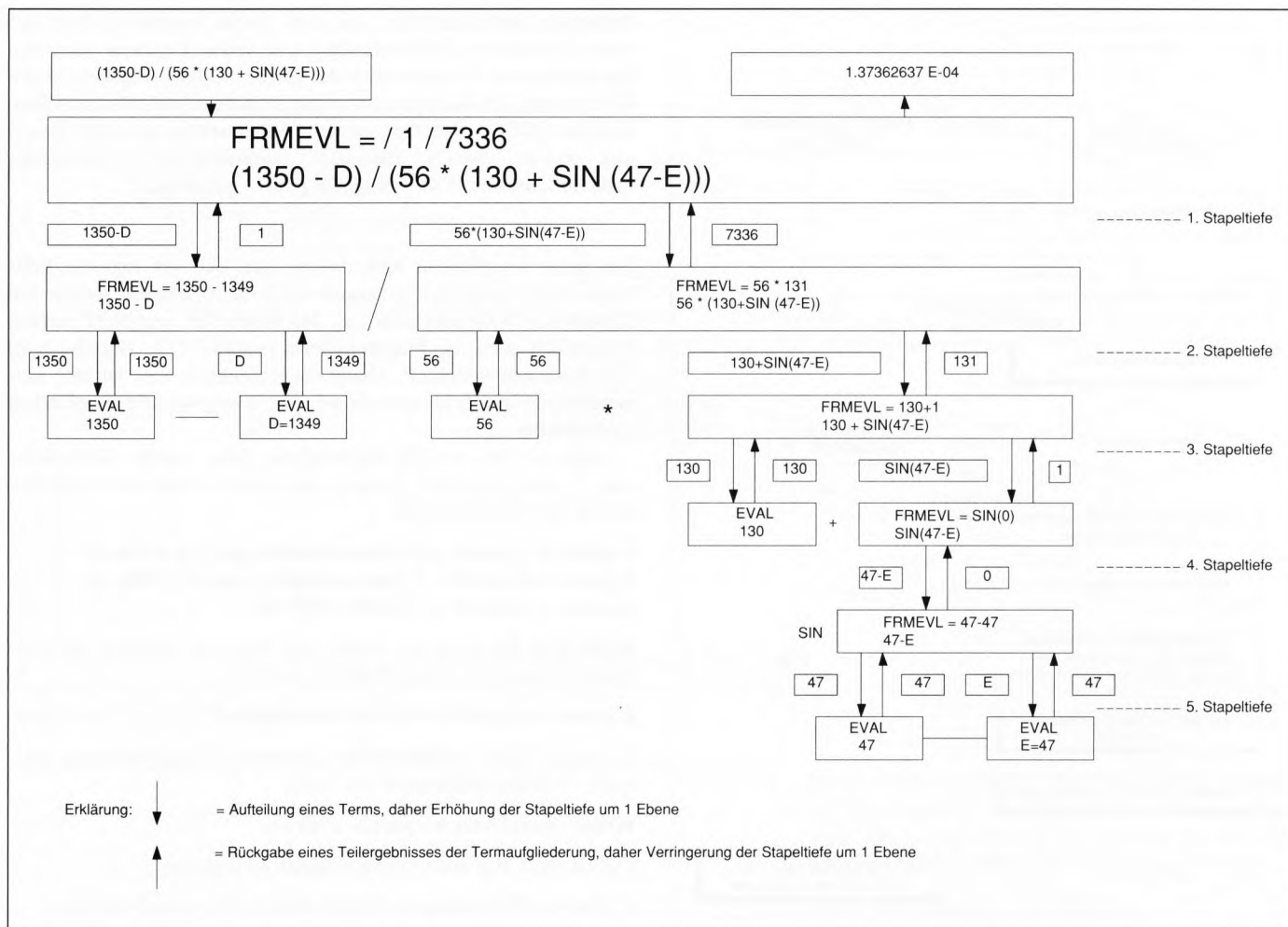


Abbildung 4.22: Beispiel für eine Rekursion in FRMEVL

gehalten habe!) oder sie sind mit der Tatsache, **daß** die Routine funktioniert, zu Recht zufrieden, und haben zumindest einen guten Eindruck bekommen, **wie** sie abläuft.

DIM \$b081: Routine zum Basic-Befehl DIM

Der Befehl DIM wird in einer Schleife abgearbeitet, die im Speicher schon bei \$b07e, also vor dem Einsprung \$b081, beginnt. Diese Schleife holt das erste Zeichen des DIM-Parameters, der ja ein Variablenname für das zu dimensionierende Array ist, und setzt das X-Register dadurch auf einen anderen Wert als \$00, was bei der aufgerufenen Routine (\$b090: späterer Einsprung in FNDVAR

\$b08b) als Dimensionierungsflag gilt. Nach der innerhalb von FNDVAR (bei \$b090 eingestiegen) erfolgten Dimensionierung prüft DIM, ob noch weitere Zeichen nach dem Variablennamen und der Array-Größe folgen; falls ja, so muß es ein Komma zur Abgrenzung zum nächsten Variablennamen sein.

FNDVAR (\$b08b):

Variable aus Basic-Text holen und Inhalt holen

Diese Routine zur Auswertung einer Basic-Variablen, deren Name (und Index, im Falle einer Array-Variablen) ab der aktuellen CHRGET-Zeiger-Position steht, hat auch einen Einsprung bei \$b090

für die DIM-Routine (\$b081). Dieser zweite Einsprung erwartet im Akku das erste Byte des Variablennamen und im X-Register einen anderen Wert als \$00, also ein Dimensionierungsflag. Ist nämlich dieses Dimensionierungsflag gesetzt, wird das Variablenarray angelegt.

Soll über FNDVAR (\$b08b) eine noch nicht verwendete Variable gesucht werden, so legt FNDVAR (\$b08b) diese automatisch an. Der Ausgangswert einer numerischen Variablen ist 0, derjenige eines Strings ein Leerstring " ".

Der erste Teil von FNDVAR (\$b08b) beschäftigt sich ausschließlich damit, anhand des Variablennamens alle Informationen zu bekommen. Auch die Syntaxprüfung erfolgt hier: Das erste Zeichen im Variablennamen muß ein Buchstabe, das zweite ein Buchstabe oder auch eine Ziffer sein. Ist der Variablenname nur 1 Zeichen lang, wird das zweite Byte auf \$00 gesetzt. Die Datentyp-Flags \$0d und \$0e werden schließlich gemäß dem Variablentyp (»%« = Integer, »\$« = String) gesetzt.

Erst bei \$b0dd sind alle Hilfsspeicher richtig gestellt:

- \$0d = Datentyp (String/numerisch)
- \$0e = Datentyp (Fließkomma/integer)
- \$45 = Byte 1 des Variablennamens (b7 gibt Datentyp an)
- \$46 = Byte 2 des Variablennamens (b7 gibt Datentyp an)

Abbildung 4.23 verdeutlicht, wie diese Hilfsspeicher gesetzt werden. Dabei werden folgende Labelnamen verwendet, die nur in diesem Flußdiagramm gültig sind:

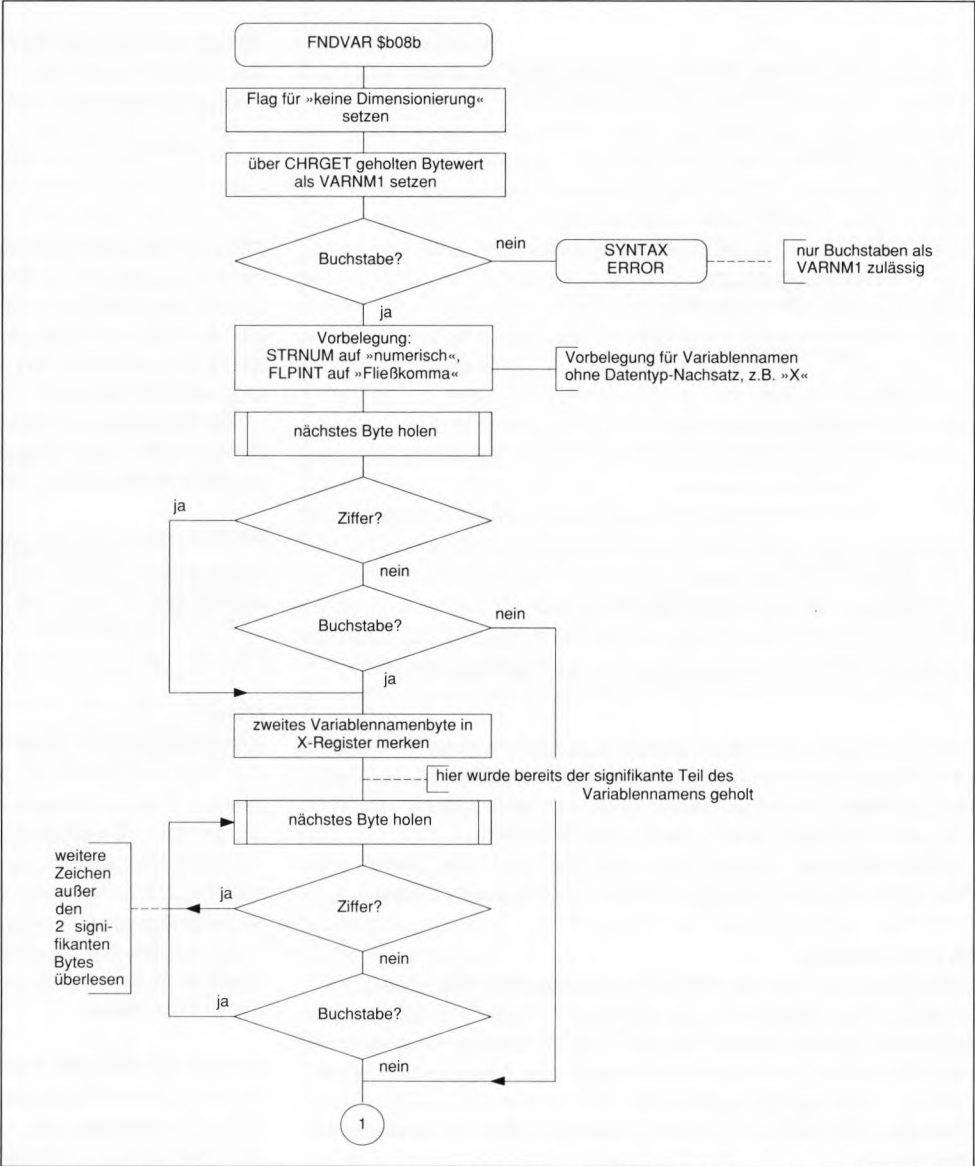
- STRNUM = \$0d :
flag/ string or numerical
- FLPINT = \$0e :
flag/ floating point or integer

Abbildung 4.23: Variablennamenauswertung in FNDVAR (Teil 1)

- VARNM1 = \$45 : variable name, byte 1
- VARNM2 = \$46 : variable name, byte 2

**CHKLTR (\$b113):
Prüfung auf Buchstabencode im Akkumulator**

Diese Hilfsroutine dient in der aufwendigen FNDVAR-Syntaxprüfung für den Variablennamen als Unteroutine. Nach ihrem



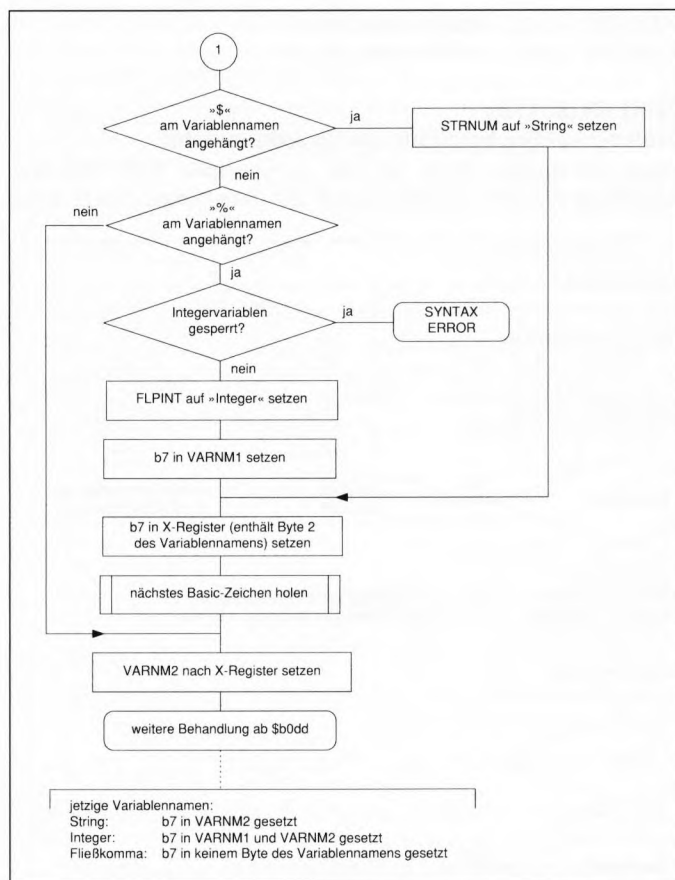


Abbildung 4.23: Variablennamenauswertung in FNDVAR (Teil 2)

Aufruf ist das Carry-Flag gesetzt, falls es sich um einen Buchstaben (ASCII-Code im Bereich \$41–\$5a) handelt, ansonsten ist das Carry-Flag gelöscht. Der Akku enthält nach »jsr chklr« denselben Wert wie vorher, andere Register werden nicht beeinflusst.

Diese Routine eignet sich vielleicht auch für eines Ihrer Programme als Ergänzung der CHRGET-Prüfung auf Ziffern.

\$b11d: Routine

zum Anlegen einer noch nicht verwendeten Variablen

Um einen Variableneintrag neu anzulegen, wird diese Routine von FNDVAR (\$b08b) ausgelöst. Der Variablenname muß dazu in \$45/\$46 stehen; die Variable bekommt den Ausgangswert (0 bei Zahlen, " " bei Strings) zugewiesen.

Es darf sich nicht um ein Array oder ein Element aus einem solchen handeln.

FIRARY (\$b194): Ermittlung der Adresse des ersten Eintrags im aktuellen Array

Zu einem Array, dessen Dimensionenzahl (1–3) in \$0b steht, wird unter der Voraussetzung, daß der Hilfszeiger \$5f/\$60 auf den Array-Anfang weist, die Adresse des ersten Eintrags dieses Arrays errechnet und in \$58/\$59 sowie dem Registerpaar A/Y zurückgegeben. Akkumulator und Y-Register werden folglich beeinflusst, das X-Register bleibt jedoch unverändert.

\$b1a5: Konstante –32768

Bei \$b1a5 liegt die MFLPT-Darstellung von –32768, der niedrigsten zulässigen Integerzahl, deren binäre Darstellung

```
%100000000 00000000
```

wäre.

\$b1aa: FAC #1 in Integerzahl umwandeln

Dieser Einsprung wird von keiner anderen ROM-Routine genutzt, ist jedoch bewußt von Commodore dem Programmierer als Arbeits-erleichterung zur Verfügung gestellt worden. Nach »jsr \$b1aa« steht im Akku (Low-Byte) und Y-Register (High-Byte) die Integerdarstellung des FAC-Inhaltes.

Die Nachkommastellen der ursprünglichen Fließkommazahl werden gänzlich vernachlässigt, es ist nur der ganzzahlige Anteil (INT-Anteil) ohne Rundung (!) relevant.

INTEVL (\$b1b2): Integerzahl aus Basic-Text auswerten

Integerzahlen werden durch »jsr intevl« sowohl aus dem Basic-Text ausgewertet als auch auf den zulässigen Bereich (von –32768 bis +32767) geprüft (bei Überschreitung: ILLEGAL QUANTITY ERROR) und ins Integerformat nach \$64 und \$65 gebracht.

\$b1d1:

Arrayvariablen-Sonderbehandlung für FNDVAR (\$b08b)

Um eine Arrayvariable auszuwerten, anzulegen oder sogar das gesamte Array zu dimensionieren (sofern das Dimensionierungsflag \$0c gesetzt ist), wird diese Routine aufgerufen.

FNDVAR (\$b08b) erkennt eine Arrayvariable übrigens daran, daß eine offene Klammer (ASCII-Code \$28) auf das letzte Byte des Variablennamens beziehungsweise den Datentypzusatz folgt.

Dieser Sonderbehandlungsteil erstreckt sich bis \$b37c. Darin versteckt sich bei \$b357 ein interessanter Einsprung für eigene Programme:

UMULT (\$b357): Multiplikation zweier 2-Byte-Werte

Um zwei 2-Byte-Werte zu multiplizieren, sind sie in \$28/\$29 sowie \$71/\$72 abzulegen und »jsr umult« auszuführen; danach steht in X/Y das Produkt der beiden Werte. Eine Überschreitung des Zahlen-

bereiches von 2 Bytes durch das Ergebnis führt zur Meldung OUT OF MEMORY ERROR.

Abbildung 4.24 stellt den Ablauf von UMULT an einem Beispiel (321 mal 65) dar.

FRE \$b37d: Routine zur Basic-Funktion FRE

Die FRE-Funktion erwartet einen beliebigen Parameter (String oder Zahl) als »Dummy«, das heißt als Parameter, der keinen Einfluß auf das Funktionsergebnis hat.

Es wird zunächst die Garbage Collection durchgeführt, um die optimale Speichernutzung herzustellen und dadurch tatsächlich die freie Speichermenge anzugeben. Die freie Speichermenge wird durch die Zeiger \$33/\$34 (unterste Adresse des Stringinhaltspeichers) und \$31/\$32 (oberste Adresse des Array-Bereiches, erhöht um 1) eingegrenzt, die also nur voneinander subtrahiert werden müssen. Die Differenz ist die freie Speichermenge und wird ins Fließkommaformat umgewandelt, wobei jedoch auch negative Zahlen auftreten können. Zu diesen muß man 65536 (%1 0000000 00000000) addieren, um ein korrekterweise positives Ergebnis zu bekommen. Der logische Fehler liegt darin, daß die Differenz ein vorzeichenloser 2-Byte-Wert ist. Solange dieser nicht den Wert 32767 (%01111111 11111111) übersteigt, ist das Vorzeichenbit also gelöscht und die Wandlung ins Fließkommaformat erfolgt fehlerfrei. Ansonsten wird das oberste Bit (b15) als negatives Vorzeichen und die anderen Bits (b0–b14) als Komplement des Betrags angesehen.

INTFAC (\$b391): Integerzahl aus Y/A in FAC als Fließkommazahl übertragen

Dieser Bestandteil der FRE-Routine wandelt eine vorzeichenbehaftete 2-Byte-Integerzahl in eine Fließkommazahl um, die im FAC abgelegt wird.

Er wird auch von Routinen zu anderen Funktionen und Operatoren sowie GETVAR (\$af6b) beim Auslesen einer Integervariablen eingesetzt; bei FRE (\$b37d) ist er bekanntermaßen etwas fehl am Platz (siehe vorausgehende Beschreibung), aber die anderen Anwendungsfälle meistert er souverän.

POS \$b39e: Routine zur Basic-Funktion POS

POS (\$b39e) holt zunächst über den Kernaleinsprung PLOT (\$fff0) die Cursorposition nach X/Y, wobei nur die Spaltennummer interessiert, die im Y-Register schon als Bytewert steht und deshalb praktischerweise direkt an BYTFAC (\$b3a2) übergeben werden kann. Darauf folgt im Speicher der BYTFAC-Einsprung als Teil der POS-Routine:

BYTFAC (\$b3a2): vorzeichenlosen Bytewert aus Y-Register in FAC übertragen

BYTFAC (\$b3a2) ist keine eigene Umwandlungsroutine, sondern löscht lediglich den Akku, der das High-Byte der sogleich an

INTFAC (\$b391) übergebenen 2-Byte-Integerzahl darstellt. Da somit auch das oberste Bit (b15) gelöscht ist, handelt es sich automatisch um eine positive Zahl.

Die Routinen zu den Funktionen LEN, ASC und PEEK, welche allesamt Bytewerte ermitteln und diese im allgemeinen Fließkommaformat zurückzugeben haben, stützen sich auf den BYTFAC-Einsprung ebenso wie POS (\$b39e), in dessen Routine BYTFAC (\$b3a2) gewissermaßen die Schlußbehandlung darstellt.

CHKDIR (\$b3a6):

Ausgabe von ILLEGAL DIRECT ERROR im Direktmodus

Die Befehle INPUT/INPUT#, GET/GET# und DEF dürfen, wie Sie von Basic-Programmierzzeiten her wissen, nur innerhalb von Programmen vorkommen. Der Grund hierfür ist, daß die Eingabebefehle ansonsten den Systemeingabepuffer verwenden, in welchem bereits die Direkteingabe steht, und somit eine ordnungsgemäße Interpretation ausgeschlossen ist. Bei DEF liegt der Grund darin, daß dazu ein fester Programmzeilenspeicher erforderlich ist, denn die Funktionsdefinition muß dort jederzeit verfügbar sein, wogegen eine Direkteingabe schon nach ihrer Abarbeitung verlorengeht.

Es handelt sich also keinesfalls um »böswillige Einschränkungen«, wie es in mancher Anfänger-Literatur dargestellt wird; Sie als Insider können sich jetzt diese verständliche Restriktion erklären.

DEF (\$b3b3): Routine zum Basic-Befehl DEF

Die Definition einer FN-Funktion geschieht, indem zunächst einmal die FN-Definitionssyntax überprüft und der Funktionsname eingeholt wird, wobei Intervariablen gesperrt sind und auch Strings verhindert werden. Die Funktionsvariable (in Klammern hinter dem Funktionsnamen angegeben) wird angelegt; vor der Funktionsdefinition selbst ist noch ein Zuweisungszeichen »=« ein syntaktisches Erfordernis.

Die Adresse der Funktionsvariablen wird schließlich auf den Stapel gelegt; dasselbe geschieht mit dem aktuellen CHRGET-Zeiger, der auf das Byte hinter »=« weist und somit die Adresse der Funktionsdefinition innerhalb dieser Definitionszeile angibt. Dadurch erklärt sich auch die zu Beginn der DEF-Routine durchgeführte Prüfung auf den Programm-Modus, denn die Definitionszeile muß aus Gründen der Datensicherheit im Programmspeicher liegen.

Dann wird der Rest des Definitionsbefehls überlesen und noch der Schlußteil der FN-Behandlung angesprochen, der die richtigen Werte vom Stapel holt und in die entsprechenden Speicherbereiche überträgt. Nach Ablauf dieses FN-Behandlungsteils wird die Ausführung also hinter dem DEF-Befehl fortgesetzt.

Abbildung 4.25 faßt die Funktionsweise von DEF/FN zusammen und ist hinter der FN-Beschreibung zu finden.

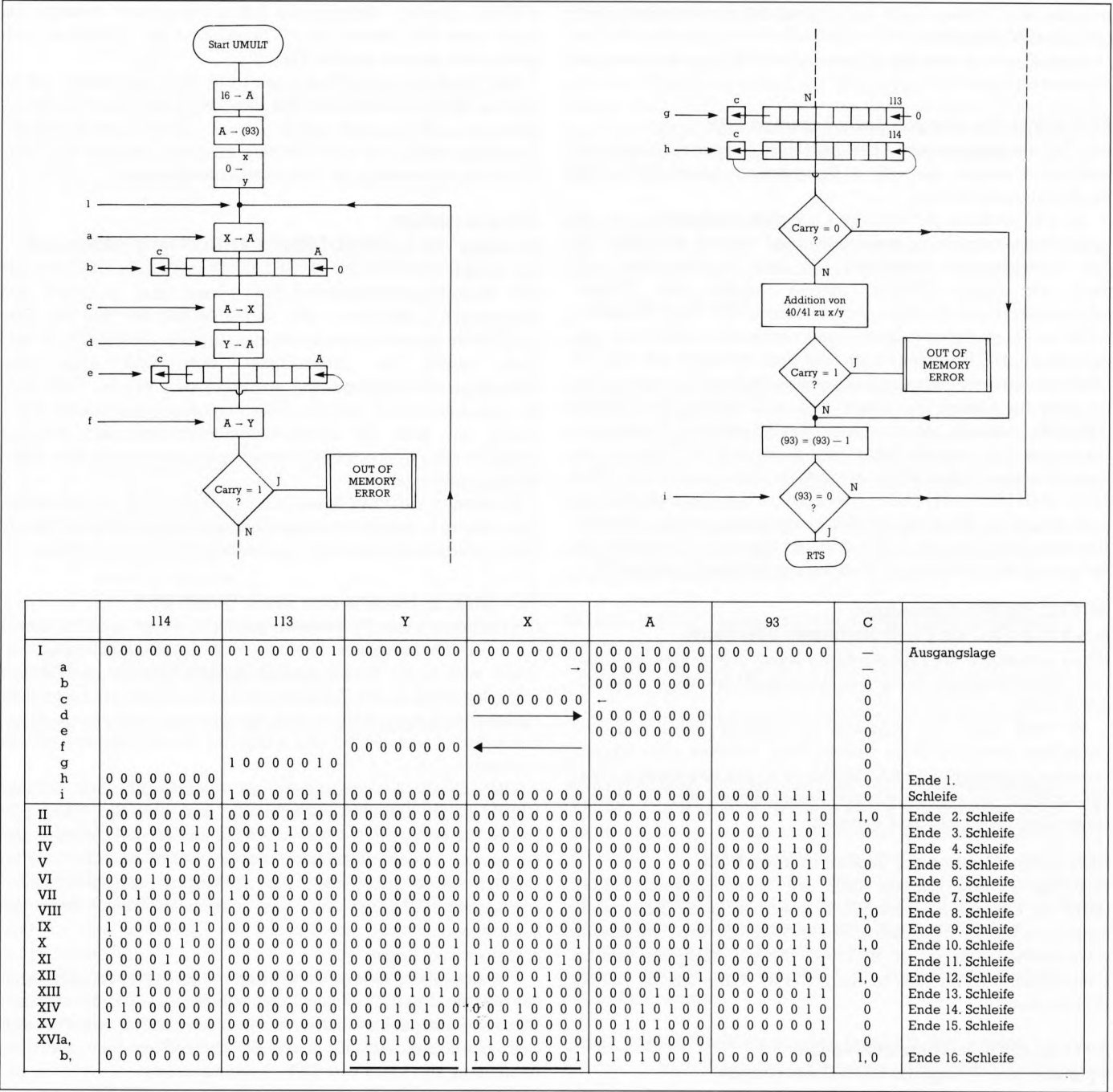


Abbildung 4.24: UMULT

CHKFNS (\$b3a1):**Syntaxprüfung für FN-Funktionsdefinition**

Diese Routine wird sowohl bei der Definition selbst als auch bei einer Funktionsverwendung selbst aufgerufen. Zunächst verlangt sie die Angabe des Schlüsselwortes FN, das als Token vorliegen muß. Dann wird die Funktionsvariable gesucht oder neu angelegt, wobei ausschließlich Fließkommavariablen zulässig sind. Die Adresse der FN-Variablen wird in \$4e/\$4f gemerkt.

In Abbildung 4.25 ist eine Dokumentation dieser Routine aufgrund ihrer Kürze nicht enthalten; sie tritt darin nur als nicht weiter erläutertes Unterprogramm »FN-Syntaxprüfung« auf.

FN \$b3f4: Routine zur Basic-Funktion FN

Zunächst erfolgt eine Syntaxprüfung, wobei die Adresse der FN-Variablen auf den Stapel gelegt wird. Anschließend holt FN (\$b3f4) einen in Klammern stehenden numerischen Ausdruck – das Funktionsargument – ein, dessen Wert schließlich in den Variablenspeicher der FN-Variablen übertragen wird. Zudem wird der Variablenadreßzeiger \$47/\$48 auf die FN-Variable gerichtet sowie der bisherige Inhalt auf den Stapel gerettet, denn das FN-Argument soll ja nur für die Dauer der FN-Funktion in der Funktionsvariablen enthalten sein.

Des weiteren kommt der CHRGET-Zeigerinhalt auf den Stapel, da für die Interpretation die Funktionsdefinition als herkömmlicher Basic-Ausdruck ausgewertet wird, wozu anschließend die CHRGET-Zeiger auf die FN-Definition zu richten sind.

Bei \$b438 läuft FRMNUM (\$ad8a) also durch die Funktionsdefinition; da die Funktionsvariable den Wert des Argumentes enthält, wird die Definition für die angegebene Zahl ausgewertet, und das Ergebnis kommt in den FAC – oder es wird eine Fehlermeldung ausgelöst, wenn die Funktionsdefinition fehlerhaft oder für das übergebene Argument nicht berechenbar ist.

Die Adresse der FN-Variablen wird unmittelbar nach dem FRMNUM-Aufruf wieder in den Hilfsspeicher \$4e/\$4f geschrieben. Es folgt eine Syntaxprüfung: Für den Fall, daß das letzte Zeichen hinter der Funktionsdefinition keine Endmarkierung eines Basic-Befehls oder einer Zeile war, tritt ein SYNTAX ERROR auf.

Ansonsten wird noch der CHRGET-Zeiger für diejenige Programmstelle, an der die definierte FN-Funktion angewendet wurde, wiederhergestellt. Der weitere Teil kommt auch bei DEF (\$b3b3) zur Ausführung: Er holt den Inhalt der FN-Namensvariablen bitweise vom Stapel und überträgt ihn in den Variablenspeicher. Durch das Sichern auf den Stapel wurde erreicht, daß die in der FN-Namensvariablen enthaltenen Werte nicht verlorengehen.

Die FN-Namensvariable enthält die Adresse der FN-Argumentvariablen, den Inhalt des CHRGET-Zeigers an der Definitionsposition sowie ein weiteres – mehr oder weniger zufälliges – Byte, das hinter dem Zuweisungszeichen der Definitionszeile steht. Dieses

Byte wird in der DEF-Routine lediglich zum Aufstocken der Informationen auf 5 Byte (= Länge eines Variableninhaltes) auf den Stapel gelegt, findet aber später keine Berücksichtigung mehr.

Als interessanter Programmteil sei noch erwähnt, wie die FN-Routine das Fehlen einer Funktionsdefinition feststellt: Es wird bei der Syntaxprüfung in CHKFNS (\$b3e1) die FNDVAR-Routine (\$b092) aufgerufen, die die FN-Variable für den Fall, daß sie noch nicht vorhanden ist (womit auch eine Funktionsdefinition fehlt), neu definiert und mit 0 belegt. 0 kann aber unmöglich der Inhalt einer definierten FN-Variablen sein, da diese zumindest eine andere Adresse als eine solche mit High-Byte \$00 (das wäre eine Zeropage-Adresse!) angeben muß. Es existiert also keine Funktionentabelle oder ein ähnliches Instrument, um festzustellen, ob Funktionen existieren.

Abbildung 4.25 faßt DEF und FN zusammen.

STR \$b465: Routine zur Basic-Funktion STR\$

Zum Verständnis dieser Routine ist zu erwähnen, daß zu einer Stringfunktion ein beliebiger Parameter bereits beim Eintritt in die Funktionsroutine ausgewertet worden ist. STR (\$b465) läßt dabei nur numerische Parameter zu, die dann im FAC stehen und mittels Aufruf der FLPSTR-Routine in einen ASCII-String umgewandelt werden, der ab \$00ff im Speicher steht. Diese Adresse wird dann an STRLIT (\$b487) zurückgegeben, wo der aktuell ermittelte String auf den temporären Stringstapel kommt.

\$475: Hilfsroutine zur Ermittlung der Stringparameter und Organisation von String-Speicherplatz

Steht die Adresse des aktuellen Variableneintrags in \$64/\$65, so wird der dafür benötigte Stringspeicherplatz, dessen Größe als Bytezahl im Akku zu stehen hat, bei »jsr \$b475« organisiert.

Die Adresse des Stringinhaltes befindet sich danach in \$62/\$63 sowie dem Registerpaar X/Y, die Stringlänge in \$61 sowie dem Akkumulator.

STRLIT (\$b487): Übergabe eines bei einer Stringfunktion ermittelten Strings auf den temporären Stringstapel

Wird ein String, der durch Anführungszeichen eingegrenzt wird, übertragen, so erfolgt hier der Einsprung. Dann wird das Anführungszeichen als Suchbyte 1 und Suchbyte 2 gesetzt. Es folgt der Einsprung zum Übertragen eines Strings, wobei Suchbyte 1 und 2 nicht belegt werden:

\$b48d: Einsprung zur Übertragung eines Strings in den temporären Stringstapel

Die in A/Y zu übergebende Stringadresse (auch beim STRLIT-Einsprung anzugeben!) wird zunächst in einen temporären Deskriptor sowie einen Hilfsspeicher für die Suchroutine geschrieben. Dann

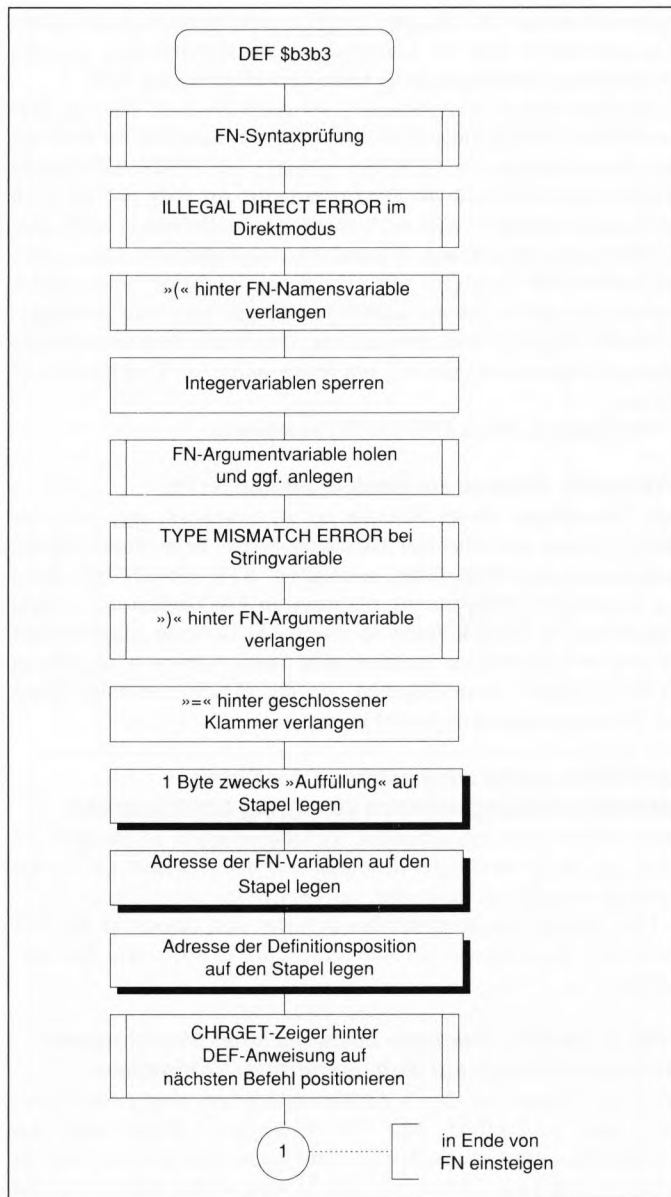


Abbildung 4.25: DEF und FN im Überblick (Teil 1)

wird die Stringlänge anhand der beiden Suchbytes ermittelt und in \$61 (Stringlängenspeicher im temporären Deskriptor) abgelegt. Die Endadresse des Strings wird aus der Anfangsadresse plus dieser Stringlänge ermittelt und im Hilfszeiger \$71/\$72 vermerkt. Darauf-

hin wird der String in den richtigen Speicherbereich übertragen und auf dem temporären Stringstapel als oberster Eintrag vermerkt. Ist der temporäre Stringstapel bereits voll, erfolgt die Meldung FORMULA TOO COMPLEX ERROR.

\$b4f4: Stringeintrag von vorgegebener Länge (Akku) im Stringspeicher anlegen

Diese Routine wird nur von der \$b475-Hilfsroutine verwendet; sie schafft einen Stringeintrag von der vorgegebenen Länge (Bytezahl im Akku), setzt dabei alle Hilfszeiger richtig und löst bei Speicherplatzmangel eine Garbage Collection aus.

Danach befindet sich jedenfalls in X/Y die Adresse des Strings und im Akku wieder die Stringlänge, obwohl der Akku innerhalb der Routine des öfteren verändert wird.

Eine programmtechnische Besonderheit soll hier erklärt werden: Bei \$b4f7–\$b500 liegt eine simulierte Subtraktion vor. Dabei wird die vor \$b4f7 im Akku enthaltene Stringlänge von der Anfangsadresse des Stringinhaltspeichers (Zeiger \$33/\$34) subtrahiert. Eine herkömmliche Subtraktion unter diesen Voraussetzungen hätte einen schwerwiegenden Nachteil: »SBC #stringlänge« wäre nicht möglich, da die **Stringlänge** im Akku steht – und nicht derjenige Wert, von dem sie abzuziehen ist. Eine Ausweidlösung für \$b4f7–\$b500 wäre lediglich folgendes (Listing 4.7):

```

100 -sta hilfsspeicher ; Stringlänge in
                        ; Hilfsspeicher merken
110 -lda $33           ; LB der Stringinhalts-
                        ; speicher-
120 -                 ; Anfangsadresse holen
130 -sbc hilfsspeicher ; Stringlänge subtrahieren
140 -ldy $34           ; HB der Stringinhalts-
                        ; speicher-
150 -                 ; Anfangsadresse holen
160 - bcs weiter      ; C=1: kein Subtraktions-
                        ; übertrag
170 - dey             ; HB dekrementieren, da
                        ; Übertrag
180 -weiter ...
  
```

Listing 4.7: Ausweidlösung für \$b4f7 mit »SBC hilfsspeicher«

Aber gerade dies wollten die C64-Programmierer vermeiden. Der Grund ist einfach, daß ein »hilfsspeicher« erforderlich wäre, wodurch noch mehr RAM vom Interpreter beansprucht würde als ohnehin schon.

Deshalb wird auf eine andere Lösung ausgewichen, die ohne »hilfsspeicher« auskommt. Diese macht sich folgendes mathematische Gesetz zunutze:

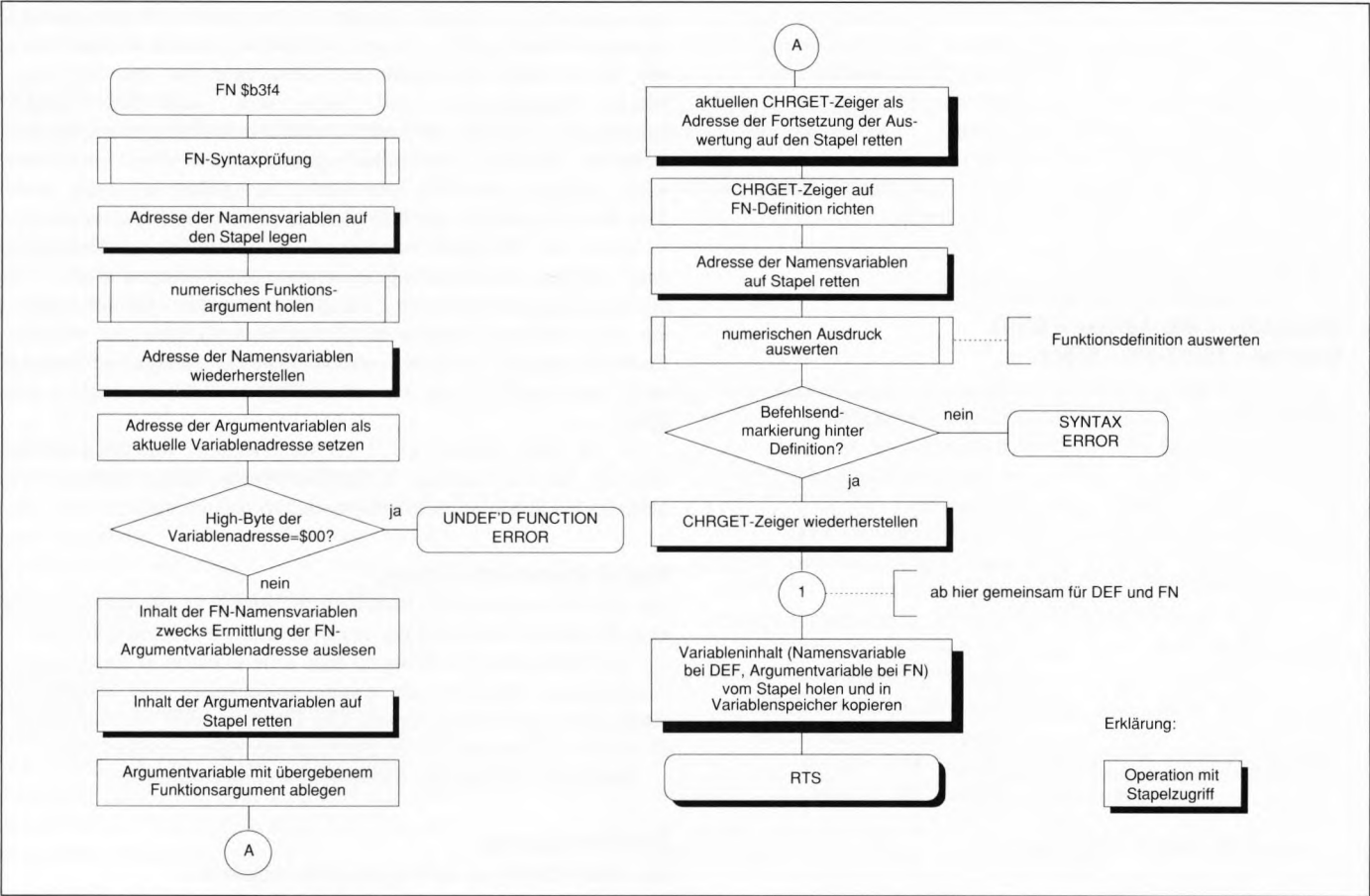


Abbildung 4.25: DEF und FN im Überblick (Teil 2)

Adresse – Länge = Adresse + (– Länge)

Nun kann das Kommutativgesetz angewendet werden, nach welchem die beiden Summanden in ihrer Reihenfolge variabel sind:

Adresse + (– Länge) = (– Länge) + Adresse

Und genau diesen letzten Rechenweg schlägt die Routine bei \$b4f7–\$b500 auch ein: Zunächst wird die Stringlänge komplementiert, was einem Vorzeichenwechsel gleichkommt. Anschließend wird das Carry-Flag gesetzt, so daß schließlich die Addition bei \$b4fa außer der komplementierten Stringlänge auch 1 addiert (sonst wäre das Ergebnis nicht korrekt). Die Übertragsprüfung muß wiederum wie bei einem SBC-Befehl erfolgen: Ist Carry=1 bei einem »echten« Additionsübertrag, so ist es in diesem Fall (wie bei einer Subtraktion) 0,

falls ein Übertrag vorliegt, der durch Dekrementieren des High-Bytes Berücksichtigung findet.

Falls Sie das Prinzip noch nicht kannten oder noch nicht ganz verstanden haben, helfen Ihnen sicher die beiden folgenden Beispiele:

- 1. Stringlänge = \$40, Adresse = \$7f50;
Ergebnis = \$7f50–\$40 = \$7f10**

```
,b4f7 eor #ff %11111111 ; Akku enthält danach $bf
,b4f9 sec                ; Carry setzen
,b4fa adc 33             ; $50 (LB der Adresse) +
                        ; 1 (Carry) addieren
                        ; Ergebnis:
                        ; $bf+$50+$01 = $10
```

```

; Additionsübertrag (C=1),
; daher: kein
; Subtraktionsübertrag
,b4fc ld 34 ; HB der Adresse ($7f)
; laden
,b4fe bcs b501 ; fertig, da kein
; Subtraktionsübertrag
; (s. $b4fa)

; Ergebnis: A/Y = $7f10

```

2. Stringlänge = \$0f, Adresse = \$5c03; Ergebnis = \$5c03-\$0f = \$5bf4

```

,b4f7 eor #ff %11111111 ; Akku enthält danach $f0
,b4f9 sec ; Carry setzen
,b4fa adc 33 ; $03 (LB der Adresse) +
; 1 (Carry) addieren
; Ergebnis:
; $f0+$03+$01 = $f4
; kein Additionsübertrag
; (C=0), daher:
; Subtraktionsübertrag!
,b4fc ld 34 ; HB der Adresse ($5c)
; laden
,b4fe bcs b501 ; noch nicht fertig, da
; Subtraktionsübertrag!
,b500 dey ; HB auf $5b setzen
; (wegen Übertrag)

; Ergebnis: A/Y = $5bf4

```

\$b516: bedingte Ausführung der Garbage Collection

Diese Routine führt die Garbage Collection für den Fall durch, daß das entsprechende Flag (Adresse \$0f) nicht dagegenspricht. Anschließend wird dieses Garbage-Collection-Flag gesetzt, damit eine mehrfache Ausführung der Garbage Collection, die auch keinen Speicherplatzgewinn mehr zu bringen vermag, ausgeschlossen ist.

Dieser \$b516-Programmteil gehört zur Routine für die Organisation von Stringinhaltspeicherplatz.

GARCOL (\$b526): Garbage Collection

Die Garbage Collection (Reorganisation des Stringinhaltspeichers durch Löschen der nicht mehr benötigten Stringinhalte) wurde schon wiederholt angesprochen. Deshalb soll hier nur auf deren Funktionsweise eingegangen werden.

Im wesentlichen gibt es jeweils zwei aktuell bedeutsame Adressen: zum einen die Adresse des aktuellen Stringinhaltspeichers,

zum anderen die Adresse des gerade überprüften Variableneintrags. In einer Schleife wird zu jedem Stringinhaltspeicher-Eintrag ermittelt, ob er noch von irgendeinem Deskriptor aus dem Variablenbereich angesprochen wird. Falls nein, wird der Stringinhaltspeicher dadurch verkleinert, daß der unterhalb des ungenutzten Bereiches liegende Stringinhaltspeicher nach oben verschoben wird, wodurch unterhalb des neuen Stringinhaltspeichers freier Speicherplatz entsteht; der Stringinhaltspeicher wird also reduziert.

Sollte der Stringinhaltspeicher-Eintrag von einem Deskriptor einer gültigen Variablen angesprochen worden sein, so wird die in diesem Deskriptor enthaltene Länge zur aktuellen Adresse addiert, um den nächsten Eintrag im Stringinhaltspeicher zu erhalten; Endmarkierungen in Form von \$00 kennt der Stringinhaltspeicher nicht, da ja auch Strings wie A\$=CHR\$(0)+CHR\$(0) möglich sein sollen.

Es sei noch einmal auf 3.4.6 hingewiesen, wo sie auch ein Beispiel für die Garbage Collection finden, deren Routine sich übrigens bis \$b63c (einschließlich) im Speicher erstreckt.

\$b63d: Stringverknüpfung

Bei der Stringauswertung innerhalb des FRMEVL-Komplexes dient diese Routine zur Behandlung des Operators »+« bei zwei Strings.

Dazu wird der erste String an eine neue Position in den Stringinhaltspeicher gebracht, der zweite ausgewertet und unmittelbar hinter den ersten String kopiert. Der Ergebnisstring hat eine Länge, die durch Addition der beiden Einzelstringlängen errechnet wird.

Abbildung 4.26 faßt die Stringverknüpfung zusammen.

STRVAR (\$b67a):

aktuellen String in Stringspeicher kopieren

Diese Routine setzt voraus, daß die Adresse des aktuellen Stringdeskriptors in \$6f/\$70 steht. Dann wird der Deskriptor nach A (Stringlänge) und X/Y (Stringinhaltsadresse) gelesen. Bei \$b688 wird auch von \$b4c7 eingestiegen, um einen String an der in X/Y enthaltenen Adresse in den Stringinhaltspeicher zu kopieren, wobei die Zieladresse in \$35/\$36 enthalten sein muß.

Bei \$b68c wird noch erwartet, daß die Quelladresse bereits in \$22/\$23 steht.

Der String-Hilfszeiger \$35/\$36 wird danach um die Stringlänge erhöht; dies spielt bei der Stringverknüpfung eine tragende Rolle, denn dort soll der zweite Verknüpfungsstring hinter den ersten kopiert werden.

\$b6a3: Prüfung

auf Stringausdruck und FRESTR-Ausführung

Dieser Einsprung prüft, ob der letzte über FRMEVL (\$ad9e) eingeholte Parameter ein String ist (falls nein: TYPE MISMATCH

ERROR) und führt dann die unmittelbar folgende FRESTR-Routine (\$b6a6) aus:

FRESTR (\$b6a6):
String aus Basic-Text weiterverarbeiten

Wenn bei FRMEVL (\$ad9e) ein String eingeholt wurde, so bedient sich der Interpreter dieser Routine, sobald der String in den Stringinhaltsspeicher übernommen werden soll; dazu kommt seine Adresse nach \$22/\$23, und der Stringbereichszeiger wird angepaßt.

Dafür wird der String vom temporären Stringstapel entfernt, wozu folgende Hilfsroutine dient:

\$b6db: Eintrag im temporären Stringstapel löschen

Hier wird für den Fall, daß der letzte String vom temporären Stringstapel in den Stringinhaltsspeicher übertragen wird, dessen Eintrag im temporären Stringstapel zur Entlastung entfernt.

CHR \$b6ec:
Routine zur Basic-Funktion CHR\$

Diese Routine holt den Bytewert-Parameter von CHR\$ ein, legt einen 1-Byte-String an, beschreibt diesen mit dem Bytewert und gibt das auf diese Weise gewonnene Ergebnis am temporären Stringstapel zurück.

LEFT \$b700:
Routine zur Basic-Funktion LEFT\$

Die LEFT\$-Routine bildet den linken Teilstring der angegebenen Länge, indem der rechte Reststring bei der Bildung des Ergebnisstrings »abgeschnitten« (überlesen) wird. In die LEFT\$-Routine steigen auch RIGHT (\$b706) und MID (\$b70d) ein, da RIGHT\$ genau von der anderen Seite – von links – den Reststring abschneidet und MID\$ schließlich im Extremfall beidseitige Reststrings überliest.

An dem Befehl bei \$b70e jedoch führen alle drei Stringfunktionen (MID\$, LEFT\$ und RIGHT\$) vorbei. Dort werden folgende im voraus ermittelten Werte erwartet:

\$50/\$51 = Adresse des Stringdeskriptors für den Ausgangsstring

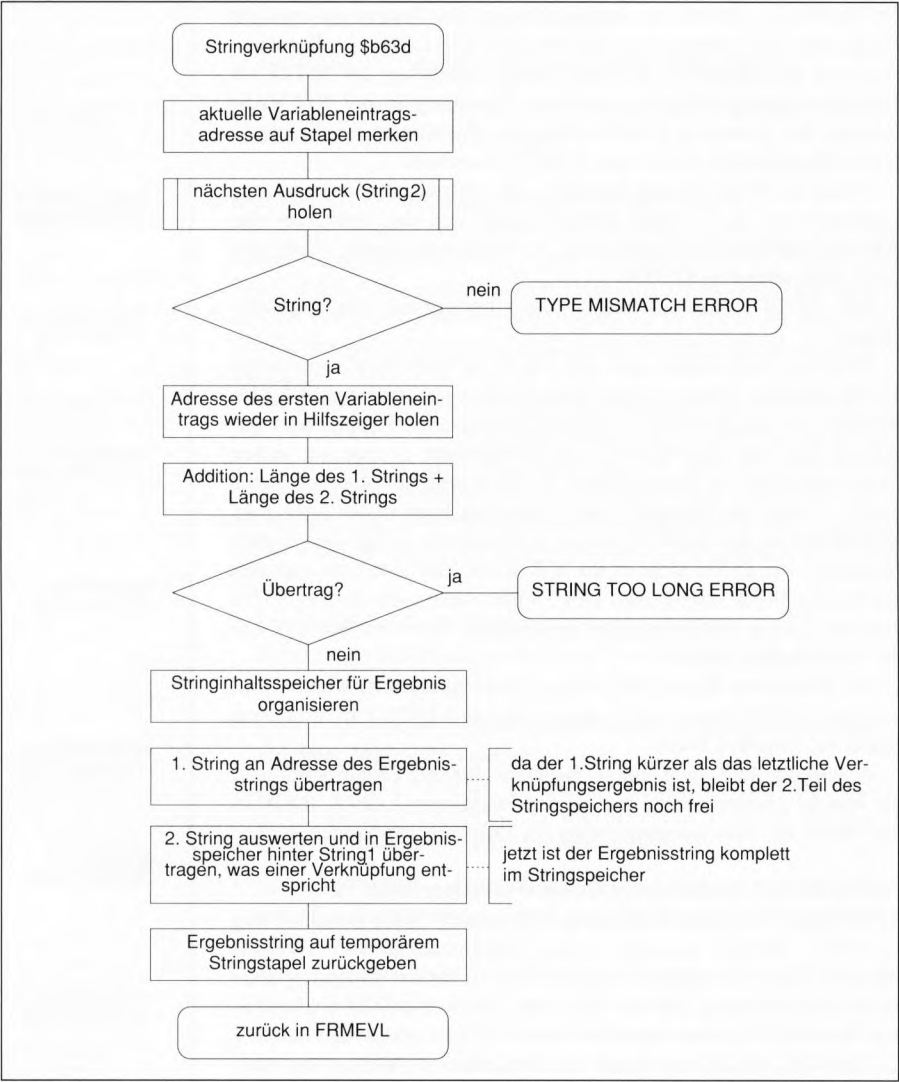


Abbildung 4.26: Ablauf einer Stringverknüpfung

Akkumulator = Länge des Ergebnisstrings
oberstes Byte auf Stapel = Länge des linken Reststrings

Die jeweils spezifischen Teile zu LEFT\$, RIGHT\$ und MID\$ haben diese Parameter bis \$b70e zu beschaffen, so daß dann der durch die Parameter eindeutig beschriebene Teilstring ermittelt werden kann. Dabei wird die Anfangsadresse des Teilstrings als Anfangsadresse des zu kopierenden Strings in \$22/\$23 abgelegt (siehe

\$b71b–\$b725), indem zur Anfangsadresse des Ausgangsstrings die Länge des linken Reststrings addiert wird. Die Länge des Ergebnisses wird als Länge des zu kopierenden Teilstrings bei \$b725 zur Übergabe bereitgestellt, bis dann ein Einsprung in die STRVAR-Routine den Teilstring herauskopiert. Die Zieladresse des Ergebnisses wurde schon bei \$b70f nach \$35/\$36 berechnet.

Wenn im ROM-Listing die Rede von »String an Adresse (. . .) kopieren« ist, so ist damit gemeint, daß sich der String **an der Adresse befindet**; die Zieladresse, zu welcher der String übertragen wird, steht immer in \$35/\$36.

Bei \$b729 schließlich erfolgt die Übermittlung des Ergebnisstrings.

Bleibt noch zu klären, wie von LEFT (\$b700) die Parameter bis \$b70e ermittelt werden. Dabei übernimmt der Aufruf der PREAM-Routine, der schon bei \$b700 als erstes erfolgt, den Hauptteil der Arbeit; er holt den numerischen LEFT\$-Parameter (Länge des linken Teilstrings) ein. Ist dieser größer als der String selbst (oder gleich groß), so wird die Stringlänge als Länge des Ergebnisses festgelegt, andernfalls ist der LEFT\$-Parameter bereits die Länge des Ergebnisstrings; bei \$b70d steht er im X-Register, das also entweder den bereits bei \$b700 eingeholten LEFT\$-Parameter oder die bei \$b70a geladene Länge des Ausgangsstrings enthält, und wird bei \$b70e in den Akkumulator geladen.

Die Länge des linken Reststrings, der abgeschnitten werden soll, wird bei LEFT\$ immer auf 0 gesetzt, da die LEFT\$-Funktion »von links« das Ergebnis bildet.

Abbildung 4.27 zeigt einige Beispiele für die Teilstringbildung, die jeweils demonstrieren, wie bei den Funktionen LEFT\$, RIGHT\$ und MID\$ aus dem Ausgangsstring das Ergebnis entnommen wird.

RIGHT \$b72c: Routine zur Basic-Funktion RIGHT\$

RIGHT (\$b72c) ermittelt die nötigen Parameter (siehe Beschreibung zu LEFT \$b700) ebenfalls unter Zuhilfenahme von PREAM (\$b761). Durch Einsteigen in LEFT (\$b700) bei der Einsprungstelle \$b706 wird bewirkt, daß der Akku als Länge des links abzuteilenden Reststrings weiterverarbeitet wird. In den Akku lädt RIGHT (\$b72c) die Anzahl der links zu überlesenden Zeichen im Ausgangsstring, welche wie folgt berechnet wird:

– (Länge des Ergebnisstrings – Länge des Ausgangsstrings)

Die Länge des Ergebnisstrings ist dabei der numerische RIGHT\$-Parameter. Die Formel stellt in der obigen Form den Rechenweg dar, den RIGHT (\$b72c) einschlägt; mathematisch gesehen läßt sie sich jedoch noch etwas vereinfachen, um die Transparenz zu vergrößern:

Länge des Ausgangsstrings – Länge des Ergebnisstrings

Bei der Funktion RIGHT\$("12345",3) ist dies also

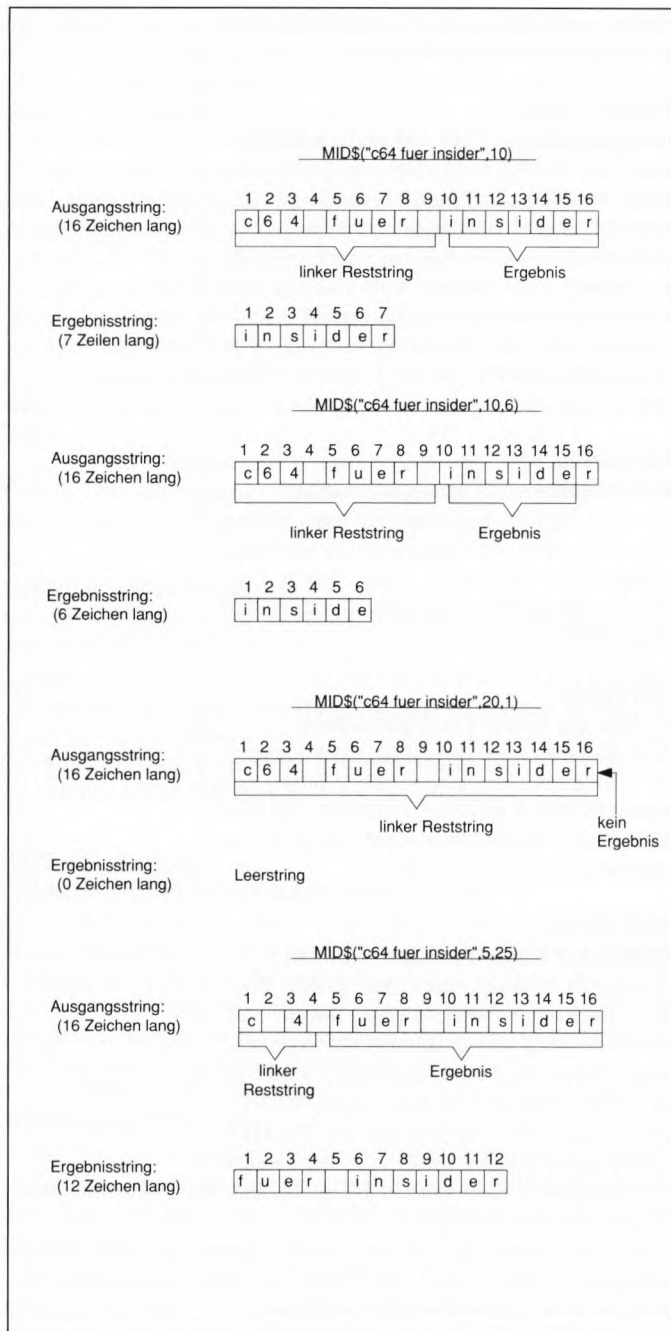


Abbildung 4.27: Bildung der Ergebnisse von LEFT\$, RIGHT\$ und MID\$ (Teil 1)

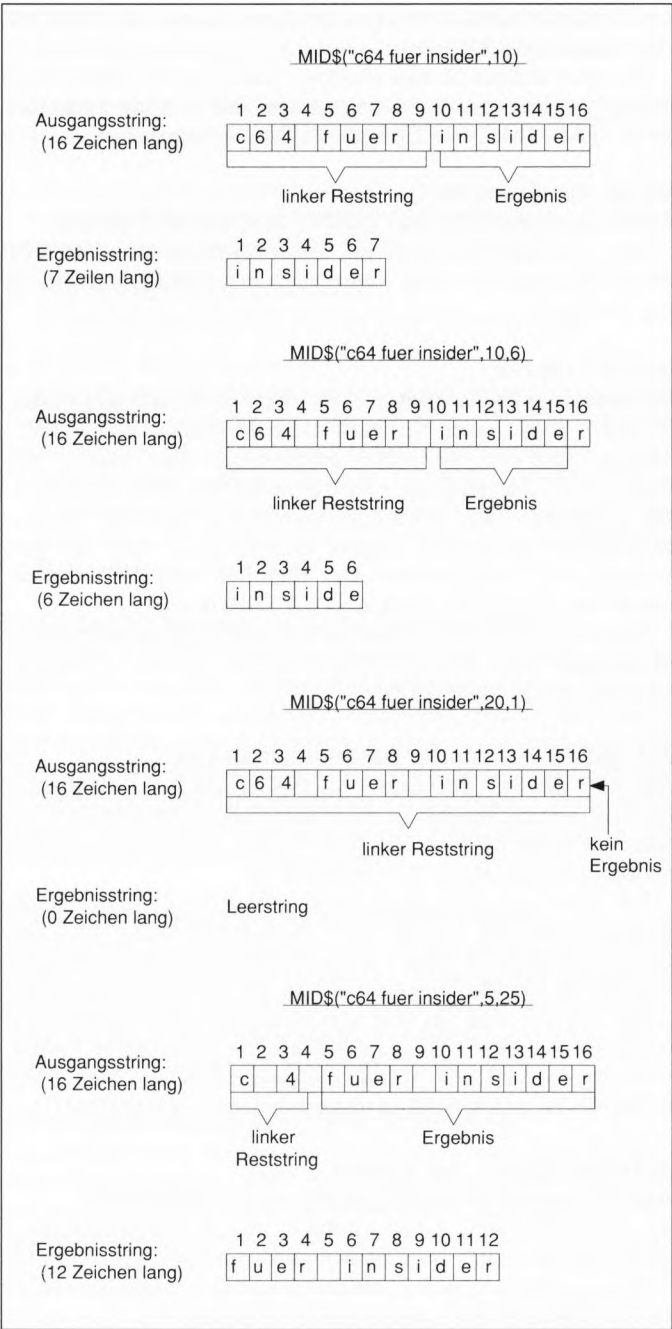


Abbildung 4.27: Bildung der Ergebnisse von `LEFT$`, `RIGHT$` und `MID$` (Teil 2)

$[Länge\ von\ "12345"] - [3] = 5 - 3 = 2$

In der Tat ist das Ergebnis »45« exakt zwei Zeichen lang.

Es sei auch hier auf Abbildung 4.27 hingewiesen, wo unter anderem `RIGHT$`-Beispiele zu finden sind.

MID \$b737: Routine zur Basic-Funktion MID\$

Die `MID$`-Funktion kann zwei oder drei Parameter haben. Der erste ist der Ausgangsstring, der zweite die Startposition (Position des ersten Bytes, von dem an »nach rechts« der Ergebnisstring zu bilden ist) und der (nicht zwingend erforderliche) dritte Parameter die Anzahl der herauszunehmenden Zeichen, also die Länge des Ergebnisstrings.

Wird der letzte Parameter nicht angegeben, so geht `MID ($b737)` von 255 Zeichen, der maximalen Stringlänge, aus. Die Länge des linken Reststrings geht in jedem Fall aus dem zweiten `MID$`-Parameter (Startposition) hervor, welcher aber um 1 verringert wird, da ja schon das an der Startposition befindliche Byte zum Ergebnis gehört. Für die Länge des Ergebnisstrings sind Fallunterscheidungen zu treffen:

- 1. Startposition-1 > Länge des Ausgangsstrings**
Dann liegt die Ergebnisstringlänge 0 vor, da bildlich gesprochen »außerhalb des Ausgangsstrings ein Ergebnis zu bilden wäre«.
- 2. Differenz zwischen Startposition-1 und Ausgangsstringlänge < dritter MID\$-Parameter**
In diesem Fall wird die Differenz als positiver Wert zur Ergebnisstringlänge, wodurch also die maximal mögliche Zahl von Zeichen Aufnahme in das Ergebnis findet.
- 3. Startposition-1 <= Länge des Ausgangsstrings, aber Differenz zwischen Startposition-1 und Ausgangsstringlänge >= dritter MID\$-Parameter**

Nur in diesem letzten und in der Praxis häufigsten Fall ist der dritte `MID$`-Parameter tatsächlich die Ergebnisstringlänge: Rechts von der Startposition sind noch genügend Zeichen vorhanden, um daraus das Ergebnis mit der gewünschten Länge zu bilden.

Zum letzten Mal möchte ich auf Abbildung 4.27 wegen der repräsentativen Beispiele zu `LEFT$`, `RIGHT$` und `MID$` hinweisen.

PREAM (\$b761): an Stringfunktion übergebenen String vom Stapel holen

Bevor eine Stringfunktion wie `RIGHT$` durch die funktionspezifische Routine interpretiert wird, durchläuft sie den Funktionsverteiler, also denjenigen Teil der `FRMEVL/EVAL`-Routine, der für die Auswertung von Funktionen zuständig ist. In diesem Funktionsverteiler wird für eine Stringfunktion zunächst ein beliebiger

Parameter (String oder Zahl) ausgewertet, sowie ein Bytewert, der – durch Komma abgetrennt – auf diesen folgt. Dies entlastet die funktionspezifischen Routinen, die die entsprechenden Werte nur noch vom Stapel einlesen müssen. Und um auch diese Aufgabe mit möglichst geringem Aufwand zu bewältigen, wurde die PREAM-Routine geschaffen. Diese prüft zunächst, ob hinter den Parametern eine geschlossene Klammer folgt; dadurch wird sichergestellt, daß kein überflüssiger Parameter angegeben wurde.

Dann wird der im Funktionsverteiler ausgewertete Parameter (nach Sicherstellung der Rücksprungadresse) vom Stapel ins X-Register geholt und bei \$b77a noch zusätzlich in den Akkumulator übertragen. Die Adresse des Stringdeskriptors wird vom Stapel nach \$50/\$51 ausgelesen und das Y-Register mit 0 belegt (Initialisierung des Offset). Es erfolgt ein Rücksprung an diejenige Routine, die PREAM aufgerufen hat.

LEN \$b77c: Routine zur Basic-Funktion LEN

Die LEN-Routine reduziert sich auf zwei Unterprogrammaufrufe: Zum einen werden die Stringparameter über STRPAR (\$b782) geholt, wonach die Stringlänge im Y-Register steht; dieser Wert wird zum anderen durch BYTFAC (\$b3a2) als Ergebnis in den FAC gebracht.

STRPAR (\$b782): Auswertung eines an Stringfunktionen übergebenen Strings

Durch einen Einsprung bei \$b6a3, der sicherstellt, daß der letzte Parameter ein String war, und dann FRESTR (\$b6a6) ausführt, wird der String weiterverarbeitet. Die Eigenleistung von STRPAR (\$b782) besteht nun darin, das Datentyp-Flag \$0d auf »numerisch« zu setzen (deshalb wird STRPAR von denjenigen Funktionsroutinen eingesetzt, die einen String als Parameter erhalten und einen Zahlenwert zurückgeben) und die Stringlänge vor dem RTS-Rücksprung vom Akku ins Y-Register zu übertragen. Dies hat zweierlei Bedeutung:

1. Die Stringlänge kann in LEN (\$b77c) sofort an BYTFAC (\$b3a2) weitervermittelt werden.
2. Die CPU-Flags, insbesondere das Zero-Flag, werden gemäß der Stringlänge gesetzt. Beispielsweise prüft ASC (\$b78b) bei \$b78e die Stringlänge nach »jsr strpar« lediglich durch einen BEQ-Befehl.

ASC \$b78b: Routine zur Basic-Funktion ASC

Die ASC-Routine ist ein typisches Beispiel für eine Funktionsroutine, die einen String als Parameter über STRPAR (\$b782) auswertet und einen Zahlenwert (den ASCII-Code des ersten Zeichens im String) zurückgibt. Bei einer Stringlänge von 0 wird ein ILLEGAL QUANTITY ERROR ausgelöst; dies ist eine leichte

Schwäche der Funktion, die vor allem beim Einlesen von Daten aus einer Diskettendatei sehr stört.

Bei allen anderen Strings allerdings wird das erste Byte (Offset 0, Stringinhaltszeiger \$22/\$23) ausgelesen und über das Y-Register an BYTFAC (\$b3a2) zur Rückgabe im FAC weitergeleitet.

\$b79b: Einsprung für

Ausführung von CHRGET (\$0073) und GETBYT (\$b79e)

Dieser Einsprung holt nicht nur einen Bytewert über GETBYT (\$b79e) ein, sondern erhöht vorher noch den CHRGET-Zeiger um eine Position.

GETBYT (\$b79e):

Bytewert aus Basic-Text in X-Register und nach \$65 holen

GETBYT (\$b79e) stützt sich auf FRMNUM (\$ad8a); die INTEVL-Routine wird erst bei \$b1b8 angesprungen und wandelt den Parameter ins Integerformat, so daß er in \$65/\$64 (\$65 = Low-Byte, \$64 = High-Byte, also »High-Low-Format«) zu finden ist. Auf einfache Weise kann geprüft werden, ob es sich um einen Bytewert (maximal 255 = \$ff) handelt: Das High-Byte muß \$00 sein, ansonsten liegt ein ILLEGAL QUANTITY ERROR vor.

Über CHRGOT (\$0079) wird deshalb zurückgesprungen, weil im Akkumulator nach dem GETBYT-Aufruf das Zeichen stehen soll, das direkt auf den Bytewert folgt.

VAL \$b7ad: Routine zur Basic-Funktion VAL

Wie ASC (\$b78b), ruft auch VAL (\$b7ad) die STRPAR-Routine auf. Eine weitere Parallele ist die Prüfung auf die Stringlänge 0; ein Leerstring erhält bei VAL den Wert 0 (siehe \$b7b2) und löst somit keinen ILLEGAL QUANTITY ERROR aus.

Ein String wird nun ausgewertet, indem der CHRGET-Zeiger übergangsweise auf den an VAL übergebenen String weist und dieser durch STRFLP (\$bcf3) in eine Fließkommazahl umgewandelt wird. Da die STRFLP-Routine das Byte \$00 als Endmarkierung des Strings erwartet, muß das Byte, das hinter dem VAL-Parameter im Stringinhaltspeicher steht, zunächst gerettet (siehe \$b7cf–\$b7d3), dann für den STRFLP-Ablauf zwischenzeitlich mit \$00 belegt (\$b7d4–\$b7d6) und schließlich restauriert werden (\$b7dd–\$b7e1).

GETWRB (\$b7eb): 2-Byte-Wert

und davon durch Komma getrennten Bytewert holen

Diese Routine holt zunächst durch die Routinenkombination FRMNUM-FACWRD einen numerischen Ausdruck nach \$14/\$15, prüft dann auf ein Komma (CHKCOM) und holt zuletzt über GETBYT (\$b79e) auch noch einen Bytewert ins X-Register.

Bei \$b7f1 wird (innerhalb von GETWRB) noch ein Einsprung durch andere Routinen genutzt:

GETCBT (\$b7f1): Kommaprüfung und Bytewert holen

Es handelt sich um die Routinenkombination CHKCOM-GETBYT.

FACWRD (\$b7f7): FAC als 2-Byte-Wert nach \$14/\$15 holen

Diese Routine besteht aus drei Einzelaufgaben:

1. FAC auf gültigen Bereich (0–65535) prüfen
Falls der zulässige Zahlenbereich 0–65535 über- oder unterschritten wird, erfolgt ein ILLEGAL QUANTITY ERROR.
2. FAC in Integerformat umwandeln
Dafür findet die FACINT-Routine (\$bc9b) Verwendung.
3. Rückgabe des Ergebnisses in \$14/\$15
Da in Schritt 2 die FACINT-Routine das Konvertierungsergebnis in \$65/\$64 einerseits im ungewöhnlichen High-Low-Format, andererseits an einer ungünstigen Speicherposition (FAC-Speicherzellen!) ablegt, überträgt FACWRD (\$b7f7) diese 2 Bytes nach \$14/\$15 und bringt sie noch dazu in die übliche Reihenfolge (Low-High-Format), so daß guten Gewissens der Rücksprung stattfinden darf.

PEEK \$b80d: Routine zur Basic-Funktion PEEK

Die PEEK-Funktion ist eine »rein-numerische Funktion«, d.h. sie erhält nicht nur einen numerischen Parameter, sondern gibt auch einen solchen zurück. Der Übergabeparameter wird bei \$b813 durch »jsr facwr« nach \$14/\$15 geholt, weshalb dieser Hilfszeiger bei \$b80d–\$b813 gerettet und bei \$b81b–\$b820 wiederhergestellt werden muß. Der Inhalt der durch den Zeiger \$14/\$15 bezeichneten Adresse wird also als Parameter nach Y geholt (\$b816–\$b81a) und bei \$b821 als Bytewert zurückgegeben.

POKE \$b824: Routine zum Basic-Befehl POKE

Die gesamte Parameterübergabe von POKE erledigt ein einziger Aufruf: Nach »jsr getwr« steht im X-Register der gewünschte Adresseninhalt (der zu POKende Wert) und der Zeiger \$14/\$15 weist auf die zu ändernde Adresse. Die Befehle bei \$b827–\$b82b genügen zum Schreiben des Wertes an die Zieladresse.

WAIT \$b82d: Routine zum Basic-Befehl WAIT

Zur Analyse dieser Routine müssen wir uns wieder einmal mit logischen Verknüpfungen (AND und EOR) herumschlagen. Um diese Logeleien möglichst zu vereinfachen, bespreche ich deshalb die beiden Syntax-Möglichkeiten von WAIT getrennt.

1. kurze Syntax: WAIT adresse,byte

In diesem Fall wird solange in der Warteschleife verharret, bis die »adresse« das »byte« enthält. Die beiden Parameter werden bei \$b82d durch »jsr getwr« auf denkbar unkomplizierte Weise ein-

geholt, so daß \$14/\$15 auf die Adresse weist; der Bytewert kommt sogleich vom X-Register in den Hilfspfeicher \$49 (siehe \$b830).

Der Vorbelegungswert für den Hilfspfeicher \$4a (zweiter Byte-Parameter, der in der verkürzten Syntax wegfällt) ist 0 (siehe \$b832). Bei \$b83e wird dann noch der Offset auf 0 gesetzt, da vom Hilfszeiger \$14/\$15 zur auszulesenden Adresse kein Offset besteht.

\$b840 liest nun das Byte an der WAIT-Adresse aus und führt eine Exklusiv-Oder-Verknüpfung mit Byte 2 durch; in diesem ersten Fall wird also »eor #00« ausgeführt, was keine Veränderung des Akkumulators mit sich bringt. Anschließend erfolgt eine Und-Verknüpfung mit Byte 1; wenn kein einziges Bit von Byte 1 sowie dem tatsächlich vorhandenen WAIT-Adreßinhalt übereinstimmt, so ist das Zero-Flag gesetzt und der BEQ-Befehl bei \$b846 setzt die WAIT-Schleife fort. Andernfalls wird der WAIT-Befehl beendet.

Wie an der Routine also zu erkennen ist, genügt die Übereinstimmung eines einzigen 1-Bits zwischen dem WAIT-Byte und dem Inhalt der WAIT-Adresse. Deshalb sind die WAIT-Bytes in den meisten Anwendungsfällen Zweierpotenzen wie 32 (2 hoch 5), weil oft nur einzelne Bits mit WAIT abgefragt werden.

Durch »WAIT 198,1« wird somit gewartet, bis ein Tastendruck die Adresse 198 (Anzahl der Zeichen im Tastaturpuffer) auf 1 erhöht; da aber nicht mit Sicherheit davon ausgegangen werden kann, daß inzwischen kein Tastendruck im Tastaturpuffer gelagert wurde, setzt man vorher meist mit »POKE 198,0« diesen Zähler zurück.

2. lange Syntax: WAIT adresse,byte1,byte2

Da Sie soeben den Fall 1 kennengelernt haben, schwant Ihnen sicher schon, was die programmtechnische Eigenheit von Fall 2 ist: In diesem zweiten Fall wird in der WAIT-Schleife noch eine Exklusiv-Oder-Verknüpfung mit einem Bytewert durchgeführt, der nicht unbedingt 0 ist (»0« als zweites Byte kann auch weggelassen werden, da die Null ohnehin als Defaultwert angenommen wird).

Sehen wir uns noch einmal die Suchschleife an. Bei \$b840 wird die WAIT-Adresse ausgelesen, bei \$b842 erfolgt die EOR-Verknüpfung mit Byte 2. Stimmen Byte 2 und der Inhalt der WAIT-Adresse überein, so steht danach \$00 im Akkumulator, und da auch der AND-Befehl bei \$b846 diesen Wert nicht mehr zu ändern vermag (0 AND x = 0) wird die WAIT-Schleife fortgesetzt. Durch die Angabe eines zweiten Bytes ist also eine Zusatzbedingung in Form einer EOR-Maske möglich.

Wenn darauf gewartet werden soll, daß ein Bit in einer Adresse auf 0 gesetzt wird, so müssen Byte 1 und 2 identisch sein:

```
WAIT 1,32,32
```

wartet auf das Drücken einer Datensettentaste, da in diesem Fall Bit 4 (Wertigkeit 32) gelöscht wird.

Des weiteren wartet

WAIT 56320,16,16

auf das Drücken des Feuerknopfes im Joystick-Port 2, während

WAIT 56321,16,16

dasselbe für Port 1 bewirkt.

Zusammenfassend läßt sich für den gesamten WAIT-Befehl sagen, daß die komplizierte Programmierung seiner Routine im richtigen Verhältnis zur Schwierigkeit in der Nutzung dieser Anweisung besteht, mit der kaum ein Basic-Programmierer etwas anzufangen weiß – von den hier aufgeführten Standardbeispielen aus der Tips- und Tricks-Urzeit einmal abgesehen.

ADD0.5 (\$b849): FAC um 0.5 erhöhen

Zuerst wird hierzu mittels eines Aufrufes von ADDMEM (\$b867) der Wert 0.5 zum FAC #1 addiert. Die MFLPT-Darstellung der Konstante 0.5 befindet sich bei \$bf11 und wird auch von der SQR-Funktion verwendet (dort aber als Exponent und nicht als Summand).

Für die Rundung zwecks Eliminierung von Nachkommastellen kann die ADD0.5-Routine vor einer Wandlung ins Integerformat hilfreich sein, da INT(100.9) bekanntlich »100« ergibt, obwohl die gebräuchliche Rundung – bei vorausgegangenem ADD0.5 – »101« ermitteln müßte.

SUBMEM (\$b850):

FAC von MFLPT-Konstante abziehen, Ergebnis in FAC

Diese Routine bereitet SUBFAC (\$b853) so vor, daß der FAC #1 von einer Konstante, deren Adresse in A/Y zu übergeben ist, subtrahiert wird. Dazu wird die Konstante in den ARG (FAC #2) übertragen, der bei der SUBMEM-Subtraktion den Minuenden (Wert, von dem etwas abgezogen wird) enthält, während FAC #1 der Subtrahend (Wert, der abgezogen wird) ist.

SUBFAC (\$b853): FAC von ARG abziehen, Ergebnis in FAC

Eine Subtraktion »ARG-FAC« wird programmtechnisch realisiert, indem das Vorzeichen des FAC invertiert wird, was einer Multiplikation mit »-1« gleichkommt. Gleichzeitig wird das Vorzeichenvergleichsbyte (\$6f) richtig gesetzt, damit auch das Ergebnis ein richtiges Vorzeichen hat.

Nun werden ARG und FAC addiert (ADDFAC \$b86a); im eigentlichen Sinne existiert also keine eigene Subtraktionsroutine, vielmehr wird die Subtraktion durch eine Addition ersetzt:

$$\text{ARG-FAC} = \text{ARG} + (-\text{FAC})$$

EQUEXP (\$b862): FAC

und ARG für Addition auf gleichen Exponenten bringen

Um den Sinn der EQUEXP-Routine zu verstehen, ist es am vorteilhaftesten, sich einmal in die Rolle der Basic-2.0-Entwickler zu versetzen. Wie kann man zwei Fließkommazahlen addieren?

Nun, auf gar keinen Fall durch einfache Addition aller Bytes der Fließkommadarstellung, denn es handelt sich lediglich bei der Mantisse um eine 4-Byte-Zahl in einer Art »Lowest/Low/High/Highest«-Format; zwei Exponenten hingegen darf man nicht addieren. An einem einfachen Beispiel aus dem Dezimalsystem läßt sich dies nachvollziehen:

$$\begin{aligned} 10^2 &= 100; & 10^5 &= 100\,000 \\ 10^2 + 10^5 &= 100 + 100\,000 = 100\,100 \\ (10+10)^{(2+5)} &= 20^7 = 1\,280\,000\,000 \\ 100\,100 &<> 1\,280\,000\,000! \end{aligned}$$

Das Ergebnis ist nicht $(10+10)^{(2+5)}$, was 20^7 (1.28 Milliarden!) entspräche, sondern 100100. Übertragen wir diese Erkenntnis nun auf das Fließkommaformat; dort wird eine Zahl aus Mantisse, Basis (immer 2) und Exponent zusammengesetzt. Auch hier dürfen die Mantissen und Exponenten nicht einfach addiert werden:

$$\begin{aligned} 1.2 &= 0.6 * 2^1; & 6.4 &= 0.8 * 2^3 \\ 0.6 * 2^1 + 0.8 * 2^3 &= 1.2 + 6.4 = 7.6 \\ (0.6+0.8) * 2^{(1+3)} &= 1.4 * 2^4 = 22.4 \\ 7.6 &<> 22.4! \end{aligned}$$

Nach diesen beiden Fehlerbeispielen drängen Sie sicher darauf zu erfahren, wie es denn nun funktioniert. Die Lösung dafür ist viel einfacher, als man zunächst vermutet, denn es gibt ja folgendes Rechengesetz:

$$a * c + b * c = (a+b) * c$$

Angewandt auf zwei Fließkommazahlen »x« und »y« mit den Mantissen »a« und »b« und einem identischen Exponenten »exponent«:

$$a * 2^{\text{exponent}} + b * 2^{\text{exponent}}$$

»c« ist also in diesem Fall durch »Basis hoch Exponent« ersetzt worden, »a« und »b« sind die Mantissen von zwei Zahlen. Normalerweise ist das Rechengesetz dann für Fließkommazahlen nicht anwendbar, da »Basis hoch Exponent«, also der Faktor »c«, nicht immer gleich sein muß – nur in Ausnahmefällen haben zwei Fließkommazahlen denselben Exponenten.

Das Problem ist also folgendes: Wie bringen wir »c« auf einen einheitlichen Wert, damit die Addition von »a« und »b« bei einem neuen »c« der Summe der beiden Fließkommazahlen entspricht? Und hier setzt EQUEXP (\$b862) ein. Diese Routine bringt FAC und ARG, also die beiden Summanden, für eine Addition auf denselben Exponenten; dabei werden die Mantissen gleichzeitig angepaßt, damit sich die Zahlenwerte nicht ändern, sondern nur auf eine einheitliche Form gebracht werden. Beim Einsprung steht im Akkumulator die Exponentendifferenz, nach der Gleichmachung der Exponenten springt EQUEXP (\$b862) in die Additionsroutine zurück.

Für eigene Anwendungen ist EQUEXP (\$b862) somit nicht vorgesehen; die ausführliche Besprechung bezieht sich somit nur auf den Nutzen der Routine, da die Beschreibung »Exponenten gleichmachen« sehr allgemein ist und sicher Unklarheit stiften könnte.

ADDMEM (\$b867): Erhöhung des FAC um den Wert einer MFLPT-Konstanten

Diese Routine kopiert eine durch A/Y angegebene MFLPT-Konstante in den ARG, so daß sie bei der anschließenden Ausführung von ADDFAC (\$b86a) zum FAC addiert wird, wobei das Ergebnis wieder im FAC steht.

ADDFAC (\$b86a): Addition des ARG zum FAC, Ergebnis in FAC

Außer den Summanden in ARG und FAC ist vor dem Aufruf der Routine das Exponentenbyte des FAC zu laden, damit das Zero-Flag darüber Auskunft gibt, ob der FAC den Wert 0 enthält ($Z=1$) oder nicht ($Z=0$). Bei einer Addition ist 0 bekanntlich neutral, d.h. » $x+0$ « liefert immer » x «. Dies wird von ADDFAC (\$b86a) sofort erkannt, um diesen Sonderfall vereinfacht zu behandeln:

```
FAC := FAC + ARG    [FAC = 0]
FAC := 0 + ARG
FAC := ARG
```

Diese Operation »FAC:=ARG« wird durch »jmp movaf« viel einfacher bewirkt als durch eine Addition einiger Nullbytes, der noch eine zeitaufwendige Exponentenangleichung vorausgehen müßte.

Vergessen Sie also nicht, das Zero-Flag immer gemäß dem FAC-Exponenten zu setzen, bevor ADDFAC (\$b86a) aufgerufen wird. Ein fälschlich gesetztes Z-Flag würde schließlich die Addition verfälschen; es ist hingegen nie falsch, mit gelöschtem Z-Flag einzusteigen, da dann in jedem Fall richtig gerechnet wird – bei FAC=0/Z=0 jedoch wird Rechenzeit vergeudet.

Der umgekehrte Sonderfall (ARG = 0) wird ebenfalls recht schnell erkannt: In diesem Fall wird über RTS zurückgesprungen, weil das Ergebnis bereits im FAC steht:

```
FAC := FAC + ARG    [ARG = 0]
FAC := FAC + 0
FAC := FAC          [erübrigt sich]
```

Kommen wir nun auf den Normalfall (FAC und ARG ungleich Null) zu sprechen. Dazu wird zunächst das aktuelle Rundungsbyte in \$56 als Wert des letzten eingebundenen Rundungsbytes vermerkt. Als nächstes wird der bereits erwähnte Sonderfall »ARG = 0« ausgesondert. Dann kommt es zur Bildung der Exponentendifferenz (Exponent des ARG – Exponent des FAC). Sind die beiden Exponenten identisch (Differenz 0), so erfolgt die weitere Behandlung mit der Addition der Mantissen; andernfalls werden vorher die Ex-

ponenten angeglichen, was durch Links- oder Rechtsverschiebung der Mantissenbytes geschieht.

Spätestens dann werden die Mantissen von FAC und ARG addiert; liegen jedoch ungleiche Vorzeichen vor, so wird subtrahiert. Anschließend wird der FAC wieder normalisiert (NORMAL-Einsprung bei \$b8d7), wozu er bitweise verschoben wird. Durch die Normalisierung wird die Exponentenangleichung rückgängig gemacht.

Daraufhin erfolgt die Rückgabe des Ergebnisses.

NORMAL (\$b8d7): Normalisierung des FAC

»jsr normal« sorgt dafür, daß der FAC normalisiert wird, d.h. die Mantisse liegt danach wieder im Bereich von »0,5« bis »1« (bzw. »-1« bis »-0,5«).

NORMAL (\$b8d7) hebt somit sämtliche vorher erfolgten Exponentenangleichungen auf, die zur Durchführung einer Rechenoperation erforderlich waren.

SQUEEZ (\$b936): Additionsübertrag behandeln

In diesem Teil von ADDFAC (\$b86a) wird ein eventueller Übertrag behandelt, indem der Exponent erhöht und die Mantisse rechtsverschoben wird. Löst dies das Übertragsproblem jedoch nicht, so ist wohl oder übel ein OVERFLOW ERROR mitzuteilen.

\$b947: Invertierung des FAC

Diese Hilfsroutine invertiert Vorzeichenbyte und Mantissenbytes des FAC.

\$b97e: Einsprung für OVERFLOW ERROR

Hier wird der Fehlercode des OVERFLOW ERROR geladen und der Fehlereinsprung aufgerufen.

\$b983: byteweise Rechtsverschiebung des RES

Der Resultsakkumulator für die Multiplikation (\$0026–\$0029) wird durch »jsr \$b983« um 1 Byte rechtsverschoben sowie um eine bestimmte Anzahl von Bits, die im Akkumulator als negativer Wert zur Initialisierung des Bitzählers übergeben wird.

Bei \$b985 kann mit einem beliebigen Offset im X-Register eingestiegen werden, der jedoch nicht die Zeropage-Adresse des zu verschiebenden Fließkomma-Akkumulators ist, sondern diese Adresse minus 1.

Soll nur die bitweise Rechtsverschiebung erfolgen, wird erst bei SHIFTR (\$b999) eingestiegen:

SHIFTR (\$b999):

Rechtsverschiebung eines Fließkomma-Akkus

Im Akku ist der Bitzähler-Initialisierungswert zu übergeben und im X-Register der Offset zum Fließkomma-Akkumulator (Zeropage-Adresse minus 1).

ROLSHF (\$b9b0): Rechtsverschiebung

Auch dieser Einsprung kann genutzt werden; hier ist das Y-Register der Bitzähler.

\$b9bc–\$b9e9: Speicherkonstanten für die LOG-Funktion

Diese Speicherkonstanten werden von der Routine zur LOG-Funktion verwendet. Bei \$b9c1 steht eine Polynomtabelle dritten Grades.

LOG \$b9ea: Routine zur Basic-Funktion LOG

Diese Funktionsroutine kann auch von eigenen Programmen aus verwendet werden; sie ermittelt zum FAC den dazugehörigen LOG-Wert (Logarithmus) zur Basis 2.

Zur Berechnung der Logarithmen zu anderen Basiswerten als 2 ist ein altes Rechengesetz anwendbar, das ich hier kurz erwähnen möchte, auch wenn viele von Ihnen dieses (noch) aus der Schule kennen dürften:

$$\log_a b = \frac{\log(b)}{\log(a)}$$

Sprich: Der Logarithmus von »b« zur Basis »a« entspricht dem Quotienten aus dem Zweierlogarithmus von »b« und demjenigen von »a«. Es muß sich dabei nicht einmal um einen Zweierlogarithmus handeln (auch der Quotient zweier Zehnerlogarithmen wäre verwendbar), doch unser C64 bietet uns nun einmal diesen an.

MEMMULT (\$ba28):

FAC mit MFLPT-Konstante multiplizieren, Ergebnis in FAC

Dieser Einsprung bereitet die Multiplikation durch MULT (\$ba2b) vor, indem eine MFLPT-Konstante als Faktor in den ARG kopiert wird. Darauf folgt die Ausführung von MULT (\$ba2b), wo der FAC mit dem ARG multipliziert wird.

MULT (\$ba2b):

FAC mit ARG multiplizieren, Ergebnis in FAC

Wie bei ADDFAC (\$b86a) ist auch bei MULT (\$ba2b) das vorherige Laden des FAC-Exponenten zwecks Prüfung auf \$00 erforderlich. Enthält der FAC den Wert 0, so wird über RTS zurückgesprungen, da eine Multiplikation »0*x« immer »0« ergibt:

```
FAC := FAC * ARG      [FAC = 0]
FAC := 0 * ARG
FAC := 0               [FAC ist bereits 0!]
```

Andernfalls werden zunächst die Exponenten addiert; stellt sich dabei heraus, daß der ARG 0 enthält, so wird durch Stapelmanipulation an diejenige Stelle, die MULT (\$ba2b) aufgerufen hat, mit dem Ergebnis 0 zurückgekehrt, das vorher in den FAC geschrieben wird.

Im Normalfall jedoch führt die Exponentenaddition dazu, daß der Ergebnis-Exponent vorausberechnet wird. Daraufhin wird der RES

(Ergebnis-Fließkomma-Akkumulator) mit Nullbytes initialisiert; er besteht nur aus 4 Bytes, da er lediglich die Mantisse des Ergebnisses beinhaltet. In diesen RES werden dann das Rundungsbyte sowie die vier Mantissenbytes hineinmultipliziert, wodurch er laufend erhöht wird. Anschließend wird die Resultatsmantisse in die FAC-Mantisse kopiert und der FAC normalisiert, da die Exponentenaddition nicht gewährleistet, daß die Mantisse im Bereich von »0,5« bis »1« (bzw. »–1« bis »–0,5«) steht.

Das angesprochene »Hineinmultiplizieren« der Mantissen läuft in der folgenden Unteroutine ab:

MLTPPLY (\$ba59):

Byte aus Akku in RES hineinmultiplizieren

Wird \$00 hineinmultipliziert, so erfolgt lediglich eine Rechtsverschiebung des RES, um die Übergabe eines vorhandenen Mantissenbytes zu berücksichtigen (andernfalls wäre die Wertigkeit der später hineinmultiplizierten Bytes vernachlässigt).

Bei anderen Werten als \$00 werden die FAC- zu den ARG-Mantissen addiert und daraufhin der RES um 1 Bit nach rechts verschoben; der übergebene Akkumulator dient also nicht als Summand, sondern nach einer kleinen Zwischenbearbeitung als Bitzähler für die Verschiebung.

MOVMA (\$ba8c): MFLPT-Konstante in ARG kopieren

Eine MFLPT-Konstante ab der in A/Y enthaltenen Adresse kopiert die MOVMA-Routine in den ARG. Die Adresse wird dazu im Hilfszeiger \$22/\$23 abgelegt und das Y-Register im weiteren Verlauf als Offset auf das jeweilige Übertragungsbyte eingesetzt; das X-Register behält seinen Wert.

Die Mantissenbytes 2–4 sowie das Exponentenbyte können ohne jegliche Vorkehrungen unmittelbar übertragen werden, doch bei Mantisse #1 wird noch das Vorzeichenbit ausgesondert und in \$6e (Vorzeichenbyte des ARG) abgelegt sowie das Vorzeichenvergleichsbyte (\$6f) angepaßt.

Ein 1-Bit ersetzt dabei das ausgesonderte Vorzeichenbit.

Zuletzt lädt MOVMA (\$ba8c) das FAC-Exponentenbyte, damit direkt nach dem Aufruf »jsr movma« solche Routinen wie ADDFAC (\$b86a) oder MULT (\$ba2b), die das Laden des Exponenten erfordern, ausführbar sind.

\$bab7: Unteroutine

zur Addition der Exponenten von FAC und ARG

Dies ist eine Unteroutine zur Berechnung des Exponenten eines Produktes (Multiplikationsergebnis).

Die Leistung der Routine besteht jedoch noch aus drei weiteren Punkten:

- Das Vorzeichen des Produktes wird berechnet und in \$66 gemerkt.

- Falls der ARG den Wert 0 enthält (Exponent des ARG = 0), so wird diejenige Routine, die »jsr \$bab7« ausgeführt hat, durch Stapelmanipulation ignoriert und sogleich die übergeordnete Routine angesprungen, wobei vorher durch einen Einsprung in die Additionsroutine der FAC auf 0 gesetzt wird.
- Ein Überschreiten des zulässigen Exponentenbereiches wird durch OVERFLOW ERROR quittiert.

FACM10 (\$bae2): FAC verzehnfachen

Im Gegensatz zu Routinen wie ADD0.5 (\$b849), die lediglich den Zeiger auf eine Konstante laden, um eine Rechenoperation mit dem FAC und einer Konstanten durchzuführen, ist FACM10 (\$bae2) eine eigenständige Multiplikationsroutine, die auf den Sonderfall »FAC := FAC * 10« getrimmt ist.

Sie ist speziell für diesen Sonderfall geschaffen worden und meistert ihn viel schneller als eine MULT-Multiplikation mit 10.

Dazu wird folgender Rechenweg eingeschlagen, an dessen mathematischer Korrektheit kein Zweifel anzubringen ist:

$$\text{FAC} := \text{FAC} * 10 = ((\text{FAC} * 4) + \text{FAC}) * 2$$

Beweis:

$$((\text{FAC} * 4) + \text{FAC}) * 2 = (\text{FAC} * 5) * 2 = \text{FAC} * 5 * 2 = \text{FAC} * 10$$

In Rechenschritten ausgedrückt, geht FACM10 (\$bae2) so vor:

1. FAC := FAC * 4

Dazu wird einfach der Exponent zur Basis 2 um 2 erhöht ($2 \uparrow 2 = 4$), was eine Multiplikation mit 4 ersetzt. Dies erscheint vielleicht auf den ersten Blick uneinsichtig, aber nach demselben Prinzip funktioniert auch der Rechenweg, zur Vertausendfacherung einer Zahl einfach 3 Nullen anzuhängen (»35 * 1000 = 35000«), da 1000 durch $10 \uparrow 3$ darstellbar ist.

2. FAC := neuer FAC + alter FAC

Dadurch wird zum vervierfachen FAC der alte FAC-Inhalt addiert. Vor der FAC-Vervierfacherung wurde dessen Inhalt in den ARG gerettet und kann jetzt durch einen Einstieg in ADDFAC (\$b86a) addiert werden.

Nach diesem Schritt ist der FAC bereits fünfmal so hoch wie zu Beginn der Routine.

3. FAC := FAC * 2

Nach diesem Schritt ist der Rechengang abgeschlossen, da der FAC insgesamt verzehnfacht wurde.

Auch diese Multiplikation mit einer Zweierpotenz geschieht durch Ändern des Exponenten: Er wird um 1 erhöht, da 2 der Faktor und $2 \uparrow 1 = 2$ ist.

Ein möglicher OVERFLOW ERROR wird hier für den Fall erkannt, daß der Exponent bei Inkrementierung den Wert \$ff überstiegen hat.

\$baf9: MFLPT-Konstante 10

Die Zahl 10 ist hier für FACD10 (\$baf9) als MFLPT-Konstante abgelegt. Die FACM10-Routine (\$bae2) hingegen verwendet diese Konstante überhaupt nicht!

FACD10 (\$baf9):

FAC durch 10 dividieren, Ergebnis in FAC

Hier liegt wieder ein Einsprung wie ADD0.5 (\$b849) vor, der eine andere Rechenroutine vorbereitet, indem er eine Konstante als Argument lädt. FACD10 (\$baf9) überträgt den FAC als Dividend (Wert, der dividiert wird) in den ARG und lädt stattdessen die Konstante 10 als Divisor (Wert, durch den geteilt wird) in den FAC, so daß anschließend bei Ausführung von DIVAF (\$bb12) derjenige Wert, der bei Aufruf von FACD10 (\$baf9) im FAC stand und dann in den ARG übertragen wurde, durch 10 dividiert wird. Das Ergebnis kommt wieder in den FAC.

Das Vorzeichenvergleichsbyte wird von FACD10 (\$baf9) vor der Division in jedem Fall auf 0 gesetzt.

DIVMF (\$bb0f): Division einer

MFLPT-Konstanten durch den FAC, Ergebnis in FAC

Dieser Einsprung lädt vor Ausführung von DIVAF (\$bb12) den Dividenten aus dem Speicher der MFLPT-Konstanten in den ARG.

FACD10 (\$baf9) springt nicht an dieser Stelle ein, sondern erst bei folgender Adresse:

DIVAF (\$bb12):

Division des ARG durch den FAC, Ergebnis in FAC

Wie ADDFAC (\$b86a) und MULT (\$ba2b), wird das Laden des FAC-Exponenten vorausgesetzt. Sollte sich dadurch herausstellen, daß der FAC (Divisor!) 0 ist, so führt dies zu einem DIVISION BY ZERO ERROR, da eine Division durch 0 nicht auf konventionelle Weise zu lösen ist – und auf gar keinen Fall innerhalb des Fließkomma-Wertebereiches!

Dann wird der Ergebnis-Exponent mittels Subtraktion (Exponent des Dividenten – Exponent des Divisors) ermittelt; um die \$bab7-Hilfsroutine nutzen zu können, multiplizierte DIVAF (\$bb12) deshalb zuvor den FAC-Exponenten mit –1, damit die Addition der beiden Exponenten einer Subtraktion des FAC-Exponenten vom ARG-Exponenten entspricht.

Anschließend wird effektiv die FAC-Mantisse durch die ARG-Mantisse dividiert; der RES (Ergebnis-Fließkomma-Akkumulator) dient dabei als Hilfsspeicher.

\$bb8a: Einsprung für DIVISION BY ZERO ERROR

»jmp \$bb8a« löst einen DIVISION BY ZERO ERROR aus.

MOVRF (\$bb8f): RES in FAC übertragen

Der Ergebnis-Fließkomma-Akkumulator für Multiplikation und Division wird von dieser Routine in den FAC übertragen. Dabei ist zu beachten, daß der RES nur aus 4 Mantissenbytes besteht und keinen Exponenten hat.

Nach der Übertragung der Mantisse erfolgt eine FAC-Normalisierung, um die Mantisse in den zulässigen Bereich von »0,5« bis »1« beziehungsweise »-1« bis »-0,5« zu bringen.

MOVMF (\$bba2): MFLPT-Konstante in FAC übertragen

Eine MFLPT-Konstante, deren Adresse in A/Y steht, überträgt »jsr movmf« in den FAC. Dabei werden die Mantissenbytes 2–4 sowie das Exponentenbyte unverändert aus der MFLPT-Konstanten in den FAC übernommen, während aus Mantisse #1 noch das Vorzeichenbit nach \$66 ausgesondert und durch ein 1-Bit ersetzt werden muß.

Abschließend löscht MOVMF (\$bba2) das FAC-Rundungsbyte, da ein aus dem Speicher übertragener Wert immer als »rundungsfrei« betrachtet wird.

MOVT4 (\$bbc7): FAC #1 in FAC #4 übertragen

Um den FAC in den FAC-Hilfsspeicher ab \$005c zu übertragen, ist MOVT4 (\$bbc7) aufzurufen. Dieser Einsprung besteht aus dem Laden der Zieladresse sowie der Ausführung von MOV TZ (\$bbcc), wozu MOV T3 (\$bbca) übersprungen wird.

Letztlich stützt sich MOVT4 (\$bbc7) auf MOVFM (\$bbd4).

MOVT3 (\$bbca): FAC #1 in FAC #3 übertragen

Entspricht MOVT4 (\$bbc7), bezieht sich allerdings auf FAC #3 (ab \$0057).

MOV TZ (\$bbcc):**FAC in Zeropage-Adresse (X-Register) übertragen**

Um den FAC an eine Zeropage-Adresse, die im X-Register übergeben wird, zu übertragen, genügt »jsr movtz«. Der FAC wird bei der Übertragung ins MFLPT-Format gewandelt.

Letztlich basiert MOV TZ (\$bbcc) auf MOVFM (\$bbd4) und leistet lediglich das Laden des Y-Registers (HB der Zieladresse) mit \$00, dem High-Byte jeder Zeropage-Adresse.

FACVAR (\$bbd0): FAC in aktuelle Variable übertragen

Die aktuelle Variablenadresse steht immer in \$49/\$4a; um den FAC dorthin zu übertragen, wird bei FACVAR (\$bbd0) eingesprungen. FACVAR (\$bbd0) lädt X/Y mit dem Inhalt des Variablenzeigers \$49/\$4a und veranlaßt die im Speicher folgende MOVFM-Routine zur Ausführung.

MOVFM (\$bbd4): FAC an Adresse in X/Y übertragen

Diese Routine rundet zunächst den FAC und richtet dann den Hilfszeiger \$22/\$23 auf die Zieladresse, die in X/Y übergeben

wurde. Dann werden die Mantissenbytes 2–4 sowie das Exponentenbyte unverändert übertragen; in Mantisse #1 hingegen wird als Bit 7 das Vorzeichenbit eingebunden (Konvertierung ins MFLPT-Format!).

MOVAF (\$bbfc): ARG in FAC übertragen

Da sowohl der FAC als auch der ARG im FLPT-Format aufgebaut sind, handelt es sich hierbei lediglich um eine Speicherverschiebung des ARG in den FAC, wobei die Bytes \$61–\$64 (Mantissen und Exponent) in einer Schleife mit den Werten aus \$69–\$6d geladen werden und das Vorzeichenbyte einzeln von \$6e nach \$66 übertragen wird.

Anschließend wird das FAC-Rundungsbyte gelöscht.

MOVFA (\$bc0c): FAC in ARG übertragen

Dies ist das exakte Gegenstück zu MOVAF (\$bbfc). Da auch der programmtechnische Grundaufbau derselbe ist, erübrigt sich eine weitere Beschreibung.

ROUND (\$bc1b): FAC runden

Zur Einbindung eines Rundungsbytes in den FAC dient ROUND (\$bc1b). Dort wird bei einem FAC-Inhalt 0 ohne weitere Bearbeitung durch RTS zurückgesprungen, bei anderen Werten jedoch wird das Rundungsbyte geprüft. Ist dieses gelöscht, erfolgt ebenfalls ein Rücksprung. Andernfalls wird die FAC-Mantisse um 1 Bit erhöht; entsteht dabei ein Übertrag, muß der Exponent ebenfalls erhöht werden.

SIGN (\$bc2b): Vorzeichen des FAC in Akku holen

Diese Routine entspricht in ihrer Wirkung der mathematischen SIGNUM-Funktion, die zu einer Zahl einen Vorzeichenfaktor ermittelt:

SIGN (positive Zahl) = 1
SIGN (0) = 0
SIGN (negative Zahl) = -1

Diese Werte werden nach »jsr sign« für den FAC ermittelt und im Akkumulator zurückgegeben; »-1« wird dabei durch »\$ff« dargestellt.

Die CPU-Flags sind gemäß dem zurückgegebenen Akku-Inhalt gesetzt. Zusammengefaßt liegen also für die drei verschiedenen Fälle folgende Werte vor:

positiv: Akku = \$01, Z = 0, N = 0
Null: Akku = \$00, Z = 1, N = 0
negativ: Akku = \$ff, Z = 0, N = 1

SGN \$bc39: Routine zur Basic-Funktion SGN

Die SGN-Funktion in Basic wird in der Mathematik als SIGNUM-Funktion (sign) bezeichnet. Sie ermittelt einen Vorzeichenfaktor für

eine Zahl: Positive Zahlen ergeben »1«, negative ermitteln »-1« und Null wird durch »0« bezeichnet.

Dies wird programmtechnisch größtenteils durch den Aufruf der SIGN-Routine (\$bc2b) bewältigt, die das Ergebnis im Akkumulator als vorzeichenbehafteten Bytewert zurückgibt. SGN (\$bc39) muß nur noch diesen Bytewert in eine Fließkommazahl konvertieren lassen. Innerhalb der dafür verantwortlichen Befehle liegen drei anderweitig genutzte Einsprünge zur Konvertierung ins Fließkommaformat:

WRDFAC (\$bc44):

2-Byte-Wert mit Vorzeichen in FAC bringen

BINFAC (\$bc49):

2-Byte-Wert umwandeln, Mantisse #3 und #4 löschen

SETFAC (\$bc4f): X-Register als

Exponent setzen, Akku zum Löschen von Rundung und Vorzeichen verwenden, Zahl aus \$62/\$63 konvertieren

ABS \$bc58: Routine zur Basic-Funktion ABS

Der Absolutwert (Betrag) einer Zahl wird am einfachsten durch Entfernen des Vorzeichens ermittelt. Um die ABS-Funktion für den FAC auszuführen, genügt deshalb eine Rechtsverschiebung des Vorzeichenbytes, wodurch das Bit 7 (Vorzeichenbit) von seiner eigentlichen Position entfernt und durch eine binäre Null ersetzt wird.

CMPFAC (\$bc5b): FAC mit Wert aus Speicher vergleichen

Mit einer MFLPT-Konstanten, deren Adresse in A/Y übergeben wird, vergleicht diese Routine den aktuellen FAC-Inhalt. Das Vergleichsergebnis ist danach im Akkumulator zu finden:

\$00 : FAC = Konstante

\$01 : FAC > Konstante

\$ff : FAC < Konstante

Es handelt sich hier um dieselben drei Bytewerte, die auch die SIGN-Routine (\$bc2b) zurückgibt. Dies wird von CMPFAC (\$bc5b) ausgenutzt, um auf den Sonderfall »Konstante = 0« möglichst schnell reagieren zu können. Im Grunde ist die Byte-Darstellung des Vergleichsergebnisses nämlich nur das Vorzeichen des Subtraktionsergebnisses von »FAC-Konstante«:

\$00 : FAC-Konstante = 0

\$01 : FAC-Konstante > 0

\$ff : FAC-Konstante < 0

Für den Spezialfall, daß der FAC mit der Konstanten 0 verglichen werden soll, bedeutet dies, daß das Vorzeichen von »FAC-Konstante«, also »FAC-0 = FAC« ermittelt werden soll; und dies erledigt SIGN (\$bc2b).

In normalen Situationen ($FAC < > 0$) wird mittels CMP-Ver- gleich das Vergleichsergebnis ermittelt.

FACINT (\$bc9b): FAC in Integerzahl umwandeln

Diese Routine wandelt den FAC-Inhalt in eine Integerzahl um, wobei die Nachkommastellen bei der Konvertierung zwangsläufig verlorengehen. Das wirklich Besondere an dieser Konvertierung ist jedoch die Tatsache, daß das 4 Byte umfassende Ergebnis in den Mantissenbytes (\$62-\$65) nicht im Low-High-Format zurückgegeben wird, sondern genau in anderer Richtung: \$62 enthält also das höchstwertige Byte und \$65 das niedrigstwertige.

Die Funktionsweise ist übrigens äußerst einfach: Die FAC-Mantisse wird solange rechtsverschoben, bis der Exponent 0 erreicht ist; dann steht ja das Ergebnis bereits in den Mantissenbytes, da die Mantisse nichts anderes als eine 4-Byte-Integerzahl ist.

INT \$bccc: Routine zur Basic-Funktion INT

Die Routine zur Basic-Funktion INT stützt sich größtenteils auf FACINT (\$bc9b).

STRFLP (\$bcf3):

ASCII-String in Fließkommaformat umwandeln

Die ASCII-Darstellung einer Zahl wird durch »jsr strflp« ins Fließkommaformat in den FAC gebracht. Dazu muß der CHRGET-Zeiger auf den Zahlenstring weisen und das erste Zeichen sollte zuvor bereits über CHRGET eingelesen worden sein.

Dabei werden alle Möglichkeiten der Zahlendarstellung berücksichtigt, also auch Vorzeichen »+«, »-«, Dezimalpunkte ».« und Exponentialangaben »E«. Sobald das erste ungültige Zeichen auftritt, wird die Konvertierung beendet.

Entscheidend sind die Hilfspeicher \$5e (Exponentialzähler) und \$5f (Dezimalstellenzähler). Der Exponentialzähler ergibt sich aus der Anzahl der Stellen vor dem Komma sowie der »E«-Angabe, während der Dezimalstellenzähler aus der Anzahl der Nachkommastellen hervorgeht. Beide Zähler werden bis zum Ende laufend aktualisiert und dann insofern behandelt, als der Wert gemäß dem Exponentialzähler mit 10 multipliziert und gemäß dem Dezimalstellenzähler durch 10 dividiert wird.

Zum »Hineinaddieren« einer neuen Ziffer dient dabei die ADDAFC-Routine:

ADDAFC (\$bd7e): Bytewert zum FAC addieren

Diese Routine addiert einen Bytewert (im Akkumulator zu übergeben) zum FAC.

\$bdb3-\$bdc1: MFLPT-Konstanten

zur Umwandlung des FAC in einen ASCII-String

In diesen Speicherzellen liegen Grenzwerte bei der Umwandlung vor, die jeweils möglichst nah an einer Zehnerpotenz liegen.

LINOUT (\$bdc2): Ausgabe der aktuellen Zeilennummer

Bei der Fehlerbehandlung wird hinter der Meldung »... ERROR« im Programm-Modus auch die Fehlerzeile durch »IN ...« erwähnt. Dies erledigt LINOUT (\$bdc2); dazu wird zunächst der Text »IN« über STROUT (\$able) gedruckt und anschließend die aktuelle Basic-Zeilenummer für die im Speicher anschließende NUMOUT-Routine als Ausgabewert geladen.

NUMOUT (\$bdcd): Integerzahl aus X/A ausgeben

Dieser Einsprung gibt die in X-Register und Akkumulator enthaltene vorzeichenlose Integerzahl als ASCII-Text aus. Dazu wird zuerst die Integerzahl ins Fließkommaformat in den FAC gebracht und von dort aus in den ASCII-Code umgewandelt, wofür FLPSTR (\$bddf) herangezogen wird. Abschließend erfolgt die Ausgabe der ASCII-Darstellung über STROUT (\$able).

\$bddd: Einsprung zur Umwandlung des FAC in einen String ohne Vorzeichenberücksichtigung

Dieser Einsprung lädt \$01 als Offset, um dadurch das Vorzeichen zu ignorieren. Im Speicher folgt FLPSTR (\$bddf).

FLPSTR (\$bddf): FAC als ASCII-String ab \$0100 ablegen

Zur Vorbereitung von FLPSTR (\$bddf) ist eine Fließkommazahl in den FAC zu laden und der Offset im Y-Register vorzubereiten; normalerweise wird mit \$00 begonnen, um ab \$0100 den String abzulegen. Vor die Darstellung der Zahl wird dabei entweder ein Minus-Zeichen (»-«) bei negativen oder ein führendes Leerzeichen bei positiven Zahlen geschrieben.

In A/Y wird die Adresse \$0100, ab welcher sich das Konvertierungsergebnis befindet, zurückgegeben. Dadurch ist eine sofortige Textausgabe durch »jsr strout« möglich.

\$bf11–\$bf15: MFLPT-Konstante für die SQR-Funktion

Die Konstante 0.5 steht hier für die SQR-Funktion (als Exponent) bereit.

\$bf16–\$bf39: Mantissen für FLPSTR

In dieser Tabelle gehören je vier Bytes zusammen; es handelt sich um die Darstellungen von Mantissen (4 Bytes) für einige Zehnerpotenzen.

\$bf3a–\$bf51: Mantissen für TISTR

Diese Tabelle entspricht im Aufbau der Tabelle \$bf16–\$bf39, wird jedoch bei der Umwandlung einer Zeitkonstanten (TI) in einen ASCII-String verwendet.

\$bf52–\$bf70: Füllbytes

Diese Füllbytes sind bei manchen Versionen des C64-ROMs unterschiedlich, doch hat dies keinerlei Bedeutung, weil diese Füllbytes nicht angesprochen werden.

Hier befinden sich aber auf gar keinen Fall Programmteile des Basic-Interpreters; deshalb gibt es einige wenige Erweiterungen des Basic-Interpreters, die diesen ins RAM kopieren und an den Speicherplätzen \$bf52–\$bf70 ausführbare Befehle ablegen.

SQR \$bf71: Routine zur Basic-Funktion SQR

Die Quadratwurzel (Basic-Funktion SQR) ist laut mathematischer Definition durch eine Potenzierung ersetzbar:

$$\text{SQR}(X) = X \uparrow (1/2) = X \uparrow 0.5$$

Deshalb hat SQR (\$bf71) keine eigene Berechnungsroutine, sondern lädt lediglich die Konstante 0.5 (ab \$bf11 im Speicher) als Exponent und läßt die Potenzierung durchführen.

MEMPOT (\$bf78):**ARG hoch Konstante berechnen, Ergebnis in FAC**

Die Adresse des Exponenten wird in A/Y, die Basis im ARG übergeben. Die Konstante kommt zur Vorbereitung von POTAFAC (\$bf7b) in den FAC, welcher am Schluß das Ergebnis der Potenzierung erhält.

POTAFAC (\$bf7b):**ARG hoch FAC berechnen, Ergebnis in FAC**

Die Potenz, deren Basis im ARG und deren Exponent im FAC steht, berechnet POTAFAC (\$bf7b). Dabei werden zwei Sonderfälle beachtet:

- Ist der Exponent 0, so ist das Ergebnis 1. Dazu wird in die EXP-Routine verzweigt, die dieses Resultat relativ schnell ermittelt.
- Ist die Basis 0, so muß als Ergebnis 0 zurückgegeben werden.

\$bfbf–\$bfec: Konstanten für die EXP-Routine

Hier steht die MFLPT-Darstellung von 1.44269504, also dem Wert

$$1/\text{LOG}(2)$$

Dahinter befindet sich eine Polynomtabelle siebten Grades.

EXP \$bfed: Routine zur Basic-Funktion EXP

Die Routine zur EXP-Funktion ist aus technischen Gründen zweigeteilt. Der erste Teil liegt bei \$bfed–\$bfff, also am Ende des Basic-ROMs; die Routine wird jedoch bei \$e000–\$e042 fortgesetzt.

\$e000: Fortsetzung der Routine zur Basic-Funktion EXP

Im Fortsetzungsteil von EXP (\$bfed) ist vielleicht die Schleife bei \$e01e–\$e028 ein lehrreiches Beispiel zur Vertauschung der Inhalte von FAC und ARG auf möglichst einfache Weise.

POLYX (\$e043):**Polynomauswertung für Polynomtabelle und FAC**

Im FAC muß ein X-Wert und ab A/Y eine Polynomtabelle stehen. Dann wird die Polynomtabelle für X^2 berechnet (Aufbau der Polynomtabelle siehe POLY \$e059) und das Ergebnis abschließend mit dem Ausgangswert für X multipliziert; effektiv wird also folgendes Polynom berechnet:

$$\begin{aligned} & x * (a0 * (x^2)^0 + a1 * (x^2)^1 + \\ & a2 * (x^2)^2 + a3 * (x^2)^3 + \dots) = \\ & x * (a0 + a1 * x^2 + a2 * x^4 + a3 * x^6 + \dots) = \\ & a0 * x + a1 * x^3 + a2 * x^5 + a3 * x^7 + \dots \end{aligned}$$

POLY (\$e059):**Anwendung einer normalen Polynomtabelle auf den FAC**

Ein reguläres Polynom der Form

$$a0 * x^0 + a1 * x^1 + a2 * x^2 + a3 * x^3 + \dots$$

wird durch »jsr poly« auf den FAC angewendet. Dabei wird der X-Wert im FAC übergeben und die Adresse der Polynomtabelle in A/Y mitgeteilt. Die Polynomtabelle beginnt mit dem Polynomgrad (höchste Koeffizientennummer) als Bytewert, wonach die MFLPT-Darstellungen der Koeffizienten in deren Reihenfolge im Speicher stehen.

\$e08d: MFLPT-Konstanten für die Funktion RND

Diese beiden Konstanten werden für die Berechnung einer Pseudo-Zufallszahl verwendet.

RND \$e097: Routine zur Basic-Funktion RND

Das Argument der RND-Funktion ist bekanntlich kein Dummy-Wert, sondern nimmt unmittelbar auf das Funktionsergebnis Einfluß. Dabei entscheidet das Vorzeichen des RND-Argumentes über die grundsätzliche Behandlung. Es werden nach der Prüfung mit SIGN (\$bc2b) folgende drei verschiedenen Fälle identifiziert:

1. Argument ist 0

Dann wird das Funktionsergebnis aus den CIA-1-Timern A und B entnommen. Aus den vier auf diese Weise erhaltenen Bytes wird eine Mantisse gebildet, die bei einem Exponentenbyte von \$80 irgendwo zwischen 0 und 1 liegt. Somit handelt es sich um einen pseudo-zufälligen Fließkommawert, der in keiner Weise von den vorher ermittelten Zufallszahlen abhängt.

2. Argument ist positiv

Hier trägt der sogenannte SEED-Wert Bedeutung. Als SEED-Wert wird der letzte RND-Rückgabewert gespeichert.

Im Falle eines positiven RND-Argumentes wird der SEED-Wert in den FAC geholt und mit 11879546 multipliziert sowie um 3.92767774E–08 erhöht. Die Timer haben keinen Einfluß auf das Funktionsergebnis, welches nun durch Vertauschen der Mantissenbytes entsteht, weil die Behandlung für Fall 3 (negatives Argument) folgt.

3. Argument ist negativ

Bei einem negativen Argument geht das Ergebnis aus dem Argument durch die Vertauschung der Mantissenbytes sowie das »gewaltsame« Setzen des Exponentenbytes \$80 und des positiven Vorzeichens hervor. Deshalb ergibt beispielsweise RND(–5) immer »3.73720468e–08«; es ist also nicht einmal eine Pseudo-Zufallszahl, weshalb eigentlich nur RND(–TI) Sinn trägt, da sich das Argument TI ständig ändert und somit bei jedem Aufruf ein anderer Funktionswert berechnet wird.

Da der Funktionswert als SEED gesetzt wird, dient das soeben erwähnte Beispiel RND(–TI) meist auch nur zum Initialisieren des SEED, während an späteren Stellen im Programm lediglich RND(0) erforderlich ist.

STRNEX (\$e0e3): Rückgabe**der im FAC stehenden Mantisse als Funktionsergebnis**

Diese Schlußbehandlung von RND setzt den Exponent auf \$80 (Bereich von »0« bis »1« bzw. »–1« bis »0«) sowie das Vorzeichen auf »positiv«. Unabhängig von der Mantisse ist das Ergebnis also ein positiver Wert zwischen 0 und 1. Das Ergebnis wird nicht nur im FAC zurückgegeben, sondern auch als SEED-Wert gemerkt.

EREXIT (\$e0f9):**Fehlerbehandlung nach I/O-Operationen des Interpreters**

Dieser Einsprung wird nur von \$e1d1 über »jmp« genutzt; normalerweise verzweigen die Basic-Kernal-Aufrufe hierher über »bcs«, wenn ein I/O-Fehler aufgetreten ist. Hier wird dann anhand des Akkumulators (Fehlercode) die richtige Fehlermeldung erzeugt und über den ERROR-Einsprung aufgerufen.

Es folgen die Basic-Kernal-Einsprünge:

Basic-BSOUT (\$e10c): Teil der BBSOUT-Routine**Basic-BASIN (\$e112):****Basic-Kernal-Aufruf von BASIN (\$ffcf)****Basic-CKOUT (\$e118): ruft spezielle BCKOUT-Routine auf**

BCHKIN (\$e11e): Basic-Kernal-Aufruf von CHKIN (\$ffc6)**BGETIN (\$e124): Basic-Kernal-Aufruf von GETIN (\$ffe4)****SYS \$e12a: Routine zum Basic-Befehl SYS**

Diese Routine liest die SYS-Zieladresse nach \$14/\$15 ein. Nach dem Einlesen der Prozessorregister seit dem letzten SYS-Aufruf (Adressen \$030c–\$030f) wird die SYS-Routine über »jmp (\$0014)« angesprungen. Aufgrund der vorher erfolgten Stapelmanipulation handelt es sich dabei jedoch effektiv um einen JSR-Aufruf; aus der durch SYS aufgerufenen Routine wird bei RTS nach \$e147 gesprungen.

Bei \$e147 werden die aus der ausgeführten Routine zurückgegebenen Prozessorregister wieder in \$030c–\$030f geschrieben und es erfolgt ein Rücksprung zur Interpreterschleife.

\$e156: Routine zum Basic-Befehl SAVE

Die Aufgabe der SAVE-Befehlsroutine besteht nur aus der Auswertung der SAVE-Parameter, welche die Unteroutine GETLSV (\$e1d4) übernimmt, sowie dem Aufruf der SAVE-Routine des Kernal (bei \$ffd8). Dieser Routine wird die Endadresse des Basic-Programms als Endadresse des abzuspeichernden Speichers übermittelt. SAVE (\$ffd8) erwartet im Akku die Zeropage-Adresse eines Zeigers, der die Anfangsadresse enthält; im Fall des SAVE-Befehls ist dies \$2b, da in \$2b/\$2c die Basic-Anfangsadresse steht.

Falls ein Ein-/Ausgabe-Fehler bei der SAVE-Abarbeitung auftritt, wird dieser in EREXIT (\$e0f9) behandelt.

Die Adresse \$e15f kann man auch als Basic-Kernal-Aufruf von SAVE (\$ffd8) betrachten; möglicherweise hilft Ihnen das in einem eigenen Programm (Beispiel: Basic-Erweiterung) weiter:

BSAVE (\$e15f): Basic-Kernal-Aufruf von SAVE (\$ffd8)

Genutzt wird dieser Einsprung vom Interpreter jedoch nur in der unmittelbaren Ausführung der SAVE-Befehlsroutine.

VERIFY \$e165: Routine zum Basic-Befehl VERIFY

Diese Routine lädt das Verify-Flag \$01 für die gemeinsame Behandlung der Befehle LOAD und VERIFY, und überspringt den ersten Befehl von LOAD.

\$e168: Routine zum Basic-Befehl LOAD

Hier wird das LOAD-Flag \$00 für die gemeinsame Behandlung von LOAD und VERIFY geladen.

\$e16a:**gemeinsame Behandlung der Befehle LOAD und VERIFY**

Zu Beginn wird das LOAD-/VERIFY-Flag in \$0a (entsprechendes Flag des Basic-Interpreters) gemerkt; \$00 steht für LOAD, \$01 für VERIFY.

Im Anschluß daran liest ein Aufruf von GETLSV (\$e1d4) die Parameter ein, woraufhin die Anfangsadresse des Basic-Programms im Speicher als Ladeadresse eingesetzt und der Lade- bzw. Verifizierungsvorgang durch den Aufruf von LOAD (\$ffd5) durchgeführt wird.

Im Falle eines I/O-Fehlers erfolgt die Fehlerbehandlung, wofür effektiv EREXIT (\$e0f9) zuständig ist.

Die Weiterverarbeitung bei problemlosem Ablauf der Kernal-Routine LOAD (\$ffd5) ist nun für VERIFY und LOAD getrennt:

1. VERIFY

Als erstes wird das Statusflag ST auf das VERIFY-ERROR-Bit (Bit 4) getestet; liegt ein VERIFY ERROR vor, so löst dies die entsprechende Fehlermeldung (Code \$1c) aus. Andernfalls soll die Meldung »VERIFYING OK« ausgegeben werden, und zwar nur im Programm-Modus – so haben es sich die Programmierer jedenfalls gedacht. Leider ist ihnen jedoch ein Fehler unterlaufen, denn anstatt das High-Byte des CHRGET-Zeigers auf 2 (High-Byte des Direkteingabepuffers) zu testen, erfolgt ein Low-Byte-Vergleich. Es wird also bis auf einige Ausnahmefälle immer die Meldung »VERIFYING OK« ausgegeben, aber in ganz speziellen Situationen kann es im Programm-Modus passieren, daß die Meldung fälschlicherweise unterdrückt wird (wenn das CHRGET-Zeiger-Low-Byte auf 2 steht). Die Wahrscheinlichkeit eines solchen Fehlverhaltens ist jedoch lediglich 1:256 und somit vernachlässigbar.

2. LOAD

Sind Ladeschwierigkeiten (nur bei Datensettenbetrieb möglich) aufgetreten, so wird dies anhand des Bit 6 im Statusbyte ST erkannt und die Fehlermeldung LOAD ERROR (Code \$1d) hervorgerufen.

Bei korrektem Laden wird eine weitere Fallunterscheidung getroffen:

2a. LOAD im Direktmodus

Die letzte von LOAD eingelesene Adresse wird zur Endadresse des Basic-Programmspeichers und somit zum Variablenbeginn ernannt. Nach Ausgabe der Meldung »READY« werden die Variablenzeiger neu gesetzt und der Variablenspeicher gelöscht, die Linkpointer berechnet und schließlich der Warmstart ausgelöst.

2b. LOAD im Programm-Modus

Der CHRGET-Zeiger wird auf den Programmspeicher-Anfang gerichtet; dadurch wird beim Rücksprung zur Interpreterschleife die Interpretation am Programmbeginn fortgesetzt. Eine erneute Linkpointerberechnung geht dem Rücksprung in die Interpreterschleife voraus, der durch Aufruf von RESTORE vollzogen wird. Die Interpreterschleife wiederum führt das Programm erneut vom Programmbeginn aus. Somit erklärt sich das Funktionieren eines Ladeprogramms der folgenden Art:

```

10 IF A=0 THEN A=1:LOAD"OBJ $033C",8,1
20 IF A=1 THEN A=2:LOAD"OBJ $C000",8,1
30 SYS 49152

```

Dieses Ladeprogramm liest zwei Maschinenprogramme von Diskette ein und startet den Maschinencode bei \$c000 (#49152).

\$e1be: Routine zum Basic-Befehl OPEN

Diese Routine wertet lediglich die OPEN-Parameter über »jsr oclpar« aus und ruft die Kernall-Routine OPEN (\$ffc0) auf gewohnte Basic-Kernal-Aufrufweise auf:

BOPEN (\$e1c1): Basic-Kernal-Aufruf zu OPEN (\$ffc0)

Die Behandlung eventueller Fehler übernimmt wieder einmal EREXIT (\$e0f9).

\$e1c7: Routine zum Basic-Befehl CLOSE

Nach der Parameterauswertung (OCLPAR \$e219 übernimmt diese Tätigkeit) wird die übergebene logische Filenummer geladen und es folgt im Speicher unmittelbar

BCLOSE (\$e1cc): Basic-Kernal-Aufruf zu CLOSE (\$ffc3)

Die Behandlung möglicher Fehler übernimmt auch hier EREXIT (\$e0f9).

GETLSV (\$e1d4):

Parameter für LOAD, SAVE und VERIFY auswerten

Die Parameterangaben bei LOAD, SAVE und VERIFY können sehr verschieden sein, da prinzipiell jeder Parameter entfallen darf, wobei an seine Stelle ein Defaultwert (Ersatzwert) tritt.

Die Parameter haben folgende Defaultwerte:

```

Filename:      " "
Geräteadresse: 1
Sekundäradresse: 0

```

Diese Einstellung nimmt GETLSV (\$e1d4) bei \$e1d4–\$e1df vor. Dann werden die weiteren Parameter anstelle der Defaults eingelesen, sofern sie angegeben sind. Dabei werden folgende Unterprogramme benötigt: COMBYT, PARTST, CHK CPR.

COMBYT (\$e200):

Komma und numerischen Parameter auswerten

Durch »jsr chkpr« wird sichergestellt, daß weitere Parameter folgen; falls nicht, wird durch Stapelmanipulation GETLSV (\$e1d4) abgebrochen.

Andernfalls wird der Bytewert geholt und im X-Register an GETLSV (\$e1d4) zurückgegeben.

PARTST (\$e206): Test auf weitere GETLSV-Parameter

Folgt eine Befehls-/Zeilen-Endmarkierung, so wird GETLSV (\$e1d4) verlassen, andernfalls erfolgt ein Rücksprung in GETLSV (\$e1d4).

CHK CPR (\$e20e): Test auf Komma und weitere Parameter

Diese Prüfroutine bedient sich keiner Stapelmanipulation und ist somit auch für eigene Zwecke verwendbar. Sie stellt sicher, daß an der CHRGET-Zeiger-Position ein Komma steht und darauf keine Befehls- oder Zeilen-Endmarkierung folgt; für den Fall eines derartigen Verstoßes ist der SYNTAX ERROR vorbehalten.

OCLPAR (\$e219):

Parameter für OPEN oder CLOSE auswerten

Diese Routine ist wie GETLSV (\$e1d4) aufgebaut und bedient sich auch der entsprechenden Unterprogrammen PARTST, COMBYT, CHK CPR. Folgende Defaults hat OPEN/CLOSE:

```

Filename:      " "
Geräteadresse: 1
Sekundäradresse: 0

```

Eine Filenummer ist hingegen unabdingbar. OPEN oder CLOSE dürfen somit nicht implizit (ohne Parameter) verwendet werden.

COS \$e264: Routine zur Basic-Funktion COS

Zur Abwälzung der Kosinusberechnung an die Sinus-Routine macht sich der Basic-Interpreter das folgende mathematische Gesetz zunutze:

$$\cos(x) = \sin(x + \pi/2)$$

Somit wird der COS-Parameter um die Konstante $\pi/2$ erhöht; im Speicher folgt SIN \$e26b.

SIN \$e26b: Routine zur Basic-Funktion SIN

Diese Routine führt einige Berechnungen mit dem Parameter durch, bis das Näherungspolynom seine Pflicht tut.

TAN \$e2b4: Routine zur Basic-Funktion TAN

Der Tangens ist mathematisch als der Quotient aus Sinus und Kosinus definiert. Deshalb berechnet TAN zunächst den Sinus des FAC und speichert ihn ab \$4e, um dann vom zuvor in FAC #3 geretteten TAN-Parameter auch den Kosinus zu ermitteln; daraufhin erfolgt die Division der beiden Werte.

\$e2e0–\$e30d: Zahlenwerte für SIN/COS

Hier liegen nicht nur Konstanten im Zusammenhang mit π ($\pi/2$ und 2π), sondern auch ein fünfgradiges Polynom ab \$e2ef.

ATN \$e30e: Routine zur Basic-Funktion ATN

Der Arcustangens (inverser Tangens) ist die Umkehrfunktion zu TAN. Ein elfgradiges Polynom berechnet diesen.

\$e33e–\$e37a: Näherungspolynom für ATN

Vor der Anwendung dieses Polynoms führt jedoch die ATN-Routine einige Berechnungen durch.

NMIBAS (\$e37b): NMI-Routine für Basic 2.0

Auf diese Stelle weist der Vektor \$a002/\$a003. Bei Auslösen eines NMI wird die Filetabelle gelöscht, das Ein-/Ausgabe-Flag initialisiert, ein Teil der Hilfsspeicher zurückgesetzt und schließlich ein Warmstart ausgelöst. Die Kernal-NMI-Routine hat bereits wichtige Initialisierungen vorgenommen, bevor sie diese Routine anspringt.

\$e38b: ERROR-Routine

Der Fehlercode aus dem X-Register wird hier zunächst auf ein möglicherweise gesetztes Bit 7 untersucht; liegt ein größerer Fehlercode als \$7f vor, so wird ein »READY.«-Warmstart durchgeführt. Andernfalls wird die Fehlermeldung ausgegeben und das laufende Basic-Programm abgebrochen.

RESBAS (\$e394): Reset-Routine für Basic 2.0

Hier werden vom Basic-Interpreter, dessen Vektor \$a000/\$a001 an diese Stelle weist, die Vektoren im Bereich \$0300–\$030b sowie alle anderen RAM-Hilfsspeicher initialisiert und die Einschaltmeldung ausgegeben. Zur Initialisierung des Basic-Programmspeichers wird einfach der NEW-Befehl simuliert sowie warmgestartet.

\$e3a2–\$e3b9: CHRGET-Routine für Initialisierung

Von diesen Adressen wird die CHRGET-Routine in ihren eigentlichen Bereich kopiert; in diesem Kapitel steht die Dokumentation bei \$0073.

\$e3ba–\$e3be: Ausgangswert für RND (SEED-Wert)

Dieser Wert »0.811635157« ist der SEED-Wert vor dem ersten RND-Aufruf, so daß sie die Funktion RND(0) gleich vom Start weg benutzen dürfen.

INITMP (\$e3bf):**RAM-Arbeitsspeicher-Initialisierung für Basic**

Diese Routine führt folgende Arbeiten aus:

- schreibt JMP-Opcodes vor Hilfs- und USR-Vektor
- richtet den USR-Vektor zunächst auf ILLEGAL QUANTITY ERROR
- initialisiert die Umwandlungsvektoren für Fließkomma und Integer

- kopiert CHRGET/CHRGOT-Routine und SEED-Wert in die Zeropage
- setzt die Länge eines Stringvariableneintrags für Garbage Collection fest
- löscht das Überlaufbyte für Fließkommarechnungen
- setzt das INPUT-Kommentarflag zurück
- gibt den temporären Stringstapel frei
- schreibt einen Linkpointer vor den Basic-Eingabepuffer
- begrenzt den Basic-Speicher nach oben und unten, so daß er maximal groß ist
- erhöht den Programmanfangszeiger und schreibt ein Nullbyte vor den Speicher

MSGNEW (\$e422):**Einschaltmeldung ausgeben und NEW-Befehl ausführen**

Zunächst organisiert MSGNEW (\$e422) freien Speicher am Programmbeginn. Dann wird die Einschaltmeldung ausgegeben. Die »BYTES FREE«-Angabe ergibt sich dabei aus der Differenz der obersten Basic-Adresse und der Programmspeicher-Anfangsadresse; als 2-Byte-Wert wird sie über NUMOUT (\$bdcd) auf den Bildschirm gebracht.

Zum Abschluß wird der NEW-Befehl ausgeführt, um den Basic-Programmspeicher zu initialisieren.

\$e447–\$e452: Initialisierungswerte für Basic-Vektoren

Diese Initialisierungswerte schreibt INIVEC (\$e453) an die richtigen RAM-Adressen (\$0300–\$030b).

INIVEC (\$e453):**Initialisierung der Basic-Vektoren \$0300–\$030b**

In einer einfachen Dekrementierschleife wird die Tabelle ab \$e447 nach \$0300 übertragen.

\$e45f: Füllbyte ohne Bedeutung**\$e460–\$e4ab: Texte für Einschaltmeldung**

Das STROUT-Format (\$00 als Endmarkierung) liegt bei diesen Texten vor.

\$e4ac: Füllbyte ohne Bedeutung**BCKOUT (\$e4ad):****CKOUT-Behandlung des Basic-Interpreters**

Die Besonderheit von BCKOUT (\$e4ad) und quasi die Existenzberechtigung liegt darin, daß der Akkumulatorinhalt nicht verlorengeht, falls kein Fehler aufgetreten ist.

\$e4b7–\$e4d2:**Füllbytes als Abgrenzung zu den Kernal-Routinen**

Unmittelbar hinter diesen irrelevanten Füllbytes beginnt dann endlich das eigentliche Kernal, das nicht mehr vom Basic-Interpreter durchgesetzt ist.

\$e4d3: mögliche Fortsetzung von \$ef94

An dieser Adresse (\$e4d3) stehen in älteren C64-Versionen Füllbytes; die neueren Fassungen (zum Beispiel das im C128 integrierte C64-Betriebssystem für den C64-Modus) verfügen allerdings über diese Ergänzung der RSRTT-Routine (in RSRTT erfolgt eine Startbitprüfung). Der vorliegende »Blinddarm« der Routine ab \$e4d3 schreibt dabei den Akku (enthält hier \$00 wegen \$ef90/\$ef92) in das Flag RINONE (Startbitprüfungsflag) und belegt den Hilfsspeicher RIPRTY (Eingabeparität für RS 232) mit 1.

Es erfolgt ein gewöhnlicher RTS-Rücksprung.

\$e4da: mögliche Hilfsroutine von \$ea07

Diese Teilroutine setzt für DELLIN (\$e9ff) die Zeichenfarbe an einer gelöschten Bildschirmposition. Es sei darauf hingewiesen, daß DELLIN (\$e9ff) beim Bildschirmlöschen (Steuerzeichen \$93) als Unteroutine fungiert; deshalb muß man bei den alten C64-Versionen, denen die \$e4da-Routine abgeht, beim POKen in den Bildschirmspeicher auch an das gleichzeitige Setzen des Farb-RAM denken. Ist aber die Routine \$e4da vorhanden, so wird beim Löschen des gesamten Bildschirms oder einer einzelnen Zeile automatisch auch das Farb-RAM initialisiert.

WATCBM (\$e4e0):**Hilfsroutine zum Warten auf Commodore-Taste**

Diese Hilfsroutine erwartet im Akku den Inhalt von \$a1 (mittelwertiges Byte der Systemuhr) und wartet dann eine gewisse Zeitspanne (ca. 8–9 Sekunden); diese Wartezeit wird durch Auslösen der Taste <C> abgebrochen.

\$e4ec: Tabelle der RS-232-Baudraten für die PAL-Version

Diese Tabelle enthält zu den vom Betriebssystem unterstützten Übertragungsgeschwindigkeiten für RS 232 die entsprechenden Timerwerte. Diese Timerwerte werden in die CIA-Register geschrieben, um die Arbeitsgeschwindigkeit des C64 haargenau einzustellen.

Die Einheit »Baud« ist nichts anderes als die Bezeichnung für »Bit/Sekunde«. Bei einer Übertragungsrate von 2400 Baud werden also 2400 Bits in 1 Sekunde übertragen. Folgende Baudraten werden theoretisch angeboten:

50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2400

Die entsprechende Tabelle für die NTSC-Version des C64 beginnt bei \$fec2.

IOBASE (\$e500): Routine zum Kernal-Einsprung \$fff3

Diese Routine lädt die Adresse \$dc00 (Basisadresse der CIA-Register im Speicher) in X- und Y-Register, um dann in die aufrufende Routine zurückzukehren.

SCREEN (\$e505): Routine zum Kernal-Einsprung \$ffed

Das Bildschirmformat des C64 wird durch »jsr screen« in X- und Y-Register geholt:

X = \$28 = #40 = Anzahl der Spalten

Y = \$19 = #25 = Anzahl der Zeilen

Da das Bildschirmformat normalerweise unveränderlich ist, hat der Aufruf dieser Routine nur dann einen Zweck, wenn das Programm auf mehreren Commodore-Computern laufen soll.

PLOT (\$e50a): Routine zum Kernal-Einsprung \$fff0

Zwei gegensätzliche Aufgaben werden von PLOT (\$fff0 – \$e50a) auf Wunsch bewältigt:

1. Carry-Flag = 0: Cursorposition setzen

Dazu wird der Cursor an die Position [X;Y] gesetzt. Das Y-Register enthält dabei einen Wert von #0 bis #39 (\$00–\$27), also die Cursorspalte, während das X-Register die Cursorzeile von #0 bis #24 (\$00–\$18) angibt.

Die Position wird zunächst in \$d3 und \$d6 verzeichnet, woraufhin die STUPT-Routine eine Aktualisierung aller anderen Hilfsspeicher des Editors vornimmt.

Um einen eigenen RTS-Befehl für die Position-Setzen-Teilbehandlung zu sparen, folgt im Speicher unmittelbar die Position-Lese-Routine, die lediglich die bereits übergebene Cursorposition erneut in die Register holt und dann über RTS zurückspringt.

2. Carry-Flag = 1: Cursorposition lesen

Die Cursorzeile wird aus \$d6, die Cursorspalte hingegen aus \$d3 entnommen. Diese Positionsangaben beginnen mit der Zählung an der HOME-Position bei [0;0]. Die rechte untere Bildschirm-ecke ist also durch [39;24] bezeichnet, aber keinesfalls durch die illegale Positionsangabe [40;25], die bereits außerhalb des zulässigen Bereiches liegt.

INTSCR (\$e518): Bildschirm initialisieren

Die INTSCR-Initialisierung des Bildschirms beinhaltet die CLEAR-Routine (\$e544) zum Löschen des Bildschirms. Vor deren Ausführung werden aber folgende Tätigkeiten ausgeübt:

- Die VIC-Register erhalten ihre Vorbelegungswerte. Dadurch schaltet der VIC in den herkömmlichen Textmodus. Verantwortliches Unterprogramm ist INTVIC (\$e5a0).

- Die Zeichensatzumschaltung über die Tastenkombination <SHIFT C=> wird zugelassen, falls sie zuvor gesperrt gewesen sein sollte.
- Der Cursor wird in die Blinkphase versetzt.
- Die Adresse der Tastaturdekodierungsroutine des Kernal wird in den KEYLOG-Vektor geschrieben. Eventuelle Manipulationen dieses Vektors werden also aufgehoben.
- Der Tastaturpuffer bekommt eine Länge von 10 Zeichen zugewiesen.
- Die Verzögerung und die Zählgeschwindigkeit für den Tastatur-Repeat (Tastenwiederholung) werden initialisiert.
- Hellblau wird zur aktuellen Zeichenfarbe (SX 64: dunkelblau).
- Der Cursor wird zwar auf »Blinken« gestellt, aber zunächst nicht dargestellt.
- Dann wird, wie schon angekündigt, der Bildschirm gelöscht:

CLEAR (\$e544): Bildschirm löschen

Diese Sonderbehandlung für das Steuerzeichen \$93 (#147) initialisiert die Tabelle LDTB1, die für den Editor die High-Bytes der Adressen aller Bildschirmzeilen enthält. Daraufhin löscht eine DEL-LIN-Schleife den gesamten Bildschirm. Das Löschen vollzieht sich dabei »von unten nach oben«, damit es als Dekrementierschleife programmiert werden konnte.

Danach sind also der Bildschirmspeicher und, sofern die Routine bei \$e4da (siehe dort) vorhanden ist, das Farb-RAM initialisiert. <SHIFT CLR/HOME> beinhaltet aber auch das Positionieren des Cursors in der linken oberen Bildschirmecke (HOME-Position). Dafür dient HOME (\$e566) als Teil von CLEAR (\$e544):

HOME (\$e566):

Cursor in linke obere Bildschirmecke setzen

Die HOME-Position (linke obere Bildschirmecke, [0;0]) steuert diese Sonderbehandlung des Steuerzeichens \$13 (#19) an. Dazu wird 0 sowohl als Spalten- als auch als Zeilenangabe gesetzt, wozu die PLOT-Routine (\$fff0 – \$e50a) allerdings keine Verwendung findet; HOME (\$e566) schreibt die Position [0;0] einfach direkt in die Hilfsregister \$d3 und \$d6.

Im Speicher folgt dann STUPT (\$e56c), um auch die anderen Editorspeicher an die neue Cursorposition anzupassen:

STUPT (\$e56c):

Hilfsspeicher des Editors gemäß \$d3/\$d6 modifizieren

Allein dadurch, daß in \$d3/\$d6 die Cursorposition steht, ist der Editor noch nicht vollständig auf die neue Cursorposition eingestellt. Deshalb ist STUPT (\$e56c) als Unterprogramm von PLOT (\$fff0 – \$e50a) für das Anpassen der Hilfszeiger verantwortlich:

\$d1/\$d2 = aktuelle Adresse im Bildschirmspeicher

\$f3/\$f4 = aktuelle Adresse im Farb-RAM

\$d5 = Länge der aktuellen logischen Bildschirmzeile (40/80)

Dabei kann es auch vorkommen, daß \$d3 und \$d6 verändert werden; der Grund dafür liegt darin, daß eine Eingabezeile seitens des Editors auch mehr als eine »echte« Zeile von 40 Zeichen Länge umfassen darf, an PLOT (\$fff0) aber immer die »echte« Positionsangabe übergeben wird.

Nach »jsr stupt« beziehen sich \$d3 und \$d6 also auf die logischen Bildschirmzeilen.

\$e591:

Unterroutine für Tastatur-Eingabeschleife von \$e621

Diese Routine berechnet die Hilfszeiger des Editors neu, wenn die aktuelle Cursorzeile (in X enthalten) nicht mit der INPUT-Cursorzeile übereinstimmt.

Gegenüber älteren C64-Versionen bestehen hier leichte Abweichungen.

\$e599: Füllbefehl

Dieser NOP-Befehl hat keine weitere Bedeutung.

\$e59a: Einsprung zur

aufeinanderfolgenden Ausführung von INTVIC und HOME

Diese Kopplung der Routinen INTVIC (\$e5a0) und HOME (\$e566) ist gewissermaßen eine Entschärfung von INTSCR (\$e518).

INTVIC (\$e5a0): VIC-Register initialisieren

INTVIC (\$e5a0) stellt zunächst die standardmäßige Ein-/Ausgabe her (Tastatur und Bildschirm). Daraufhin überträgt INTVIC (\$e5a0) die Initialisierungstabelle ab \$ecb9 in die Register des VIC. Ein im C64-Modus des C128 zusätzlich vorhandenes Registerpaar 47/48 wird nicht berücksichtigt.

NXTKEY (\$e5b4):

nächstes Zeichen aus Tastaturpuffer in Akkumulator holen

Die Tastaturabfrage erfolgt bekanntlich im Interrupt; jeder Tastendruck wird dort registriert und im Tastaturpuffer abgelegt. Dieser liegt in den 10 Speicherplätzen \$0277–\$0280. Je später ein Tastendruck erfolgt, desto später steht er im Puffer. \$0277 (erste Pufferadresse) enthält jeweils das nächste Byte, und genau dieses liest die Unteroutine NXTKEY (\$e5b4) ein. Dabei wird zusätzlich dieses Byte aus dem Tastaturpuffer entfernt, indem die Anzahl der Zeichen im Tastaturpuffer – enthalten in \$c6 – verringert wird. Dafür kopiert NXTKEY (\$e5b4) die ab \$0278 (\$0277 + 1) abgelegten ASCII-Codes nach \$0277, um Platz für neue Tastendrucke zu schaffen.

Bei der Ausführung von NXTKEY (\$e5b4) sollte das Interrupt-Flag gesetzt sein (SEI), um dagegen Vorsorge zu treffen, daß durch eine interruptgesteuerte Änderung des Tastaturpuffers das System aus den Angeln gehoben wird. NXTKEY (\$e5b4) löscht nach dem kritischen Teil automatisch das Interrupt-Disable-Flag und auch das Carry, um darauf hinzuweisen, daß kein Fehler aufgetreten ist (bei NXTKEY kann niemals ein Fehler entstehen, da die Tastatur keine »kritische« Peripherie ist).

\$e5ca: Tastatur-Eingabeschleife

Diese Schleife ist für die Tastatureingabe bei gleichzeitiger Bildschirmausgabe der gedrückten Tasten zuständig. Unter Verwendung von NXTKEY (\$e5b4) werden nach und nach alle eingegebenen Tasten auf den Bildschirm ausgegeben und somit in den Bildschirmspeicher übernommen, von wo an späterer Stelle die Auswertung geschieht.

Die Sonderbehandlung für die Tastenkombination <SHIFT RUN/STOP> wird innerhalb dieser Routine übernommen, sobald der ASCII-Code \$83 auftritt: \$03 ist der Code für <RUN/STOP>, \$83 (= \$03 + \$80) ist der Code für <SHIFT RUN/STOP>.

Ist die Eingabe vollständig eingelesen, weil das Schlußzeichen [CR] (RETURN-Taste) auftritt, wird sie verarbeitet.

SCRGET (\$e632):

Zeichen von aktueller Bildschirmposition in Akku holen

Die SCRGET-Routine holt von der aktuellen Cursorposition ein Zeichen aus dem Bildschirmspeicher und wandelt es in den ASCII-Code um, den wiederum der Akkumulator an die aufrufende Routine übermittelt. X- und Y-Register werden auf den Stapel gerettet, damit sie aus Sicht der aufrufenden Routine unangetastet bleiben.

Das aktuelle Zeichen aus dem Bildschirmspeicher wird dabei anhand der Editorspeicher \$d1/\$d2 und \$d3 nach \$d7 geholt und bei \$e640–\$e653 in den ASCII-Code konvertiert. Im Falle eines Anführungszeichens wird der Quote Mode umgeschaltet (\$e656). Das Umwandlungsergebnis gibt SCRGET (\$e632) auf den Bildschirm aus; am Eingabe-Ende wird \$0d, der RETURN-Code, zurückgegeben.

CHGQUT (\$e684): Quote-Mode-Flag bei Anführungszeichen invertieren

Steht im Akkumulator ein beliebiger ASCII-Code, so prüft CHGQUT (\$e684) zunächst, ob es sich um ein Anführungszeichen handelt. Falls nicht, wird die Routine ohne weitere Operationen verlassen; falls ja, invertiert CHGQUT (\$e684) das Quote-Mode-Flag.

CHRRAM (\$e691):

Zeichen in Bildschirmspeicher übernehmen

Zum Abschluß der Bildschirmausgabe wird der ASCII-Code des Zeichens von anderen BSOUT-Teilen in den Bildschirmcode um-

gewandelt, woraufhin er an diese Routine CHRRAM (\$e691) zur Übernahme in den Bildschirmspeicher weitergeleitet wird. Auch die Zeichenfarbe findet Berücksichtigung.

Mitten in CHRRAM (\$e691) liegt der elementare Einsprung \$e6a8:

\$e6a8: Schlußbehandlung der Bildschirmausgabe

Vor der BSOUT-Ausgabe eines Zeichens werden die Prozessorregister auf den Stapel gerettet. Die einzelnen Sonderbehandlungsroutinen für die Steuerzeichen und druckenden Zeichen springen deshalb über \$e6a8 zurück, da dort die Register wiederhergestellt werden. Außerdem kommt der INSRT-Zähler (enthält die Anzahl der noch einzufügenden Zeichen) durch Dekrementierung auf den neuen Stand.

Der Interrupt wird wieder zugelassen.

UPDTL (\$e6b6): LDTB1 aktualisieren

Eine zentrale Funktion im Editor kommt der Tabelle LDTB1 zu, die die High-Bytes der Anfangsadressen der logischen Bildschirmzeilen enthält. Diejenigen Zeilen, die den Anfang einer logischen Zeile darstellen – oder die einzige Zeile einer logischen Zeile sind, wenn diese nicht die Länge der echten Zeilen überschreitet –, sind durch ein gesetztes Bit 7 markiert.

SCROUT (\$e716): BSOUT-Routine für Gerät #3

Zur Ausgabe auf den Bildschirm dient diese Teilbehandlung von BSOUT (\$fdd2), die auch einzeln aufgerufen wird.

Im Akkumulator muß das auszugebende Zeichen stehen. Dieses wird in \$d7 gemerkt, wo es über die gesamte SCROUT-Behandlung hinweg verbleibt. Die Form der Ausgabe hängt nun einzig und allein vom ASCII-Code des Zeichens ab. Grundsätzlich gibt es zwei Möglichkeiten:

1. druckende Zeichen

Druckende Codes werden in den Bildschirmcode umgewandelt und letztlich an CHRRAM (\$e693) weitergeleitet, woran sich ja die Schlußbehandlung für SCROUT (\$e716) anschließt.

2. Steuerzeichen

Dazu werden die entsprechenden Sonderbehandlungsroutinen ausgeführt, die allesamt zu \$e6a8 verzweigen, wenn sie ihre Pflicht getan haben.

Zur Identifizierung der einzelnen Steuerzeichen über Vergleichsbefehle wird zunächst nach allen vorhandenen Steuercodes mit Ausnahme der Farbsteuerzeichen gesucht; wird bis dahin kein positives Vergleichsergebnis erzielt, so übernimmt COLCOD (\$e8cb) die Erkennung möglicherweise vorliegender Farbsteuerzeichen. Letztere Routine springt ohne jeden Effekt über RTS zurück, wenn kein Farbcode vorlag.

SETNWL (§e87c): neue Zeile einrichten

Falls bei einer Editorfunktion eine neue Bildschirmzeile entstehen muß, wird diese Routine dafür aufgerufen, die im Bedarfsfall auch gleich das Scrolling durch SCROLL (§e8ea) erledigen läßt.

Die Hilfszeiger des Editors werden vollständig aktualisiert.

CR (§e891): Sprung an Anfang der nächsten Zeile

Diese Sonderbehandlung des Steuerzeichens \$0d (#13) schaltet nicht nur auf Spalte 0 (\$d3 wird mit \$00 belegt) und richtet durch SETNWL (§e87c) eine neue Bildschirmzeile ein, sondern löscht auch die Flags INSRT (Anzahl der noch zu tätigen Einfügungen), QTSW (Quote Mode) und RVS (Reversschrift).

MOVLFT (§e8a1):**Hilfsroutine bei Cursorbewegungen nach links**

Diese Routine führt eine Cursorbewegung nach links aus und paßt dabei außer der Cursorspalte, die verringert werden muß, auch die Nummer der aktuellen Cursorzeile an.

MOVGRH (§e8b3):**Hilfsroutine bei Cursorbewegungen nach rechts**

Zur Bewegung des Cursors um 1 Spalte nach rechts wird im Falle einer Zeilenüberschreitung auch die nächste Cursorzeile angesteuert.

COLCOD (§e8cb):**Erkennung und Ausführung von Farbsteuerzeichen**

Diese Routine sucht den im Akkumulator enthaltenen ASCII-Code in einer Tabelle ab §e8da. Wird der Akkumulator durch Vergleiche nicht in dieser Tabelle wiedergefunden, so endet COLCOD (§e8cb) über RTS, ohne irgendeine Wirkung hinterlassen zu haben. Ansonsten wird der VIC-Farbcode (nicht das Steuerzeichen selbst!) in den Hilfsspeicher COLOR (\$0286), der die aktuelle Zeichenfarbe angibt, geschrieben. Bei weiteren Bildschirmausgaben kommt dieser VIC-Farbcode dann aus COLOR (\$0286) in die korrespondierenden Adressen des Farb-RAM.

§e8da–§e8e9: Tabelle**der Farbsteuerzeichen in Reihenfolge der VIC-Codes**

Diese Tabelle enthält die ASCII-Codes derjenigen Steuerzeichen, die eine Farbänderung hervorrufen. Die Position in dieser Tabelle ist gleichzeitig der dazugehörige VIC-Farbcode; als Positionsangabe wird dabei der Offset vom Tabellenanfang bei §e8da angesehen.

SCROLL (§e8ea):**Aufwärts-Scrolling des gesamten Bildschirms**

Da die Hilfsspeicher \$ac–\$af nicht nur von dieser Routine verwendet werden, rettet SCROLL (§e8ea) diese auf den Stapel und stellt

sie zum Rücksprung wieder auf ihre eingangs vorhandenen Werte zurück.

Zum Verständnis der Routine muß man sich vor Augen halten, daß das Scrolling eine Speicherverschiebung innerhalb des Bildschirmspeichers darstellt.

Dabei enthält nach Berechnung der Speichergrenzen der Zeiger \$ac/\$ad die Quelladresse aus dem Bildschirmspeicher, während \$ae/\$af die Quelladresse aus dem Farb-RAM ist. Die herkömmlichen Zeiger \$d1/\$d2 (aktuelle Bildschirmspeicheradresse) und \$f3/\$f4 (aktuelle Farb-RAM-Adresse) geben dabei die Zieladressen an.

Die Tabelle LDTB1 wird in SCROLL (§e8ea) ebenfalls aktualisiert; die tatsächliche Speicherverschiebung übernimmt aber zeilenweise die SCRLIN-Routine:

SCRLIN (§e9c8): Aufwärts-Scrolling einer einzelnen Zeile

Zur Vorbereitung müssen die Hilfszeiger \$ac/\$ad (Quelle aus Bildschirmspeicher), \$ae/\$af (Quelle aus Farb-RAM), \$d1/\$d2 (Ziel in Bildschirmspeicher) und \$f3/\$f4 (Ziel in Farb-RAM) auf die gewünschten Werte gestellt werden. Dann werden die 40 Bytes (Inhalt einer »echten« Bildschirmzeile) ab der Quelladresse zur Zieladresse übertragen.

COLADR (§e9e0):**Adresse der aktuellen Farb-RAM-Adresse errechnen**

Da Bildschirm- und Farb-RAM dieselbe Struktur haben, läßt sich aus der aktuellen Adresse im Bildschirmspeicher leicht die Farb-RAM-Adresse ermitteln. Das Low-Byte kann unverändert übernommen werden, vom High-Byte jedoch interessieren nur die beiden untersten Bits, zu denen dann durch OR-Verknüpfung das High-Byte der Anfangsadresse des Farb-RAM addiert wird.

Das Ergebnis kommt in den Hilfsspeicher \$ae/\$af; die Bildschirmspeicheradresse wird aus dem Hilfsspeicher \$ac/\$ad, nicht aber aus \$d1/\$d2 entnommen.

LINADR (§e9f0): Hilfszeiger**\$d1/\$d2 auf beliebige Bildschirmzeile richten**

Der Hilfszeiger \$d1/\$d2 enthält immer die Anfangsadresse der aktuellen Bildschirmzeile im Speicher. Diese Routine ermittelt also die Adresse einer Bildschirmzeile, deren Nummer nach der mit 0 beginnenden Numerierung im X-Register steht, indem aus der ROM-Tabelle das Low-Byte der Adresse und aus der Verknüpfung des High-Bytes laut LDTB1 (RAM-Tabelle ab \$d9) mit dem High-Byte der Bildschirmspeicher-Anfangsadresse das High-Byte der Adresse gewonnen wird.

DELLIN (§e9ff): Bildschirmzeile löschen

Eine Bildschirmzeile, deren Nummer im X-Register enthalten ist, löscht DELLIN (§e9ff), indem es diese Zeile mit Leerzeichen über-

schreibt. Die Zeile wird jedoch nicht durch Aufwärts-Scrolling eliminiert.

DELLIN (\$e9ff) fungiert als Unterprogramm von CLEAR (\$e544) und weiteren Routinen dieser Art.

\$ea12: Füllbefehl

Dieser NOP-Befehl ist durch das Einfügen des COLPTR-Aufrufes in der DELLIN-Routine bei \$ea04 entstanden, um die Adressen der darauffolgenden Routinen nicht zu beeinflussen.

Er ist somit auch nur in den einigermaßen ausgereiften C64-Versionen existent.

SETCHC (\$ea13): Zeichen- und Farbcode in Bildschirm- und Farb-RAM schreiben

Steht das auszugebende Zeichen als Bildschirmcode im Akkumulator und die Farbe in X (als VIC-Code, \$0-\$f), schreibt »jsr setchc« diese beiden Bytes an die aktuellen Positionen in Bildschirm- und Farb-RAM. Dazu wird der Cursor ausgeschaltet, damit nicht eine irrtümliche Invertierung des Zeichens durch die interrupt-gesteuerte Cursorbehandlung entsteht.

COLPTR (\$ea24): Farb-RAM-Adresse zur Bildschirmspeicheradresse ermitteln

Der Hilfszeiger \$d1/\$d2 weist immer auf die Anfangsadresse der aktuellen Bildschirmzeile. Nach »jsr colptr« zeigt \$f3/\$f4 auf die korrespondierende Stelle im Farb-RAM. Die Berechnung vollzieht sich wie in COLADR (\$e9e0), einer Routine, die sich auf COLPTR (\$ea24) stützt.

\$ea31: IRQ-Routine des Betriebssystems

Es sei hier auf Abschnitt 3.3.2 verwiesen, wo die Funktionen des standardmäßigen Interruptprogramms ab \$ea31 bereits beschrieben sind, da dies zum grundlegenden Verständnis des Systems unumgänglich ist.

Den dortigen Beschreibungen und dem ROM-Listing (Kapitel 1) ist nichts mehr hinzuzufügen.

SCNKEY (\$ea87): Routine zum Kernal-Einsprung \$ff9f

Die Tastaturabfrage läuft im Interrupt durch Aufruf dieser Routine ab. Sie geht zunächst davon aus, daß keine Taste gedrückt wurde und auch <CTRL>, <C => und <SHIFT> nicht betätigt wurden.

Dann wird anhand der CIA-Register \$dc00 und \$dc01 ermittelt, welche Taste auszuwerten ist. Der Hilfszeiger KEYTAB (\$f5/\$f6) weist dabei auf die jeweilige Tastaturliste im ROM; in Abhängigkeit von <SHIFT>, <C => und <CTRL> muß die richtige Tastaturliste anvisiert worden sein.

Im Programmteil KEYLOG (\$eadd) erfolgt dann die Umwandlung in einen ASCII-Code, der letztlich in den Tastaturpuffer kommt.

KEYLOG (\$eadd): Auswertung von Tastaturcode

Steht ein Tastaturcode in \$cb, wird in dieser Routine der ASCII-Code ermittelt. Als Vektor für KEYLOG (\$eadd) ist noch IKYLOG (\$028f/\$0290) zu erwähnen, welcher in unverändertem Zustand nach \$eb48 weist und somit nach Einstellung der richtigen Tastaturliste (abhängig von <SHIFT>, <C> = und <CTRL>) nach \$eae0 führt.

Sollte der aus der Tastaturliste entnommene ASCII-Code mit dem der letzten Taste übereinstimmen, so erfolgt die Repeat-Sonderbehandlung, die andernfalls übersprungen wird, um unverzüglich den ASCII-Code in den Tastaturpuffer zu übernehmen.

Nun aber zur Repeat-Behandlung. Diese kennt drei unterschiedlich zu behandelnde Formen des Repeat, über die das Flag RPTFLG (\$028a) entscheidet:

1. kein Repeat (RPTFLG = \$40 = %01000000)

Ein erneut ermittelter Tastendruck wird geflissentlich ignoriert.

2. Repeat nur für , , <SPACE> und <CRSR> (RPTFLG = \$00 = %00000000)

Es erfolgt ein Vergleich mit denjenigen Tastencodes, bei denen der Repeat durchgeführt werden soll. Werden diese gefunden, so wird bei \$eb0d der Repeat durchgeführt, als ob er für alle Tasten zulässig wäre.

Wurde keine Taste gedrückt, so wird die Repeatbehandlung übersprungen.

3. Repeat für alle Tasten (RPTFLG = \$80 = %10000000)

Dieser Fall ist am einfachsten: Die aktuelle Taste wird auf konventionelle Weise in den Tastaturpuffer übernommen, sobald der Repeat-Verzögerungszähler abgelaufen ist.

\$eb48: Beginn der Tastaturdekodierung mit Prüfung von <SHIFT>, <C => und <RETURN>

Die Zusatztasten <SHIFT>, <C => und <CTRL>, die nur in Verbindung mit anderen Tasten eine Funktion auslösen, werden hier behandelt, wobei das Flag für diese Tasten (SHFLAG \$028d) die Grundlage bildet. Beim Drücken einer einzigen Taste dieser Art (z.B. nur <SHIFT>) wird der KEYTAB-Zeiger auf die richtige Tastaturliste gerichtet. Für <SHIFT C=> erfolgt eine Sonderbehandlung zur Zeichensatzumschaltung, sofern nicht das Flag MODE (\$0291) dagegenspricht. Dieses Blockierflag wird durch die Steuerzeichen \$08 und \$09 gesetzt beziehungsweise gelöscht.

Anschließend wird in jedem Fall bei \$eae0 mit der Umwandlung des Tastencodes in einen ASCII-Code fortgefahren, wo auch

mögliche Repeats (Tastenviederholungen) einer Sonderbehandlung unterliegen.

\$eb79–\$eb80:

Tabelle der Basisadressen für die Tastaturtabellen

Die Adressen der verschiedenen Tastaturtabellen entnimmt die Unterroutine ab \$eb48 dieser Tabelle. Als X-Offset steht dabei

- \$00 für \$eb81, die Tastaturtabelle ohne <SHIFT>, <C=> oder <CTRL>
- \$02 für \$ebc2, die Tastaturtabelle bei gedrückter SHIFT-Taste
- \$04 für \$ec03, die Tastaturtabelle bei gedrückter CBM-Taste
- \$06 für \$ec78, die Tastaturtabelle bei gedrückter CTRL-Taste

\$eb81–\$ebc1: Tastaturtabelle #0 (Tasten ohne Zusatzaste)

Der Tastencode (\$00–\$40) wird in dieser Tabelle als Offset auf den ASCII-Code der Taste verwendet. Ein Wert \$ff bedeutet in der Tabelle, daß kein ASCII-Code zu einem Tastencode feststellbar ist.

\$ebc2–\$ec02: Tastaturtabelle #1 (Tasten mit <SHIFT>)

Wie \$eb81–\$ebc1, aber für die Tasten in Kombination mit <SHIFT>.

\$ec03–\$ec43: Tastaturtabelle #2 (Tasten mit <C=>)

Wie \$eb81–\$ebc1 und \$ebc2–\$ec02, aber für die Tasten in Kombination mit der Commodore-Taste.

\$ec44:

Bearbeitung der Steuerzeichen zur Zeichensatzwahl

Durch den ASCII-Code \$0e kommt die Klein-/Groß-Schrift, durch \$8e die Groß-/Grafik-Schrift zum Zuge. Diese Sonderbehandlung erkennt mittels CMP die entsprechenden Steuerzeichen und führt sie aus, wobei für die BUSINESS-Schrift (Steuerzeichen \$0e, Klein-/Groß-Schrift) Bit 1 in VIC-Register #24 gesetzt, für die GRAPHICS-Schrift (Steuerzeichen \$8e, Groß-/Grafik-Schrift) gelöscht wird.

Danach wird die Schlußbehandlung für SCROUT (\$e716) ausgelöst, wozu nach \$e6a8 verzweigt wird.

Lag keines der beiden Steuerzeichen \$0e und \$8e vor, so wird auf \$08 und \$09 geprüft; diese Codes sperren die Kombination <SHIFT C=> oder geben sie frei. Dazu wird das Flag MODE (\$0291) gesetzt oder gelöscht.

Handelte es sich auch nicht um die Steuercodes \$08 oder \$09, wird die Bildschirmausgabe ohne weiteres beendet.

\$ec78–\$ecb8: Tastaturtabelle #3 (Tasten mit <CTRL>)

Wie \$eb81–\$ebc1, \$ebc2–\$ec02 und \$ec03–\$ec43, aber für die Tasten in Kombination mit der Commodore-Taste.

\$ecb9–\$ece6: Initialisierungstabelle für VIC-Register

INTVIC (\$e5a0) überträgt diese Tabelle in die VIC-Register. Das ROM-Listing (Kapitel 1) geht ausführlich auf die Bedeutung der einzelnen Bits und Bytes ein.

\$ece7–\$ecf:

ROM-Tabelle des Textes für <SHIFT RUN/STOP>

Die Tastenkombination <SHIFT RUN/STOP> dient als Ersatz für die folgenden Tastendrücke (beim herkömmlichen C64; der SX64 wird am Ende von Kapitel 1 angesprochen):

<l> <o> <a> <d> <CR> <r> <u> <n> <CR>

Diese ASCII-Tabelle enthält die Zeichencodes, die bei der Sonderbehandlung in den Tastaturpuffer übertragen werden.

\$ecf0–\$ed08: Tabelle der

Low-Bytes der Basisadressen für die Bildschirmzeilen

In dieser Tabelle stehen die Low-Bytes der Basisadressen der Bildschirmzeilen in Bildschirmspeicher. Da der Bildschirmspeicheranfang nur durch Ändern des High-Bytes verschoben werden kann, bleiben diese Low-Bytes immer gleich.

Im Normalzustand beginnt das Bildschirm-RAM bei \$0400 (#1024); dann haben die einzelnen Bildschirmzeilen folgende Anfangsadressen, deren Low-Bytes übrigens unterstrichen sind, da sie in dieser Tabelle ab \$ecf0 im Speicher stehen:

```
Zeile #00: $0400
Zeile #01: $0428
Zeile #02: $0450
Zeile #03: $0478
Zeile #04: $04a0
Zeile #05: $04c8
Zeile #06: $04f0
Zeile #07: $0518
Zeile #08: $0540
Zeile #09: $0568
Zeile #10: $0590
Zeile #11: $05b8
Zeile #12: $05e0
Zeile #13: $0608
Zeile #14: $0630
Zeile #15: $0658
Zeile #16: $0680
Zeile #17: $06a8
Zeile #18: $06d0
Zeile #19: $06f8
Zeile #20: $0720
Zeile #21: $0748
```

Zeile #22: \$0770
 Zeile #23: \$0798
 Zeile #24: \$07c0

TALK (\$ed09): Routine zum Kernal-Einsprung \$ffb4

Um ein Gerät am IEC-Bus zum Senden von Daten (»talk«, englisches Wort für »sprechen, unterhalten«) zu bewegen, muß ihm das TALK-Signal gesendet werden. Dies besteht aus einem gesetzten Bit 6 in der Geräteadresse, welche im Akkumulator übermittelt wird. Die TALK-Routine selbst setzt lediglich Bit 6 und überspringt mittels BIT-Trick den LISTEN-Einsprung; die gemeinsame Behandlung für TALK (\$ffb4 → \$ed09) und LISTEN (\$ffb1 → \$ed0c) sendet dann das TALK-Bit und die Geräteadresse als Bytewert auf den IEC-Bus.

LISTEN (\$ed0c): Routine zum Kernal-Einsprung \$ffb1

Wie TALK (\$ffb4 → \$ed09), setzt diese Routine im Akkumulator (übermittelte Geräteadresse) das entsprechende Bit. Das LISTEN-Signal besteht dabei aus Bit 5.

Im Speicher folgt die gemeinsame Behandlung für TALK (\$ffb4 → \$ed09) und LISTEN (\$ffb1 → \$ed0c).

\$ed0e: gemeinsame Behandlung für TALK und LISTEN

Die TALK- und LISTEN-Routine haben jeweils die Voraussetzung für \$ed0e geschaffen: Im Akkumulator steht die Geräteadresse mit dem entsprechenden Bit (LISTEN: Bit 5; TALK: Bit 6).

Dieses Byte wird hier durch den IECBYT-Aufruf (»\$ed19 jsr iecbyt«) auf den seriellen Bus gelenkt. Ein möglicher DEVICE NOT PRESENT ERROR wird dadurch erkannt, daß das anzusprechende Gerät nicht reagiert.

SECOND (\$edb9): Routine zum Kernal-Einsprung \$ff93

Diese Routine sendet eine im Akkumulator enthaltene Sekundäradresse auf den IEC-Bus als Ergänzung des LISTEN-Signals. Anschließend wird ATN auf »high« gesetzt.

TKSA (\$edc7): Routine zum Kernal-Einsprung \$ff96

Die Sekundäradresse (im Akku zu übergeben) eines TALK-Gerätes sendet diese Routine auf den IEC-Bus, woraufhin das Signal »CLOCK IN high« erwartet wird, bevor ein Rücksprung erfolgen kann.

Zwischenzeitlich muß der Interrupt abgeschaltet werden, um freien Zugriff auf die CIA-Register zu haben. Nach »jsr tksa« ist aber das Interrupt-Disable-Flag auf jeden Fall gelöscht.

CIOUT (\$eddd): Routine zum Kernal-Einsprung \$ffa8

Zur Ausgabe des Akkumulators auf den IEC-Bus dient diese Routine, die den 1-Byte-Puffer BSOUR (Adresse \$95) berücksichtigt: Steht in diesem bereits ein Byte, so wird zunächst dieses gesen-

det; andernfalls wird das auszugebende Byte lediglich in BSOUR (\$95) gepuffert.

UNTALK (\$edef): Routine zum Kernal-Einsprung \$ffab

Soll eine TALK-Kommunikation (Computer empfängt Daten von Peripherie) beendet werden, muß dies dem anzusprechenden Gerät als UNTALK-Signal mitgeteilt werden, damit es aufhört, Daten an den Computer zu schicken. Diese Routine überträgt das Bitmuster \$5f (%01011111), das UNTALK repräsentiert, über die gemeinsame Behandlung von UNTALK (\$ffab → \$edef) und UNLSN (\$ffae → \$edfe) auf den IEC-Bus.

UNLSN (\$edfe): Routine zum Kernal-Einsprung \$ffae

Zum Abschluß einer LISTEN-Kommunikation (Computer schickt Daten an Peripherie) sendet UNLSN (\$ffae → \$edfe) das UNLISTEN-Signal, das durch das Bitmuster \$3f (%00111111) vertreten wird.

Die Übertragung erledigt die gemeinsame Behandlung von UNTALK (\$ffab → \$edef) und UNLSN (\$ffae → \$edfe), da sich die beiden Routinen nur im Bitmuster unterscheiden.

\$ee00: gemeinsame Behandlung von UNTALK und UNLSN

Hier wird das entsprechende Signal durch einen Teil der TALK/LISTEN-Routine auf den IEC-Bus gesendet; einigen weiteren Signalen folgt eine Verzögerungsschleife, nach deren Ablauf CLOCK und DATA auf »high« gesetzt werden.

IECIN (\$ee13): Routine zum Kernal-Einsprung \$ffa5

Das Einlesen eines einzelnen Bytes vom IEC-Bus in den Akku führt IECIN (\$ffa5 → \$ee13) durch. Dabei werden in einer Schleife die einzelnen Bits über den Datenport A von CIA 2 eingeholt und bei \$ee65 in den Byte-Speicher \$a4 eingebunden, den schließlich bei \$ee80–\$ee84 der Akkumulator zurückgibt.

CLCKHI (\$ee85): CLOCK auf »high« setzen

Durch Löschen von Bit 4 (CLOCK-Bit) in Datenport A von CIA 2 entsteht das Signal »CLOCK high«.

CLCKLO (\$ee8e): CLOCK auf »low« setzen

Durch Setzen von Bit 4 (CLOCK-Bit) in Datenport A von CIA 2 entsteht das Signal »CLOCK low«.

DATAHI (\$ee97): DATA auf »high« setzen

Durch Löschen von Bit 5 (DATA-Bit) in Datenport A von CIA 2 entsteht das Signal »DATA high«.

DATALO (\$eea0): DATA auf »low« setzen

Durch Setzen von Bit 5 (DATA-Bit) in Datenport A von CIA 2 entsteht das Signal »DATA low«.

DEBPIA (\$eea9): Datenport A von CIA 2 auslesen

Das DATA- und das CLOCK-Bit von Datenport A in CIA 2 holt DEBPIA (\$eea9) in die Prozessorflags C (DATA IN) und N (CLOCK IN). Dabei wird der Akkumulator verändert; er enthält den linksverschobenen Inhalt von \$dd00 (Datenport A, CIA 2).

WAIT.1 (\$eeb3): 1 Millisekunde warten

Diese Warteschleife dauert ziemlich exakt 1000 Taktzyklen, also etwa ein Tausendstel einer Sekunde (= 1 Millisekunde).

\$eebb: Ausgabe-Teilroutine des NMI bei RS232-Betrieb

Diese Teilroutine ist lediglich die Fortsetzung von \$fe9d aus. Sie überträgt das nächste Bit inklusive Neuberechnung der Parität und Erkennung des Stop-Bits am Byte-Ende.

Entscheidend ist dabei der Bitzähler BITTS (\$b4); wird dieser auf 0 heruntergezählt, so erfolgt die Übertragung des nächsten Bytes durch die Routine RSTBGN (\$ef06).

Ein gelöscht Bit wird als SPACE (Bitmuster %00000000), ein gesetztes Bit als MARK (Bitmuster %11111111) gesendet.

Die Übertragungsbits werden dem 1-Byte-Puffer RODATA (\$b6; RS232-Ausgabedaten, engl.: RS Output DATA) durch Rechtsverschiebung nacheinander entnommen.

RSTBGN (\$ef06):**Übertragung des nächsten Bytes auf RS232**

Ist der RS232-Bitzähler BITTS (\$b4) abgelaufen, so sorgt RSTBGN (\$ef06) dafür, daß das nächste Ausgabebyte aus dem RS232-Ausgabepuffer in den 1-Byte-Puffer RODATA (\$b6) gelangt und der Bitzähler wieder initialisiert wird. Nach dem Rücksprung ist es nur eine Frage der Zeit, bis beim nächsten RS232-NMI auch dieses Byte übertragen wird.

Ist der RS232-Ausgabepuffer leer, erfolgt eine Sonderbehandlung.

CALCBT (\$ef4a):**Berechnung der Anzahl der RS232-Ausgabebits**

Die Wortlänge für RS232 ist von 5 bis 9 frei wählbar. Damit die Bitzähler für RS232 jeweils auf die richtigen Werte gestellt werden können, liefert diese Hilfsroutine die gewünschte Bitzahl im X-Register.

Der Berechnung dient der RS232-Hilfsspeicher \$0293 (Kontrollregister) als Grundlage.

RSRCVR (\$ef59): Auswertung**eines über RS232 einzulesenden Bit im NMI**

Wurde ein Bit nach \$a7 (INBIT) geholt, so wertet RSRCVR (\$ef59) dieses aus. Falls es sich um ein Startbit handelt (erkennbar am

gesetzten Startbitflag, RINONE \$a9) erfolgt eine Sonderbehandlung RSRTTRT (\$ef90).

Andernfalls wird der Bitzähler \$a8 (BITCI) verringert; sind schon alle Bits empfangen, wird das Byte durch »beq \$ef97« in den Eingabepuffer übernommen. Im nächsten Durchgang erfolgt die Stopbit-Prüfung.

Im Normalfall jedoch handelt es sich um ein ordinäres Datenbit, das bei Aktualisierung der Parität RIPRTY (\$ab) in das Datenbyte \$aa (RIDATA, RS Input DATA) eingebunden wird.

RSRABL (\$ef7e): RS232-Empfang initialisieren

Diese Routine initialisiert außer dem Interrupt-Kontrollregister von CIA 2 und der korrespondierenden Adresse \$02a1 (ENABL, RS232-Flag) auch das Startbitflag RINONE (\$a9).

RSRTTRT (\$ef90): Startbitprüfung

Soll ein Startbit erwartet werden, obwohl keines im Hilfsspeicher INBIT (\$a7, eingelesenes Bit von RS232) vorhanden ist, so initialisiert der Aufruf von RSRABL (\$ef7e) erneut den RS232-Empfang.

Bei gefundenem Startbit wird das Startbitflag gelöscht und die Eingabeparität initialisiert, wofür die Routine bei \$e4d3 – also an ganz anderer Adresse – fortfährt (siehe auch dort).

\$ef97: Byte in RS232-Empfangspuffer übernehmen

Sind alle Bits von RS232 eingelesen worden, so kommt diese Routine ins Spiel. Sie prüft zunächst, ob noch Platz im RS232-Eingabepuffer vorhanden ist, um frühzeitig den Status für »Eingabepuffer voll« zu erkennen und mitzuteilen. Normalerweise wird allerdings das Byte problemlos in den Eingabepuffer übernommen.

Damit ist es jedoch noch lange nicht getan: Jetzt folgt die Prüfung der gewünschten Parität, die dazu mit der tatsächlich vorhandenen verglichen wird. Im Falle eines Paritätsfehlers wird Bit 0 im RS232-Statusregister gesetzt.

CKORS (\$efe1): CKOUT-Behandlung für RS232

Diese Routine lenkt als Bestandteil von CKOUT (\$ffc9) die Ausgabe auf RS232 um. Dazu wird das gewünschte Ausgabefile (Filenummer eingangs im Akku enthalten) als Ausgabekanal in DFLTO (\$9a) gesetzt und daraufhin je nach Handshake-Form die entsprechenden Signale über RS232 gesendet.

Antwortet das an RS232 angeschlossene Gerät nicht mit einem DSR-Signal (Data Set Ready, deutsch »Datensatz bereit«), führt dies zum MISSING-DSR-Status.

BSORS (\$f014): BSOUT-Behandlung für RS232

Die Ausgabe auf RS232 vollzieht sich in diesem Bestandteil von BSOUT (\$ffd2), indem die NMI-Routine zur Datenausgabe aktiviert

wird (sofern sie dies nicht schon ist) und das auszugebende Datenbyte in den Ausgabepuffer an dessen letzte Position kommt.

Der Offset auf das letzte Byte im Ausgabepuffer (RODBE \$029e, RS Output Data Buffer End, »Ende des RS232-Ausgabedatenpuffers«) wird dabei für jedes neue Byte verringert, wodurch effektiv der Puffer »nach unten« anwächst.

CKIRS (\$f04d): CHKIN-Behandlung für RS232

Dieser Bestandteil von CHKIN (\$ffe6) setzt die gewünschte Filenummer (im Akku übergeben) als Ausgabefilenummer und teilt dies dann durch die entsprechenden Signale dem RS232-Gerät mit, das mit der Meldung DTR (Data Transmission Ready, »Datenübertragung bereit«) antworten muß.

GETRS (\$f086): GETIN-Behandlung für RS232

Das Einholen eines Bytes über GETIN (\$ffe4) für RS232 übernimmt GETRS (\$f086). Dabei wird das nächste Byte aus dem Eingabepuffer eingelesen; ist dieser leer, wird dies im RS232-Statusbyte vermerkt und ein Nullbyte als Ergebnis geliefert.

RSP232 (\$f0a4): Warten auf RS232-Bereitschaft

Um weitere RS232-Operationen zu tätigen, ist oftmals vorauszusetzen, daß RS232 wieder frei verfügbar ist. Auf diesen Zustand lauert die RSP232-Routine, die solange wartet, bis der RS232-NMI nicht mehr aktiv ist. Dann nämlich ist sichergestellt, daß keine RS232-Datenübertragung mehr stattfindet.

\$f0bd–\$f12a:

Tabelle der Systemmeldungen im ASCII-Code

Diese ASCII-Tabelle enthält die Systemmeldungen des Kernals, wobei im letzten Byte jeder Systemmeldung das Bit 7 gesetzt ist. Im ROM-Listing (Kapitel 1) steht in Klammern jeweils der Offset von \$f0bd zum Beginn der jeweiligen Systemmeldung, denn dieser wird an die Routine \$f12b übergeben.

Die ASCII-Tabelle enthält folgende Texte an den angegebenen Adressen:

```
$f0bd = $f0bd+$00 : I/O ERROR #
$f0c9 = $f0bd+$0c : SEARCHING
$f0d4 = $f0bd+$17 : FOR
$f0d8 = $f0bd+$1b : PRESS PLAY ON TAPE
$f0eb = $f0bd+$2e : PRESS RECORD & PLAY ON TAPE
$f106 = $f0bd+$49 : LOADING
$f10e = $f0bd+$51 : SAVING
$f116 = $f0bd+$59 : VERIFYING
$f120 = $f0bd+$63 : FOUND
$f127 = $f0bd+$6a : OK
```

\$f12b: Ausgabe einer Systemmeldung anhand des Offset

Wird der bei \$f0bd–\$f12a erwähnte Offset einer Systemmeldung im Y-Register abgelegt, gibt »jsr \$f12b« den gewünschten Text aus. Befindet sich der Computer allerdings im Programm-Modus, so unterdrückt er die Meldung.

GETIN (\$f13e): Routine zum Kernals-Einsprung \$ffe4

In 3.3.3 wurde GETIN (\$ffe4 – \$f13e) bereits als Beispiel einer universellen Routine dargestellt. Dementsprechend wird auch anhand des aktuellen Eingabegerätes die richtige Teilbehandlung selektiert. Nach allen Vergleichen und Tests kristallisieren sich letztlich folgende Möglichkeiten heraus:

1. GETKB (\$f142): GETIN von Tastatur

Dabei wird zuerst geprüft, ob sich noch mindestens 1 Byte im Tastaturpuffer befindet; bei leerem Tastaturpuffer wird \$00 zurückgegeben (ASCII-Code für »keine Taste«). Andernfalls wird der Interrupt abgestellt (SEI), damit beim darauffolgenden Einlesen des nächsten Zeichens aus dem Tastaturpuffer in der NXTKEY-Routine (\$e5b4) keine Schwierigkeiten auftreten.

2. \$f14e: GETIN von RS 232

Zuerst wird das Y-Register in \$97 gerettet und danach die GETRS-Routine (\$f086) aufgerufen, die das nächste Byte aus dem RS232-Eingabepuffer holt. Vor dem Rücksprung aus der Routine bekommt das Y-Register noch seinen Ausgangswert aus \$97 wieder.

3. \$f166: Behandlung weiterer Geräte (IEC-Bus, Bildschirm, Datensette)

Dafür wird in die BASIN-Teilroutine von \$f166 an eingestiegen, da sich BASIN- und GETIN-Behandlung für IEC-Bus, Bildschirm und Datensette in keiner Weise unterscheiden. Eine Besprechung finden Sie bei der BASIN-Dokumentation, die sich nun anschließt:

BASIN (\$f157): Routine zum Kernals-Einsprung \$ffcf

Auch BASIN (\$ffcf → \$f157) ist eine universelle Routine, d.h. sie funktioniert mit verschiedenen Standard-Eingabegeräten und paßt sich diesen bereitwillig an. Dazu wird anhand von Vergleichen die jeweilige Routine ausgesondert:

1. BINKB (\$f15b): BASIN von Tastatur

Dabei wird die aktuelle Cursorposition als Eingabe-Cursorposition gesetzt und ein Zeichen von dieser Position am Bildschirm in den Akkumulator geholt, wofür SCRGET (\$e632) herangezogen wird.

2. BGISC (\$f16a): BASIN (oder auch GETIN) vom Bildschirm
Dabei wird das INPUT/GET-Flag gesetzt und ein Zeichen über SCRGET (\$e632) von der aktuellen Bildschirmposition geholt, die zuvor durch Übertragung von LNMX (\$d5) nach INDX (\$c8) definiert wurde.
3. \$f179: BASIN (oder auch GETIN) von Datasette
Dazu wird das nächste Byte aus dem Kassettenpuffer in den Akkumulator geholt; ist der Kassettenpuffer bereits vollständig ausgelesen, so wird zuerst ein neuer Datenblock von Kassette in den Puffer geholt. Dies erledigt die JTGET-Unterroutine (\$f199). Ein File-Ende wird nach dem JTGET-Aufruf erkannt und durch Setzen des EOF-Status im Statusbyte des Kernal berücksichtigt. Die JTGET-Routine folgt zwar direkt im Speicher, wird aber aus Gründen der Übersichtlichkeit erst nach Sonderbehandlung (4) beschrieben.
4. BSIRS (\$f1b8): BASIN von RS232
Für die BASIN-Eingabe von RS232 wird zunächst die GETRS-Behandlung aufgerufen. Tritt dabei ein anderes Byte als \$00 auf, wird dieses als Ergebnis zurückgegeben. Andernfalls wird die Endmarkierung mit einer Sonderbehandlung quittiert, in welcher der Wert \$0d (CR, Carriage Return) zurückgegeben wird, wie Sie es von BASIN (\$ffc6 → \$f157) kennen.

JTGET (\$f199):

nächstes Byte aus Kassettenpuffer in Akkumulator holen

Diese Routine erhöht den Zeiger auf das nächste Byte im Kassettenpuffer, da dieser um 1 Byte entleert werden soll. In der Regel wird das dadurch gewonnene Byte als Ergebnis verwendet. Ist der Puffer jedoch inhaltslos, so liest ein Aufruf von RBLK (\$f841) den nächsten Datenblock von Kassette in den Puffer ein. Danach initialisiert JTGET (\$f199) den BUFPNT-Offset für den Kassettenpuffer wieder mit \$00 und ruft sich dann selbst auf, um gleich das erste Byte des neu eingelesenen Datenblockes zu verwenden.

BSOUT (\$f1ca): Routine zum Kernal-Einsprung \$ffd2

Dies ist die bei weitem am häufigsten verwendete Betriebssystem-routine. Sie unterscheidet in ihrer Abarbeitung nach dem aktuellen Ausgabegerät:

1. SCROUT (\$e716): BSOUT auf Bildschirm
Dazu wird die SCROUT-Routine von \$f1d2 aus aufgerufen.
2. IECOUT (\$eddd): BSOUT auf IEC-Bus
Hierfür ist die IECOUT-Routine wegen \$f1d8 zuständig.
3. \$f1e5: BSOUT auf Datasette
Hier wird das Ausgabebyte in den Kassettenpuffer geschrieben – sofern da noch Platz ist. Ist der Kassettenpuffer bereits voll, wird

er zuerst auf die Kassette »entleert« und nimmt dann das neue Ausgabezeichen gleich als erstes Byte auf.

4. \$f017: BSOUT auf RS232

Die Adresse \$f017 ist ein Einsprung in BSORS (\$f014); durch den um 3 Byte späteren Einsprung, der von \$f208 aus erfolgt, wird das Starten der RS232-NMI-Routine übersprungen.

CHKIN (\$f20e): Routine zum Kernal-Einsprung \$ffc6

Damit die Eingabe von einem beliebigen File erfolgen kann, wird dieses File als aktuelles Eingabefile über CHKIN (\$ffc6 → \$f20e) eingestellt. Existiert kein File mit der in X enthaltenen Nummer, tritt der I/O ERROR #3 (FILE NOT OPEN) auf.

Dem Betriebssystem selbst genügt dabei, wenn die Filenummer nach DFLTIN (\$99) kommt; in manchen Ausnahmen will aber auch das anzusprechende Gerät informiert sein.

1. CKIRS (\$f04d): CHKIN für RS232

Diese Routine ist an der entsprechenden Stelle dokumentiert und wird aus CHKIN (\$ffc6 → \$f20e) bei \$f227 über JMP angesprungen.

2. CKITAP (\$f22a): CHKIN für Datasette

Bei Datasetteneingaben ist zuerst sicherzustellen, daß auch eine Eingabe-Sekundäradresse vorliegt. Andernfalls erfolgt die Meldung NOT INPUT FILE (I/O ERROR #6).

Die Datasette selbst, die ja kein »intelligenter Massenspeicher« wie die Floppy ist und nur vom Computer gesteuert wird, nimmt keinerlei Kenntnis von der CHKIN-Ausführung.

3. SDFLTIN (\$f233): CHKIN durch Setzen von DFLTIN; für Tastatur/Bildschirm

Diese Routine dient auch für die anderen CHKIN-Behandlungen als Routinenabschluß. Bei Tastatur oder Bildschirm genügt jedoch SDFLTIN (\$f233) völlig.

4. CKISER (\$f237): CHKIN für IEC-Bus

Dazu wird zunächst das TALK-Signal gesendet; bei einer Geräteadresse mit gesetztem Bit 7 wird zusätzlich »ATN high« übermittelt. Dies ist bei Druckern die Aufforderung, am Ende jeder Zeile den LF-Code (\$0a) zu senden.

In jedem Fall kommt dann die Sekundäradresse auf den seriellen Bus.

CKOUT (\$f250): Routine zum Kernal-Einsprung \$ffc9

Damit die BSOUT-Ausgabe auf ein anderes Gerät erfolgen kann, muß auf diesem ein geeignetes Ausgabefile existieren, dessen Filenummer im X-Register übermittelt wird. Ist dieses File noch nicht oder nicht mehr in der Filetabelle vorhanden, entsteht der I/O ERROR #3 (FILE NOT OPEN).

Andernfalls wird je nach gewünschtem Ausgabegerät die richtige Sonderbehandlung ausgelöst:

1. CKORS (\$efel): CKOUT auf RS232
Diese Routine wird bei \$f26c über JMP angesprungen.
2. CKOTAP (\$f26f): CKOUT auf Datasette
Hierbei muß sichergestellt werden, daß es sich um ein Ausgabefile handelt; liegt stattdessen die Sekundäradresse für Eingabefiles vor, wird mit I/O ERROR #7 (NOT OUTPUT FILE) abgebrochen.
3. SDFLTO (\$f275): CKOUT durch Setzen von DFLTO
Diese Behandlung reicht bei Bildschirm und Tastatur aus; sie dient gleichzeitig als Schlußbehandlung der anderen CKOUT-Behandlungen.
4. CKOSER (\$f279): CKOUT auf IEC-Bus
Dazu ergeht die LISTEN-Anforderung an das CKOUT-Gerät, worauf das Senden der Sekundäradresse folgt. War das angesprochene Gerät verfügbar, schließt SDFLTO (\$f275) die CKOSER-Behandlung ab; andernfalls entsteht der I/O ERROR #5 (DEVICE NOT PRESENT).

CLOSE (\$f291): Routine zum Kernal-Einsprung \$ffc3

Ein File mit der im X-Register übergebenen Nummer schließt CLOSE (\$ffc3 → \$f291), indem es zum einen das File aus der Kernal-Filetabelle entfernt, zum anderen aber auch dem Gerät, auf dem das File geöffnet ist, das Schließen des Files mitteilt.

1. \$f2ab: CLOSE für RS232
Dabei wird noch der jeweilige RS232-Puffer (entweder der Ein- oder der Ausgabepuffer) wieder als Datenbereich freigegeben. Auch die CIA-Register werden initialisiert, da sie durch den RS232-NMI erheblich beeinflusst wurden.
2. \$f2c8: CLOSE für Datasette
Hierzu wird noch der letzte Datenblock auf Kassette geschrieben, auf den der Datenblock-Header »End Of Tape« – wenn gewünscht – folgt.
3. \$f642: CLOSE für IEC-Bus
Diese Sonderbehandlung wird von \$f2ee über JSR aufgerufen; dahinter folgt im Speicher die Routine DELFLE (\$f2f1):
4. DELFLE (\$f2f1): CLOSE durch Entfernen des Fileeintrages
Bei Tastatur/Bildschirm reicht diese DELFLE-Behandlung bereits zum korrekten Schließen des Files aus; den Sonderbehandlungen für die anderen Geräte dient DELFLE (\$f2f1) lediglich als Abschluß, nachdem zuvor die Peripherie zum Schließen des Files veranlaßt wurde.

LOOKUP (\$f30f):

Offset einer Filenummer in Filetabelle ermitteln

Sollen die Geräte- und Sekundäradresse zu einem File mit der in X enthaltenen Nummer ermittelt werden, so ist »jsr lookup« dabei eine wertvolle Hilfe. Danach steht nämlich im X-Register nicht mehr die Filenummer, sondern der Offset dieses Files in der Filetabelle des Kernal (s. Kap. 3.3.8).

Gleichzeitig wird das Statusbyte zurückgesetzt.

JLTLK (\$f314): späterer Einstieg in LOOKUP (\$f30c)

Soll das Statusbyte seinen alten Wert beibehalten und nicht dem X-Register, sondern dem Akkumulator die Nummer eines zu suchenden Files entnommen werden, wird anstelle von LOOKUP (\$f30c) der Einsprung JLTLK (\$f314) gewählt.

Auch er liefert im X-Register den Offset des Files innerhalb der Filetabelle.

GETFLS (\$f31f):

Fileparameter anhand des File-Offsets entnehmen

Um aus der Kernal-Filetabelle, die aus den Einzeltabellen LAT (logische Filenummern, \$0259–\$0262), FAT (Geräteadressen, \$0263–\$026c) und SAT (Sekundäradressen, \$026d–\$0276) besteht, zu einem bestimmten File die logische Filenummer, die Geräte- und die Sekundäradresse entnehmen zu können, erwartet diese Routine im X-Register den File-Offset, der in der Regel von den Routinen LOOKUP (\$f30f) und JLTLK (\$f314) berechnet wird.

CLALL (\$f32f): Routine zum Kernal-Einsprung \$ffe7

Diese Routine löscht die Kernal-Filetabelle, indem sie die Anzahl der offenen Files mit 0 beziffert. Somit wird kein File mehr vom Kernal als offen betrachtet. Auch die eingestellten Ein-/Ausgabegeräte werden zurückgesetzt, wobei auf den IEC-Bus sogar das entsprechende Signal (UNLISTEN oder UNTALK) gesendet wird. Ein Schließen der Files seitens der Peripheriegeräte erfolgt also nicht, d.h. die Floppy würde eines der berüchtigten »Sternchen-Files« (*PRG, *SEQ, ...) erzeugen.

CLALL (\$ffe7 → \$f32f) ist also kein vollwertiger Ersatz für CLOSE (\$ffc3 → \$f291); normalerweise setzt man CLALL (\$ffe7 → \$f32f) gleich am Anfang eines Programms zum Aufräumen der Filetabelle ein.

OPEN (\$f34a): Routine zum Kernal-Einsprung \$ffc0

Obwohl diese Routine keine Parameter in Registern erwartet – die Filespezifikationen wurden durch die Routinen SETLFS und SETNAM vorbereitet –, hat OPEN (\$ffc0 → \$f34a) von allen Kernal-Einsprüngen unbestritten das reichhaltigste Angebot an potentiellen Fehlerquellen. Dies sind die drei interessantesten Möglichkeiten:

- I/O ERROR #6 (NOT INPUT FILE), wenn 0 als Filenummer angegeben wurde
- /O ERROR #2 (FILE OPEN), wenn das zu öffnende File bereits in der Filetabelle steht
- I/O ERROR #1 (TOO MANY FILES), wenn die Kapazität der Filetabelle bereits voll ausgeschöpft ist

Normalerweise liegt jedoch kein Problem vor, und die Fileparameter können in die Filetabelle übertragen werden. Dabei entsteht mit einer einzigen Ausnahme keine Veränderung: Bit 5 und 6 in der Sekundäradresse werden gesetzt.

Daraufhin sind dann gerätespezifische Sonderbehandlungen erforderlich, wenn es sich beim anzusprechenden Gerät nicht um Ta-statur oder Bildschirm handelt.

1. \$f37f: OPEN auf IEC-Bus

Dazu wird die Routine IECOPN (\$f3d5) aufgerufen, die an gegebener Stelle in diesem Kapitel beschrieben ist.

2. \$f388: OPEN auf RS 232

Hierfür wird RSOPEN (\$f409) angesprungen; die Dokumentation finden Sie am dazugehörigen Platz.

3. \$f38b: OPEN auf Datasette

Hierbei tritt ein I/O ERROR #9 (ILLEGAL DEVICE NUMBER) auf, wenn kein Kassettenpuffer existieren sollte, d.h. wenn der herkömmliche Kassettenpuffer blockiert ist.

Danach wird eine Unterscheidung zwischen Schreib- und Lesefiles getroffen:

3a. \$f399: OPEN für Lesefiles auf Datasette

Hierzu wird auf das Drücken der PLAY-Taste am Datenrekorder gewartet, wozu den Benutzer der Text »PRESS PLAY ON TAPE« auffordert.

Auch die Meldung »SEARCHING« mit eventueller Angabe des Filenamens wird daraufhin ausgegeben. Liegt kein Filename vor (Länge = 0), so wird das nächste File auf Datasette geöffnet (Behandlung ab \$f3af), wobei der Header-Block eingelesen und angezeigt wird. Wurde ein bestimmter Filename angegeben, so wird nach diesem gesucht; ist er nicht vor der End-Of-Tape-Markierung zu finden, löst dies einen I/O ERROR #4 (FILE NOT FOUND) aus.

3b. \$f3b8: OPEN für Schreibfiles auf Datasette

Hierzu wird nach Warten auf <RECORD & PLAY> der Anfangsblock des Files mit der Headermarke \$04 auf Band geschrieben. Der Offset für den Kassettenpuffer wird hinter diesen Header gesetzt, damit der Header vor dem Überschreiben durch Filedaten geschützt ist.

IECOPN (\$f3d5): File auf IEC-Bus öffnen

Diese Unteroutine von OPEN (\$ffc0 → \$f34a) bricht bei einer Sekundäradresse über \$7f – also wenn Bit 7 gesetzt ist – ab, weil dann kein Filename gesendet werden muß. Dies ist natürlich auch dann der Fall, wenn der Filename fehlt, da die Länge des Filenamens mit 0 angegeben ist.

Andernfalls wird über LISTEN (\$ed0c) und SECOND (\$edb9) das Gerät angesprochen und in einer IECOUT-Schleife der Filename byteweise übermittelt, bis schließlich das UNLISTEN-Signal die Übertragung abschließt.

RSOPEN (\$f409): File auf RS232 öffnen

Wie IECOPN (\$f3d5), ist auch RSOPEN (\$f409) eine Unteroutine von OPEN (\$ffc0 → \$f34a). Sie initialisiert zunächst die CIA-Register für die RS232-Datenübertragung und entnimmt dann dem Filenamens die Übertragungsparameter (Kontroll-, Befehls-, Zeit- und Statusregister). Dadurch sind sämtliche Einstellungen (z.B. Übertragungsrate) eindeutig definiert, oder können zumindest daraus berechnet werden (z.B. Anzahl der Bits pro Wort).

Ist allerdings kein Filename vorhanden, werden die Standardeinstellungen verwendet.

Danach wird der richtige Timer-Verzögerungswert (abhängig von Baudrate und C64-Version – PAL oder NTSC) eingestellt.

Mit am wichtigsten ist die Schlußbehandlung (ab \$f45c), in welcher je nach Bedarf ein Ein- oder Ausgabepuffer im Speicher gebildet wird.

ICIARS (\$f483):

CIA-Register nach RS232-Betrieb initialisieren

Um die CIA-Register nach den Veränderungen durch den RS232-NMI wieder in den Initialisierungszustand zu versetzen, ist diese Routine vorhanden.

Sie initialisiert auch ENABL (\$02a1), einen Hilfsspeicher der RS232-Routinen, der mit dem Interrupt-Kontrollregister korrespondiert.

LOAD (\$f49e): Routine zum Kernaleinsprung \$ffd5

Zunächst wird die Ladeadresse, also die Adresse, an welche das File geladen werden soll, aus X/Y in den Zeiger \$c3/\$c4 übertragen. Dann erfolgt ein Sprung über den Vektor ILOAD (\$0330/\$0331) zur standardmäßig vorgesehenen Routine \$f4a5.

Diese LOAD/VERIFY-Routine merkt sich als erstes die gewünschte Arbeitsweise als Flag in \$93: Wurde im Akku \$00 übergeben, soll ein File geladen werden, ansonsten soll es verifiziert, also mit einem File auf Diskette oder Kassette verglichen werden.

Dann wird in Abhängigkeit vom Ladegerät fortgefahren, wobei ein Ladevorgang von Tastatur, RS232 oder Bildschirm mit der Mel-

dung I/O ERROR #9 (ILLEGAL DEVICE NUMBER) »abgeschmettert« wird.

Es bleiben also nur zwei Sonderbehandlungen übrig: eine für IEC-Bus-Geräte und eine für Datasette.

1. \$f4b8: LOAD/VERIFY für IEC-Bus

Zunächst wird ein nicht vorhandener Filename mit I/O ERROR #8 (MISSING FILENAME) quittiert, weil dieser nur bei Datasette optional ist.

Regulär folgt jedoch die Ausgabe der Meldung »SEARCHING FOR <name>«. Dann wird das einzulesende File über IECOPN (\$f3d5) mit Sekundäradresse \$60 (0 + Offset) am IEC-Bus geöffnet, belegt jedoch keinen Eintrag in der Filetabelle des Kernals, da die eigentliche OPEN-Routine (\$ffc0) bewußt gemieden wird.

Daraufhin wird das Ladegerät durch das TALK-Signal und die Sekundäradresse zum Senden von Daten aufgefordert. Ist beim Auslesen der absoluten Ladeadresse des Programms nach \$ae/\$af keine Antwort des angesprochenen Gerätes feststellbar, wird der I/O ERROR #4 (FILE NOT FOUND) mitgeteilt.

Bevor nun die eigentliche Datenübertragung beginnen kann, muß noch die effektive Anfangsadresse des Lade-/Verifizier-Vorgangs definiert sein, die entweder die in X/Y übergebene »relative« Ladeadresse (gespeichert in \$c3/\$c4) oder die im File enthaltene »absolute« Ladeadresse (hier in \$ae/\$af enthalten) ist.

Darüber entscheidet einzig und allein die Sekundäradresse: Ist diese 0, so wird die relative Ladeadresse aus \$c3/\$c4 nach \$ae/\$af übertragen. Dieser Hilfszeiger \$ae/\$af weist im weiteren Verlauf immer auf das aktuelle Byte im Speicher.

Nach Ausgabe der LOADING-Meldung wird in einer Schleife das File byteweise in den Speicher geholt beziehungsweise mit dem Speicher verglichen. Am Ende des Files (EOF) wird durch Senden von UNTALK und anschließendes IECCLS (\$f642) der Ladevorgang beendet.

Abbildung 4.28 ist ein Flußdiagramm der LOAD/VERIFY-Routine, an dem vor allem die Schleifenstruktur von LOAD (\$ffd5 → \$f49e) leicht nachvollziehbar ist.

2. \$f539: LOAD/VERIFY für Datasette

Die Meldung I/O ERROR #9 (ILLEGAL DEVICE NUMBER) tritt gleich zu Beginn auf, wenn der Kassettenpuffer nicht verfügbar ist.

Normalerweise wird aber das Öffnen eines Files auf Kassette wie über OPEN (\$ffc0) bewirkt, wobei jedoch das Eintragen in die Filetabelle umgangen wird. Ist dann erst einmal das einzulesende File offen, kann die Anfangsadresse des einzulesenden Programms bei einer Headermarke \$03 (Maschinenprogramm) aus dem Headerblock nach \$c3/\$c4 entnommen werden. Bei relativem Laden (andere Headermarke als \$03) wird jedoch die in \$c3/\$c4 enthaltene Adresse nicht mehr geändert, sondern gilt als endgültige Ladeadresse.

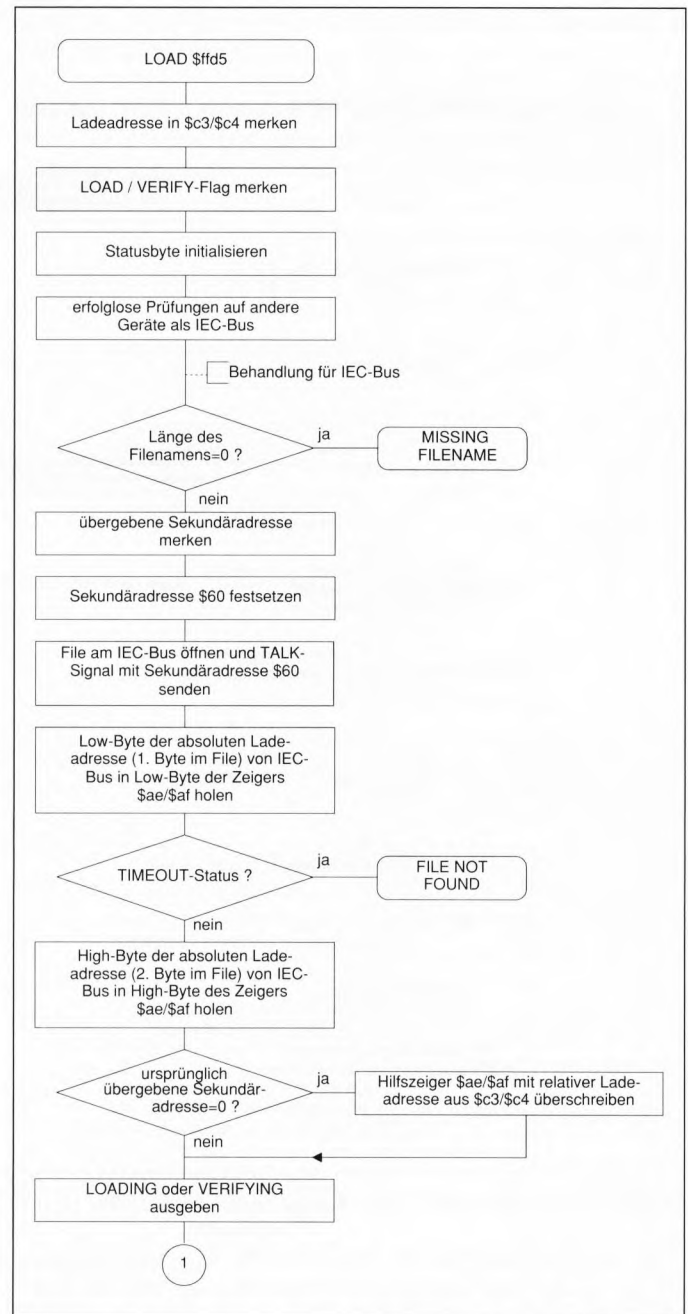


Abbildung 4.28: Die LOAD/VERIFY-Routine für den IEC-Bus (Teil 1)

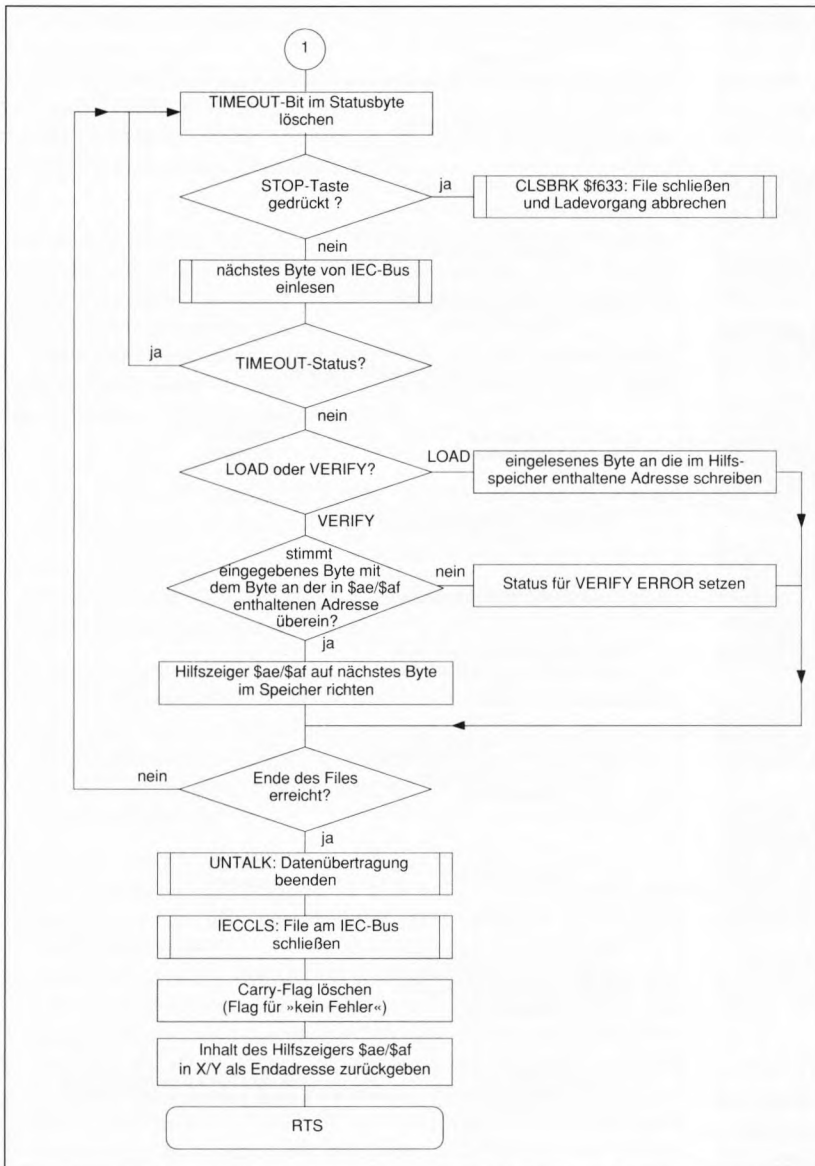


Abbildung 4.28: Die LOAD/VERIFY-Routine für den IEC-Bus (Teil 2)

In jedem Fall wird dann die Endadresse für den Ladevorgang aus der im Headerblock angegebenen Programmlänge und der Ladeadresse errechnet und in \$ae/\$af gemerkt.

Dann wird nach Ausgabe der LOADING-Meldung ein File über die TPREAD-Routine (§f84a, siehe dort) eingelesen und an-

schließend die Endadresse des Ladevorgangs aus \$ae/\$af zur Rückgabe nach X/Y eingeholt.

SRCMSG (§f5af): Ausgabe der SEARCHING-Meldung im Direktmodus

Beim Laden/Verifizieren eines Programms muß dieses zuerst gesucht werden. Da dieser Vorgang einige Zeit dauern kann, wird der Anwender durch SEARCHING- und LOADING-Meldungen auf dem laufenden gehalten. Diese Routine dient zur Ausgabe der SEARCHING-Meldung, die jedoch nur im Direktmodus ausgegeben wird; im Programm-Modus erfolgt ein Rücksprung ohne Textausgabe.

Auf jeden Fall wird der Text »SEARCHING« gedruckt; stellt sich dann noch heraus, daß ein Filename existiert, so kommt zusätzlich der Text »FOR« auf den Bildschirm, worauf die byteweise Ausgabe des Filenamens folgt.

Durch Eingabe von »SYS 62895« (#62895 = \$f5af) kann man somit diese Meldung erzwingen, um den letzten verwendeten Filenamens zu erfahren – ein ganz netter Trick, der nicht nur Einsteigern nützliche Dienste erweist.

LOADNG (§f5d2): Ausgabe von LOADING oder VERIFYING

Da LOAD und VERIFY in einer einzigen Routine zusammengefaßt sind, muß bei der Ausgabe der Steuermeldung zwischen LOADING und VERIFYING unterschieden werden. Genau dies erledigt LOADNG (§f5d2) unter Verwendung der Routine \$f12b (Ausgabe einer Systemmeldung); ausschlaggebend ist dabei das LOAD/VERIFY-Flag 93.

SAVE (§f5dd): Routine zum Kernaleinsprung \$ffd8

Beim Abspeichern eines Programmes werden die Parameter wie folgt übergeben:

- SETLFS (§ffba) und SETNAM (§ffbd) definieren das File.
- Bei »jsr save« enthält das Registerpaar X/Y die Endadresse des abzuspeichernden Bereiches (plus 1, da die Adresse in X/Y die erste nicht mehr zu speichernde Adresse darstellen soll) und der Akkumulator eine Zeropage-Adresse, ab welcher wiederum die Anfangsadresse des SAVE-Bereiches im Low-High-Format steht.

Die Endadresse legt die SAVE-Routine zunächst in \$ae/\$af und die Anfangsadresse in \$c1/\$c2 ab. Dann wird über den Vektor ISAVE

(\$0332) verzweigt, der in unverändertem Zustand nach \$f5ed führt. Dort wird je nach gewünschtem Gerät eine Sonderbehandlung aufgerufen, wobei jedoch Tastatur, RS 232 und Bildschirm mit I/O ERROR #9 (ILLEGAL DEVICE NUMBER) quittiert werden. Für Datasette und Diskettenlaufwerk (am IEC-Bus), die beiden SAVE-Medien, sind also unterschiedliche Routinen erforderlich. Die IEC-Bus-Routine ist dabei erheblich kürzer, da das DOS (Disk Operating System, »Diskettenbetriebssystem«) die Arbeit des Kernals auf das Senden der Bytes reduziert.

1. \$f5fa: SAVE auf IEC-Bus

Als Sekundäradresse wird hierzu \$61 festgelegt (Sekundäradresse für Speichervorgänge). Liegt jedoch kein Filename vor, entsteht der I/O ERROR #8 (MISSING FILENAME), da der Filename nur bei Datasette optional ist. Andernfalls wird ein File am IEC-Bus geöffnet (IECOPN \$f3d5), wozu jedoch kein Eintrag in der Filetabelle erforderlich ist, und die Meldung »SAVING <filename>« ausgegeben. Danach ergeht die LISTEN-Aufforderung mitsamt Sekundäradresse \$61 an das entsprechende Gerät. Zur Initialisierung vor der SAVE-Schleife wird außer dem Y-Offset für die indizierte Adressierung noch der Hilfszeiger \$ac/\$ad, der immer auf das aktuelle Byte im Speicher weist, mit der Startadresse initialisiert, die sich in \$c1/\$c2 bereits vor dem Sprung über den SAVE-Vektor befunden hat. Vor der SAVE-Schleife kommen noch das Low- und das High-Byte der Startadresse als Anfang des Files auf den IEC-Bus, um im File eine absolute Ladeadresse anzugeben. Die SAVE-Schleife schreibt dann den Speicherbereich byteweise auf Diskette, wobei nach jedem Byte eine Abfrage der STOP-Taste erfolgt. Abbildung 4.29 stellt die SAVE-Routine für den IEC-Bus grafisch dar.

Als Unterprogramm zum Schließen eines Files dient dabei

IECCLS (\$f642): File am IEC-Bus schließen

Durch den Einsprung bei \$f63f wird zusätzlich vor dem eigentlichen Schließen des Files das UNLISTEN-Signal gesendet.

2. \$f65f: SAVE auf Datasette

Als erstes wird geprüft, ob der Kassettenpuffer verfügbar ist; wenn nicht, wird die Meldung I/O ERROR #9 (ILLEGAL DEVICE NUMBER) ausgelöst. Nach dem Warten auf <RECORD & PLAY> und der Meldung »SAVING <filename>« wird die richtige Headermarke aus der Sekundäradresse ermittelt:

Sekundäradresse \$00 oder \$02: Headermarke \$01 (Anfang eines Basic-Programms)

Sekundäradresse \$01 oder \$03: Headermarke \$03 (Anfang eines Maschinenprogramms)

Diese Headermarke wird durch Aufruf der Routine TAPEHE (\$f76a) auf Kassette geschrieben, woraufhin die Daten des Files abgespeichert werden. Bei einer Sekundäradresse \$02 oder \$03 folgt

darauf das Anbringen der End-Of-Tape-Markierung, was wiederum TAPEHE (\$f76a) übernimmt.

SAVING (\$f68f): Ausgabe der Systemmeldung SAVING

Diese Hilfsroutine gibt außer dem Text »SAVING« auch den Filenamen aus, wofür allerdings in SRCMSG (\$f5af) bei \$f5c1 eingestiegen wird.

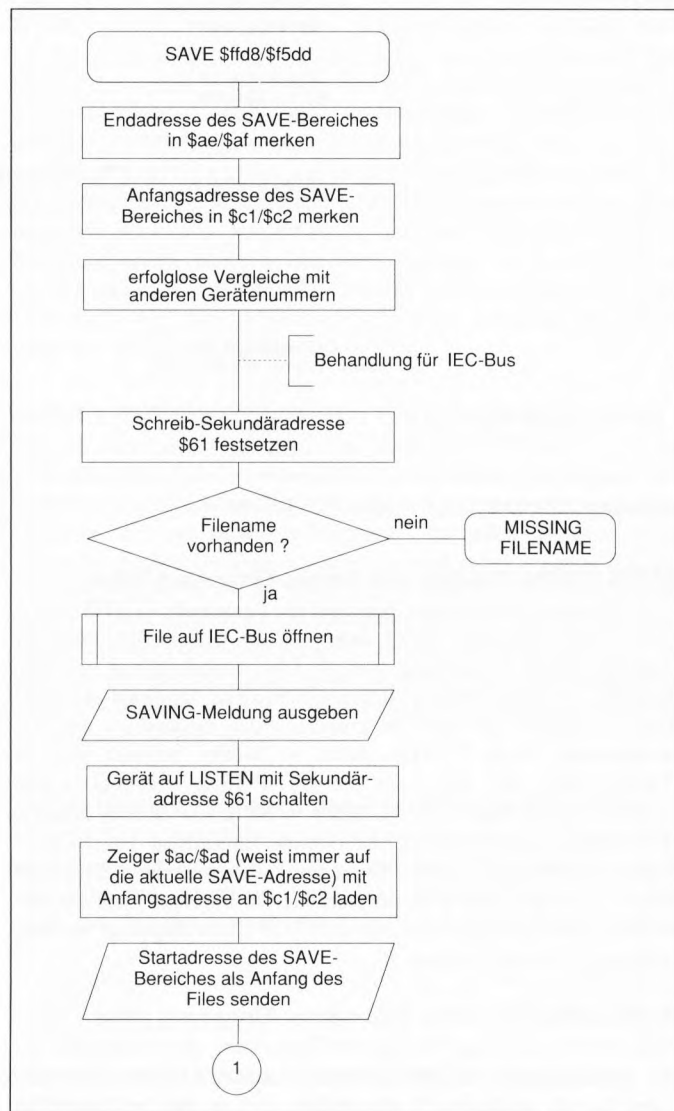


Abbildung 4.29: Die SAVE-Routine für den IEC-Bus (Teil 1)

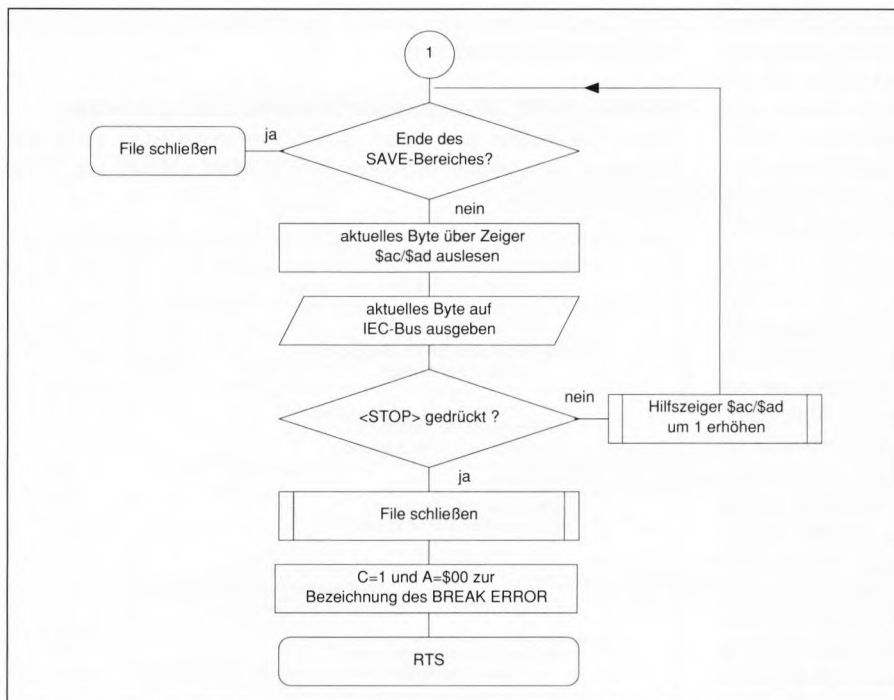


Abbildung 4.29: Die SAVE-Routine für den IEC-Bus (Teil 2)

UDTIM (\$f69b): Routine zum Kernaleinsprung \$f69b

Diese Routine inkrementiert zunächst die Systemuhr (TI/TIS) um 1 »jiffy« (1/50 Sekunde). Wird dabei auf 24 Uhr erhöht, muß der Übertrag durch Umstellung auf 0 Uhr berücksichtigt werden. Daraufhin wird auf jeden Fall die STOP-Taste abgefragt; UDTIM (\$f69b → \$f69b) ist nämlich grundlegender Bestandteil des Systeminterrupt. Nach UDTIM (\$f69b → \$f69b) befindet sich in STKEY (\$91) das Flag, ob die STOP-Taste gedrückt wurde (STKEY = \$7f) oder nicht (STKEY = \$00). Die Kernaleinsprung STOP (\$f6e1) stützt sich auf die richtige Einstellung von STKEY (\$91); UDTIM (\$f69b → \$f69b) ist somit Voraussetzung für STOP (\$f6e1). Soll in einem selbstprogrammierten Interrupt nicht die Systemuhr weitergezählt, wohl aber die STOP-Taste abgefragt werden, so ist bei \$f6bc einzusteigen.

RDTIM (\$f6dd): Routine zum Kernaleinsprung \$f6dd

Nach Setzen des Interrupt-Disable-Flags kann die Systemuhr als Jiffy-Anzahl (Anzahl in 1/60 Sekunden) nach A/X/Y geholt werden, wobei A das niedrigst-, X das mittel- und Y das höchstwertige Register ist. Um einen eigenen RTS-Befehl für RDTIM (\$f6dd → \$f6dd) »wegzurationalisieren«, folgt im Speicher SETTIM (\$f6db →

\$f6e4), wo die soeben eingelesene Zeit erneut gesetzt wird. Auch das wiederholte Setzen des Interrupt-Flags bleibt dann ohne Wirkung, wichtig sind nur das Löschen des Interrupt-Flags bei \$f6eb sowie der RTS-Befehl bei \$f6ec.

SETTIM (\$f6e4):**Routine zum Kernaleinsprung \$f6db**

Nach Setzen des Interrupt-Disable-Flags kann die Systemuhr als Jiffy-Anzahl (Anzahl in 1/60 Sekunden) gemäß A/X/Y gesetzt werden, wobei A das niedrigst-, X das mittel- und Y das höchstwertige Register ist. Anschließend wird das Interrupt-Flag wieder gelöscht und über RTS an die aufrufende Routine zurückgekehrt.

STOP (\$f6ed):**Routine zum Kernaleinsprung \$f6e1**

Ist STKEY (\$91) im Interrupt auf \$7f (Flag für »<STOP> gedrückt«) gestellt worden, wird von STOP (\$f6e1 → \$f6ed) Z=1 und C=1 zurückgegeben, die Filetabelle gelöscht und der Tastaturpuffer als »leer« gekennzeichnet. Wurde <STOP> nicht gedrückt, so wird Z=0 und C=0 zurückgegeben. Still-

schweigend wird die interruptgesteuerte STOP-Abfrage in der UDTIM-Routine (\$f69b → \$f69b) vorausgesetzt.

\$f6fb–\$f72b: Ausgabe der Meldung I/O ERROR #. . .

Hierfür stehen zunächst neun verschiedene Einsprünge für die neun verschiedenen I/O-ERRORs zur Verfügung. Diese Einsprünge laden jeweils die richtige Fehlernummer in den Akkumulator und überspringen durch BIT-Tricks die darauffolgenden LDA-Befehle. Nach Initialisierung der I/O-Geräte und Ausgabe der Systemmeldung »I/O ERROR #« wird die Fehlernummer durch Umwandlung in den ASCII-Code ausgegeben. Vor dem RTS-Rücksprung wird noch das Carry-Flag gesetzt (Flag für »Fehler aufgetreten«) und die Fehlernummer (zuvor auf den Stapel gerettet) in den Akkumulator geladen.

Hier eine selbsterklärende Aufstellung der einzelnen I/O-ERROR-Einsprünge:

```

IOERR1 ($f6fb): I/O ERROR #1 (too many files)
IOERR2 ($f6fe): I/O ERROR #2 (file open)
IOERR3 ($f701): I/O ERROR #3 (file not open)
IOERR4 ($f704): I/O ERROR #4 (file not found)
IOERR5 ($f707): I/O ERROR #5 (device not present)
  
```



```
IOERR6 ($f70a): I/O ERROR #6 (not input file)
IOERR7 ($f70d): I/O ERROR #7 (not output file)
IOERR8 ($f710): I/O ERROR #8 (missing filename)
IOERR9 ($f713): I/O ERROR #9 (illegal device
                  number)
```

GETFHD (\$f72c):

nächsten Header auf Datasette öffnen und melden

Diese Hilfsroutine liest zuerst den nächsten Block in den Kassettenpuffer ein; anhand der Headermarke wird dann festgestellt, ob es sich um einen Anfangsblock zu einem File handelt oder nicht:

```
Header $01: Anfang eines Basic-Programms → FOUND-Meldung
Header $02: Fortsetzungsblok eines Files → weiter suchen
Header $03: Anfang eines Maschinen- → FOUND-Meldung
           programms
Header $04: Anfang eines Datenfiles → FOUND-Meldung
Header $05: End Of Tape (Band-Ende) → Abbruch der Routine
```

Hinter der FOUND-Meldung, deren Ausgabe bei \$f74b beginnt, wird noch der Filename byteweise ausgedruckt, wobei jedoch nur die ersten 16 Zeichen berücksichtigt werden; weitere Zeichen können aber durchaus im Kassettenpuffer stehen! Manche Programme verwenden diese Inkonsistenz des Betriebssystems als Kopierschutz, indem sie einen Filenamen, der länger als 16 Zeichen ist, bis auf das letzte Byte abfragen.

Vor dem Rücksprung wird übrigens noch auf Drücken von <CBM> oder <SPACE> in einem begrenzten Zeitraum gewartet.

TAPEHE (\$f76a): WBLK-Routine für den Anfangsblock eines Programmfiles

Diese Routine ist nicht mit WBLK (\$f864) zu verwechseln! Bei \$f76a wird lediglich eine im Akkumulator angegebene Headermarke mit Anfangs- und Endadresse eines SAVE-Bereiches in den Kassettenpuffer geschrieben, in den auch der Filename übertragen wird. Daraufhin wird der Kassettenpuffer als Datenblock auf Kassette abgelegt.

GETBFA (\$f7d0):

Adresse des Kassettenpuffers holen und testen

GETBFA (\$f7d0) liest in X- und Y-Register die Anfangsadresse des Kassettenpuffers ein. Ist der Kassettenpuffer blockiert, so ist als Anfangsadresse ein Wert unter \$0200 angegeben; um dies festzustellen, wird das High-Byte (Y-Register) unmittelbar vor dem Rücksprung mit dem High-Byte von \$0200, nämlich 2, verglichen. Anhand des Carry-Flags ermittelt eine aufrufende Routine also, ob der Kassettenpuffer frei ist (C=1) oder nicht (C=0).

BFSAEA (\$f7d7):

Kassettenpuffer als Speicherbereich festsetzen

Beim Schreiben oder Lesen von Daten auf Kassette spielen die Routinen TPREAD (\$f84a) und TPWRIT (\$f870) eine elementare Rolle; sie erwarten in \$c1/\$c2 den Anfang und in \$ae/\$af das Ende eines Speicherbereiches, der auf Band geschrieben werden soll. Dies ist entweder ein Speicherbereich, in welchem ein Programm liegt, oder aber der Kassettenpuffer. Für die Routinen RBLK (\$f841) und WBLK (\$f864), welche den Kassettenpuffer einlesen bzw. schreiben, dient BFSAEA (\$f7d7) dabei als Unterprogramm zum Stellen der Zeiger \$c1/\$c2 und \$ae/\$af auf Anfang und Ende des Kassettenpuffers, damit die anschließende Ausführung von TPREAD/TPWRIT das gewünschte Ergebnis bringt.

BFSAEA (\$f7d7) stützt sich zunächst auf GETBFA (\$f7d0), um die Anfangsadresse zu ermitteln und schreibt dann deren Low-Byte nach \$c1 (Low-Byte von \$c1/\$c2); dazu wird \$c0, die Größe des Kassettenpuffers, addiert, und das Ergebnis als Low-Byte von \$ae/\$af gesetzt. Ein eventueller Übertrag wird nach dem Setzen des High-Bytes der Anfangsadresse (\$c2) durch Erhöhung des High-Bytes der Endadresse (\$af) berücksichtigt.

SRCTFL (\$f7ea): vorgegebenes File auf Datasette suchen

Um ein durch den Filenamen definiertes File auf Band zu finden, wird SRCTFL (\$f7ea) aufgerufen. Diese Routine liest solange den nächsten Header von Datasette ein, bis der darin enthaltene Filename mit dem aktuellen Filenamen in allen Bytes übereinstimmt. Eine End-Of-Tape-Markierung wird daran erkannt, daß der GETFHD-Aufruf bei \$f7ea außer dem gesetzten Carry (positives Ergebnis beim Vergleich der Headermarke mit \$05) auch die Headermarke \$05 im Akkumulator mitteilt; da diese Kombination aus C=1 und A=5 für das aufrufende Programm praktischerweise gleichzeitig einen I/O ERROR #5 (FILE NOT FOUND) bezeichnet, genügt ein einfacher RTS-Rücksprung.

TBFUL (\$f80d): Test auf freien Platz im Kassettenpuffer

Diese Hilfsroutine erhöht den Offset auf das aktuelle Byte im Kassettenpuffer; ist der maximal zulässige Offset (\$bf) schon überschritten, wird mit gesetztem Carry-Flag zurückgekehrt, woran die aufrufende Routine erkennt, daß kein Platz mehr im Kassettenpuffer frei ist.

Auf jeden Fall steht in X/Y danach die Adresse des Kassettenpuffers.

WTPLAY (\$f817): Warten auf PLAY-Taste der Datasette

WTPLAY (\$f817) prüft, ob die PLAY-Taste an der Datasette gedrückt ist; wenn ja, wird mit gelöschtem Carry-Flag zurückgekehrt, andernfalls mit C=1 und A=0, wenn durch Auslösen der STOP-

Taste am Computer ein Abbruch (BREAK ERROR) erzwingen wurde.

Der WTPLAY-Ablauf wird durch die Meldungen »PRESS PLAY ON TAPE« und »OK« für den Anwender kommentiert.

TSPLAY (\$f82e): PLAY-Taste an Datasette prüfen

Diese Routine liefert ein gelöscht Zero-Flag, wenn die PLAY-Taste noch nicht gedrückt ist; andernfalls wird Z=1 zurückgegeben.

WTRCPL (\$f838):

<RECORD & PLAY> an Datasette erwarten

Zum Abspeichern von Daten ist es erforderlich, daß der Anwender die Tasten <RECORD> und <PLAY> an der Datasette einrasten läßt. Dies prüft die vorliegende Routine, die sich von WTPLAY (\$f817) nur darin unterscheidet, daß der Text »PRESS RECORD & PLAY ON TAPE« statt »PRESS PLAY ON TAPE« ausgegeben wird. Ansonsten springt WTRCPL (\$f838) sogar in WTPLAY (\$f817) ein.

Dadurch wird von WTRCPL (\$f838) auch zurückgekehrt, wenn nur die PLAY-, nicht aber die Aufnahmetaste eingerastet ist. Da es allerdings beim besten Willen keine Softwarelösung für die Prüfung der RECORD-Taste gibt, darf man dem Betriebssystem nichts vorwerfen.

RBLK (\$f841):

Datenblock von Datasette in Kassettenpuffer einlesen

Das Einlesen eines Datenblocks in den Kassettenpuffer vollzieht sich, indem der Kassettenpuffer als aktueller Speicherbereich an TPREAD (\$f84a; folgt direkt im Speicher) übergeben wird. Zusätzlich wird das Statusbyte des Kernals initialisiert und das LOAD/VERIFY-Flag auf »LOAD« gestellt.

TPREAD (\$f84a):

Datenbereich von Datasette in Speicher einlesen

Der nächste Datenblock wird von TPREAD (\$f84a) in den Speicherbereich eingelesen, der folgendermaßen definiert ist:

```
$c1/$c2 : Anfangsadresse
$ae/$af : Endadresse (+1)
```

Dabei werden die Hilfszeiger für die Interrupt-Routine initialisiert, so daß daraufhin die Interruptroutine (READ \$f92c) für Kassetten-Leseoperationen gestartet werden kann. Dies wiederum geschieht durch Schreiben der READ-Adresse \$f92c in den IRQ-Vektor; die Kontrolle über den Interrupt hat die TAPE-Routine ab \$f875 (allgemeine Kassettenbehandlung).

WBLK (\$f864): Datenblock von Kassettenpuffer auf Datasette schreiben

Hierzu wird der Kassettenpuffer als abzuspeichernder Bereich gesetzt und auf <REORD & PLAY> gewartet, woraufhin die TPWRIT-Routine beginnt:

TPWRIT (\$f870):

Datenblock aus Speicher auf Datasette schreiben

Den abzuspeichernden Speicherbereich definieren dieselben Zeiger wie bei TPREAD (\$f84a):

```
$c1/$c2 : Anfangsadresse
$ae/$af : Endadresse (+1)
```

Dann wird er durch Starten der Interrupt-Routine WRTZ (\$fc6a) geschrieben; die Routine WRTZ (\$fc6a) schreibt zunächst die Synchronisation, bis sie auf ihre »Kollegin« WRTN (\$fbcd) zum Schreiben der eigentlichen Daten umschaltet.

TAPE (\$f875): allgemeine Kassettenbehandlung

Diese Routine steuert eine im Interrupt ablaufende Kassettenoperation, deren IRQ-Index im X-Register übergeben wird (\$0e = read \$f92c; \$08 = wrtz \$fc6a).

TSSTOP (\$f8d0): Prüfung der

STOP-Taste der Tastatur während Kassettenoperationen

Diese Routine prüft mittels Aufruf von STOP (\$ffe1), ob die STOP-Taste der Tastatur gedrückt wurde. Wenn nicht, wird über RTS mit Z=0 zurückgekehrt. Andernfalls wird die Kassette gestoppt und dann mit C=1 und A=0 (Flags für BREAK ERROR) fortgefahren.

TSSTOP (\$f8d0) ist also die Ergänzung der Routine STOP (\$fff1) für die speziellen Bedürfnisse der Kassettenroutinen.

SETPIN (\$f8e2): Datasette für Lesevorgang vorbereiten

Dazu wird unter anderem der Timer auf den richtigen Verzögerungswert gestellt. Gleichzeitig wird in die IRQ-Behandlung für die Datasette (\$ff43) eingesprungen.

\$f92c: IRQ-Routine »read«; serielles Lesen von Datasette

Diese IRQ-Routine erledigt die Ladeoperationen von TPREAD (\$f84a), wobei die Daten bitweise anhand von Pulsen erkannt werden. Sind erst einmal alle Bits übertragen, wird daraus ein Byte gebildet und in den Speicher geschrieben. Aus Gründen der Datensicherheit wird dabei jedes Bit zweimal eingelesen, wobei keine Abweichung auftreten darf.

Auch eine Parität dient der Sicherstellung der korrekten Datenübertragung.

Für VERIFY existiert ab \$fadf eine Sonderbehandlung.

Insgesamt erstreckt sich diese Routine bis \$fb8d (!). Daran erkennt man schon, wie speicherplatzaufwendig die Kassettenroutinen sind, da ja die Datasette kein eigenes Betriebssystem wie die Floppy mit ihrem DOS (Disk Operating System) hat, und kann gut verstehen, warum viele Floppy-Speeder auf Kosten der Kassettenroutinen programmiert werden und dadurch Speicherplatz für unglaublich viele Funktionen gewinnen können.

STACUR (\$fb8e): Hilfszeiger \$ac/\$ad initialisieren

Während des Ablaufes solcher I/O-Operationen wie LOAD und SAVE weist der Zeiger \$ac/\$ad immer auf das aktuelle Byte im Speicher. Diese Routine initialisiert ihn mit der Anfangsadresse, die dazu in \$c1/\$c2 enthalten sein muß.

NEWCH (\$fb97):

Register für serielles Lesen und Schreiben initialisieren

Diese Hilfsroutine initialisiert einige Hilfsppeicher, die für Kassettenoperationen – genauer gesagt: deren IRQ-Routinen – die Grundlage bilden, wie etwa das Paritätsbyte, um nur ein Beispiel zu nennen.

\$fba6–\$fbcc:Kassetten-Unterroutinen

Die an diesen Adressen liegenden Routinen senden eine Flanke (0- oder 1-Bit) beziehungsweise schließen eine Operation ab. Sie dienen als Unterprogramme für die Schreibe-IRQ-Routinen.

\$fbcd: IRQ-Routine »wrtn«

Diese Routine zum Schreiben eines Speicherbereiches auf Kassette erstreckt sich insgesamt bis \$fc69; sie arbeitet in zwei Durchläufen (Passes), da jedes Bit zweimal geschrieben wird.

Vor den Daten muß eine Synchronisation stehen, die eine andere IRQ-Routine auf Datasette sendet:

\$fc6a: IRQ-Routine »wrtz«

Vor irgendwelchen Daten auf Kassette ist immer eine Synchronisation erforderlich, die in WRTZ (\$fc6a) zunächst geschrieben wird; danach schaltet WRTZ (\$fc6a) automatisch auf die IRQ-Routine WRTN (\$fbcd) um, damit auch die Daten selbst auf die Kassette gelangen.

STPTAP (\$fc93): Kassettenbetrieb beenden

Um den Kassettenbetrieb zu beenden, genügt es STPTAP (\$fc93) nicht, nur den Motor auszuschalten. Auch die CIA-Register müssen wieder initialisiert sowie die normale IRQ-Routine eingeschaltet werden.

\$fcb8: Einsprung

zum Beenden des Kassettenbetriebs mit IRQ-Abschluß

Dieser Einsprung beendet über STPTAP (\$fc93) den Kassettenbetrieb und springt dann zum IRQ-Abschluß für Kassettenoperationen (\$fc54).

BSIV (\$fcbd): IRQ-Vektor gemäß IRQ-Offset setzen

Diese Routine erwartet im X-Register einen Offset, um dann die gewünschte IRQ-Routine zu starten. Dabei sind folgende Offsets möglich:

\$08 = wrtz \$fc6a: Synchronisation vor Daten auf Kassette schreiben

\$0a = wrtn \$fbcd: Daten auf Kassette schreiben

\$0c = nirq \$ea31: normaler IRQ

\$0e = read \$f92c: Daten von Kassette lesen

Der Offset wird dabei von \$fd93 an gezählt und bezieht sich somit auf eine ROM-Tabelle ab \$fd9b (\$fd93+\$08 = Basis + niedrigster Offset).

TAPMOF (\$fcc): Motor der Datasette ausschalten

Der Datasettenmotor wird durch Setzen von Bit 5 im Prozessorport (Adresse \$0001) ausgeschaltet.

CMPSTE (\$fcd1): Vergleich von \$ac/\$ad mit \$ae/\$af

Um die beiden Hilfszeiger \$ac/\$ad (aktuelle I/O-Adresse) und \$ae/\$af (I/O-Endadresse) zu vergleichen, dient diese Routine. Ist danach das Carry-Flag gesetzt, so ist die I/O-Endadresse bereits erreicht, ansonsten ist das Carry gelöscht.

INCSAL (\$fcd): Hilfszeiger \$ac/\$ad erhöhen

Dadurch wird auf die nächste I/O-Adresse geschaltet, welche ja jeweils der Zeiger \$ac/\$ad enthält.

RESET (\$fce2): Kaltstart des Betriebssystems

Diese Routine wird bei einem Hardware-Reset, dem Einschalten des Computers (das ein Reset-Signal auslöst) oder dem softwaremäßigen Start über »SYS 64738« durchlaufen. Sie initialisiert alle Komponenten des Computers (Speicher, Bausteine) und löst dann einen Basic-Start aus. Liegt jedoch eine sogenannte Modulkennung »CBM80« ab Adresse \$8003 vor, so wird das Modul über den Vektor \$8000/\$8001 gestartet. Abbildung 4.30 beschreibt den Ablauf eines Reset.

Im folgenden finden Sie die Dokumentationen zu den einzelnen Unterprogrammen der RESET-Routine, die auch im Speicher unmittelbar folgen.

CHKCBM (\$fd02): Prüfung auf Modulkennung "CBM80"

Diese Routine liefert ein gesetztes Zero-Flag, wenn ab \$8003 die Modulkennung »CBM80« (\$c3 \$c2 \$cd \$38 \$30) steht; ist keine solche Kennung vorhanden, wird Z=0 zurückgegeben. Als Vergleichspunkt dient folgende Tabelle:

\$fd10–\$fd14: CBM80-Tabelle

Die Tabelle enthält die ASCII-Codes

\$c3 \$c2 \$cd \$38 \$30

»C«, »B« und »M« sind also als Großbuchstaben gespeichert.

RESTOR (\$fd15): Routine zum Kernal-Einsprung \$ff8a

Dieser Einsprung lädt die Adresse der Initialisierungstabelle \$fd30 für die Vektoren \$0314–\$0333 zur Übergabe an VECTOR (\$ff8d → \$fd1a):

VECTOR (\$fd1a): Routine zum Kernal-Einsprung \$ff8d

Dieser Routine wird in X/Y die Adresse einer Initialisierungstabelle übertragen. Ist C=0, so wird diese Initialisierungstabelle in die Vektoren \$0314–\$0333 geschrieben; bei C=1 werden hingegen die Vektoren \$0314–\$0333 ab der angegebenen Adresse abgelegt. Die Behandlung des Carry-Flags in der Schleife ist jedoch nicht korrekt. Für den Fall nämlich, daß die Initialisierungstabelle in die Vektoren soll (C=0), wird bei \$fd25 ein Byte aus der Tabelle entnommen – so weit, so gut. Dann allerdings wird dieses Byte nicht nur bei

\$fd29 in den Vektorenbereich \$0314–\$0333, sondern auch zuvor bei \$fd27 »in sich selbst« geschrieben – also an die Adresse, von welcher das Byte zuvor ausgelesen wurde. Solange eine Initialisierungstabelle im RAM steht, ist dies nicht weiter von Belang: Effektiv ändert sich die Initialisierungstabelle nicht, es wird auch kein anderer Speicherbereich fälschlicherweise manipuliert. Liegt jedoch, wie etwa beim RESTOR-Einsprung \$fd15, die Initialisierungstabelle im ROM, so wird ein Byte bei \$fd25 aus dem ROM gelesen, aber bei \$fd27 fälschlicherweise ins RAM an gleicher Adresse geschrieben, da der STA-Befehl nicht das ROM, sondern nur das RAM ansprechen kann. Befanden sich nun an den entsprechenden Stellen im RAM unter dem ROM wichtige Daten, sind diese nach Ausführung von VECTOR (\$fd1a) verloren! Deshalb soll man die RESTOR-Routine, die auf VECTOR basiert, laut »GEOS Programmer's Reference Guide« (Programmierhandbuch zu GEOS; GEOS ist das neuartige Betriebssystem für den C64 mit der grafischen Benutzeroberfläche) unter GEOS nicht verwenden, da das GEOS-Betriebssystem im RAM liegt und sonst Schaden nehmen könnte.

\$fd30–\$fd4f: Initialisierungstabelle für Vektoren \$0314–\$0333

Diese Tabelle wird von RESTOR (\$ff8a → \$fd15) an VECTOR (\$ff8d → \$fd1a) übergeben.

RAMTAS (\$fd50): RAM-Bereich ermitteln

Um die Obergrenze des verfügbaren RAM zu ermitteln, wird von dieser Routine die erste ROM-Adresse gesucht, also die erste Adresse, deren Inhalt nicht über Schreibzugriffe der CPU geändert werden kann. Zu beachten ist dabei, daß an der RAM-Adresse unter der ersten ROM-Speicherzelle – normalerweise \$a000 – ein Prüfbyte steht, anhand dessen die Schreib-/Lese-Fähigkeit der Adresse getestet wurde. Weitere Leistungen dieser Routine bestehen im

- Löschen des Bereiches \$0002–\$03ff
- Setzen der ersten ROM-Adresse als Speicher-Obergrenze für Basic
- Festlegen von \$0800 als Anfang des Basic-RAM
- Positionieren des Bildschirmspeichers bei \$0400

Da allerdings die RAM-Testschleife jedes einzelne Byte von \$0800–\$a000 testet, dauert die Ausführung eine zwar geringe, aber dennoch merkbare Zeit; diese Verzögerung ist dafür verantwortlich, daß nach einem Reset oder Einschalten der C64 nicht sofort verfügbar ist!

\$fd9b–\$fda2: Tabelle der Adressen der IRQ-Routinen

Diese Tabelle wird nur von BSIV (\$fcbd) verwendet.

IOINIT (\$fda3): Routine zum Kernal-Einsprung \$ff84

Hier werden die CIA-Register initialisiert; der Timer wird dabei je nach PAL- oder NTSC-Version des C64 geladen (siehe \$fddd–\$fdf8).

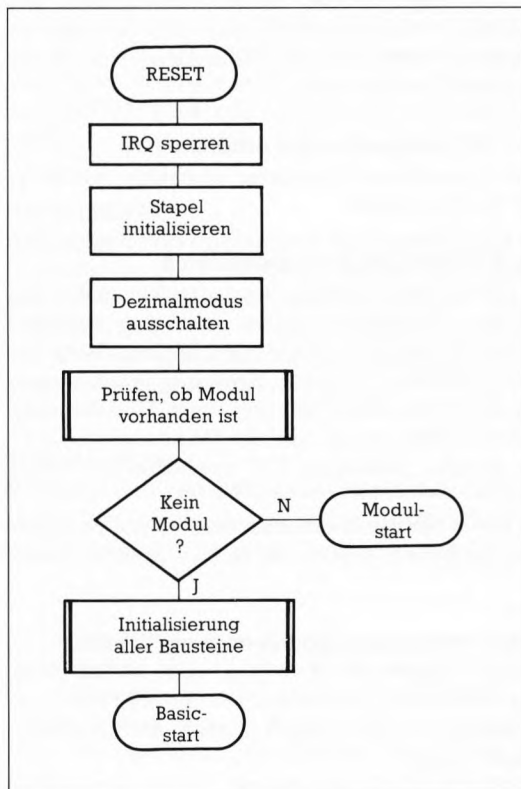


Abbildung 4.30: Diesem Weg folgt ein RESET

SETNAM (\$fdf9): Routine zum Kernal-Einsprung \$ffb7

Der Akkumulator wird in \$b7 als Länge des Filenames (FNLEN) und die Adresse des Filenames im Speicher in \$bb/\$bc als FNADR gesetzt.

SETLFS (\$fe00): Routine zum Kernal-Einsprung \$ffb7

SETLFS (\$fe00), oft auch SETPAR genannt, setzt den Akkumulator als logische Filenummer in LA (\$b8), das X-Register als Gerätenummer in FA (\$ba) und das Y-Register als Sekundäradresse in SA (\$b9).

READST (\$fe07): Routine zum Kernal-Einsprung \$ffb7

Die Funktion der beiden Statusbytes für IEC-/Datasetten- und RS232-Betrieb haben Sie in 3.3.6 kennengelernt. Diese Routine liest das zur aktuellen Geräteadresse gehörige Statusbyte aus und gibt es im Akkumulator zurück.

SETMSG (\$fe18): Routine zum Kernal-Einsprung \$ff90

Der Akkumulator wird hier als Flag für die Ausgabe von Systemmeldungen gesetzt. Dabei gibt es folgende vier Einstellungsmöglichkeiten:

- \$00 = keine Meldung
- \$40 = I/O ERROR #x
- \$80 = Steuermeldungen wie LOADING
- \$c0 = alle Meldungen (\$80+\$40)

Um einen RTS-Befehl einzusparen, folgt die READST-Behandlung für IEC-Bus/Datasette, die als Nebeneffekt in den Akkumulator das Statusbyte ST (\$90) lädt.

\$fe1a: READST-Behandlung für anderes Gerät als RS232

Dazu wird ST (\$90) in den Akkumulator ausgelesen. Um einen RTS-Befehl zu sparen, folgt der ERSTAT-Einsprung, der jedoch keine Ergebnisveränderung mit sich bringt.

ERSTAT (\$fe1c): Fehlerbits aus Akku in Statusbyte ST einblenden

Steht im Akku ein Fehlerbit (oder auch mehrere), wird dieses/werden diese in das Statusbyte ST (Adresse \$90) eingebunden. Sind sie bereits gesetzt, ändern sie sich nicht mehr. Dann erfolgt ein Rücksprung, wobei der Akkumulator das neue Statusbyte enthält.

SETTMO (\$fe21): Routine zum Kernal-Einsprung \$ffa2

SETTMO (\$ffa2 → \$fe21) schreibt den Akkumulator in das TIMEOUT-Flag.

MEMTOP (\$fe25): Routine zum Kernal-Einsprung \$ff99

Bei gelöschtem Carry-Flag wird die in X/Y enthaltene Adresse als Speicherobergrenze gesetzt, bei gesetztem Carry hingegen die Speicherobergrenze nach X/Y geholt. Der relevante Zeiger ist MEMSIZ (\$0283/\$0284).

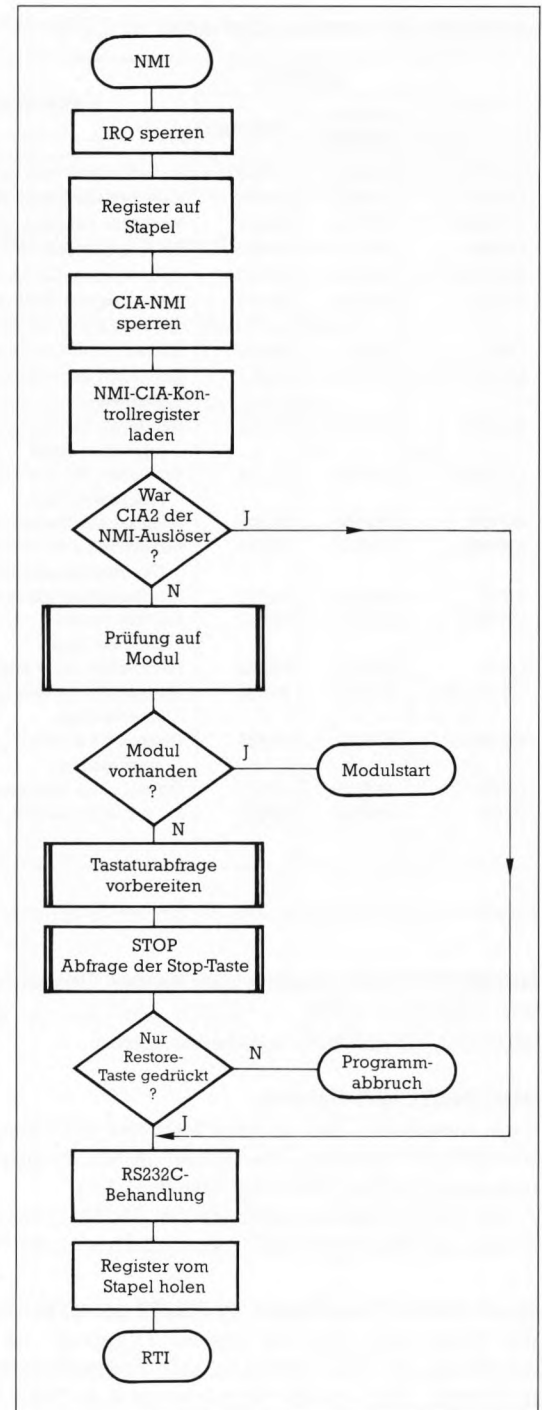


Abbildung 4.31: Flußdiagramm zum Ablauf einer NMI-Unterbrechung

AUFRUFBARE KERNAL-ROUTINEN

NAME	ADRESSE		FUNKTION
	HEXA-DEZIMAL	DEZIMAL	
ACPTR	\$FFA5	65445	Byte-Eingabe zum seriellen Port
CHKIN	\$FFC6	65478	Kanal für Eingabe öffnen
CHKOUT	\$FFC9	65481	Kanal für Ausgabe öffnen
CHRIN	\$FFCF	65487	Zeicheneingabe
CHROUT	\$FFD2	65490	Zeichenausgabe
CIOUT	\$FFA8	65448	Byte-Ausgabe über den seriellen Bus
CINT	\$FF81	65409	Bildschirm-Editor-Initialisierung
CLALL	\$FFE7	65511	Schließen aller Kanäle und Dateien
CLOSE	\$FFC3	65475	Schließen einer bestimmten logischen Datei
CLRCHN	\$FFCC	65484	Schließen der Ein- und Ausgabekanäle
GETIN	\$FFE4	65508	Zeichen aus Tastaturpuffer lesen
IOBASE	\$FFF3	65523	Basisadreß-Rückmeldung der Ein-/Ausgabegeräte
IOINIT	\$FF84	65412	Ein-/Ausgabeinitialisierung
LISTEN	\$FFB1	65457	LISTEN-Befehl für Geräte am seriellen Bus
LOAD	\$FFD5	65493	RAM laden von Peripherie
MEMBOT	\$FF9C	65436	Unteren Speicherzeiger lesen/setzen
MEMTOP	\$FF99	65433	Oberen Speicherzeiger lesen/setzen
OPEN	\$FFC0	65472	Öffnen einer logischen Datei
PLOT	\$FFF0	65520	X-, Y-Cursorposition lesen/setzen→

NAME	ADRESSE		FUNKTION
	HEXA-DEZIMAL	DEZIMAL	
RAMTAS	\$FF87	65415	RAM initialisieren, Kassettenpuffer einrichten, Bildschirm auf \$0400 setzen
RDTIM	\$FFDE	65502	Uhrzeit lesen
READST	\$FFB7	65463	Ein-/Ausgabestatuswort lesen
RESTOR	\$FF8A	65418	Standard Ein-/Ausgabevektoren rückstellen
SAVE	\$FFD8	65496	RAM-Inhalt auf Peripheriegerät abspeichern
SCNKEY	\$FF9F	65439	Tastatur abfragen
SCREEN	\$FFED	65517	X-, Y-Bildschirmaufbau ermitteln
SECOND	\$FF93	65427	Sekundäradresse nach LISTEN übertragen
SETLFS	\$FFBA	65466	Logische, erste und Sekundäradresse setzen
SETMSG	\$FF90	65424	KERNAL-Meldungen steuern
SETNAM	\$FFBD	65469	Dateinamen festlegen
SETTIM	\$FFDB	65499	Uhrzeit setzen
SETTMO	\$FFA2	65442	Zeitsperre für seriellen Bus setzen
STOP	\$FFE1	65505	Stop-Taste abfragen
TALK	\$FFB4	65460	TALK-Befehl für Geräte am seriellen Bus
TKSA	\$FF96	65430	Sekundäradresse nach TALK übertragen
UDTIM	\$FFEA	65514	Uhrzeit inkrementieren
UNLSN	\$FFAE	65454	UNLISTEN-Befehl für seriellen Bus
UNTLK	\$FFAB	65451	UNTALK-Befehl für seriellen Bus
VECTOR	\$FF8D	65421	Abspeichern von RAM

Tabelle 4.3: Alphabetische Liste der Kernal-Einsprünge

MEMBOT (\$fe34): Routine zum Kernal-Einsprung \$ff9c

Wie MEMTOP (\$ff99 → \$fe25), aber für die in MEMSTR (\$0281/\$0282) enthaltene Speicheruntergrenze.

NMI (\$fe43): NMI-Routine

Nach Ausschalten aller anderen Interrupts wird über den Vektor \$0318/\$0319 verzweigt. Den Ablauf in der normalerweise angesprungenen Routine \$fe47 zeigt Abbildung 4.31.

Mit allen Sonderbehandlungen für RS232 erstreckt sich diese Routine bis \$ff40; zwischendrin liegt jedoch noch eine Tabelle:

\$fec2–\$fed5: Baud-Raten für RS232 bei NTSC-Version

Wie \$e4ec–\$e4ff, aber für die NTSC-Version. Da diese etwas schneller als die PAL-Version ist, aber dieselbe Geschwindigkeit erzielt werden soll, sind die Timerkonstanten in \$fec2–\$fed5 größer,

um den Timer noch stärker zu verlangsamen, damit im Endeffekt die Übertragungsgeschwindigkeit von PAL- und NTSC-Version übereinstimmt.

\$ff41–\$ff42: Füllbefehle

Diese NOPs zeigen keine Wirkung.

TPIRQ (\$ff43): IRQ-Einsprung für Datasettenbehandlung

Dieser Einsprung wird nur von \$f927 aus verwendet, um die IRQ-Routine für Kassettenoperationen zu aktivieren. Hier wird das BREAK-Flag gelöscht und dann der normale IRQ abgearbeitet:

\$ff48: IRQ-Routine

Hier werden sowohl IRQs als auch BREAKs bearbeitet. Anhand von Bit 4 im Prozessorstatus wird über den jeweiligen Vektor gesprungen:

Bit 4 = 1: Sprung über \$0314/\$0315 zur IRQ-Behandlung (\$ea31)
Bit 4 = 0: Sprung über \$0316/\$0317 zur BREAK-Behandlung (\$fe66)

CINT (\$ff5b): Routine zum Kernal-Einsprung \$ff81

Hier erfolgt eine gründliche Initialisierung des Editors sowie des Timers. Die Routinen INTSCR (\$e518) und IOINIT (\$fda3) werden dabei ganz bzw. teilweise als Unterprogramme eingesetzt.

\$ff6e: Timer-Initialisierung für IRQ

Dies ist die Fortsetzung von \$fdf6 aus, wo die CIA-Register für den Interrupt vorbereitet werden.

\$ff80: Füllbyte

Dieses Byte ist von Version zu Version des C64 verschieden. Im C64-Modus des C128 steht hier \$03.

\$ff81–\$fff5: Kernal-Sprungtabelle

Eine alphabetische Auflistung der Kernal-Routinen ist Tabelle 4.3.

\$fff6–\$fff9: Füllbytes

Diese ASCII-Tabelle der Buchstabenfolge »RRBI« enthält möglicherweise die Initialen zu zwei Programmierernamen. Einen Einfluß auf die ROM-Programme hat sie jedoch nicht.

\$fffa/\$fffb: NMI-Vektor für den Prozessor

Zeigt nach \$fe43.

\$fffc/\$fffd: RESET-Vektor für den Prozessor

Zeigt nach \$fce2.

\$fffe/\$ffff: IRQ-Vektor für den Prozessor

Zeigt nach \$ff48.

Kapitel 5

Die ROM-Routinen im Überblick

Dieses Kapitel besteht aus einer kurzen Zusammenfassung aller ROM-Routinen in der Reihenfolge, in welcher sie in Kapitel 4 beschrieben sind.

a) Basic-Interpreter (\$a000–\$e4d2)

\$0073	CHRGET	nächstes Zeichen aus Basic-Text holen
\$0079	CHRGOT	letztes Zeichen aus Basic-Text holen
\$a000	ROM-Vektor	ROM-Vektor für Basic-Kaltstart
\$a002	ROM-Vektor	ROM-Vektor für Basic-NMI
\$a004	Tabelle	ROM-Kennung
\$a00c	Tabelle	Adressen für die Befehlsroutinen
\$a052	Tabelle	Adressen für die Funktionsroutinen
\$a080	Tabelle	Prioritäten und Adressen der Operatoren
\$a09e	Tabelle	Basic-Schlüsselwörter im Klartext
\$a19e	Tabelle	Basic-Fehlermeldungen im Klartext
\$a328	Tabelle	Adressen der Fehlermeldungstexte
\$a364	Tabelle	Texte für Fehler-/Steuermeldungen
\$a38a	SRCSTK	Suche der Stapeleinträge des Interpreters
\$a3b8	–	Bereitstellung von Variablenspeicher
\$a3bf	BLTUC	Speicherblockverschiebung
\$a3fb	GETSTK	Prüfung auf ausreichenden Stapelspeicherplatz
\$a408	GETFVM	Prüfung/Bereitstellung von Basic-Speicher
\$a435	–	OUT OF MEMORY ERROR
\$a437	ERROR	Fehlerbehandlung
\$a480	MAIN	Warmstart
\$a49c	–	Einbindung einer Basic-Zeile aus dem Eingabepuffer
\$a533	TLNKPRGT	Linkpointer-Neuberechnung
\$a560	TGETSYBT	Eingabe in Systemeingabepuffer holen
\$a579	TCRUNCHT	Tokenisierung des Eingabepuffers
\$a613	FNDLIN	Basic-Zeile im Speicher suchen
\$a642	NEW	Routine zum Basic-Befehl NEW
\$a644	NEWIN	Einstieg in NEW für eigene Programme

\$a659	NEWCLR	Initialisierung von CHRGET-Zeiger und Variablen
\$a65e	CLR	Routine zum Basic-Befehl CLR
\$a68e	STXTPT	CHRGET-Zeiger auf Basic-Programm-anfang stellen
\$a69c	LIST	Routine zum Basic-Befehl LIST
\$a742	FOR	Routine zum Basic-Befehl FOR
\$a7ae	INTPRT	Interpreterschleife
\$a81d	RESTORE	Routine zum Basic-Befehl RESTORE
\$a82c	BSTOP	Behandlung der STOP-Taste während Basic-Ablauf
\$a82f	STOP	Routine zum Basic-Befehl STOP
\$a831	END	Routine zum Basic-Befehl END
\$a857	CONT	Routine zum Basic-Befehl CONT
\$a871	RUN	Routine zum Basic-Befehl RUN
\$a883	GOSUB	Routine zum Basic-Befehl GOSUB
\$a8a0	GOTO	Routine zum Basic-Befehl GOTO
\$a8d2	RETURN	Routine zum Basic-Befehl RETURN
\$a8f8	DATA	Routine zum Basic-Befehl DATA
\$a8fb	ADCGPT	CHRGET-Zeiger um Y-Register erhöhen
\$a906	GOSNXT	Offset zum nächsten Befehlsende ermitteln
\$a909	GOSEND	Offset zum nächsten Basic-Zeilende ermitteln
\$a928	IF	Routine zum Basic-Befehl IF
\$a94b	ON	Routine zum Basic-Befehl ON
\$a96b	LINGET	Zeilennummer aus Basic-Text auswerten
\$a9a5	LET	Routine zum Basic-Befehl LET
\$aa1d	STRCGT	Zeichen aus Basic-String holen
\$aa80	–	Routine zum Basic-Befehl PRINT#
\$aa86	CMD	Routine zum Basic-Befehl CMD
\$aaa0	PRINT	Routine zum Basic-Befehl PRINT
\$aaca	–	Fortsetzung von GETSYB (\$a560)
\$ab1e	STROUT	beliebigen String ausgeben
\$ab21	PRTSTR	aktuellen String ausgeben
\$ab24	–	günstigster Einsprung in STROUT (\$ab1e) !

\$ab3b	RGTSPC	<CRSR RIGHT> oder <SPACE> ausgeben
\$ab3f	SPCOUT	<SPACE> ausgeben
\$ab41	RGTOUT	<CRSR RIGHT> ausgeben
\$ab44	QUMOUT	Fragezeichen ausgeben
\$ab47	BBSOUT	Basic-BSOUT-Behandlung
\$ab4d	–	Fehlerbehandlung bei INPUT, READ und GET
\$ab7b	GET	Routine zum Basic-Befehl GET
\$aba5	–	Routine zum Basic-Befehl INPUT#
\$abbf	INPUT	Routine zum Basic-Befehl INPUT
\$ac06	READ	Routine zum Basic-Befehl READ
\$ad1e	NEXT	Routine zum Basic-Befehl NEXT
\$ad8a	FRMNUM	numerischen Ausdruck auswerten
\$ad8d	CHKNUM	ausgewerteten Ausdruck auf »numerisch« prüfen
\$ad8f	CHKSTR	ausgewerteten Ausdruck auf »String« prüfen
\$ad90	CHKTYP	ausgewerteten Ausdruck auf Datentyp untersuchen
\$ad9e	FRMEVL	beliebigen Ausdruck auswerten
\$ae33	FACSTK	FAC aus Stapel legen
\$ae83	EVAL	nächsten Ausdrucksbestandteil auswerten
\$aed4	NOT	Routine zum Basic-Operator NOT
\$aef1	BRCEVL	Ausdruck in Klammern auswerten
\$aef7	CHKBCL	Prüfung auf schließende Klammer
\$aefa	CHKBRO	Prüfung auf öffnende Klammer
\$aefd	CHKCOM	Prüfung auf Komma
\$aeff	CHKBYT	Prüfung auf beliebiges Byte
\$af08	SYNERR	SYNTAX ERROR
\$af14	–	Prüfung auf Sondervariable
\$af28	GETVAR	Variable aus Basic-Text holen
\$afe6	OR	Routine zum Basic-Operator OR
\$afe9	AND	Routine zum Basic-Operator AND
\$b081	DIM	Routine zum Basic-Befehl DIM
\$b08b	FNDVAR	Variable aus Basic-Text auswerten und suchen
\$b113	CHKLTR	Prüfung auf Buchstabencode im Akkumulator
\$b11d	–	Anlegen einer noch nicht verwendeten Variablen
\$b194	FIRARY	Berechnung der Adresse des ersten Array-Elementes
\$b1a5	Konstante	MFLPT-Darstellung von –32768
\$b1aa	–	FAC in Integerzahl umwandeln
\$b1b2	INTEVL	Integervariable aus Basic-Text auswerten
\$b1d1	–	Sonderbehandlung für Arrayvariablen zu FNDVAR (\$b08b)
\$b357	UMULT	Multiplikation zweier 2-Byte-Integerwerte
\$b37d	FRE	Routine zur Basic-Funktion FRE
\$b391	INTFAC	Integerzahl in Fließkommazahl umwandeln
\$b39e	POS	Routine zur Basic-Funktion POS
\$b3a2	BYTFAC	Bytewert aus Y-Register in FAC übertragen
\$b3a6	CHKDIR	Ausgabe von ILLEGAL DIRECT ERROR im Direktmodus
\$b3b3	DEF	Routine zum Basic-Befehl DEF
\$b3a1	CHKFNS	Prüfung auf FN-Syntax
\$b3f4	FN	Routine zur Basic-Funktion FN
\$b465	STR	Routine zur Basic-Funktion STR\$
\$b475	–	Stringparameter ermitteln, Speicherplatz organisieren
\$b487	STRLIT	String-Rückgabe auf dem temporären Stringstapel
\$b4f4	–	Stringeintrag von vorgegebener Länge anlegen
\$b516	–	bedingte Ausführung der Garbage Collection
\$b526	GARCOL	Garbage Collection
\$b63d	–	Stringverknüpfung
\$b67a	STRVAR	String in Variablenspeicher übernehmen
\$b6a3	–	Prüfung auf Stringausdruck und FRESTR-Ausführung
\$b6a6	FRESTR	String aus Basic-Text weiterverarbeiten
\$b6db	–	Eintrag im temporären Stringstapel löschen
\$b6ec	CHR	Routine zur Basic-Funktion CHR\$
\$b700	LEFT	Routine zur Basic-Funktion LEFT\$
\$b72c	RIGHT	Routine zur Basic-Funktion RIGHT\$
\$b737	MID	Routine zur Basic-Funktion MID\$
\$b761	PREAM	Parameter einer Stringfunktion vom Stapel holen
\$b77c	LEN	Routine zur Basic-Funktion LEN
\$b782	STRPAR	Auswertung eines an Funktion übergebenen Strings
\$b78b	ASC	Routine zur Basic-Funktion ASC
\$b79b	–	Ausführung von CHRGET (\$0073) und GETBYT (\$b79e)
\$b79e	GETBYT	Bytewert aus Basic-Text auswerten
\$b7ad	VAL	Routine zur Basic-Funktion VAL
\$b7eb	GETWRB	2-Byte-Wert, Komma und Bytewert auswerten
\$b7f1	GETCBT	Komma und Bytewert auswerten
\$b7f7	FACWRD	FAC in 2-Byte-Integerwert umwandeln
\$b80d	PEEK	Routine zur Basic-Funktion PEEK
\$b824	POKE	Routine zum Basic-Befehl POKE
\$b82d	WAIT	Routine zum Basic-Befehl WAIT
\$b849	ADD0.5	FAC um 0.5 erhöhen

\$b850	SUBMEM	FAC := Konstante – FAC
\$b853	SUBFAC	FAC := ARG – FAC
\$b862	EQUEXP	FAC und ARG auf gleichen Exponenten bringen
\$b867	ADDMEM	FAC := FAC + Konstante
\$b86a	ADDFAC	FAC := FAC + ARG
\$b8d7	NORMAL	FAC normalisieren
\$b937	SQUEEZ	Additionsübertrag behandeln
\$b947	–	Invertierung des FAC
\$b97e	–	OVERFLOW ERROR
\$b983	–	byteweise Rechtsverschiebung des RES
\$b999	SHIFTR	Rechtsverschiebung eines Fließkomma-Akkumulators
\$b9bf	ROLSHF	Rechtsverschiebung
\$b9bc	Tabelle	Konstanten für die LOG-Funktion
\$b9ea	LOG	Routine zur Basic-Funktion LOG
\$ba28	MEMMULT	FAC := FAC * Konstante
\$ba2b	MULT	FAC := FAC * ARG
\$ba59	MLTPLY	Mantissenbyte in RES einbinden
\$ba8c	MOVMA	ARG := Konstante
\$bab7	–	Exponentenaddition für ARG und FAC
\$bae2	FACM10	FAC verzehnfachen
\$baf9	Konstante	MFLPT-Darstellung von 10
\$baf9	FACD10	FAC := FAC/10
\$bb0f	DIVMF	FAC := Konstante/FAC
\$bb12	DIVAF	FAC := ARG/FAC
\$bb8a	–	DIVISION BY ZERO ERROR
\$bb8f	MOVRF	FAC := RES
\$bba2	MOVMF	FAC := Konstante
\$bbc7	MOVT4	FAC #4 := FAC #1
\$bbca	MOVT3	FAC #3 := FAC #1
\$bbcc	MOVTZ	FAC #1 an Zeropage-Adresse übertragen
\$bbd0	FACVAR	FAC in Variablenspeicher übertragen
\$bbd4	MOVFM	Konstante := FAC
\$bbfc	MOVAF	FAC := ARG
\$bc0c	MOVFA	ARG := FAC
\$bc1b	ROUND	FAC runden
\$bc2b	SIGN	Vorzeichen des FAC in Akku holen
\$bc39	SGN	Routine zur Basic-Funktion SGN
\$bc44	WRDFAC	2-Byte-Wert mit Vorzeichen in FAC bringen
\$bc49	BINFAC	2-Byte-Wert umwandeln, Mantisse #3 und #4 löschen
\$bc4f	SETFAC	FAC-Inhalt nach Registern setzen
\$bc58	ABS	Routine zur Basic-Funktion ABS
\$bc5b	CMPFAC	FAC mit Konstante vergleichen
\$bc9b	FACINT	FAC in Integerzahl umwandeln
\$bccc	INT	Routine zur Basic-Funktion INT
\$bfc3	STRFLP	ASCII-String in Fließkommaformat umwandeln
\$bd7e	ADDAFC	Bytewert zum FAC addieren
\$bdb3	Tabelle	MFLPT-Konstanten zur Umwandlung des FAC in String
\$bdc2	LINOUT	Ausgabe der aktuellen Zeilennummer
\$bdc4	NUMOUT	positive Integerzahl ausgeben
\$bddd	–	Umwandlung des FAC in String ohne Vorzeichen
\$bddf	FLPSTR	FAC in ASCII-String umwandeln
\$bf11	Konstante	MFLPT-Darstellung von 0.5
\$bf16	Tabelle	Mantissen für FLPSTR-Umwandlung
\$bf3a	Tabelle	Mantissen für TISTR-Umwandlung
\$bf52	Füllbytes	–
\$bf71	SQR	Routine zur Basic-Funktion SQR
\$bf78	MEMPOT	FAC := ARG ↑ Konstante
\$bf7b	POTAFAC	FAC := ARG ↑ FAC
\$bfbf	Tabelle	Konstanten für EXP
\$bfed	EXP	Routine zur Basic-Funktion EXP
\$e000	–	Fortsetzung der EXP-Routine
\$e043	POLYX	Auswertung für Polynomtabelle und FAC
\$e059	POLY	Anwendung einer normalen Polynomtabelle auf den FAC
\$e08d	Tabelle	MFLPT-Konstanten für die RND-Funktion
\$e097	RND	Routine zur Basic-Funktion RND
\$e0e3	STRNEX	Rückgabe der im FAC stehenden Mantisse von Funktion
\$e0f9	EREXIT	Fehlerbehandlung nach Interpreter-I/O
\$e10c	–	Teil der BSSOUT-Routine
\$e118	–	ruft spezielle Basic-CHKOUT-Routine auf
\$e11e	BCHKIN	CHKIN für Basic
\$e124	BGETIN	GETIN für Basic
\$e12a	SYS	Routine zum Basic-Befehl SYS
\$e156	–	Routine zum Basic-Befehl SAVE
\$e15f	BSAVE	SAVE für Basic
\$e165	VERIFY	Routine zum Basic-Befehl VERIFY
\$e168	–	Routine zum Basic-Befehl LOAD
\$e1be	–	Routine zum Basic-Befehl OPEN
\$e1c1	BOPEN	OPEN für Basic
\$e1c7	–	Routine zum Basic-Befehl CLOSE
\$e1cc	BCLOSE	CLOSE für Basic
\$e1d4	GETLSV	Parameterauswertung für LOAD, SAVE und VERIFY
\$e200	COMBYT4	Komma und numerischen Parameter auswerten
\$e206	PARTST	Test auf weitere Parameter
\$e20e	CHKCPR	Test auf Komma und weitere Parameter

\$e219	OCLPAR	Parameter für OPEN und CLOSE auswerten
\$e264	COS	Routine zur Basic-Funktion COS
\$e26b	SIN	Routine zur Basic-Funktion SIN
\$e2b4	TAN	Routine zur Basic-Funktion TAN
\$e2e0	abelle	MFLPT-Konstanten für SIN/COS
\$e30e	ATN	Routine zur Basic-Funktion ATN
\$e33e	Tabelle	Polynom für ATN
\$e37b	NMIBAS	NMI-Routine für Basic 2.0
\$e38b	–	ERROR-Routine
\$e394	RESBAS	RESET-Routine für Basic 2.0
\$e3a2	Tabelle	Initialisierungstabelle für CHRGET und SEED
\$e3bf	INITMP	Arbeitsspeicher für Basic initialisieren
\$e422	MSGNEW	Einschaltmeldung ausgeben und NEW ausführen
\$e447	Tabelle	Initialisierungswerte für Vektoren
\$e453	INIVEC	Initialisierung der Vektoren \$0300–\$030b
\$e45f	Füllbyte	–
\$e460	Tabelle	Einschaltmeldung
\$e4ac	Füllbyte	–
\$e4ad	BCKOUT	CKOUT für Basic
\$e4b7	Füllbytes	–

b) Kernal (\$e4d3–\$ffff)

\$e4d3	–	mögliche Fortsetzung von \$ef94
\$e4da	–	mögliche Hilfsroutine von \$ea07
\$e4e0	WATCBM	auf <CBM> oder <SPACE> warten
\$e4ec	Tabelle	Timerkonstanten für PAL-Baudraten
\$e500	IOBASE	Routine zum Kernal-Einsprung \$fff3
\$e505	SCREEN	Routine zum Kernal-Einsprung \$ffed
\$e50a	PLOT	Routine zum Kernal-Einsprung \$ffff0
\$e518	INTSCR	Bildschirm initialisieren
\$e544	CLEAR	Routine zum Steuerzeichen \$93
\$e566	HOME	Routine zum Steuerzeichen \$13
\$e56c	STUPT	Hilfsspeicher des Editors aktualisieren
\$e591	–	Unterprogramm der Tastatur-Eingabeschleife
\$e599	Füllbefehl	–
\$e59a	–	Routinenkombination INTVIC-HOME
\$e5a0	INTVIC	VIC-Register initialisieren
\$e5b4	NXTKEY	nächstes Zeichen aus Tastaturpuffer holen
\$e5ca	–	Tastatur-Eingabeschleife
\$e632	SCRGET	Zeichen aus Bildschirmspeicher in Akku holen
\$e684	CHGQUT	Anführungszeichenmodus umschalten

\$e691	CHRRAM	Zeichen in Bildschirmspeicher übernehmen
\$e6a8	–	Schlußbehandlung der Bildschirmausgaberoutinen
\$e6b6	UPDTL	LDTB1 initialisieren
\$e716	SCROUT	BSOUT-Behandlung für Gerät #3 (Bildschirm)
\$e87c	SETNWL	neue Zeile einrichten
\$e891	CR	Sprung an Anfang der nächsten Zeile
\$e8a1	MOVLFT	Cursorbewegung nach links (Hilfsroutine)
\$e8b3	MOVrgT	Cursorbewegung nach rechts (Hilfsroutine)
\$e8cb	COLCOD	Erkennung und Ausführung von Farbsteuerzeichen
\$e8da	Tabelle	Farbsteuerzeichen
\$e8ea	SCROLL	Aufwärts-Scrolling des gesamten Bildschirms
\$e9c8	SCRLIN	Aufwärts-Scrolling einer einzelnen Zeile
\$e9e0	COLADR	aktuelle Farb-RAM-Adresse berechnen
\$e9f0	LINADR	Adresse der aktuellen Zeile berechnen
\$e9ff	DELLIN	Bildschirmzeile löschen
\$ea12	Füllbefehl	–
\$ea13	SETCHC	Zeichen- und Farbcode in Speicher übertragen
\$ea24	COLPTR	korrespondierende Farb-RAM-Adresse berechnen
\$ea31	IRQ	IRQ-Routine des Kernal
\$ea87	SCNKEY	Routine zum Kernal-Einsprung \$ff9f
\$eadd	KEYLOG	Auswertung von Tastaturcode
\$eb48	–	Unterprogramm zur Prüfung von SHIFT/CBM/CTRL
\$eb79	Tabelle	Basisadressen der Tastaturtabellen
\$eb81	Tabelle	Tastaturtabelle #0
\$ebc2	Tabelle	Tastaturtabelle #1
\$ec03	Tabelle	Tastaturtabelle #2
\$ec44	–	Steuerzeichen zur Zeichensatzauswahl bearbeiten
\$ec78	Tabelle	Tastaturtabelle #3
\$ecb9	Tabelle	Initialisierungswerte der VIC-Register
\$ece7	Tabelle	LOAD<cr>RUN<cr>
\$ecf0	Tabelle	Low-Bytes der Basisadressen der Bildschirmzeilen
\$ed09	TALK	Routine zum Kernal-Einsprung \$ffb4
\$ed0c	LISTEN	Routine zum Kernal-Einsprung \$ffb1
\$edb9	SECOND	Routine zum Kernal-Einsprung \$ff93
\$edc7	TKSA	Routine zum Kernal-Einsprung \$ff96
\$eddd	CIOUT	Routine zum Kernal-Einsprung \$ffa8
\$edef	UNTALK	Routine zum Kernal-Einsprung \$ffb4
\$edfe	UNLSN	Routine zum Kernal-Einsprung \$ffa8

\$ee13	IECIN	Routine zum Kernal-Einsprung \$ffa5
\$ee85	CLCKHI	CLOCK auf HIGH setzen
\$ee8e	CLCKLO	CLOCK auf LOW setzen
\$ee97	DATAHI	DATA auf HIGH setzen
\$eea0	DATALO	DATA auf LOW setzen
\$eea9	DEBPIA	Datenport A von CIA 2 auslesen
\$eeb3	WAIT.1	1 Millisekunde warten
\$eebb	–	Ausgabe-Teilroutine des NMI bei RS232-Betrieb
\$ef06	RSTBGN	Übertragung des nächsten Bytes auf RS232
\$ef4a	CALCBT	Anzahl der Bits pro Datenwort für RS232 berechnen
\$ef59	RSRCVR	Auswertung eines über RS232 eingelesenen Bits
\$ef7e	RSRABL	RS232 für Empfang initialisieren
\$ef90	RSRTRT	Startbitprüfung (RS232)
\$ef97	–	Byte in RS232-Empfangspuffer übernehmen
\$efe1	CKORS	CKOUT für RS232
\$f014	BSORS	BSOUT für RS232
\$f04d	CKIRS	CHKIN für RS232
\$f086	GETRS	GETIN für RS232
\$f0a4	RSP232	auf Ende des RS232-Betriebs warten
\$f0bd	Tabelle	Systemmeldungen
\$f12b	–	Ausgabe einer Systemmeldung
\$f13e	GETIN	Routine zum Kernal-Einsprung \$ffe4
\$f157	BASIN	Routine zum Kernal-Einsprung \$ffcf
\$f199	JTGET	nächstes Byte aus Kassettenpuffer auslesen
\$f1ca	SOUT	Routine zum Kernal-Einsprung \$ffd2
\$f20e	HKIN	Routine zum Kernal-Einsprung \$ffc6
\$f250	KOUT	Routine zum Kernal-Einsprung \$ffc9
\$f291	CLOSE	Routine zum Kernal-Einsprung \$ffc3
\$f30f	LOOKUP	Offset eines Files in der Filetabelle ermitteln
\$f314	JLTLK	späterer Einstieg in LOOKUP (\$f30f)
\$f31f	GETLFS	Fileparameter anhand des File-Offsets ermitteln
\$f32f	CLALL	Routine zum Kernal-Einsprung \$ffe7
\$f34a	OPEN	Routine zum Kernal-Einsprung \$ffc0
\$f3d5	IECOPN	File auf IEC-Bus öffnen
\$f409	RSOPEN	File auf RS232 öffnen
\$f48a	ICIARS	CIA-Register nach RS232-Betrieb initialisieren
\$f49e	LOAD	Routine zum Kernal-Einsprung \$ffd5
\$f5af	SRCMSG	Ausgabe der SEARCHING-Meldung im Direktmodus
\$f5d2	LOADNG	Ausgabe von LOADING oder VERIFYING

\$f5dd	SAVE	Routine zum Kernal-Einsprung \$ffd8
\$f642	IECCLS	File am IEC-Bus schließen
\$f68f	SAVING	Ausgabe der SAVING-Meldung
\$f69b	UDTIM	Routine zum Kernal-Einsprung \$ffea
\$f6dd	RDTIM	Routine zum Kernal-Einsprung \$ffde
\$f6e4	SETTIM	Routine zum Kernal-Einsprung \$ffdb
\$f6ed	STOP	Routine zum Kernal-Einsprung \$ffe1
\$f6fb	IOERR1	Ausgabe der Meldung »I/O ERROR #1«
\$f6fe	IOERR2	Ausgabe der Meldung »I/O ERROR #2«
\$f701	IOERR3	Ausgabe der Meldung »I/O ERROR #3«
\$f704	IOERR4	Ausgabe der Meldung »I/O ERROR #4«
\$f707	IOERR5	Ausgabe der Meldung »I/O ERROR #5«
\$f70a	IOERR6	Ausgabe der Meldung »I/O ERROR #6«
\$f70d	IOERR7	Ausgabe der Meldung »I/O ERROR #7«
\$f710	IOERR8	Ausgabe der Meldung »I/O ERROR #8«
\$f713	IOERR9	Ausgabe der Meldung »I/O ERROR #9«
\$f72c	GETFHD	nächsten Header von Kassette holen
\$f76a	TAPEHE	WBLK-Routine für Programm-anfangsblock
\$f7d0	GETBFA	Anfangsadresse des Kassettenpuffers holen und testen
\$f7d7	BFSAE	Kassettenpuffer als Ein-/Ausgabebereich festlegen
\$f7ea	SRCTFL	bestimmtes File auf Kassette suchen
\$f80d	TBFUL	Test auf freien Platz im Kassettenpuffer
\$f817	WTPLAY	auf PLAY-Taste an Datensette warten
\$f82e	TSPLAY	PLAY-Taste an Datensette prüfen
\$f838	WTRCPL	auf <RECORD & PLAY> an Datensette warten
\$f841	RBLK	Datenblock von Kassette in Kassettenpuffer einlesen
\$f84a	TPREAD	Datenbereich von Kassette einlesen
\$f864	WBLK	Kassettenpuffer auf Kassette schreiben
\$f870	TPWRIT	Datenbereich auf Kassette schreiben
\$f875	TAPE	allgemeine Kassettenbehandlung
\$f8d0	TSSTOP	Computer-STOP-Taste während Kassettenbetrieb prüfen
\$f8e2	SETPIN	Kassette für Lesevorgang vorbereiten
\$f92c	–	IRQ-Routine »read« für Lesen von Kassette
\$fb8e	STACUR	Hilfszeiger \$ac/\$ad initialisieren
\$fb97	NEWCH	Register für seriellles Lesen/Schreiben initialisieren
\$fba6	–	Kassetten-Unterroutinen
\$fbcd	WRTN	IRQ-Routine für Schreiben von Daten auf Kassette
\$fc6a	WRTZ	IRQ-Routine für Schreiben der Synchronisation

\$fc93	STPTAP	Kassettenbetrieb beenden
\$fcb8	–	Beenden des Kassettenbetriebs mit IRQ-Abschluß
\$fcdb	BSIV	Interrupt-Routine gemäß Offset aktivieren
\$fcca	TAPMOF	Motor der Datasette ausschalten
\$fcd1	CMPSTE	Hilfszeiger \$ac/\$ad mit \$ae/\$af vergleichen
\$fcdb	INCSAL	Hilfszeiger \$ac/\$ad erhöhen
\$fce2	RESET	Reset-Routine des Kernal
\$fd02	CHKCBM	Prüfung auf Modulkennung »CBM80«
\$fd10	Tabelle	Modulkennung »CBM80« als Vergleichspunkt
\$fd15	RESTOR	Routine zum Kernal-Einsprung \$ff8a
\$fd1a	VECTOR	Routine zum Kernal-Einsprung \$ff8d
\$fd30	Tabelle	Initialisierungswerte für die Vektoren \$0314–\$0333
\$fd50	RAMTAS	Routine zum Kernal-Einsprung \$ff87
\$fd9b	Tabelle	Adressen der IRQ-Routinen
\$fda3	IOINIT	Routine zum Kernal-Einsprung \$ff84
\$fdf9	SETNAM	Routine zum Kernal-Einsprung \$ffb7
\$fe00	SETLFS	Routine zum Kernal-Einsprung \$ffb7
\$fe07	READST	Routine zum Kernal-Einsprung \$ffb7

\$fe18	SETMSG	Routine zum Kernal-Einsprung \$ff90
\$fe1a	–	READST-Behandlung für alle Geräte außer RS232
\$fe1c	ERSTAT	Fehlerbits aus Akku in Statusbyte ST einblenden
\$fe21	SETTMO	Routine zum Kernal-Einsprung \$ffa2
\$fe25	MEMTOP	Routine zum Kernal-Einsprung \$ff99
\$fe34	MEMBOT	Routine zum Kernal-Einsprung \$ff9c
\$fe43	NMI	NMI-Routine des Kernal
\$fec2	Tabelle	Timerkonstanten für NTSC-Baudraten
\$ff41	Füllbefehle	–
\$ff43	TPIRQ	IRQ-Einsprung für Datasettenbehandlung
\$ff48	–	allgemeine IRQ/BREAK-Routine
\$ff5b	CINT	Routine zum Kernal-Einsprung \$ff81
\$ff6e	–	Timer-Initialisierung für IRQ
\$ff80	Füllbyte	–
\$ff81	Sprungtabelle	Kernal-Einsprünge
\$fff6	Füllbytes	–
\$fffa	ROM-Vektor	Adresse der NMI-Routine
\$fffc	ROM-Vektor	Adresse der RESET-Routine
\$ffff	ROM-Vektor	Adresse der IRQ/BREAK-Routine

Kapitel 6

Memory Map

SPEICHERBELEGUNG DES COMODORE 64

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
D6510	0000	0	6510 Datenrichtungsregister
P6510	0001	1	6510 8-Bit-Ein-/Ausgabe- register
	0002	2	Nicht benutzt
ADRAY1	0003-0004	3-4	Sprungvektor: Umwandlung Gleitpunktzahl/Ganze Zahl
ADRAY 2	0005-0006	5-6	Sprungvektor: Umwandlung Ganze Zahl/Gleitpunktzahl
CHARAC	0007	7	Suchzeichen
ENDCHR	0008	8	Flag: Suchen nach einem Anführungszeichen am Ende eines Strings
TRMPOS	0009	9	Bildschirmspalte ab letztem TAB
VERCK	000A	10	0 = LOAD, 1 = VERIFY
COUNT	000B	11	Eingabepufferzeiger, Anzahl der Elemente
DIMFLG	000C	12	Flag: Standard-Felddimensio- nierung
VALTYP	000D	13	Datentyp: \$FF = String, \$00 = Numerisch
INTFLG	000E	14	Datentyp: \$80 = Ganze Zahl, \$00 = Gleitpunktzahl
GARBFL	000F	15	Flag: DATAs lesen/LIST auf- listen "garbage collection"
SUBFLG	0010	16	Flag: Benutzerfunktionsaufruf
INPFLG	0011	17	Flag: \$00 = INPUT, \$40 = GET, \$98 = READ
TANSGN	0012	18	Flag: Vorzeichen des TAN/Flag für Gleichheit bei Vergleich
	0013	19	Flag: INPUT-Kommentar
LINNUM	0014-0015	20-21	Ganzzahliger Wert
TEMPPT	0016	22	Zeiger: Temporärer Stringstapel
LASTPT	0017-0018	23-24	Letzte Stringadresse
TEMPST	0019-0021	25-33	Stapel für temporäre Strings
INDEX	0022-0025	34-37	Bereich für Hilfszeiger
RESHO	0026-002A	38-42	Gleitpunktergebnis der Multiplikation

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
TXTTAB	002B-002C	43-44	Zeiger: Anfang BASIC-Text
VARTAB	002D-002E	45-46	Zeiger: Anfang BASIC-Variablen
ARYTAB	002F-0030	47-48	Zeiger: Anfang BASIC-Felder
STREND	0031-0032	49-50	Zeiger: Ende BASIC-Felder (+ 1)
FRETOP	0033-0034	51-52	Zeiger: Anfang der String- Speicherung
FRESPC	0035-0036	53-54	Hilfszeiger für Strings
MEMSIZ	0037-0038	55-56	Zeiger: Oberste BASIC-Adresse
CURLIN	0039-003A	57-58	Derzeitige BASIC-Zeilen- nummer
OLDLIN	003B-003C	59-60	Vorherige BASIC-Zeilenummer
OLDTXT	003D-003E	61-62	Zeiger: BASIC-Anweisung für CONT
DATLIN	003F-0040	63-64	Derzeitige DATA-Zeilenummer
DATPTR	0041-0042	65-66	Zeiger: Derzeitige DATA- Adresse
INPPTR	0043-0044	67-68	Vektor: INPUT-Routine
VARNAM	0045-0046	69-70	Derzeitiger BASIC-Variablen- name
VARPNT	0047-0048	71-72	Adresse der aktuellen Variablen
FORPNT	0049-004A	73-74	Variablenzeiger für FOR/NEXT
	004B-0060	75-96	Zwischenspeicher für BASIC- Zeiger/Daten
FACEXP	0061	97	Gleitpunktakkumulator # 1: Exponent
FACHO	0062-0065	98-101	Gleitpunktakkumulator # 1: Mantisse
FACSGN	0066	102	Gleitpunktakkumulator # 1: Vorzeichen
SGNFLG	0067	103	Zeiger: Polynomauswertung
BITS	0068	104	Gleitpunktakkumulator # 1: Überlauf
ARGEXP	0069	105	Gleitpunktakkumulator # 2: Exponent
ARGHO	006A-006D	106-109	Gleitpunktakkumulator # 2: Mantisse

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
ARGSGN	006E	110	Gleitpunktakkumulator #2: Vorzeichen
ARISGN	006F	111	Ergebnis des Vorzeichen- vergleichs: Akku #1 Akku #2
FACOV	0070	112	Gleitpunktakkumulator #1: Niederwertige Stelle (Rundung)
FBUFPT	0071–0072	113–114	Zeiger: Kassettenpuffer
CHRGET	0073–008A	115–138	Unterroutine: Nächstes Byte vom BASIC-Text lesen
CHRGOT	0079	121	Erneutes Lesen des gleichen Text-Bytes
TXTPTR	007A–007B	122–123	Zeiger: Derzeitiges Byte des BASIC-Textes
RNDX	008B–008F	139–143	Eingangswert der RND- Funktion
STATUS	0090	144	KERNAL-Ein-/Ausgabestatus- wort: ST
STKEY	0091	145	Flag: STOP-Taste/RVS-Taste
SVXT	0092	146	Zeit-Konstante für Kassette
VERCK	0093	147	Flag: 0 = LOAD, 1 = VERIFY
C3PO	0094	148	Flag: serieller Bus – Zeichen im Puffer
BSOUR	0095	149	Zeichen im Puffer für seriellen Bus
SYNO	0096	150	Kassetten SYNC.-Nr. (EOT von Kassette empfangen)
LDTND	0097	151	Temporäre Datenadresse
	0098	152	Anzahl der offenen Dateien/ Dateitabellen-Index
DFLTN	0099	153	Standard-Eingabegerät (0)
DFLTO	009A	154	Standard-Ausgabegerät (CMD) (3)
PRTY	009B	155	Paritätsbyte vom Band
DPSW	009C	156	Flag: Byte empfangen
MSGFLG	009D	157	Flag: \$80 = Direktmodus, \$00 = Programm
PTR1	009E	158	Bandfehler/Zeichenpuffer
PTR2	009F	159	Bandfehler korrigiert
TIME	00A0–00A2	160–162	Echtzeituhr (ca.) 1/60 s
CNTDN	00A3–00A4	163–164	Temporärer Datenbereich
	00A5	165	Kassetten Sync.: Abwärts- zählung beim Schreiben
BUFPNT	00A6	166	Zeiger: Kassettenpuffer
INBIT	00A7	167	RS-232-Eingabebits/Kassette temp.
BITCI	00A8	168	RS-232-Eingabebit-Zählung/ Kassette temp.
RINONE	00A9	169	RS-232 Flag: Startbit- überprüfung
RIDATA	00AA	170	RS-232-Eingabebyte-Puffer/ Kassette temp.

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
RIPRTY	00AB	171	RS-232-Eingabeparität/ Kassette, Zählung
SAL	00AC–00AD	172–173	Zeiger: Kassettenpuffer/Bild- schirm scrollen
EAL	00AE–00AF	174–175	Kassettenende/Programmende
CMPO	00B0–00B1	176–177	Kassetten-Zeit-Konstante
TAPE1	00B2–00B3	178–179	Zeiger: Anfang des Kassetten- puffers
BITTS	00B4	180	RS-232 nächstes Bit zum Scrollen/Kassette temp.
NXTBIT	00B5	181	RS-232 Nächstes zu über- tragendes Bit/Kassetten- kennzeichen EOT
RODATA	00B6	182	RS-232 Bytepuffer
FNLEN	00B7	183	Länge der aktuellen Datei- namen
LA	00B8	184	Logische Dateinummer
SA	00B9	185	Aktuelle Sekundäradresse
FA	00BA	186	Aktuelle Gerätenummer
FNADR	00BB–00BC	187–188	Zeiger: Aktueller Dateiname
ROPRTY	00BD	189	RS-232 Parität/Kassette, temp.
FSBLK	00BE	190	Anzahl der zum Lesen/ Schreiben verbleibenden Blocks
MYCH	00BF	191	Serieller Puffer
CAS1	00C0	192	Kassettenmotor-Flag
STAL	00C1–00C2	193–194	Ein-/Ausgabestartadresse
MEMUSS	00C3–00C4	195–196	Zeiger auf Vektoradressen des KERNAL
LSTX	00C5	197	Derzeitig gedrückte Taste: CHRS(n); 0 = Keine Taste
NDX	00C6	198	Anzahl der Zeichen im Tastatur- puffer (Warteschlange)
RVS	00C7	199	Flag: Ausdruck negativer Zeichen – 1 = ja, 0 = nein
INDX	00C8	200	Zeiger: Ende der logischen Zeile für Eingabe
LXSP	00C9–00CA	201–202	Cursor X/Y-Position für Eingabe
SFDX	00CB	203	Flag: Gedrückte Taste
BLNSW	00CC	204	Cursor an/aus: (0 = blinkender Cursor)
BLNCT	00CD	205	Zähler für blinkenden Cursor
GDBLN	00CE	206	Zeichen für Cursorposition
BLNON	00CF	207	Flag: Cursor in Blinkphase
CRSW	00D0	208	Flag: INPUT oder GET über Tastatur
PNT	00D1–00D2	209–210	Zeiger: Derzeitige Bildschirm- zeile
PNTR	00D3	211	Cursorspalte in derzeitiger Zeile
QTSW	00D4	212	Flag: Editor im Anführungs- zeichen-Modus, \$00 = NEIN

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
LNMX	00D5	213	Physische Bildschirmzeilenlänge
TBLX	00D6	214	Zeile, in der sich Cursor befindet
INSRT	00D7	215	Temporärer Datenbereich
	00D8	216	Flag: Einfügemodus, >0 = Anzahl der Einfügungen
LDTB1	00D9–00F2	217–242	Bildschirmzeilen-Verknüpfungstabelle/Editor temp.
USER	00F3–00F4	243–244	Zeiger: Derzeitiger Farb-RAM des Bildschirms
KEYTAB	00F5–00F6	245–246	Vektor: Tastatur Decodiertabelle
RIBUF	00F7–00F8	247–248	RS-232-Eingabepuffer-Zeiger
ROBUF	00F9–00FA	249–250	RS-232-Ausgabepuffer-Zeiger
FREKZP	00FB–00FE	251–254	Freier Platz in der Zero-Page für Betriebssystem
BASZPT	00FF	255	Temp. BASIC-Datenbereich
	0100–01FF	256–511	Stapelspeicher des Mikroprozessors
	0100–010A	256–266	Arbeitsbereich Umwandlung Gleitpunkt in ASCII
BAD	0100–013E	256–318	Bandfehler
BUF	0200–0258	512–600	System-Eingabepuffer
LAT	0259–0262	601–610	KERNAL-Tabelle: Aktive logische Dateinummern
FAT	0263–026C	611–620	KERNAL-Tabelle: Geräte-Nr. für jede Datei
SAT	026D–0276	621–630	KERNAL-Tabelle: Sekundäradresse jeder Datei
KEYD	0277–0280	631–640	Tastaturpuffer (Warteschlange) (FIFO)
MEMSTR	0281–0282	641–642	Zeiger: Startadresse des RAM für Betriebssystem
MEMSIZ	0283–0284	643–644	Zeiger: Ende des RAM für Betriebssystem
TIMOUT	0285	645	Flag: Zeitüberschreitung auf IEEE-Bus
COLOR	0286	646	Derzeitiger Zeichenfarbcode
GDCOL	0287	647	Hintergrundfarbe unter Cursor
HIBASE	0288	648	Bildschirmspeicher-Anfang (Page)
XMAX	0289	649	Größe des Tastaturpuffers
RPTFLG	028A	650	Flag: Tastenwiederholung, \$80 = Wiederholen
KOUNT	028B	651	Zählgeschwindigkeit für Wiederholen
DELAY	028C	652	Zähler für Wiederholungsverzögerung
SHFLAG	028D	653	Flag: Taste SHIFT/Taste CTRL/ C = Taste
LSTSHF	028E	654	Letztes SHIFT-Muster der Tastatur

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
KEYLOG	028F–0290	655–656	Zeiger auf Tastatur-Decodiertabelle
MODE	0291	657	Flag: \$80 = SHIFT unwirksam, \$00 = wirksam
AUTODN	0292	658	Flag: Automatisches Scrollen (abwärts), 0 = EIN; #0 = AUS
M51CTR	0293	659	RS-232: 6551 Kontrollregister
M51CDR	0294	660	RS-232: 6551 Befehlsregister
M51AJB	0295–0296	661–662	RS-232 nicht Standard (Bit-Zeit)
RSSTAT	0297	663	RS-232: 6551 Statusregister
BITNUM	0298	664	RS-232 Anzahl der noch zu übertragenden Bits
BAUDOF	0299–029A	665–666	RS-232 Baud-Rate: Full Bit Time (µs)
RIDBE	029B	667	RS-232 Eingabepuffer-Ende
RIDBS	029C	668	RS-232 Eingabepuffer-Anfang (Page)
RODBS	029D	669	RS-232 Ausgabepuffer-Anfang (Page)
RODBE	029E	670	RS-232 Ausgabepuffer-Ende
IRQTMP	029F–02A0	671–672	Enthält IRQ-Vektor während Kassetten-Ein-/Ausgabe
ENABL	02A1	673	RS-232
	02A2	674	Temp. Speicherung für Lesen von Kassette
	02A3	675	Temp Storage For Cassette Read
	02A4	676	Temp D1IRQ Indicator For
	02A5	677	Cassette Read
	02A6	678	Temp For Line Index
IERROR	02A7–02FF	679–767	PAL/NTSC Flag, 0 = NTSC, 1 = PAL
	0300–0301	768–769	Vektor: BASIC-Fehlermeldung anzeigen
IMAIN	0302–0303	770–771	Vektor: BASIC-Warmstart
ICRNCH	0304–0305	772–773	Vektor: BASIC-Text in Token umwandeln
IQPLOP	0306–0307	774–775	Vektor: BASIC-Text listen
IGONE	0308–0309	776–777	Vektor: BASIC-Befehl ausführen
IEVAL	030A–030B	778–779	Vektor: BASIC-Tokens-Auswertung
SAREG	030C	780	Speicher für 6502 .A-Register
SXREG	030D	781	Speicher für 6502 .X-Register
SYREG	030E	782	Speicher für 6502 .Y-Register
SPREG	030F	783	Speicher für SP6502 SP-Register
USRPOK	0310	784	USR-Sprung
USRADD	0311–0312	785–786	USR-Adresse niederwertiges Byte/höherwertiges Byte
	0313	787	Nicht benutzt

MARKE (LABEL)	ADRESSE (HEX)	ADRESSE (DEZ)	BESCHREIBUNG
CINV	0314–0315	788–789	Vektor: Hardware Interrupt (IRQ) (EA31)
CBINV	0316–0317	790–791	Vektor: BRK-Interrupt (FE66)
NMINV	0318–0319	792–793	Vektor: Nicht maskierbarer Interrupt (NMI) (FE47)
IOPEN	031A–031B	794–795	KERNAL OPEN-Routine-Vektor
ICLOSE	031C–031D	796–797	KERNAL CLOSE-Routine-Vektor
ICKIN	031E–031F	798–799	KERNAL CHKIN-Routine-Vektor
ICKOUT	0320–0321	800–801	KERNAL CHKOUT-Routine-Vektor
ICLRCH	0322–0323	802–803	KERNAL CLRCHN-Routine-Vektor
IBASIN	0324–0325	804–805	KERNAL CHRIN-Routine-Vektor
IBSOUT	0326–0327	806–807	KERNAL CHROUT-Routine-Vektor
ISTOP	0328–0329	808–809	KERNAL STOP-Routine-Vektor
IGETIN	032A–032B	810–811	KERNAL GETIN-Routine-Vektor
ICLALL	032C–032D	812–813	KERNAL CLALL-Routine-Vektor
USRCMD	032E–032F	814–815	Benutzer-IRQ
ILOAD	0330–0331	816–817	KERNAL LOAD-Routine-Vektor
ISAVE	0332–0333	818–819	KERNAL SAVE-Routine-Vektor
TBUFFR	0334–033B	820–827	Nicht benutzt
	033C–03FB	828–1019	Kassettenpuffer
VICSCN	03FC–03FF	1020–1023	Nicht benutzt
	0400–07FF	1024–2047	1024 Byte Bildschirmspeicher-Bereich
	0400–07E7	1024–2023	Video-Matrix: 25 Zeilen x 40 Zeichen
	07F8–07FF	2040–2047	Sprite-Datenzeiger
	0800–9FFF	2048–40959	Normaler BASIC-Programmbereich
	8000–9FFF	32768–40959	VSP-ROM-8192 Bytes (Optional)
	A000–BFFF	40960–49151	BASIC-ROM-8192 Bytes (oder 8K-RAM)
	C000–CFFF	49152–53247	RAM-4096 Bytes
	D000–DFFF	53248–57343	Ein-/Ausgabegerät und Farb-RAM oder Zeichengenerator-ROM oder RAM-4096 Bytes
	E000–FFFF	57344–65535	KERNAL ROM-8192 Bytes (oder 8K-RAM)

EIN-/AUSGABEANORDNUNG
BEIM COMMODORE 64

HEXA- DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG
0000	0	7–0	MOS 6510 Datenrichtungsregister (xx101111) Bit = 1: Ausgabe, Bit = 0: Eingabe X = Spielt keine Rolle
0001	1		MOS 6510 Mikroprozessor Ein-Chip
D000–D02E	53248–54271	0	Ein-/Ausgabeport /LORAM-Signal (0 = BASIC-ROM ausschalten)
		1	/HIRAM-Signal (0 = KERNAL-ROM ausschalten)
		2	/CHARAN-Signal (0 = Zeichen-ROM ausschalten)
		3	Kassettendaten-Ausgabeleitung
		4	Kassettschalter 1 = Schalter geschlossen
		5	Kassetten-Motorsteuerung 0 = EIN, 1 = AUS
		6–7	Nicht belegt
			MOS 6566 VIDEO- INTERFACESTEUERUNG (VIC)
			Sprite 0, Position X
			Sprite 0, Position Y
D000	53248		Sprite 1, Position X
D001	53249		Sprite 1, Position Y
D002	53250		Sprite 2, Position X
D003	53251		Sprite 2, Position Y
D004	53252		Sprite 3, Position X
D005	53253		Sprite 3, Position Y
D006	53254		Sprite 4, Position X
D007	53255		Sprite 4, Position Y
D008	53256		Sprite 5, Position X
D009	53257		Sprite 5, Position Y
D00A	53258		Sprite 6, Position X
D00B	53259		Sprite 6, Position Y
D00C	53260		Sprite 7, Position X
D00D	53261		Sprite 7, Position Y
D00E	53262		Sprites 0–7, Position X (msb der X-Koordinate)
D00F	53263		VIC-Steuerregister
D010	53264		Raster-Vergleich: (Bit 8) Siehe 53266
D011	53265	7	Erweiterter Farbtext-Modus: 1 = Einschalten
		6	Bit-Map-Modus: 1 = Einschalten
		5	

HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG
D012	53266	4	Bildschirm löschen: 0 = Löschen
		3	Wahl von 24/25 Reihen Text- anzeige: 1 = 25 Reihen
		2-0	Rollen zur Y-Punktposition (0-7)
			Leseraster/Schreibraster Wert für Vergleich IRQ
			Lichtgriffel, Position X
D013	53267		Lichtgriffel, Position Y
D014	53268		Sprite-Anzeige: 1 = Einschalten
D015	53269		VIC-Steuerregister
D016	53270		Nicht benutzt
D017	53271	7-6	DIESES BIT STETS AUF 0 SETZEN!
		5	Mehrfarbenmodus: 1 = Einschalten (Text oder Bit-Mappe)
		4	Wahl von 38/40 Spalten Text- anzeige: 1 = 40 Zeichen
		3	Rollen zu Position X
		2-0	Sprites 0-7 vergrößern 2 x vertikal (Y)
D018	53272		VIC-Speicher-Steuerregister
D019	53273	7-4	Video-Matrix-Basisadresse
		3-1	Zeichengenerator-Basisadresse
			VIC-Interrupt-Flag (Bit = 1: Einschalten des IRQ)
		7	Beliebige VIC-IRQ-Bedingung setzen
		3	IRQ-Flag wird durch Lichtgriffel getriggert
D01A	53274	2	IRQ-Flag für Sprite-Kollision
		1	IRQ-Flag für Sprite-/Hinter- grundkollision
		0	IRQ-Flag für Rastervergleich
			IRQ-Maskenregister: 1 = Interrupt einschalten
			Sprite-/Hintergrund-Anzeige- priorität: 1 = Sprite
D01B	53275		Sprites 0-7 Mehrfarbenmodus gewählt: 1 = Mehrfarben- modus
D01C	53276		Sprites 0-7, vergrößern 2 x horizontal (X)
D01D	53277		Sprite-Kollisionserkennung
D01E	53278		Sprite-/Hintergrundkollisions- Erkennung
D01F	53279		Rahmenfarbe
D020	53280		Hintergrundfarbe 0
D021	53281		Hintergrundfarbe 1
D022	53282		Hintergrundfarbe 2
D023	53283		

HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG
D024	53284		Hintergrundfarbe 3
D025	53285		Sprite-Mehrfarbenregister 0
D026	53286		Sprite-Mehrfarbenregister 1
D027	53287		Farbe von Sprite 0
D028	53288		Farbe von Sprite 1
D029	53289		Farbe von Sprite 2
D02A	53290		Farbe von Sprite 3
D02B	53291		Farbe von Sprite 4
D02C	53292		Farbe von Sprite 5
D02D	53293		Farbe von Sprite 6
D02E	53294		Farbe von Sprite 7
D400-D7FF	54272-55295		MOS 6581 SOUND- INTERFACE-DEVICE (SID)
D400	54272		Stimme 1: Frequenzsteuerung - Unteres Byte
D401	54273		Stimme 1: Frequenzsteuerung - Oberes Byte
D402	54274		Stimme 1: Pulswellen-Breite - Unteres Byte
D403	54275	7-4 3-0	Nicht benutzt Stimme 1: Pulswellen-Breite - Oberes Nybble
D404	54276	7 6 5 4 3 2 1 0	Stimme 1: Steuerregister Geräuschwellenform wählen, 1 = Ein Pulswellenform wählen, 1 = Ein Sägezahnwellenform wählen, 1 = Ein Dreieckswellenform wählen, 1 = Ein Testbit: 1 = Oszillator 1 abschalten Oszillator 1 mit Oszillator- ausgabe 3 ringmodulieren, 1 = Ein Oszillator 1 mit Oszillator 3 synchronisieren, 1 = Ein GATE-Bit: 1 = Beginn von ATTACK/DECAY/SUSTAIN, 0 = Start des RELEASE- Abschnitts
D405	54277	7-4 3-0	Hüllkurvegeber 1: Steuerung des ATTACK-/DECAY-Zyklus Wahl der ATTACK-Zyklusdauer: 0-15 Wahl der DECAY-Zyklusdauer: 0-15
D406	54278	7-4	Hüllkurvegeber 1: Steuerung des SUSTAIN-/RELEASE- Zyklus Wahl des SUSTAIN-Pegels: 0-15

HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG	HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG
D407	54279	3–0	Wahl der RELEASE-Dauer: 0–15			7	Wahl der Geräuschwellenform, 1 = Ein
D408	54280		Stimme 2: Frequenzsteuerung – Unteres Byte			6	Wahl der Impulswellenform, 1 = Ein
D409	54281		Stimme 2: Frequenzsteuerung – Oberes Byte			5	Wahl der Sägezahnwellenform, 1 = Ein
D40A	54282	7–4	Stimme 2: Pulswellen-Breite – Unteres Byte			4	Wahl der Dreieckswellenform, 1 = Ein
D40B	54283	3–0	Nicht benutzt			3	Testbit: 1 = Oszillator 3 ausschalten
			Stimme 2: Pulswellen-Breite – Oberes Nybble			2	Oszillator 3 mit Oszillator- ausgabe2 ringmodulieren, 1 = Ein
		7	Stimme 2: Steuerregister			1	Oszillator 3 mit Oszillator- frequenz 2 synchronisieren, 1 = Ein
		6	Wahl der Geräuschwellenform, 1 = Ein			0	GATE-Bit: 1 = Beginn von ATTACK/DECAY/SUSTAIN, 0 = Start des RELEASE- Abschnitts
		5	Wahl der Pulswellenform, 1 = Ein	D413	54291		Hüllkurvengeber 3: Steuerung des ATTACK-/DECAY-Zyklus
		4	Wahl der Sägezahnwellenform, 1 = Ein			7–4	Wahl der ATTACK-Dauer: 0–15
		3	Wahl der Dreieckswellenform, 1 = Ein	D414	54292	3–0	Wahl der DECAY-Dauer: 0–15
		2	Testbit: 1 = Oszillator 2 ausschalten				Hüllkurvengeber 3: Steuerung des SUSTAIN-/RELEASE- Zyklus
		1	Oszillator 2 mit Oszillator- ausgabe 1 ringmodulieren, 1 = Ein			7–4	Wahl des SUSTAIN-Pegels: 0–15
		0	Oszillator 2 mit Oszillator- frequenz 1 synchronisieren, 1 = Ein			3–0	Wahl der RELEASE-Dauer: 0–15
D40C	54284		GATE-Bit: 1 = Beginn von ATTACK/DECAY/SUSTAIN, 0 = Start des RELEASE- Abschnitts	D415	54293		Filtergrenzfrequenz: Unteres Nybble (Bits 2–0)
		7–4	Hüllkurvengeber 2: Steuerung des ATTACK-/DECAY-Zyklus	D416	54294		Filtergrenzfrequenz: Oberes Byte
D40D	54285	3–0	Wahl der ATTACK-Dauer: 0–15	D417	54295		Filterresonanz-Steuerung/ Stimmeneingabe-Steuerung
			Wahl der DECAY-Dauer: 0–15			7–4	Wahl der Filterresonanz: 0–15
		7–4	Hüllkurvengeber 2: Steuerung SUSTAIN-/RELEASE-Zyklus			3	Externe Filtereingabe: 1 = Ja, 0 = Nein
		3–0	Wahl des SUSTAIN-Pegels: 0–15			2	Ausgabe von Stimme 3 filtern: 1 = Ja, 0 = Nein
D40E	54286		Wahl der RELEASE-Dauer: 0–15			1	Ausgabe von Stimme 2 filtern: 1 = Ja, 0 = Nein
D40F	54287		Stimme 3: Frequenzsteuerung – Unteres Byte			0	Ausgabe von Stimme 1 filtern: 1 = Ja, 0 = Nein
D410	54288		Stimme 3: Frequenzsteuerung – Oberes Byte	D418	54296		Filtermodus und Lautstärke wählen
D411	54289	7–4	Stimme 3: Pulswellen-Breite – Unteres Byte			7	Ausgabe von Stimme 3 abschalten: 1 = AUS, 0 = EIN
D412	54290	3–0	Nicht benutzt				
			Stimme 3: Pulswellen-Breite – Oberes Nybble				
			Stimme 3: Steuerregister				

HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG	HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG		
D419	54297	6	Hochpaßfiltermodus wählen: 1 = Ein	DC0B	56331	7	Tageszeituhr: Stunden + Flag AM/PM (Bit 7)		
		5	Wahl des Bandfiltermodus: 1 = Ein	DC0C	56332		Serieller Bus Ein-/Ausgabedatenpuffer		
		4	Wahl des Tiefpaßfiltermodus: 1 = Ein	DC0D	56333		CIA-Interrupt-Steuerregister IRQ-Flag (1 = Auftreten von IRQ)/Löschflag setzen		
		3–0	Wahl der Lautstärke: 0–15	DC0E	56334		4	Flag 1 IRQ (Lesen von Kassette/serieller Bus SRQ-Eingabe)	
D41A	54298	Analog-/Digitalwandler: Drehregler 2 (0–255)	3			Serieller Bus (Interrupt)			
D41B	54299	Oszillator 3, Zufallszahlen- Generator	2			Tageszeituhr-Interrupt			
D41C	54300	Ausgabe von Hüllkurvengeber 3	1			Timer B-Interrupt			
D500–D7FF	54528–55295	SID-Images	0			Timer A-Interrupt			
D800–DBFF	55296–56319	Farb-RAM (Nybbles)	7			CIA-Steuerregister A			
DC00–DCFF	56320–56575	MOS 6526 Komplexes Interfaceadapter (CIA) # 1	6			Tageszeituhr-Frequenz: 1 = 50 Hz, 0 = 60 Hz			
DC00	56320	Datenport A (Tastatur, Steuer- knüppel, Drehregler, Licht- griffel)	5			Serieller Bus Ein-/Ausgabe- modus: 1 = Ausgabe, 0 = Eingabe			
		7–0	Nummer der Tastaturspalte für Tastatur-Abfrage			4	Timer A: 1 = CNT-Signale, 0 = System-Uhr 02		
		7–6	Drehregler Port A/B (01 = Port A, 10 = Port B)			3	Force Load Timer A: 1 = Ja		
		4	Steuerknüppel A Feuerknopf: 1 = Feuer			2	Modus von Timer A: 1 = one- shot, 0 = kontinuierlich		
DC01	56321	3–2	Steuerknüppel A Feuerknopf: 1 = Feuer			2	Ausgabemodus von Timer A zu PB6: 1 = Toggle, 0 = Impuls		
		3–0	Drehregler-Feuerknöpfe			1	0 = Impuls		
		3–0	Steuerknüppel-Richtung (0–15)			DC0F	56335	1	Ausgabe von Timer A an PB6: 1 = Ja, 0 = Nein
		Daten-Port B (Tastatur, Steuer- knüppel, Drehregler): Spielport 1	0					Start/Stop von Timer A: 1 = Start, 0 = Stop	
7–0	Nummer der Tastatur-Reihe für Tastaturabfrage	7	CIA-Steuerregister B						
7	Timer B: Impulsausgabe	Alarm/ TOD-Uhr: 1 = Alarm, 0 = Takt							
DC02	56322	6	Timer A: Impulsausgabe	6–5	Wahl des Modus von Timer B: 00 = Taktimpuls von System 02 zählen 01 = Positive CNT-Über- gänge zählen 10 = Underflow-Impulse von Timer A zählen 11 = Underflows von Timer A zählen, wenn CNT positiv				
		4	Steuerknüppel Feuerknopf 1: 1 = Feuer			4–0	Entspricht CIA-Steuer- register A – für Timer B		
		3–2	Drehregler-Feuerknopf			DD00–DDFF	56576–56831	MOS 6526 Komplexes Inter- faceadapter (CIA) # 2	
		3–0	Steuerknüppel-Richtung						DD00
Datenrichtungsregister – Port A (56320)	Timer A: Unteres Byte								
DC03	56323	Datenrichtungsregister – Port B (56321)	Timer A: Oberes Byte						
DC04	56324	Timer B: Unteres Byte							
DC05	56325	Timer B: Oberes Byte							
DC06	56326	Tageszeituhr: 1/10 s							
DC07	56327	Tageszeituhr: Sekunden							
DC08	56328	Tageszeituhr: Minuten							
DC09	56329								
DC0A	56330								

HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG	HEXA-DEZIMAL	DEZIMAL	BITS	BESCHREIBUNG	
DD01	56577	7	Serieller Bus-Dateneingabe	DD0F	56591	4	Force Load Timer A: A = Ja	
		6	Serieller Bus-Impulseingabe			3	Modus von Timer A: 1 = one-shot, 0 = kontinuierlich	
		5	Serieller Bus-Datenausgabe			2	Ausgabemodus von Timer A zu PB6: 1 = Toggle, 0 = Impuls	
		4	Serieller Bus-Impulsausgabe				1	Ausgabe von Timer A an PB6: 1 = Ja, 0 = Nein
		3	Serieller Bus-ATN-Signal- ausgabe			0	Start/Stop von Timer A: 1 = Start, 0 = Stop	
		2	RS-232-Datenausgabe (User-Port)				7	CIA-Steuerregister B Alarm/TOD-Clock: 1 = Alarm, 0 = Clock
		1–0	VIC-Chip Bank-select (Standard = 11)			6–5	Wahl von Timermodus B: 00 = Impulse von System 02 zählen 01 = Positive CNT-Über- gänge zählen 10 = Underflowimpulse von Timer A zählen 11 = Underflows von Timer A zählen, wenn CNT positiv	
		Datenport B (User-Port, RS-232)	4–0				Entspricht CIA-Steuer- register A – für Timer B	
		7				RS-232 Datensatz bereit	DE00–DEFF	56832–57087
		6	RS-232 Clear to send					
5	User							
4	RS-232 Carrier Detect							
3	RS-232 Ring Indicator							
2	RS-232 Daten-Terminal							
1	RS-232 Request to send							
0	RS-232 Received							
DD02	56578	Datenrichtungs-Register – Port A	DF00–DFFF	57088–57343	Reserviert für künftige Ein-/Ausgabeerweiterungen			
DD03	56579	Datenrichtungs-Register – Port B						
DD04	56580	Timer A: Unteres Byte						
DD05	56581	Timer A: Oberes Byte						
DD06	56582	Timer B: Unteres Byte						
DD07	56583	Timer B: Oberes Byte						
DD08	56584	Tageszeituhr: 1/10 s						
DD09	56585	Tageszeituhr: Sekunden						
DD0A	56586	Tageszeituhr: Minuten						
DD0B	56587	Tageszeituhr: Stunden + Flag AM/PM (Bit 7)						
DD0C	56588	Serieller Bus Ein-/Ausgabe- datenpuffer						
DD0D	56589		CIA-Interruptsteuerregister					
		7	NMI-Flag (1 = Auftreten eines NMI)/ Löschflag setzen					
		4	Flag 1 NMI (RS-232 Received Data Input)					
		3	Interrupt-Serieller Bus					
		1	Timer B-Interrupt					
		0	Timer A-Interrupt					
			CIA-Steuerregister A					
		7	TOD-F: 1 = 50 Hz, 0 = 60 Hz					
		6	Serieller Bus Ein-/Ausgabe- modus: 1 = Ausgabe, 0 = Eingabe					
		5	Timer A: 1 = CNT-Signale, 0 = System-Uhr 02					

Kapitel 7

Ausblick: GEOS und C64-verwandte Betriebssysteme

In diesem Buch haben Sie bis zu dieser Stelle wohl alles erfahren, was über das Betriebssystem des C64 zu sagen ist. Deshalb soll zur Abrundung des Werkes noch einmal der Blickwinkel auf andere Systeme erweitert werden.

Sie lernen sowohl GEOS, das alternative Betriebssystem für den C64, als auch die Betriebssysteme der anderen 8-Bit-Heimcomputer von Commodore in einer Vorstellung kennen.

7.1 GEOS – Graphical Environment Operating System

Ich gehe davon aus, daß Sie GEOS schon einmal aus Anwendersicht erlebt haben, einen Testbericht kennen oder sogar schon über GEOS verfügen. Dann haben Sie sicherlich darüber gestaunt, wie es den Programmieren von Berkeley Softworks, der Entwicklerfirma, gelungen ist, so viel Leistungsstärke aus dem C64 herauszuholen. Dabei wird GEOS auch für Programmierer interessant, denn wenn Sie Ihre Programme unter GEOS entwickeln, sind diese unvergleichlich anwenderfreundlicher als konventionelle C64-Programme – obwohl sich der Aufwand beim Programmieren keinesfalls vergrößert.

Sie können unter GEOS auf alle Routinen der grafischen Benutzeroberfläche zurückgreifen und hantieren spielend mit Pull-down-Menüs, Icons und Windows, wobei sie sich um deren Organisation dank GEOS nicht selbst zu kümmern brauchen. Bevor wir auf die Programmierung unter GEOS zu sprechen kommen, seien Ihnen hier ein paar Abbildungen vorgestellt, die jedem C64-Anwender das Wasser im Munde zusammenlaufen lassen – und hoffentlich auch Ihnen als Programmierer!

Das GEOS-Programm liegt größtenteils im RAM unter dem Das ROM, also an den Adressen \$a000-\$ffff. Das normale C64-Betriebssystem wird von GEOS ausgeblendet, aber zu wenigen Zwecken (z.B. Software-Reset) wieder eingeschaltet.

GEOS arbeitet im hochauflösenden Grafikmodus; während das C64-Betriebssystem einen Bildschirmcode in den Bildschirm-

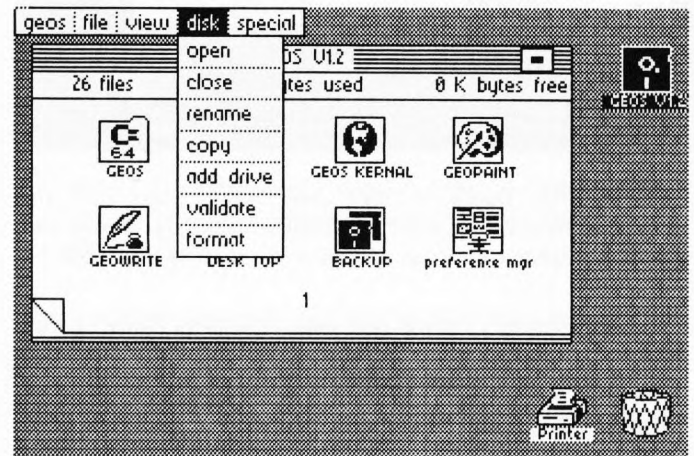


Abbildung 7.1: GEOS-Desktop. Von diesem simulierten Schreibtisch aus steuert der Anwender in der Regel mittels Joystick alle Operationen wie das Löschen, Umbenennen, Kopieren oder Starten von Dateien.

speicher schreibt und der VIC die Darstellung übernimmt, wird unter GEOS die Zeichenmatrix direkt in die Bitmap der hochauflösenden Grafik übertragen. Dadurch lassen sich Text und Grafik mischen. Es können auch mehrere Zeichensätze gleichzeitig dargestellt werden, da ein Zeichen nur einmal in die Bitmap geschrieben wird und das Vorhandensein des Zeichensatzes danach nicht mehr erforderlich ist.

GEOS verfügt auch über eigene Floppy-Routinen, die um den Faktor 5-6 schneller sind als die Datenübertragung des C64-Kernal. Das Diskettenformat wird von GEOS geändert, um noch mehr Informationen zu einem File speichern zu können (z.B. Name des Autors). Des weiteren kennt GEOS eine alternative Filestruktur (VLIR-Files), die vor allem beim Erstellen von Overload-Software (nachladende Programme) äußerst effizient ist.

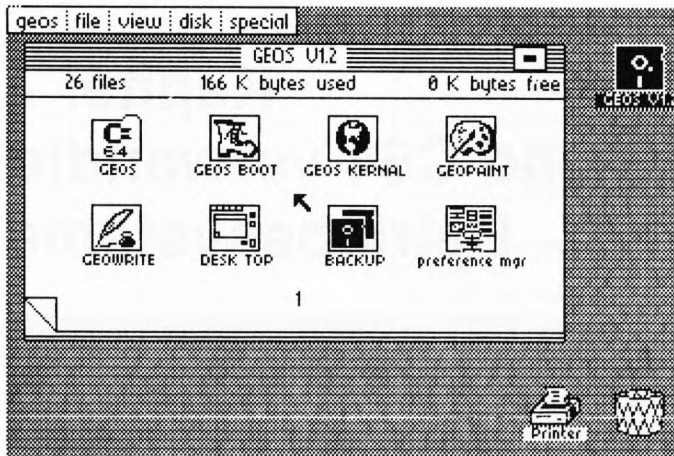


Abbildung 7.2: Durch Auslösen eines Menüpunktes wird eine weitere Menüleiste aufgerollt (Pull-down-Menü). In diesem wird dann die endgültige Funktion oder ein weiteres Menü aufgerufen.

Unter GEOS ist ab \$c100 eine Sprungtabelle zu finden, die wie die Kernal-Sprungtabelle ab \$ff81 funktioniert. Sie enthält 151 Routinen für I/O- und Grafikoperationen, Diskettenzugriffe, Programmierung der Benutzeroberfläche (Windows, Pull-down-Menüs, Icons), Speicheroperationen, Pseudo-Multitasking sowie arithmetische Zwecke (Integerberechnungen).



Abbildung 7.3: Die Kommunikation zwischen Programm und Anwender läuft in Windows ab, die nach ihrer Verwendung wieder zugunsten des vorherigen Bildschirmzustandes entfernt werden.

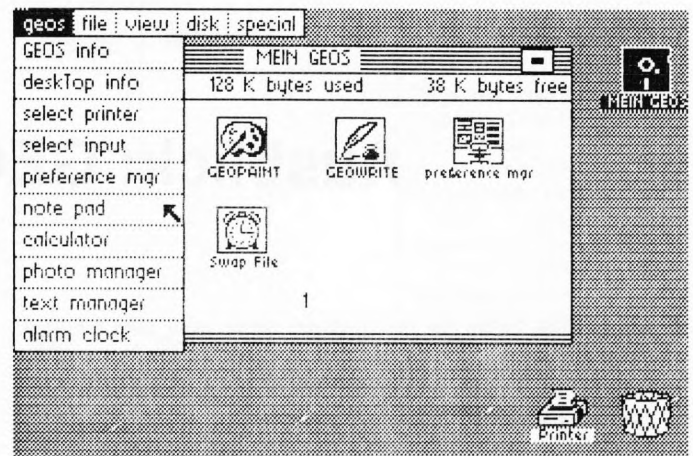


Abbildung 7.4: Desk Accessories sind Hilfsprogramme, die aus anderen Programmen heraus aufgerufen werden können (GEOS-Menü).

GEOS-Programme sind auch durch eine andere Struktur gekennzeichnet. Während bisherige C64-Programme im wesentlichen aus einer kontinuierlichen Folge von Routinen bestehen, sind typische GEOS-Programme nach folgendem Schema aufgebaut:

- In einer Initialisierung werden alle Menüs und Icons am Bildschirm aufgebaut.



Abbildung 7.5: GEOS kann verschiedene Zeichensätze gleichzeitig darstellen (hier in der Textverarbeitung GeoWrite)

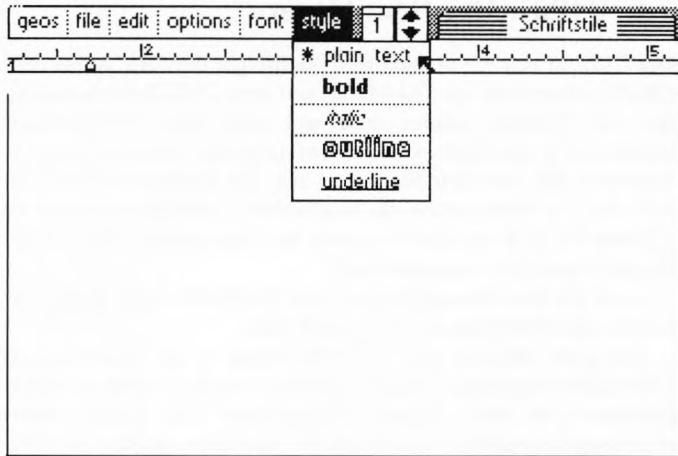


Abbildung 7.6: GEOS beherrscht auch verschiedene Schriftarten spielend (hier in der Textverarbeitung GeoWrite)

– Dann verharrt das Programm in einer Warteschleife, in der GEOS aktiv wird: Es managt dann das Pseudo-Multitasking und stellt fest, ob eine Funktion ausgelöst wird (Anklicken von Menü oder Icon). Falls der Anwender eine bestimmte Funktion angewählt hat, wird die dazugehörige Routine aufgerufen, welche zuvor in der Initialisierung festgelegt wurde. Nach Ausführung der Routine kehrt das Programm in die Warteschleife zurück.

Sie sehen also, daß die Programmierung unter GEOS nicht schwieriger als beim ROM-Betriebssystem ist – nur anders, und zwar besser!

Wenn Sie die Programmierung von GEOS erlernen wollen, möchte ich Sie auf mein Buch »C64 – Alles über GEOS« aus der Commodore-Sachbuchreihe hinweisen, in welchem der Programmierteil über die Hälfte des Umfangs ausmacht und alle GEOS-Routinen vorstellt. Die Vorstellung erfolgt etwa wie in Kapitel 4 dieses Buches. Gleichzeitig erhalten Sie ein komplettes Entwicklungssystem, bestehend aus dem Assembler Hypra-Ass, unzähligen GEOS-spezifischen Hilfsprogrammen sowie dem Monitor SMON, den ich eigens zu diesem Zweck an GEOS angepaßt habe.

Diese Programme und viele Beispiele zur Anwendung und Programmierung von GEOS liegen auf einer doppelseitig bespielten Diskette bei.

Möglicherweise ist auch schon das offizielle Programmierhandbuch »The Official Programmer's Reference Guide« von Berkeley Softworks, den GEOS-Entwicklern, in deutscher Überset-

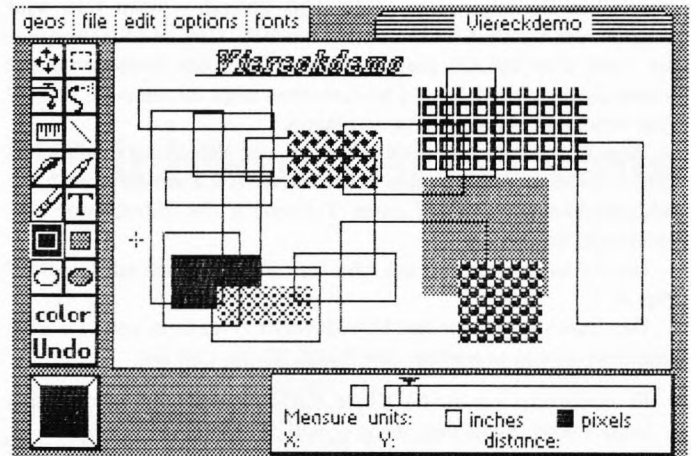


Abbildung 7.7: Verschiedene Füllmuster beim Zeichnen stellen kein Problem dar (hier im Zeichenprogramm GeoPaint; links sehen Sie die »Werkzeugleiste«)

zung erhältlich. Auf jeden Fall läuft im 64'er-Magazin ein GEOS-Programmierungskurs, zu dem sie jedoch unbedingt noch das Buch »C64 – Alles über GEOS« erwerben sollten.

7.2 VC20 – Der Vorläufer

Das C64-Betriebssystem ist im wesentlichen nur eine Anpassung des VC20-Betriebssystems an das größere Bildschirmformat sowie die andere Speicherstruktur. Die Kernal-Sprungtabelle liegt an gleicher Adresse.

Die Basic-Interpreter von C64 und VC20 sind fast identisch. Allerdings liegt der Basic-Interpreter des VC20 an anderer Adresse als der des C64.

Das ROM des VC20 setzt sich ebenso aus zwei 8-Kbyte-Blöcken zusammen wie das C64-ROM.

Diese großen Ähnlichkeiten verwundern nicht übermäßig, ist doch der VC20 der unmittelbare Vorläufer des C64.

7.3 C16, C116 und Plus/4 – Aus der Art geschlagen

Den C16/116/Plus 4 (alle diese drei Geräte werden oft auch nur als C16 bezeichnet, womit dann die Produktreihe gemeint ist) als Nachfolger des C64 zu bezeichnen, wäre nicht ganz korrekt. Zwar

stimmt das Betriebssystem größtenteils überein – beim C16 wird lediglich der Baustein TED anstelle von VIC, CIA und SID verwendet, wird aber ähnlich programmiert –, aber das Kassettenformat weicht so stark ab, daß Sie C64-Kassetten nicht mit einem C16/116/Plus 4 einlesen können – und umgekehrt.

Eine Besonderheit des C16 liegt in der Window-Technik: Mit Hilfe von Steuerzeichen oder Betriebssystemroutinen kann auch die Bildschirmbehandlung auf einen Teilbereich des Bildschirms eingeschränkt werden.

Die Kernal-Einsprünge des C64 funktionieren auch auf C16/116/Plus 4.

Der Basic-Interpreter des C16 (Basic 3.5) stellt in zwei Punkten eine Erweiterung gegenüber dem Basic 2.0 des C64 dar:

- Er verarbeitet bis zu 64 Kbyte RAM (kompletter Adreßraum!) mittels Bank-Switching, wenn er voll ausgebaut ist.
- Basic 3.5 verfügt über Befehle zur Programmierung von Grafik und Sound, über Programmierhilfen, über Diskettenbefehle und Anweisungen zum strukturierten Programmieren. Die Basic-2.0-Befehle werden jedoch uneingeschränkt weiterverarbeitet.

Eine Schwäche des C16/116/Plus 4 ist zweifellos die Unfähigkeit, Sprites darstellen zu können. Dafür beherrscht er weitaus mehr Farben, und hat einen integrierten Maschinensprachemonitor im ROM, den TEDMON. Dessen Routinen können mitunter äußerst nützlich sein, beispielsweise zur hexadezimalen Ausgabe von Integerzahlen.

Das ROM des C16 umfaßt insgesamt 32 Kbyte (die Anwenderprogramme des Plus/4 nicht mitgerechnet), also doppelt so viel wie der C64. Der zusätzlich benötigte Speicher setzt sich aus den Routinen für die weiteren Basic-Befehle sowie dem Maschinensprachemonitor zusammen.

Insgesamt ist die Maschinenprogrammierung des C16/116/Plus 4 komfortabler als beim C64. Die Umsetzung von C64-Programmen sollte ernsthaft in Erwägung gezogen werden, zumal der Assembler HYPRA-ASS auf den C16 umgesetzt wurde und auch C64-Quelltexte weiterverarbeitet (erhältlich bei Markt & Technik).

Ein kommentiertes ROM-Listing des C16 ist in der Commodore-Sachbuchreihe erhältlich: Christian Quirin Spitzner: »C16, C116 und Plus/4 ROM-Listing« (436 Seiten)

7.4 C128 – Der Nachfolger

Der C128 ist nun der unmittelbare Nachfolger des C64. Außer dem C64-Betriebssystem im C64-Modus und dem CP/M-Betriebssystem, das von Diskette geladen wird und unter dem Z80-Prozessor abläuft, hat er sein eigenes C128-Betriebssystem. Dieses ist eine Erweiterung des C64-Betriebssystems um die Window-Technik, die auch der C16 bietet, sowie die Möglichkeit, wahlweise auch mit 80 Zeichen pro Zeile zu arbeiten, wofür ein Chip namens VDC (Video Display Controller) eingesetzt wird.

Auch der Maschinensprachemonitor TEDMON ist im C128 vorhanden, der TED-Chip des C16 jedoch nicht.

Der große Speicher des C128 (128 Kbyte in der Grundversion) wird durch sogenanntes Bank-Switching verwaltet, wofür ein neuer Baustein – die MMU (Memory Management Unit, deutsch »Speicherverwaltungseinheit«) zuständig ist. Ansonsten werden die C64-Bausteine (VIC, SID, CIA) sowie der VDC für den 80-Zeichen-Modus verwendet.

Das leistungsstarke Basic 7.0 beinhaltet nun alle Befehle von Basic 2.0 und 3.5 sowie viele weitere, auch zur Spriteprogrammierung, die mit dem C128 im Gegensatz zum C16/116/Plus 4 möglich ist.

Das C128-ROM ist 44 Kbyte groß (12 Kbyte Betriebssystem, 28 Kbyte Basic-Interpreter, 4 Kbyte Monitor) und vollständig im »C128 ROM-Listing« aus der Commodore-Sachbuchreihe beschrieben (456 Seiten).

Wie Sie Ihre C64-Programme (Basic und Maschinensprache) auf den C128 umsetzen, erfahren Sie neben vielen anderen Informationen rund um den C128 in meinem Buch »Vom C64 zum C128 – Tips & Tricks« (Markt & Technik, 290 Seiten inklusive Diskette mit Hilfs- und Beispielpogrammen, darunter viele Assembler-Quelltexte). Falls Sie einen Systemwechsel auf den C128 in Erwägung ziehen, lernen Sie in diesem Buch die C128-Eigenheiten kennen; da auf Ihre C64-Kenntnisse aufgebaut wird, beherrschen Sie bald das mächtige Basic 7.0 und das Betriebssystem des C128.

Dabei erhalten Sie, wie es der Untertitel des Buches sagt, viele Tips & Tricks sowie Hilfsprogramme auf Ihren garantiert erfolgreichen Weg zum C128. Somit ist klar zu verstehen, warum der C128 als C64-Aufsteigergerät eine feste Größe ist.

Stichwortverzeichnis

* 406
+ 406
- 406
/ 406
< 406
= 406
> 406

A

ABS 406
Addition 405
Additionsübertrag 497
Adresse 290
Adreßtabelle 292, 394
AND 405 f.
Anführungszeichenmodus 498
Anti-Reset-Kennung 404
Applikation 382
Argumente 396
Array-Bereich 391
Array-Elemente 391
Array-Variablen 390
ASC 406
ASCII-Code 291
ASCII-String 497
ASCII-Tabellen 291
Assembler 385
Assembler-Quelltexte 289
ATN 406
Aufwärts-Scrolling 498
Ausgabefile 384

B

Bank-Switching 512
Basic 2.0 512

-, 3.5 512
-, 7.0 512
Basic-Befehl 289
Basic-Fehlermeldungen 407
Basic-Interpreter 382, 387
Basic-Kaltstart 404
Basic-Kernal-Aufrufe 401 f.
Basic-Kernal-Einsprung 402
Basic-NMI 404
Basic-Programm 383
Basic-Programmanzeiger 384
Basic-ROM 289
Basic-ROM-Vektoren 402
Basic-Speicher 388
Basic-Stack 400
Basic-Vektoren 388
Basic-Zeiger 393
Basic-Zeilenende 495
Basis 397
Basisadresse 291
Baudraten 500
Befehle 404
Befehlsende 495
Befehlsroutinen 394
Befehlstoken 394
Benutzeroberfläche 509
Berkeley Softworks 509
Betriebssystem 381
Betriebssystem-ROM 401
Bildschirmspeicher 387, 498
Bitmap 509
Bit-Trick 292
BRANCH-Befehle 290

C

C16 511
C116 511

C64-Modus 512
CHKBCL 397
CHKBRO 397
CHKBYT 397
CHKCOM 397
CHKNUM 396
CHKSTR 396
CHKTYP 396
CHR\$ 406, 496
CHRGET 396, 403
CHRGET-Routine 388
CHRGOT 396, 403
CIA 384, 499
CLOCK 499
CLOSE 406, 497
CLR 406
CMD 406
Compiler 382
Computer 381
CONT 406
Copyright-Information 404
COS 406
CP/M 512
Cross-Reference 292
CRSR RIGHT 496
Cursor 289, 387

D

DATA 406, 499
Datasette 381
Datenport A 499
Datensichtgerät 387
Datentyp 396
Datenwort 499
DEF 406
Dekrementierbefehle 291

Deskriptoren 393
 DIM 406
 Dimensionen 393
 Direktmodus 384
 Disassembler 289
 Division 405
 DIVISION BY ZERO ERROR 497
 druckende Zeichen 387
 Drucker 381

E

Editor 394
 Ein-/Ausgabe 382
 Ein-/Ausgabebereich 499
 Ein-/Ausgabe-Fehler 384
 einfache Variablen 390
 Eingabefile 384
 Eingabemodus 388
 Einschaltmeldung 388,498
 END 406
 Endlosschleife 394
 Endmarkierung 389
 ERROR-Routine 498
 EXP 406
 Exponent 397
 Exponentenaddition 497
 Exponentenbyte 398

F

FAC #3 497
 –, #4 497
 FACWRD 396
 Farb-RAM 498
 Farbsteuerzeichen 498
 Fehlerbehandlungsroutine 410
 Fehlercode 410
 Fehlereinsprung 400
 Fehlermeldungen 384,400
 Fehlervektor 410
 File 384
 Fileeinträge 385
 Filenummer 385
 Fileparameter 499
 Filespezifikationen 385
 Filetabelle 384,499
 Firmware 289,381

Fließkomma-Akkumulatoren 398
 Fließkommaformat 397
 Fließkommavariablen 390
 Fließkommazahlen 391,397
 Floppy 381
 FLPT 398
 FLPT-Format 398
 FN 406
 FOR 406
 FOR/NEXT-Eintrag 401
 FOR/NEXT-Variablenzeiger 408
 FRE 406
 FRMNUM 396
 FRMSTR 396
 Funktionen 399,404

G

Garbage-Collection 394
 GEOS 382,509
 Geräteadresse 385
 geschweifte Klammer 290
 GET 406,496
 GETBYT 396
 GETWRB 396
 Gleitkommazahlen 397
 GO 406
 GOSUB 406
 GOSUB/RETURN-Eintrag 400
 GOTO 406
 Graphical Environment Operating
 System 509
 Groß-/Grafik-Zeichensatz 387

H

Hardware 381
 Hardware-Reset 404
 Header 499
 Headerbyte 408
 Hilfsspeicher 289
 hochauflösender Grafikmodus 509
 Home-Position 387
 Horner-Schema 399
 Hypra-Ass 385,403,511
 Hypra-Load 404

I

Icons 509
 IEC-Bus 499
 IERROR 410
 IF 406
 indizierte Variablen 390
 INPUT 406,496
 INPUT# 406
 INPUT\$ 496
 INT 406
 Intervariablen 390,392
 Interpreter 289
 Interpreter-ROM 401
 Interpreterschleife 394,404
 Interrupt-Routine 500
 Interrupts 383
 IRQ-Routinen 383,499
 IRQ/BREAK-Routine 500

J

jiffy 383
 –, clock 383
 Joystick 381

K

Kaltstart-Routine 388
 Kassettenoperationen 401
 Kassettenpuffer 499
 Kernäl 382
 Kernäl-ROM 382,401
 Kernäl-Sprungtabelle 382
 Kernel 382
 Klein-/Groß-Zeichensatz 387
 Koeffizienten 399
 Kommentare 289

L

Label 291
 LDTB1 498
 LEFT\$ 406,496
 LEN 406
 LET 406
 LIFO-Struktur 400
 Lightpen 381

Line by line 290
LINGET 396
Linkpointer 389
linksbündig 398
Linksbündigkeit 399
LIST 406
LISTEN-Signal 477
LOAD 406, 497
LOADING 499
LOG 406
Low-High-Darstellung 389
Low-High-Format 291

M

Mantisse 397
Mantissenbyte 398
Maschinensprache 289
Maus 381
Memory Dump 291
–, Management Unit 512
–, Map 501
MFLPT 398
MFLPT-Format 398
MID 406
MID\$ 496
MMU 512
Mnemonic 291
Modulkennung 384, 500
Monitor 381
Multiplikation 405

N

Nachkommastellen 398
NEW 406
NEXT 406, 496
NMI-Routine 404
normalisieren 497
normalisierte Darstellung 398
Normalisierung 398
NOT 405 ff.
Nullbyte 388

O

Offset 291
ON 406
Opcode 290

OPEN 406, 497
Operating System 382
Operatoren 404
OR 405 f.
OUT OF MEMORY ERROR 410
OVERFLOW ERROR 391, 497
Overload-Software 509

P

Parameter 394
PEEK 406
PLAY-Taste 499
Plus/4 511
POKE 406
Polynom 399
Polynomroutinen 401
Polynomtabelle 497
POS 406
Potenzierung 405
PRINT 406
PRINT# 406
Priorität 405
Prioritätsflags 405
Programmbereich 408
Programmende 390
Programmschleifen 290
Prozessorport 383
Prozessorstapel 384
Pseudo-BITs 292
Pseudo-JMP 290 f.
Pseudo-Multitasking 383, 510
Pull-down-Menüs 509

R

RAM-Hilfsspeicher 388
RAM-Programme 403
RAM-Vektor 404
READ 406, 496
Read Only Memory 381
Rechtsverschiebung 497
REM 406
RES 497
Reset 384
Reset-Routine 403
Resetschalter 384
RESTORE 406

RETURN 406
Reverssschrift 387
RIGHT\$ 406, 496
RND 406
ROM-Kennung 404
ROM-Listing 289
ROM-Modul 384
ROM-Routine 289
ROM-Tabellen 403
ROM-Vektor 384
ROMs 381
RS232 499
RS232-Empfangspuffer 499
RUN 406
RUN/STOPRESTORE 404
runden 497

S

SAVE 406, 497
Schleifenrichtung 401
Schleifenvariablen 401
Schlüsselwörter 404
Schriftfarbe 387
SEARCHING 499
Sekundäradresse 385
Selbstmodifikation 403
sequentielle Datenspeicherung 390
SGN 406
Shift L 407
SID 384
simulierte Subtraktionen 291
SIN 406
SMON 290, 511
Software 381
SPACE 496
SPC 406
Speicheraufteilung 393
Speicherbelegung 501
Speicherblockverschiebung 408
Splitting 290
SQR 406
ST 384, 500
Stapel 400
Stapeleinträge 407
Stapelmanipulationstrick 405
Stapelüberlauf 410
Startbitprüfung 499

Statusbyte 384
STEP 406
Steuerfunktion 387
Steuermeldung 385, 399
Steuerzeichen 387
STOP 406
STOP-Taste 383, 495
STR\$ 406
String 495
Stringdeskriptor 395
String-Inhalt 393
Stringinhaltsspeicher 393
String-Müll 395
Stringoperationen 395
Stringvariablen 390
Stringverknüpfung 395, 496
STROUT-Format 407
Subtraktion 405
Symbolik des ROM-Listings 290
Synchronisation 499
SYS 406
Systemeingabepuffer 394
Systemvariable 385

T

TAB(406
TALK-Signal 477
TAN 406
Tastatur 383
Tastatur-Eingabeschleife 498
Tastaturpuffer 383, 498
Tastaturtabelle 498
-, #0 498

-, #1 498
-, #2 498
-, #3 498
TED 512
TEDMON 512
Teilergebnisstring 395
temporärer Stringstapel 496
THEN 406
Timer-Initialisierung 500
Timerkonstanten 500
TO 406
Token 405
Token-Format 394
Tokenisierung 405
Tokenisierungsroutine 405
Tokenisierungstabelle 406
Trennzeichen 394

U

UNLISTEN-Signal 477
UNTALK-Signal 477
USR 406

V

VAI 406
Variablen 388
Variablenbereich 408
Variableneintrag 391
Variablennamen 390
Variablenspeicherplatz 408
Variablentypen 390
VC 20 511

VDC 512
Vergleichsoperatoren 405
VERIFY 406, 497
Verzweigung 290
Verzweigungsbedingung 292
Verzweigungspfeile 290
VIC 384
Video Display Controller 512
VLIR-Files 509
Vorkommastelle 398
Vorzeichen 497
vorzeichenbehaftete Bytes 397
Vorzeichenbit 397
Vorzeichenwechsel 405

W

waagrechte Linien 290
WAIT 406
Warmstart 388, 394
Windows 509

Z

Z80-Prozessor 512
Zahlenformate 291, 397
Zeichenfarbe 387
Zeichenmatrix 509
Zeileneintrag 388
Zeilenende 394
Zeilenendmarkierung 389
Zeileninhalt 389
Zeilennummer 389, 495
Zeropage-Adresse 291

C64 für Insider

FLORIAN MÜLLER,

geboren am 21. Januar 1970, besucht derzeit ein Münchner Gymnasium. Er beschäftigt sich seit 1983 mit Heimcomputern, insbesondere mit dem Commodore 64. Seit 1985 ist er Mitarbeiter der Redaktion 64'er, für die er schon zahlreiche Artikel verfaßt und bearbeitet hat. Im Markt&Technik-Buchverlag erschienen folgende Bücher: »Vom C64 zum C128 - Tips & Tricks«, »C64 - Alles über GEOS«. Für die Zukunft sind weitere Veröffentlichungen zu GEOS auf C64 und C128 sowie zu seinem neuen Computer, dem Commodore Amiga, geplant.

Zum Inhalt:

Wie kein anderer Computer, ist der C64 mit seinen Besitzern gewachsen. Heutzutage kann es ein vollausgebauter C64 auch mit größeren Systemen aufnehmen. Das ist zu einem großen Teil auf den hohen Wissensstand vieler C64-Anwender zurückzuführen.

Dieses Buch zeigt Ihnen auf über 500 Seiten, wie der C64 auf Maschinenebene arbeitet. Sie brauchen aber keine profihaften Assemblerkenntnisse mitzubringen, um dieses Buch effektiv einzusetzen. Der gewünschte »Durchblick« stellt sich ein, wenn Sie es als Werkzeug zur Programmierung benutzen. Dabei entwickeln Sie mehr und mehr ein tiefgreifendes Verständnis für das »Innenleben« Ihres C64 und

erkennen auch komplizierte Programmstrukturen im ROM.

Es ist informativ, aber auch verständlich gehalten und besteht aus ROM-Listing zu Basic-Interpreter und Betriebssystem, Cross-Reference, Systemdokumentation sowie Speicherbeschreibung. Dabei ist es auch für die Leser unter Ihnen interessant, die bereits ein ROM-Listing besitzen, da die gegebenen Informationen weit über bisherige Bücher dieser Art hinausgehen.

Das kommentierte ROM-Listing ist nach einem völlig neuartigen Konzept aufgebaut, das sich nicht nur in einem größeren Format widerspiegelt; es überzeugt sowohl durch Informationsfülle (jedes Byte wird kommentiert) als auch durch herausragende optische Gestaltung mit Rastern, Verzweigungspfeilen und ergänzenden Kommentaren. Es ist ein kommentiertes Monitorlisting, das aber um alle diejenigen Elemente eines Assembler-Quelltextes ergänzt wurde, die die praktische Anwendung unterstützen.

Ein eigenes Kapitel behandelt den effektiven Einsatz des ROM-Listing und enthält obendrein die erste Cross-Reference über den gesamten C64-Speicher. In einem weiteren Abschnitt erfahren Sie alles über die grundsätzlichen Komponenten und Begriffe, worauf eine umfangreiche Systemdokumentation mit Beschreibung jeder ROM-Routine hinsichtlich ihrer Nutzung und Programmierung folgt. Diese Doku-

mentation wird von einer knappen Routinenübersicht ergänzt und einer Memory Map, die alle relevanten Speicherzellen des C64 anspricht.

»C64 für Insider«

- praxisnah und informativ - behandelt folgende Bereiche:

- Systemüberblick mit ausführlicher Erklärung von Begriffen und Zusammenhängen der verschiedenen Komponenten
- eingehende Dokumentation aller ROM-Routinen
- Cross-Referenz-Liste über den gesamten C64-Speicher
- Memory Map (Speicherbelegungskarte)

Zum Abschluß wird noch kurz das neue C64-Betriebssystem GEOS vorgestellt. Ein ausführliches Stichwortverzeichnis rundet dieses Werk ab, das bestimmt jedem ernsthaften Programmierer zu einem nützlichen Hilfsmittel wird und einen neuen Standard für Systemhandbücher darstellt.

Hardware-Anforderungen:

C64 oder C64c (auch die Besonderheiten von SX 64 und C128 im C64-Modus finden Berücksichtigung)

Software-Anforderungen:

Assembler für den C64 (z.B. Hypra-Ass, Programmservice 64'er Magazin, Bestellnummer: L68507A)

ISBN N 3-89090-481-5



4 001057 904811



DM 59,-

sFr 54,30

öS 460,20