BASIC COMPILER

von T.Barndt & F.Thielen

mit Fließkommerarithmetik

für

Schneider CPC 464

Herzlichen Glückwunsch zum Kauf dieses Compilers!

Mit diesem hochwertigen Software-Produkt verfügen Sie über ein wertvolles Werkzeug zur Übersetzung von Programmen einer Computer-Hochsprache (BASIC) in Z80-Maschinensprache. Wir hoffen, daß Sie bei der Arbeit mit diesem Programm die gleiche Freude haben wie wir bei der Entwicklung.

Die Autoren

Frank Thielen & Thomas Barndt, im Juli 1985

SOFTWARE TEAM
JOACHIM GÜNSTER
MÜHLEN STR. 12
5431 BODEN

1. Was ist ein Compiler?

In Ihrem Schneider CPC-464 befindet sich ein sogenannter Interpreter, der Ihnen sofort beim Einschalten des Rechners zur Verfügung steht. Mit diesem Interpreter können Sie sogar viel längere Programme bearbeiten als mit diesem Compiler, und der Befehlsvorrat des eingebauten Schneider-Basic ist größer. Wozu also ein Compiler?

Um den Unterschied zwischen Interpreter und Compiler zu verdeutlichen, muß zuerst etwas über ihre Funktionsweise gesagt werden. Ein Interpreter liest ein (Basic-)Programm Zeichen für Zeichen und übersetzt (interpretiert) es in ausführbare Anweisungen in Maschinensprache. Steht also im Programm eine Schleife mit einer Anweisung, die einige tausend mal durchlaufen wird, so muß der Interpreter diese Anweisung auch einige tausend mal übersetzen und ausführen. Die Übersetzungszeit addiert sich also zur reinen Ausführungszeit der übersetzten Anweisungen.

Bei einem Compiler entfällt dagegen die Übersetzungszeit bei der Abarbeitung eines Programms, da hier das Programm einmal für alle späteren Läufe übersetzt wird. Wenn Sie ein Programm geschrieben haben, wird es zuerst auf syntaktische Richtigkeit (Schreibfehler usw.) überprüft. Dabei werden auch solche Fehler entdeckt, die ein Interpreter möglicherweise erst bei einem ganz bestimmten Programmlauf (etwa einer bestimmten Eingabe) bemerkt. Ein Programmteil, das normalerweise übersprungen wird, wird vom Interpreter erst dann bearbeitet und auf Fehler überprüft, wenn es auch tatsäclich einmal angesprungen wird. Dies ist eine enorme Fehlerquelle, da Programme so auch nach umfangreichen Prüfungen immer noch Fehler enthalten können, die dann irgendwann einmal zu Tage treten. Ein compiliertes (übersetztes) Programm kann dagegen keine Syntaxfehler mehr enthalten, da der Compiler diese alle beim Übersetzen entdeckt und auch gar nicht übersetzen kann, wenn das sogenannte Quellprogramm noch Fehler enthält.

Ein gravierender Nachteil im Vergleich zum Interpreter ist allerdings der erhöhte Speicherplatzbedarf des Compilersystems. Einmal braucht das Quellprogramm einen bestimmten Speicherplatz (beim Interpreter auch), dazu kommt aber noch der Platz für den Compiler selbst, der normalerweise im Vergleich zu einem Interpreter gleichen Sprachumfangs mehr Platz braucht. Zusätzlich muß bei einem Compiler das übersetzte Programm, der sogenannte Objektcode, gespeichert werden. Auch benötigt der Compiler Zwischenspeicher für Tabellen, symbolische Adressen usw.

Nachteilig gegenüber einem Interpreter ist auch die sogenannte Compilezeit, naemlich die Zeit, die der Compiler zum Erstellen des Objektcodes benötigt.

Demgegenüber ist aber der enorme Zeitvorteil zu sehen, den ein compiliertes Programm bietet. Das folgende Programm z.B.

```
10 i%=-16384
20 POKE i%,0
30 i%=i%+1
40 IF i%<>0 THEN 20
```

(es löscht den Bildschirmspeicher) benötigt im Schneider-Basic 55.16 Sekunden, das compilierte Programm läuft 1.85 Sekunden.

2. Funktionsweise dieses Compilers

Dieser Compiler, der ein etwas modifiziertes Basic, das sogenannte FTB-Basic, versteht, übersetzt Programme direkt in Z80-Maschinensprache. Dabei werden etliche eingebaute (ROM-)Routinen des Schneider-Basic und des Betriebssystems des CPC-464 benutzt. Deshalb sind die erzeugten Programme extrem kurz; auch weil sie compilereigene Routinen nur dann enthalten, wenn sie tatsächlich bebutzt werden. Wenn Sie also in einem Programm keine Stringaddition benutzen, wird die dafür benötigte Routine auch nicht in den Objektcode übernommen.

Der Compiler benötigt für die korrekte Übersetzung eines Programms fünf Durchläufe, sogenannte Passes. Die ersten beiden Durchläufe dienen der syntaktischen Überprüfung des Quellprogramms, sie werden auch Parser genannt und können ohne die nachfolgende Übersetzung gestartet werden. Wollen Sie z.B. ein noch unvollständiges Programm auf Fehler überprüfen, brauchen Sie es nicht gleich zu übersetzen, sondern können zuerst einen Syntax-Check vornehmen lassen und sparen so die Übersetzungszeit.

Pass 1 des Parsers überprüft dabei alle Sprachelemente statisch, d.h. voneinander unabhängig, auf syntaktische Richtigkeit. Dies können Schraibvehler, Zuweisungen mit Operanden falschen Typs, inkorrekte Variablennamen usw. sein. Werden hier schon Fehler gefunden, wird Pass 2 gar nicht erst gestartet, sondern gleich abgebrochen. In beiden Durchläufen, also Pass 1 und 2, werden alle Zeilen auf Fehler überprüft und normalerweise auch alle Fehler gemeldet. Sind in einer Zeile jedoch mehrere Fehler, so können eventuell nicht alle erkannt werden. Ein Beispiel:

```
100 IF a$-"Otto" THENN 200
```

Offensichtlich fehlt in dieser Zeile der Vergleichsoperator ('-' ist kein gültiger), und das THEN sollte ja wohl auch etwas anders aussehen. Nachdem der Parser den ersten Fehler (das Fehlen des Vegleichsoperators) gemeldet

hat, kann er in dieser Zeile nicht mehr (vernünftig) weiter überprüfen, so daß der zweite Fehler unentdeckt bleibt. Haben Sie aber den ersten Fehler verbessert, ohne den zweiten zu bemerken, wird der Parser beim nächsten Mal auch das falsche THEN melden.

Der zweite Durchlauf des Parsers überprüft alle Anweisungen des Programms im Zusammenhang. So wird gemeldet, falls Sie zu einem FOR das NEXT vergessen haben (oder umgekehrt), ob Sie alle benutzten Felder DIMensioniert haben und ob alle Zeilennummern, die Sie anspringen wollen, auch existieren.

Der eigentliche Übersetzer besteht wiederum aus drei Durchläufen, von denen der erste der eigentliche Compiler ist. Vorarbeit dazu hat schon der Parser im ersten Durchlauf geliefert, indem er alle benutzten Variablen und Felder herausgesucht hat. Diese Tabelle wird zur Speichererspарnis in den Bildschirmspeicher geschrieben, wo auch der Compiler seinen Code erzeugt. Wundern Sie sich also nicht, wenn seltsame Punkte auf Ihrem Schirm erscheinen, dies sind die Bitmuster der Tabellen und des Zwischencodes.

Der Übersetzer also, Pass 3 des Compilers, übersetzt alle Anweisungen Ihres Programms in einen Zwischencode, der dann nachher in den Maschinencode übertragen wird. Sie können besonders in dieser Phase schön beobachten, wie der Compiler sich das Programm vornimmt und der Bildschirm beschrieben wird.

Nach der eigentlichen Übersetzung werden die symbolischen Adressen in absolute umgewandelt (Sie können sehen, wie einige Bytes auf dem Bildschirm geändert werden). Auch wird schon ein Teil des Objektcodes in den Speicher geschrieben.

Die Arbeit vervollständigt Pass 5, der auch am längsten braucht. Leider ist hier von der Arbeit des Compilers nichts zu sehen; der Bildschirm bleibt unverändert. Pass 5 erzeugt den fertigen Objektcode und meldet die erfolgreiche Übersetzung mit der ersten Speicheradresse, die das Programm benutzt, der Anzahl der benutzten Speicherstellen und der Startadresse für den Aufruf.

Jede der fünf Phasen der Übersetzung wird außerdem durch eine individuelle Farbe des Randes gekennzeichnet; die Reihenfolge ist schwarz (Farbe 0), blau (2), rot (6), grün (18) und gelb (24). Auch Grünmonitorbesitzer können so am immer heller werdenden Rand den Fortschritt der Compilierung erkennen.

3. Tips zur Programmierung

Hier sollen Sie nun einige Tips erhalten, wie Sie Ihre Programme effizienter gestalten können und wie einige Dinge im Detail (in dem ja bekanntlich der Teufel steckt) funktionieren.

Am Anfang eines Programms stehen meist die Feldvereinbarungen (DIM-Anwei-

sungen). Abweichend zum Schneider-Basic müssen diese Vereinbarungen jedoch nicht zu Beginn stehen: Da der Compiler vor der Codegenerierung eine Variablentabelle anlegt, ist es unerheblich, an welcher Stelle Sie DIMensionieren. Es müssen aber alle benutzten Felder vereinbart werden, auch solche, die weniger als zwölf Elemente enthalten. Alle festen Indizes, also solche, die durch eine Konstante erfolgen (wie z.B. alpha_1$(34)), werden im Pass 2 auf Indexüberschreitung überprüft. Variable Indizes, wie etwa feld%(i%), können natürlich erst während der Laufzeit, das heißt während der Ausführung des Objektcodes, überprüft werden. Aud Wunsch können Sie diese Prüfung aber auch verhindern. Dadurch wird das Objektprogramm etwas schneller und kürzer, aber wenn Sie dann schreibenden Zugriff auf ein Feld mit einem falschen Index vornehmen, kann es sein, daß Sie entweder im Nachbarfeld landen oder im gerade laufenden Programm, wodurch der Rechner prompt abstürzt. Wenn Sie jedoch einiges Geschick haben, können Sie mit dieser Methode absichtlich Nebeneffekte erzeugen. Alle Variablen und Felder werden in der Reihenfolge ihres Auftretens oder ihrer Dimensionierung reserviert. Integervariablen brauchen (als Einzelvariablen oder pro Feldelement) 2 Byte, Realvariablen 5 Byte und Strings 81 Byte (dabei enthält das erste Byte die Länge des Strings und alle folgenden die Zeichen). Wenn Sie also

```
10 DIM feld%(3)
20 i%=0
30 j%=2
40 DIM array%(1)
```

schreiben, landen Sie (mit unterdrücktem Index-Check) durch k%=4 und feld%(k%) in der Variable i%, durch k%=5 und feld%(k%) in j% und durch k%=-3 mit array%(k%) in feld%(3). Konstante Indizes können für solche Tricks nicht verwendet werden, da schon Pass 2 solche Fehler meldet und verhindert.

Sprechen Sie ein Feld ohne Index an, so erhalten Sie die Komponente 0; so z.B. entspricht array% array%(0). Wenn Sie übrigens ein Feld mehrfach dimensionieren, so werden nach der ersten Vereinbarung alle weiteren ignoriert, es gelten dann also auch die Indexgrenzen der ersten Dimensionierung.

Alle Variablennamen dürfen aus maximal acht Zeichen (das Typkennzeichen nicht mitgerechnet) bestehen. Dabei ist das erste Zeichen ein Buchstabe, alle weiteren können Buchstaben, Ziffern oder der Unterstreichstrich sein. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden, auch nicht in Schlüsselworten. Leerzeichen dürfen überall und beliebig verwendet werden; im Gegensatz zum Schneider-Basic können sie aber auch ganz weggelassen werden, so daß Zeilen wie

```
10IFa%=b%THEN20
```

korrekt übersetzt werden. Allerdings sollten Sie zur besseren Lesbarkeit der Programme dies nicht überstrapazieren.

Die verschiedenen Variablentypen wurden schon angesprochen, sie entsprechen im Wesentlichen denen des Schneider-Basic. Zusätzlich können bei Zuweisungen im Programm Integervariablen Werte von 32768 bis 65535 zugewiesen werden (dieser Bereich gilt auch für Integerkonstanten wie z.B. hinter 'CALL'). Allerdings ist dieser Bereich den negativen Integerzahlen überlagert, d.h. 65535 ist dasselbe wie -1, 65534 entspricht -2 usw. Da die Eingabe ('INPUT') von Integerzahlen über das Betriebsystem abläuft, können Sie das hier nicht, jedoch werden hier ausnahmsweise auch Hexadezimal- und Binärzahlen wie im Schneider-Basic angenommen. Seien Sie jedoch vorsichtig, wenn Sie nämlich bei einer Integer-Eingabe eine Realzahl wie z.B. 3.141 eingeben, erhalten Sie genau wie im umgekehrten Fall ein undefiniertes Ergebnis.

Raelzahlen haben den gleichen Definitionsbereich wie ueblich, Strings dagegen eine feste Länge von 80 Zeichen. Das heißt natürlich nicht, daß Sie nun Stringkonstanten auf 80 Zeichen auffüllen müssen (wie dies z.B. in Standard-Pascal bei einer Zuweisung an ein PACKED ARRAY nötig ist), vielmehr sind lediglich fest 80 Zeichen für einen String reserviert (plus 1 Byte Längeninformation), die Sie nicht vollschreiben brauchen. Wenn Sie durch Stringaddition längere Strings aufbauen, gehen alle über 80 hinausgehende Zeichen verloren. Die Eingabe von Strings erfolgt wie bei 'LINE INPUT' im Schneider-Basic, führende Leerzeichen und Kommas werden also übernommen.

Wie üblich bei einem Compiler werden beim Start eines Programms die Variablen NICHT initialisiert (mit Null oder leeren Strings vorbelegt), so daß Sie dies bei Bedarf selbst tun müssen. In einem eine Summe berechnenden Programm wie

```
10 PRINT "Eingabe Endwert fuer Summation: ";
20 INPUT endwert%
30 FOR zaehler%=1 to endwert%
40   summe%=summe%+zaehler%
50 NEXT zaehler%
60 PRINT "Die Summe betraegt ";
70 PRINT summe%
```

würde nur durch einen großen Zufall das richtige Ergebnis herauskommen, da summe% im Gegensatz zum Schneider-Basic nicht mit Null vorbelegt ist. In diesem konkreten Beispiel sollten Sie also die Zeile

```
5 summe%=0
```

hinzufügen. Normalerweise bleiben die Variableninhalte übrigens bis zum nächsten Programmlauf erhalten, so daß sie also weiterverwendet werden können. Dies gilt natürlich nur, falls Sie die Anzahl der Variablen und Feldelemente und ihre Reihenfolge nicht verändert haben!

FTB-Basic ist äußerst typenstreng (wie z.B. auch ADA), Zuweisungen zwischen Variablen verschiedenen Typs sind verboten. Dafür stehen Ihnen Konversionsfunktionen wie CREAL und CINT zur Verfügung. Alle Variablen und Feldnamen tragen ihr Typkennzeichen, es darf nicht weggelassen werden. Auch bestimmte Operatoren sind nur bestimmten Typen zugeordnet, die ganzzahlige Division (dargestellt durch den umgekehrten Schrägstrich) und die Restbildung 'MOD' ist nur für Integerzahlen erlaubt, die normale Division '/' und die Potenzierung '^' nur für Realzahlen.

Ein besonderer Gag ist der Präfix-Operator '-', der nicht nur für die Vorzeichenumkehr bei Real und Integer sorgt, sondern auch bei Strings erlaubt ist: Er dreht den gesamten String um. So wird aus -"Compiler" z.B. "relipmoC", und -"NEGER" ergibt "REGEN".

An Funktionen steht Ihnen eine ganze Reihe zur Verfügung; zu erklären bleibt:

WAIT$ entspricht INKEY$, wartet aber auf einen Tastendruck. Eine Schleife

```
10 a$=INKEY$
20 IF a$="" THEN 10
```

können Sie bequem durch

```
10 a$=WAIT$
```

ersetzen.

ABS und SGN sind nur für Real-Zahlen erlaubt; für Integerzahlen stehen IABS und ISGN zur Verfügung.

DEEK (Doppel-pEEK) liest eine 16-Bit-Zahl aus zwei aufeinanderfolgenden Speicherstellen (die erste Speicherstelle ist das niederwertige Byte) und liefert demzufolge Werte zwischen -32768 und 32767 (Zahlen über 32767 wie üblich als negative Werte).

Das hervorragende Schneider-Basic mußte für diesen Compiler natürlich erheblich gekürzt werden; schließlich sollte der Compiler ja auch in den Rechner passen. Trotzdem wurden einige Erweiterungen aufgenommen, um die Programmierarbeit zu erleichtern.

Zu den üblichen FOR-NEXT und WHILE-WEND wurde die aus Pascal bekannte REPEAT-UNTIL-Schleife übernommen. Alle Anweisungen zwischen REPEAT und UNTIL werden so lange wiederholt, bis der logische Ausdrucke hinter UNTIL nicht mehr erfüllt ist. Im Gegensatz zur WHILE-Schleife steht die Schleifenbedingung also am Ende der Schleife und braucht nicht wie bei WHILE vor Beginn der Schleife abfragbar zu sein. Deshalb wird eine REPEAT-Schleife auch mindestens

einmal durchlaufen. Geschickt lassen sich so falsche Eingaben abfangen, nach dem Motto:

```
REPEAT
  INPUT
UNTIL Eingabe in Ordnung
```

Eine Restriktion zum normalen Basic, die Sie unbedingt beachten müssen, sind die (arithmetischen) Ausdrücke. Operatoren wie '+' oder '*' dürfen nur einfache Variablen oder Konstanten als Operanden haben, keine weiteren Ausdrücke oder Feldelemente. Ein Minus-Operator darf ebenfalls nur vor einem Operanden, nicht vor einem Ausdruck oder Feldelement stehen. Dazu beachten Sie am besten die Syntaxdiagramme, die Ihnen alle nötigen Informationen geben und an die sich auch der Parser hält.

Weitere Erweiterungen gegenüber dem Schneider-Basic sind:

LPRINT: Syntaktisch genau wie PRINT, nur gelangen Ausgaben auf den Drucker (im Microsoft-Basic üblich, entspricht PRINT # 8).

CALL: Hier werden ROM-Aufrufe richtig ausgeführt (d.h. bei Adressen unterhalb von &4000 und oberhalb &BFFF kommen Sie automatisch ins ROM) und Parameter können direkt in die CPU-Register übergeben werden. Dabei werden auf Wunsch die Register(-paare) A, HL, DE und BC mit den Konstanten oder Variablen geladen.

DOKE: (Doppel-pOKE) Lädt zwei aufeinanderfolgende Speicherstellen mit einem 16-Bit-Wert (Umkehrung zur Funktion DEEK).

Zu den FOR-Schleifen bleibt noch etwas zu sagen. Bei ihnen wird im Gegensatz zum Schneider-Basic der Endwert und die Schrittweite (soweit angegeben) immer aktuell eingesetzt. Wird also bei einer aufwärts laufenden Schleife der Endwert im Schleifenbereich irgendwann einmal unter oder auf den aktuellen Wert der Laufvariablen gesetzt, so wird die Schleife nach Erreichen des NEXT sofort abgebrochen. Auch können Konstruktionen wie 'FOR i%=1 TO 100 STEP i%' benutzt werden, im Gegensatz zum Schneider-Basic nimmt die Laufvariable dann die Werte 1, 2, 4, 8, 16, 32, 64 an und hat nach dem Verlassen der der Schleife den Wert 128. Zur Geschwindigkeitssteigerung ist zu sagen, daß FOR-Schleifen i.a. langsamer als REPEAT- und WHILE-Schleifen sind. Das liegt daran, daß bei jedem Durchlauf das Vorzeichen der Schrittweite abgefragt werden muß, um die Schleife in allen Richtungen korrekt zu beenden.

Eine Warnung: Seien Sie vorsichtig mit STEP 0, Sie bauen dadurch Endlos-

schleifen!

Programme können (und sollten) in der Testphase mit der Break-Option compiliert werden. So werden an allen Stellen, die den Quellcode-Zeilenanfängen entsprechen, Abfragen der ESC-Taste erzeugt, mit der Sie Endlosprogramme anhalten können. Auch wird hier eine Überprüfung des System-Stackpointers durchgeführt und bei Stacküberlauf das Programm mit einer Meldung abgebrochen (dies passiert z.B. bei einer Rekursion ohne Boden, wie ´100 GOSUB 100´). Ohne die Break-Option kann ein solcher Fehler nicht entdeckt werden: ein Absturz ist die Folge.

Zu einer weiteren Fehlermeldung mit Programmabbruch kann nur eine Indexüberschreitung (s.o.) führen. Alle anderen Fehler, wie z.B. Division durch Null, werden nicht gemeldet und führen auch nicht zum Programmabbruch. Allerdings sind die numerischen Ergebnisse solcher Operationen undefiniert oder entsprechen dem Rechner-Unendlich. Sie können also nach Herzenslust die Logarithmen und Wurzeln von negativen Zahlen berechnen.

Sollten Sie einmal ein Programm schreiben, das sehr viele Variablen benötigt, die dann mit dem Compiler nicht mehr in den Rechner passen, so ist auch dafür eine Lösung vorgesehen. Normalerweise liegt direkt hinter HIMEM der Variablenbereich des compilierten Programms und dahinter der eigentliche Objektcode. Sie können aber auf Wunsch den Variablenbereich vom Programm trennen und ihn irgendwo in den Speicher legen; dann wird der Objektcode direkt hinter HIMEM abgelegt. Da beim Compilieren im geplanten Variablenbereich des Programms nichts passiert, kann dieser Bereich auch im Bereich des Compilers liegen. Sie sollten aber Quell- und Objektprogramm vor dem Start aufzeichnen, da bei einem schreibenden Zugriff vom Programm auf die Variablen (bei Zuweisungen an die Variablen) der Bereich verändert wird. Geschieht dies im Compiler, kann dieser natürlich nicht mehr laufen. Wenn Sie aber, wie im Normalfalle, ein Programm entwickelt haben und dann ohne Compiler laufen lassen, haben Sie als Variablenspeicher den gesamten freien Speicher (abzüglich des Speichers, den das Programm braucht) zur Verfügung. So können auch Programme laufen, die 40k und mehr an Speicherplatz benötigen.

Wenn Sie übrigens ein (mit dem Menuepunkt ´O´) abgespeichertes Objektprogramm mit ´RUN"´ vom Band laden und (automatisch) starten, stürzt der Rechner nach Beendigung des Programms unweigerlich ab. Sie sollten also Ihre Programme mit einer Warteschleife, einem ´a$=WAIT$´ oder ähnlichem am Ende versehen. Wenn Sie ein Objektprogramm mit ´LOAD´ laden und sich die nach der Compilierung angezeigte Startadresse (die NICHT mit der ersten Speicheradresse des Programms übereinstimmt) gemerkt haben, können Sie es mit ´CALL (Startadresse)´ aufrufen, der Rechner stürzt dann nicht ab. So kann natürlich auch ein

Basic-Programm ein compiliertes Programm aufrufen, genau wie ein aus DATA-Statements 'gePOKEtes' oder assembliertes Maschinenprogramm.

Noch ein letzter Hinweis:
Der vom Compiler erzeugte Code ist NICHT RELOKATIBEL. Versuchen Sie also nicht, ein Objektprogramm an eine andere Speicherstelle zu verschieben.

Laden und Start des Compilers:

Zum Laden legen Sie die Programmcassette in den Datacorder ein und drücken 'CTRL' und die kleine 'ENTER'-Taste im Ziffernblock. Danach befolgen Sie bitte die Anweisungen des Computers. Vergessen Sie bitte nicht, die Stoptaste nach Beendigung des Ladevorganges zu drücken; der Druck der Antriebswelle deformiert mit der Zeit die Gummirolle und das Magnetband.

Das Programm startet automatisch und stoppt nach kurzer Zeit, nachdem einige Vorarbeiten erledigt sind. Sie sind dann im einem Zustand, von wo der Compiler endgültig mit 'goto 100' oder der Taste 9 im Ziffernblock gestartet wird. Nach einer Unterbrechung oder Beendigung (mit dem Menuepunkt 'E') kann der Compiler immer wieder gestartet werden; dabei geht aber der Text (das Quellprogramm verloren). Deshalb können Sie (außer beim ersten Start) auch mit 'goto 1000' oder der Taste 8 im Ziffernblock weitermachen, dann bleibt der Text erhalten.

Vor dem Start oder nach Verlassen des Compilers befinden Sie sich in einem etwas gekürzten Basic. es kennt nur noch die Befehle 'goto' und 'gosub', die, wenn Sie nicht die definierten Tasten benutzen, auch unbedingt klein geschrieben werden müssen. Grossschrift oder irgendwelche andere Befehle bringen den Rechner sofort zum Absturz.

TEIL 3

DER EDITOR MODUS

Nach dem Starten des Programms befinden Sie sich im Editor-Modus.
Die dort zur Verfügung stehenden Kommandos bestehen aus einem Buchstaben und evtl. ein oder zwei Parametern.
Werden numerische Parameter für Zeilennummern in leerer Form (z.B. L ,120) eingegeben, so erhalten sie ihren zugehörigen Grenzwert. In unserem Beispiel bedeutet dies, daß als erster Parameter der Wert der niedrigsten Zeilennummer des Programms angenommen wird.
In den folgenden Kommandobeschreibungen werden die von dem jeweiligen Kommando genutzten Parameter angegeben.
Steht zu Beginn der Kommandozeile eine Zahl, so wird diese als Zeilennummer erkannt. Die folgenden Zeichen werden in den Programmspeicher übernommen. Folgen auf eine Zeilennummer keine Zeichen, so wird, falls unter dieser Zeilennummer bereits eine Zeile eingetragen wurde, diese gelöscht.
Wird eine bereits vorhandene Zeile eingegeben, so wird die alte Zeile überschrieben.

Das C-Kommando

Parameter: keine
Bedeutung: Schaltet in den Compile-Modus um.

Das S-Kommando

Parameter: Filename
Bedeutung: Der Programmtext wird unter dem Namen Filename auf Cassette abgespeichert.
Beispiel : S test
Hierbei wird das gesamte Programm unter dem Namen "test" abgespeichert. Vor der Anwendung dieses Kommandos müssen die Tasten REC und PLAY gedrückt werden, da der Kassettenmotor automatisch startet. Es werden keine Kassettenmeldungen ausgegeben.

Das G-Kommando

Parameter: Filename
Bedeutung: Der Programmtext mit dem angegebenen Namen wird von Cassette in den Speicher geladen.
Wird kein Name angegeben, so wird das nächste auf Cassette gefundene Programm geladen.
Es werden keine Kassettenmeldungen ausgegeben.

Das O-Kommando

Parameter: Filename
Bedeutung: Der Objectcode des zuletzt compilierten Programms wird unter dem Filenamen auf Cassette abgespeichert. So kann das Program ohne Compiler mit RUN" wieder geladen und gestartet werden.
Beispiel : O otto
Der Objectcode wird unter dem Namen otto.obj abgespeichert. Vor der Anwendung dieses Kommandos müssen die Tasten REC und PLAY gedrückt werden, da der Kassettenmotor automatisch

startet. Es werden keine Kassettenmeldungen ausgegeben.

Das P-Kommando
Parameter: <Anfang>,<Ende>
Bedeutung: Der Programmtext von Zeile "Anfang" bis "Ende" wird ausgedruckt.
Vorsicht ! Wenn kein Drucker angeschlossen ist wartet der Computer auf ein Signal des Druckers. Der Editor kann dann keine Kommandos mehr entgegennehmen.
Beispiel : P 50,160
Die Zeilen 50-160 werden ausgedruckt.

Das R-Kommando
Parameter: keine
Bedeutung: Das zuletzt compilierte Programm wird ausgeführt.

Das M-Kommando
Parameter: keine
Bedeutung: Es wird von MODE 2 nach MODE 1 umgeschaltet oder umgekehrt.

Das W-Kommando
Parameter: keine
Bedeutung: Es wird von SPEED WRITE 0 nach SPEED WRITE 1 umgeschaltet oder umgekehrt.

Das D-Kommando
Parameter: <Anfang>,<Ende>
Bedeutung: Die Zeilen von "Anfang" bis "Ende" werden gelöscht. Werden keine Parameter angegeben, so wird dieses Kommando ignoriert.
Beispiel : D ,
Das gesamte Programm wird gelöscht.

Das L-Kommando
Parameter: <Anfang>,<Ende>
Bedeutung: Die Zeilen von "Anfang" bis "Ende" werden gelistet. Werden keine Parameter angegeben, so wird das gesamte Programm angezeigt.

Das H-Kommando
Parameter: keine
Bedeutung: Die zur Verfügung stehenden Kommandos werden angezeigt.

Das I-Kommando
Parameter: keine
Bedeutung: Es werden folgende Informationen angezeigt:
- HIMEM
- dimensionierte Größe des Textfeldes
- belegte Größe des Textfeldes
- der zur Verfügung stehende freie Speicherplatz FRE("")
- Cassettengeschwindigkeit

Das E-Kommando

Parameter: keine
Bedeutung: Damit wird das Programm (der Compiler) beendet. Dadurch geht Ihre Textdatei verloren; Sie sollten sie also vorher auf Band sichern.

Das Y-Kommando

Parameter: keine
Bedeutung: Es wird zwischen den Farben für Grün- und Farbmonitor umgeschaltet.

Das X-Kommando

Parameter: <Basicspeicherobergrenze>
Bedeutung: HIMEM wird neu vereinbart, wenn dies möglich ist.

Das T-Kommando

Parameter: <Textfilegröße>
Bedeutung: Das Textfeld wird neu dimensioniert (maximal bis 1000 Zeilen).

A: DIR

Parameter: keine

Zeigt die auf der Diskette enthaltenen Files an. Dies funktioniert nur, wenn ein Laufwerk angeschlossen ist; anderenfalls hat die Funktion keine Wirkung.

V: CHANGE DEVICE

Parameter: keine

Schaltet von Diskette auf Cassette um und umgekehrt. Falls Sie über kein Diskettenlaufwerk verfügen, hat diese Funktion keine Wirkung.

COMPILIEREN VON VORHANDENEN PROGRAMMEN

Die Aufzeichnung eines Quellprogramms erfolgt rein seriell und ohne Zusatzinformationen (wie Länge usw.) als normales Textfile. Deshalb kann ein im ASCII-Format (mit ´SAVE "name",A´) aufgezeichnetes Schneider-Basic-Programm geladen und, falls es keine Fehler oder dem Compiler unbekannte Sprachelemente enthält, compiliert werden.

Auch der umgekehrte Weg ist möglich: Sie können mit dem Editor des Compilers erstellte Programme aufzeichnen, als normale Basic-Programme in den Rechner laden und mit dem Interpreter laufen lassen, wieder vorausgesetzt, daß Sie keine dem Schneider-Interpreter unbekannte Sprachelemente benutzt haben.

Sogar das Erstellen von Quellprogrammen mit einem Textverarbeitungsprogramm ist möglich, falls dieses zur Aufzeichnung der Texte das normale Format für ein Textfile benutzt. Bedenken Sie aber, daß jede Zeile mit einer Zeilennummer beginnen muß.

TEIL 4

DER COMPILE MODUS

Durch das C-Kommando des Editors gelangen Sie in den Compile-Modus. Dort stehen Ihnen 9 Funktionen zur Verfügung:

0) Anzeigen des Status der Break- und Indexcheckoptionen.
1) Überprüfung des Programms auf syntaktische Korrektheit und Berechnung des benötigten Speichers für die benutzten Variablen. Dieser Punkt ist zum Testen des Programms gedacht, um Ihnen das Warten auf den Compilevorgang zu ersparen. Wenn Sie dann den Compiler endgültig starten, werden eventuelle Fehler natürlich auch gemeldet.
2) Festlegung des Speicherbereichs für die im Programm benutzten Variablen. Sie können die Startadresse der Variablen beliebig heruntersetzen. Dabei kann auch der Compiler überschrieben werden. Da dies jedoch erst nach dem Start des compilierten Programms geschieht, bleibt dieses lauffähig. Sie sollten jedoch nicht versuchen, den Compiler danach wieder zu starten.
3) Festlegen des Variablenspeichers hinter HIMEM.
4) Durch Anwählen dieses Menuepunktes wird eine regelmäßige Abfrage der ESC-Taste zum Stoppen des Programms erzeugt. Dadurch wird das Programm zwar langsamer, jedoch ist dies eine gute Testhilfe, um sich gegen Endlosschleifen zu schützen. Außerdem wird regelmäßig der Stackpointer überprüft, um einen Überlauf durch eine zu tiefe Verschachtelung von Unterprogrammen zu vermeiden. Sie sollten fertige Programme aber ohne diese Option compilieren, um sie schneller zu machen.
5) Verhindern der Abfrage der ESC-Taste (voreingestellt).
6) Vereinbaren des Indexchecks während der Laufzeit (voreingestellt).
7) Abschalten des Indexchecks für Feldvariablen.
   Wenn Sie den Indexcheck unterdrücken, wird während des Programmlaufs nicht überprüft, ob bei einer Zuweisung mit indizierten Variablen ein falscher Index angegeben ist. Dies kann bei einem dennoch auftretenden Fehler schlimmstenfalls zu einem "Absturz" des Rechners führen.
8) Compilieren des Programmes.
9) Rückkehr in den Edit-Modus.

TEIL 5

HANDHABUNG UND SYNTAX

Wir gehen davon aus, dass der Benutzer dieses Compilers das Basic des Schneiders kennt und mit der Syntax und Semantik allgemein gebraeuchlicher Basic-Befehle vertraut ist. Dieses Handbuch beschraenkt sich daher auf die Erklaerung der Besonderheiten der Befehle sowie der Benutzung des Editors von FTB-Basic.

Es ist besonders darauf zu achten, dass FTB-Basic streng zwischen den drei Variablentypen Integer, Real und String unterscheidet. Die Typkennzeichen hinter den Variablennamen sind unbedingt erforderlich, und Zuweisungen zwischen Variablen verschiedenen Typs sind nicht zulaessig. Realkonstanten muessen stets einen Nachkommaanteil haben. Darauf ist besonders bei dem Befehl INPUT zu achten, da dieser sonst klaglos falsche Ergebnisse liefert.

FTB-Basic laesst nur einen Befehl pro Zeile zu. Ausdruecke mit mehr als zwei Operanden sind ebenfalls nicht zulaessig.

Es besteht die Moeglichkeit, die Abfrage der BREAK-Taste im Programm zu unterdruecken; dadurch werden die Programme noch schneller. Jedoch kann dann das Programm in einer Endlosschleife nur noch durch ein Zuruecksetzen des Rechners gestoppt werden und geht dadurch verloren.

Ebenso kann der Indexcheck bei indizierten Variablen abgeschaltet werden. Sollte jedoch dann eine Indexueberschreitung stattfinden, so ueberschreibt das Programm andere Variablen oder sich selbst und kann dadurch "abstuerzen".

Variablennamen muessen mit einem Buchstaben beginnen und duerfen maximal acht Zeichen lang sein. Feldvariablen koennen nur eindimensional vereinbart werden.

In den Syntaxerlaeuterungen fuer Funktionen werden die Argumente mit ihren Typkennzeichen kenntlich gemacht. Die Benutzung anderer Typen fuehrt zu Fehlermeldungen. Zum Beispiel ist a%=sin(x!) unzulaessig, da diese Funktion einen reellen Wert liefert. Dies wird in der Syntaxerlaeuterung wie folgt dargestellt: SIN(!) --> !

Angaben in eckigen Klammern [] sind wahlfrei, d.h. sie koennen angegeben oder weggelassen werden. Die Klammern duerfen nicht mit eingegeben werden.

Ein Sternchen hinter einer wahlfreien Angabe bedeutet, dass die Angabe mehrfach vorkommen kann.

Angaben in spitzen Klammern <> sind Platzhalter und duerfen nicht eingegeben werden.

Eine Ziffernreihe, die aus mindestens einer Ziffer bestehen muss, wird zum Beispiel so dargestellt:

```
<Ziffer>[<Ziffer>]*
```

Dieser Ausdruck koennte folgendermassen aussehen:

```
43
oder 443
oder 876543 etc.
```

2. OPERATOREN:
   Real   : +,-,*,/,^

   Integer: +,-,*,\,MOD

   String : +
     Ein negatives Vorzeichen vor Strings fuehrt zu deren Umkehrung.

3. LOGISCHE OPERATOREN:
   =, <>, <, >, >=,<=

4. BOOLESCHE WERTE:
   TRUE, FALSE
     Diese duerfen als logischer Ausdruck verwendet werden. So ist zum Beispiel das folgende Programm zulaessig:

```
10 i%=0
20 repeat
30   if i%=4711 then 60
40   i%=i%+1
50 until false
60 print"Ende"
```

5. KONSTANTEN:
     Stringkonstanten koennen eine maximale Laenge von 80 Zeichen annehmen.

     Die Realkonstanten besitzen die gleiche Darstellung wie im Schneider-Basic.

     Integerkonstanten koennen Werte im Bereich von -32768 bis 65535 annehmen. Jedoch werden alle Zahlen, die groesser sind als 32767, in ihr negatives Komplement umgewandelt.

6. VARIABLEN
     Variablennamen duerfen 8 Zeichen lang sein und muessen mit einem Typkennzeichen enden (#,!,%). Der Unterstreichstrich im Namen ist erlaubt. Gross- und Kleinbuchstaben werden nicht unterschieden.

     Stringvariablen koennen maximal 80 Zeichen aufnehmen. Bei Stringadditionen werden ueberfluessige Zeichen ignoriert.

     Die Realvariablen besitzen die gleiche Darstellung wie im Schneider-Basic.

     Integervariablen siehe Integerkonstanten.

7. SCHLUESSELWOERTER

Wenn keine Erklaerungen gegeben sind, so gelten die im Schneider-handbuch abgedruckten bzw. die allgemein gueltigen. Ein Pfeil hinter der Funktion deutet auf den Typ, den diese Funktion liefert. Darauf ist bei Zuweisungen zu achten.

ABS(!) --> !
Liefert den Absolutwert des Realoperanden.

ATN(!) --> !
Liefert den Arcustangens des Operanden.

ASC($) --> %
Liefert den ASCII-Code des ersten Zeichens des Stringoperanden.

BORDER %

CALL %[,%[,%[,%[,%]]]]
Startet ein Maschinenprogramm ab der durch den ersten Wert festgesetzten Adresse. Es werden immer Adressen im ROM aufgerufen, es sei denn, es werden reine RAM-Adressen angegeben.
Dabei werden die auf den Aufruf folgenden Werte in folgender Reihenfolge in die Register uebergeben: A HL DE BC
Beispiel: CALL 48442,0,44196 ruft den betriebssysteminternen Editor auf und uebergibt im HL-Register die Startadresse des zu edierenden Textes.
Hierbei wird eine RAM-Adresse aufgerufen.

CHR$(%) --> $
Liefert das zugehoerige Zeichen im ASCII-Code

CINT(!) --> %

CLG

CLS

COS(!) --> !

CREAL(%) --> !

DEEK(%) --> %
Liefert den 16-Bit Wert der durch den Operanden bestimmten Speicherzellen.

DEG

DIM <Variablenname>(<positive Integerkonstante>)
Es duerfen nur eindimensionale Variable vereinbart werden.

```
DOKE <Adresse>,<Integervariable oder -konstante>

      Schreibt den durch den zweiten Operanden festgelegten 16-Bit Wert in die
      durch den ersten Operanden bestimmten Speicherzellen.

DRAW %,%

END

EXP(!) --> !

FOR <Zuweisung> TO <Variable> [ STEP <Variable> ]
      Beginn einer FOR-NEXT Schleife. Als Schleifenvariablen duerfen nur
      Integervariablen verwendet werden.

GOTO <Zeilennummer>

GOSUB <Zeilennummer>

GRAPHPAPER %
      Setzt die Hintergrundfarbe fuer Grafik fest

GRAPHPEN %
      Waehlt den Schreibstift fuer Grafik

IABS(%) --> %
      Liefert den Absolutwert des Operanden

IF <logischer Ausdruck> THEN <Zeilennummer>

INK %,%

INKEY$ --> $

INPUT <Variable>
      Die angegebene Variable wird von der Tastatur eingelesen.
      Feldvariablen koennen nicht mit INPUT eingelesen werden. Verwenden Sie
      zu diesem Zweck eine Hilfsvariable.

ISGN(%) --> %
      SGN-Funktion fuer Integerzahlen

JOY(%) --> %

LEFT$($,%) --> $

LEN($) --> %
```

```
LET <Zuweisung>
      LET kann weggelassen werden.

LOCATE %,%

LOG(!) --> !

LOG10(!) --> !

LPRINT [Operand[;]]
      Leitet Ausgabe an Drucker. Siehe PRINT.

MID$($,%,%) --> $
      Zuweisungen an diese Funktion sind nicht moeglich.

MODE %

MOVE %,%

NEXT [<Variable>]

ORIGIN %,%

PEEK(%) --> %

PLOT %,%

POKE %,%

PRINT [Operand[;]]
      Feldvariablenelemente koennen nicht ausgegeben werden. Zu diesem Zweck
      muss man eine Hilfsvariable benutzen.
      PRINT kann nicht durch ein Fragezeichen ersetzt werden.

RAD

REM <Text>
      REM kann nicht durch ein Hochkomma ' ersetzt werden.

REPEAT
      Startmarkierung fuer eine REPEAT - UNTIL Schleife

RETURN

RIGHT$($,%) --> $

SGN(!) --> !

SIN(!) --> !

STEP <Integervariable oder -konstante>

SQR(!) --> !
```

TAN(!) --> !

TEST(%,%) --> %

TEXTPAPER %
Setzt den Schreibstift fuer den Texthintergrund

TEXTPEN %
Setzt den Schreibstift fuer Text

THEN <Zeilennummer>
Wenn der logische Ausdruck nach IF wahr ergibt, wird zu der nach THEN stehenden Zeilennummer verzweigt.

TO <Integervariable oder -konstante>

UNTIL <logischer Ausdruck>
Wenn der logische Ausdruck nach UNTIL falsch ergibt, wird zu dem zugehoerigen REPEAT verzweigt. REPEAT - UNTIL Schleifen werden ebenso geschachtelt wie WHILE - WEND oder FOR - NEXT.

WAIT$ --> $
Liefert wie INKEY$ ein Zeichen von der Tastatur, wartet jedoch auf eine Eingabe. Dadurch kann man die Zeilen

```
10 a$=inkey$
20 if a$="" then 10
```

ersetzen durch

```
10 a$=wait$
```

WHILE

WEND

Syntax der Sprache FTB-Basic

Die Syntax ist hier in etwas modifizierter BNF (Backus-Naur-Form) beschrieben. Dabei werden sogenannte Metasymbole benutzt, Symbole, die nicht zur eigentlichen Sprache gehoeren und dazu dienen, andere Symbole zu beschreiben:

[ ] : Eckige Klammern schliessen Sprachelemente ein, die weggelassen werden koennen

{ } : Geschweifte Klammern schliessen Sprachelemente ein, die wiederholt oder weggelassen werden koennen (ist hinter der schliessenden Klammer eine Zahl angegeben, so gibt sie die maximale Anzahl der Wiederholungen an)

< > : Spitze Klammern schliessen Sprachelemente ein, die an anderer Stelle erklaert werden

: Alternative (eins von mehreren durch den Strich getrennten Sprachelementen kann ausgewaehlt werden)

::= : Definitionszeichen; das links vom Definitionszeichen stehende Sprachelement (Definiendum) wird durch die rechts davon stehende Symbol- oder Sprachelementfolge (Definiens) beschrieben

.. : Bereichsangabe: Ein aus dem Bereich stammendes Symbol oder Element (z.B. 3..8: 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8)

```
Programm ::= { <Zeile> }

Zeile ::= <positive Integerkonstante> <Anweisung>

positive Integerkonstante ::= 1..65535

Anweisung ::= CLS ¦
              CLG ¦
              DEG ¦
              RAD ¦
              RETURN ¦
```

```
Variable ::= <einfache Variable> |
             <Feldelement>

einfache Variable ::= <Name> <Typkennzeichen>

Name ::= <Buchstabe> { <Buchstabe> | <Ziffer> | <Unterstrich> }7

Typkennzeichen ::= % |
                   $ |
                   !

Zeichen ::= <Buchstabe> |
            <Ziffer> |
            <Unterstrich> |
            sonstiges darstellbares Zeichen ohne Anfuehrungszeichen

Buchstabe ::= a..z |
              A..Z

Ziffer ::= 0..9

Unterstrich ::= _

Feldelement ::= <Name> <Typkennzeichen> ( <Integeroperand> )

Operand ::= <einfache Variable> |
            <Konstante>

Konstante ::= <Realkonstante> |
              <Stringkonstante> |
              <Integerkonstante>

Realkonstante ::= [ - | + ] <positive Integerkonstante>
                  [ . <positive Integerkonstante> ]
                  [ [ - | + ] E <positive Integerkonstante> ]

Stringkonstante ::= " { <Zeichen> }80 "

Integerkonstante ::= -32768..65535
```

```
                END |
                REM <Text> |
                GOTO <positive Integerkonstante> |
                GOSUB <positive Integerkonstante> |
                REPEAT |
                UNTIL <logischer Ausdruck> |
                WHILE <logischer Ausdruck> |
                WEND |
                FOR <einfache Integervariable> = <Integeroperand> TO
                  <Integeroperand> [ STEP <Integeroperand> ] |
                NEXT [ <einfache Integervariable> ]
                IF <logischer Ausdruck> THEN <positive Integerkonstante> |
                [ LET ] <Variable> =
                  <Funktionsname> [ ( <Operand> { , <Operand> } ) ] |
                  <Operand> |
                  <Operand> <Operator> <Operand> |
                  - <Operand> |
                  <Feldelement> |
                PRINT [ <Operand> [ ; ] ] |
                LPRINT [ <Operand> [ ; ] ] |
                INPUT <einfache Variable> |
                CALL <positive Integerkonstante> { , <Integeroperand> }4 |
                LOCATE <Integeroperand> , <Integeroperand> |
                INK <Integeroperand> , <Integeroperand> |
                POKE <Integeroperand> , <Integeroperand> |
                DOKE <Integeroperand> , <Integeroperand> |
                PLOT <Integeroperand> , <Integeroperand> |
                MOVE <Integeroperand> , <Integeroperand> |
                DRAW <Integeroperand> , <Integeroperand> |
                BORDER <Integeroperand> |
                TEXTPEN <Integeroperand> |
                TEXTPAPER <Integeroperand> |
                GRAPHPEN <Integeroperand> |
                GRAPHPAPER <Integeroperand> |
                DIM <einfache Variable> ( <positive Integerkonstante> )

Text ::= { <Zeichen> | " }

logischer Ausdruck ::= <Operand> < | <= | = | >= | > | <> <Operand> |
                       TRUE |
                       FALSE
```

```
Operator ::= + | - | * | / | \ | MOD | ^

einfache Integervariable ::= <Name> %

Integeroperand ::= <einfache Integervariable> | <Integerkonstante>

Funktionsname ::= RIGHT$ |
                  LEFT$ |
                  MID$ |
                  CHR$ |
                  INKEY$ |
                  WAIT$ |
                  SIN |
                  COS |
                  TAN |
                  ATN |
                  EXP |
                  SQR |
                  LOG |
                  LOG10 |
                  ABS |
                  SGN |
                  CREAL |
                  IABS |
                  ISGN |
                  CINT |
                  LEN |
                  ASC |
                  PEEK |
                  DEEK |
                  TEST |
                  JOY
```