BASIC 64

Der BASIC-Compiler für den COMMODORE 64

EIN DATA BECKER PROGRAMM

Wichtiger Hinweis

Das vorliegende Handbuch und das dazugehörige Programm wurden vom Autor mit größter Sorgfalt erarbeitet und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf Programmfehler oder fehlerhafte Angaben im Handbuch zurückgehen, übernommen werden kann. Für die schriftliche, Mitteilung eventueller Fehler sind wir jederzeit dankbar.

DATA BECKER
Merowinger Str. 30
4000 Düsseldorf

ISBN-Nr.: 3-89011-534-9

DATA BECKER INFORMIERT :
----

1)LADESCHWIERIGKEITEN

Bei der Vervielfältigung unserer Software verwenden wir ausschließlich qualitativ hochwertige Disketten, die während des Dupliziervorgangs auf 100%ige Lesefähigkeit geprüft werden. Schon bei der kleinsten Unstimmigkeit weist unser Dupliziercomputer den Datenträger zurück. Eine derart überprüfte Programmdiskette müßte von jedem 1541-Laufwerk von Commodore problemlos gelesen werden können. Haben Sie trotz alledem Leseschwierigkeiten bei DATA BECKER Software, so ist der Fehler in den meisten Fällen bei Ihrer Diskettenstation zu suchen. Eine Floppy kann schon nach wenigen Bestriebsstunden "dejustiert" sein, d.h. der Schreib-Lesekopf wird nicht mehr 100% richtig positioniert, Bei herkömmlicher Software (z.B. eigenen Programmen) tritt kein Lesefehler auf, da diese Programme ja auch mit einem schlecht justierten Laufwerk gespeichert wurden. Unsere Programmdisketten stellen jedoch höhere Ansprüche an die Justage der Diskettenstation.
Überprüfen Sie deshalb bitte, ob diese Diskette sich bei einem Freund oder Ihrem DATA BECKER Händler in den C-64 laden läßt. Ist dies der Fall, so müßten Sie Ihre Floppy justieren lassen.
Ergeben sich auf anderen Floppys dieselben Probleme, so kann es sein, daß die Diskette z.B. auf dem Postwege (durch magnetische Aussonderungsverfahren) Schaden genommen hat. In diesem Fall wird Ihre Diskette natürlich problem- und kostenlos umgetauscht.

2)UMTAUSCH DEFEKTE DISKETTEN

Sollte aus irgendeinem anderen Grund Ihre Diskette beschädigt oder gar zerstört werden (z.B. mechanische Schäden durch Anwenderverschulden), so senden Sie uns bitte Ihre Originaldiskette mit einem Verrechnungsscheck über DM 10.- zurück. Sie erhalten dann umgehend eine neue Programmdiskette.

3)KOPIERVERSUCHE KÖNNEN DIE DISKETTE ZERSTÖREN

Unsere Programmdisketten sind ausnahmslos gegen Kopieren geschützt Natürlich ist kein Kopierschutz absolut sicher, aber wir werden mit allen uns zur Verfügung stehenden rechtlichen Mitteln versuchen, den Schwarzkopierern das Handwerk zu legen.

Inhaltsverzeichnis

Vorwort

BASIC 64 ist ein optimierender Basic-Compiler für den Commodore 64, der Ihre in Basic geschriebenen Programme schneller und leistungsfähiger macht. Dabei können Sie wählen, ob BASIC 64 Ihre Programme in einen P-Code oder direkt in Maschinensprache übersetzen soll. Das übersetzte Programm wird dadurch bis zu 6mal schneller als das Original-Programm.

In vielen Fällen, besonders beim Einsatz von Integer-Variablen, wird durch den in BASIC 64 enthaltenen Programmoptimierer Ihr Basic-Programm sogar 6-10mal schneller als das Original. Dies gilt ebenfalls für die Verarbeitung von Zeichenketten, außerdem ist die gefürchtete Garbage Collection bei compilierten Programmen wesentlich schneller als die des Interpreters (max. 1 Sekunde statt mehrere Minuten), was die Anwendung von Basic für anspruchsvollere Programme erst möglich macht.

Obwohl zu jedem compilierten Basic-Programm noch ein Runtimemodul mit einer Länge von 5K hinzukommt, werden besonders größere Basic-Programme durch das Compilieren kürzer (um ca. 25%). Außerdem stehen dem compilierten Programm 62K für Programm und Daten zur Verfügung und nicht nur 38K, wie bei Verwendung des Basic-Interpreters.

BASIC 64 kann aus Ihren Basic-Programmen aber auch direkt Maschinenspracheprogramme erzeugen, welche zwar ca. doppelt so lang sind wie das Original, aber nochmals um einiges schneller als ein P-Code-Programm. Mit BASIC 64 compilierte Programme werden dadurch in vielen Fällen 10-20fach schneller als ein Basic-Programm. Ein Mischen von kurzem P-Code und extrem schneller Maschinensprache ist ebenfalls möglich.

Selbstverständlich compiliert BASIC 64 Programme beliebiger Größe und auch Overlay-Pakete, wobei die Compiliergeschwindigkeit ca. 1K Byte pro Minute beträgt. Ebenfalls unterstützt werden die meisten Befehle folgender Basic-Erweiterungen: Supergrafik. 64, Simons' Basic, Exbasic Level II, Basic 4.0 (enthalten in Diskomat und Master 64) und andere.

BASIC 64 bietet Ihnen viele weitere Möglichkeiten, wie z.B.: Zwei Optimierungsstufen, variabler Codestart, variable Speicherbenutzung, Umdefinieren von Datentypen von Variablen, Berechnung von konstanten Ausdrücken und Zeichenketten während der Compilierung, Optimieren und Umstellen von Formeln, Syntaxprüfung von Programmen, Erstellen einer Zeilenadressenliste, Start von Unterprogrammen bei Laufzeitfehlern, IF .. THEN .. ELSE, Datenschnittstelle zum Assembler Profi-Ass, etc.

BASIC 64 ist kompatibel zum eingebauten Basic-Interpreter des Commodore 64 und bildet zusammen mit diesem ein ideales Programmentwicklungssystem, mit dem schnelle und leistungsfähige Programme geschrieben werden können, für die bisher das Programmieren in Maschinensprache nötig war.

1. Kapitel

1.1 Bedienung von BASIC 64

Die Bedienung des BASIC 64 Compilers ist denkbar einfach:

- Speichern Sie Ihr Basic-Programm auf einer Diskette ab. Diese muß nicht unbedingt leer sein, sollte aber genügend freie Speicherkapazität haben (bei größeren Programmen bis zu 300 Diskettenblöcke). Sie brauchen sich von Ihrem Programm nicht unbedingt eine Sicherheitskopie erstellen, da BASIC 64 das Programm nur einliest und nicht löscht.

- Legen Sie nun die Diskette mit BASIC 64 in das Laufwerk, laden Sie den Compiler mit LDAD"BASIC 64",8 und starten Sie ihn mit RUN.

- Nachdem der Compiler sich mit "BASIC 64 Compiler" und der Versionsnummer gemeldet hat, erscheinen auf dem Bildschirm die Auswahlmöglichkeiten des Compilers. Nehmen Sie nun die BASIC 64-Diskette aus dem Laufwerk und legen Sie die Diskette mit dem zu compilierenden Programm ein.

- Drücken Sie die "Return"-Taste. Sie haben dadurch die Möglichkeit Nr. 1 ausgewählt, was Sie auch mit Hilfe der Taste "1" erreicht hätten.

- Der Compiler fragt Sie nun nach dem Namen Ihres Programms, geben Sie diesen ein und drücken Sie "Return".

- Der Compiler übersetzt Ihr Programm nun in ein P-Code Programm. Auf dem Bildschirm wird die gerade bearbeitete Zeilennummer angezeigt, danach gibt der Compiler einige Angaben über das generierte Programm aus. Sind in Ihrem Programm Fehler enthalten, so meldet der Compiler diese mit den entsprechenden Fehlermeldungen. Der Compiler bricht dabei nicht ab, sondern setzt seine Arbeit fort, um evtl. weitere Fehler zu finden.

- Nachdem der Compiler seine Arbeit beendet hat, meldet er sich mit "READY". Drücken Sie nun die Taste "N", um dem Compiler mitzuteilen, daß Sie keine weiteren Programme compilieren wollen. Durch Drücken einer anderen Taste können Sie den Compiler neu starten.

- Waren in Ihrem Programm Fehler enthalten, so müssen diese verbessert werden und anschließend das Programm nochmals compiliert werden.

- Das compilierte Programm steht nun auf der Diskette. Um es von dem Basic-Programm zu unterscheiden, hat der Compiler vor den Programmnamen ein "P-" gehängt. Laden Sie das Programm mit LOAD"P-Programmname" und starten Sie es mit RUN.

- Dieses Programm läuft nun wesentlich schneller als ein Basic-Programm.

- Evtl. noch auftretende Fehler werden genauso gemeldet, wie

bei einem Basic-Programm. Allerdings wird keine Zeilennummer, sondern die Speicherstelle, an welcher der Fehler auftrat, genannt. Mit Hilfe einer Zeilenliste können Sie die fehlerhafte Stelle im Original-Programm feststellen. Eine Zeilenliste wird vom Compiler auf Wunsch ebenfalls erzeugt.

- Sollte Ihnen die Geschwindigkeit des Programmes immer noch nicht ausreichen, so gibt es genügend Möglichkeiten, dies zu ändern (Optimierungsstufe 2, Generieren eines Maschinenprogrammen, Verwendung von Integer-Variablen, etc.). Dieses wird in den entsprechenden Kapiteln erläutert.

Achtung:

Zum vollständigen Verständnis des Compilers sollten Sie unbedingt aufmerksam das Handbuch lesen.

1.2 BASIC 64 und Floppy-Express

Soweit Sie den Data Becker Floppy-Express besitzen, koennen Sie BASIC 64 auch mit LOAD"FLADER",8 laden und mit RUN starten. Der Compiler arbeitet dann schneller.

Compilierte Programme können ebenfalls den Floppy-Express benutzen. Dieses geschieht wie folgt:

- Drücken Sie nach dem Laden des Compilers die Taste "3" und danach "E".

- Geben Sie die Zahl 52224 (entspricht $CC00) ein und drücken Sie zweimal "Return".

- Compilieren Sie das Programm, indem Sie die entsprechende Diskette einlegen, "Return" drücken und den Programmnamen eingeben.

Vor dem Laden des auf diese Art compilierien Programmes kann nun die Floppy-Express-Version 10 aktiviert werden.

2. Kapitel

2.1 Auswahlmöglichkeiten

Nachdem Sie BASIC 64 gestartet haben, erscheint auf dem Bildschirm das Hauptmenü. Die vier Wahlmöglichkeiten dieses Menüs sind entsprechend durchnumeriert und lassen sich durch Drücken der entsprechenden Zahlentaste starten. Der wichtigste Punkt dieses Menüs ist der Punkt 1, welcher das Compilieren eines Programmen startet. Aus diesem Grund kann man diesen Punkt nicht nur mit der Taste "1" anwählen, sondern auch mit der "Return"-Taste. Punkt 2 startet ebenfalls den Compiler, allerdings mit einer anderen Optimierungsstufe. Der Unterschied zwischen den beiden Optimierungsstufen wird in Kapitel 7 beschrieben. Zum Compilieren von Overlaypaketen dient der Menüpunkt 4. Dies wird in Kapitel 6 beschrieben. Durch Anwahl des Menüpunktes 3 gelangen Sie in ein weiteres Menü.

2.2 Das erweiterte Entwicklungspaket

Zur Entwickelung von Programmen mit speziellen Anwendungen verfügt BASIC 64 über die Möglichkeit, einige Eigenschaften des zu generierenden Programmes zu verändern. Wählen Sie dazu im Hauptmenü den Menüpunkt 3 an. Auf dem Bildschirm erscheinen nun alle veränderbaren Werte. Alle Menüpunkte sind in alphabetischer Reihenfolge mit Buchstaben bezeichnet. Durch Drücken der entsprechenden Buchstabentaste können Sie nun die gewünschte Einstellung erreichen. Eine genaue Beschreibung der einzelnen Punkte dieses Menüs finden Sie in Kapitel 9. Ein einfaches Beispiel für die Benutzung des Entwicklungspaketes ist das Compilieren eines Programmen mit Befehlen einer Basic-Erweiterung:

Nachdem Sie Punkt 3 im Hauptmenü ausgewählt haben, erscheint die Art der verwendeten Basic-Erweiterung unter Menüpunkt H. Nach Start des Compilers ist noch keine bestimmte Basic-Erweiterung eingestellt. Durch mehrmaliges Drücken der Taste "H" können Sie nun die von Ihnen verwendete Basic-Erweiterung einstellen und mit "Return" wieder in das Hauptmenü gelangen. Durch nochmaliges Drücken der "Return"-Taste starten Sie wie gewohnt den Compiliervorgang. Andere Änderungen der voreingestellten Werte können genauso einfach vorgenommen werden. Natürlich können auch mehrere Werte verstellt werden. Der Einfachheit halber werden die Wahlmöglichkeiten des Entwicklungspaketes in den folgenden Kapiteln nur noch mit Punkt A bis Punkt M bezeichnet. Die Wahlmöglichkeiten des Hauptmenüs dagegen mit Punkt 1 bis Punkt 4. Beachten Sie, daß nur die jeweils auf dem Bildschirm erscheinenden Punkte angewählt werden können.

2.3 Compileranweisungen

Alle Auswahlmöglichkeiten können nur vor dem eigentlichen Compilieren benutzt werden. In vielen Fällen ist es nötig, während des Compilierens Anweisungen an den Compiler zu geben. Diese Anweisungen müssen dann innerhalb des Programmas notiert werden. Damit der Basic-Interpreter diese Anweisungen überspringt, müssen sie nach einem REM-Befehl folgen. Das Symbol zur Kennzeichnung dieser Anweisungen ist der Klammeraffe "@". Das Format für eine Compileranweisung ist somit folgendes:

```
REM@ Anweisung
```

Eine Liste aller möglichen Compileranweisungen finden Sie in Kapitel 9. Die wichtigste Compileranweisung ist folgendes

```
REM@ I=Variablenname,Variablenname...
```

Diese Anweisung bewirkt, daß alle in der Anweisung genannten Variablen vom Compiler als ganzzahlige Variablen angesehen werden (Integer-Variablen). Der Anwendungsbereich dieser Compileranweisung liegt in der Entwicklung von besonders schnellen Programmen.

2.4 Ein Beispiel für eine Programmentwicklung

Ein einfaches Beispiel für ein Programm, das möglichst schnell ablaufen soll, ist eine Primzahlberechnung. Folgendes Programm wird deswegen häufig als Benchmarktest für die Geschwindigkeit von Computern, Programmiersprachen und Compilern benutzt:

```
10 REM SIEB DES ERATHOSTENES
20 DIM Z%(10000)
30 FOR I=1 TO 10000
40 Z%(I)=l
50 NEXT
60 PRINT 2
70 FOR I-1 TO 10000
90 IF Z%(I)=0 THEN 150
90 PRINT I*2+1
100 K=1
110 IF I+K*(I*2+1)>10000 THEN 150
120 Z%(I+K*(I*2+1))=0
130 K=K+l
140 GOTO110
150 NEXT
```

Würden Sie nun die Zeit messen, die dieses Programm zur Ausführung benötigt, so würden Sie hauptsächlich die Zeit messen, die zur Ausgabe der Primzahlen auf dem Bildschirm benötigt wird. Um das Programm zum Vergleich von Rechengeschwindigkeiten benutzen zu können, wird üblicherweise folgende Änderung vorgenommen:

```
90 Q=I*2+1
```

Trotz dieser Änderung benötigt das Programm noch eine ziemlich lange Rechenzeit. Das Programm wird ca. 3mal schneller, wenn Sie es wie in Kapitel 1 beschrieben compilieren. In Kapitel l wurde bereits erwähnt, daß die erreichte Geschwindigkeit sich noch weiter steigern läßt. Bei diesem Programm gibt es mehrere Möglichkeiten:

- Schreiben Sie das Programm so um, daß nur noch ganzzahlige Variablen benutzt werden. Dies geschieht durch das Anhängen eines "%"-Zeichens hinter jeden Variablennamen. Das Programm wird dadurch schneller und der P-Code sogar kürzer. Beachten Sie, daß der Basic-Interpreter keine ganzzahligen Variablen in FOR-NEXT-Schleifen zulässt, das Programm müßte also umgeschrieben werden.

- Diese Mühe können Sie sich ersparen, indem Sie folgende Compileranweisung benutzen:

```
5 REM@ I=I,K,Q
```

Besonders für die Schleifenvariable I ist dies sehr sinnvoll, da Sie dieses Programm dann auch mit dem Basic-Interpreter ausprobieren können, ohne es umzuschreiben.

- Auch die Compileranweisung ist in diesem speziellen Fall unnötig, da das Programm ausschließlich mit ganzen Zahlen rechnet. Anstatt den Compiler mit dem Menüpunkt 1 zu starten, starten Sie ihn einfach mit 2.

- Zusätzlich können Sie bei jedem Programm den Compiler anweisen, Maschinensprachencode zu erzeugen, dies geschieht durch Menüpunkt A (im Entwicklungspaket).

Nachdem Sie das Generieren von Maschinensprache gewählt und das Programm mit Menüpunkt 2 compiliert haben, läuft es ca. 15mal schneller als das Basic-Programm. Die Beachtung von einigen Regeln beim Programmieren und Compilieren lohnt sich also sehr.

3. Kapitel

Der Programmoptimierer

Beim Erstellen von Programmen, die mit maximaler Geschwindigkeit laufen sollen, ist es nützlich, wenn Sie die Wirkungsweise des Compilers kennen und zeitkritische Teile innerhalb Ihres Programmes entsprechend anpassen. Bei Programmen, bei denen dies versäumt wurde oder die nicht zum Compilieren gedacht waren, kann dies mit der Optimierungsstufe 2 nachgeholt werden. Dies wird in Kapitel 7 beschrieben. Andererseits kommt es oft vor, daß Anpassungen, die ein Programm für den Interpreter schneller machen, unnötig sind, da der Compiler sie nicht benötigt oder automatisch ausführt. Auch derartige Fälle sollte man kennen, da dadurch Arbeit gespart werden kann.

3.1 Allgemeine Wirkungsweise des Compilierens

Bei längerer Arbeit mit dem Basic-Interpreter macht man folgende Erfahrungen bezüglich der Geschwindigkeit von Programmen:

- Ein GOTO/GOSUB arbeitet langsamer, als ein RETURN, obwohl beides eigentlich einen Sprung innerhalb des Programmes bedeutet.

- Eine Schleife, die mit Hilfe einer IF-THEN-Konstruktion programmiert wird, arbeitet langsamer als eine FOR-NEXT-Schleife.

- Formeln, in denen Konstanten benutzt werden, werden langsamer berechnet, als Formeln mit Variablen, dies gilt für Zahlen und meistens auch für Zeichenketten.

- Variablen, die im Programm erst später benutzt werden, arbeiten langsamer als andere.

- Die Geschwindigkeit der Sprünge GOTO und GOSUB ist bei Vorwärtssprüngen abhängig von der Sprungweite, bei Rückwärtssprüngen von der Entfernung des Sprungzieles zum Programmanfang.

- Integervariablen werden langsamer verarbeitet als Gleitkommavariablen. Eine Speicherersparnis von Integervariablen ist nur bei Feldern vorhanden.

- Bei größeren Programmen werden die Sprünge GOTO/GOSUB und die Verarbeitung von Variablen um ein Vielfaches langsamer.

- Durch das Einfügen von Leerzeichen werden Programme zwar übersichtlicher, aber wesentlich langsamer.

- Strukturierte Programme sind meistens langsamer als Spaghetti-Programme.

Alle diese Punkte und einige weitere treten bei compilierten Programmen nicht mehr auf. Es ist somit nicht nötig, Programme an den Basic-Interpreter anzupassen, wenn diese sowieso compiliert werden sollen. Natürlich gibt es auch bei compilierten Programmen Operationen, die schneller arbeiten als andere. Es ist oft nützlich, diese zu kennen und zu benutzen. Dies führt zu der Frage, warum und wann compilierte Programme schneller arbeiten, als nichtcompilierte. Die wichtigsten Punkte sind:

- Die Arbeit des Interpretierens von Basic-Programmen wird verlagert auf das Interpretieren von einer Art Pseudo-Maschinensprache (P-Code), was wesentlich schneller ist. BASIC 64 verfügt zusätzlich über die Möglichkeit, echte Maschinensprache zu erzeugen und die Interpretation ganz zu sparen.

- Das Interpretieren von Formeln wird vom Compiler übernommen. Das compilierte Programm berechnet Formeln nach einem schnelleren Verfahren.

- Beim Verarbeiten einer Variablen muß diese nicht mehr in einer Tabelle gesucht werden, sondern es kann direkt darauf zugegriffen werden.

- Zur Auffindung des Zieles eines GOTO-GOSUB-Sprunges ist es nicht nötig, das gesamte Programm zu durchsuchen, der Sprung erfolgt direkt. Beispiel: Ein GOTO in einem Maschinenprogramm dauert 3 Mikrosekunden, bei längeren Basic-Programmen dagegen bis zu einer zehntel Sekunde.

- Konstanten müssen vom Interpreter erst in die binäre Form gebracht werden, da sie im Programm in dezimaler Form gespeichert sind, ähnliches gilt für die Zeilennummern von GOTO/ GOSUB, im compilierten Programm sind Konstanten dagegen binär gespeichert.

Hieraus ergeben sich einige wichtige Konsequenzen. Das folgende Beispiel belegt dies:

```
10 FOR I=1 TO 1000:NEXT
20 PRINT"FERTIG MIT FOR-SCHLEIFE"
30 I=1
40 I=I+1:IF I<=1000 THEN40
50 PRINT"FERTIG MIT IF-THEN-SCHLEIFE"
```

Startet man dieses Programm, so ist die erste Schleife wesentlich schneller beendet als die zweite Schleife. Nach dem Compilieren laufen beide Schleifen dagegen ungefähr gleich schnell. Die Geschwindigkeit von compilierten Programmen ist nicht abhängig von der Anzahl der verwendeten Befehle oder von der Struktur des Programmen, sondern lediglich von den ausgeführten Operationen selbst. Computerintern werden beide Schleifen nämlich mit fast der gleichen Methode ausgeführt (der Verwaltungsaufwand der FOR-NEXT-Schleife ist etwas höher, da sie flexibler einsetzbar ist).

Bei den bisher erwähnten Ergebnissen des Compilierens handelt

es sich um Methoden, die von den meisten Basic-Compilern angewandt werden. BASIC 64 benutzt dagegen weitere Verfahren, um ein Programm zu beschleunigen. Dabei handelt es sich um das Optimieren von Formeln, die Verwendung von Integer-Operationen, eine schnelle Zeichenkettenverarbeitung und das wahlweise Generieren von Maschinensprache.

3.2 Optimierungen bei Formeln

Zur Berechnung größerer Formeln entstehen Zwischenergebnisse, die der Computer auf einem Stapel (Stack) abspeichert. Der Compiler stellt Formeln um, mit dem Ziel, sowenig Zwischenergebnisse wie möglich zu erzielen. Dadurch wird nicht nur das compilierte Programm kürzer, sondern es kann auch schneller ausgeführt werden. Sie brauchen bei der Benutzung größerer Formeln somit nicht mehr darauf zu achten, daß diese möglichst schnell interpretiert werden können. Der Compiler ändert sogar teilweise Formeln, um eine schnellere Berechnung zu ermöglichen. So ist z.B. die Multiplikation zweier Gleitkommazahlen schneller als die Division. In vielen Fällen ist es möglich, Gleitkommadivision durch Multiplikation zu ersetzen (Kehrwert). Der Compiler wendet derartige Verfahren natürlich nur an, wenn dabei das Ergebnis der Formel erhalten bleibt.

Eine wesentliche Arbeitserleichterung bei der Programmierung von mathematischen Programmen ist die Fähigkeit von BASIC 64, Formeln, in denen viele Konstanten enthalten sind, bereits während des Compilierens teilweise oder vollständig zu berechnen. Das folgende Beispiel belegt dies:

```
10 S=1.414
```

Diese Zuweisung ist zwar sehr ungenau, wird vom Interpreter aber schneller bearbeitet als:

```
10 S=SQR(2)
```

Nach dem Compilieren arbeiten beide Versionen gleich schnell, die zweite Version ist aber genauer und leichter zu programmieren. Die Berechnung von konstanten Operationen führt der Compiler natürlich auch dann aus, wenn diese in größeren Formeln enthalten sind. Oft werden durch das Umstellen von Formeln auch versteckte Konstantenoperationen entdeckt und berechnet.
Ein Fall, bei dem eine Berechnung der Konstante vor dem Compilieren nicht möglich ist, sind die Zeichenketten. Nicht alle 256 möglichen Zeichencodes lassen sich innerhalb von Anführungszeichen darstellen. Diese Codes werden trotzdem sehr oft benötigt, z.B. bei der Arbeit mit der Floppy.

Der Zugriff auf einen bestimmten Datensatz einer relativen Datei könnte z.B. so lauten:

```
PRINT#2,"P"+CHR$(2)+CHR$(10)+CHR$(1)+CHR$(5)
```

Die Berechnung dieser Zeichenkette ist nicht nur sehr lang, sondern auch langsam. Im compilierten Programm belegt sie

dagegen nur die notwendigen 5 Bytes und braucht nicht mehr berechnet zu werden, da dies bereits der Compiler getan hat.

3.3 Verarbeitung von Zeichenketten

Ist es Ihnen schon einmal passiert, daß ein Programm plötzlich anhält, minuten- oder stundenlang nichts geschieht und das Programm dann weiterarbeitet, als sei nichts gewesen? Wenn nicht, dann starten Sie einmal folgendes Programm (ohne Compiler):

```
10 DIM A$(9000):A$=CHR$(64):B$=CHR$(65)
20 FOR I=0 TO 9000
30 A$(I)=A$
40 NEXT
50 FOR I=0 TO 9000
60 A$(I)=B$:PRINT A$(I);
70 NEXT
```

Der erste Programmteil läuft relativ schnell ab, das Programm beginnt mit dem Ausdrucken der Zeichenketten und stoppt plötzlich. Ein Drücken der STOP-Taste bleibt wirkungslos. Nach mehr als einer Stunde setzt das Programm seine Arbeit fort und beendet sie dann wie vorgesehen.

Der Grund für dieses merkwürdige Verhalten liegt in der Speicherverwaltung des Basic-Interpreters. Bei jeder Zuordnung einer Zeichenkette an eine Variable wird der vorherige Inhalt der Variablen nicht gelöscht, sondern bleibt im Speicher liegen. Diese Speicherverschwendung hat zur Folge, daß irgendwann der Speicher unvermeidbar voll ist. Damit das Programm seine Arbeit fortsetzen kann, wird eine Routine gestartet, die alle nicht mehr benötigten Zeichenketten sucht und sie aus dem Speicher enfernt (Garbage Collection). Diese Routine benötigt natürlich um so mehr Zeit, je größer die Anzahl der benutzten Zeichenketten ist. Die benötigte Zeit ist allerdings nicht proportional zur Anzahl der Zeichenketten, sondern steigt quadratisch an. Für Programme, die viele Zeichenketten verarbeiten, ist die Programmiersprache BASIC somit nicht geeignet. Dies ist ein wesentlicher Grund, wieso anspruchsvollere Programme (Textverarbeitung, Dateiverwaltung) meistens in Maschinensprache geschrieben sind. Es gibt allerdings eine Lösung für dieses Problem. Sie brauchen Ihre Programme nur mit BASIC 64 zu compilieren. Bei obigem Beispiel wird nach dem compilieren aus der mehrstündigen Pause ein kurzes Stocken des Programmen. Auch in den extremsten Fällen ist die Garbage Collection eines compilierten Programmes nach maximal einer Sekunde beendet. Sollte Ihnen dies immer noch zu lange dauern, so gibt es die Möglichkeit, das Auslösen einer Garbage Collection vom Programm zu steuern und an weniger zeitkritischen Stellen des Programmen ausführen zu lassen. Dies geschieht durch Benutzung der FRE-Funktion.

BASIC 64 führt noch weitere Optimierungen bezüglich der schnelleren Ausführung von Zeichenkettenoperationen aus. Besonders bei komplexeren Zeichenkettenformeln sind diese sehr wirkungsvoll, wie z.B. bei:

```
A$=LEFT$(A$,X%-1)+B$+MID$(A$,X%)
```

Die Zeichenkette B$ wird in die Kette A$ an der Stelle X% eingefügt. Der Basic-Interpreter würde zuerst mehrere Teilzeichenketten generieren und diese dann aneinanderhängen. Das compilierte Programm ist dagegen derart optimiert, daß es nur die Ergebniszeichenkette berechnet. Zwischenergebnisse entstehen nicht. Dadurch arbeitet diese zusammengesetzte Formel fast genauso schnell, wie eine einzelne Funktion. Durch diese Fähigkeit und die schnelle Garbage Collection wird die Programmiersprache Basic nun auch für anspruchsvollere Programme einsetzbar.

3.4 Integer-Optimierung

Im Commodore-Basic existieren zwei verschiedene Datentypen zur Darstellung von numerischen Werten. Diese sind die Gleitkommazahlen (REAL) und die ganzen Zahlen (INTEGER). Eine Variable zur Aufnahme von Gleitkommazahlen kann einen beliebigen Namen haben, dabei werden nur die ersten beiden Zeichen vom Interpreter beachtet. Eine Variable, die als letztes Zeichen ihres Namens ein %-Zeichen besitzt, kann dagegen nur ganze Zahlen aufnehmen, der Interpreter beachtet nur die ersten beiden Zeichen des Namens und das %-Zeichen. Eine Variable, die auf diese Weise gekennzeichnet wurde, kann nur Werte zwischen -32768 und +32767 ohne Nachkommastellen aufnehmen. Obwohl derartige Variablen normalerweise selten benutzt werden, so ist der diesen Variablen zugrundeliegende Datentyp ein wesentlich wichtigerer als der Gleitkommadatentyp. Dies bemerkt ein Basic-Programmierer normalerweise nicht, da der Basic-Interpreter beide Datentypen ineinander umwandelt, wenn dies nötig ist, und manchmal sogar, wenn dies nicht nötig ist.

Es folgt eine Liste aller Befehle, Operationen und Funktionen, zu deren Ausführung der Basic-Interpreter nur ganze Zahlen benötigt und eine entsprechende Umwandlung der Gleitkommadarstellung veranlaßt:

On Goto, On Gosub, Wait, Load, Save, Verify, Poke, Cmd, Sys, Open, Close, Tab, Spc, Not, And, Or, Fre, Pos, Peek, Len, Asc, Chr$, Left$, Right$, Mid$, Index von Feldern, etc.

Bei folgenden Operationen verwendet der Interpreter die Gleitkommadarstellung und wandelt ganze Zahlen entsprechend um:

Schleifenzähler, Input, Print, Read, IF, Def, +, -, *, /, >, <, =, <=, >=, <>, Sgn, Int, Abs, Usr, Sqr, Rnd, Log, Exp, Cos, Sin, Tan, Atn, Str$, Val.

Bei folgenden Operationen ist eine Umwandlung in das Gleitkommaformat nicht immer nötig, obwohl der Basic-Interpreter dies trotzdem macht:

Schleifenzähler, Input, Print, Read, If, +, -, *, /, >, <, =, <=, >= <>, Int.

Der Basic-Interpreter führt zusätzlich alle Berechnungen im Gleitkommaformat durch; handelt es sich um eine ganzzahlige Operation, so wird vor der Operation das Format gewandelt und hinterher wieder zurück. Bei der Verwendung von Integer-Variablen wird vor der Operation zuerst in eine Gleitkommazahl gewandelt und danach wieder zurück.

Aus diesen Aufzählungen geht hervor, daß nichtmathematische Programme vollständig ohne Gleitkommavariablen auskommen können. Bei mathematischen Programmen könnte die Benutzung von Gleitkommavariablen unter Umständen ebenfalls stark gesenkt werden. Auf jeden Fall ist der Großteil aller Datentypwandlungen des Basic-Interpreters überflüssig. Hinzu kommt, daß die vom Interpreter benutzten Routinen zur Umwandlung von Datentypen langsam und umständlich arbeiten. Weiterhin sind Gleitkommaoperationen um Größenordnungen langsamer als ganzzahlige Operationen. Da der Basic-Interpreter bei einer entsprechenden Wahlmöglichkeit immer Gleitkommazahlen einsetzt, merkt der Basic-Programmierer hiervon nichts. Ein Basic-Programm könnte also um ein Vielfaches schneller arbeiten, wenn die Gleitkommadarstellung nur noch eingesetzt wird, wenn sie nötig ist.

Ein mit BASIC 64 compiliertes Programm macht immer maximalen Gebrauch von der ganzzahligen Darstellung, hierbei ist der Compiler allerdings auf die Mitarbeit des Programmierers angewiesen:

- Der Compiler kann bei der Übersetzung des Programmen noch nicht feststellen, welche Datentypen während der Laufzeit in welchen Variablen benutzt werden.

- Das laufende Programm könnte dies zwar feststellen, es würde dadurch allerdings wesentlich langsamer, da es nicht nur nach jeder Rechenoperation den Datentyp feststellen müßte, sondern auch eventuell den Datentyp wandeln muß.

Aus diesen Gründen nimmt BASIC 64 an, daß es sich bei allen Variablen um Gleitkommavariablen handelt, mit Ausnahme der folgenden Fälle:

- Variablen, die mit einem %-Zeichen gekennzeichnet sind, wie dies der Einteilung des Basic-Interpreters entspricht.

- Variablen, die durch eine entsprechende Compileranweisung zu Integer-Variablen gemacht wurden.

- Alle Variablen, die in einem Programm vorkommen, das mit Optimierungsstufe 2 compiliert wird (Kapitel 7).

In Kapitel 2 haben Sie bereits gesehen, daß es sehr lohnend ist, wenn Integer-Variablen benutzt werden. Durchschnittlich sind 90% aller Variablen eines Programmes durch Integer-Variablen ersetzbar. Der Basic-Interpreter läßt es allerdings nicht zu, daß Integer-Variablen als Zählvariablen einer FOR-NEXT-Schleife eingesetzt werden. In diesem Fall kann man dies mit Hilfe einer Compileranweisung umgehen.

Häufig in Schleifen eingesetzte Variablen, wie I,J,K sollten deshalb zur Aufnahme von Integer-Zahlen durch eine entsprechende Compiler-Anweisung reserviert und nur zu diesem Zweck eingesetzt werden. Es gibt noch viele weitere Gründe, Integer-Variablen einzusetzen:

- Eine Integer-Variable belegt 2 Bytes Speicherplatz, eine Gleitkommavariable dagegen 5. Besonders bei Feldern ist dies ein entscheidender Vorteil von Integer-Variablen, der übrigens auch beim Basic-Interpreter auftritt.

- Compilierte Programme werden durch die Verwendung von Integer-Variablen kürzer, da es für die Adressen von Integer-Variablen eine Kurzform gibt. Dies gilt nur bei Generierung von P-Code.

- Viele besonders einfache Integer-Operationen gehören zum Befehlsschatz des Mikroprozessors 6510. Sie können deshalb in wenigen Mikrosekunden ausgeführt werden. Hierzu gehören z.B.:

I%+1, I%-1, I%*2, I%+J%, I%-J%.

Alle Programmiersprachen, die durch einen Compiler realisiert werden, unterscheiden streng zwischen Integer- und Gleitkommavariablen. Sollten Sie irgendwann auf eine leistungsfähigere Programmiersprache, wie z.B. Pascal, umsteigen, so ist es von Vorteil, mit dieser Unterscheidung vertraut zu sein. Es gibt sogar Programmiersprachen, die nur mit ganzen Zahlen arbeiten (Forth, Assembler).

3.5 Der Maschinensprache-Generator

Grundsätzlich lassen sich Compiler in zwei verschiedene Klassen einteilen, je nach der Art des von ihnen erzeugten Codes. Ein häufig angewendetes Verfahren ist das Generieren eines Pseudo-Codes (Speedcode), welcher dann von einem entsprechenden Interpreter abgearbeitet wird. Der Vorteil dieses Codes ist seine Kürze. Compilierte Programme werden dadurch je nach Programm 50-80% kürzer als das Original. Dies ist besonders wichtig, da der Compiler zu jedem Programm noch eine Programmsammlung (Runtimemodul) hinzufügt, die das compilierte Programm benötigt, und es dadurch verlängert. Der Nachteil des P-Codes ist die geringere Ausführungsgeschwindigkeit gegenüber einem Maschinenspracheprogramm. Obwohl der von BASIC 64 generierte P-Code speziell für den Commodore 64 entwickelt wurde, trifft dies auch für diesen P-Code zu.

Eine Alternative zur Generierung eines P-Codes ist die Übersetzung des Programmes in Maschinensprache. Dies hat den gewaltigen Vorteil, daß das Programm dadurch mit maximaler Geschwindigkeit abläuft, es aber wesentlich länger als ein P-Code Programm ist. Normalerweise verfügen nur größere Computer über Compiler, die Maschinensprache generieren können, da bei diesen Computern die Rechenzeit kostbar, Speicherplatz dagegen genügend vorhanden ist.

Viele Basic-Programme für den Commodore 64 nutzen den

Speicher nicht vollständig aus, hauptsächlich weil mit Hilfe des Interpreters nur 38K Byte der verfügbaren 64K Byte nutzbar sind. Aus diesem Grund kann BASIC 64 wahlweise einen kurzen P-Code oder längeren Maschinencode generieren oder sogar beides mischen (7. Kapitel). Durch das Übersetzen eines Programmen in Maschinesprache fällt bei Ablauf des Programmen natürlich das Interpretieren des P-Codes weg. Der Mikroprozessor des Commodore 64 (6510) benötigt zur Abarbeitung eines Befehles zwischen 2 und 7 Mikrosekunden. Diese Befehle sind gegenüber den Basic-Befehlen derart primitiv, daß mehrere Befehle nötig sind, um einen Basic-Befehl auszuführen. In vielen Fällen müssen sogar ganze Unterprogramme aufgerufen und ausgeführt werden. Ein Maschinenspracheprogramm ist nur dann wesentlich schneller als ein P-Code-Programm, wenn die Übersetzung mit wenigen Befehlen erfolgen kann, da in diesen Fällen das Interpretieren des P-Codes das Programm stark bremst. Basic-Befehle können nur dann in wenige Maschinenbefehle übersetzt werden, wenn es sich um einfache Befehle handelt. Einfache Befehle erkennt man an ihrer hohen Ausführungsgeschwindigkeit, wie z.B. alle Operationen mit Integer-Variablen und Integer-Zahlen. Die meisten Integer-Operationen lassen sich mit wenigen Maschinensprachebefehlen ausführen. In diesen Fällen wird das Programm nicht nur schneller, weil das Interpretieren des P-Codes entfällt, sondern auch weil spezielle Möglichkeiten der Maschinensprache direkt benutzt werden.

Zusammenfassend kann man sagen, daß ein Maschinenspracheprogramm nur dann wesentlich schneller arbeitet als ein P-Code-Programm, wenn das P-Code-Programm schon ziemlich schnell arbeitet. Bei Programmen, die fast ausschließlich komplexe Gleitkommaoperationen (SIN, SOR, etc.) benutzen, bringt die Generierung von Maschinensprache keine nennenswerte Geschwindigkeitssteigerung.

Die Frage, wie sich die Optimierungsmöglichkeiten von BASIC 64 und das Generieren von Maschinencode im Einzelfall auswirken, läßt sich natürlich nur durch Ausprobieren beantworten. Durch geschickte Ausnutzung aller Möglichkeiten lassen sich die meisten Programme beachtlich beschleunigen. Ein kleines Beispiel dafür haben Sie ja bereits in Kapitel 2 gesehen.

Das Einschalten des Maschinencodegenerators geschieht über Punkt "A" im Entwicklungspaket. Drücken Sie hierzu nach Start des Compilers die Taste "3", um in das Entwicklungspaket zu gelangen. Mit der Taste "A" können Sie nun das Generieren von 6502/6510-Code wählen und dann mit "Return" wieder ins Hauptmenü gelangen. Das vom Compiler generierte Programm erhält nun ein "M-" vor den Namen, anstelle des bei P-Code üblichen "P-". Ansonsten ist die Bedienung des Compilers dieselbe, wie bei Benutzung des P-Code-Generators.

4. Kapitel

Details

4.1 Arbeitsweise des Compilers

Nach der Auswahl der Punkte 1 oder 2 im Hauptmenü und der Eingabe des Programmnamens beginnt der Compiler mit der Übersetzung des Programmes (Kapitel 1). Der Compiler unterscheidet hierbei zwischen zwei Arbeitsphasen. Diese Phasen werden mit Pass 1 und Pass 2 bezeichnet.

Pass 1:

In Pass 1 wird das Programm interpretiert, optimiert und der entsprechende Code (P-Code oder Maschinensprache) erzeugt. Der Compiler gibt dabei die laufende Zeilennummer und jedes Befehlstrennzeichen (Doppelpunkt) aus. Findet der Compiler eine Compileranweisung (REM), so gibt er ein "R" aus. Befehle oder Funktionen, die nicht innerhalb des Commodore-Basic definiert sind (z.B. aus Basis-Erweiterungen), führen zur Ausgabe des Buchstabens "E".

Pass 2:

In Pass 2 wird der generierte Code nochmals vom Compiler durchgegangen und vervollständigt, weiterhin wird das Runtimemodul an das Programm gebunden und die DATA-Zeilen in das Programm eingefügt. Während dieser Arbeiten gibt der Compiler folgende Meldungen aus:

- Data-Code-Start: Ab dieser Speicheradresse liegen die Data-Zeilen im compilierten Programm.

- Objekt-Code-Start: Ab der angegebenen Speicheradresse beginnt das eigentliche Programm.

- Strings: Dieser Bereich ist vollständig frei. Das compilierte Programm benutzt ihn zur Ablage von Zeichenketten. Die Anfangsadresse dieses Bereiches ist identisch mit dem Ende des compilierten Programmes. Über der Endadresse des Zeichenkettenbereiches bis zum Ende des Speichers (normalerweise 65535) werden alle Variablen und Variablenfelder gespeichert, die das Programm benötigt.

- Extentions: Sollte das Programm Befehle einer Basis-Erweiterung benutzen, so nennt der Compiler deren Anzahl.

- Errors: Sind im Programm Fehler aufgetreten, so nennt der Compiler diese in der entsprechenden Zeile und gibt vor Beendigung des Pass 2 eine Liste aller Zeilen aus, in denen Fehler auftraten.

4.2 Fehlermeldungen

Die in Basic möglichen Fehlermeldungen teilen sich in zwei Gruppen auf:

Programmfehler:

Fehler, die dazu führen, daß der Compiler einen Befehl nicht übersetzen kann, werden bereits während der Compilierung gemeldet. Der Compiler gibt dann dieselbe Fehlerbeschreibung wie der Basic-Interpreter aus, wenn dieser den entsprechenden Befehl ausführen müßte. Vom Compiler entdeckt werden folgende Fehler:

Syntax - Bedeutung wie in Basic.

Redim'd Array - Bedeutung wie in Basic.

Type mismatch - Bedeutung wie in Basic.

Bad Subscript - Ein Feldzugriff besitzt eine falsche Indexanzahl gegenüber der Dimensionierung.

Undef'd Statement - Ein GOTO/GOSUB-Befehl bezieht sich auf eine nicht vorhandene Zeile. Dieser Fehler wird erst in Pass 2 entdeckt.

Out of memory - Programm und Variablen passen nicht mehr in den Speicher, dies passiert normalerweise nur, wenn die Speicherobergrenze mit dem Entwicklungspaket herabgesetzt wurde.

Runtime - Der Compiler versucht Formeln, so weit wie möglich, während der Compilierung zu berechnen (hauptsächlich bei Konstanten). Sollten diese Formeln nicht berechenbar sein, so gibt der Compiler diese Meldung aus, da es sich dabei um einen Fehler handelt, der normalerweise erst während der Laufzeit des Programmes aufgetreten wäre und deshalb nicht näher bestimmt werden kann. Ein Beispiel hierfür ist eine konstante Division durch Null:

```
10 A=1/0
```

System error - Diese Meldung wird ausgegeben, wenn das Betriebssystem des Commodore 64 einen Fehler findet, z.B. wenn die Floppy nicht eingeschaltet ist.
In fast allen Fällen kann der Compiler nach der Entdeckung eines Fehlers seine Arbeit wieder aufnehmen und nach weiteren Fehlern suchen.

Achtung:

Ein Programm, in dem Fehler vorhanden sind, ist natürlich nur eingeschränkt lauffähig, dies gilt hauptsächlich für die Zeilen nach den Fehlern. Sie sollten die vorhandenen Fehler beheben und das Programm neu compilieren.

Laufzeitfehler:

Viele Fehler können vom Compiler nicht entdeckt werden, da sie vom Ablauf des Programmes abhängen. Es handelt sich dabei um alle noch nicht erwähnten Fehlermeldungen. Diese Fehlermeldungen haben dieselbe Bedeutung, wie beim Interpreter, es ist folgendes zu beachten:

Out of Memory - Diese Meldung bedeutet entweder, daß der Zeichenkettenbereich nicht alle Zeichenketten aufnehmen kann oder daß der Stapel für FOR, GOSUB und Klammerebenen voll ist. Die Verwendung zuvieler Variablen wird bereits vom Compiler entdeckt.

Bad Subscript - Der Index eines Feldzugriffes liegt über den Feldgrenzen, eine falsche Indexanzahl wird bereits vom Compiler entdeckt.

Formula too complex - Diese Meldung wird vom Interpreter ausgegeben, wenn eine Zeichenkettenformel zu tief verschachtelt ist, was normalerweise nicht auftritt. Ein compiliertes Programm gibt diese Meldung nie aus, da eine tiefere Verschachtelung ebenfalls verarbeitet wird.

Illegal Quantity - Diese Meldung wird vom Basic-Interpreter auch ausgegeben, wenn die Ausführung des Befehles eigentlich möglich wäre, z.B. bei:

```
10 A=ASC("")
```

Beim Einlesen von Dateien mit GET wird ein Nullbyte als Leerstring eingelesen. Umgekehrt gilt ein Leerstring dagegen für den Interpreter nicht als Nullbyte, sondern er gibt eine Fehlermeldung aus. In diesem Fall und weiteren eindeutigen Fällen gibt das compilierte Programm keine Fehlermeldung aus.

Bei der Umwandlung von Gleitkommazahlen in Integer-Werte oder bei Rechnungen mit Integer-Zahlen kann es vorkommen, daß das Ergebnis nicht im Integer-Bereich liegt. Der Compiler setzt allerdings nur Integer-Berechnungen ein, wenn ein solcher Fall bei der Ausführung des Programmen durch den Interpreter ein "Illegal Quantity" zur Folge hätte. Sollte dies bei Ihrem Programm niemals der Fall sein, so können Sie sicher sein, daß auch bei compilierten Programmen kein Bereichsüberlauf auftritt. Aus diesem Grund stellt das compilierte Programm bei Integer-Berechnungen aus Geschwindigkeitsgründen keine Bereichsüberprüfung an. Sollte bei Zwischenergebnissen dennoch einmal der Bereich überschritten werden und das Endergebnis wieder im Integer-Bereich liegen, so wird das Ergebnis auch mit Integer-Operationen korrekt ermittelt. Dies kann man sich z.B. beim Poke-Befehl zunutze machen (Kapitel 7).

Der Input-Befehl des C-64-Basics besitzt einige Besonderheiten, die bei fehlerhaften Eingaben wirksam werden:

Die Eingabe einer Zeichenkette anstelle einer Zahl führt zur Meldung "Redo from Start" und zur vollständigen Wiederholung des Input-Befehls. Das compilierte Programm verhält sich genauso.

Bei einer unvollständigen Eingabe fordert das Programm mit zwei Fragezeichen zur Eingabe weiterer Werte auf (Compiler/Interpreter).

Bei gar keiner Eingabe wird der Input-Befehl dagegen ignoriert und die Variablen nicht verändert. Dies ist eine Besonderheit des C-64-Basics, die auf anderen Commodore-Computern nicht vorhanden ist. BASIC 64 wurde speziell für den Commodore 64 entwickelt und unterstützt deshalb diese oft benutzte Möglichkeit.

Bei Eingabe von zuvielen Werten gibt der Interpreter die Meldung "Extra ignored" aus. Ein compiliertes Programm ignoriert diese Werte dagegen ohne eine (oft störende) Meldung. Der Input-Befehl ist dadurch innerhalb von Bildschirm-Masken besser einsetzbar.

Disketten-Fehler:

Bei Auftreten eines Fehlers bei der Arbeit der Floppy-Disk meldet der Compiler den Fehler und beendet das Compilieren. Die Bedeutungen der Diskettenfehlermeldungen sind im Handbuch der Floppy-Disk erläutert. Meldungen, wie READ ERROR, WRITE ERROR, NO CHANNEL und WRITE FILE OPEN deuten auf eine beschädigte Diskette hin, diese sollte ausgewechselt werden.

Bei der Arbeit mit BASIC 64 und dem Basic-Interpreter im allgemeinen kann es vorkommen, daß ein Basic-Programm verbessert und neu abgespeichert werden muß. Dies geschieht meistens mit folgendem Befehl:

```
SAVE "£:Name",8
```

Aufgrund eines Fehlverhaltens des Diskettenbetriebssystems kann es bei Anwendung dieses Befehles in seltenen Fällen zu Datenverlusten auf der benutzten Diskette kommen. Dies ist natürlich unabhängig davon, ob BASIC 64 benutzt wird oder nicht. Es ist sehr empfehlenswert, folgende Ersatzbefehle zu benutzen:

```
OPEN 15,8,15,"S:Name": CLOSE 15: SAVE "Name",8
```

4.3 Die Zeilenliste

Treten während der Laufzeit eines Programmen Fehler auf, so wird eine entsprechende Meldung ausgegeben. Anstelle einer Zeilennummer kann das Programm nur eine Speicheradresse ausgeben, da compilierte Programme nicht mehr in Zeilen organisiert sind. Zur Feststellung der fehlerhaften Zeile wird eine Zeilenliste benötigt, welche der Compiler generieren kann. Dazu dient der Punkt "D" im Entwicklungspaket, mit welchem man das Erstellen einer Zeilenliste einschalten kann. Bei Programmen, die noch nicht ausprobiert wurden, sollte immmer eine Zeilenliste generiert werden. Die Benutzung dieser Liste ist denkbar einfach.

- Notieren Sie sich die Speicheradresse, in der der Fehler auftrat.

- Laden Sie die Zeilenliste mit LOAD"Z-Programmname",8.

- Listen Sie die Liste bis zur gewünschten Stelle mit LIST-Fehleradresse.

- Auf der rechten Seite der Liste finden Sie nun die entsprechenden Zeilen, die den Speicheradressen auf der linken Seite zugeordnet sind. Die letzte genannte Zeile enthält den Fehler. Da der Fehler immer vor der genannten Speicheradresse auftrat, kann er in seltenen Fällen auch am Ende der vorhergehenden Zeile liegen.

- Verbessern Sie das Programm und compilieren Sie es erneut. Sie können dem Programm auch die Anweisung geben, eine spezielle Programmzeile zu starten, wenn ein Fehler auftritt (Kapitel 6).

4.4 Gleitkommmaberechnungen.

Der Interpreter führt Gleitkommaberechnungen auf 9 Stellen (32 Bit) genau aus. Alle weiteren Stellen werden gerundet, wobei die Genauigkeit vor dem Runden sogar 40 Bit beträgt. Trotzdem sind die Gleitkommaroutinen vorsichtig zu benutzen. Ein drastisches Beispiel hierfür ist folgende Basic-Zeile:

```
10 IF 3↑4*4↑3=4↑3*3↑4 THEN PRINT "MULTIPLIKATION UMKEHRBAR"
```

Die Ausführung dieses Programmes bringt eine merkwürdige Erkenntnis. Das Resultat einer Gleitkommamultiplikation ist abhängig von der Reihenfolge der Faktoren. Benutzen Sie in Ihren Programmen derartige Gleitkommaberechnungen nur für rechnerische Zwecke, niemals aber, um den Ablauf von Programmen zu steuern, sonst ist Ihr Programm abhängig von Rundungsfehlern der Gleitkommarechnung. Besser wäre z.B. folgende Zeile:

```
10 IF ABS(3↑4*4↑3-4↑3*3↑4)<.001 THEN PRINT "OK"
```

Nach dem Compilieren von Programmen werden Formeln teilweise auf andere Art (durch Optimierungen) und teilweise genauer berechnet. Rundungsfehler wirken sich somit in sehr seltenen Fällen anders aus. Besonders drastisch ist dies bei der Funktion SQR, die nach dem Compilieren wesentlich genauer arbeitet. Einen weiteren Rechenfehler des Interpreters zeigt folgendes Beispiel:

```
10 PRINT INT(30.4*10)
```

Als Ergebnis erscheint die Zahl 303, obwohl 304 zu erwarten ist. Programme, die von Rundungsfehlern abhängig sind, sollte es aber normalerweise nicht geben.

4.5 Felddimensionierung

Die Grenzen von Feldern müssen bereits während der Compilierung bekannt sein; dies hauptsächlich aus folgenden

Gründen:

- Felder können auch in dem Adressbereich von $A000-$FFFF liegen, was einen zusätzlichen Speicher von 24K bedeutet. Einzelvariablen dürfen dort allerdings nicht liegen, da auf diesen Bereich auch Maschinenprogramme nicht immer zugreifen können. Daraus resultiert eine komplizierte Speicherverwaltung, die nur der Compiler übernehmen kann.

- Auf Felder sollte möglichst schnell und direkt zugegriffen werden können. Der Compiler benötigt dazu die Felderadressen.

In manchen Fällen sind die Grenzen eines Feldes während der Compilierung nicht bekannt, z.B. bei folgendem DIM-Befehl:

```
10 INPUTX:DIMA(X)
```

In diesem Fall gibt der Compiler den Feldnamen, gefolgt von einer Klammer und einem Fragezeichen, aus. Geben Sie dann die maximale Grösse des Feldes an. Im Beispiel wäre dies der maximale Wert für X. Bei mehreren Dimensionen fragt der Compiler nur nach den unbekannten Werten; trennen Sie die Werte dabei durch Drücken der "Return-Taste".

Obwohl dem compilierten Programm normalerweise genügend Speicher zur Verfügung steht, ist die Frage nach dem maximalen Feldindex oft auch eine Frage des benötigten Speicherplatzes. In Kapitel 9 finden Sie eine Liste der verschiedenen Datentypen und ihres Speicherverbrauchs. Alle in einem Programm benutzen Felder sollten soweit wie möglich in den ersten Zeilen des Pragrammes dimensioniert werden, dies erlaubt nicht nur eine schnellere Änderung, sondern hilft dem Compiler auch bei einer effizienten Speicheraufteilung und ist deshalb besonders wichtig.

Bei der Erstellung von Programmen, die mit großen Feldern arbeiten, tritt das Problem auf, daß dem interpretierten Programm weniger Speicher (38K) als dem compilierten Programm (62K) zur Verfügung steht. In diesen Fällen ist es sinnvoll, das Programm zuerst mit einem kleineren Feld auszuprobieren und es dann zu compilieren:

```
10 N=15000:DIM X%(N)
```

Beim compilieren fragt der Compiler dann nach dem Maximalindex des Feldes, wo dann ein größerer Wert eingesetzt werden kann. In diesem Beispiel wäre z.B. 25000 ein möglicher Index.

4.6. Direktmodusbefehle

Einige Basic-Befehle lassen sich normalerweise nicht innerhalb von Programmen einsetzen oder haben dort eine andere Wirkung. Dieses sind folgende Befehle:

LIST, LOAD, SAVE, VERIFY und CONT.

Der Befehl LOAD lädt ein Programm in den Speicher und startet

es, wenn er innerhalb eines Programmes eingesetzt wird. Führt das compilierte Programm kein CLR durch, so kann es Variablen des vorherigen Programmes benutzen. Dieser Effekt kann in Basic nur benutzt werden, wenn einige Regeln beachtet werden, bei compilierten Programmen ist dies nicht nötig (Kapitel 6). Der Befehl SAVE kann dazu benutzt werden, bestimmte Speicherbereiche abzuspeichern. Dazu müssen die entsprechenden Speicherzellen (43-46, siehe 64 Intern) mit Poke geändert und nach dem Save auf ihren vorherigen Wert zurückgesetzt werden. Während dieser Operation kann der Interpreter auf keine Variablen zugreifen, ein compiliertes Programm ist dazu in der Lage.
Die übrigen Befehle ergeben in compilierten Programmen keinen Sinn und werden vom Compiler ignoriert. Z.B. kann ein compiliertes Programm natürlich nicht mehr aufgelistet werden, da es kein Basic-Programm mehr ist.

4.7 Integer-Schleifen

Integer-Variablen dürfen normalerweise nicht als Zähler in FOR-NEXT-Schleifen benutzt werden, mit einer entsprechenden Compileranweisung oder der Optimierungsstufe 2 ist dies trotzdem möglich. Integer-Schleifen sind nicht nur schneller, sie belegen auch weniger Stapelplatz und können deswegen tiefer ineinander geschachtelt werden. Außerdem ist die Verarbeitung und der Zugriff auf die Schleifenvariable innerhalb der Schleife schneller. Folgendes Beispielprogramm benutzt zwar eine Integer-Schleife, kann aber trotzdem mit dem Interpreter ausgeführt werden:

```
5 REM@ I=I,J
10 FOR I=1T0 100:PRINT I
20 FOR J=1T0 50
30 NEXTJ,I
```

Eine Einschränkung ergibt sich allerdings für die Benutzung der Integer-Schleifen. Aus Geschwindigkeitsgründen wird kein STEP-Wert gespeichert. Die Schrittweite ist immer der Wert 1 und der Befehl STEP ist bei Integer-Schleifen nicht erlaubt. Normalerweise liegt es in der Natur ganzzahliger Schleifen, daß sie den Schrittwert 1 besitzen. Um schnelle Schleifen mit anderen STEP-Werten zu benutzen, kann z.B. auf folgende Hilfskanstruktion ausgewichen werden, die aufgrund des geringeren Verwaltungsaufwandes und der feststehenden Sprungadresse sogar ein wenig schneller ist, als eine ähnliche FOR-NEXT-Schleife:

```
10 I%=1
20 REM Schleifeninneres
10 I%=I%+2:IF<=1000 THEN20
```

Die höhere Geschwindigkeit tritt besonders bei Verwendung des Maschinensprachegenerators auf. Etwas langsamer, aber dafür einfacher, ist folgendes Programm:

```
5 REM@ I=I
10 FOR I=1 TO 1000
20 REM
30 I=I+1:NEXT
```

5. Kapitel

Basic Erweiterungen

5.1 Compilierbarkeit von Erweiterungen

Der Basic-Befehlsschatz des Commodore 64 ist zwar sehr leistungsfähig, er besitzt aber nicht die Möglichkeit, die speziellen Fähigkeiten des Commodore 64 voll auszunutzen. Besonders bei der Benutzung der hochauflösenden Grafik ist das Entwickeln entsprechender Unterprogramme sehr mühsam und eine Basic-Erweiterung ist notwendig. Leider existiert für den Commodore 64 keine Standard-Erweiterung. Aus diesem Grund ist BASIC 64 in der Lage, die Befehle der verschiedenen Basic-Erweiterungen ebenfalls zu compilieren und sie in das fertige Programm einzubinden. Damit möglichst viele Erweiterungen compilierbar sind, wendet BASIC 64 ein allgemeines Verfahren zum Compilieren von Erweiterungen an. In einigen Erweiterungen existieren Befehle, die mit diesem Verfahren nicht compilierbar sind, sowie einige Befehle, die überhaupt nicht in ein compiliertes Programm eingefügt werden können. Für die Compilierbarkeit von Erweiterungs-Befehlen sind folgende Punkte zu beachten:

- Ein compilierbarer Befehl hat folgendes Format:

```
BEFEHL Wert,Wert,Wert...
```

Jeder Wert kann eine Konstante, Variable oder Formeln eines beliebigen Datentyps sein. Die Anzahl der Werte ist beliebig. Als Trennzeichen für Werte sind alle Zeichen und Basic-Wörter erlaubt, die normalerweise nicht innerhalb von Formeln auftreten. Beispiele für Befehle, die dieses Format erfüllen, sind:

```
PLOT ,1,3*SIN(X) TO 4,A*B     ;4 Werte
GMODE 0,1                     ;2 Werte
```

Viele Erweiterungen besitzen auch zusätzliche Funktionen, die man mit folgendem Format benutzen kann:

```
Variable = FUNKTION(Wert,Wert,...)
```

Die Klammern können entfallen, wenn der Funktion kein Wert übergeben wird. Nicht immer kann 'der Compiler den Ergebnisdatentyp einer Funktion aus der Formel entnehmen. Es ist deswegen ratsam, das Funktionsergebnis immer einer Variablen zuzuweisen, die den entsprechenden Datentyp aufnehmen kann.

Beispiele für compilierbare Funktionen sind:

```
X=EVAL(A$)
A$=INSERT(B$,C$,2)
```

- Alle Befehle, die innerhalb von Programmen keinen Sinn

ergeben, sind natürlich nicht compilierbar. Hierzu gehören alle Programmierhilfen, wie z.B. TRACE, RENUMBER, AUTO, etc.

- Befehle, die den Ablauf von Programmen direkt verändern, sind nicht compilierbar. Hierzu gehören z.B. Sprungbefehle (CALL, EXEC, CGOTO, etc.) sowie Programmstrukturen (REPEAT, UNTIL, etc.).

- Befehle, die in die Speicherverwaltung des Programmes eingreifen, sind nicht compilierbar. Diese Befehle sind allerdings ziemlich selten (GLOBAL, LOCAL, etc.).

- Befehle, die einer Variablen einen Wert zuordnen, sind nicht compilierbar. In den üblichen Erweiterungen werden hierzu allerdings die compilierbaren Funktionen benutzt. Derartige Befehle sind deshalb ebenfalls selten (INPUTLINE, etc.).

Glücklicherweise betreffen diese Einschränkungen nur wenige und teilweise selten benutzte Befehle der üblichen Basic-Erweiterungen. Wichtige Befehle, wie z.B. die Grafik-Befehle, sind dagegen compilierbar. Durch Spezialanpassungen an die meistbenutzten Basic-Erweiterungen werden darüber hinaus viele Befehle dieser Erweiterungen vom Compiler akzeptiert, die normalerweise nicht compilierbar wären. Dies betrifft insbesondere die Supergrafik und das Basic 4.0.

5.2 Supergrafik 64 und Supergrafik 64 Plus

Diese Basic-Erweiterung wird von BASIC 64 fast vollständig unterstützt, da der Befehlsaufbau dieser Erweiterung dem Compiler bekannt ist. Sie können somit fast jedes Programm, das für die Supergrafik geschrieben wurde, auch compilieren. Ausnahmen bilden hierbei folgende Befehle:

Utilitiesi Die Befehle MERGE, RENUM und DTASET sind nicht compilierbar. Die Befehle MERGE und RENUM tauchen normalerweise innerhalb von Programmen sowieso nicht auf. Die ältere Version der Supergrafik enthält keine Utilities.

Der Befehl SREAD ist nicht compilierbar. Er läßt sich aber leicht durch normale Befehle ersetzen:

```
SREAD A$
```

wird zu

```
FOR X=1T063:READ B:A$=A$+CHR$(B):NEXT
```

Sollten Sie einen Spriteeditor besitzen (z.B. den Spriteformer der neuen Supergrafik), so ist es empfehlenswert, Sprites direkt von Diskette einzulesen, da bei einer Änderung des Sprites das Programm nicht noch einmal compiliert zu werden braucht.

Der Befehl IF# und der zugehörige Befehl IRETURN sind nicht compilierbar. Die Abfrage der Joysticks läßt sich auch im

normalen Basic programmieren:

```
10 POKE56322,224:REM Tastatur abschalten
20 J%=PEEK(56320):REM Abfrage Port 2
30 IF(J% AND1)=0THEN PRINT"OBEN"
40 IF(J% AND2)=0THEN PRINT"UNTEN"
50 IF(J% AND4)=0THEN PRINT"LINKS"
60 IF(J% AND8)=0THEN PRINT"RECHTS"
70 IF(J% AND16)=0THEN PRINT"KNOPF"
80 POKE56322,255:REM Tastatur ein
```

Zur Abfrage von Port 1 wird die Adresse in Zeile 20 auf 56321 geändert.

Die ebenfalls mit IF# mögliche Kollisionsabfrage von Sprites läßt sich folgendermaßen ersetzen:

```
10 IF PEEK(53278)THEN PRINT"SPRITE-SPRITE KOLLISION"
20 IF PEEK(53279)THEN PRINT"SPRITE-HINTERGRUND KOLLISION"
30 POKE 53278,0:POKE 53279,0:REM Register löschen
```

Es lassen sich sogar die betroffenen Sprites feststellen, da in diesen Speicherstellen jedes Bit für ein Sprite steht.

Achtung:

Programme, in denen nichtcompilierbare Befehle der Supergrafik: vorkommen, sind nicht ablauffähig oder verhalten sich anders als das nichtcompilierte Programm. Achten Sie unbedingt darauf, daß diese Befehle in compilierten Programmen nicht benutzt werden. Lesen Sie vor dem Compilieren von Programmen der Supergrafik unbedingt die Abschnitte 5.6 und 5.9.

5.3 Simons' Basic

Beim Simons' Basic sind ebenfalls alle wichtigen Befehle compilierbar. Insbesondere gilt dies für die Grafikbefehle und die vielfältigen Funktionen. Alle Befehle, die den bereits erwähnten Bedingungen nicht entsprechen, sind nicht compilierbar. Beim Simons' Basic sind dies:

- Programmierhilfen, d.h. alle Befehle, die innerhalb von Programmen sowieso nicht eingesetzt werden oder im Simons' Basic als Programmierhilfen bezeichnet werden, wie:

AUTO, RENUMBER, CGOTO, RESET, MERGE, PAGE, OPTION, DELAY, FIND, TRACE, RETRACE, DUMP, COLD, DISAPA, SECURE,.OLD.

- Eingabekontrollen:

FETCH.

- Zahlenumwandlung:

% = Binär in Dezimal, $ = Hexadezimal in Dezimal.

- DESIGN- und PRINT AT-Befehl

- Programmstrukturen (ELSE ist compilierbar):

REPEAT .., RCOMP, LOOP .., PROC.., CALL, EXEC, LOCAL, GLOBAL.

- Fehlerbehandlung:

ON ERROR, NO ERROR, OUT.

Für die Fehlerbehandlung besitzt BASIC 64 eine eigene Möglichkeit (Kapitel 6).

Viele dieser nichtcompilierbaren Befehle lassen sich äußerst einfach ersetzen:

DESIGN:

Es gibt mehrere Möglichkeiten, diesen Befehl zu ersetzen. Die einfachste Möglichkeit zeigt folgendes Programm:

```
10 A=64*13:REM Basisadresse
20 FOR I=A TO A+62 STEP3
30 READ A$:FOR J=0 TO 2
40 W=0:FOR K=1 TO 8
50 W=W*2:IF MID$(A$,J*8+K,1)="B" THEN W=W+1
60 NEXT:POKE I+J,W:NEXT:NEXT
```

Sie können Ihr Sprite nun wie beim Befehl DESIGN konstruieren, der Klammeraffe wird dabei nur durch den Befehl DATA ersetzt.

Es ist natürlich auch möglich, den Befehl DESIGN als Sprite-Editor zu benutzen. Nach der Konstruktion eines Sprites mit DESIGN und dem Starten des Konstruktions-Programmes kann das Sprite mit PEEK aus den entsprechenden Speicherstellen ausgelesen und auf Diskette abgespeichert und bei Bedarf wieder geladen werden. Dies hat den Vorteil, daß ein dieses Sprite benutzendes Programm nicht neu compiliert zu werden braucht, wenn das Sprite geändert wird.

PRINT AT:

Dieser Befehl läßt sich leicht ersetzen:

```
10 PRINT AT(X,Y);Z
```

wird zu:

```
10 POKE 211,X:POKE 214,Y:SYS 58732:PRINT Z
```

Programmstrukturen:

Die Programmstrukturen des Simons' Basic sind leicht durch die üblichen Befehle GOTO, GOSUB und IF ersetzbar:

```
10 REPEAT
20 ...
30 UNTIL X=0
```

Normales Basic:

```
10 REM
20 ...
30 IF NOT X=0 THEN10
```

Struktur:

```
10 LOOP
20 ...
30 EXIT IF X=0
40 ...
50 END LOOP
60 ...
```

Ersatz:

```
10 REM
20 ...
30 IF X=0 THEN 60
40 ...
50 GOTO 10
60 ...
```

Struktur:

```
10 PROC LABEL
20 ...
30 END PROC
40 ...
50 EXEC LABEL
```

Ersatz:

```
10 REM
20 ...
30 RETURN
40 ...
50 GOSUB 10
```

Struktur:

```
10 PROC LABEL
20 ...
30 CALL LABEL
```

Ersatz:

```
10 REM
20 ...
30 GOTO 10
```

Befehl:

```
10 LOCAL A,B
20 ...
30 GLOBAL
```

Ersatz:

```
10 AH=A:BH=B
20 ...
30 A=AH:B=BH
```

Das Programm wird durch diese Änderungen schneller, was durch das anschließende Compilieren nochmals verstärkt wird. Das Programm wird durch diese Änderungen nicht unbedingt unstrukturierter, da diese Befehle teilweise selber keine echten Strukturen sind (CALL, EXIT).

Achtung:

Programme, in denen nichtcompilierbare Befehle des Simons' Basic vorkommen, sind nicht ablauffähig oder verhalten sich anders als das nichtcompilierte Programm. Achten Sie unbedingt darauf, daß diese Befehle in compilierten Programmen nicht benutzt werden. Lesen Sie vor dem Compilieren von Programmen des Simons' Basic unbedingt die Abschnitte 5.6 und 5.9.

5.4 Basis 4.0 (Diskomat, Master 64)

Einige Basic-Erweiterungen besitzen eine Syntax, die beim Compilieren zu Fehlern führen kann:

```
BACKUP D1 TO D0
```

Die Bezeichnungen D0 und D1 stehen für die entsprechenden Diskettenlaufwerke, der Compiler würde sie allerdings als Variablen ansehen und eine entsprechde Berechnung in das Programm einfügen. Für Erweiterungen, die eine derartige Syntax aufweisen, gibt es ein spezielles Compilierverfahren, bei dem alle Variablen oder Formeln, die in Befehlen der Erweiterung benutzt werden, in Klammern gesetzt werden müssen. Dies ist beim Basic 4.0 sowieso der Fall und dieser Compiliermodus eignet sich deswegen besonders für Befehle des Basic 4.0. Folgendes Beispiel zeigt eine typische Anwendung:

```
10 INPUT X
20 CATALOG D(X)
```

Basic 4.0 ist in den Programmen Diskomat und Master 64 enthalten. Die Variablen DS und DS$ können nur direkt nach Befehlen des Basic 4.0 benutz werden und nur jeweils eine der beiden Variablen. Bei Diskomat kann DS überhaupt nicht benutzt werden. Die Verwendung der Variablen DS und DS$ sollte also möglichst durch ein Auslesen des Fehlerkanales ersetzt werden. Die Variablen DS und DS$ werden ansonsten wie normale Funktionen einer Erweiterung behandelt. Nach der Beendigung eines compilierten Programmes unter Master 64 ist die Erweiterung nicht mehr aktiviert und eine Weiterarbeit nicht möglich.

Bei der Verwendung von Diskomat ist darauf zu achten, daß der Wert für das Speicherende (Menüpunkt "E") versionsabhängig ist.

Achtung:

Programme, in denen nichtcompilierbare Befehle des Master 64 bzw. des Diskomat vorkommen, sind nicht ablauffähig oder verhalten sich anders als das nichtcompilierte Programm. Achten Sie unbedingt darauf, daß diese Befehle in compilierten Programmen nicht benutzt werden. Lesen Sie vor dem Compilieren von Programmen des Master 64 bzw. des Diskomat unbedingt die Abschnitte 5.6 und 5.9.

5.5 Anwendung von Erweiterungen

EXBASIC LEVEL II

Im Advanced Development Package (Punkt 3) des Compilers können Sie über die Taste 'H' schon alle notwendigen Voreinstellungen für die Exbasic Erweiterung aufrufen. Allerdings ist das Hauptanwendungsgebiet von Exbasic die aktive Hilfe bei der Programmerstellung. Hier gelten generell die Bedigungen zum Compilieren, wie bei anderen Erweiterungen - also z.B. kein Compilieren von Hilfsbefehlen.
Ferner muß beachtet werden, daß Exbasic ursprünglich für die Commodore Computer der Serien 2000, 3000, und 8000 konzipiert wurde. Hier gibt es mehrere Versionen dieser Erweiterung, die sich unter Umständen sehr unterschiedlich verhalten. Testen Sie deshalb bitte vorher aus, welche Funktionen und Befehle Ihrer Erweiterung durch den Compilier vollständig übersetzt werden.
Beachten Sie bitte auch, daß die IF THEN ELSE Struktur durch einen Doppelpunkt gekennzeichnet sein muß:
IF A = 10 THEN B = 10 : ELSE B = 0

- Achten Sie darauf, daß Ihr Programm keine nichtcompilierbaren Befehle enthält. Basic-Erweiterungen werden ständig weiterentwickelt, so daß die Compilierbarkeit von Befehlen auch versionsabhängig sein kann. Im Zweifelsfalle muß dies ausprobiert werden.

- Löschen Sie die Basic-Erweiterung aus dem Speicher, nachdem Sie Ihr Programm gespeichert haben. Dies geschieht am sichersten durch ein kurzes Ausschalten des Computers. Ein Modul muß aus dem Modulschacht entfernt werden.

- Starten Sie den Compiler. Wählen Sie nun Punkt "3" und stellen Sie mit der Taste "H" die benutzte Basis-Erweiterung ein. Achten Sie auf den Unterschied zwischen den beiden Versionen der Supergrafik.

- Drücken Sie "Return" und fahren Sie wie in Kapitel 1 beschrieben fort.

- Sobald der Compiler einen Erweiterungsbefehl findet, gibt er ein "E" aus.

Achtung:
Der Compiler kann anhand der vom Interpreter abgespeicherten Codes nicht erkennen, ob es sich um einen compilierbaren Befehl handelt. Dies stellt sich erst beim Programmablauf heraus. Der Compiler kennt weder jeden Befehl einer Basic-Erweiterung noch ist er kompatibel zu einer Erweiterung. Der Compiler akzeptiert lediglich Befehle einer Erweiterung und fügt diese so in das Programm ein, daß diese Befehle beim Ablauf des Programmes dem Interpreter übergeben werden können.

- Nach Beendigung des zweiten Durchlaufs (Pass 2) nennt der Compiler die Anzahl der benutzten Erweiterungsbefehle.
- Zum Ablauf des compilierten Programmen ist zuvor die entsprechende Erweiterung zu laden bzw. das Modul einzustecken. Sollte das Programm nichtcompilierbare Befehle enthalten, so meldet entweder die Basic-Erweiterung einen Fehler oder das Programm arbeitet nicht wie erwartet. Bei Beachtung der Regeln für die Compilierbarkeit von Befehlen dürfte dies nicht passieren.

- Die Speicherzellen von 704 (Sprite 11) bis 767 dürfen nicht mit POKE-Befehlen geändert werden. Dies muß hauptsächlich beim Simons' Basic beachtet werden, da in diesen Bereich kein Sprite gelegt werden darf.

- Die Basic-Erweiterung ist nur bei der Ausführung von Befehlen der Erweiterung aktiv. Nach einer Unterbrechung des Programmes mit RESTORE ist die Basic-Erweiterung möglicherweise abgeschaltet.

Sollte sich ein compiliertes Programm, welches Befehle einer Erweiterung benutzt, einmal nicht genauso verhalten, wie das uncompilierte Programm, so kann dies folgende Ursachen haben:

- Der Compiler wurde falsch bedient, z.B. wenn die benutzte Basic-Erweiterung falsch eingestellt wurde. Lesen Sie sich in diesem Fall nochmals die Abschnitte 1.1 und 5.6 durch.

- Es wurden Befehle einer Erweiterung benutzt, die nicht mit einem compilierten Programm zusammenarbeiten können. Hierüber informiert der Abschnitt 5.1 und die entsprechenden Abschnitte zu den einzelnen Erweiterungen.

- Es wurden Befehle benutzt, die in der verwendeten Version der Erweiterung nicht vollständig implementiert sind. Hierüber informiert Abschnitt 5.9.

5.6 Arbeitsweise von compilierten Befehlen einer Erweiterung

Zur Ausführung eines compilierten Erweiterungsbefehles übergibt das Programm den Befehl an den erweiterten Interpreter, welcher ihn dann ausführt. Es wird dabei nur das Befehlsgerüst übergeben, alle Formeln sind bereits vom Programm berechnet. Nach der Ausführung des Befehles übernimmt das compilierte Programm wieder die Kontrolle und läuft weiter. Da das compilierte Programm eine schnellere, effektivere und vollständig andere Speicher- und

Programmverwaltung als der Interpreter besitzt, verläuft das Umschalten auf den Interpreter nicht bei allen Befehlen reibungslos. Dieses ist der Grund, warum bestimmte Erweiterungsbefehle in einem compilierten Programm nicht ablauffähig sind. Weiterhin wird die Ausführung der Befehle einer Erweiterung durch das Compilieren nicht beschleunigt, da das compilierte Programm zur Ausführung dieser Befehle auf die Routinen des Interpreters zurückgreifen muß. Beschleunigt werden lediglich die Befehle des eingebauten Commodore-Basic.
Bei Beachtung folgender Regeln können Sie auch bei Programmen, die Befehle einer Erweiterung benutzen, die von BASIC 64 gewohnten Geschwindigkeiten erreichen:

- Benutzen Sie sowenig Erweiterungsbefehle wie möglich.

- Benutzen Sie Erweiterungsbefehle nur in Programmteilen, die nicht zeitkritisch sind.

- Benutzen Sie Grafiken zur Darstellung des Rechenergebnisses und nicht während der Berechnung.

- Benutzen Sie statt mehrerer Einzelbefehle einen komplexeren Befehl. Das Zeichnen einer Linie mit einem Befehl ist z.B. schneller als die Aneinanderreihung von Punkten für denselben Zweck.

- Immer wenn sich Befehle einer Erweiterung durch normale Basic-Befehle ersetzen lassen, sollte dies auch getan werden. Dies gilt besonders für Zeichenkettenfunktionen. Komplexe Zeichenkettenformeln im normalen Basic arbeiten schneller als die entsprechenden Einzelbefehle einer Erweiterung, da der Compiler die entsprechenden Optimierungsmöglichkeiten besitzt.

- Grafikbefehle verwenden als Koordinatenangaben nur ganze Zahlen. Die Berechnung von Koordinaten und ähnlichem sollte also möglichst mit Integer-Variablen erfolgen.

5.7 Andere Erweiterungen

Sollten Sie eine hier nicht erwähnte Basic-Erweiterung besitzen, so kann BASIC 64 in den meisten Fällen auch Befehle dieser Erweiterung compilieren. Sollte die Erweiterung überhaupt mit BASIC 64 zusammenarbeiten können, so gelten für die Compilierbarkeit von Erweiterungen dieselben Kriterien, wie für die anderen Erweiterungen auch. Im Einzelfall muß dies natürlich dann ausprobiert werden. Da der Compiler die verwendete Erweiterung nicht kennt, kann diese auch nicht über die Taste "H" im Entwicklungspaket ausgewählt werden. Es müssen dagegen alle Werte per Hand eingestellt werden. Der Compiler benötigt Speicherende, Anzahl der Bytes pro Befehl und den Code für den ELSE-Befehl. Diese Werte müssen vor Start des Compilers festgestellt werden, soweit dies nicht im Handbuch der Erweiterung vermerkt ist.

Speicherende:

Folgender Befehl berechnet das Speicherende bei aktivierter Erweiterung:

```
PRINT PEEK(55)+256*PEEK(56)
```

Das Speicherende wird über Punkt "E" eingestellt. Bytes pro Token:
Suchen Sie sich einen repräsentativen Befehl der Erweiterung und tippen Sie ein:

```
NEW

10 Befehl
```

Der Befehl muß vollkommen isoliert ohne Werteangaben stehen.

```
PRINT PEEK(2054)
```

Erhalten Sie einen Wert ungleich Null, so müssen Sie mit Punkt "I" zwei Bytes pro Token einstellen.

Else-Code:

Soweit Ihre Erweiterung einen ELSE-Befehl besitzt, muß dieser eingestellt werden:

```
NEW

10 ELSE

PRINT PEEK(2053)
```

Dieser Wert gibt das erste Byte des Befehls. Das zweite Byte ist nur nötig bei Erweiterungen mit zwei Byte pro Befehl:

```
PRINT PEEK(2054)
```

Diese Werte können über Punkt "J" eingegeben werden.

Bei der Benutzung von ELSE muß vor diesen Befehl ein Doppelpunkt gestellt werden.

5.8. Verschiedene Versionen von Basic-Erweiterungen

Einige Befehle aus Basic-Erweiterungen sind nur teilweise implementiert, d.h. die Befehle arbeiten nicht in allen Fällen so, wie es das Handbuch der Erweiterung beschreibt. Hieraus können sich unter Umständen Unterschiede zwischen dem Verhalten von compilierten und nichtcompilierten Befehlen ergeben. Dies trifft insbesondere dann zu, wenn nicht alle Parameter eines Befehles durch Variablen ersetzt werden dürfen. Beispiele für solche Befehle sind MOVE und TEXT im Simons' Basic, bei denen einige Parameter nur Konstanten sein dürfen. Derartige Befehle verhalten sich nach dem Compilieren oder nach dem Einsetzen von Variablen fehlerhaft und sollten

nicht benutzt werden.

Es ist damit zu rechnen, daß in späteren Versionen des Simon's Basic diese Befehle einwandfrei laufen. Möglicherweise sind in früheren Versionen von Basic-Erweiterungen weitere derartige Befehle enthalten.

6. Kapitel

Komplexe Programme

6.1 Compilieren von Overlay-Paketen

Im Basic des Commodore 64 existiert die Möglichkeit, mit Hilfe des Befehles LOAD ein Programm zu laden und zu starten, wobei alle bisherigen Variableninhalte erhalten bleiben. Diese Version des Befehles LOAD wird immer dann ausgeführt, wenn der Befehl innerhalb eines Programmen verwendet wird. Hierbei ergeben sich bei Verwendung des Interpreters allerdings folgende Einschränkungen:
- Zeichenkettenvariablen, die eine im Programm abgelegte Zeichenkette enthalten, gehen durch das Überladen des Programmen verloren, z.B. bei:

```
10 DATA "TEST"
20 READ A$
30 B$= "KETTE"
40 LOAD"OVERLAY",8
```

Das in Zeile 40 geladene und gestartete Programm mit dem Namen "OVERLAY" kann auf die Variableninhalte von A$ und B$ nicht zugreifen, diese sind verloren.

- Das aufrufende Programm muß länger sein, als das aufgerufene, da der Interpreter das aufgerufene Programm sonst verkürzt. Dies hat zur Folge, daß bei Overlaypaketen das zuerst geladene Programm auch das längste sein muß. Dies ist meistens nicht der Fall, aus diesem Grund wird von Basic-Programmierern oft ein Trick angewandt, mit dem das erste Programm sich selber vergrößert. Dabei werden die Speicherstellen 45 und 46 mit POKE geändert und mit CLR die Variablentabelle neu angelegt.

Beim Compilieren von Programmen, die den Overlay-Mechanismus benutzen, ist zu beachten, daß diese Programme nicht auf die gewohnte Art compiliert werden können:

Zum Compilieren eines Programmen aus dem Paket benötigt der Compiler eine Symbol-Tabelle, in der alle benutzten Variablen des Programmpaketes und deren Speicherbelegung aufgeführt sind. Zum Erstellen dieser Symboltabelle dient der Overlay-Pass 1 (nicht zu verwechseln mit dem Compiler-Pass 1).

Overlay-Pass 1:

Compilieren Sie alle Programme des Programmpaketes einzeln. Vor dem Start des eigentlichen Compilers mit "1" oder "2" ist die Taste "4" (Overlay) und danach "1" (Pass 1) zu drücken. Der Compiler erzeugt nun kein fertiges Programm, weshalb er auch schneller arbeitet, es wird lediglich die Tabelle der Variablen generiert und nach jedem Compilerdurchlauf vervollständigt.

Nachdem die Tabelle vollständig ist, können nun die Programme des Overlay-Paketes compiliert werden. Hierzu dient der Overlay-Pass 2.

Overlay-Pass 2:

Compilieren Sie alle Programme des Paketes, drücken Sie aber vor dem Compilieren die Taste "4" (Overlay) und "2" (Pass 2). Der Compiler lädt nun vor dem Compilieren die entsprechende Overlay-Tabelle. Alle Möglichkeiten des Entwicklungspaketes (Menü-Punkt 3) lassen sich weiterhin benutzen. Nachdem alle Programme mit Overlay-Pass 2 compiliert wurden, kann das Paket gestartet werden.

Beim Compilieren von Overlay-Paketen ist weiterhin folgendes zu beachten:

- Die compilierten Programme müssen natürlich noch auf die richtigen Namen umbenannt werden, da der Compiler vor den Namen von compilierten Programmen ein "P-" bzw. "M-" hängt.

- Programme dürfen ihre Länge nicht mit den bereits erwähnten POKE-Befehlen selber einstellen. Dies ist bei compilierten Programmen nicht nur unnötig (der Compiler übernimmt dies), sondern sogar unsinnig, da compilierte Programme meistens eine andere Länge besitzen, als das Original. Derartige POKE-Befehle sollten also vor dem Compilieren entfernt werden.

- Konstante Zeichenketten werden nicht im Programm gespeichert und bleiben nach dem Laden eines anderen Programmes erhalten (anders als beim Interpreter). Dies kann mit Menüpunkt "L" auch wieder abgeschaltet werden.

- Sollen größere Overlaypakete compiliert werden, so müssen die Einzelprogramme auf mehrere Disketten verteilt werden, da die compilierten Programme ja ebenfalls Diskettenplatz benötigen. Der Compiler lädt die Overlaytabelle bereits vor der Eingabe des Programmnamens, die Tabelle kann dadurch von einer anderen Diskette als das zu compilierende Programm geladen werden. Wichtig ist dabei nur, daß in Overlay-Pass 1 die jeweils zuletzt generierte Tabelle geladen wird (letzter Compilerdurchlauf) und in Overlay-Pass 2 ist immer die letzte in Overlay-Pass 1 generierte Tabelle zu laden.

- Der Compiler speichert die Overlay-Tabelle mit dem Namen "S-OVERLAY" auf Diskette ab. Diese Tabelle muß gelöscht werden, wenn ein neues Overlay-Paket compiliert werden soll, da diese Tabelle ansonsten als Grundtabelle für das neue Paket benutzt wird.

- Mit dem Menüpunkt "M" kann ein Befehl an die Floppy gesendet werden. Auf diese Weise können Sie z.B. bereits compilierte Programme löschen (natürlich nur bei Programmen, von denen Kopien existieren), um Platz auf der Diskette zu schaffen. Nach der Ausführung des Befehles wird die Meldung der Floppy angezeigt und muß mit der "Return"-Taste bestätigt werden.

6.2 Fehlerbehandlung

Ein compiliertes Programm gibt dieselben Fehlermeldungen aus, wie dies der Basic-Interpreter tun würde. Natürlich kann nach einer Fehlermeldung das Programm nicht mit GOTO fortgesetzt werden und die vom Programm erarbeiteten Daten sind verloren (Maschinencodeprogramme kann man mit SYS weiterarbeiten lassen, siehe Kapitel 8). Viele Fehlermeldungen lassen sich auch nicht durch Programmänderung vermeiden. So ist es z.B. nicht möglich, festzustellen, ob ein Drucker angeschlossen ist oder ob ein Rechenergebnis einen Wartebereich überschreitet, ohne daß die entsprechende Meldung ausgegeben wird. Aus diesem Grund verfügt BASIC 64 über die Möglichkeit, zu einer bestimmten Basic-Zeile zu verzweigen, anstatt eine Fehlermeldung auszugeben. Die gewünschte Zeile kann mit Punkt "K" im Entwicklungspaket eingestellt werden. Besser ist eine entsprechende Compiler-Anweisung, da dadurch die entsprechende Zeilennummer auch innerhalb eines Programmas angegeben und geändert werder, kann. Die Zeilennummer Null steht hierbei für das Abschalten der Fehlerbehandlung. Beim nächsten Fehler wird dann eine Meldung ausgegeben. Ein typisches Anwendungsbeispiel ist folgender Fall:

```
10 REM@ E1000
20 I=-10
30 PRINT 1/I
40 I=I+1:IF I<10THEN30
50 END
1000 PRINT "FEHLER":REM@ E0
1010 GOT040
```

Der Start einer Basic-Zeile bei Auftreten eines Fehlers hat noch weitere Wirkungen:

Alle FOR und GOSUB werden vom Stapel gelöscht. Die zugehörigen Befehle NEXT und RETURN dürfen nicht mehr ausgeführt werden. Dies hat den Vorteil, daß der Programmierer sich nicht mehr um die Beendung von Schleifen und Unterprogrammen kümmern muß, der Stapel kann durch vergessene GOSUB und FOR also nicht überlaufen.

Die Nummer des Fehlers wird vom compilierten Programm in der Speicherzelle 700 vermerkt, sie kann dort vom Programm mit PEEK ausgelesen werden. Die Fehlernummer hat folgende Bedeutung:

Fehlermeldungen des Betriebssystems:

```
1 too many files
2 file open
3 file not open
4 file not found
5 device not present
6 not input file
7 not output file
8 missing filename
```

9 illegal device number

Basic-Fehlermeldungen:

10 Hext without for
11 syntax
12 return without gosub
13 out of data
14 illegal quantity
15 overflow
16 out of memory
17 undef'd Statement
18 bad subscript
19 redim'd array
20 division by zero
21 illegal direct
22 type mismatch
23 string too long
24 file data
25 formula too complex
26 can't continue
27 undef'd function
29 verify
29 load

Einige Fehlernummern treten nicht auf, da diese Fehler vom Compiler bereits gemeldet werden.
Das Drücken der STOP-RESTORE-Taste stellt ebenfalls einen Fehler dar und wird bei eingeschalteter Fehlerbehandlung ebenfalls abgefangen.
Das Abfangen von Fehlermeldungen funktioniert natürlich erst nach dem Compilieren.

6.3 Das Runtimemodul

Das Runtimemodul enthält alle Routinen, die zur Ausführung des compilierten Programmes benötigt werden. Diese Routinen enthält jedes compilierte Programm. Soll Diskettenplatz gespart werden, so können Sie die Generierung des Runtimemoduls abschalten (Menüpunkt "G"). Beim aufeinanderfolgenden Laden von Programmen braucht nur das erste Programm ein Runtimemodul zu enthalten. Alle weiteren Programme müssen dann absolut mit LOAD"Name",8,1 geladen werden. Bei Overlay-Paketen kann dadurch Ladezeit und Diskettenplatz gespart werden. Ein einzelnes Runtimemodul erhalten Sie, wenn Sie folgendes Programm Compilieren:

10 REM

Runtimemodul und Programm können nun einzeln absolut geladen werden. Ein Programm ohne Runtimemodul ist nur lauffähig, wenn es absolut geladen wird und sich zuvor bereits ein Runtimemodul im Speicher befand.

6.4 Code-Start

Mit dem Menü-Punkt "F" besteht die Möglichkeit, die

Startadresse eines compilierten Programmes zu erhöhen. Der Compiler fügt dann zwischen Runtimemodul und Programmstart entsprechend freien Speicherplatz ein. Natürlich kann das Generieren eines Runtimemoduls auch abgeschaltet werden und das Modul separat geladen werden. Das Programm muß dann allerdings mit "SYS Startadresse" gestartet werden, da die Startadresse verschoben wurde. Eine sinnvolle Adresse für den Code-Start ist die Adresse 16384. In diesem Fall steht der Speicherbereich von 8192 bis 16191 als Grafikspeicherplatz frei und der Bereich von 16192 bis 16383 kann für Sprites benutzt werden.

Programme, die diesen Grafikspeicher benutzen können, sind auch mit dem Basic-Interpreter entwickelbar. Hierzu brauchen Sie nur den Interpreter anzuweisen,
den Code-Start ebenfalls zu verlegen:

```
POKE 44,64:POKE 16384,0:NEW
```

Nun können Sie Ihr Grafik-Programm entwickeln, ausprobieren und anschließend compilieren.

6.5 Unterbrechen von compilierten Programmen

Die Unterbrechung von compilierten Programmen mit der STOP-Taste ist nur selten möglich, z.B. bei der Verwendung von Basic-Erweiterungen oder von einigen Ein-Ausgabebefehlen. Eine Fortsetzung mit CONT ist nicht möglich. Soll verhindert werden, daß ein Programm unterbrochen werden kann, so schaltet folgender Befehl die Stop-Taste ganz ab:

```
POKE 788,PEEK(788)+3
```

Das Drücken der Restore-Taste startet eine Routine in den ROM's des Commodore 64. Da das compilierte Programm diese ROM's zeitweise nicht benutzt und abschaltet, kann das Drücken der Stop-Restore-Taste in seltenen Fällen wirkungslos bleiben und eine weitere Programmausführung verhindern. Dies gilt besonders bei der Verwendung von Basic-Erweiterungen. Das Unterbrechen eines Programmes mit Restore kann abgefangen werden, indem dem Compiler eine Zeile angegeben wird, die bei Drücken der Restore-Taste ausgeführt wird (Menüpunkt "K" - Fehlerbehandlung).

7. Kapitel

Geschwingkeit

7.1 Die Optimierungsstufen

Normalerweise wird das Compilieren mit dem Punkt "1" bzw. der Return-Taste im Hauptmenü gestartet. Es besteht allerdings die Möglichkeit, das Compilieren auch mit dem Punkt "2" zu starten, wie dies in Kapitel 2 bereits erwähnt wurde. Diese beiden Punkte unterscheiden sich lediglich in der Art, wie der Compiler das Programm optimiert.

Optimizer I:

Bei Wahl dieser Optimierungsstufe Werden alle möglichem Optimierungen und Programmveränderungen nur dann durchgeführt, wenn es absolut sicher ist, daß sich am Ablauf des Programmes dadurch nichts ändert. Die Optimierungsstufe 1
ist somit vollständig kompatibel zum Basic-Interpreter.
Berechnungen mit Integer-Werten werden nur dann als ganzzahlige Operationen ausgeführt, wenn sichergestellt ist, daß als Ergebnis nur eine ganze Zahl im Integer-Wertebereich sinnvoll ist, dies ist aber bei den meisten Basic-Operationen der Fall.

Optimizer II:

Diese Optimierungsstufe besitzt mehrere wichtige Unterschiede zur Stufe 1 und zum Basic-Interpreter:

- Alle Variablen, mit Ausnahme der Zeichenkettenvariablen, werden als Integer-Variablen eingestuft, d.h. der Compiler nimmt an, daß hinter jeder Variable ein "%"-Zeichen steht; das compilierte Programm arbeitet entsprechend schneller. Variablen-Felder werden dagegen mit dem richtigen Datentyp eingestuft.

- Die Division zweier Integer-Variablen oder Integer-Werte wird von. Optimierungsstufe 1 immer mit einer Gleitkommadivision ausgeführt. Stufe 2 führt dagegen eine derartige Division als ganzzahlige Operation durch und vernachlässigt die Nachkommastellen, in seltenen Fällen führt dies zu einem Unterschied zum Interpreter. Meistens ist dies nicht der Fall, da bei Operationen mit Integer-Variablen keine Gleitkommaergebnisse gefragt sind. Eine Integer-Division ist wesentlich schneller als eine Gleitkommadivision.

- Die Funktion INT Wird von Stufe 2 nicht als Abschneiden der Nachkommastellen angesehen, sondern als eine Umwandlung in den Datentyp Integer. Die Weiterverarbeitung des entsprechenden Wertes in einer Formel wird mit Integer-Operationen geschehen. Ein Unterschied zum Interpreter ergibt sich dadurch, daß die Funktion INT nicht mehr auf Werte anwendbar ist, die außerhalb des Integer-Bereiches liegen (-32768 bis 32767).

Die Optimierungsstufe 2 hat verschiedene Anwendungsgebiete.

Sie ist für die folgende Art von Programmen gedacht:

- Programme, bei denen es wesentlich stärker auf eine hohe Geschwindigkeit ankommt, als auf die Kompatibilität zum Interpreter.

- Programme, bei denen von Anfang an klar ist, daß alle Variablen nur Integer-Werte aufnehmen.

- Programme, bei denen nicht auf die Verwendung von Integer-Variablen geachtet wurde und bei denen dies nun der Compiler nachholen soll.

Folgendes Beispiel ist eine typische Anwendung der Optimierungsstufe 2:

```
10 A=INT(RND(1)*1000)
```

Die Variable A dient nur zur Aufnahme einer ganzen Zahl, trotzdem wird Gleitkommarechnung eingesetzt (RND(1)*1000), zumindest diese Zeile des entsprechenden Programmen kann mit Optimierungsstufe 2 compiliert werden.

Programme, bei denen die Optimierungsstufe 2 nur dazu eingesetzt wird, um die Verwendung von Integer-Variablen dem Compiler zu überlassen, benötigen meist einige wenige Gleitkommavariablen. Da die Optimierungsstufe 2 diese nicht erkennt, ist es nötig, diese Variablen dem Compiler mit einer Compileranweisung mitzuteilen.

Anweisung:

```
REM@ R-Variable,Variable,...
```

Die aufgezählten Variablen werden von Optimierungsstufe 2 als Gleitkommavariablen eingestuft.

Beispiel:

```
10 FOR I=1 TO 1000
20 A=SQR(I): PRINT A;
30 NEXT
```

Soll dieses Programm mit Optimizer II compiliert werden, so ist folgende Zeile einzufügen:

```
5 REM@ R=A
```

Die Optimierungsstufen können auch innerhalb eines Programmen gewechselt werden:

```
REM@ 01   ;schaltet auf Stufe 1
REM@ 02   ;schaltet auf Stufe 2
```

Das Umschalten der Optimierungsstufe hat nur einen Einfluß auf den Datentyp von Variablen, wenn diese Variablen vor dem Umschalten noch nicht benutzt wurden.
Die Optimierungsstufe 2 hat keinen Einfluß auf den Datentyp

von Feldern. Felder, die zur Aufnahme von ganzen Zahlen gedacht sind, sollten immer mit einem "%"-Zeichen gekennzeichnet werden, da der Basic-Interpreter dadurch sehr viel Speicherplatz spart. Dies macht das Ausprobieren eines solchen Programmes mit dem Interpreter meist erst möglich. Nach dem Compilieren ist der Speicherplatz für Felder sowieso wesentlich größer.

Achtung!

Wenden Sie die Optimierungsstufe 2 nur bei Programmen an, die Sie selber entwickelt haben und bei denen Sie die Arbeitsweise des Programmes und die Verwendung von Datentypen aus diesem Grund kennen. Dies gilt auch für alle anderen zusätzlichen Möglichkeiten des Compilers gegenüber dem Interpreter (z.B. Compileranweisungen).

7.2 P-Code (Speedcode) und Maschinensprache

Sehr große Programme lassen sich meistens nicht vollständig in Maschinensprache übersetzen, da ein Programm, das in Maschinencode übersetzt wurde, immer länger ist als das Original. In größeren Programmen gibt es natürlich auch zeitkritische Programme, bei denen eine Übersetzung in Maschinensprache notwendig ist. Aus diesem Grund verfügt BASIC 64 über die Möglichkeit, den Code-Generator während des Compilierens zu wechseln. Hierzu existieren zwei Compileranweisungen:

REM@ M

Diese Anweisung veranlaßt den Compiler, alle folgenden Programmzeilen in Maschinensprache zu übersetzen. Wird diese Programmzeile durchlaufen, so wird das entsprechende nachfolgende Maschinenprogramm gestartet.

REM@ P

Nach dieser Anweisung werden die folgenden Zeilen wieder in P-Code übersetzt. Beim Durchlaufen dieser Programmzeile wird der P-Code-Interpreter gestartet und die folgenden Befehle wieder interpretiert.

Diese beiden Compileranweisungen sollten Sie nur einsetzen, wenn Sie deren Wirkungsweise vollständig verstanden haben. Beim Einsatz dieser Befehle ergeben sich nämlich folgende Einschränkungen:

- Der Mikroprozessor des Commodore 64 kann nur Maschinencode ausführen und keinen P-Code.

- Der P-Code-Interpreter kann nur P-Code interpretieren und keine Maschinensprache.

Sollte der Code wechseln, ohne daß eine entsprechende Umschaltstelle (Compileranweisung) durchlaufen wird, so ist das Verhalten des Programmes nicht vorhersagbar, auf keinen Fall arbeitet es korrekt weiter. Demzufolge dürfen keine

Programmsprünge innerhalb eines Programmen erfolgen, die in einem Programmteil mit einem anderen Code enden. Programmsprünge führen lediglich die Befehle GOTO, GOSUB, RETURN und meistens auch NEXT durch.

Das Umschalten des Code-Generators kann in blockstrukturierten Programmen gefahrlos an Anfang und Ende eines Blockes erfolgen. Ein Programmblock ist dadurch gekennzeichnet, daß in ihn weder hineingesprungen noch aus ihm hinausgesprungen wird. Er wird an seinem Anfang gestartet und an seinem Ende wieder verlassen. Größere Programme sind meistens blockstrukturiert, ein teilweises Umschalten auf Maschinensprache ist also möglich. Bei Programmen, die weder blockstrukturiert sind noch irgend eine andere erkennbare Struktur besitzen, sollte auf das Umschalten des Code-Generators innerhalb des Programmes verzichtet werden.

Besonders sinnvoll und einfach ist das Umschalten des Code-Generators bei Unterprogrammen. Ein derartiges Unterprogramm könnte z.B. folgende allgemeine Form haben:

```
1000 REM@ M
...
1980 REM@ P
1990 RETURN
```

Die Zeilen von 1001 bis 1979 werden bei diesem Beispiel vom Compiler in Maschinencode übersetzt. Das Unterprogramm darf nur mit GOSUB1000 gestartet werden und nur mit dem Durchlaufen der Zeilen 1980 und 1990 wieder verlassen werden (notfalls mit GOT01980). Bei strukturierten Programmen ist dies sowieso der Fall. Soll innerhalb des Unterprogrammes ein anderes Unterprogramm aufgerufen werden, so ist zuvor der Code-Generator wieder umzuschalten. Im Beispielprogramm würde dies so aussehen:

```
1500 REM@ P
1510 GOSUB5000
1520 REM@ M
```

Hierbei ist es natürlich egal, welchen Code-Typ das aufgerufene Unterprogramm besitzt, solange wenigstens die erste Zeile (mit der Compileranweisung) in P-Code übersetzt wurde. Meistens rufen zeitkritische Unterprogramme keine weiteren Unterprogramme auf.

7.3 Integer-Wertebereich beim Poke-Befehl

Die vielen Möglichkeiten des Commodore 64 sind in Basic nur mit Hilfe der Befehle PEEK und POKE nutzbar. Diese Befehle sollten also möglichst schnell ausgeführt werden. Der Einsatz von Integer-Variablen ist in Kombination mit dem POKE-Befehl natürlich besonders sinnvoll, da Speicheradressen durch ganze Zahlen dargestellt werden. Es gibt genau 65536 verschiedene Speicheradressen im Commodore 64 (0-65535) und eine Integer-Variable kann ebenfalls 65536 verschiedene Zahlenwerte enthalten (-32768 bis 32767). Leider sind die entsprechenden Zahlenbereiche nicht identisch. Für

Speicheradressen über 32767 müssen somit Gleitkommawerte und Gleitkommavariablen benutzt werden. In Verbindung mit dem POKE-Befehl besitzt der Compiler die Möglichkeit, Integer-Operationen auch für Werte über 32767 auszuführen, allerdings unter Verzicht auf die negativen Zahlen. Der Benutzer des Compilers bemerkt dies nur durch eine höhere Programmgeschwindigkeit.

```
5 REM@I=I
10 FOR I=1024 TO 2023
20 POKE I,65
30 POKE I+54272,0
40 NEXT
```

Schneller wird das Programm nun durch folgende Änderung:

```
6 OF%=30000
30 POKE I+OF%+24272,0
```

Diese Änderung bewirkt, daß nur noch Integer-Werte verarbeitet werden und daß eine Gleitkommaaddition zu zwei Integer-Additionen wird. Derartige Tricks sollten allerdings nur an extrem zeitkritischen Programmstellen angewendet werden.

8. Kapitel

Speicheradressen und Maschinensprache

Dieses Kapitel sollten Sie unbedingt durchlesen, wenn Ihre Basic-Programme mit Assembler-Programmen zusammenarbeiten sollen oder wenn Sie in Ihren Programmen viele POKE-Befehle benutzen.

8.1 Speicherbelegung

Die Speicherbelegung von compilierten Programmen ergibt sich aus den vom Compiler ausgegebenen Werten:

0 - 1024	Systemspeicher
1024 - 2023	Bildschirmspeicher
2048 - Codestart	Runtimemodul
Codestart - Stringsanfang	Programm-Code
Stringsanfang - Stringsende	Freier Speicher für Zeichenketten
Stringsende - Memory-Top	Speicher für Variablen und Felder

Das compilierte Programm legt die benutzten Variablen in der folgenden Form ab:
Integer-Variablen:

2 Bytes: low Byte, high Byte

Gleitkomma-Variablen:

5 Bytes: Exponent, 4-Byte Mantisse

Zeichenketten-Variablen:

3 Bytes: Länge, Adresse der Zeichenkette low Byte, Adresse high Byte

Zeichenketten:

2 Bytes plus 1 Byte pro Zeichen

Ein Assemblerprogramm kann die Inhalte der Variablen ändern. Die Adressen der Variablen erhält man über die Symbol-Tabellen. Bei Zeichenketten darf weder die Lage der Zeichenkette noch deren Länge von einem Assemblerprogramm
geändert werden, dies kann nur das compilierte Basic-Programm.

8.2 Speicheradressen

Bei der Verwendung von POKE- und PEEK-Befehlen ist darauf zu achten, daß das compilierte Programm einige Speicherzellen anders benutzt als der Interpreter. Für die Verwendbarkeit von Poke-Befehlen gilt folgendes%

- Der Bereich von 53248 bis 57343 ($D000-$DFFF) ist wie beim Basic-Interpreter ansprechbar.

- Der Bildschirmspeicher liegt weiterhin von 1024-2023.

- Alle Speicherstellen von 144 bis 828, die vom Betriebssystem benutzt werden, behalten ihre Funktion (Tastaturpuffer, gedrückte Taste, IO-Vektoren, etc.).

Änderungen ergeben sich lediglich bei folgenden Speicherzellen:

Der Cassettenpuffer wird vom compilierten Programm benutzt. Soll der Cassettenpuffer z.B. für Sprites benutzt werden, so kann er mit einer Compileranweisung freigegeben werden:

```
REM@ S Adresse
```

Der Cassettenpuffer wird nun unterhalb dieser Adresse vom compilierten Programm nicht benutzt. Zur Freigabe des gesamten Puffers dient folgender Befehl:

```
REM@ S1024
```

Der Bereich von $2C0 bis $2FF wird bei der Verwendung von Basic-Erweiterungen mit Daten belegt, ansonsten ist dieser Bereich frei.

Speicherzellen, die der Basic-Interpreter zur Interpretation des Programmes benutzt, verlieren ihre Bedeutung teilweise. Erhalten bleiben nur die Speicherzellen, die sich normalerweise mit POKE-Befehlen sinnvoll benutzen lassen. Hierzu gehören:

- Gleitkommaregister
- Zufallszahl
- Chrget-Routine
- Statuswort ST
- RND-Wert
- Zeiger auf Programmstart (43-44)
- Zeiger Auf Programmende (45-46)
- Zeiger auf Beginn des Stringstapels (51-52)
- Zeiger auf nächstes DATA-Element (65-66)

So ist es z.B. möglich, das Programmende (45-46) zu erhöhen und in dem freigewordenen Bereich Assembler-Programme, Daten oder compilierte Basic-Programme mit verschobenem Code-Start zu speichern.

Bei der Verwendung von Adressenspeicher (2-Bytes) ist darauf zu achten, daß in diesen Speicherzellen möglicherweise andere Adressen gespeichert werden, als bei Verwendung des Interpreters. Dies gilt besonders für die Speicherverwaltung (43-56) und für die Fehlerbehandlung (768-769).

Eine neue Bedeutung bekommt die Speicherstelle 64, sie wird für den Strukturstapelzeiger benutzt:

```
10 A%=PEEK(64):REM Stapelzustand merken
```

```
20 FORI=ITO1000
30 POKE64,A%:REM Stapelzustand wiederherstellen
40 NEXT:REM ergibt NEXT WITHOUT FOR
```

Der Bereich von $A000-$FFFF wird vom compilierten Programm normalerweise mit Variablen belegt. Die ROMs von $A000-$BFFF und $E000-$FFFF sind trotzdem zugreifbar. Soll im Bereich von $C000-$CFFF ein Maschinenprogramm, ein Zeichengenerator, etc. gespeichert werden, so kann die niedrigste vom compilierten Programm nicht mehr zu benutzende Adresse dem Compiler als Speicherende mitgeteilt werden. Dies geschieht über Menüpunkt "E". Zur Freigabe des Bereichs ab $C000 müßte der Wert für Memory-Top auf 49152 gesetzt werden.

8.3 Spezielle Befehle

SYS:

Oft wird der Befehl SYS in der folgenden Form benutzt:

SYS Adresse,Wert,Wert...

Obwohl der Basic-Interpreter die nachfolgenden Werte nicht berechnet, meldet er keinen Fehler, wenn das aufgerufene Assemblerprogramm dies macht. Eine derartige Erweiterung des SYS-Befehls wird vom Compiler wie eine Basic-Erweiterung behandelt. Es gilt das in Kapitel 6 zur Compilierbarkeit von beliebigen Basic-Erweiterungen Gesagte.

STOP:

Dieser Befehl bewirkt ein Beenden des compilierten Programmes. Ähnlich wie bei der Ausgabe einer Fehlermeldung kann der Befehl CONT nicht benutzt werden. Bei der Unterbrechung mit STOP oder durch eine Fehlermeldung wird eine Speicheradresse ausgegeben. Handelt es sich bei dem compilierten Programm um Maschinencode, so kann an dieser Speicheradresse das Programm mit SYS fortgesetzt werden. Eine Erhaltung aller Daten und ein Weiterlaufen des Programmes ist allerdings nicht immer garantiert, insbesondere bei echten Fehlermeldungen.

LOAD:

Der Befehl LOAD besitzt in compilierten Programmen eine weitere Möglichkeit:

LOAD"Name",8,128

Die entsprechende Datei wird in den Speicher an die Stelle geladen, von der sie abgespeichert wurde (z.B. mit einem Maschinensprachemonitor). Im Gegensatz zum Laden mit ",8,1" wird das Programm nicht neu gestartet, sondern nach dem LOAD-Befehl fortgesetzt. Dies ist besonders nützlich zum Laden von Assemblerprogrammen, Grafiken, etc.

8.4 Symbol-Tabellen als Schnittstelle zu Profimat

BASIC 64 verfügt über die Möglichkeit, Symboltabellen zu laden und abzuspeichern. In einer Symboltabelle sind alle Variablen eines Programmes aufgeführt und die Adressen genannt, an denen sie abgelegt sind. Die Symboltabellen selbst werden in der Form abgespeichert, wie sie der Compiler intern verarbeitet.

Abspeichern einer Symboltabelle:

Zum Abspeichern einer Symboltabelle braucht dem Compiler nur ein Name für die Tabelle genannt zu werden. Dies geschieht über Menüpunkt "C". Der Name "OVERLAY" ist für Overlaysymboltabellen reserviert. Das Abspeichern einer Symboltabelle erfolgt nach dem Compilieren.

Laden einer Symboltabelle:

Mit Menüpunkt "B" kann der Name einer zu ladenden Symboltabelle angegeben werden. Das Laden der Tabelle geschieht vor dem Compilieren. Alle in der Symboltabelle genannten Variablen und Speicheradressen werden übernommen. Dies ist nützlich, wenn mehrere Programme auf gemeinsame Variablen zugreifen sollen, z.B. um dieselben Assemblerprogramme benutzen zu können. Beim Compilieren von Overlay-Paketen macht der Compiler hiervon Gebrauch.

Verarbeitung einer Symboltabelle:

Auf der BASIC 64 Diskette befindet sich das Programm SYMBOL. Nach dem Laden und Starten des Programmes fragt dieses nach dem Dateinamen der Symboltabelle. Nach der Eingabe des Namens können Sie zwischen zwei Möglichkeiten wählen.

Das Drücken der Taste "1" veranlaßt das Programm dazu, die Symboltabelle in das Format des Assemblers Profi-Ass umzuwandeln. Der Name der umgeformten Symboltabelle ist identisch mit dem Namen der vom Compiler erzeugten Tabelle. trotzdem wird die alte Tabelle nicht gelöscht, da der Compiler ein "S-" vor den Namen der Tabelle gehängt hat. Dieses braucht nicht beachtet zu werden. Der Assembler Profi-Ass kann Symbol-Tabellen mit dem Pseudo-Opcode ".LST" laden (siehe Profi-Ass-Handbuch). Die Namen der Variablen kann ein Assemblerprogramm nun benutzen, als ob sie im Programm definiert wären. Dadurch ist es möglich, daß ein assembliertes Programm auf Variablen eines compilierten Programmes zugreift. Zur Unterscheidung der einzelnen Datentypen untereinander und zu Symbolen des jeweiligen Assemblerprogrammes formt das Programm Symbol die Namen von Variablen um, im folgenden Beispiel steht "na" für die ersten beiden Bytes des Variablennamens:

Einzelvariablen

```
na   - na
na%  - naIN
na$  - naST
```

Felder:

```
na   - ARna
na   - ARnaIN
na   - ARnaST
```

Die Darstellung ist nicht abhängig von der Optimierungsstufe. Sie wird direkt aus dem Variablennamen abgeleitet. Folgender Ausschnitt aus einem Assemblerprogramm vertauscht zwei Basic-Integer-Variablen:

```
100 LST 8,2,"Name,S,R"
110 ...
1000 LDA AIN
1010 LDX BIN
1020 STX AIN
1030 STA BIN
1040 LDA AIN+1
1050 LDX BIN+1
1060 STX AIN+1
1070 STA BIN+1
1080 ...
```

Dies entspricht ungefähr folgendem Basic-Programm:

```
100 H%=A%:A%=B%:B%=H%
110 ...
```

Sollten Sie keinen Assembler besitzen, so können Sie die Symboltabellen trotzdem verarbeiten. Drücken Sie dazu nach Start des Programmen Symbol und Eingabe des Filenamens die Taste "2" zur Ausgabe eines Listings. Das Programm fragt nun nach der Gerätenummer, unter der das Listing ausgegeben werden soll. Sinnvoll sind folgende Geräte-Adressen:

```
0 = Device 0 = Bildschirm
4 = Device 4 = Drucker
8 = Device 8 = Sequentielle Datei auf der Floppy.
```

Das Programm gibt nun die Namen der Variablen, gefolgt von der Speicheradresse, aus.

8.5 Zugriff auf Speicherbänke

Der Commodore 64 besitzt 64K Ram und 20K ROM sowie einige weitere Speicherzellen (Register, Farbspeicher, etc.). Mit dem Basic-Befehl POKE bzw. PEEK sind dagegen nur 64K Speicheradressen zugänglich. Allerdings läßt sich die Aufteilung dieses Speicherbereiches umschalten:

```
POKE 1,52   ;Der gesamte Speicher wird auf RAM geschaltet,
der Bereich von $A000 - SFFFF ist dadurch zugreifbar.

POKE 1,51   ;Der Zeichengenerator ist im Bereich von $D000-$DFFF
zugreifbar.

POKE 1,55   ;Stellt den Einschaltzustand wieder her.
```

Normalerweise muß in der Speicherzelle 1 immer der Wert 55 stehen, da der Basic-Interpreter sonst abgeschaltet wäre und dadurch nicht arbeiten könnte. Compilierte Programme können dagegen auch ohne Basic-ROMs laufen. Dabei sind einige Einschränkungen zu beachten:

- Vor dem Ändern der Speicherzelle 1 muß der Interrupt abgeschaltet werden. Dies geschieht mit:

```
POKE 56334,PEEK(56334)AND254
```

Nachdem in die Speicherzelle 1 wieder der Wert 55 gePOKEd wurde, sollte der Interrupt wieder eingeschaltet werden:

```
POKE 56334,PEEK(56334)0R1
```

- Während die Speicherzelle 1 einen anderen Wert als 55 enthält, dürfen keine Ein-Ausgabebefehle benutzt werden.

- Weiterhin dürfen in dieser Zeit keine Gleitkommaoperationen durchgeführt werden, es ist also mit Integer-Variablen und Integer-Werten zu arbeiten. Beachten Sie, daß POKE-Adressen über 32767 nicht direkt mit Integer-Variablen ansprechbar sind (Kapitel 7).

Durch die Zugriffsmöglichkeit auf den Zeichengenerator kann dieser z.B. kopiert und danach verändert werden. Der Speicherbereich von $E000-$FFFF könnte z.B. für Grafik benutzt werden und vieles mehr. Bei der Benutzung von Speicher ist dies dem Compiler• mitzuteilen, indem das Speicherende herabgesetzt wird.

9. Kapitel

9.1 Übersicht über die Möglichkeiten von BASIC 64 und deren Anwendung

Optimierungsstufe 1:

Durch Drücken der Taste "1" im Hauptmenü oder durch "Return" starten Sie den Compiler mit der Optimierungsstufe 1. Diese Stufe ist vollständig kompatibel zum Basic-Interpreter V2.0. Alle durchgeführten Programmoptimierungen verändern in keinem Fall den Ablauf oder das Verhalten des Programmas und dienen nur dem Erreichen einer höheren Geschwindigkeit. Um in Stufe 1 die Geschwindigkeit von Berechnungen mit ganzen Zahlen (INTEGER) nutzen zu können, müssen Variablen, die nur ganze Zahlen aufnehmen sollen, entsprechend gekennzeichnet werden. Im Commodore-Basic geschieht dies durch Anhängen eines Prozentzeichens (%) an den Programmnamen.

Optimierungsstufe 2:

Die Stufe 2 wird durch Drücken der Taste "2" gewählt und startet ebenfalls den Compiler. Sie unterscheidet sich von der Stufe 1 dadurch, daß alle Variablen als Integer-Variablen angesehen werden, auch bei Variablen, bei denen dies der Basic-Interpreter nicht macht. Sollten trotzdem einige Variablen Nachkommastellen aufnehmen müssen, so sind diese mit einer entsprechenden Compileranweisung zu kennzeichnen.
Weiterhin führt die Stufe 2 Optimierungen mit der Integer-Division und der Integer-Umwandlung (INT) durch, die nicht in allen Fällen kompatibel zum Interpreter sind.

Das Einstellen von Compilerparametern:

Bei Start des Compilers mit den Tasten "1" oder "2" besitzt der Compiler eine Liste von Werten und Anweisungen, die er zum Compilieren benötigt. Um diese Liste anzuzeigen, drücken Sie im Hauptmenü die Taste "3". Auf dem Bildschirm erscheint nun die Liste. Diese können Sie nun ändern und danach mit "Return" wieder ins Hauptmenü gelangen. Vor die einzelnen Punkte der Auflistung sind Buchstaben gestellt, die Sie nur zu drücken brauchen, um einen Punkt zu ändern:

Punkt "A" - Code Generator:

Der Compiler kann wahlweise P-Code, Maschinensprache (6502/6510) oder gar keinen Code erzeugen. Maschinenprogramme erhalten ein "M-" vor den Namen, andere Programme ein "P-". Die vom Compiler vorgegebene Einstellung ist der P-Code-Generator.

Punkt "B" - Laden von Symboltabellen:

Hiermit kann der Compiler eine Variablenliste laden, die beim Compilieren eines anderen Pragrammes abgespeichert wurde. Dies hat zur Folge, daß beide Programme dieselben Adressen für die entsprechenden Variablen benutzen. (Overlay-Pakete und ähnliche Anwendungen)

Punkt "C" - Abspeichern von Symboltabellen:

Abgespeicherte Symboltabellen können beim Compilieren von anderen Programmen geladen oder vom Assembler Profi-Ass zur Addressenbelegung von Labels benutzt werden. Dies ist nützlich, wenn ein Basic-Programm mit SYS Unterprogramme in Maschinensprache aufruft.

Punkt "D" - Generieren einer Zeilenliste:

Nach Compilieren eines Programes speichert der Compiler wahlweise eine Zeilenliste des Programmes ab. Diese Liste kann mit LOAD"Z-Programmname" geladen und mit LIST gelistet werden. Jeder Basic-Zeile (rechte Seite) ist eine Speicheradresse (linke Seite) zugeordnet (für Fehlermeldungen oder Start eines Programmteiles mit SYS).

Punkt "E" - Speicherende:

Dieser Wert gibt die erste nicht verfügbare Speicheradresse an. Da der Commodore 64 über 64 K Ram verfügt, benutzt der Compiler als Speicherende den Wert 65536. Sie können diesen Wert aber auch herabsetzen, wenn Sie Speicher für Maschinenprogramme, Basic-Erweiterungen, etc. benötigen. Der Basic-Interpreter benutzt als Speicherende übrigens den Wert 40960, wodurch 24576 Bytes ungenutzt bleiben.

Punkt "F" - Code Start:

Der generierte Code wird normalerweise vom Compiler direkt hinter das Runtimemodul gelegt. Sie können diesen Wert aber auch heraufsetzen, um nach dem Runtimemodul andere Daten zu speichern oder gar mehrere Programme im Speicher unterzubringen. Derartig verschobene Programme können nicht mit RUN gestartet werden, sondern mit SYS Startadresse, da RUN nur das erste Programm im Speicher startet.

Punkt "G" - Generieren eines Runtimemoduls:

Sie können.das automatische Einbinden des Runtimemoduls (5K) abschalten und dann Modul und Programm einzeln von Diskette laden. Dies spart Diskettenplatz und bei Overlaypaketen Ladezeit.

Punkt "H" - Einstellen der verwendeten Basic-Erweiterung:

Zur Auswahl stehen die Basic-Erweiterungen Supergrafik 64 Plus, Supergrafik 64, Simons' Basic, Exbasic Level II und Disk-Basic (Basic 4.0 enthalten in Master 64 und Diskomat). Alle weiteren notwendigen Parameter stellen sich dann automatisch ein. Sollten Sie eine andere Basic-Erweiterung besitzen, so stellen Sie "others" ein und wählen Sie die übrigen Parameter selbst.

Punkt "I" - Anzahl der Bytes pro erweitertem Basic-Befehl:

Außer dem Simons' Basic verwenden alle Erweiterungen ein Byte. Dieser Punkt wird aber automatisch eingestellt.

Punkt "J" - Code für den Befehl ELSE:

In vielen Erweiterungen ist der wichtige Befehl ELSE vorhanden. Der Compiler muß den Code für diesen Befehl kennen. Dieser Wert wird automatisch eingestellt.

Punkt "K" - Fehlerabbruch-Zeile:

Bei Auftreten eines Fehlers in einem Basic-Programm während der Laufzeit wird normalerweise eine entsprechende Meldung ausgegeben. Sie können dies abschalten, indem Sie dem Compiler mitteilen, welche Zeile bei Auftreten eines Fehlers ausgeführt werden soll. Zeile 0 bedeutet Ausgabe einer Meldung.

Punkt "L" - Overlay-Verhalten:

Der Basic-Interpreter speichert Zeichenkettenkonstanten direkt im Programm. Bei Overlay-Paketen ist dies sehr störend. Damit das compilierte Programm sich anders verhält, können Sie dies dem Compiler mitteilen.

Punkt "M" - Befehlskanal der Floppy:

Mit Hilfe dieses Punktes können Sie Befehle an die Floppy senden und die Fehlermeldung auslesen, welche mit "Return" bestätigt werden muß.

Compilieren von Overlayprogrammen:

Programme, die sich gegenseitig laden und starten, sollen meistens auch dieselben Variablen bzw. deren Inhalt benutzen. In Basic wird eine entsprechende Speicherverwaltung normalerweise mit Hilfe von Poke-Befehlen erreicht. Bei compilierten Programmen übernimmt der Compiler diese Aufgabe.
Das Compilieren von Overlay-Paketen geschieht in zwei Durchgängen:

Durchgang 1:

Compilieren Sie alle Programme des Overlay-Paketes. Drücken Sie dabei gleich nach der Aktivierung des Compilers "4" (Overlay) und danach "1" (Durchgang 1). Beim Durchgang 1 wird kein Programm erzeugt, sondern nur eine Tabelle, der Durchgang 1 ist deswegen relativ schnell beendet.

Durchgang 2:

Compilieren Sie nun alle Programme nochmals, wobei Sie diesmal "4" und "2" (Durchgang 2) drücken. Der Compiler generiert nun das gewünschte Programm, wobei Sie mit Hilfe der Taste "3" im Hauptmenü evtl. Parameter verändern können.

Sollte der Diskettenspeicherplatz nicht ausreichen, so können Sie natürlich auch die Originalprogramme jeweils nach deren Compilierung löschen (Punkt "M" im Untermenü).
Um ein Overlaypaket zu starten, müssen natürlich noch die Namen der compilierten Programme so geändert werden, daß die

Programme sich gegenseitig laden können.

Compileranweisungen:

Oft ist es nötig, auch während der Compilierung noch Einfluß auf den Compiler zu nehmen. Dies geschieht mit Hilfe von speziell gekennzeichneten REM-Anweisungen innerhalb des Programmes:

Umschalten auf Generieren eines Maschinencodes: Format:

REM@ M

Wirkung:

Der Compiler erzeugt ab sofort Maschinensprache. Außerdem wird in das Programm eine Anweisung eingefügt, welche während der Laufzeit dem P-Code-Interpreter mitteilt, daß nun ein Maschinenprogramm folgt.

Zurückschalten auf P-Code:

Format:

REM@ P

Wirkung:

Der Compiler generiert wieder P-Code. Zusätzlich wird in das Programm ein Befehl eingefügt, welcher den P-Code-Interpreter aktiviert.

Ein- und Ausschalten von Fehlerbehandlung:

Format:

REM@ EZeilennummer

Wirkung:

Dies entspricht der Wirkung des Punktes "K" im Untermenü des Compilers. Allerdings wird die Fehlerbehandlung erst aktiviert, wenn die entsprechende Zeile durchlaufen wird. Die Zeilennummer 0 schaltet wieder auf normale Fehlermeldungen. Diese Möglichkeit entspricht dem Befehl ON ERROR GOTO in einigen Basic-Erweiterungen, ist aber auch ohne Erweiterung einsetzbar.

Deklarieren von Integervariablen:

Format:

REM @I=Variable,Variable...

Alle genannten Variablen (Gleitkommavariablen ohne %-Zeichen)

werden vom Compiler als Integer-Variablen eingestuft, was eine schnellere Programmausführung zur Folge hat. Zusätzlich kann man hiermit Integervariablen innerhalb von FOR-NEXT-Schleifen einsetzen, was der Basic-Interpreter normalerweise nicht zuläßt. Dieser Befehl ist nur in Verbindung mit Optimierungsstufe 1 sinnvoll.

Deklarieren von Gleitkommavariablen:

Format:

REM@ R=Variable,Variable...

Alle genannten Variablen werden als Gleitkommavariablen eingestuft, was natürlich nur bei Anwendung von Optimierungsstufe 2 nötig ist.

Umschalten der Optimierungstufe:

Format:

REM@ OStufennummer

Wirkung:

Die Optimierungsstufe wird umgeschaltet. Dies hat nur Auswirkungen auf den Datentyp von Variablen, wenn diese im Programm bis zu dieser Stelle nicht erwähnt wurden.

Zuordnen von Variablenadressen:

Format:

REM @AVariable=Adresse

Wirkung:

Die Variable wird an die angegebene Adresse gelegt. So kann man z.B. eine Integervariable in ein Spritesteuerregister legen und das entsprechende Sprite damit extrem einfach und schnell steuern. Adressen unter 768 sind nicht erlaubt.

Freigabe des Bandpuffers:

Format:

REM@ SAdresse

Wirkung:

Der Compiler legt innerhalb des Bandpuffers automatisch alle Variablen ab, auf die besonders schnell und effektiv zugegriffen werden soll. Sie können diesen Bereich für Ihre eigenen Zwecke benutzen, indem Sie die Adresse angeben, bis zu der Sie den Randpuffer benötigen.

Beispiel:

Dies gibt den Bandpuffer vollständig frei.

Compilieren von Basic-Erweiterungen:

Obwohl Sie sich viele Basic-Erweiterungen durch Unterprogramme selber programmieren können, wo dies bisher aus Geschwindigkeitsgründen nicht möglich war, so ist die Anwendung von neuen Basic-Befehlen natürlich wesentlich einfacher und bei Grafik-Befehlen auch meistens notwendig.

BASIC 64 ist in der Lage, die meisten Befehle aus beliebigen Basic-Erweiterungen zu compilieren und in das Programm einzubinden. Um ein Programm zu compilieren, das Befehle aus einer Basic-Erweiterung benutzt, brauchen Sie nur vor dem Starten des eigentlichen Compilierens mit dem Punkt "1" im Hauptmenü folgendes auszuführen:

- Bei Laden des Compilers darf die Basic-Erweiterung nicht aktiviert sein (evtl. Computer aus-einschalten oder Modul ziehen).

- Wählen Sie das Untermenü mit Hilfe der Taste "3" an.

- Drücken Sie so lange die Taste "H", bis die von Ihnen verwendete Basic-Erweiterung erscheint.

- Drücken Sie "Return".

- Starten Sie den Compiler mit "Return", "1" bzw. "2".

- Damit das compilierte Programm lauffähig ist, muß die Erweiterung aktiviert sein.

Bei der Verwendung von Befehlen aus Basic-Erweiterungen sind nicht alle Befehle mit der Speicher- und Programmorganisation des Compilers verträglich. Es ist folgendes zu beachten:

Supergrafik 64 und Supergrafik 64 Plus:

Die meisten Befehle und Sekundärbefehle der Supergrafik werden unterstützt mit Ausnahme der folgenden Befehle:

SREAD, IF#, IRETURN und Programmierhilfen.

Zumindest die Befehle SREAD und IF# können leicht durch normale Basic-Befehle ersetzt werden. Programmierhilfen sind innerhalb von Basic-Programmen sowieso nicht einsetzbar und deswegen auch nicht compilierbar (Renumber, etc.).

Simons' Basic:

Die meisten Befehle und Funktionen werden unterstützt, mit Ausnahme von einigen Toolkitbefehlen und Programmstrukturen. Funktionen und alle sinnvoll in Programmen einsetzbaren Befehle, wie z.B. die Grafikbefehle,. werden unterstützt.

Exbasic Level II:

Wie bei allen anderen Erweiterungen werden keine Programmierhilfen und nicht alle Programmstrukturbefehle unterstützt. Trotzdem können die Toolkitbefehle natürlich zur Programmentwicklung eingesetzt werden.

Basic 4.0 (Master 64, Diskomat):

Diese Wahlmöglichkeit ist für Basic-Erweiterungen gedacht, die eine komplizierte Syntax und Tokenhandhabung besitzen. Um derartige Befehle verwenden zu können, müssen alle zu berechnenden Befehlsparameter in Klammern gesetzt werden, wie dies z.B. beim Basic 4.0 sowieso vorgeschrieben ist. Unterstützt werden Befehle des Basic 4.0.

Ausnutzung der vielfältigen Optimierungsmöglichkeiten von BASIC 64:

Obwohl das Compilieren Ihr Programm wesentlich beschleunigt, läßt sich die Geschwindigkeit von Programmen nochmals steigern, wenn man bereits beim Schreiben darauf achtet, welche Operationen nach dem Compilieren besondes schnell ausgeführt werden. Meistens genügt aber die Beachtung folgender Regeln, um eine hohe Geschwindigkeit zu erreichen:

- Erfahrungen, die Sie mit dem Basic-Interpreter bezüglich der relativen Geschwindigkeit von einzelnen Operationen gemacht haben, gelten für compilierte Programme nicht mehr.

- Operationen mit Integer-Variablen arbeiten wesentlich schneller als mit Gleitkommavariablen. Erfahrungsgemäß ist der Großteil aller Variablen in den meisten Programmen durch Integer-Variablen ersetzbar (Werte zwischen -32768 und +32767 ohne Nachkommastellen). Integer-Variablen können Interpreter und Compiler durch Anhängen eines "%"-Zeichens an den Namen angezeigt werden. Weiterhin kann man dem Compiler dies mit Hilfe von REM-Anweisungen oder mit der Optimierungsstufe 2 mitteilen.

Integer-Werte werden z.B. in folgenden Funktionen und Anwendungen ausschließlich benutzt:

Schleifenzähler, Felderindex, Poke, Peek, On, Wait, Dateiparameter bei Open, Sys, Tab, Spc, Fre, Zahlenparameter bei Zeichenkettenfunktionen, Asc, Chr$, logische Operationen, oft auch bei Vergleichen und und vielen weiteren Anwendungen.

In all diesen Fällen ist die Verwendung von Gleitkommavariablen unsinnig und langsamer. Bei Verwendung des Basic-Interpreters ist dies nicht spürbar, da dieser alle Berechnungen mit Gleitkommazahlen ausführt.

- Zeichenkettenoperationen werden von compilierten Programmen anders ausgeführt als vom Interpreter. Dadurch sind besonders bei komplexeren Zeichenkettenformeln hohe Geschwindigkeiten erreichbar.

- Das Aufräumen des Speichers bei Zeichenkettenoperationen (Garbage Collection) dauert maximal 1 Sekunde. Der Basic-Interpreter ist dagegen durch die Garbage-Collection meist minutenlang blockiert. Besonders bei häufiger Benutzung von Zeichenkettenverarbeitung wird das Compilieren dadurch unvermeidbar.

- Der Compiler nimmt dem Programm viele Arbeiten ab, auf diese braucht somit nicht geachtet zu werden, wie z.B. Auffinden von Sprungzielen bei GOTO,GOSUB, Interpretieren von Befehlen und Formeln, Syntaxprüfung, Dezimal-Gleitkomma-Integer Umwandlung, Auffinden von Variablen, Umformen und Optimieren von Formeln, Berechnung von Operationen mit konstanten Werten und konstanten Zeichenketten, etc.

10. Kapitel

Weitere Anwendungen

10.1 Ein-Ausgabe

Peripheriegeräte, wie z.B. Floppy und Drucker, arbeiten nach dem Compilieren eines Programmas natürlich genauso langsam wie zuvor. Beim Abspeichern und Laden von Daten mit der Floppy ist es deswegen empfehlenswert, die zur Verfügung stehende Geschwindigkeit der Floppy maximal auszunutzen. Vor und nach jedem Basic-Befehl, der die Floppy anspricht, sendet das Betriebssystem einen Steuercode. Die Übertragung dieses Codes belegt ebenfalls Zeit. Soll dies vermieden werden, so müssen möglichst große Datenmengen mit einem Befehl übertragen werden:

```
90 REM@ I=I
100 FOR I=1 TO 100
110 PRINT#1,CHR$(I);
120 NEXT
```

Dieser Programmausschnitt könnte z.B. auch so lauten:

```
90 REM@ I=I
100 FOR I=1 TO 100
110 PRINT#1,CHR$(I);CHR$(1+1);
120 I=1+1:NEXT
```

Besonders langsam arbeitet der Befehl GET#, da er nur einzelne Bytes einliest. Zum Senden von Daten sollte deshalb alle 80 Zeichen ein Zeilenendekennzeichen gewählt werden, solange die Art der Daten dies zuläßt.

```
90 REM@ I=1
100 A$="":FOR I=1 TO 80
110 GET#1,B$:A$=A$+B$
120 NEXT
```

Bei einem entsprechenden Datenformat könnte dies vereinfacht werden zu:

```
100 INPUT#1,AS
```

Hierdurch würde nicht nur das mehrmalige Senden von Steuercodes, sondern auch die Zusammensetzung einer Zeichenkette gespart werden.

Die Datenausgabe auf dem Bildschirm ist meistens bereits schnell genug, schließlich müssen die Daten noch abgelesen werden. Beim Aufbau von Bildschirmmasken oder ähnlichem kann es allerdings passieren, daß der Bildschirm durch die Erneuerung der Maske flackert. Hierbei tritt derselbe Effekt ein, der auch beim Schreiben von Programmen auftritt. Sobald der Cursor über den rechten Bildschirmrand hinaus in die nächste Zeile gelangt, werden alle weitere Zeilen nach unten

geschoben. Sobald ein Programm den Bildschirm löscht, um neue oder geänderte Daten auszudrucken, kann dieser Effekt auftreten. Der Neuaufbau der Bildschirmdarstellung verzögert sich dadurch. Dies kann verhindert werden, indem das alte Bild einfach überdruckt wird, dabei werden keine neuen Zeilen eingefügt.

10.2 Gleitkommafunktionen

Komplexe Gleitkommafunktionen (Sin, Cos, Tan, Exp, Log, etc.) arbeiten derart langsam, daß sie einen entscheidenden Einfluß auf die Geschwindigkeit der entsprechenden Programmteile haben. Da diese Funktionen auch vom compilierten Programm mit einer Genauigkeit von 9 Stellen berechnet werden, werden diese Funktionen nicht wesentlich schneller berechnet als vom Interpreter, da der Mikroprozessor des Commodore 64 hier an seine Grenzen stößt. Schneller wird lediglich der restliche der Teil der entsprechenden Formeln und Programme. Oft ist es dagegen nicht nötig, diese Funktionen mit der vollen Rechengenauigkeit zu berechnen, z.B. bei Grafiken mit geringer Auflösung, etc. In diesen Fällen kann man einfachere Formeln benutzen:

```
10 A=SIN(X):REM -Pi/2<X<Pi/2
```

Schneller ist:

```
10 A=X-X*X*X/6
```

Auf ähnliche Art lassen sich die meisten Gleitkommafunktionen beschleunigen. Für kleine X-Werte lassen sich weitere Umformungen vornehmen:

```
COS(X)=1-X*X/2
TAN(X)=X+X*X*X/3
```

Spürbar wird der Geschwindigkeitsgewinn natürlich erst nach dem Compilieren.

Programme, bei denen jede Möglichkeit zur Programmbeschleunigung ausgenutzt werden soll, können durch Abschalten des Videoprozessors geringfügig beschleunigt werden:

```
POKE 53265,PEEK(53265)AND239   ;anschalten des Bildschirms
POKE 53265,PEEK(53265)OR16     ;abschalten des Bildschirms
```

Weiterhin besteht die Möglichkeit, den Interrupt, wie in Abschnitt 8.5 beschrieben, abzuschalten. Dies deaktiviert allerdings die Tastatur und das Einstellen von TI$.

10.3 Grafik

Das Zeichnen von Grafiken dauert mit dem Basic-Interpreter äußerst lange. Besonders bei Grafiken zeigen sich deswegen die Vorteile des Compilers gegenüber dem Interpreter. Die folgenden Routinen benutzen fast ausschließlich Integer-Operationen:

```
5 REM@ 02
10 V=53248:AD=8192
20 POKEV+17,59:POKEV+24,24
30 FORI=1024T02023
40 POKEI,1
50 NEXT
60 FORI=8192T016383
70 POKEI,0
80 NEXT
90 L=1:FORI=0T07:MA%(1)=L:L=L*2:NEXT:GOT010000
100 REM DRAW X,Y,X2,Y2
105 X1=X:Y1=Y:IFYI>Y2THENH=Y1:Y1=Y2:Y2=H:H=X1:X1=X2:X2=H
110 DY=(Y2-Y1)*100:DX=X2-X1:V=SGN(X2-X1):Y1=Y1*100:IFV=0THENV=1
115 IFDX=0THENDX=1
116 D=ABS(INT(DY/DX))
120 X=X1
140 FORY=INT(Y1/100)TOINT((Yl+D)/100)-1:GOSUB1000:NEXT
150 Y1=Y1+D
160 IFX<>X2THENX=X+V:GOT0140
170 RETURN
1000 REM PLOT X,Y
1001 REM@ M
1010 OY=320*INT(Y/8)+(YAND7)
1020 OX=8*INT(X/8)
1030 MA=MA%(7-(XAND7))
1040 AV=AD+OX+OY
1050 POKEAV,PEEK(AV)ORMA:REM@ P
1060 RETURN
2000 REM CIRCLE X,Y,R
2001 REM@ 01
2010 S=R*R:Xl=X:Yl=Y
2030 FORX2=OTOCOS(.786)*R+1
2040 Y2=SQR(ABS(X2*X2-S))
2050 X=X2+Xl:Y=Y2+Yl:GOSUB1000
2060 Y=Y1-Y2:GOSUBI000
2070 X=Y2+Xl:Y=X2+Yl:GOSUB1000
2080 X=X1-Y2:GOSUB1000
2085 X2=-X2:IFX2<OTHEN2050
2090 NEXT:RETURN
10000 REM DEMO
10005 FORA=50T0150STEP10
10010 X=A:Y=A:X2=300-A:Y2=160-A:GOSUB100
10020 NEXT
10030 FORW=50T080STEP10
10040 X=150:Y=90:R=W:GOSUB2000
10050 NEXT
10060 GOT010060
```

Zum Compilieren des Programmes muß der Code-Start auf 16384 verschoben werden, um den Grafikspeicher freizugeben (Kapitel 6). Das Programm besitzt folgende Grafikroutinen:

```
Gosub 100  ;Zeichnet eine Linie von X,Y nach X2,Y2

Gosub 1000 ;Zeichnet einen Punkt mit den Koordinaten X,Y

Gosub 2000 ;Zeichnet einen Kreis mit dem Mittelpunkt X,Y und dem
Radius R.
```

Das Beispielprogramm ab Zeile 10000 können Sie natürlich gegen eigene Grafikprogramme austauschen. Das Programm nutzt die Möglichkeiten von BASIC 64 voll aus. Nach dem Entfernen der Compileranweisungen in Zeile 1001 und 1050 kann es sogar vollständig in Maschinensprache übersetzt werden. Zur vollen Ausnutzung der grafischen Fähigkeiten des Commodore 64 stehen dadurch schnelle Routinen zur Verfügung, die jederzeit ausgebaut werden können.

Durch die höhere Geschwindigkeit der compilierten Programme stehen Ihnen mit der Programmiersprache Basic nun Möglichkeiten zur Verfügung, an die Sie bisher noch nicht gedacht haben.

Viel Spaß wünscht Ihnen
Ihr BASIC 64 Autor
Thomas Helbig

WICHTIGE HINWEISE ZUM HANDBUCH

Im vorliegenden Handbuch finden Sie an einigen Stellen Zeichen, die Sie auf der Tastatur des C64 vergeblich suchen werden, wie z.B. PRINT§1,A$ oder 2^3.
Diese Zeichen haben Ihre Ursache in der Textverarbeitung.

Für das Parapraphenzeichen '§' nehmen sie das Ziffernkreuz # zu erreichen über SHIFT/3.

Für das "DACH" '^' nehmen Sie den Pfeil nach oben - zum Potenzieren - zu erreichen über die entsprechende Taste ↑.

*** Original-Ende ***

(Anmerkungen:
In dieser PDF-Version ist bereits '§' *durch* '#' *und* '^' *durch* '↑'*ersetzt worden.*
Einige Fehler des Original-Handbuchs sind korrigiert,
z.B. das Fehlen von @ hinter REM an mehreren Stellen,
und die falsche Schreibweise "Simon's Basic".
)