PROFIMAT

PROFI-MON, der Monitor und

PROFI-ASS, der Assembler für den

COMMODORE 64

EIN DATA BECKER PROGRAMM

PROFIMAT

PROFI-MON, der Monitor und

PROFI-ASS, der Assembler 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
Merowingerstr. 30
4000 Düsseldorf

A C H T U N G

DIESES PROGRAMM IST GEGEN KOPIEREN GESCHÜTZT

Natürlich ist kein Kopierschutz 100%ig sicher, aber wir werden mit allen uns zur Verfügung stehenden rechtlichen Mitteln versuchen, solchen Software-Piraten das Handwerk zu legen.

+++ KOPIERVERSUCHE KÖNNEN DIE DISKETTE ZERSTÖREN! +

Sollte aus irgendeinem Grund Ihre Diskette beschädigt oder gar zerstört werden, so senden Sie uns bitte Ihre Originaldiskette mit einem Verrechnungsscheck über DM 10.- an uns zurück. Sie erhalten dann postwendend eine neue Diskette.

PROFIMAT

PROFIMAT ist ein Programmiersystem zur Erstellung und zum Austesten von Maschinenprogrammen auf dem Commodore 64.

PROFIMAT besteht aus dem Maschinensprachemonitor

PROFI-MON 64

und dem Assembler

PROFI-ASS 64

Auf den folgenden Seiten finden Sie die ausführliche Beschreibung der beiden Programme.

PROFI-MON 64

Der Maschinensprache-Monitor für den Commodore 64

1. Befehle

R	Register display	Registerinhalte anzeigen
M	Memory display	Speicherinhalt anzeigen
G	Go	Maschinenprogramm ausführen
L	Load	Maschinenprogramm laden
S	Save	Maschinenprogramm abspeichern
D	Disassemble	Maschinenprogramm disassemblieren
C	Compare	Speicherbereiche vergleichen
T	Transfer	Speicherbereiche verschieben
H	Hunt	Speicherbereich durchsuchen
F	Fill	Speicherbereich beschreiben
B	Bank	Speicherkonfiguration wählen
W	Walk	Einzelschrittmodus
Q	Quicktrace	Tracen mit Unterbrechungspunkten
U	Unterbrechung	Unterbrechungspunkt setzen
X	Exit to BASIC	Rückkehr zu BASIC

2. Laden von PROFI-MON 64

Der Monitor belegt 3 KByte Speicher von $C000 bis $CBFF außerhalb des BASIC-Bereichs und wird von Diskette geladen. Geben Sie dazu

```
LOAD "PROFI-MON 64",8,1
```

ein und drücken Sie RETURN. Auf dem Bildschirm erscheint nun

```
SEARCHING FOR PROFI-MON 64
LOADING

PROFI-MON 64 V2.0 WIRD GELADEN ...
```

Der Monitor meldet sich nun mit

```
*** PROFI-MON 64 V2.0 ***
(C) 1984 DATA BECKER GMBH

C*
```

und zeigt die Registerinhalte an.

Sämtliche Monitor- Ein- und Ausgaben können nun als 2- oder 4-stellige Hexadezimalzahlen erfolgen.

3. Anzeige der Registerinhalte >R

Die Inhalte der Prozessorregister werden angezeigt.

PC	Programmzähler
IRQ	Interruptvektor
SR	Statusregister
AC	Akkumulator
XR	X-Register
YR	Y-Register
SP	Stapelzeiger

Zusätzlich werden die Flags des Statusregisters SR noch einzeln angezeigt:

N	Negativflag
V	Überlaufflag
-	unbenutzt
B	Breakflag
D	Dezimalflag
I	Interruptflag
Z	Zeroflag
C	Carryflag

Beispiel: >R

```
    PC   IRQ   SR AC XR YR SP  NV-BDIZC
>; 0003  EA31  32 34 02 A2 F8  00110010
```

Wollen Sie die Registerinhalte ändern, so gehen Sie einfach mit dem Cursor an die entsprechende Stelle, überschreiben mit dem neuen Wert und drücken RETURN. Die neuen Registerinhalte werden übernommen. Die Flags werden im Statusregister geändert und angezeigt.

4. Anzeige der Speicherinhalte >M XXXX YYYY

XXXXX = Anfangsadresse, YYYY = Endadresse, jeweils vierstellige Hexzahlen. Die entsprechenden Speicherinhalte werden angezeigt. Wird die Endadresse YYYY weggelassen, so wird nur eine Zeile ausgegeben. Die ASCII-Darstellung wird hinter der Bytedarstellung revers dargestellt. Nichtdruckende Steuerzeichen erscheinen als Punkt.

Beispiel: >M A0A0 A0AF

```
>: A0A0  C4 46 4F D2 4E 45 58 D4 DFORNEXT
>: A0A8  44 41 54 C1 49 4E 50 55 DATAINPU
```

Speicherinhalte können Sie analog dem R-Befehl ändern, indem Sie die Bytewerte überschreiben und RETURN drücken.

5. Programmausführung >G XXXX

Der Go-Befehl bewirkt einen Sprung zu der Adresse XXXX und führt das Maschinenprogramm, das dort steht aus. Ist XXXX nicht angegeben, so wird dafür der Wert des Programmzählers PC genommen.
Trifft das Maschinenprogramm auf dem Befehl BRK ($00), so meldet sich der

Monitor mit B* (Break) wieder und die Register werden angezeigt. Der Programmzähler zeigt auf die Adresse nach dem BRK-Befehl. Beim Austesten empfiehlt es sich daher, das Maschinenprogramm mit BRK (Hex 00) abzuschließen.

6. Laden eines Maschinenprogramms >L "NAME",XX,YYYY

Das Programm "NAME" wird von Gerät XX ab der Adresse YYYY geladen. Normalerweise wird YYYY weggelassen; das Programm wird dann an die Adresse geladen, die beim SAVE-Befehl angegeben wurde. Wenn Sie die Geräteadresse XX auch weglassen, so wird dafür Nummer 8 angenommen.

```
Beispiel:   >L "NAME",08
            SEARCHING FOR NAME
            LOADING
            >
```

Wollen Sie von Kassette laden, so geben Sie für XX 01 ein.

7. Abspeichern eines Maschinenprogramms >S "NAME",XX,YYYY,ZZZZ

XX ist wieder die Geräteadresse, YYYY ist die Startadresse und ZZZZ die Endadresse plus eins des abzuspeichernden Programms.

Beispiel: >S "NAME",01,C900,C9DE

Das Programm "NAME" von Adresse $C900 bis $C9DD wird auf Kassette abgespeichert.

8. Disassemblieren eines Maschinenprogramms >D XXXX YYYY

Das Maschinenprogramm von Adresse XXXX bis YYYY wird in mnemonischer Form (Operationscodes) ausgegeben. Wird die Endadresse YYYY nicht mit angegeben, so wird nur eine Zeile ausgegeben. Bei nicht existierenden Opcodes werden 3 Fragezeichen ausgegeben.

```
Beispiel:   >D B016 B021
            >, B016   20 90 AD   JSR $AD90
            >, B019   B0 13      BCS $B02E
            >, B01B   A5 6E      LDA $6E
            >, B01D   09 7F      ORA #$7F
            >, B01F   25 6A      AND $6A
            >, B021   85 6A      STA $6A
```

Befindet sich an den angezeigten Adressen RAM, so können Sie die Bytes nach der Adresse ändern. Durch drücken von RETURN wird die Änderung wirksam und der Befehl wird neu disassembliert. Dabei wird in der nächsten Zeile automatisch die folgende Adresse ausgegeben und der Cursor steht auf dem ersten Byte des Befehls, so daß Sie sofort die nächste Anweisung ändern können. Abbrechen können Sie diesen Modus, in dem Sie die Zeichen hinter der Adresse löschen bevor Sie RETURN drücken.

9. Vergleichen von Speicherbereichen >C XXXX YYYY ZZZZ

Der Speicherbereich von Adresse XXXX bis YYYY wird byteweise mit dem Bereich ab Adresse ZZZZ verglichen. Unterschiedliche Adressen werden dabei ausgegeben.

Beispiel: >C 8000 8100 9000
8056
Die Adresse $8056 stimmt nicht mit Adresse $9056 überein.

10. Verschieben von Speicherbereichen >T XXXX YYYY ZZZZ

Der Speicherbereich von Adresse XXXX bis YYYY wird nach Adresse ZZZZ kopiert.

Beispiel: >T 6000 6FFF 3000
Die Speicherbereich von $6000 bis $6FFF wird nach $3000 bis $3FFF kopiert. Der ursprüngliche Bereich bleibt unverändert.

11. Durchsuchen von Speicherbereichen. Dabei gibt es zwei Möglichkeiten; suchen von Bytekombinationen oder suchen von ASCII-Texten.

11.1 Suchen nach Bytekombinationen >H XXXX YYYY BB BB BB

Der Speicherbereich von Adresse XXXX bis YYYY wird nach der Bytekombination BB durchsucht. Es können bis zu 29 Bytes eingegeben werden.

Beispiel: >H E000 EFFF 20 D2 FF
Der Speicherbereich von $E000 bis $EFFF wird nach der Kombination $20 $D2 $FF (Aufruf der Ausgaberoutine) durchsucht. Adressen, an denen diese Kombination steht, werden ausgegeben.

11.2 Suchen nach Texten >H XXXX YYYY "TEXT"

Der Speicherbereich von Adresse XXXX bis YYYY wird nach dem ASCII-Text "TEXT" durchsucht. Der Text kann bis zu 29 Zeichen lang sein. Adressen, ab denen der Text steht, werden ausgegeben.

Beispiel: >H A000 AFFF "READY"
A378

12. Füllen von Speicherbereichen >F XXXX YYYY ZZ

Der Bereich von Adresse XXXX bis YYYY wird mit dem Byte ZZ gefüllt.

Beispiel: >F 8000 8FFF 00

13. Umschalten der Speicherkonfiguration >BX

Mit diesem Befehl haben Sie den Zugriff auf den kompletten Speicher des Commodore 64. Nach dem Starten des Monitors beziehen sich alle Befehle auf die normale Speicherkonfiguration. PROFI-MON erlaubt aber noch zwei weitere Konfigurationen. Mit >BA können Sie die Speicherbelegung komplett auf RAM umschalten, während >BC den Zeichengenerator zugänglich macht. Auf die normale ROM-Konfiguration können Sie mit >BR zurückschalten. Diese Belegungen beziehen

sich jedoch nur auf die Befehle

M, D, C, T, H und F

Zur Erläuterung finden Sie die drei Konfigurationen noch einmal tabellarisch dargestellt.

Adressbereich	>BR	>BA	>BC
$E000 - $FFFF	ROM	RAM	RAM
$D000 - $DFFF	I/O	RAM	CHARACTERROM
$C000 - $CFFF	RAM	RAM	RAM
$A000 - $BFFF	ROM	RAM	RAM
$0000 - $9FFF	RAM	RAM	RAM

14. Einzelschrittmodus >W XXXX

Eine Besonderheit von PROFI-MON ist der Einzelschrittmodus. Mit ihm kann man Maschinenprogramme Befehl für Befehl abarbeiten. Der Befehl hat die gleiche Syntax wie der G-Befehl, startet also an der angegebenen Adresse bzw. ab der Adresse des Programmzähler, falls nur W eingegeben wurde. Wenn Sie W eingegeben haben, so wird der Befehl an dieser Adresse ausgeführt. Nun werden die Inhalte der Register und die Flags im selben Format wie beim R-Befehl angezeigt. In der nächsten Zeile wird der folgende Befehl disassembliert. Drücken Sie nun eine Taste, so wird der nächste Befehl ausgeführt und die resultierenden Registerinhalte werden wieder angezeigt. Abbrechen können Sie den Einzelschrittmodus mit der STOP-Taste.

```
Beispeil:   >W BC16
            >; BC18 EA31 22 69 34 00 F6  00100010
            >, BC18   86 70       STX $70
```

Der Einzelschrittmodus funktioniert bei allen "normalem" Programmen. Bei Programmen, die die I/O-Bausteine benutzen, sollte der Einzelschrittmodus nicht benutzt werden.

15. Programmausführung mit Unterbrechungspunkten >Q XXXX

Oft würde ein Abarbeiten eines Maschinenprogramms im Einzelschrittmodus zu lange dauern. Deshalb bietet PROFI-MON die Möglichkeit, die Ausführung eines Maschinenprogramms zu kontrollieren und Unterbrechungspunkte zu setzen.

Sie können also bestimmen, daß ein Maschinenprogramm dann unterbrochen werden soll, wenn das Programm eine bestimmte Stelle erreicht hat. Sollte das Programm nicht auf der Unterbrechungspunkt kommen, so können Sie es auch mit der STOP-Taste unterbrechen. Das Setzen dieses Unterbrechungspunkt geschieht mit dem U-Befehl, der im nächsten Abschnitt beschrieben ist. Die Syntax des Q-Befehls ist analog dem G- und W-Befehl.

16. Setzen von Unterbrechungspunkten >U XXXX YYYY

Wenn Sie den Q-Befehl verwenden wollen, so müssen Sie erst einen Unterbrechungspunkt setzen. Dazu benutzen Sie den U-Befehl. Setzen Sie für XXXX die Adresse, an der das Programm unterbrochen werden soll. Starten Sie Ihr Programm nun mit dem Q-Befehl, so wird es beim Erreichen der Adresse XXXX unterbrochen. Sie befinden sich nun automatisch im Einzelschrittmodus (W). Mit STOP können Sie nun abrechren bzw. mit einer beliebigen Tasten schrittweise weiter machen. Der U-Befehl bietet aber noch zusätzlich die Möglichkeit, das laufende Programm erst dann zu unterbrechen, wenn es zum wiederholten Male den Unterbrechungspounkt passiert hat. Dazu können Sie mit dem zweiten Parameter YYYY angegeben, beim wievielten Male dies geschehen soll, z.B.

```
>U 1000 0050
```

Hier wird das Programm dann unterbrochen, wenn zum 80. Male die Adresse $1000 assiert wurde. Sie können also Werte bis hexadezimal $FFFF angeben, das sind 65535.

17. Rücksprung zu BASIC >X

Mit >X kommt man wieder ins Commodore BASIC zurück. Das BASIC befindet sich im selben Zustand wie vor dem Aufruf. Nach dem X-Befehl können Sie mit SYS 2 oder einem SYS auf jede Adresse, die Null enthält, wieder in den Monitor gehen, sofern Sie zwischendurch nicht RUN/STOP RESTORE gedrückt haben (ansonsten mit SYS 12*4096).

Fehlermeldungen

Haben Sie eine fehlerhafte Eingabe gemacht, so wird PROFI-MON diese zurückweisen und an dieser Stelle ein Fragezeichen '?' ausgeben. Korrigieren Sie in diesem Falle das Eingabeformat.

Neben diesen syntaktischen Fehlern werden durch PROFI-MON 64 die Fehlerroutinen des Betriebssystems aktiviert. Treten bei Ein/Ausgabe-Operationen, z.B. bei LOAD und SAVE Fehler auf, so erscheint eine Fehlermeldung in der Form

I/O ERROR #X

wobei X eine Zahl zwischen 1 und 9 sein kann, die folgende Bedeutung haben:

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

PROFI-ASS 64 V2.0
Komfortabler 6510-MACRO-Assembler
für den Commodore 64

PROFI-ASS 64 ist ein komfortabler 2-Pass 6510- bzw. 6502-Macro-Assembler für den Commodore 64. Er ist vollkommen in Maschinensprache geschrieben und belegt 8 KByte RAM-Speicher und wird einfach von Diskette geladen. Er erlaubt eine formatfreie Eingabe über den residenten BASIC-Editor, komplette Assemblerlistings, ladbare Symboltabellen, verschiedene Möglichkeiten zur Speicherung des erzeugten Opcodes, redefinierbare Symbole und eine Anzahl verschiedener Pseudoopcodes (Assembleranweisungen) einschließlich der Möglichkeit, Assemblersequenzen zu Macros zusammenzufassen sowie der bedingten Assemblierung. Die Syntax lehnt sich stark an das MOS-Standardformat an.

Anwendung von PROFI-ASS 64

PROFI-ASS 64 wird von Diskette geladen werden und belegt die obersten 8 KByte des BASIC-RAMs (Adresse $8000 - $9FFF). Der meist für Maschinenprogramme benutzte Bereich von $C000 bis $CFFF bleibt frei und kann z.B. für PROFI-MON 64 (Adresse $C000 bis $CBFF) sowie für die eigenen Maschinenprogramme benutzt werden.

Laden von PROFI-ASS 64

Legen Sie dazu die PROFI-MAT-Diskette ein und geben Sie

```
LOAD "PROFI-ASS 64",8,1
```

ein. Auf dem Bildschirm erscheint dann

```
LOADING

PROFI-ASS 64 V2.0 WIRD GELADEN ...

*** PROFI-ASS 64 V2.0 ***
(C) 1984 DATA BECKER GMBH
2
]0000-0000
NO ERRORS
READY.
```

Beim Laden schützt sich PROFI-ASS 64 selbst vor Überschreiben durch BASIC. Ein eventuell im Speicher stehendes BASIC- oder Assembler-Quellprogramm bleibt dabei erhalten. Es stehen Ihnen dann noch 30717 Bytes für Ihr Assemblerprogramm zu Verfügung.

Die zwei in der Meldung bedeutet Beginn von Pass 2, darunter steht dann der Adressbereich des erzeugten Codes sowie eine Fehlerstatistik.

Assemblerquellprogramme werden einfach wie ein BASIC-Programm mit Zeilennummern eingegeben. Genauso wie in BASIC können Sie auch Zeilen ändern, löschen oder einfügen. Dadurch ist kein eigener Editor erforderlich und es steht Ihnen mehr Speicherplatz für Ihr Quellprogramm zur Verfügung; insgesamt 30

KByte. Mit PROFI-ASS 64 können Sie sogar mehrere Assemblerbefehle durch Doppelpunkt getrennt in eine Zeile schreiben, genau wie in BASIC.

Sie können Ihre Assemblerprogramme übersichtlicher gestalten, indem Sie als erstes Zeichen einen Pfeil nach oben (↑) schreiben. Danach werden dann führende Leerzeichen akzeptiert, der Pfeil wird von PROFI-ASS 64 ignoriert. Dadurch können Sie Ihre Programme nach Belieben einrücken.

PROFI-ASS 64 benutzt fast das gleiche Quellformat wie der MOS Standard. Auch wenn Sie diesen Standard bereits kennen, sollten Sie diese Beschreibung lesen, da sowohl die Unterschiede als auch der Standard beschrieben werden. Die Beispiele erläutern die Befehle.

Diese Anleitung ist natürlich kein Programmierlehrgang für den 6510; Sie sollten sich also schon ein bißchen in der Syntax des 6510-Assemblercodes auskennen. Zu diesem Thema bieten wir Ihnen eine ganze Menge an Literatur an, so können wir Ihnen zugeschnitten auf Ihren Commodore 64 das DATA BECKER Buch "Maschinenspracheprogrammierung auf dem Commodore 64" bzw. speziell zur Anwendung von Macros und Fließkommaarithmetik "Maschinensprache für Fortgeschrittene" empfehlen.

Zeilen mit PROFI-ASS 64 Quellcode bestehen aus Symbolen, Befehlsmnemonics, den Operanden und Kommentaren. Zusätzlich gibt es noch mehrere sogenannte "Pseudo-Opcodes", die keine Maschinenbefehle darstellen, sondern PROFI-ASS 64 anweisen, spezielle Dinge zu tun. Diese Pseudoops werden später im Handbuch beschrieben.

Jeder Programmzeile mit einem Mnemonic oder einem Pseudop kann man ein Label (Marke, Symbol) geben. Soll eine Zeile ein Label erhalten, so schreibt man es einfach vor die Anweisung, gefolgt von einem oder mehreren Leerzeichen. Ein Label muß mit einem Buchstaben beginnen, danach können Buchstaben oder Ziffern oder Punkte folgen. Unterschiedliche Labels müssen sich in den ersten 8 Zeichen unterscheiden. Nicht alphanumerische Zeichen sind nicht erlaubt, obwohl nur ein Leerzeichen ein Label vom nachfolgenden Befehl trennt.

Befehlsmnemonics können nach einem Label oder am Anfang der Zeile eingegeben werden, falls kein Label da ist. Alle Mnemonics bestehen aus drei Buchstaben. Mnemonics sind reservierte Worte und dürfen nicht als Label benutzt werden. Beginnt der Befehl mit einem Punkt ".", so handelt es sich um ein Pseudoop. Es gibt noch drei Pseudoops, die nicht mit einem Punkt, sondern speziellen Sonderzeichen beginnen. Alle Pseudoops müssen durch ein Leerzeichen von Ihren Operanden getrennt werden, mit der Ausnahme von "=" und "*=". Pseudoops, die mit einem Punkt beginnen, werden nur in den ersten drei Buchstaben unterschieden, im Assemblerlisting jedoch vollständig ausgedruckt.

Eine Assemblerzeile kann an jeder Stelle durch ein Semikolon ";" abgeschlossen werden; alles was danach folgt gilt als Kommentar. Kommentare werden beim Assemblerlisting mit ausgedruckt, ansonsten aber nicht beachtet. Ein Doppelpunkt innerhalb eines Kommentar beendet diesen und beginnt eine neue Anweisung, sofern er nicht innerhalb von Anführungsstrichen steht. Der Assembler erkennt auch Zeilen, die nur einen Kommentar enthalten, d.h. die mit einem Semikolon beginnen. Solche Zeilen werden ohne die Zeilennummer ausgedruckt.

Das Operandenfeld enthält die Adressierungsart und einen Ausdruck für den Befehl oder den Pseudoop. Es darf wie gewöhnlich von einem Semikolon gefolgt werden.

Die Adressierungsarten mit den Ausdrücken haben folgende Syntax:

#ausdruck	unmittelbare Adressierung
ausdruck	absolute oder relative Adressierung
ausdruck,x absolut,x	indiziert mit x
ausdruck,y absolut,y	indiziert mit y
(ausdruck,x)	indiziert indirekte Adressierung
(ausdruck),y	indirekt indizierte Adressierung
(ausdruck)	indirekte Adressierung

PROFI-ASS 64 konvertiert die absolute Adressierung automatisch zur Zero-Page-Adressierung, wenn der Ausdruck einen Wert von kleiner als 256 hat. Will man absolute Adressierung erzwingen, kann man vor den Ausdruck ein Ausrufungszeichen setzen. LDA !5,X erzeugt den Code BD 05 00, die absolute Form von LDA; während LDA 5,X die Zero-Page-Adressierung B5 05 ergibt. Dies ist nützlich, wenn man den Wrap-Around-Effekt bei der indizierten Adressierung mit Adressen unter 256 vermeiden will.

Ausdrücke

PROFI-ASS 64 erhält seine Vielseitigkeit durch die Fähigkeit, beliebig komplizierte Ausdrücke zu berechnen. Der Assembler hat eine rekursive Routine zur Berechnung von beliebig verschachtelten Ausdrücken, was Ihnen weit größere Möglichkeiten einräumt als der MOS Standard oder andere Assembler zulassen. Ein PROFI-ASS 64-Ausdruck kann überall dort stehen, wo das Wort "Ausdruck" in der obigen Aufzählung erscheint. Auch bei den Pseudoops ist solch ein Ausdruck erlaubt, sofern ein numerischer Ausdruck erwartet wird. Die Ausdrucksberechnung mit PROFI-ASS 64 ist deshalb so leistungsfähig, damit Sie Ihre Assemblerprogramme vollständig symbolisch bezeichnen können. Dadurch ist das Ändern und Verschieben von PROFI-ASS 64-Programmen besonders einfach und übersichtlich.

Die Syntax von Ausdrücken ist sehr einfach und hat den MOS-Standard als Untermenge. Ausdrücke werden so eingegeben, wie man sie in einen Taschenrechner eingibt, der keine bindende Operatoren, jedoch Klammern hat. Alle Operatoren werden strikt von links nach rechts abgearbeitet, jedoch sind sowohl runde als auch eckige Klammern erlaubt, um die Reihenfolge der Abarbeitung zu ändern.

Ein Ausdruck kann durch verschiedene Zeichen abgeschlossen werden. Das Ende der Programmzeile schließt einen Ausdruck immer ab. Doppelpunkt, Semikolon und Komma beenden ebenfalls einen Ausdruck, sofern sie nicht in Anführungsstrichen stehen. Eine schließende Klammer beendet ebenfalls einen Ausdruck, sofern keine unpaarige offene Klammer vorliegt. Dadurch werden geschachtelte Ausdrücke bei der indizierten Adressierung möglich.

Folgende Operatoren stehen zur Verfügung:

+	addiert Werte
-	subtrahiert rechten Wert von linkem Wert
*	multipliziert Werte
/	dividiert linken Wert durch den rechten Wert
!	logisches OR von zwei Werten
&	logisches AND von zwei Werten
↑	logisches XOR (exklusiv oder) von zwei Werten
>	verschiebt linkes Argument um soviel Bits nach rechts, wie das rechte

Argument angibt
< verschiebt linkes Argument um soviel Bits nach links, wie das rechte Argument angibt

Alle Operationen werden mit 16 Bit Werten durchgeführt, jedoch führen verschiedene Operationen zu Überläufen, z.B. Multiplikation mit Werten über 32767 oder Verschieben um mehr als 15 Bit nach links; es erscheint dann ein 'ILLEGAL QUANTITY ERROR'. Diese Fehlermeldung erscheint auch bei einer Division durch Null. Bei Addition und Subtraktion wird ein Ergebnis über 65535 jedoch als negative Zahl im Zweierkomplement betrachtet.

Die Operanden selbst können in verschiedenen Formen erscheinen. Im folgenden wird die Syntax anhand von Beispielen beschrieben.

Operanden-Typen

Typ	Beispiel	Syntax
Hexadezimal	$1C3	$(Hexziffer)*
Dezimal	127	Ziffer*
Binär	%110011	%(0 oder 1)*
PC	*	
ASCII-Zeichen	"A"	"Zeichen"
Label	SYMB	alphabetisch(aplhanumerisch)*
Ausdruck	("Z"+6)	(Ausdruck)

Bei der Syntax bedeutet ein geklammerter Begriff mit einem Stern dahinter, das er beliebig oft wiederholt werden kann.

Jeder der obigen Terme kann mit den weiter oben beschriebenen Operatoren verknüpft werden. Diese können gruppenweise geklammert werden, um die Reihenfolge der Verarbeitung zu ändern. Vor jedem Operand, einschließlich einem geklammerten Ausdruck, kann ein Minuszeichen stehen, was bedeutet, daß das Zweierkomplement zu nehmen ist. Einem kompletten Ausdruck kann noch ein modifizierendes Zeichen vorausgehen. Die Bedeutung des Ausrufungszeichen wurde bereits weiter oben erklärt. Zusätzlich ist noch das "größer" und "kleiner"-Zeichen erlaubt. ">" vor einem Ausdruck bedeutet, daß Sie nur das High-Byte wollen, während "<" das Low-Byte kennzeichnet. Dies ist bei der direkten Adressierung oder beim .BYTE Pseudoop erforderlich. Der High-Byte Operator (>) tut dasselbe wie

```
Ausdruck > 8
```

und der Low-Byte Operator ist gleichzusetzen mit

```
Ausdruck & $FF
```

Beispiel-Ausdrücke

```
>LABEL-1+(TABLEN*2)
WERT-*
"0"-"A" < 3 + ("D" - "A" > 2&%111)
```

Klammern können so tief wie erforderlich geschachtelt werden. Die Modifizierer können nicht auf geklammerte Teilausdrücke angewandt werden.

Pseudo-Opcodes

Die meisten Pseudo-Opcodes beim PROFI-ASS 64 beginnen mit einem Punkt (".").
Alle diese "Punkt"-Opcodes müssen von nachfolgenden Zeichen durch mindestens ein Leerzeichen getrennt werden. Zusätzlich gibt es noch drei spezielle Pseudoops, die durch Sonderzeichen definiert sind. Pseudoops werden nur anhand Ihrer ersten drei Buchstaben erkannt; alles was bis zum nächsten Leerzeichen dahinter folgt, wird ignoriert, jedoch beim Listing mit ausgedruckt.

Die drei speziellen Pseudoops dienen zur Definition von Symbolen und des Programmzeigers. Der einfachste davon ist der Operator zur Definition eines Symbols, das Gleichheitszeichen ("="). Um einem Symbol einen Wert (einen Ausdruck) zuzuweisen, schreiben Sie einfach:

```
SYMBOL = AUSDRUCK
```

Die Zuweisung geschieht nur bei Pass 1 der Assemblierung; jede nochmalige Definition zu einem späteren Zeitpunkt ergibt einen 'REDEFINITION ERROR'. Das Gleichheitszeichen wird benutzt, um Konstanten und Adressen in symbolischer Form zu definieren, so daß bei Programmänderungen immer nur eine Zeile geändert werden muß.

Ähnlich dem Operator zur Definition von Symbolen ist der Zuweisungsoperator, der als Pfeil nach links geschrieben wird und mit der gleichen Syntax benutzt wird. Im Unterschied zu Gleich ist eine Redefinition jedoch möglich, die sowohl in Pass 1 als auch in Pass 2 durchgeführt wird. Dies kann für verschiedene Zwecke einschließlich der Erzeugung von Assemblerschleifen (siehe .GOTO) benutzt werden. Hier einige Beispiele:

```
SYMBOL ← WERT
ZAHL ← ZAHL - 1 ; DEKREMENTIEREN EINES SYMBOLS
PROGRAMM ← *
```

Der dritte spezielle Pseudoop kontrolliert den Programmzähler. Er wird als Stern gleich ("*=") geschrieben, was bedeutet "weise dem Programmzähler einen Wert zu". Der eine Zweck ist einfach die Definition des Programmstart, der ohne Angabe bei $C000 liegt. Ebenso kann man damit Speicherbereich zuweisen. *=*+32 definiert z.B. einen 32-Byte-Block ab dem augenblicklichen Programmzählerstand und bewegt gleichzeitig den Programmzähler um 32 Byte nach oben. Meistens wird er jedoch zur Symboldefinition benutzt. Steht ein Symbol im Labelfeld, so wird ihm der Wert des Programmzeigers zugeordnet *bevor der Programmzeiger geändert wurde.* Wenn Sie einen Block von Variablen in der Page 2 zuweisen wollen, können Sie folgendermaßen vorgehen:

```
*= $200 ; setzt Programmzeiger auf Beginn von Page 2
ADRESS *= *+1 ; Ein-Byte-Adresse, auf null gesetzt
TABELLE *= *+32 ; Tabelle beginnt bei $201
LABEL *= *+1 ; LABEL hat den Wert $233
ZWEI *= *+2 ; Zweibyte-Zeiger
TEST *= $800 ; TEST hat den Wert $236, Code beginnt bei $800.
```

Um eine Tabelle im Programm zu definieren, dienen folgende Befehle:

```
        ...
        LDA #5
        RTS
TABEL   *=  *+256 ; 256 Byte Tabelle
TEST    LDA #>ADRESSE*3
        ...
```

Im allgemeinen können Sie "*=" benutzen, um den Programmzähler zur Symboldefinition beliebig zu bewegen, Sie sollten ihn jedoch nicht zurückbewegen, um Code zu assemblieren. Dies ist nur erlaubt, wenn Sie Code direkt in den Speicher assemblieren, um ihn dort laufen zu lassen, oder wenn Sie überhaupt keinen Code erzeugen. Wenn Sie z.B. Code bei $1000 assemblieren, können Sie den Programmzähler normalerweise nicht nach $0F00 zurücksetzen, um dort Code zu asssemblieren. Zur Labeldefinition ist dies erlaubt; Sie müssen dann jedoch wieder zu einer Adresse zurückkehren, die höher als die Adresse war, in die das letzte Byte mit Programmcode ging. Es ist durchaus üblich, die Symboldefinition- und Zuweisungspseudoops zu benutzen und mit dem Programmzähler zu springen. Bei der Zuweisung mit "*=" wird einem Label immer der Wert *vor der Programmzähleränderung* zugewiesen wird. Steht in einer Assemblerzeile ein Label, so wird im Listing immer anstelle des Programmzählers der Wert des Labels ausgegeben, so daß Sie ihn bei Unklarheiten direkt aus dem Listing entnehmen können.

Punkt-Pseudo-Opcodes

.BYTE

Der .BYTE Pseudoop wird benutzt, um Ein-Byte-Werte an der Programmzählerposition einzufügen. Als Operanden dienen gültige PROFI-ASS 64-Ausdrücke, die durch Komma getrent sind. Die Anzahl ist nur durch die Zeilenlänge und die Länge des PROFI-ASS 64-Puffers begrenzt. Es können beliebige Ausdrücke mit beliebigen Operatoren benutzt werden; das Ergebnis muß jedoch einen Ein-Byte-Wert ergeben, sonst gibt es einen 'ILLEGAL QUANTITY ERROR'. Zwei-Byte-Werte können also mit ">" und "<" modifiziert werden, um High- oder Low-Byte zu erhalten. Ein Ein-Byte-Wert liegt im Bereich 0 bis 255 oder $FF80 bis $FFFF. Die hohen Werte sind erlaubt, da sie normalerweise negative Zahlen von -1 bis -128 bedeuten. Deshalb ist auch die Zeile ".BYTE -1" erlaubt. .BYTE dient zum Definieren von Tabellen wie Sprungtabellen oder Zeigern. Man kann damit auch Befehle "einschmuggeln", wie z.B. den BIT-Befehl; ein bekannter Programmierkniff:

```
      .BYTE $2C  ; absoluter BIT-Befehl
LABEL LDA #-1 ; versteckter LDA-Befehl
```

.WORD

Der .WORD Pseudopcode wird benutzt, um Zwei-Byte Adressen im Standardformat Low, High in den Objektcode einzufügen. ".WORD ADRESSE" ist dasselbe wie ".BYTE <ADRESSE, >ADRESSE". Mit dem .WORD-Befehl können Sie wie beim .BYTE-Befehl mehrere Werte durch Kommas getrennt in eine Zeile schreiben. Der .WORD-Befehl dient in erster Linie dazu, Adresstabellen aufzubauen.

.FILE

Wenn man mehrere Quellprogramme verketten will, so dient dazu der .FILE Pseudoop. Die Syntax ist folgende:

.FILE Gerätenummer, "filename"

wobei 'Gerätenummer' die Nummer der Floppy (8) oder der Datasette (1) ist und 'filename' ist der Name des Assemblerprogramms, das nachgeladen werden soll. Wenn Sie also ein sehr langes Assemblerprogramm schreiben, so können Sie es in mehreren Teilen schreiben und diese beim Assemblieren mit .FILE miteinander verketten. Eine solche verkettete Folge von Programmen sollte mit .END Gerätenummer, "Name des ersten Programms" abgeschlossen werden. Siehe dazu auch .END sowie die Beispiele.

.IF

Das .IF wird für bedingte Assemblierung benutzt. Es hat einen Ausdruck als Argument und wird in Pass eins und zwei berechnet. Falls der Ausdruck nicht null ist, wird der Code in der selben Zeile nach .IF assembliert. Meist wird dort ein .GOTO zu einer weiteren Verzweigung stehen. Der zusätzlich Code in der Zeile muß durch ':' getrennt sein. Mit .IF, .GOTO und der Redefinition von Symbolen ist es möglich, Assemblerschleifen aufzubauen. Obwohl .IF nur auf ungleich Null testet, ist es mit einfachen Tricks möglich, andere Vergleiche anzustellen. Das Shiften

um 15 Bits nach rechts gibt ein Ergebnis von 1, wenn der Ausdruck negativ war und 0 bei positiv. Zwei Zahlen können also verglichen werden, indem man sie voneinander abzieht und das Resultat auf positiv oder negativ überprüft.

.GOTO

Der .GOTO Pseudoop veranlaßt einen unbedingten Sprung zu der Zeilennummer, die als Argument angegeben ist. Diese Zeilennummer kann auch ein Ausdruck sein. Die Zeilenummer bezieht sich immer auf das gerade geladene Programm (wenn Sie .FILE benutzen). Sie können also nicht zwischen verschiedenen Files hin- und herspringen. Die Zeilennummer kann jedoch beliebig im Programm stehen. Zusammen mit .IF und dem Dekrementieren eines Symbols läßt sich so eine Schleife aufbauen. Probieren Sie folgendes Beispiel:

```
10 SYS 32768 : REM AUFRUF DES ASSEMBLERS
20 .OPT P ; LISTING AUF BILDSCHIRM
30 OFFSET← 5 ; ANZAHL DER SCHLEIFEN
40 LDA $C000 + OFFSET
50 OFFSET ← OFFSET - 1 ; DEKREMENTIEREN
60 .IF OFFSET : . GOTO 40
70 .END
```

.GTB

Dieser Pseudop steht für Go To BASIC. Er hat kein Argument und übergibt die Kontrolle an BASIC. Die im Programm folgenden BASIC-Befehle werden ausgeführt. Die Rückkehr zum Assembler geschieht über einen speziellen Einsprungpunkt im Assembler. Die Adresse ist 5 Bytes vor der letzen Adresse des Assemblers, das ist $9FFA oder dezimal 40954. Es soll noch darauf hingewiesen werden, daß der Gebrauch des BASIC während des Assemblierens eingeschränkt ist; einige Statements können den Arbeitsspeicher des Assemblers überschreiben und dürfen nicht ausgeführt werden. Speziell ist dies der INPUT-Befehl oder jeder andere Befehl, der ab Byte 9 des BASIC-Eingabepuffer schreibt (Adresse $0209). Das GET-Statement ist jedoch erlaubt. Sie sollten jedoch niemals dem Benutzer die Kontrolle geben.

.ASC "TEXT"

Dieser Pseudoop dient zum Einfügen von Text in das Assemblerprogramm. Der Text wird dabei in Anführungszeichen eingeschlossen. Dadurch ist es möglich, auch Cursorsteuerzeichen in den Text einzubetten. Der Text kann bis zu 55 Zeichen lang sein; längere Texte teilen Sie einfach auf mehrere .ASC-Befehle auf. Der MOS-Standard verlangt hier den .BYTE-Pseudoop, wobei Strings in einfache Hochkommas ' eingefügt werden. Dies sollten Sie beim Konvertieren von Standard-Programmen beachten.

.SYS

Hiermit kann ein eigenes Maschinenprogramm während der Assemblierung aufgerufen werden. Als Einsprungadresse kann ein beliebiger Ausdruck stehen. Der Befehl ist identisch mit dem SYS-Befehl im BASIC. Der Aufruf des Programms geschieht in Pass eins und zwei. Der .SYS-Befehl kann z.B. von Programmierern,

die sich mit der internen Arbeitsweise von PROFI-ASS 64 gut auskennen, zum Generieren von eigenen Pseudoops benutzt werden.

.STM

Dies wird benutzt, um die Untergrenze des Speichers für die Symboltabelle höher zu setzen. Diese Symboltabelle wächst vom oberen Ende des Speichers ($8000) nach unten; genauso wie BASIC die Strings abspeichert. Beim Start des Assemblers wird die Untergrenze auf das Ende von BASIC-Programm und Variablen gelegt. Sie können sie höher setzen, wenn Sie mit .FILE arbeiten oder gepufferte Objektcodeausgabe (.OPT O) benutzen. Wird der Platz für die Symboltabelle zu klein, gibt es einem 'SYM TABLE OVERFLOW', der den Assembliervorgang abbricht.

.SST Gerätenummer, Sekundäradresse, "Filename"

Symboltabellen können auf IEC-Busgeräten wie z.B. Floppydisk abgespeichert und von dort wieder geladen werden. .SST wird nur in Pass eins durchgeführt und speichert die Symboltabelle so wie sie bis zu diesem Punkt generiert wurde. Das erste Argument ist die IEC-Gerätenummer, normalerweise 8 für die Floppydisk. Die Sekundäradresse kann dann zwischen 2 und 14 liegen. Der Dateinamen wird wie bei einem OPEN-Befehl angegeben, erfodert also ein ",S,W" am Ende des Namens für sequentiell und write. Dieser Pseudoop wird auch benötigt, wenn man ein sortiertes Listing der Symbole und Labels erzeugen will. Diese Datei kann als Eingabe für das Programm zum Symbollisten dienen. Der .SST-Befehl ist z.B. auch dann nützlich, wenn Sie Quellprogramme unabhängig voneinander assemblieren, die jedoch auf Unterroutinen des anderen Programms zugreifen müssen. Speichern Sie dazu einfach die Symboltabelle mit .SST am Ende des Assemblerprogramms ab und lesen Sie sie zu Beginn des anderen Programms mit .LST wieder ein.

.LST Gerätenummer, Sekundäradresse, "Filename"

Dies ist das Gegenstück zu .SST. Hierbei kann man eine Symboltabelle laden, die von einen anderen Programm erzeugt wurde, z.B. eine Tabelle von Betriebssystemroutinen. Auf doppelte Symbole wird dabei nicht geprüft; der zuerst definierte Wert wird dabei nur überschrieben. Auch Überlauf der Symboltabelle wird beim Laden nicht erkannt, gibt jedoch einen Fehler, sobald Sie danach ein neues Symbol definieren.

.FLP

Falls Sie öfters die Fließkommaarithmetik des BASIC-Interpreters benutzen, können Sie mit dem Befehl .FLP auf einfache Weise Fließkommakonstanten im internen 5-Byte-Format im Objectcode ablegen. Das vereinfacht die Anwendung der Fließkommaroutinen bedeutend. Hinter dem .FLP-Befehl können eine oder durch Komma getrennt mehrere Fließkommakonstanten folgen, z.B.

```
.FLP 10, 1E8
```

Jede Fließkommazahl belegt 5 Bytes; unser Beispiel generiert also 10 Bytes. Beachten Sie jedoch, daß im Listing immer nur drei Bytes des Objectcodes erscheinen.

.END

Dieser Pseudoop beendet ein Quellprogramm und ist optional, also nicht erforderlich. .END veranlaßt ein .GTB am Ende von Pass zwei; weiterer BASIC-Text danach würde also abgearbeitet. Sie könnten also mit SYS Ihr gerade assembliertes Maschinenprogramm aufrufen.

.END kann aber auch ein optionales Argument haben. Dieses Argument hat dasselbe Format wie .FILE und veranlaßt den Assembler bei verketteten Programmen am Ende von Pass 1 das erste Programm wieder zu laden und ab der Zeile, die den SYS 32768 enthielt, mit Pass zwei weiterzumachen. Der Filenamen muß also der Name des ersten Programms in der Kette sein (welches das SYS 32768 enthält). In Pass 2 hat der Filename dann keine Auswirkung mehr.

.SYM

Diesen Pseudoop können Sie benutzen, wenn Sie nach dem Assemblieren eine Liste aller definierten Symbole mit Ihren Werten haben wollen. Diese Liste wird entsprechend der Ausgabeoption (.OPT P, siehe dort) auf den Bildschirm oder in eine Druckdatei gegeben. In einer Zeile werden 4 Symbole mit ihren Werten in hexadezimaler Form ausgegeben. Wenn Sie eine andere Anzahl pro Zeile wünschen, können Sie diese zusammen mit dem .SYM-Befehl angeben, z.B. .SYM 2, wenn Sie auf dem Bildschirm arbeiten. Die Symbole werden dabei in der umgekehrten Reihenfolge der Definition ausgegeben. Wollen Sie eine alphanumerisch sortierte Liste, so müssen Sie die Symboltabelle mit .SST abspeichern und mit dem Programm SYMPRINT auf Ihrer PROFI-MAT-Diskette die Tabelle ausgeben.

Haben Sie mit Macros gearbeitet, so werden sämtliche Macronamen ebenfalls ausgegeben. Bei jedem Macro steht zusätzlich noch, wie oft es im Programm aufgerufen wurde, ebenfalls als zweistellige Hexzahl.

.PAG

Dieser Pseudoop hat drei verschiedene Funktionen und dient zur Formatkontrolle des Assemblerlistings. Ohne weitere Parameter bewirkt er einen Seitenvorschub im Assemblerlisting. Damit können Sie also bestimmte Abschnitte des Assemblerlistings auf jeweils einer neuen Seite beginnen lassen. PROFI-ASS fügt nach jeder 60. Zeile automatisch einen Seitenvorschub im Listing ein und beginnt die nächste Seite mit einer Überschrift sowie der laufenden Seitennummer. Haben Sie andere Blattlängen, so können Sie die Anzahl der Zeilen pro Seite ebenfalls mit dem .PAG-Befehl bestimmen, z.B.

```
.PAG 64
```

Dadurch wird PROFI-ASS 64 angewiesen, 64 Zeilen auf eine Seite zu schreiben. Es werden Werte bis zu 255 akzeptiert. Eine weitere Funktion ist die Bestimmung des linken Randes. Dies ist gedruckten Listings, die man abheften will, sehr nützlich. Der zweite Parameter von .PAG gibt an, wieviel Leerzeichen vor jeder Assemblerzeile ausgedruckt werden sollen. Der Standardwert ist null. Mit

```
.PAG ,10
```

werden die Listings um zehn Zeichen eingerückt. Das Komma ist erforderlich, um den zweiten Parameter zu kennzeichnen. Man kann die obigen Befehle auch zu einem

kombinieren:

```
.PAG 64,10
```

.TIT

Hiermit können Sie zu der Standardüberschrift

```
PROFI-ASS 64 V2.0     SEITE 1
```

die auf jeder Seite erscheint, noch einen eigenen Text hinzufügen. Dieser Text wird mit dem .TIT-Befehl innerhalb von Anführungszeichen angegeben, z.B.

```
.TIT "HARDCOPY-ROUTINE"
```

Dieser Text wird dann vor die Standardüberschrift gesetzt, wir erhalten also

```
HARDCOPY-ROUTINE     PROFI-ASS 64 V2.0     SEITE 1
```

.OPT

Der .OPT - Pseudoop steht für OPTion und gibt Ihnen die Entscheidung über das Assemblerlisting und den Objektcode. Die Syntax ist folgende:

```
.OPT option,option,option ...
```

Folgende Optionen stehen zur Verfügung:

P - Print. Diese Option wählen Sie, wenn das Assemblerlisting auf dem Bildschirm ausgeben werden soll. Alle weiteren P-Optionen (s.u.) geben zusätzlich auch auf Bildschirm aus, da der Bildschirm im allgemeinen das schnellste Ausgabemedium ist. Das Listing wird automatisch formatiert. Unabhängig von der P Option werden Zeilen, die einen Fehler oder einen .FILE-Befehl enthalten, in Pass eins und zwei immer ausgeben.

P# - Print to File. Mit dieser Option können Sie ein Listing z.B. zum Drucker schicken. Dazu müssen Sie zu Programmbeginn vor dem SYS 32768 mit einem OPEN-Befehl eine logische Datei für den Drucker öffnen, z.B. OPEN 1,4. Die logische Filenummer (in unserem Beispiel 1) wird dann für das Doppelkreuz eingesetzt, z.B. .OPT P1. Durch diese Technik können Sie nach einem entsprechenden OPEN-Befehl Ihr Assemblerlisting auch auf Diskette oder Kassette schreiben. Durch die Wahl der logischen Filenummer können Sie wie in BASIC bestimmen, ob nach jedem Wagenrücklauf (CHR$(13)) noch ein Zeilenvorschub (CHR$(10)) gesandt werden soll. Dies erreicht man, indem man beim Öffnen der Druckdatei eine logische Filenummer größer als 127 wählt, z.B. OPEN 130,4 und dann .OPT P130 benutzt.

P=Ausdruck - Mit dieser Option können Sie die Ausgabe über eine eigene Routine abwickeln. Der Ausdruck muß dazu die Startadresse Ihres Programms ergeben. Das auszugebende Zeichen wird im Akku übergeben. Eine Null zeigt dabei das letzte Zeichen an (File schließen). Damit besteht die Möglichkeit, eigene Ausgabetreiber zu bedienen (z.B. über den USER-Port).

O - Object bedeutet Objektcode-Ausgabe. Ohne weitere Zeichen geht der Objektcode in einen speziellen Puffer direkt oberhalb des Assemblerprogramms, wo normalerweise die Arrayvariablen liegen; es werden dazu auch die gleichen Zeiger verwendet.

OO - Object at Origin. Diese Option schreibt den Objektcode direkt an die Speicherstellen, für den er geschrieben wurde. Dies ist sehr nützlich zum schnellen Austesten von Programmen und erlaubt alle Freiheiten beim Bewegen des Programmzeigers. Das Abspeichern des Codes mit Hilfe eines Monitors auf Kassette ist damit auch möglich. Soll ein Assemblerprogramm in dem Speicherbereich laufen, in dem das Quellprogramm steht, so läßt sich diese Methode natürlich nicht anwenden.

O# - Ähnlich P# erlaubt dies die Ausgabe des Objectcodes in eine IEC-Datei. Die Datei muß wieder vorher geöffnet werden, und zwar als Programmdatei zum Schreiben (Sekundäradresse 1), z.B. OPEN 1,8,1,"Programm". Mit .OPT O1 ginge der Objectcode in diese Datei. Zuerst schreibt PROFI-ASS 64 die Startadresse in die Datei und dann den generierten Code. Wird der Assembliervorgang normal beendet, so wird die Programmdatei wieder geschlossen. Das so erzeugte Maschinenprogramm kann direkt mit LOAD oder von einem Monitor aus wieder geladen werden. Beachten Sie, daß .OPT O# mit Kassette nicht möglich ist. Siehe dazu auch den nächsten Abschnitt und den Anhang.

O=Ausdruck. Dies erlaubt die Ausgabe des Objectcodes über eine eigene Routine mit der selben Syntax wie beim .OPT P= Befehl. Die Object-Ausgabe-Routine muß jedoch etwas komplizierter sein, da sie nur einmal pro Assemblerzeile aufgerufen wird. Einige Symbole, die dazu benötigt werden, sind im Anhang aufgelistet. Das wichtigste davon ist LENGHT, welches die Anzahl der auszugebenen Bytes minus eins angibt. Wenn LENGHT z.B. null ist, muß ein Byte ausgeben werden. Ihre Routine muß noch auf zwei spezielle Werte testen. Ein Wert von $80 bedeutet, daß das Object-File geöffnet werden muß; zu diesem Zeitpunkt können Sie die Startadresse übertragen. Ein Wert von $C0 bedeutet 'Schließen der Datei'. Ansonsten enthält LENGHT eine kleine Zahl von 0 an aufwärts. Die auszugebenden Daten sind an zwei Stellen gespeichert. Die ersten drei Bytes stehen in der Zero-Page ab Adresse OP. Werden mehr als drei Bytes Objectcode erzeugt (z.B. mit .BYTE, .WORD oder .ASC), so stehen die zusätzlichen Bytes ab Adresse OBJBUF. Ihre Ausgaberoutine kann alle Register und alle Flags ändern (mit Ausnahme des Dezimalflags). Beim Gebrauch der Zero-Page ist jedoch Vorsicht geboten. Im Anhang ist ein Programm aufgelistet, das die Ausgabe des Objectcode im Hex-Format auf ein beliebiges File ermöglcht. Damit wäre es prinzipiell auch möglich, den Objectcode direkt auf Datasette abspeichern.

M - Wenn Sie mit Macros arbeiten, so können Sie entscheiden, ob bei jedem Macroaufruf nur die Zeile mit dem Macroaufruf oder auch die Zeilen gelistet werden, die die Macrodefinition mit den aktuellen Parametern enthalten. Geben Sie nichts an, so wird bei jedem Macroaufruf das komplette Macro gelistet. Mit .OPT M können Sie dies unterdrücken und nur die Zeile mit dem Macroaufruf erscheint dann im Listing.

N - Sie können mit .OPT N die Ausgabeoptionen jederzeit wieder löschen. N löscht alle Optionen einschließlich der M-Option. Soll eine Option erhalten bleiben oder wieder eingeschaltet werden, so schreiben Sie nochmal diese Option. Wenn Sie zum Beispiel den Druck abschalten

wollen, die Objectcodeausgabe jedoch weiterhin auf File 2 gehen soll, so schreiben Sie:

```
.OPT N,O2
```

und

```
.OPT P
```

wenn das Listing wieder ausgegeben werden soll.

Ein Beispielprogramm

Das folgende Beispielprogramm schreibt den Inhalt der Zeropage ab Zeile LINE auf den Bildschirm. Es zeigt den generellen Aufruf und Umgang mit dem Assembler.

```
10 SYS 32768 ; PROFI-ASS 64 AUFRUFEN
20 .OPT P,OO ; OBJECTCODE IN SPEICHER
30 *= $C000 ; PROGRAMMSTARTADRESSE
40 LINE = 10 ; BILDSCHIRMZEILE
50 VIDEO = $400 ; ADRESSE DES BILDSCHIRMSPEICHERS
60 COLOR = $D800 ; ADRESSE DES FARBRAMS
70 FARBE = 1 ; WEISSE SCHRIFT
80 LDX #0
90 SCHLEIFE LDA 0,X ; BYTE AUS ZEROPAGE HOLEN
100 STA VIDEO+(40*LINE),X ; ZEICHEN AUSGEBEN
110 LDA #FARBE
120 STA COLOR+(40*LINE),X ; FARBE SETZEN
130 INX
140 BNE SCHLEIFE
150 RTS
160 .END
```

Wenn Sie nun das Programm mit 'RUN' starten, erscheint auf dem Bildschirm dabei folgendes Listing:

```
2
PROFI-ASS 64 V2.0    SEITE 1

20:    C000                  .OPT P,OO    ; OBJECTCODE IN SPEICHER
30:    C000                  *=   $C000   ; PROGRAMMSTARTADRESSE
40:    000A          LINE    =    10      ; BILDSCHIRMZEILE
50:    0400          VIDEO   =    $400    ; ADRESSE DES BILDSCHIRMSPEICHERS
60:    D800          COLOR   =    $D800   ; ADRESSE DES FARBRAMS
70:    0001          FARBE   =    1       ; WEISSE SCHRIFT
80:    C000 A2 00             LDX  #0
90:    C002 B5 00    SCHLEIFE LDA  0,X     ; BYTE AUS ZEROPAGE HOLEN
100:   C004 9D 90 05          STA  VIDEO+(40*LINE),X  ; ZEICHEN AUSGEBEN
110:   C007 A9 01             LDA  #FARBE
120:   C009 9D 90 D9          STA  COLOR+(40*LINE),X  ; FARBE SETZEN
130:   C00C E8                INX
140:   C00D D0 F3             BNE  SCHLEIFE
150:   C00F 60                RTS
]C000-C010
NO ERRORS
```

Im folgenden Beispiel wird der Objectcode direkt auf Diskette und das Listing zum Drucker geschickt. Das Quellprogramm besteht aus mehreren einzelnen Programmen.

```
10 OPEN 1,8,1, "0:OBJECTCODE"
20 OPEN 2,4 : REM DRUCKER
30 SYS 32768
40 .OPT O1,P2
50 ; ASSEMBLERBEFEHLE
...
1000 .FILE 8, "PROGRAMM 2"
```

Programm 2 enhält

```
10 ; WEITERE BEFEHLE
...
1000 .FILE 8, "PROGRAMM 3"
```

Programm 3 enhält

```
10 ; WEITERE BEFEHLE
...
1000 .END 8, "PROGRAMM 1"
```

wobei Programm 1 das Programm ist, welches das SYS 32768 enthält.

MACROS

Nach dem wir bis jetzt die meisten Assembleranweisungen kennengelernt haben, kommen wir nun zu einer besonders leistungsfähigen Eigenschaft von PROFI-ASS 64, den Macros. Was sind Macros und wozu benutzt man Sie?

Mit den Macros haben wir die Möglichkeit, eine beliege Anzahl von Befehlen und Assembleranweisungen zusammenzufassen und ihnen eine Namen zu geben. Hat man einmal ein Macro in dieser Weise definiert, so kann man später diese Anweisungsfolge beliebig oft durch den Assembler in das Quellprogramm einfügen lassen; es genügt dazu lediglich die Angabe des Macronamens. Ein Beispiel soll dies deutlich machen.

In Maschinenprogrammen steht man oft vor der Aufgabe, den Inhalt eines 16-Bit-Zeigers zu erhöhen, der aus zwei aufeinander folgenden Zeropage-Adressen besteht. Die Befehlsfolge dazu sieht so aus:

```
        INC ZEIGER
        BNE LABEL
        INC ZEIGER+1
LABEL   ...
```

An einer anderen Stelle müssen Sie die Variable TEMP erhöhen:

```
        INC TEMP
        BNE LABEL1
        INC TEMP+1
LABEL1  ...
```

Mit Macros haben wir nun die Möglichkeit, eine derartige Anweisungsfolge einmal zu definieren und später immer wieder darauf zurückzugreifen. Dazu gibt es zwei neue Pseudops. Der erste dient zur Einleitung einer Macrodefinition, der zweite beendet die Definition. Damit man sich später auf das Macro beziehen kann, muß es einen Namen bekommen. Dabei gelten die gleichen Konventionen wie bei den anderen Symbolen (erstes Zeichen ein Buchstabe, danach Buchstaben, Ziffern oder Punkt, acht signifikante Stellen). Unsere Definition sieht dann so aus:

```
INC.PNT  .MAC ADRESSE
         INC  ADRESSE
         BNE  .LABEL
         INC  ADRESSE+1
LABEL    .MEND
```

Unser Macro heißt also INC.PNT. Mit dem Pseudoop .MAC wird eine Macrodefinition eingeleitet. Dann können Parameter folgen, bei uns ADRESSE genannt. Nun folgt der Code so wie er auch im normalen Text geschrieben wird. Eine Besonderheit finden wir in der Zeile BNE .LABEL. Jedesmal, wenn wir uns auf ein Label beziehen, das innerhalb des Macros definiert wird, müssen wir vor den Namen einen Punkt schreiben. Die letzte Zeile enthält also die Labeldefinition und gleichzeitig das Ende der Definition mit .MEND. Jetzt können wir das neu definierte Macro aufrufen:

```
        'INC.PNT ZEIGER
```

Diese eine Zeile ersetzt nun die obige Anweisungsfolge. Wir schreiben also lediglich einen Apostroph gefolgt vom Macronamen sowie eventuellen Parametern.

In unserem Fall war es ein Parameter; ein Macro kann auch gar keine oder mehrere Parameter enthalten, in letzteren Fall werden mehrere Parameter durch Komma getrennt. Ein Macro ohne Parameter sehen Sie im nächsten Beispiel.

```
RAM   .MAC
      SEI
      LDA $01
      AND #%11111110
      STA $01
      .MEN
```

Dieses Macro benötigt keine Parameter und auch keine sogenannte lokale Label, also Label innerhalb der Macrodefintion. Macros ohne Parameter erzeugen jedesmal den selben Code und ließen sich prinzipiell durch Unterprogramme ersetzen. Hier noch kurz etwas grundsätzliches zum Unterschied zwischen Macros und Unterprogrammen. Macros sind Hilfsmittel während der Assemblierung und erzeugen bei jedem Aufruf Objectcode. Unterprogramme kann man dagegen als Hilfsmittel zur Ausführungszeit betrachten und stehen nur einmal im Objectprogramm.

Durch die geschickte Anwendung von Macros besonders auch in Kombination mit der bedingten Assemblierung, kann man sich die Assemblerprogrammierung sehr komfortabel gestalten. Hat man für die verschiedenen Grundaufgaben Macros bereit, so kann das eigentliche Hauptprogramm lediglich das Grundgerüst enthalten sowie die Aufrufe der Macros.

Hier noch einige Hinweise zur Anwendung von Macros.

Macros müssen grundsätzlich zu Beginn des Assemblerprogramms definiert werden, ehe man sie aufrufen kann. Arbeiten Sie mit .FILE mit verketteten Quellprogrammen, so müssen die Macros im ersten Programm stehen. Wenn man innerhalb von Macros Label definiert, so muß beim Bezug darauf wie bereits erwähnt, vor dem Namen des Labels ein Punkt folgen. Dies gilt auch innerhalb von Ausdrücken. Solche Labels werden nur mit sechs signifikanten Stellen geführt. Rufen Sie solche Macros mehrmals auf und lassen Sie sich die Symboltabelle ausgeben, so werden die Labels mehrmals mit den jeweils unterschiedlichen Werten gelistet. Zur Unterscheidung wird an den Namen durch Doppelpunkt getrennt die Nummer des Aufrufes mit angegeben, z.B.

```
LABEL:00   0006   LABEL:01   C020   LABEL:02   C035
```

Die Nummer Null bezeichnet den Labelwert innerhalb der Defintion, realtiv zum Macrobeginn.

Werden innerhalb von Macros Label definiert, so müssen in unterschiedlichen Macros auch unterschiedliche Namen verwendet werden, so kommt es zu einem 'REDEFINITION ERROR'. Parameter können Sie jedoch gleich benennen, da diese bei jedem Macroaufruf mit den aktuellen Werten vorsorgt werden. Bei Macroaufruf können selbstverständlich beliebige PROFI-ASS Ausdrücke verwenden; diese werden vom Assembler brechnet und den Parametern übergeben, z.B.

```
  'INC.PNT  ZEIGER-8*2
```

Hier wird z.B. der Wert der Symbols ZEIGER genommen, davon 8 abgezogen sowie das Ergebnis mit 2 malgenommen. Die Reihenfolge der Abarbeitung können Sie wie üblich durch Setzen von Klammern bestimmen.

Als Beispiel haben wir nun ein Programm, daß praktisch nur aus Macroaufrufen besteht. Zwei Macros wurden definiert. Das erste dient zum Setzen des Cursors. Das Betriebsystem des Commodore 64 stellt uns dazu geeignete Routinen zur Verfügung. Das Macro mit dem Namen Cursor erwartet nun zwei Parameter; zum einen die Zeile, in die der Cursor gesetzt werden soll, sowie die zugehörige Spalte. Wollen wir nun in unserem Programm den Cursor an eine bestimmte Position setzen, so genügt der Aufruf des Macros, z.B.:

```
'CURSOR 10,20
```

Das zweite Macro dient zur Ausgabe beliebiger Texte. Als Parameter dient die Adresse des Textes. Der Text muß mit einem Nullbyte abgeschlossen werden.

Im Programm finden Sie nun erst die Definitionen der beiden Macros und anschließend das eigentliche Programm, das lediglich aus vier Macroaufrufen sowie einem RTS besteht. Anschließend sind noch die Texte im Programm abgelegt.

Nach dem Quellprogramm finden Sie das Assemblerlisting.

```
100 SYS 32768
110 .OPT P,OO
120 ; DEMOPROGRAMM FUER MACROS
130 ;
140 ; CURSOR SETZEN
150 CURSOR .MAC ZEILE,SPALTE
160 LDX #ZEILE
170 LDY #SPALTE
180 STX $D6
190 STY $D3
200 JSR SETCRSR ; CURSOR SETZEN
210 .MEN
220 ;
230 ; STRING AUSGEBEN
240 PRTSTR .MAC TEXT
250 LDA #<TEXT
260 LDY #>TEXT
270 JSR STROUT ; TEXT AUSGEBEN
280 .MEN
290 ;
300 SETCRSR = $E56C
310 STROUT = $AB1E
320 ;
330 *= $C000
340 ;
350 'CURSOR 10,10
360 'PRTSTR TEXT1
370 'CURSOR 0,20
380 'PRTSTR TEXT2
390 RTS
400 ;
410 TEXT1 .ASC "TEXT NR. 1" : .BYT 0
420 TEXT2 .ASC "TEXT NR. 2" : .BYT 0
430 ;
440 .END
```

Hier nun das Assemblerlisting

```
PROFI-ASS 64 V2.0    SEITE 1

110:   C000                    .OPT P,OO
120:                   ; DEMOPROGRAMM FUER MACROS
130:                   ;
140:                   ; CURSOR SETZEN
150:   '               CURSOR  .MAC ZEILE,SPALTE
160:   '                       LDX  #ZEILE
170:   '                       LDY  #SPALTE
180:   '                       STX  $D6
190:   '                       STY  $D3
200:   '                       JSR  SETCRSR  ; CURSOR SETZEN
210:   '                       .MEN
220:                   ;
230:                   ; STRING AUSGEBEN
240:   '               PRTSTR  .MAC TEXT
250:   '                       LDA  #<TEXT
260:   '                       LDY  #>TEXT
270:   '                       JSR  STROUT  ; TEXT AUSGEBEN
280:   '                       .MEN
290:                   ;
300:   E56C            SETCRSR =    $E56C
310:   AB1E            STROUT  =    $AB1E
320:                   ;
330:   C000                    *=   $C000
340:                   ;
350:   C000                    'CURSOR 10,10
+      C000 A2 0A              LDX  #ZEILE
+      C002 A0 0A              LDY  #SPALTE
+      C004 86 D6              STX  $D6
+      C006 84 D3              STY  $D3
+      C008 20 6C E5           JSR  SETCRSR  ; CURSOR SETZEN
+      C00B                    .MEN
360:   C00B                    'PRTSTR TEXT1
+      C00B A9 25              LDA  #<TEXT
+      C00D A0 C0              LDY  #>TEXT
+      C00F 20 1E AB           JSR  STROUT  ; TEXT AUSGEBEN
+      C012                    .MEN
370:   C012                    'CURSOR 0,20
+      C012 A2 00              LDX  #ZEILE
+      C014 A0 14              LDY  #SPALTE
+      C016 86 D6              STX  $D6
+      C018 84 D3              STY  $D3
+      C01A 20 6C E5           JSR  SETCRSR  ; CURSOR SETZEN
+      C01D                    .MEN
380:   C01D                    'PRTSTR TEXT2
+      C01D A9 30              LDA  #<TEXT
+      C01F A0 C0              LDY  #>TEXT
+      C021 20 1E AB           JSR  STROUT  ; TEXT AUSGEBEN
+      C024                    .MEN
390:   C024 60                 RTS
400:                   ;
410:   C025 54 45 58 TEXT1     .ASC "TEXT NR. 1"
410:   C02F 00                 .BYT 0
```

```
420:   C030 54 45 58 TEXT2    .ASC "TEXT NR. 2"
420:   C03A 00                .BYT 0
430:                ;
]C000-C03B
NO ERRORS
```

Sehen wir uns das Listing einmal näher an. Sie erkennen, daß innerhalb der Macrodefinition anstelle des Programmzählers ein Apostroph ' erscheint. Das Feld für den Objectcode ist leer, da ja durch die Definition des Macros noch kein Code erzeugt wird (Zeile 150 - 210, 240 - 280).

In Zeile 350 ist dann der erste Macroaufruf. Der aktuelle Programmzeiger sowie der erzeugte Code erscheint nun im Listing. Anstelle der Zeilennummer erscheint nun ein Plus-Zeichen +, das Ihnen anzeigt, daß der erzeugte Kode aus dem Macroaufruf stammt. Sie erkennen, daß die Symbole ZEILE und SPALTE die Werte haben, die sie im Macroaufruf zugewiesen bekommen haben. Analog werden auch die weiteren Macroaufrufe wiedergegeben.

Haben Sie viele Macros in Ihren Quellprogrammen oder werden einige Macros sehr oft aufgerufen, so gibt es die Möglichkeit, den vom Macro erzeugten Code nicht im Listing erscheinen zu lassen. Es wird dann lediglich die Zeile mit dem Macroaufruf ausgegeben. Dazu müssen Sie bei den Optionen .OPT M angegeben. Sehen Sie dazu das nächste Beispiel.

```
PROFI-ASS 64 V2.0    SEITE 1

110:   C000                  .OPT P,M,OO
120:                ; DEMOPROGRAMM FUER MACROS
130:                ;
140:                ; CURSOR SETZEN
150:   '            CURSOR   .MAC ZEILE,SPALTE
160:   '                     LDX  #ZEILE
170:   '                     LDY  #SPALTE
180:   '                     STX  $D6
190:   '                     STY  $D3
200:   '                     JSR  SETCRSR  ; CURSOR SETZEN
210:   '                     .MEN
220:                ;
230:                ; STRING AUSGEBEN
240:   '            PRTSTR   .MAC TEXT
250:   '                     LDA  #<TEXT
260:   '                     LDY  #>TEXT
270:   '                     JSR  STROUT  ; TEXT AUSGEBEN
280:   '                     .MEN
290:                ;
300:   E56C          SETCRSR  =    $E56C
310:   AB1E          STROUT   =    $AB1E
320:                ;
330:   C000                  *=   $C000
340:                ;
350:   C000                  'CURSOR 10,10
360:   C00B                  'PRTSTR TEXT1
370:   C012                  'CURSOR 0,20
380:   C01D                  'PRTSTR TEXT2
390:   C024 60               RTS
```

```
400:                    ;
410:   C025 54 45 58 TEXT1     .ASC "TEXT NR. 1"
410:   C02F 00                 .BYT 0
420:   C030 54 45 58 TEXT2     .ASC "TEXT NR. 2"
420:   C03A 00                 .BYT 0
430:                    ;
]C000-C03B
NO ERRORS
```

Dadurch wird das Listing kürzer und oft auch übersichtlicher. Im nächsten Beispiel haben wir noch dem .SYM-Befehl benutzt, der neben den Symbolen und ihren Werten auch die definierten Macro ausgibt sowie die Anzahl ihrer Aufrufe als zweistellige Hexadezimalzahl.

```
PROFI ASS 64 V2.0    SEITE 1

110:   C000                     .OPT P,M,OO
115:   C000                     .SYM
120:                    ; DEMOPROGRAMM FUER MACROS
130:                    ;
140:                    ; CURSOR SETZEN
150:   '                CURSOR  .MAC ZEILE,SPALTE
160:   '                        LDX  #ZEILE
170:   '                        LDY  #SPALTE
180:   '                        STX  $D6
190:   '                        STY  $D3
200:   '                        JSR  SETCRSR  ; CURSOR SETZEN
210:   '                        .MEN
220:                    ;
230:                    ; STRING AUSGEBEN
240:   '                PRTSTR  .MAC TEXT
250:   '                        LDA  #<TEXT
260:   '                        LDY  #>TEXT
270:   '                        JSR  STROUT  ; TEXT AUSGEBEN
280:   '                        .MEN
290:                    ;
300:   E56C             SETCRSR =    $E56C
310:   AB1E             STROUT  =    $AB1E
320:                    ;
330:   C000                     *=   $C000
340:                    ;
350:   C000                     'CURSOR 10,10
360:   C00B                     'PRTSTR TEXT1
370:   C012                     'CURSOR 0,20
380:   C01D                     'PRTSTR TEXT2
390:   C024 60                  RTS
400:                    ;
410:   C025 54 45 58 TEXT1     .ASC "TEXT NR. 1"
410:   C02F 00                 .BYT 0
420:   C030 54 45 58 TEXT2     .ASC "TEXT NR. 2"
420:   C03A 00                 .BYT 0
430:                    ;
]C000-C03B
NO ERRORS
```

```
PROFI-ASS 64 V2.0    SEITE 2

SYMBOLTABLE:
TEXT2      C030    TEXT1      C025    TEXT       C030    SPALTE     0014
ZEILE      0000    STROUT     AB1E    SETCRSR    E56C

7 SYMBOLS DEFINED

MACROTABLE:
PRTSTR     02      CURSOR     02
2 MACROS DEFINED
```

Fehlermeldungen

PROFI-ASS 64 hat eine Reihe von Fehlermeldungen, die sowohl in Pass eins als auch in Pass zwei ausgegeben werden. Erkennt der Assembler einen Fehler, so werden 4 Sterne gefolgt von der Fehlermeldung ausgegeben. Danach wird die fehlerhafte Zeile auf dem Bildschirm ausgegeben, unabhängig von den P-Optionen. Zusätzlich wird sie noch entsprechend der gewählten P-Option ausgegeben. Bei einem Syntax Error wird vor den vier Sternen noch eine Ziffer ausgegeben, die den Fehler näher beschreibt. Es gibt 10 verschiedene Syntax Error, die unten aufgelistet sind. Bei Macros können noch weitere Syntaxfehler auftreten, die durch einen vorangestellten Buchstaben gekennzeichnet sind.

Einige Fehler sind "fatal", das heißt sie führen zum Abbruch der Assemblierung wenn sie auftreten. Bei fatalen Fehler wird vorher eine Zeile mit Ausrufungszeichen (!) ausgegeben. Der Assembliervorgang wird nach Ausgabe der Fehlermeldung abgebrochen. Bei einem Fehler wird das erste Byte des erzeugten Objectcodes zu Null, was der 6502 BRK-Befehl ist. Wenn Sie versuchen, ein solches Programm auszuführen, wird beim Durchlaufen der fehlerhaften Zeile ein BRK-Befehl ausgeführt, was entweder zu einem Warmstart oder, falls Sie einen Monitor geladen haben, in den Monitor führt. Im allgemeinen sollten Sie jedoch den Fehler erst korrigieren, bevor Sie ein Assemblerprogramm ausführen.

Ein Fehlertyp, den PROFI-ASS 64 nicht entdecken kann, ist der sogenannte Phasenfehler. Dieser Fehler tritt normalerweise nie auf, kann jedoch bei bestimmten Konstellationen bei der bedingten Assemblierung mit .BYTE oder .WORD Pseudoops in besonderen Fällen auftreten. Ein Phasenfehler entsteht dann, wenn sich der Wert des Programmzählers in Pass eins von dem in Pass zwei unterscheidet. Dadurch entsteht dann unbrauchbarer Code. Sie können einen Phasenfehler jedoch mit einer .IF-Anweisung erkennen:

```
PHASE .IF PHASE-* : PHASE ERROR
```

Normalerweise hat PHASE immmer den Wert des Programmzählers und der Code hinter dem Doppelpunkt wird nie assembliert. Tritt jedoch eine Phasenverschiebung auf, ist das Resultat nicht Null und das zusätzliche Statement gibt einen Syntax Error, den Sie erkennen können.

Fehlerstatistik

Vor dem Beginn des zweiten Pass gibt PROFI-ASS die Anzahl der Fehler in Pass 1 aus, sofern dort welche aufgetreten sind, z.B.

```
2 ERRORS IN PASS 1
```

Nach Pass zwei, wenn der Assemblerdurchlauf komplett ist, erscheint immer die Anzahl der Fehler im zweiten Pass. War der Assemblerlauf fehlerfrei, so heißt es

```
NO ERRORS
```

sind dagegen Fehler aufgetreten, so wird deren Anzahl ausgegeben, z.B.

```
4 ERRORS
```

Meldungen

SYNTAX - Dieser Fehlermeldung geht eine Ziffer voraus, die den Fehler näher beschreibt. Diese Ziffern haben folgende Bedeutung:

0 - Label für Leeranweisung nicht erlaubt (Die Zeile enthält nur eine Zeichenkette).
1 - Ungültiger Opcode
2 - Ungültige Adressierungsart - diesen Befehl gibt es nicht mit dieser Adressierungsart.
3 - Unbekannter Operator in einem Ausdruck (nicht erlaubte Zeichen in einem Ausdruck).
4 - Unpaarige Klammern.
5 - Ungültiger Ausdruck - ungültiges Zeichen in einem Ausdruck oder ein Leerstring "".
6 - Fehlendes Komma - von einem Pseudoop wurde ein Komma erwartet.
7 - Ungültiger Pseudoop. Der .XXX String wurde nicht als Pseudoop erkannt.
8 - Symbol beginnt nicht mit einem Buchstaben. Es wurde ein Symbol erwartet aber ein nicht alphabetisches Zeichen gefunden.
9 - Opcode mit unzulässiger Adressierungsart.

Bei Macros können noch folgende Syntaxfehler auftreten:

B - .MEND-Befehl ohne vorheriges .MAC.
C - Nicht abgeschlossene Macrodefinition.
D - Verschachtelte Macrodefinition - Macros innerhalb von Macros sind nicht zulässig.
F - Falsche Parameterzahl. Die Anzahl der Parameter in der Macrodefinition und beim Aufruf stimmen nicht überein.

ILLEGAL QUANTITY - Ihr Ausdruck ergibt einen Wert, der außerhalb der erlaubten Grenzen für diesen Befehl oder Pseudoop liegt. Der Ausdruck ergibt einen Wert größer als 65535.

OVERFLOW - Der Eingabepuffer, den PROFI-ASS 64 benutzt, um Ihre Zeile zu dekodieren, ist zu klein. Teilen Sie in einem solchen Fall die Zeile in mehrere Anweisungen auf oder benutzen Sie temporäre Variablen, um den Ausdruck zu vereinfachen.

BRANCH OUT OF RANGE - Ein relativer Sprung (Branch-Befehl) über eine Distanz von mehr als 128 Byte wurde versucht.

REDEFINITION - Es wurde der Versuch gemacht, ein Symbol zweimal zu definieren ohne den Redefinitionsoperator zu benutzen.

UNDEF'D STATEMENT - Ein Label in einem Ausdruck ist nicht definiert.

REVERSAL - Es wurde der Versuch unternommen, Code an eine Adresse zu assemblieren, die niedriger als die letzte Adresse war. Dieser Fehler tritt nicht auf, wenn Sie direkt in den Speicher assemblieren. Dies ist ein fataler Fehler, genauso wie alle folgenden.

SYM TABLE OVERFLOW - Sie haben versucht, mehr Symbole zu definieren als Platz in der Symboltabelle ist. Setzen Sie entweder das Minimum mit .STM niedriger oder teilen Sie Ihr Programm in mehrere Teilprogramme auf. Diese Fehlermeldung kann auh beim Nachladen eines Quellprogramms mit .FILE

erscheinen, falls das nachgeladen Programm zu groß war und die Symboltabelle teilweise überschrieben hat. Teilen Sie auch hier Ihr Quellprogramm in mehrere Teilprogramme auf.

OUT OF MEMORY - Der Puffer für den Objectcode (.OPT O Modus) ist zu klein. Sie können z.B. Diskettenausgabe wählen.

UNDEF'D STATEMENT - Ein .GOTO zu einer nicht vorhandenen Zeile im Programm (genau wie in BASIC). Im Gegensatz zu dem gleichnamigen obigen Fehler ist dieser Fehler fatal und führt zum Abbruch.

DEVICE NOT PRESENT - Das angesprochene IEC-Bus Gerät ist nicht am Bus oder antwortet nicht.

IEEE - ein anderer Fehler auf dem IEC-Bus.

DISK - Diskettenfehler. Die Fehlermeldung des Diskettenlaufwerks wird vorher ausgegeben.

Anhang

Das folgende Quellprogramm soll ein weiteres Beispiel für die Anwendung von PROFI-ASS sein. Es demonstriert die Objectcode-Ausgabe über eine eigene Routine. Dabei soll jedes Byte im Hexformat auf ein vorher geöffnetes File mit der logischen Nummer eins gehen. Damit ist es z.B. möglich, den Objectcode direkt auf Datasette zu schreiben. Mit einen weiter unten abgedruckten BASIC-Programm ist das Einlesen von Code in diesem Format möglich.

```
100 SYS 32768 ; AUFRUF DES ASSEMBLERS
110 .OPT P,OO
120 LENGHT = $4E ; AUSZUGEBENDE BYTES MINUS 1
130 OP = $4B ; PUFFER FUER DIE ERSTEN DREI BYTES
140 ADR = $56 ; PROGRAMMSTARTADRESSE
150 OBJBUF = $15B ; PUFFER FUER WEITERE BYTES
160 CHKOUT = $FFC9 ; AUSGABE AUF LOGISCHES FILE
170 CLRCH = $FFCC ; AUSGABE AUF DEFAULT
180 PRINT = $FFD2 ; AUSGABE EINES ZEICHENS
190 CLOSE = $FFC3
200 LF = 1 ; LOGISCHE FILENUMMER
210 *= $C000 ; STARTADRESSE
220 LDA LENGHT
230 CMP #$C0 ; SCHLIESSEN
240 BEQ CLOSEF
250 LDX #LF : JSR CHKOUT  ; AUSGABE AUF LOGISCHES FILE 1
260 LDX #0 : LDA LENGHT
270 CMP #$80 ; OEFFNEN
280 BEQ STARTADR
290 OUT LDA OP,X
300 OUT1 JSR WROB ; BYTE ALS HEXZAHL AUSGEBEN
310 CPX LENGHT
320 BEQ EX1
330 INX
340 CPX #3
350 BCC OUT
360 LDA OBJBUF-3,X
370 JMP OUT1
380 EX1 JMP CLRCH
390 CLOSEF LDA #LF
400 JMP CLOSE
410 STARTADR LDA ADR : JSR WROB ; STARTADRESSE LOW
420 LDA ADR+1 : JSR WROB ; STARTADRESSE HIGH
430 JMP CLRCH
440 WROB PHA ; BYTE ALS HEXZAHL AUSGEBEN
450 LSR : LSR : LSR : LSR ; OBERES NIBBLE
460 JSR ASCII
470 PLA
480 AND #%1111 ; UNTERES NIBBLE
490 ASCII CLC
500 ADC #-10
510 BCC ASC1
520 ADC #6
530 ASC1 ADC #"9"+1
540 JMP PRINT
550 .END
```

Wenn Sie dieses Programm assemblieren, so erhalten Sie folgendes Assemblerlisting:

```
PROFI-ASS 64 V2.0    SEITE 1

110:  C000                    .OPT P,OO
120:  004E          LENGHT  =   $4E     ; AUSZUGEBENDE BYTES MINUS 1
130:  004B          OP      =   $4B     ; PUFFER FUER DIE ERSTEN DREI BYTES
140:  0056          ADR     =   $56     ; PROGRAMMSTARTADRESSE
150:  015B          OBJBUF  =   $15B    ; PUFFER FUER WEITERE BYTES
160:  FFC9          CHKOUT  =   $FFC9   ; AUSGABE AUF LOGISCHES FILE
170:  FFCC          CLRCH   =   $FFCC   ; AUSGABE AUF DEFAULT
180:  FFD2          PRINT   =   $FFD2   ; AUSGABE EINES ZEICHENS
190:  FFC3          CLOSE   =   $FFC3
200:  0001          LF      =   1       ; LOGISCHE FILENUMMER
210:  C000                      *=  $C000   ; STARTADRESSE
220:  C000 A5 4E                LDA LENGHT
230:  C002 C9 C0                CMP #$C0    ; SCHLIESSEN
240:  C004 F0 24                BEQ CLOSEF
250:  C006 A2 01                LDX #LF
250:  C008 20 C9 FF             JSR CHKOUT   ; AUSGABE AUF LOGISCHES FILE 1
260:  C00B A2 00                LDX #0
260:  C00D A5 4E                LDA LENGHT
270:  C00F C9 80                CMP #$80    ; OEFFNEN
280:  C011 F0 1C                BEQ STARTADR
290:  C013 B5 4B    OUT         LDA OP,X
300:  C015 20 3C C0 OUT1        JSR WROB    ; BYTE ALS HEXZAHL AUSGEBEN
310:  C018 E4 4E                CPX LENGHT
320:  C01A F0 0B                BEQ EX1
330:  C01C E8                   INX
340:  C01D E0 03                CPX #3
350:  C01F 90 F2                BCC OUT
360:  C021 BD 58 01             LDA OBJBUF-3,X
370:  C024 4C 15 C0             JMP OUT1
380:  C027 4C CC FF EX1         JMP CLRCH
390:  C02A A9 01    CLOSEF      LDA #LF
400:  C02C 4C C3 FF             JMP CLOSE
410:  C02F A5 56    STARTADR    LDA ADR
410:  C031 20 3C C0             JSR WROB    ; STARTADRESSE LOW
420:  C034 A5 57                LDA ADR+1
420:  C036 20 3C C0             JSR WROB    ; STARTADRESSE HIGH
430:  C039 4C CC FF             JMP CLRCH
440:  C03C 48       WROB        PHA         ; BYTE ALS HEXZAHL AUSGEBEN
450:  C03D 4A                   LSR
450:  C03E 4A                   LSR
450:  C03F 4A                   LSR
450:  C040 4A                   LSR         ; OBERES NIBBLE
460:  C041 20 47 C0             JSR ASCII
470:  C044 68                   PLA
480:  C045 29 0F                AND #%1111  ; UNTERES NIBBLE
490:  C047 18       ASCII       CLC
500:  C048 69 F6                ADC #-10
510:  C04A 90 02                BCC ASC1
520:  C04C 69 06                ADC #6
530:  C04E 69 3A    ASC1        ADC #"9"+1
540:  C050 4C D2 FF             JMP PRINT
```

```
]C000-C053
NO ERRORS
```

Wenn Sie dieses Programm assemblieren, können Sie den Objectcode mit dieser Routine im Hexformat auf Kassette schreiben:

```
100 OPEN 1,1,1,"OBJECTCODE" : REM BAND ZUM SCHREIBEN
110 SYS 32768
120 .OPT P,O=$C000 ; OBJECTCODE UBER EIGENE ROUTINE
...
```

Das Laden des Programms vom Band kann dann über ein kleines Ladeprogramm in BASIC geschehen.

```
100 OPEN 1,1,0,"OBJECTCODE" : REM BAND ZUM LESEN
110 GOSUB 1000 : AD = A :REM LOW BYTE STARTADRESSE
120 GOSUB 1000 : REM HIGH BYTE STARTADRESSE
130 AD = A*256 + AD : REM STARTADRESSE
140 IF ST=64 THEN CLOSE1 : END :REM PROGRAMMENDE
150 GOSUB 1000 : REM BYTE LESEN
160 POKE AD,A : AD = AD + 1
170 GOTO 140
1000 REM HEXZAHL LESEN
1010 GET#1, A$,B$
1020 H = ASC(A$)-48+(A$>="A")*7 : REM HIGH NIBBLE
1030 L = ASC(B$)-48+(B$>="A")*7 : REM LOW NIBBLE
1040 A = L+16*H : RETURN
```

Ihre PROFI-MAT-Diskette enthält noch ein BASIC-Programm namens "SYMPRINT". Dieses Programm dient zur Ausgabe einer Symboltabelle in alphabetischer Reihenfolge, die Sie zuvor mit .SST auf Diskette geschrieben haben.

Das Programm fragt nach dem Namen der Symboltabelle auf Diskette sowie der Nummer des Ausgabegeräts (3-Bildschirm, 4-Drucker, 8-Floppy)
Bei Diskettenausgabe müssen Sie den Filenamen angeben, unter dem die sortierte Tabelle abgespeichert werden soll.
Abschließend können Sie noch bestimmen, wieviel Symbole auf einer Zeile gedruckt werden. Auf den Bildschirm passen zwei in Zeile, bei Druckausgabe können Sie vier wählen. Das Ausgabeformat entspricht dem des .SYM-Befehls beim Assemblieren.