C-Compiler

"Pretty C"

Version 1.0 für Kleincomputer

robotron Z9001, KC85/1 u. KC87

Autor: Dr. R. Wobst

(Als Manuskript gedruckt)

VEB Robotron-Vertrieb Berlin

****
****
*** ***
*** Pretty C: Ein C-Compiler für den KC 85/1 ***
*** ***
****
****

Version 1.0

Reinhard Wobst, 22.09.1967

==========

Inhaltsverzeichnis

==========

Bitte zuerst lesen!

Diese Dokumentation setzt die Kenntnis der Sprache C voraus. Literatur hierzu ist im Anhang D angeführt; es ist günstig, wenn Sie auf Kernighan/Ritchie [1] zurückgreifen können. Die Unterschiede zu der dort definierten Sprache sowie Besonderheiten vorliegender Implementierung erfahren Sie im Anhang A.

Bevor Sie mit Pretty C zu arbeiten beginnen, sollten Sie sich mit den Kapiteln 0 und 1 einen Überblick verschaffen.

Der Compiler ist sehr eng mit dem Editor EC verbunden. Es ist notwendig, EC wenigstens grob zu beherrschen, bevor Sie Programme schreiben. Lesen Sie also Kapitel 2 wenigstens flüchtig durch, erst dann Kapitel 3.

Um nun mit der Arbeit zu beginnen, müssen Sie Pretty C installieren. Dieser Vorgang ist in Kapitel 9 beschrieben.

Weiterhin sind die Kapitel 4,5,6 und 8 für die Arbeit mit Pretty C wichtig. Kapitel 7 und der Anhang B betreffen schließlich die effektive Programmierung mit Pretty C.

Kapitel 0: Einführung

0.0 Warum C?

"C ist eine Universalprogrammiersprache." So lautet der erste Satz der Einleitung zum bekanntesten Buch über C, KERNIGHAN/RITCHIE [1]. Und wenig später heißt es: "...obwohl sie schon 'Systemprogrammiersprache' genannt wird, da sie zum Schreiben von Betriebssystemen geeignet ist, kann man in ihr ebenso gut die Mehrzahl numerischer Aufgaben sowie Probleme der Textverarbeitung oder Datenbanken programmieren."
Das sei zum Anfang dieser Einleitung zitiert, denn C wird noch viel zu oft nur als "Systemsprache" betrachtet. Obendrein unterstützt eine Reihe C-Compiler keine Gleitkommaarithmetik.
Schließlich wurde C zwar im Zusammenhang mit dem Betriebssystem UNIX entwickelt, ist aber keinesfalls daran gebunden.

Wieso wird C zunehmend populär?

- C-Compiler sind relativ klein und auf vielen Rechnern verfügbar, vom Mikrorechner bis zum Großcomputer.
- Dabei sind C-Programme erstaunlich gut von einem Rechner auf einen anderen übertragbar.
- Übersetzte C-Programme sind klein und schnell.
- C ist eine relativ kleine, aber moderne Sprache, die einen sehr kompakten und übersichtlichen Programmierstil erlaubt. Sie ist äußerst flexibel und dadurch sehr leistungsfähig.
  Erste Programmiererfolge stellen sich rasch ein.

Natürlich hat C auch Nachteile, und diese sollen nicht verschwiegen werden:

- C zwingt nicht zu gutem Programmierstil. Durch die großen Freiheiten in C kann man trickreiche, unverständliche Programme schreiben, die ebenso unverständliche Fehler oder Abstürze produzieren.
- Insbesondere kann man komplizierte Datentypen auch wirklich schwer lesbar deklarieren (es geht aber auch anders!).
- Bit- und Shiftoperationen haben zu niedrige Priorität, automatische Konvertierungen sind gewissenhaft zu beachten.
- Gleitkommaoperationen werden (bis jetzt) intern stets doppelt genau ausgeführt.

Weniger zu Lasten der Sprache als der Compiler geht, daß Arithmetikfehler von vielen Compilern noch nicht erkannt werden (also z.B. Division durch 0, Integer-Überlauf). Das kann aber für die Sprache Wahl einer Programmiersprache ebenso wichtig sein wie die Sprache selbst.
(Übrigens erkennt Pretty C Arithmetikfehler.)

0.1 Zielstellung für den KC 85/l

Es gab drei Gründe, Pretty C zu entwickeln:

1. Den Lerneffekt: Der Nutzer soll möglichst komfortabel möglichst viel von der Sprache kennen- und nutzen lernen.
2. Die Nutzung der Sprache selbst: Programmentwicklung auch schwierigerer Aufgaben auf diesem Rechner (wobei C natürlich Vorteile gegenüber BASIC und Pascal haben soll!).
3. Die Entwicklung von C-Modulen für größere Rechner in komfortabler Umgebung.

Das erforderte vom Compiler:

- sehr hohe Speichereffektivität (für Compiler, Programme, Arbeitsspeicher);
- Kompatibilität zu anderen Compilern;
- möglichst vollständige Implementierung der Sprache.

Dabei sollen die übersetzten Programme natürlich schnell sein!

Beim KC 85/1 gibt es nun eine einschneidende Randbedingung:
Als externer Speicher steht nur das relativ langsame und umständlich zu handhabende Magnetband zur Verfügung. Praktisch zwingt das zu einem sog. Einpaßcompiler, d.h. der lesbare Quelltext wird sofort in die fertige Maschinensprache übersetzt. Die Möglichkeiten der Optimierung, das Einbinden von Bibliotheken usw. sind damit stark eingeschränkt. Auf der anderen Seite ist Pretty C dadurch sehr bequem zu bedienen.
Bisher genügte wahrscheinlich nur der Hisoft-C-Compiler für den ZX Spectrum der Firma Sinclair dieser Randbedingung. Aus mehreren Gründen kam dieser Compiler für den KC 85/1 aber nicht in Betracht.

0.2 Realisierung im vorliegenden Compiler

0.2.1 Speicherbedarf

Pretty C benötigt 19KBytes Hauptspeicher (300H-ca.4F00H) für Compiler, Editor und Laufzeitsystem sowie Pufferspeicher von 210H bis 2FFH und von 90H bis 190H während der Compilierung.
Mit 32k Speicher können bereits kleinere Programme übersetzt werden.

0.2.2 Realisierte Sprache

Folgende wesentliche Sprachelemente wurden nicht realisiert:

- Bitfelder und der enum-Typ sind nicht implementiert.
- Initialisierungen sind (noch) nicht zulässig, daher z.B. auch kleine Deklarationen der Form
  extern a[];
- Funktionen der Speicherklasse 'static' gibt es nicht.
- Der Präprozessor versteht nur die Kommandos '#define' und '#undef'.

Erweiterungen sind die inline-Anweisung zum Einfügen von Maschinencode in den Quelltext (s.3.3.6) sowie die in 0.2.6 erwähnte getrennte Gleitkommaarithmetik

Detaillierte Informationen findet der Leser im Anhang A.
Die hier nicht genannten Unterschiede sind unbedeutend.

0.2.3 Entwicklung größerer Programme, Variabilität

Der Quelltext kann abschnittsweise übersetzt werden und auf beliebig viele Textpuffer im Speicher und Textfiles auf Band verteilt sein. Letztere

können gleichzeitig mit dem Einlesen ohne Zwischenspeichern übersetzt werden.

Nach den einzelnen Übersetzungsläufen wird der Compiler verlassen. Das kann auch genutzt werden, um weitere Textpuffer in den Speicher vom Band nachzuladen.

Die Kommandos für die einzelnen Läufe können auch in einen File (Textpuffer) geschrieben und dann von einem Kommandoprozessor automatisch abgearbeitet werden.

Diese Möglichkeiten erlauben es, auch größere Programme zu übersetzen, deren Quelltextlänge die Speicherkapazität wesentlich überschreitet (insbesondere braucht nicht bei der übersichtlichen Gliederung und Kommentierung des Quelltextes gespart zu werden!).
Des weiteren erhalten damit "extern"-Variablen ihren Sinn.

0.2.4 Fehlersuche

Ein Compiler ohne Möglichkeiten der Fehlersuche ist ziemlich wertlos, daher wurde diesem Punkt viel Aufmerksamkeit geschenkt.

Folgende Möglichkeiten stehen zur Verfügung:

- Bereits bei der Programmerstellung hilft der keyword-support des Editors (s.2.2.2), Tippfehler zu vermeiden.

- Das vom Compiler erzeugte Listing ist stets strukturiert. Vergessene geschweifte Klammern sind somit leichter zu erkennen.

- Während der Compilierung entdeckte Fehler werden (i.a. sehr gut) lokalisiert im Klartext angezeigt (s.3.2).
  Mit dem "ESC+A"-Kommando des Editors springt man danach sofort zum Fehlerort im Quelltext.

- Laufzeitfehler erscheinen ebenfalls im Klartext, zusammen mit Fehlerort (PC) und Aufrufkette der Funktionen (backtracing). Durch einen speziellen Fehlerlauf des Compilers werden Laufzeitfehler im Quelltext ebenso präzise wie Compilezeitfehler angezeigt. Auch hier kann man sofort zum Editor wechseln. In gleicher Weise verfährt Pretty C auch mit dem Abbruch eines Programmes durch 'STOP'.

- Mit Ausnahme von "printf" und "scanf" (s.3.3.1) wird zur Laufzeit überprüft, ob die Anzahl der Argumente im Funktionsaufruf und in der Funktionsdeklaration übereinstimmen.
  Das vermeidet ansonsten sehr schwer zu findenden Fehler.

- Feldindizes sollten wegen der Art, wie in C Felder aufgefaßt werden, nicht auf Korrektheit zur Laufzeit geprüft werden. Jedoch gibt es eine Compileroption (Schalter /I+), bei der bei jeder Operation "*p" bzw. "p[...]" geprüft wird, ob die berechnete Adresse in einem der 3 Datenbereiche liegt (statische und dynamische Variablen sowie Konstanten).

- Wichtigstes und leistungsfähigstes Werkzeug von Pretty C zur Fehlersuche ist ausschließlich der Debugger, mit dem das übersetzte Programm schritt- oder abschnittsweise - bei gleichzeitiger Anzeige der entsprechenden Anweisungen im Quelltext - abgearbeitet wird. Dabei dürfen auch einzelne Variable abgefragt und verändert werden, wenn auch nur über Adressen, nicht über ihre Namen.
  Der Debugger ist ein (über eine spezielle Schnittstelle im Compiler gerufenes) C-Programm, das vom Anwender beliebig verbessert werden kann.

0.2.5 Ein- und Ausgabe im Dialog

Für CC und EC gelten folgende Regeln:

Ausgabe: Alle Ausgaben können mit PAUSE (Control/S) angehalten und durch Drücken einer beliebigen anderen Taste fortgesetzt werden.
Wird jedoch im Pause-Zustand die Taste STOP betätigt, so bricht das Programm ab.

Eingabe: Bei der Eingabe von Zeichenketten sind die Zeichenkommandos des Editors wirksam (s.2.3.1). CC fordert nur beim Schalter /TP eine Zeichenkette an.

Betreffs Ein- und Ausgabe im Laufzeitsystem GO sei auf 5.2 verwiesen.

0.2.6 Unterstützung numerischer Bedingungen

In Erweiterung der ursprünglich konzipierten Sprache trennt Pretty C float- und double-Arithmetik. Das ist notwendig, da einerseits die Rechenzeit-unterschiede zwischen einfach und doppelt genauer Arithmetik beträchtlich sind, andererseits Anwendungen, die größere Stellenzahlen erfordern, nicht von vornherein ausgeschlossen werden sollen - auch wenn Pretty C dadurch ca. 1KBytes länger wird.

Kapitel 1: Arbeitsweise von Pretty C, allgemeiner Ablauf der Übersetzung

1.0 Allgemeines

Pretty C besteht aus 4 Komponenten:

- einem Compiler (CC)
- einem Editor (EC)
- einem Laufzeitsystem (GO), das das übersetzte Programm startet und Fehlermeldungen ausgibt und
- einem Kommandoprozessor (C@), der mehrere Übersetzungsläufe hintereinander automatisch steuert.

Weiterhin gehören noch ein Installierungsprogramm (s.Kap.9), ein Debugger (s.Kap.8) und Bibliotheksfunktionen (Kap.6) zu Pretty C.

Alle vier Teilprogramme werden direkt vom Monitor aus über Kommandozeilen gestartet, in denen über sogenannte Schalter einzelne Optionen gewählt werden (vgl.1.2.1).
Nach Beendigung der Arbeit kehrt jedes der vier Programme zum Monitor zurück.
Da Pretty C ständig im Speicher bleibt, ist ein schneller Zyklus Editieren-Übersetzen-Testen möglich, ähnlich wie bei Turbo-Pascal.

1.1 Quelltext

Der Quelltext kann als Textpuffer im Speicher stehen oder als File auf Magnetband:

- Textpuffer sind Speicherbereiche, die von CC und EC über willkürliche Namen angesprochen werden. Die Zahl der Textpuffer ist nur durch die Speicherkapazität beschränkt.

- Quelltextfiles auf Magnetband sind vom Editor ausgelagerte Textpuffer. Sie können vom Editor wieder eingelesen werden oder auch vom Compiler direkt beim Einlesen übersetzt werden.

- Der Quelltext eines Programms kann beliebig auf mehrere Puffer und Files verteilt sein und abschnittsweise übersetzt werden.

1.2 Kommandozeilen

1.2.1 Funktion der Kommandozeilen

Über die Kommandozeile wird das entsprechende Teilprogramm (Compiler, Editor, Laufzeitsystem) gestartet. Gleichzeitig kann durch Angabe eines Namens ein bestimmter Textpuffer angesprochen werden (s.Kap.0 und 2) Nach dem Namen (oder an seiner Stelle, falls er fehlt) dürfen sogenannte Schalter folgen. Das sind spezielle Steuerkommandos, über die das Teilprogramm zusätzliche Informationen erhält, wie etwa: Listen des Quelltextes bei Übersetzung ja/nein, nur Liste aller Textpuffer anzeigen, bestimmte Adressen einstellen usw.

Dabei gelten die Grundregeln:

- Fehlende Teile der Kommandozeile werden durch Standardannahmen ergänzt.

- Beim ersten Lauf von CC/EC sind spezielle Standardannahmen gültig. Sie werden in den Beschreibungen ausgegeben.

- Namen und einige Schalter bleiben in allen weiteren Läufen solange gültig, bis sie geändert werden.

- Kommandozeilen werden sequentiell von links nach rechts abgearbeitet. Bei fehlerhaften Zeilen werden also i.a. einige Schalter abgearbeitet.
  Es empfiehlt sich, Kommandozeilen nach solchem Fehler nochmals komplett einzugeben.

Einige Schalter veranlassen Programme nur zu speziellen Funktionen, nach deren Abarbeitung Pretty C abbricht. (Beispiel: Löschen eines Textpuffers).

1.2.2 Syntax

Die allgemeine Form der Kommandozeilen ist (die eckigen Klammern gehören nicht zum Kommando, von ihnen eingeschlossene Teile dürfen fehlen):

```
cmd [name][/s1][/s2]...[;comment]
```

Dabei bedeuten:

cmd — 'CC','EC' oder 'GO', je nach startendem Programm;

name — Ein bis zu 8 Zeichen langer Identifikator entsprechend der C-Sprachdefinition (nur Groß- und Kleinbuchstaben, Ziffern und '_' verwendet, keine Ziffern an erster Stelle).

/sn — Ein Schalter (stets aus '/' und zwei Zeichen bestehend), der für das gemäß 'cmd' gestartete Programm zulässig ist.
Einige Schalter fordern zusätzlich Angabe einer Ziffer:
/LI:1
oder einer Adresse:
/CR:8800

;comment — Unmittelbar nach dem letzten Schalter darf ein Kommentar folgen, der mit ';' beginnen muß (nur sinnvoll bei Anwendung des Kommandoprozessors, s.Kap.4).

Beispiele für Kommandozeilen:

```
CC

EC Alpha/CR:A100

CC /AS:100/BP:2/LI:3;1.Lauf
```

1.2.3 Fehlermeldungen

***** error: illegal parameter**

Eine Zahl bzw. Adresse in einem Schalter hat einen unzulässigen Wert.
Beispiel: CC /LI:4

***** error: illegal switch**

Ein angegebener Schalter ist unzulässig, z.B. nicht für dieses Teilprogramm erklärt.
Beispiele: CC /DE
EC /CR , wenn kein neuer Puffer mehr angelegt werden kann.

***** error: not found**

Es gibt im Speicher keinen Textpuffer mit dem angegebenen oder implizit eingestellten Namen.

1.3 Zum Zyklus Editieren-Übersetzen-Testen

Bei kleineren Programmen kann man ausschließlich auf Standardannahmen zurückgreifen. Dadurch wird die Bedienung sehr einfach, wie das folgende Beispiel zeigt.

Wir haben ein Programm neu einzugeben. Entsprechend 2.4, Bsp.1, brauchen wir nur 'EC/CR' als Kommando einzugeben - der Editor wird gestartet und wählt automatisch Adresse und Namen des Puffers. Der Quelltext wird nun wie im Beispiel eingegeben. Eine ev. nötige Bibliothek wird zusätzlich vom Band gelesen und angehängt. Danach wird EC verlassen.

Die Übersetzung erfolgt durch das Kommando 'CC'. Standardmäßig wird dabei kein Quelltext gelistet, ev. ist das mit dem Schalter '/LI:1' (einmalig) einzustellen.
Im allgemeinen wird CC dabei Fehler bemerken. Der Compiler bricht ab, und wir starten den Editor neu ('EC'). Der Fehler ist vor allem mit dem Adress-Kommando schnell lokalisiert: ESC,A,ENTER drücken.
Wir können nun korrigieren.
Das wiederholen wir bis zur Fehlerfreiheit.

Nun starten wir das übersetzte Programm mit 'GO'. Hier gibt es zwei Arten von Fehlern:

- Eine gerufene Funktion ist nicht definiert (verursacht z.B. durch Tippfehler). Dieser Fehler wird noch vor dem Programmstart erkannt und ist i.a. schnell beseitigt.

- Ein echter Laufzeitfehler tritt auf, wie etwa Division durch 0. Hier gibt es die in 0.2.4 erwähnten Mittel zur Fehlersuche. Im allgemeinen werden wir einen Compiler-Testlauf durchführen und nach dem Lokalisieren des Fehlers erneut editieren.

  Weil die Compilierung sehr schnell abläuft (ca. 1KBytes Quelltext pro Sekunde), läßt sich dieser Testzyklus GO-CC-EC-CC-GO in weniger als 1 Minute durchführen.

Bei mehreren Programmen gleichzeitig lassen wir zweckmäßig die Bibliothek in einem gesonderten Textpuffer. Dann starten wir den Compiler auch mindestens zweimal, wobei ab 2. Übersetzung jedesmal der Schalter '/CN' (continue) zu setzen ist. Sinnvollerweise sollte man sich dabei des Kommandoprozessors bedienen (Kap.4).

Programme mit sehr langem Quelltext sind am schnellsten zu Übersetzen, wenn sie abschnittsweise zwischen den einzelnen Compilierungsgängen vom Editor eingelesen werden - am besten mit Schnelladeroutinen.
Die Übersetzung direkt vom Band ist nur für sehr spezielle Fälle sinnvoll (z.B. bei Speicherplatzproblemen).

KAPITEL 2: Der Editor EC

2.0 Allgemeines

Zum C-Compiler gehört EC, ein relativ einfacher, zeilenorientierter Editor von ca. 2k Länge (wobei viele Hilfsprogramme ebenso wie 'CC' und 'GO' genutzt werden). Der gesamte Text wird dabei im RAM gespeichert, was eine schnelle Reaktion auf Kommandos ermöglicht.
EC kann gleichzeitig mehrere Textpuffer verwalten, auf die über Namen zugegriffen wird (der Nutzer soll sowenig wie möglich gezwungen werden, sich mit Speicheraufteilung und Hexadezimaladressen zu beschäftigen). Die Textpuffer sind gegenseitig vor Überschreiben geschützt.
Weiterhin können Files auf Magnetband geschrieben, eingelesen und dabei im RAM aneinandergehängt werden. Zusammen mit einigen Befehlen zur Blockverarbeitung (definieren, einfügen, löschen) können so Quelltextteile beliebig zusammengefügt werden.

2.1 Kommandozeile

Die allgemeine Form entsprechend 1.2.2 ist

```
EC [name][/S1][/S2]...[;comment]
```

mit mindestens einem Leerzeichen zwischen 'EC' und 'name' bzw. erstem Schalter.

2.1.1 Schalter /CR (CREATE)

Syntax: /CR[:nnnn]
'nnnn' ist dabei eine hexadezimale Adresse.

Wirkung: Es wird ein neuer Textpuffer erzeugt.
Drei Fälle sind zu unterscheiden:

1. Die Adresse ist angegeben: Der Textpuffer wird ab dieser Adresse erzeugt. Sie muß größer als etwa 5200H und kleiner als das generierte RAM-Ende sein.
   Bei unzulässigem Wert erfolgt Fehlermeldung.
   ACHTUNG! Es ist streng darauf zu achten, daß 'nnnn' nicht innerhalb eines anderen Textpuffers liegt!
   Diesen Fall einer expliziten Adresse sollte man nur bei wirklicher Notwendigkeit anwenden.
2. Keine Adresse ist angegeben.

2.1 Im Speicher ist noch kein Textpuffer: Der neue Puffer wird ab der generierten Adresse DFSOT erzeugt (vgl. Kap.9).

2.2 Ansonsten wird der neue Puffer hinter dem letzten im Speicher stehenden Puffer erzeugt:
Endet der letzte Puffer auf Adresse klnmH, so wird der neue Puffer auf (kl+1)00H erzeugt. Dadurch kann der vorherige Puffer nachträglich noch um mind. 100H Bytes erweitert werden.

2.1.2 Schalter /DI (DIRECTORY)

Alle im Speicher befindlichen Textpuffer werden mit Name, Anfangs- und Endadresse gelistet. Danach bricht EC ab.

2.1.3 Schalter /DE (DELETE)

Der Textpuffer mit dem angegebenen (oder Standard-) Namen wird gelöscht, und EC bricht ab.

Sollte ein Textpuffer versehentlich gelöscht worden sein, so braucht man nur auf sein erstes Byte mit einem Monitorprogramm ECH zu schreiben.

2.2 Allgemeiner Ablauf des Editierens

Hier werden nur einige allgemeine Regeln besprochen. Der konkrete Ablauf des Editierens hängt völlig von den gegebenen Kommandos ab; letztere werden unter 2.3 erläutert.

2.2.1 Pufferverwaltung

Nach Eingabe der Kommandozeile prüft EC, ob bereits ein Textpuffer dieses Elements existiert.

Wenn ja: Bei gesetztem /CR-Schalter Fehlermeldung.
sonst: Anzeige der 1. Zeile des Puffers (Editmodus).

Wenn nein: Bei nicht gesetztem /CR-Schalter Fehlermeldung, sonst Einrichten eines neuen Puffers entspr. 2.1.1.

2.2.2 Zum Editieren

Edit- und Insertmode:

Beim Editmode werden die Zeilen aus dem Textpuffer angezeigt und können verändert werden.

Beim Insertmodus werden neue Zeilen eingegeben und in den Puffer eingefügt. Dieser Mode wird durch spezielle Kommandos erreicht und verlassen.
Im Insertmodus sind nur Zeichenkommandos, STOP und ENTER wirksam.
Bei leerem Puffer (insbesondere bei einem neu eingerichteten) springt EC automatisch in den Insertmodus.

Es gibt drei wichtige Hilfen:

1. Auf der Tastatur nicht vorhandene Zeichen (z.B. [,{,~) können aus vorhandenen durch nachfolgendes Drücken von '@' dargestellt werden (s.2.3.1, Sondertasten).

2. Im Insertmode rückt der Cursor automatisch so viele Zeichen ein, wie im Text die letzte Zeile eingerückt wurde.
   Dadurch wird eine gegliederte Programmgestaltung unterstützt.
   Falls noch nichts geschrieben wurde, springt der Cursor durch die Taste '|<-' an den Zeilenanfang.

3. Durch die Taste 'COLOR' wird eine Unterstützung beim Schreiben von C-Schlüsselwörtern gegeben, auch von '#define' und '#undef'. Dieser '**keyword-support**' ist im EDIT- wie im Insertmode wirksam.
   Dazu wird COLOR einmal gedrückt und danach begonnen, das Schlüsselwort zu schreiben. Sobald das Schlüsselwort eindeutig bestimmt ist, wird es von EC ergänzt und akustisch quittiert. Ist dagegen eine Ergänzung nicht möglich, gibt EC nur das akustische Signal uns schaltet den keyword-support ab.

Beispiele:

```
     COLOR drücken:    Akustisches Signal
     'c' drücken:      Es können 'case' oder 'char' oder 'continue' gemeint
                       sein.
     Jetzt 'o':        EC ergänzt sofort zu 'continue'.
                       Der keyword-support wird abgeschaltet.
```

Oder:

```
     COLOR,'d'   :     'do' oder 'default' oder 'double'?
     'o'         :     'do' oder 'double'?
     'u'         :     eindeutig handelt es sich um 'double'; Ergänzung.
```

Bei Eingabe von 'do' ist also der keyword-support sinnlos; beim folgenden Leerzeichen oder '{' wird der Support auf jeden Fall mit Fehlermeldung (akustisch) abgebrochen.

Zeilenanfangszeichen:

Nach dem Start von EC erscheint vor jeder Zeile ein Sonderzeichen, das vom Zustand des Editors und den gegebenen Befehlen abhängt.
Dieses Zeichen ist wichtig, weil eine **Textzeile max. 255 Zeichen lang** sein, d.h. sich über mehrere Bildschirmzeilen erstrecken kann.
Arten von Sonderzeichen:

```
_ (Unterstreichung):   Standardzeichen, erscheint immer außer in den
                       folgenden Fällen
^                  :   beim Rückwärtslisten
~ (Tilde)          :   Ersetzt '_' bzw. '^' beim Löschen der Zeile
*                  :   zeigt den Insertmode an.
$                  :   Zusätzlich zu '_' oder '^', wenn die Zeile im mar-
                       kierten Block liegt (s.u.)
```

2.3 Kommandos

Kommandos sind spezielle Tastendrücke, die sofort wirken (ohne folgendes 'ENTER'). Es gibt Kommandos auf 3 Ebenen:

Blockkommandos: Mehrere Zeilen können zu einem sog. Block zusammengefaßt werden und sind als Ganzes zu löschen oder an anderer Stelle einzufügen.

Zeilenkommandos: Zeilen können eingefügt, ersetzt und gelöscht werden, und der Cursor kann auf verschiedene Arten zeilenweise positioniert werden (Vor- und Rückwärtslisten, Such- und Positionierkommandos).

Zeichenkommandos: Innerhalb einer Zeile können Zeichen überschrieben oder gelöscht, Leerzeichen eingefügt und der Cursor beliebig positioniert werden (zeichen- und wortweise).

Außerdem existieren noch einige Sonderkommandos für Editorsteuerung, Magnetbandarbeit, Listen Druck usw.

Im folgenden wird von jeder Taste der ASCII-Code angegeben und gleichzeitig wird die Möglichkeit, sie mit Hilfe der CONTROL-Taste einzugeben. Dabei bedeuten z.B. "^D", daß gleichzeitig CONTROL und 'D' zu drücken sind, und "ESC+A", daß erst ESC, dann A zu drücken sind.

2.3.1 Zeichenkommandos

Diese Kommandos sind stets wirksam, auch im Insertmodus und bei Stringeingabe in Sonderkommandos (FIND, READ, WRITE, VERIFY).

Cursorpositionierung:

Taste	Code	Wirkung
-->	9(^I)	Cursor rückt um 1 Position nach rechts, Zeichen bleiben unverändert.
<--	8(^H)	Analog nach links; wirkungslos, wenn der Cursor auf dem Zeilenanfang steht.
-->|	24(^X)	Wortweises Springen nach rechts, aber nie über das Zeilenende hinaus und immer mindestens 2 Positionen. Als WORT gilt jede von Leerzeichen begrenzte Gruppe von Zeichen, die selbst kein Leerzeichen enthält.
|<--	25(^Y)	Wortweises Springen nach links.

Sondertasten:

Taste	Code	Wirkung
INS	26(^Z)	An der Cursorposition wird ein Leerzeichen eingeschoben, die Zeichen ab Cursorposition rücken nach rechts.
DEL	31(^^)	Das Zeichen an der Cursorposition wird gelöscht, die Zeichen rechts davon rücken nach links. INS und DEL verschieben das Zeilenende.
@	64	Dient zur Darstellung von Sonderzeichen, die auf der KC 85/1-Tastatur nicht vorhanden sind. Sie werden aus vorhandenen durch nachfolgend eingegebenes '@' entsprechend folgender Umwandlungsreihen gebildet: ([{)] } / | \ - ~ Steht links vom Cursor keines dieser Zeichen, so wird '@' normal ausgegeben.
COLOR	20(^T)	Einschalten des keyword-supports (2.2.2): Schlüsselwörter brauchen nur so weit getippt zu werden, bis sie eindeutig bestimmt sind. Der keyword-support ist vor jedem Schlüsselwort neu einzutippen, kann aber auch noch eingeschaltet werden, wenn bereits ein Teil des Schlüsselwortes geschrieben worden ist. Sowohl Einschalten als auch Ausführen (bzw. Nichtausführen bei Fehler) des Supports werden akustisch angezeigt.

Alle anderen druckbaren Zeichen werden normal ab Cursorposition geschrieben, nur daß dabei Klein- in Großbuchstaben konvertiert werden und umgekehrt: Die KC 85/1-Tastatur arbeitet unter EC also wie eine normale Schreibmaschinentastatur (außer nach ESC+C, s.2.3.4).

2.3.2 Zeilenkommandos

Im Insertmode und bei Stringeingabe wirken von diesen Kommandos nur 'ENTER' und 'STOP'.

Taste	Code	Wirkung
ENTER	13(^M)	Insertmode/Stringeingabe: Abschluß der Zeile Editmodus: Ersetzen der Zeile im Puffer durch die angezeigte. Anzeige der nächsten Zeile. **Achtung!** 1. Nur durch 'ENTER' können Zeilen im Puffer ersetzt werden. 2. Leerzeichen am Zeilenende werden abgehängt, sofern sie rechts vom Cursor stehen. 3. Die Zeile darf maximal 255 Zeichen lang sein, sonst Fehlermeldung.
STOP	3(^C)	Abbruch des Insertmodes, bei Stringeingabe: Abbruch von EC. In allen anderen Fällen wirkungslos.
Cursor hoch	11(^K)	Sprung aus vorhergehender Zeile, falls diese existiert (Rückwärtslisten).
Cursor runter	10(^K)	Sprung zur nächsten Zeile (falls das Pufferende erreicht ist, Meldung). Bei beiden Cursorkommandos bleibt der Puffer unverändert, daher sind sie sehr nützlich als 'undo'-Kommandos: Falls die Zeile versehentlich zerstört, aber noch nicht mit ENTER abgeschlossen wurde, kann z.B. durch 'Cursor runter - Cursor hoch' die Zeile nochmals unverändert übernommen werden.
ESC+T		(d.h. 'ESC' und 'T' nacheinander drücken) TOP: Positionieren des Cursors auf die erste Pufferzeile.
ESC+B		BOTTOM: Positionieren des Cursors auf die letzte Pufferzeile
ESC+F		FIND: Suchen nach einer Zeichenkette ab **folgender** Zeile. Nach 'F' werden ein Leerzeichen ausgegeben und eine Zeichenkette erwartet. Diese ist entsprechend 2.3.1 einzugeben und mit 'ENTER' abzuschließen.
ESC+A		ADDRESS: Die Eingabe einer hexadezimalen Adresse wird erwartet (mit 'ENTER' abzuschließen). Der Cursor wird auf die Zeile positioniert, deren Anfang und Ende diese Adresse einschließen. - Wird an Stelle der Adresse nur 'ENTER' gegeben, so nimmt EC die beim letzten CC-Lauf angezeigte Quelltextadresse eines Fehlers (häufigste Anwendung) - Ist der Adreßwert unsinnig, so wird das Kommando übergangen.
^D	4	DELETE: Löschen der aktuellen Zeile ('_'/'^' wird durch '~' ersetzt) und Anzeige der nächsten bzw. letzten Pufferzeile.
^E	5	EXTEND: Übergang zum Insertmode. Die nun eingegebenen Zeilen werden **nach** der aktuellen eingefügt.

2.3.3 Blockkommandos

Diese Kommandos sind nur im Editmodus wirksam; sie erfordern zweimaliges Drücken der ESC-Taste.

ACHTUNG!

1. Blöcke können **nicht** editiert werden. Dazu ist erst ihre Definition mit dem Forget-Kommando aufzuheben.
2. Nach dem Verlassen des Editors geht eine ev. Blockdefinition verloren.

Tasten	Wirkung
ESC+ESC+M	MARK block: Block markieren; die angezeigte Zeile wird 1. Zeile des Blockes. Danach ist der Cursor auf die letzte Zeile des zu markierenden Blockes zu fahren und erneut 'ESC+ESC+M' zu geben. Ab sofort werden Blockzeilen durch vorgesetztes '$' gekennzeichnet und dürfen nicht mehr editiert werden!
ESC+ESC+T	TOP of block: Springen des Cursors zur 1. Zeile des Blockes.
ESC+ESC+B	BOTTOM of Block: Springen des Cursors zur letzten Zeile des Blockes.
ESC+ESC+I	INSERT block: Der Block wird **nach** der angezeigten Zeile als Kopie eingeschoben. Der Block selbst bleibt unverändert. Danach zeigt EC entweder die ursprünglich folgende oder aber die letzte Pufferzeile an.
ESC+ESC+DEL	DELETE block: Löschen aller Zeilen des Blockes. Wirkung analog zum Zeilenlöschkommando ^D.
ESC+ESC+@	FORGET block: Löschen er Blockdefinition.
ESC+ESC+ESC	Wirkungsloses Kommando, Sprung zurück zum Kommandomode (nützlich, falls irrtümlich die ESC-Taste gedrückt wurde).

2.3.4 Sonderkommandos

Tasten	Wirkung	
ESC+E	END:	Abbruch EC
ESC+D	DISPLAY:	Anzeigen von 4 hexadezimalen Adressen: 1. Textanfang 2. Anfang der angezeigten Zeile 3. Textende 4. Ende des RAM-Bereiches für den Puffer
ESC+L	LIST:	Listen des Textes ab angezeigter Zeile ohne Zeilenanfangszeichen. 'Das Listen kann durch PAUSE' (Code 19, ^S) unterbrochen werden. Abbruch im Haltezustand mit 'STOP', Fortsetzung mit einem beliebigen anderen Zeichen.
ESC+P	PRINT:	Wie LIST, jedoch parallele Ausgabe auf den Drucker.
ESC+u	UPPER:	Der Textanfang wird logisch auf die aktuelle Zeile gelegt, d.h., z.B. bei 'TOP' wird diese Zeile als erste Textzeile angesprungen. Das Kommando wird zum Zerlegen von Files benötigt (s.2.4, Bsp.6). Nach Verlassen von EC wird die Wirkung von 'ESC,u' rückgängig gemacht.
ESC+C	CAPS:	Die Tastatur arbeitet wieder wie üblich beim KC 85/1, d.h., es werden standardmäßig Großbuchstaben geschrieben. Bei Wiederholung dieses Kommandos wird seine Wirkung aufgehoben.

Bandkommandos:

ESC+W WRITE: Ausgabe des gesamten Puffers auf Magnetband. Der Filename (beliebige 1-8 Zeichen) wird als String angefordert, danach ist die Länge der Pausen zwischen den Blöcken als Zeichen einzugeben:
ENTER = '0' = keine Pause = Normalfall;
'1'...'9': 0,1 ... 0,9 sec.
(Das wird für die Übersetzung von Quelltext direkt von Band benötigt).
Wird statt des Filenamens nur 'ENTER' gegeben, so nimmt EC den Puffernamen.
Der Typ der Files kann durch Verwendung von Namen mit 11 Zeichen vergeben werden, vgl. 2.6.

ESC+R READ: Lesen eines Files von Magnetband, Anhängen an das Pufferende.
Der Filename wird als String erfragt und mit dem gefundenen Filenamen auf Band verglichen. Bei Nichtübereinstimmung erfolgt eine Fehlermeldung.
WIRD STATT DES Filenamens nur 'ENTER' gegeben, so wird dieser Vergleich unterdrückt. Der Name des gefundenen Files wird in jedem Fall angezeigt.

ESC+V VERIFY: Prüfen (auf Lesefehler) eines mit EC erstellten Files auf Band. Der Ablauf ist analog zu READ, nur ohne Veränderung des Puffers. Das Fileende erkennt EC selbst.

2.3.5 Zusammenfassung aller Kommandos

Entsprechend den Tastennamen werden die Erklärungen in englisch gegeben.
Wie oben bedeutet z.B. '^D', daß CONTROL-Taste und 'D' gleichzeitig zu drücken sind.

Eine Taste:

Cursor left	1 position left	Cursor down	1 line down
Cursor right	1 position right	Cursor up	1 line up
|<--	1 word left	-->|	1 word right
INS	insert blank	DEL	delete char
@	modify character	COLOR	keyword-support
ENTER	substitute line	STOP	stop insertmode/ stop EC

Zwei Tasten:

^D	delete line	^E	extend (insert line)
ESC+T	top	ESC+B	bottom
ESC+F	find	ESC+A	address
ESC+W	write	ESC+R	read
ESC+V	verify		
ESC+L	list	ESC+P	print
ESC+D	display	ESC+E	end
ESC+u	upper (set top of buffer)	ESC+C	caps

Drei Tasten:

ESC+ESC+M	mark block begin/end	ESC+ESC+@	forget block
ESC+ESC+T	top of block	ESC+ESC+B	bottom of block

ESC+ESC+I insert block ESC+ESC+DEL delete block

ESC+ESC+ESC dummy

2.4 Beispiele

Es empfiehlt sich, die folgenden Beispiele wenigstens teilweise nachzuvollziehen, um mit der Arbeitsweise des Editors vertraut zu werden.

Nicht eingegangen wird auf die Zeichenkommandos, da sie schlecht im Text demonstrierbar sind. Hier gilt die alte Rechnerweisheit: Probieren geht über studieren!

Beispiel 1: Eingabe eines Textes, RAM noch leer

In diesem einfachsten Fall soll die Minimalvariante vorgeführt werden. Wir arbeiten nur mit Standardannahmen, d.h., wir lassen alle entbehrlichen Informationen weg.

Die Kommandozeile ist dann einfach

```
EC /CR
```

EC erzeugt einen Puffer auf der Standardadresse 8000H (die für uns uninteressant ist, solange wir keine Speicherplatzprobleme haben und nur mit einem Puffer arbeiten):

```
EC /CR
filename:CQ
*
```

Das '*' zeigt an, daß EC im Insertmodus ist, denn der Puffer ist leer. Wir schreiben nacheinander den Text - diesmal noch ohne Fehler! - und schließen jede Zeile mit 'ENTER' ab:

```
...
filename:CQ
*1.Zeile          (ENTER)
*2.Zeile          (ENTER)
*letzte Zeile     (ENTER)
*
```

Der Text ist eingegeben. Wir wollen EC verlassen und müßten dazu 'ESC,D' geben, d.h. ein Subkommando. Das ist aber im Insertmode nicht möglich, Wir verlassen daher den Insertmode mit 'STOP'. Normalerweise würde EC jetzt die auf die letzte eingegebene Zeile folgende Zeile anzeigen. Das geht aber nicht, denn wir sind bereits am Pufferende. Daher erscheint die Meldung

```
+ end of text +
```

```
_letzte Zeile
```

- EC zeigt die letzte Pufferzeile an.
Nun drücken wir 'ESC' = Ankündigung eines Sonderkommandos, und der Cursor wechselt mit "piep" auf die neue Zeile.
Durch Drücken von 'E' können wir endlich EC verlassen:

```
...
_letzte Zeile
E
>
```

Mit '>' hat sich bereits wieder das Betriebssystem gemeldet.

Anmerkung: Natürlich wird man sich in der Regel bei der Eingabe vertippen. Werden diese Fehler noch in der gleichen Zeile bemerkt, so können sie mit den Zeichenkommandos (s.2.3.1) sofort korrigiert werden, ansonsten im Editmode.

BEISPIEL 2: Wiederfinden des Puffers, Retten auf Band

Ist inzwischen mit keinem anderen Puffer gearbeitet worden, brauchen wir wieder nur

```
EC
```

einzugeben. Da ein Puffer mit dem zuletzt (implizit) verwendeten Namen CQ existiert und nicht leer ist, springt EC in den Editmode:

```
EC

filename:CQ
_1.Zeile
```

Wir können nun den Text editieren.
Zum Retten auf Band drücken wir 'ESC' und danach 'W' (Rechnerausschriften unterstrichen):

```
1.Zeile
W filename:Bsp pause:
```

Im Normalfall wird 'ENTER' gedrückt (keine Pause zwischen den Blöcken auf Band), und EC fordert auf:

```
start tape
```

Das ist die letzte Gelegenheit, mit STOP abzubrechen (ansonsten auch bei Eingabe von Filename und Pausenlänge).
Nach Drücken von 'ENTER' beginnt das Schreiben auf Band. Jeder geschriebene Block wird mit '.' auf dem Bildschirm angezeigt. Danach springt EC wieder in den Editmode zur zuletzt angezeigten Zeile. In unserem Fall ergibt sich etwa folgendes Bild:

```
_1.Zeile
W filename:Bsp pause:

start tape

..
_1.Zeile
```

BEISPIEL 3: Einlesen eines Files in einen neuen Puffer

Dieser Anwendungsfall ist ebenfalls sehr häufig. Wir erzeugen einen neuen Puffer, in den der File aus Beispiel 2 gelesen werden soll. Damit wird der Puffer aus Beispiel 1 auf dem Umweg über Magnetband dupliziert.

Zunächst richten wir einen neuen Puffer ein:

```
EC /CR

*** error: old buffer!
>
```

Das war ein Fehler, denn als Name wurde CQ angenommen, aber ein solcher Puffer existiert bereits. Wir starten also neu:

```
EC Copy/CR
filename:Copy
*
```

EC ist wie üblich im Insertmode. Damit wir das Sonderkommando 'Read' ('ESC','R') geben können, müssen wir aber im Editmode sein. Wir haben also mindestens 1 Zeile einzugeben, damit der Puffer nicht leer ist und wir den Insertmodus verlassen dürfen. Wir drücken 'ENTER' (Eingabe einer Leerzeile), 'STOP','ESC','R' und sehen auf dem Bildschirm:

```
filename:Copy
*
*

_
R filename:
```

Wir verzichten auf den Vergleich der Filenamen und drücken zweimal 'ENTER' (Bandgerät einschalten!):

```
...
R filename:
```

start tape

Bsp found

_

Die Datei ist eingelesen, wir befinden uns immer noch in der 1.Zeile. Wäre ein Lesefehler aufgetreten (z.B. 'bad record'), so hätten wir nach kurzem Zurückspulen irgendeine Taste drücken müssen ('ENTER' ist dabei unsichtbar).
Die erste "Hilfszeile" kann nun gelöscht werden, wenn sie stört: CONTROL/D drücken -

```
~     (Löschen der Zeile)
_1.Zeile
```

- und die erste Zeile des Files wird angezeigt.

BEISPIEL 4: Anhängen eines Files, Listen

Im Anschluß an Beispiel 3 wollen wir eine 2.File anhängen. Wir geben wieder 'ESC','R':

```
_1.Zeile
R filename:last
```

start tape

*** not found (falsche Bandstelle!)

Bsp found

filename:last

start tape

last found

_1.Zeile

Den so entstandenen Puffer können wir uns ansehen -
'ESC','L' (Listen ab angezeigter Zeile):

```
_1.Zeile
L

1.Zeile
2.Zeile
letzte Zeile
allerletzte Zeile
```

+ end of text +

_allerletzte Zeile

Der neue File bestand nur aus einer Zeile.

BEISPIEL 5: Löschen von Puffern und Pufferinhalten, Blockbefehle

Wir kontrollieren zunächst einmal die vorhandenen Puffer:

```
EC /DI

8000 8029 CQ
8200 823B Copy
```

Um 'Copy' zu löschen, brauchen wir nur

```
EC Copy/DE
```

oder, falls zuletzt mit 'Copy' gearbeitet wurde (**Vorsicht!**)

```
EC /DE
```

zu geben.
Manchmal soll aber auch der Puffer erhalten bleiben und nur neu gefüllt werden:
Wir stellen zunächst die erste Zeile ein - entweder automatisch durch Neustart von EC:

```
EC Copy

filename:Copy
_1.Zeile
```

- oder, falls wir schon Copy bearbeiten, durch das Top-Kommando:

```
-allerletzte Zeile      ('ESC','T')
T
_1.Zeile
```

Dann markieren wir den Blockanfang:

```
_1.Zeile                ('ESC','ESC','M')

M
_1.Zeile
```

Nun fahren wir zur letzten Zeile und markieren diese als Blockende:

```
_1.Zeile                ('ESC','B')
B
_allerletzte Zeile      ('ESC','ESC','M')

M
$_allerletzte Zeile
```

Ab sofort gehören alle Zeilen des Puffers zum Block, daher das vorgesetzte '$'.
Wenn wir den Block löschen, wird der Puffer leer:

```
$_allerletzte Zeile     ('ESC','ESC','DEL')

+ end of text +

*
```

Der Editor versucht nach Löschen des Blockes die dem Block folgende Zeile anzuzeigen; das ist aber das Pufferende (daher die Meldung). Beim Versuch, die letzte **Puffer**zeile anzuzeigen, merkt er, daß der Puffer leer ist und geht in den Insertmode.

BEISPIEL 6: Zerlegen eines Files

Soll z.B. ein Textfile auf Band in zwei Files zerlegt werden, so lesen wir dieses File zunächst ein (Beispiel 3). Danach springen wir die erste Zeile des zukünftigen 2.Teils an:

```
EC SPLIT/CR
filename:SPLIT
*                       ('ENTER')
*                       ('STOP')

+ end of text +

_                       ('ESC','R')
R filename:             ('ENTER')

start tape              ('ENTER')

big found

_                       (^D)
~
_1.Zeile                ('ESC','F')
F letz
_letzte Zeile
```

Nun verlagern wir den Textanfang auf diese Zeile:

```
...
_letzte Zeile           ('ESC','u')
u
_letzte Zeile
```

Von der Wirkung des upper-Kommandos können wir uns z.B. durch 'ESC','T' oder 'Cursor hoch' überzeugen (keine Reaktion).

Der Puffer enthält scheinbar nur noch den 2. Textteil, und wir lagern ihn wie üblich auf Band aus:

```
_letzte Zeile           ('ESC','W')
W filename:2 pause;     (...)

start tape

..

_letzte Zeile           ('ESC,'E')
E>
```

Wir rufen EC erneut:

```
EC
filename:SPLIT
_1.Zeile
```

Wie ersichtlich, ist das Kommando 'ESC','u' "vergessen".
Analog zu Beispiel 5 löschen wir den 2. Textteil mit Blockkommandos (im vorliegenden Fall wäre es natürlich viel einfacher, die zwei letzten Zeilen mit '^D' zu löschen, aber in realen Fällen wird der 2.Teil ja wesentlich größer sein):

```
_1.Zeile                ('ESC','F')
F letz
_letzte Zeile           ('ESC','ESC','M')

M
_letzte Zeile           ('ESC','B')
B
_allerletzte Zeile      ('ESC','ESC','M')
```

```
M
$_allerletzte Zeile     ('ESC','ESC','DEL')

+ end of text +

_2.Zeile
```

Die letzte Pufferzeile wird angezeigt.
Dieser 1.Teil kann nun analog ausgelagert oder noch weiter zerlegt werden.

2.5 Meldungen und Fehlerdiagnosen

2.5.1 Schwere Fehler, die den Abbruch von EC bewirken

***** error: illegal command**

Die Kommandozeile ist fehlerhaft, meist wurde ein unerlaubter Schalter angegeben, bzw. bei /CR kann kein neuer Puffer mehr eingerichtet werden.

Abhilfe: Neueingabe der Kommandozeile (bzw. Puffer auslagern oder löschen).

***** error: illegal parameter**

Die Adresse beim '/CR'-Schalter ist unzulässig. Entweder ist sie nicht durch 100H teilbar oder aber zu klein (kleiner als ca 5200H) bzw. zu groß (größer als das in RAMTP eingestellte Speicherende, s.Kap.9).

Abhilfe: Neueingabe mit anderen Adressen.

***** error: old buffer!**

Der '/CR'-Schalter wurde verwendet, aber ein Puffer dieses Namens existiert bereits.

Abhilfe: Neueingabe mit anderem Puffernamen (oder gleichnamigen Puffer löschen).

***** error: textbuffer full**

Die obere Schranke des RAM-Bereiches für den Textpuffer wurde erreicht.
Dieser Fehler tritt in zwei Fällen auf.

1. Beim Abschließen einer Zeile mit 'ENTER'. Diese Zeile geht verloren, der restliche Puffer bleibt erhalten.
2. Beim Einlesen von Band. Der letzte gelesene Block geht verloren.

Abhilfe:
- EG erneut rufen, Puffer auf Band schreiben und auf andere Adresse legen (auch Verschieben mit einfachem Dienstprogramm möglich);
- Bei Installierung (Kap.9) RAMTP verändern, wenn möglich;
- falls im RAM-Bereich hinter dem Puffer ein weitere Puffer folgt (und damit den RAM-Bereich für den ersten Puffer begrenzt), kann auch dieser Puffer gelöscht oder verschoben werden.

***** error: statement too long**

Eine Zeile mit mehr als 255 Zeichen sollte abgespeichert werden. Die Zeile geht verloren.

2.5.2 Meldungen mit Fortsetzung von EC

***** not found**

- Ein String wurde beim Find-Kommando nicht gefunden.
- Beim Read—Kommando hat der gelesene File einen anderen Namen als angegeben.
- Ein nicht existierender Puffer sollte editiert oder gelöscht werden (in diesem Fall bricht EC natürlich ab).

no block defined

Ein Blockkommando - außer "mark" - wurde gegeben, obwohl kein Block definiert ist.

block , protected

Eine Zeile aus einem Block sollte verändert oder gelöscht werden. Diese Fehlermeldung erscheint erst bei Eingabe von 'ENTER' oder '^D'.

Abhilfe: Forget-Kommando geben, editieren, Block neu definieren.

+ end of text +

Häufigste Meldung; das Pufferende wurde erreicht. Danach wird stets die letzte Pufferzeile angezeigt. Ist der Puffer leer geworden, geht EC sofort nach der Meldung in den Insertmode.
Vgl. auch die Beispiele in 2.4.

'bad record' und 'record not found'

Meldung des Betriebssystems (KC 85/1) beim Bandlesen; Abbruch mit 'STOP', Fortsetzung mit beliebigem anderen Zeichen (wobei 'ENTER' nicht angezeigt wird!).

2.6 Ergänzungen

Blockdefinitionen

Obwohl Zeilen eines Blockes nicht editiert oder gar gelöscht werden dürfen, kann der Text vor und nach dem definierten Block selbstverständlich editiert werden. Die Adressen der Blockzeilen werden dabei nötigenfalls intern verschoben.
Wird nach der letzten Zeile vor einem Block Text eingeschoben (Insertmode), dann erscheint "$*" als Zeilenanfangszeichen, so als ob die Zeile in den Block eingefügt werden sollte. Zu diesem Zeitpunkt sind nämlich die Blockadressen noch nicht verschoben, daher die scheinbare Markierung als Blockzeile.
Beim Einfügen der Zeile wird jedoch ordnungsgemäß getestet, und das Erscheinen von "$" ist lediglich als Schönheitsfehler aufzufassen.
Allerdings sollte man einen Block ohnehin erst definieren, wenn man ihn einfügen oder löschen will.

Aufbau der Textpuffer

Jeder Textpuffer beginnt auf einer durch 100H teilbaren Adresse; diese muß größer als ca. 5200H und kleiner als RAMTP (die installierte größte RAM-Adresse) sein.
Zu Beginn jedes Puffers steht die charakteristische Bytefolge

EC ED EE 'name'

Der Name ist ggf. mit Nullen auf 8 Zeichen aufgefüllt.
Diese Bytefolge dient EC zur Erkennung und Markierung von Puffern.
Dann folgen '0D' und unmittelbar darauf der Text im ASCII-Code. Jede Textzeile endet mit 'ENTER' (0D). Eine Zeile, die mit ETX (03) beginnt, wird als Pufferende interpretiert.

Aufbau der Files auf Magnetband

Der Puffer wird in Blöcken zu 128 Bytes aufgezeichnet.
Die Blöcke werden mit 0 beginnend durchnumeriert.
Block 0 ist Steuerblock und enthält außer dem Filenamen (erste 8 Zeichen) keine relevanten Informationen.
Block 1 beginnt mit der ersten Textzeile.
Ein Block, der ETX (03) enthält, wird als letzter Block erkannt.
Danach folgt ein bedeutungsloser Block mit der Nummer FFH.

Files dürfen auch länger als 32k sein, d.h. einen Block mit der Nummer FFH als nicht -letzten Block enthalten!

Verwendung langer Filenamen

Beim Write-Kommando werden auch Strings mit mehr als 8 Zeichen akzeptiert. Beispielsweise können 9.-11.Zeichen als Filetyp verwendet werden (ev. zwecks Kompatibilität mit anderen Programmen).
Jedoch werden beim Read- und beim Verify-Kommando höchstens 8 Zeichen verglichen und beim Lesen angezeigt.

Bildschirmspeicherung

EC arbeitet unmittelbar mit dem Bildwiederholspeicher. Bei 'ENTER' im Editmode wird die veränderte Zeile direkt dem Bildspeicher entnommen.
Die Zeilenzahl ist in der Grundversion mit 24 Zeilen festgelegt. und kann mit dem Install-Programm verändert werden (Kap.9).

Systemuhr beim KC 85/1

Die Systemuhr läuft bei EC schneller.
Dadurch wird auch die Tastenwiederholfunktion schneller

Dieser Effekt wird durch Ersetzen der Tastatur-Interruptroutine durch eine eigene erreicht (Ändern der Interruptsäule in 208H/209H). Diese Routine hat die gleichen Effekte wie die Systemroutine, nur daß der CTC-Kanal 2 (Port 82H) mit der Zeitkonstante 60H statt 96H geladen wird.

Kapitel 3: Der Compiler CC

3.0 Kommandozeile

Der Compiler wird mit der Kommandozeile

```
CC [name][/s1][/s2]...[;comment]
```

gestartet. Betreffs Syntax und Bedeutung der Elemente sei auf 1.2 verwiesen.

3.0.1 Schalter /LI (LIST)

Format: /LI:n

Syntax: n=0,1,2,3

Standard: /LI:0 beim ersten Lauf;
Schalter bleibt bis zur Änderung für alle weiteren Läufe gültig.

Wirkung: Der Schalter steuert die Bildschirmausgabe während der Übersetzung.

n=0 Nur ev. Fehlermeldungen (zusammen mit der entsprechenden Quelltextzeile) erscheinen.

n=1 Der Quelltext wird beim Übersetzen gelistet.
Das Listing ist stets strukturiert: Bei Verschachtelungstiefe n wird jede Zeile um 4n oder 2n Positionen (je nach Installierung) eingerückt. Für das globale Niveau gilt n=0. Es zählt das Niveau zu Zeilenbeginn.
Bei Fehlermeldungen wird die beanstandete Zeile jedoch in ursprünglicher Form ausgegeben.

n=2,3 Die Adressen der definierten Variablen, Funktionen und Strukturkomponenten werden angezeigt, bei n=3 auch der Quelltext wie bei n=1.
Diese Option ist für die Nutzung des Debuggers wichtig. Dabei erscheinen diese Adressen:

- zu Beginn jedes Blockes, nachdem der ev. Deklarationsteil gelesen wurde. Das gilt auch für die Funktionskörper;

- am Ende jedes Files (Textpuffer oder Magnetbandfile) für alle globalen Variablen sowie Funktionen, die in diesem File definiert wurden.

Strukturen erscheinen unter ihrem Namen, dem **keine** Adresse folgt (bei namenlosen Strukturen erscheint nur eine Leerzeile). Danach wird die Liste der Komponenten mit den Offsets ausgegeben.

Zu beachten ist:

- Adressen mit vorgesetztem '$' sind Offsets (Relativadressen von dynamischen Variablen), sonst absolute Adressen.
- Die Adressen von Strukturkomponenten sind stets Offsets, unabhängig von ev. vorgesetztem '$'.
- Adressen von Funktionen haben keine Bedeutung.

Beispiel:

```
*** Pretty C *** R.Wobst 1987
V1.0

int im,jm,kg
test(a,b)

      int a;
      float b;
--------------------
** var **
b $0000
a $FFFC
--------------------
** struct **
--------------------
      {static int i,j,k;
       struct s1 {int si,sj,sk;};
       union {int ui,uj,uk;} u;
--------------------
** var **
u $FFFA
k 9FF5
j 9FF7
i 9FF9
--------------------
** struct **

uk $0000
uj $0000
ui $0000
--------------------
s1
sk $FFFC
sj $FFFE
si $0000
--------------------
       return;
      }
*** no errors found ***
--------------------
** var **
kg 9FFB
jm 9FFD
im 9FFF
test 432E
--------------------
** struct **
--------------------
```

3.0.2 Schalter /CN (CONTINUE)

Format: /CN

Syntax: entfällt

Standard: Schalter gilt nur einen Lauf.

Wirkung: Ein vorher abgeschlossener Übersetzungslauf wird fortgesetzt. Z.B. wirken die Kommandos

```
CC name1
CC name2/CN
CC name3/CN
```

so, als wären die drei Textpuffer aneinandergehängt. Dieser Schalter ist bei verstreutem Quelltext wichtig (s.0.2.3). Zusätzlich können bei jedem Lauf andere Optionen gewählt werden (z.B. beim Einbinden des Debuggers).

3.0.3 Schalter /TP (TAPE)

Format: /TP

Syntax: entfällt

Standard: Schalter gilt nur einen Lauf

Wirkung: Der Quelltext liegt als File auf Band vor und wird während des Einlesens sofort übersetzt. Der Filename wird analog wie beim Editorkommando 'read' erfragt, vgl 2.3.4 und 2.4, Bsp.3.
Ein ev. Puffername wird überlesen, denn der Text wird nicht gespeichert.

3.0.4 Schalter /PT (PRINT)

Format: /PT

Syntax: entfällt

Standard: Schalter gilt nur einen Lauf

Wirkung: Parallel zur Bildschirmausgabe wird während der Übersetzung gedruckt.
Ein Druckertreiber muß initialisiert sein.

3.0.5 Schalter /DS (DISPLAY)

Format: /DS

Syntax: entfällt

Standard: Nach der Abarbeitung dieser Funktion bricht CC ab, der Schalter wird zurückgesetzt.

Wirkung: Anzeige aktueller Parameter:
- der aktuelle Filename;
- gesetzte Schalter, die länger als einen Lauf wirken (LI, BP, I±, F±, AS, WE);
- noch nicht definierte, aber bereits deklarierte Funktionen;
- noch nicht definierte externe Variable;
- am Fileende noch gültige Macros;
- belegter Speicherplatz (von-bis) für:
  - Debug-Adressen,
  - Konstanten,
  - übersetztes Programm,
  - statische Variable,
  - Funktionsnamen.

Noch nicht definierte externe Namen (Funktionen und extern-Variable) und Macros werden nur angezeigt, wenn eine Compilierung erfolgreich abgeschlossen und noch kein GO gegeben wurde.

3.0.6 Schalter /BP (BREAKPOINT)

Format: /BP:n

Syntax: N=0,1,2

Standard: n=1; Schalter gilt für alle folgenden Läufe bis zur Änderung.

Wirkung: Setzen von Unterbrechungspunkten im übersetzten Programm:

n=0 Der generierte Code enthält keine Breakpoints. Das übersetzte Programm wird so am schnellsten, kann aber nicht mehr mit STOP abgebrochen werden (vgl. 5.2).

n=1 Am Ende jeder Schleife steht ein Breakpoint, an dem das Programm zur Laufzeit abgebrochen werden kann (s.5.2).

n=2 Nach jedem Ausdruck wird ein Breakpoint gesetzt. Diese Option darf nur zum Einbinden des Debuggers verwendet werden, vgl. 8.1.

3.0.7 Schalter /WE (WORKSPACE END)

Format: /WE:nFF oder /WE:0

Syntax: nFF darf maximal gleich der generierten Endadresse des Speichers (RAMTP) sein und muß wenigstens 300H größer als das Compilerende sein.

Standard: nFF = (Adresse des niedrigstgelegenen Textpuffers)-1. Ist noch kein Textpuffer vorhanden, so wird nFF gleich RAMTP gesetzt.
Nach Setzen des Schalters gilt er für alle folgenden Läufe. Durch /WE:0 wird wieder die Standardannahme eingestellt.

Wirkung: Der Arbeitsbereich für den Compiler (Listen und Stack) und für das übersetzte Programm (Daten) werden durch nFF nach oben begrenzt.
Details vgl. B.2.

3.0.8 Schalter /CS (CONSTANT SPACE)

Format: /CS:n00

Syntax: n muß eine Hexadezimalzahl größer 0 sein.

Standard: n=4 (entspricht 1KByte); der Schalter gilt für alle folgenden Läufe bis zur Änderung.

Wirkung: Für Funktionennamen und Konstanten werden n00H Bytes reserviert.
Details vgl. B.2.

3.0.9 Schalter /AS (DEBUG SPACE)

Format: /AS:n00

Syntax: n muß eine Hexadezimalzahl zwischen 0 und 40H sein.

Standard: n=0; der Schalterwert überträgt sich auf alle mit /CN angehängten Compilerläufe.

Wirkung: Es werden n00H Bytes für Debug-Adressen reserviert. Dieser Speicherplatz wird nur bei der Option /BP:2 beschrieben, d.h., wenn der Debugger eingebunden werden soll.

3.0.10 Schalter /I+ (ADDRESS CHECK ON)

Format: /I+

Syntax: entfällt

Standard: /I- (s.u.); der Schalter wirkt auf alle folgenden Compilerläufe bis zur Änderung.

Wirkung: Zur Laufzeit wird bei jeder Dereferenz-Operation (d.h., "*p" und "p[index]", wobei p ein Pointer oder ein Feld ist) geprüft, ob die berechnete Adresse in einem der drei Datenbereiche liegt:
- Bereich für dynamische Variable zum aktuellen Zeitpunkt,
- Bereich für statische Variable,
- Konstanten.

Ist das nicht der Fall, so erfolgt die Fehlermeldung "pointer out of address space".
Das übersetzte Programm wird dadurch langsamer und etwas länger.

3.0.11 Schalter /I- (ADDRESS CHECK OFF)

Format: /I-

Syntax: entfällt

Standard: /I- beim ersten Lauf; für alle folgenden Compilerläufe gilt der zuletzt eingestellte Wert.

Wirkung: Die Überprüfung der berechneten Adressen bei "*"- und Index-operationen (s.3.0.10) wird nicht durchgeführt, das übersetzte Programm dadurch schneller.

3.0.12 Schalter /EL (ERROR LOCATE)

Format: /EL[:adr]

Syntax: adr muß eine hexadezimale Adresse (ein- bis vierstellig, Groß-buchstaben) sein.

Standard: adr = error-PC zur Laufzeit (s.u.);
Schalter wirkt nur einen Lauf, ebenso der Standard.

Wirkung: Zuordnung von aufgetretenen Laufzeitfehlern zum Quelltext. Zwei Formen des Kommandos sind möglich:

/EL Unmittelbar zuvor ist ein Laufzeitfehler aufgetreten. Die Übersetzung muß mit den gleichen Schaltern (bis auf /EL) wie zuvor erfolgen! Der Compiler übersetzt (ev. in mehreren Läufen) bis zu der Stelle, an der der Laufzeitfehler auftrat und zeigt auf die entsprechende Anweisung mit der Meldung "here! ".

/EL: adr Analog zur Version /EL, aber diesmal braucht der Laufzeitfehler nicht unmittelbar vorher aufgetreten zu sein. Als 'adr' kann der error-PC der backtrace-Meldung (s.5.1) genommen werden, aber ebenso jeder andere PC aus der Aufrufkette.

ACHTUNG! Das Programm wird bei /EL nicht fertig übersetzt!

3.0.13 Schalter /F+ (FLOAT ON)

Format: /F+

Syntax: entfällt

Standard: /F- (s.u.); der Schalter wirkt auf alle folgenden Compilerläufe bis zur Änderung.

Wirkung: Gleitkommakonstanten ohne Exponentialteil erhalten den Typ 'float'.

3.0.14 Schalter /F- (FLOAT OFF)

Format: /F-

Syntax: entfällt

Standard: Schalter gesetzt; der Schalter wirkt auf alle folgenden Compilerläufe bis zur Änderung.

Wirkung: Gleitkommakonstanten ohne Exponentialteil erhalten den Typ 'double'.

3.1 Fehlermeldungen während der Compilierung

3.1.1 Fehlerbehandlung, Aufbau der Meldungen

Pretty C bricht nach Erkennen eines Fehlers ab. Das ist zwar die primitivste Art der Fehlerbehandlung, aber zwei Gründe sprechen dafür:

1. Sinnvolle Fortsetzung der Übersetzung ist oft ein schwieriges Unterfangen, wenn nicht viele unsinnige Folgefehler entstehen sollen.
   Z.B. stellt gerade ein fehlendes Semikolon jeden Compiler vor Probleme (s.u.).
   Gute Fehlerbehandlung macht den Compiler groß und kompliziert, was nicht den Forderungen in 0.1 entspricht.

2. Fehlerkorrekturen sind schnell und bequem durchzuführen, daher stört das Vorgehen in kleinen Schritten nicht so sehr.
   Insbesondere wird der Quelltext fast immer in einem Textpuffer stehen, da Syntaxfehler zunächst in unabhängigen Moduln beseitigt werden.

Aufbau der Fehlermeldungen

Das sei an einem Beispiel erläutert.

```
CC /LI:0

      if (n==3) {m=k; k=1; l=n;}
-----------------*

"transform"821B

>>m<<
*** error: name not declared
```

Bedeutung: Die fehlerhafte Zeile wird immer angezeigt, auch bei '/LI:0'. Der '*' in der folgenden Zeile zeigt auf den gerade analysierten Teil der Zeile, hier den Namen 'm' (man beachte, daß Quelltextzeilen auch über mehrere Bildschirmzeilen gehen können - der Stern muß sich also nicht immer auf die vorangegangene Bildschirmzeile beziehen!).

"transform" ist der Name der Funktion, in der der Fehler auftrat, '821B' die Fehleradresse im Quelltext (auf die '*' zeigt), und '>>m<<' besagt, daß der Name 'm' nicht deklariert wurde. Die Quelltextadresse ist nützlich, um den Fehler mit dem Editor schnell zu lokalisieren - vgl. Address-Kommando 'ESC+A' von EC.

Bei Fehlern außerhalb von Funktionen steht als Funktionsname '(global)'.

An dieser Stelle sei gleich noch auf zwei kritische Fehlerarten hingewiesen:

```
CC /LI:1

...
            n+m*m; ++m
            if(n>k) break; else if(n&k) pull_down();
-----------*

"main"8041

*** error: missing operator
```

Diese Meldung erscheint unverständlich, besonders wenn nur die "fehlerhafte" Zeile angezeigt wird. In Wirklichkeit fehlt hinter '++m' das Semikolon, und die if-Anweisung wird von CC zunächst als Fortsetzung der Anweisung '++m' betrachtet.

Noch schwieriger liegt der Fall bei

```
CC /LI:1

...

main() {int n; while(scan(&n)) {print(n);}

print(m) int m; {...
-------*

"main"803C
>>m<<
*** error: name not declared
```

Hier fehlt die geschweifte Klammer '}', die 'main' abschließt. Darauf deutet auch der Funktionsname 'main' in der Meldung hin. 'print(m)' wird als Funktionsruf interpretiert, und als Argument erscheint die nicht deklarierte Variable m.
Eine wichtige Hilfe bei der Suche solcher Fehler ist das strukturierte Listing, das CC stets erzeugt (mit Ausnahme der fehlerhaften Zeile, s.3.0.1).

Übrigens: Würde CC fortsetzen, kämen in diesem Fall wahrscheinlich noch die Fehlermeldungen 'reserved name' und 'missing operator'; das Übergehen dieser "Anweisung" aber würde noch mehr Fehlermeldungen hervorbringen.
Wie sehen, daß ein Abbruch mit dem ersten Fehler hier durchaus sinnvoll ist.

3.1.2 Liste der Fehlermeldungen

Die Fehler werden alphabetisch geordnet aufgeführt.

***** error: conflicting declarations**

Eine Funktion bzw. externe Variable ist an verschiedenen Stellen von verschiedenem Typ.

***** error: conflicting type**

Unzulässige Typkombination, wie z.B. in 'pointer+pointer', oder verschiedenen Typen beim '?:'-Operator (vgl.Anh.A).
Abhilfe: Im letzten Fall Castoperatoren verwenden.

***** error: constant space overflow**

Der reservierte Speicherplatz ist zu klein, besonders bei Verwendung langer Strings.
Abhilfe: Vergrößerung mit dem Schalter '/CS'.

***** error: data space overflow**

Der reservierte Speicherplatz reicht für die Übersetzung bzw. spätere Abarbeitung des Programms nicht aus.
Abhilfe: Verringerung der Anzahl der Textpuffer, Schalter /WE und /CS; mehr Hinweise in 3.6.

***** error: debug space overflow**

Der Schalter /BP:2 ist gesetzt, aber der mit /AS:n00 reservierte Speicherplatz reicht nicht aus.
Abhilfe: Mehr Platz reservieren, oder kleinere Programmabschnitte debuggen.

***** error: declaration illegal**

Die Deklaration ist zwar syntaktisch richtig, aber logisch falsch, z.B. wenn Funktionsargumente als "static" erklärt werden.

***** declaration syntax**

In einer Deklaration ist ein Syntaxfehler entdeckt worden. Beispiele: long char c; int f(a);

***** error: end of text**

Das Textende wurde während einer globalen Deklaration erreicht.

***** error: foating overflow**

Eine Gleitkommakonstante überschreitet die zulässigen Werte, oder bei Berechnung eines konstanten Ausdrucks trat Gleitkomma-Überlauf auf.

***** error: floating zero division**

Division durch Null in einem konstanten Ausdruck mit Gleitkommazahlen.

***** error: here!**

Dieser "Fehler" erscheint beim Testlauf mit '/EL' (s.3.1). Er markiert die Anweisung, in der der Laufzeitfehler auftrat.

***** error: illegal character**

Die Programmzeile enthält ein unzulässiges Zeichen (Steuerzeichen, '$','#'...).

***** error: illegal constant**

Eine Ziffer oder Adresse der Kommandozeile ist unzulässig.

***** error: illegal name**

Ein nichtdeklarierter Name für die Komponente einer Struktur wurde verwendet.
Beispiel:

```
union {int i; long j;} un;
un.k=1l;
```

***** error: illegal statement**

Die Anweisung ist an dieser Stelle unzulässig, wie etwa 'break;', das nicht von einer Schleife erfaßt wird.

***** error: illegal use of operator**

Ein Operand wurde auf unzulässige Terme angewandt, etwa wie in '++(m+n)'.

***** error: input conversion**

Bei einer Zahl ist ein Konvertierungsfehler aufgetreten (z.B. falsche Darstellung von Gleitkommazahlen: 1.2EE3).

***** error: integer overflow**

Beim Konvertieren einer ganzen Zahl bzw. beim Auflösen eines konstanten ganzzahligen Ausdrucks ist Überlauf aufgetreten.

***** error: integer zero division**

Beim Auflösen eines konstanten ganzzahligen Ausdrucks sollte durch 0 dividiert werden.

***** error: label not declared**

Eine Marke wurde in einer 'goto'-Anweisung verwendet, ist aber in der gleichen Funktion nicht deklariert (oder nur in einem untergeordneten Block, vgl. Anhang A).

***** error: missing "}"**

Das Textende wurde während einer Funktionsdeklaration erreicht.

***** error: missing ";"**

Eine Anweisung wie 'goto', 'break' oder 'inline(...)' wurde nicht mit einem Semikolon abgeschlossen.

***** error: missing ":"**

Nach 'case' oder 'default' folgt kein ':'; bei einem if-else-Operator '...?...:...' fehlt das ':'.

***** error: missing "?"**

Bei einem if-else-Operator '...?...:...' fehlt '?'.

***** error: missing expression**

An dieser Stelle wird ein Ausdruck erwartet.
Beispiel: if() {...

***** error: missing "if"**

Zu einem 'else' fehlt das 'if'.

***** error: missing constant expression**

An dieser Stelle dürfen nur konstante Ausdrücke stehen.
Beispiel: int a[3][n+1]; case f(1): ...

***** error: missing function**

Hier muß die Definition einer Funktion stehen.
Beispiel: n; /* außerhalb einer Funktion */

***** error: missing label**

Nach 'goto' folgt keine Marke.

***** error: missing name**

An dieser Stelle muß ein Identifikator (ev. eine unäre Operation) folgen.
Beispiel: a+/b; a*b+;

***** error: missing operator**

An dieser Stelle muß ein Operator folgen.
Beispiel: a+b c;

***** error: missing "while"**

Eine 'do'-Schleife endet nicht mit 'while'.

***** error: multiple declared**

Ein Name (einer Variablen/Funktion/Struktur...) wurde mehrfach deklariert.

***** error: multiple declared "default"**

Innerhalb eines 'switch'-Blockes erscheinen mehrere 'default'-Anweisungen.

***** error: multiple declared label**

Ein Markenname wurde innerhalb einer Funktion mehrfach deklariert.

***** error: multiple use of constant**

Die (wertmäßig) gleiche Konstante erscheint in mehreren 'case'-Anweisungen des gleichen 'switch'-Blocks.

***** error: name not declared**

Eine Variable wurde ohne vorherige Deklaration verwendet.

***** error: nesting too deep**

Diese Verschachtelungstiefe ist zu groß (mehr als ca. 25). Das Programm muß umgeschrieben werden.
Dieser Fehler ist selten.

***** error: not implemented**

Ein nicht implementiertes Sprachelement wurde verwendet.

***** error: reserved name**

Ein Schlüsselwort sollte als Indikator dienen.

***** error: statement too long**

Eine Anweisung ist zu lang und muß in mehrere zerlegt werden. Dieser Fehler ist sehr selten.

***** error: syntax**

Ein nicht näher zu bestimmender Syntaxfehler wurde entdeckt.

***** error: type mismatch**

Ein Operator sollte auf einen unzulässigen Datentyp angewandt werden (z.B. '~' auf float), oder in einer Konstruktion wie z.B. "case n:" hat ein Term nicht den geforderten Typ - z.B. wenn hier n vom Typ 'float' ist.

***** error: unbalanced parantheses**

Die Zahl der öffnenden (runden bzw. eckigen) Klammern ist ungleich der Zahl der schließenden.

3.2 End-Meldung

Findet Pretty C während der Compilierung keine Fehler, so beendet es die Übersetzung bei Erkennen des Textendes:

Bei Schalter /LI:0 ohne Meldung,

bei Schalter /LI:n mit n=1,2,3 mit der Meldung

"*** no errors found ***".

3.3 Vordefinierte Funktionen

3.3.0 Allgemeines

Einige wichtige Funktionen können nicht in C ausgeführt werden. Das betrifft z.B. Systemrufe und Erzeugen eines Laufzeitfehlers, aber auch die üblichen Ein- und Ausgabefunktionen printf und scanf, da Pretty C keine variable Argumentzahl zuläßt.
Weil Pretty C keinen Linker benutzt, müssen solche Funktionen vom Compiler vordefiniert werden.
Bei "printf/scanf" und "debug" werden dabei C-Funktionen über eine Maschinenschnittstelle gerufen. Praktisch erfährt der Nutzer also keine Einschränkungen durch vordefinierte Funktionen.

3.3.1 (s)printf/(s)scanf: Ein- und Ausgabe

printf, sprintf, scanf und sscanf sind eigentlich nur "Softwareschnitt-stellen" in Maschinensprache. Das ist nötig, denn sie werden mit variabler Argumentzahl gerufen.
Sie rufen selbst weitere C-Funktionen, und zwar:

```
-p von printf aus, --P von sprintf aus,
-s von scanf aus, --s von sscanf aus.
```

*** In diesem Sinne sind die Namen '_p, __p, _s, __s' reserviert. ***

Die vier Funktionen haben folgende Struktur:

```
int _p(u)  /  int __p(u)  /  int _s(u)  /  int __s(u)
char *u[];
```

Hier ist u ein Feld von Pointern auf jeweils ein Argument, wobei in _p und _s

u[0] der Formatstring,

in __p und __s hingegen

u[0] der Ausgabestring,
u[1] der Formatstring

sind.
Werden n Argumente übergeben (einschließlich Format- und ev. Ausgabestring), so ist u[n] gleich 0.

In _p und __p ist der Typ dieser Pointer entsprechend zu konvertieren, d.h., wenn beispielsweise u[2] die Adresse einer Variablen vom Typ 'long' ist, so ist der Wert dieser Variablen über

```
* (long*) u[2]
```

anzusprechen.

In _s und __s sind die Argumente selbst Pointer, d.h., es ist beispielsweise

```
** (long**) (u[2])
```

zu schreiben, um auf eine long-Variable zuzugreifen, deren Adresse in 'scanf' übergeben wurde.

Der Return-Wert wird von der Schnittstelle an das aufrufende Programm weitergegeben.

Damit ist es dem Nutzer möglich, die formatierte Ein- und Ausgabe selbst zu schreiben bzw. die mitgelieferten Funktionen zu kürzen oder zu erweitern.
Eine Beschreibung der mitgelieferten Funktionen befindet sich in Kap.6.1.

ACHTUNG!! _p, __p, _s, __s und _bdos benutzen gemeinsame Arbeitszellen. Soll z.B. _bdos von _p aus gerufen werden, so ist unbedingt vor und nach diesem Ruf _bsav aus der Bibliothek CEXT zu rufen, weil sonst unsinnige Ergebnisse entstehen - vgl. 6.2.

Des weiteren dürfen alle vier oben beschriebenen Funktionen nicht mit dem Schalter /I+ übersetzt werden (die Adresse von u[] liegt außerhalb des Datenbereichs).

3.3.2 exit: Programmabbruch

Typ und Argumente:

```
int exit(n)
     int n;
```

Wirkung:

Abbruch des Programms;
n=0: ohne Meldung
n≠0: mit backtrace-Meldung wie bei Laufzeitfehlern. Damit kann sich der Nutzer eigene Fehlermeldungen aufbauen, z.B.:

```
isqrt(x)
  int x;
  {
    if(x<0)
      {
       printf("\n\7error: negative argumant in isqr");
       exit(1);
      }
  ...
```

3.3.3 _bdos: Systemruf

Typ und Argumente:

```
struct {int de;
        int be;
        int  a;
       }
              *_bdos();
_bdos(n[,de[,be[,a]]])
```

(Wie üblich können Teile in eckigen Klammern fehlen.)

Wirkung:

Der bdos-Ruf Nr.n wird ausgeführt. Die Anzahl der Argumente ist variabel; ihre Werte werden vor Ausführung des Rufes entsprechend in die Register DE, BC und A geladen.
Nach Rückkehr können diese Register als Komponenten der Struktur, auf die _bdos zeigt, ermittelt werden.
Bei Fehlern während des Systemrufes wird 0 zurückgegeben.

Beispiele:

```
_bdos(0)          Systemwarmstart
_bdos(2,k)        Druck des Zeichens mit ASCII-Code k
_bdos(1)->a       Eingabe eines Zeichens
```

3.3.4 _user: Ruf von Maschinenprogrammen

Typ und Argumente:

```
int _user(jmp,par)
   unsigned jmp,*par;
```

Wirkung:

Es wird zur Adresse "jmp" entsprechend dem Assemblerbefehl CALL gesprungen. HL enthält dabei die Adresse einer Parameterliste, auf die der Pointer "par" zeigt.
Das Maschinenunterprogramm darf alle Register verwenden, auch Schattenregister.
Soll bei Rückkehr ein Resultat übergeben werden, so muß HL auf dieses zeigen.
Bei Hin- und Rückvermittlung von Parametern ist darauf zu achten, daß Pretty C die Daten im Speicher stets von oben nach unten ablegt (s.a.B.3). Insbesondere befindet sich das low-Byte eines integer-Wertes im Byte mit der größeren Adresse!

3.3.5 debug: Debuggerschnittstelle

Falls ein Quelltextteil mit /BP:2 übersetzt wurde, d.h., zum Verfolgen mit dem Debugger, so wird beim Start des Programms eine C-Funktion "debug" gesucht.

*** In diesem Sinne ist der Name 'debug' reserviert. ***

Die Funktion 'debug' wird vom Nutzer mit eingebunden und hat das Format:

```
int debug(val,add,off)
    unsigned off;
    char *add,*val;
```

Die Argumente haben folgende Bedeutung:

1.Aufruf von debug:
      val = Start der Debug-Liste (zur Struktur s.u.);
      add = 0 (daran ist der erste Aufruf zu erkennen);
      off = Länge der Debug-Liste (s.u.).

Folgende Aufrufe von debug:
      val = Adresse des gerade berechneten Ausdruckw (Typ s.u.);
      add = absolute Adresse dieses Ausdrucks im Quelltext;
      off = Basisadresse, zu der die offsets der dynamischen (auto-)
            Variablen zu addieren sind.

Zum Typ von val:

      Die Typvereinbarung für val ist rein formal.
      Wichtig ist nur, daß val die Adresse des Wertes des gerade berechneten Ausdrucks ist. Wenn dieser z.B. vom Typ "double" ist, kann sein Wert über

            (double)val

      erreicht werden.

Zur Struktur der Debug-Liste:

      struct {char *source; char *ret;} list[off];

      val zeigt beim ersten Ruf auf list[off-1]. Über die Werte von list[i].source kann man die Adressen aller möglichen Haltepunkte (breakpoints) im Quelltext ermitteln und so selbst Haltepunkte auswählen und setzen.
      list[i].ret hat für das C-Programm keine Bedeutung.

Über diese Schnittstelle ist es möglich, einen Quelltextdebugger selbst zu schreiben. Der mitgelieferte ist in Kap.8 erläutert.

3.3.6 Die inline-Anweisung

Mit dieser Anweisung kann Maschinencode direkt im Quelltext untergebracht werden. Sie wird formal wie der Ruf einer Funktion vom Typ 'void' geschrieben und erscheint daher in diesem Abschnitt.

Syntax:

      inline(byte1,byte2,byte3...);

Dabei muß 'byte' eine Konstante vom Typ char oder int mit einem Wert kleiner als 256 sein.

*** Es dürfen dabei alle Register verwendet werden. ***

Beispiel:

      if(i==1) inline(62,'*',0xcd,5,0xF3);

Wirkung: Falls i gleich 1 ist, werden die Assemblerbefehle
           LD A, '*'
           CALL 0F305H
         ausgeführt.

In Kenntnis des Abschnitts B.3 kann der Anwender damit kritische, kurze Teile von Programmen in Maschinensprache codieren, ohne die _user-Funktion rufen zu müssen (z.B. putchar in CLIB, strlen in CEXT).

3.4 Ein Beispiel: Das Primzahlsieb

Im folgenden entwickeln wir ein kleines Beispielprogramm, mit dem Sie erste Erfahrungen mit Pretty C sammeln können. Vor allem sollen Sie damit experimentieren.

Wir wollen eine "echte Anwendung" programmieren: Berechne alle Primzahlen bis 1000 (oder einer anderen Zahl) und gebe sie als Tabelle aus.
Das geschieht in mehreren Schritten.

1. Algorithmus.

Wir benutzen das übliche "Sieb des Erastothenes"
- Schreibe alle Zahlen von 2 bis 1000 auf (die 1 ist nach Definition sowieso keine Primzahl).
- Wir wissen, daß 2 eine Primzahl ist. Streiche alle Vielfachen von 2 kleiner als 1000, denn das können keine Primzahlen sein.
- Nun suche die kleinste Zahl größer als 2, die noch nicht gestrichen wurde. Das ist die 3. Sie muß Primzahl sein, da sie kein Vielfaches einer kleineren Zahl ist.
- Streiche alle Vielfachen von 3 kleiner als 1000.
- Suche die kleinste nicht gestrichene Zahl größer als 3. Auch sie muß eine Primzahl sein.
  usw.

Dieser Algorithmus ist der schnellste (und einfachste), um alle Primzahlen bis zu einer vorgegebenen Schranke zu finden.

2. Überlegungen zum Umsetzen des Algorithmus.

Module: Bei dieser kleinen Aufgabe reicht offenbar das main - Programm aus.

Daten: Das "Aufschreiben und Streichen" von Zeilen muß programmtechnisch umgesetzt werden. Es liegt nahe, dafür ein Feld 'prim' zu vereinbaren. Der Feldindex entspricht der Zahl, und das Feldelement enthält die Information, ob die Zahl bereits gestrichen wurde. Für letzteres braucht man nur zwei Werte (z.B. 0 und 1), wozu der Typ 'char' reicht.
Das Feld muß mindestens 999 Elemente enthalten, nämlich für die Zahlen 2...1000 je eines. Das Programm wird jedoch lesbarer (und kürzer!), wenn wir das Feld größer definieren, sodaß der Index gleich der entsprechenden Zahl sein kann, d.h., prim[1000] definiert ist.
Alle Elemente setzen wir zu Beginn gleich 1 (genauer: '\1'). Damit haben wir bereits den Kopf des Programms:

```
char prim[1001];
int i;
for (i=2; i<=1000; i++) prim[i]=1;
```

Algorithmus: Der Algorithmus ist nun leicht umzusetzen.

```
for (i=2; i<=1000; i++)
  {
   if (prim[i]!=0)
     {  /* Primzahl gefunden, streiche alle Vielfachen */
      k=i;
      while ( (k+=i) <= 1000) prim[k]=0;
     }
  }
```

In der while-Schleife wird k jeweils um i vergrößert und gleichzeitig getestet - eine elegante, für C typische Programmierung.

Ausgabe: Wir wollen 6 Zeichen je Primzahl reservieren, d.h., 6 Zahlen auf einer Zeile ausgeben (36 Zeichen). Das ist am einfachsten getan,

wenn wir bei jeder Ausgabe einer Primzahl einen Zähler erhöhen und dessen Rest bei Teilung durch 6 betrachten. Dieser Rest ist jedes 6.Mal gleich 0, und nur dann wird auf eine neue Zeile gewechselt:

```
k=0;
for (i=2; i<1000; ++i)
  {
   if (prim[i] !=0 )
     {
      printf("%6d",i);
      ++k;
      if (k%6 == 0) printf("\n");
     }
  }
```

3. Verbesserungen, Schreiben des Programms.

- Wenn wir später Primzahlen bis zu einem größeren Wert berechnen wollen, müssen wir überall im Programm die '1000' und die '1001' ersetzen. Sinnvollerweise definieren wir sie daher als Konstante mit dem Präprozessor.
- Ein Test 'if(p[i] !=0)' ist redundant, da der Ausdruck nach Sprachdefinition genau dann wahr ist, wenn er ungleich 0 ist. 'if(p[i])' reicht also aus.
- Weiter kann '++k; if (k%6==0)' verkürzt werden zu 'if (!(++k%6))'.

Das Programm sieht nun so aus:

```
/* Sieb des Erastothenes */

#define SIZE 1000

main()
 {
  char prim[SIZE+1];
  int i,k;
  for (i=2; i<=SIZE; ++i) prim[i]=1;
      /* alle Zahlen "aufschreiben" */
  for (i=2; i<=SIZE; ++i)
    {
      if (prim[i])      /* Suche nach der kleinsten, nicht */
       {                /* gestrichenen Zahl */
        k=i;            /* alle Vielfachen zu streichen */
        while (k+=i)<=SIZE) prim[k]=0;
       }
    }                   /* end for */

  /* Druck */

  printf("\nAlle Primzahlen bis %d:\n\n",SIZE);

  k=0;                  /* Spaltenzähler für Tabelle */
  for (i=2; i<=SIZE; ++i)
    {
      if (prim[i])
        {
         printf("%6d",i);
         if ( !(++k%6)) printf("\n");
        }
    }
}

/* ------- */
```

4. Editieren.

Schreiben Sie das Programm wie unter 2.4, Bsp.1, in einen Puffer. Vergleichen Sie es nochmals mit der Vorlage, dann retten Sie es auf Band (2.4, Bsp.2). Hängen Sie mit dem Read-Kommando die Bibliothek CLIBM an.

5. Compilieren.

Übersetzen Sie es durch die Zeile

CC /LI:1

Es dürften keine Fehler auftreten (obiger Quelltext wurde als Block herausgeschnitten und getestet!).
Sollten doch (Tipp-)Fehler auftreten, geben Sie: ESC,ESC+A,korrigieren, ESC+E und nochmals CC (Option /LI:1 bleibt eingestellt).

Achten Sie auf das strukturierte Listing, das der Compiler erzeugt.

6. Programmlauf.

Geben Sie

GO

Nach etwa 4 sec. werden die Primzahlen ausgegeben.

7. Nun machen Sie folgende Experimente:

- Streichen Sie eine geschweifte Klammer (öffnende, schließende, letzte), und übersetzen Sie neu.
  Schreiben Sie 'IF' statt 'if', 'if' statt 'i', 'k=+i' statt 'k+=i'. Was passiert? Versuchen Sie, die Fehlermeldungen zu erklären. Verwenden Sie ESC+A beim Editieren.

- Vergrößern Sie SIZE auf 5000, 10000 und mehr (bis zum "data space overflow"). Stoppen Sie die Zeit für ein möglichst großes SIZE.

- Deklarieren Sie mit '/BP:0' und stoppen Sie die Zeit nochmals. Versuchen Sie auch hier, mit STOP anzuhalten.

- Ersetzen Sie '++i' in der ersten for-Schleife durch '--i' und übersetzen Sie mit dem Schalter '/I+'. Was passiert zur Laufzeit?

- Übersetzen Sie nochmals mit '/I-'. Beim Lauf stürzt das Programm ab, der Speicherinhalt wird zerstört.

3.5 Verwendung von Bibliotheken

Pretty C benutzt keinen Linker. Fast alle Bibliotheksfunktionen sind in Pretty C geschrieben.
Im Normalfall stellt sich der Nutzer die für seine Zwecke am häufigsten benötigten Bibliotheksfunktionen mit EC zusammen, ev. auch mehrere Files für verschiedene Zwecke.
In einfachsten Fällen, wo z.B. nur printf/scanf/putchar/getchar benötigt werden, kann man die Bibliotheksfunktionen auch an den Quellfile hängen, um bei der Programmentwicklung bequemer arbeiten zu können.

Kurze, vollkommen ausgetestete Maschinenprogramme können mittels der inline-Anweisung als C-Funktionen geschrieben werden; sonst sind sie nachzuladen und über _user zu erreichen.

Zur Schnittstelle C/Assembler vgl. Anhang B3.

Es ist sicher (z.B. gegenüber BASIC oder Pascal) etwas umständlicher, jedesmal I/O-Funktionen einbinden zu müssen. Die Vorteile überwiegen hierbei aber eindeutig - außerordentliche Freiheiten wie eigene I/O-Routinen, Verbesserungen der mitgelieferten, Reduzierung auf die wirklich benötigten Funktionen.

3.6 Wenn man an die Grenzen stößt

Tritt während der Compilierung oder der Laufzeit der Fehler "data space overflow" auf, so reicht der Speicherplatz nicht aus. Am unkritischsten ist eine Ursache für den Laufzeitfehler: Eine Funktion wird direkt oder indirekt solange rekursiv gerufen, bis der Stack voll ist. Das ist fast immer ein Programmfehler.

In allen anderen Fällen bleiben dem Programmierer folgende Möglichkeiten:

- Der Raum für Daten und Stack wird durch den Textpuffer mit der kleinsten Anfangsadresse begrenzt. Der Puffer kann nach hinten verschoben werden.

- Sind sehr große dynamische Felder definiert (Speicherklasse auto), so kann der Textpuffer manchmal geschickt zwischen Stack und Liste während der Compilierung untergebracht werden (vgl. Anh.B2). Das Speicherende ist dann mit /WE:nn zu setzen.

- Der Quelltext ist ggf. in mehrere Files zu zerlegen. Diese Files werden nacheinander übersetzt /Schalter /CN), wobei sie zwischen den Compilerläufen eingelesen und im Speicher möglichst hinten plaziert werden. Damit kann man schon sehr große Programme übersetzen.

- Reduzieren von Bibliotheksfunktionen. Z.B. kann man in '*_p' viele case-Anweisungen streiche, wenn die entsprechenden Formate nicht verwendet werden. Treten im Programm keine Gleitkommatypen auf, so sollte man CLIBM statt CLIB nutzen.

- Bei zu großen Moduln muß mit /TP direkt vom Band übersetzt werden. Dann geht der schnelle Zyklus Editieren-Compilieren-Testen allerdings verloren.

- Schließlich kann oft auch der für Konstanten reservierte Speicherplatz /standardmäßig 1KByte) verkleinert werden (Schalter /CS).

Mehr Möglichkeiten hat der Nutzer bei Version 1.0 nicht, da ein Auslagern des übersetzten Codes auf Band noch nicht möglich ist.

3.7 Hinweise für effektive Programmierung

Über effektive Programmierung in C allgemein zu schreiben, würde den Rahmen dieser Dokumentation sprengen.
Wichtig sind aber einige Hinweise, die speziell für Pretty C Vorteile bringen:

- Wie in C vorgeschrieben, sind konstante Ausdrücke äquivalent zu Konstanten. Sie werden bereits zur Compilezeit aufgelöst. Alle in solchen Ausdrücken verwendeten Konstanten gehen wieder verloren und beanspruchen keinen Speicherplatz, nur der berechnete Wert wird gemerkt.
  Das gilt auch für konstante Teilausdrücke, die geklammert sind.

- Konvertierungen beanspruchen viel Speicherplatz und Zeit. Es ist zum Beispiel sehr unklug zu schreiben:

```
long l,m; m = 1 + 3;
```

Hier muß die '3' erst auf 'long integer' konvertiert werden. Vor dieser Konvertierung wird die '3' in eine Stackzelle geladen, deren Adresse berechnet werden muß. Gegenüber

```
long l,m; m = 3 + 3l;
```

hat man 2 Bytes Konstantenspeicher gespart, aber 6 Bytes Code und viel Rechenzeit verschwendet!

- In diesem Zusammenhang sei daran erinnert, daß 'char'-Variable in C-Ausdrücken auf den 'integer' konvertiert werden! Dieser Datentyp ist also mit Bedacht zu verwenden. Pretty C führt Konvertierungen char>int nur bei Zuweisungen nicht aus.

- Im Unterschied zu üblichen Empfehlungen sind Operationen mit 'float'-Variablen **schneller** als mit 'double'-Typen, da Pretty C zwei getrennte Gleitkommaarithmetiken realisiert und den Typ 'float' nicht automatisch auf 'double' konvertiert.

- Postinkrement- und dekrementoperationen sind auf eine andere Hardware zugeschnitten. Auf dem U880 müssen sie relativ umständlich realisiert werden ('++i' kostet 6 Bytes, 'i++' dagegen 13 Bytes Code).
  Das gilt aber nur, wenn das Resultat dieser Operatoren weiterverwendet wird, vgl. Anh.B3.

- Bei '#undef' wird eine Definition nur logisch gelöscht, sie beansprucht weiterhin Speicherplatz!

- switch-Anweisungen sind effektiver als entsprechende if-else-Ketten.

- Zu Schleifen: for-Anweisungen werden bei Fehlen von Teilausdrücken optimiert. 'for(;;)' ist die effektivste Codierung einer Endlosschleife in Pretty C (außer 'goto', das man ja selten verwenden sollte).
  Für einfache Laufanweisungen ähnlich der DO-Schleife in FORTRAN ist folgende Möglichkeit die effektivste (schnell und klein):

  ```
  i = n+1; while (--i) {...
  ```

  (Die Schleife wird n-mal durchlaufen.)

3.8 Fehlerquellen in C-Programmen

Nach ersten Erfolgen wird ein C-Programmierer sicherlich bald längere und schwierigere Programme schreiben. Diese Programme sind schnell von Syntaxfehlern befreit, aber sie "laufen einfach nicht". Die Fehlersuche ist wahrscheinlich die Hauptbeschäftigung des C-Programmierers. Wenigstens in gedrängter Form soll deswegen etwas zum Programmierstil und vor allem zu häufigen Fehlerquellen etwas gesagt werden.

Zum Programmierstil:

- C ist auf modulare Programme orientiert, und das sollte man unbedingt voll ausnutzen. Keine C-Funktion sollte länger als 200 Zeilen sein (in der Regel viel kürzer, manchmal nur eine Zeile!). Assemblerprogrammierer überlegen in der Regel sehr gründlich, wann es effektiv ist, ein Unterprogramm zu schreiben (hinsichtlich Speicherplatz und Laufzeit). Nur einmal benötigte Programmteile werden eingeschoben oder angesprungen. Dieser Stil ist in C grundfalsch. Grenzen Sie Funktionen nach ihren Aufgaben ab, auch wenn viele Funktionen nur einmal aufgerufen werden. Unterteilen Sie Funktionen mit zu komplexen Aufgaben in Teilaufgaben, bis diese zu klein werden. Sie bezahlen dafür kaum Laufzeit und etwas mehr Speicherplatz, gewinnen aber viel bei Klarheit der Programmierung und Änderungsfreundlichkeit. Versuchen Sie unbedingt, rechnerabhängige Details in gesonderten Funktionen unterzubringen. Das sind gewichtige Vorteile. Sie wissen nie, ob Sie das Programm nicht eines Tages auf einen anderen Rechner übertragen müssen.

- Überlegen Sie sich Datenfluß und Datenstruktur in Ihrem Programm schon während der Einteilung der Teilaufgaben. Das heißt nicht, daß man schon jede Laufvariable vorher planen soll, sondern: Die wenigen entscheidenden Variablen, Felder und Strukturen sollte man sich nicht erst beim Schreiben des Codes überlegen. Verwenden Sie so wenig wie möglich globale Variable. Deklarieren Sie Funktionen und Variable, wenn möglich, als 'static' (wobei Pretty C leider keine 'static'-Funktionen kennt).

- Kommentieren Sie Ihre Programme hinreichend, damit sich später ein anderer wieder hineinfinden kann – das sind z.B. Sie nach 3 Monaten.
  Treiben Sie es aber nicht soweit, daß die Übersichtlichkeit verlorengeht (bekanntes Beispiel: ++i; /* new increase i */).
  Empfehlenswert sind:

  - Übersichtliche Abgrenzung der einzelnen Funktionen, Beschreibung ihrer Aufgabe (muß nicht lang sein!). Was wird beim Aufruf übernommen, was zurückgegeben? Welche Änderungen können wo gemacht werden?

  - Kommentierung der Bedeutung einzelner Variablen, Strukturen usw., wenn sie wichtig sind.

  - Markierung wichtiger Programmteile.

- Nutzen Sie den Präprozessor zur Definition von Konstanten, die ev. Geändert werden können (Felddimensionen, rechnerabhängige Größen). Kommentieren Sie auch diese Konstanten.

- Seien Sie sehr vorsichtig mit "Trickprogrammierung", der Hauptfehler in C-Programmen. Überlegen Sie sich, welche Funktionen wirklich zeitkritisch sind. Das heißt nicht, daß "i=i+1" statt "++i" stehen soll.
  Es gibt eine goldene Regel zur Programmierung:
  Erst bringe das Programm zum Laufen,
  dann mache es fehlerfrei,
  dann mache es schnell und klein.

Zu Fehlerquellen:

Es gibt ebenso viele Fehlerquellen in C-Programmen wie C-Programmierer. Dennoch sollen einige "beliebte" Quellen angeführt werden. Die Reihenfolge ist dabei keine Wichtung, ohnehin ist dieses Gebiet schwer zu systematisieren.
Bedenken Sie aber bitte stets: Fehler sind in C allzu leicht gemacht, und das gestartete Programm stürzt auf unerklärliche Weise ab.
R e t t e n S i e I h r e n Q u e l l t e x t , b e v o r S i e d a s P r o g r a m m s t a r t e n . Der KC 85/1 hat keinen Speicherschutz.

- Abstürze bei Pretty C können eigentlich nur drei Ursachen haben (falls Sie keinen Compilerfehler gefunden haben!):
  - Ein Feldindex ist aus dem zulässigen Bereich herausgelaufen.
  - Einem Pointer wurde irgendwo ein unsinniger Wert zugewiesen.
  - Ein Pointer vom Typ "Zeiger auf eine Funktion" zeigt auf **k e i n e** Funktion (obwohl dieser Fehler mit 99,6% Wahrscheinlichkeit die Fehlermeldung "number of arguments" bewirkt). Die ersten beiden Ursachen sind mit dem Schalter /I+ (3.0.10) auszuschließen.

- Die Typen der Argumente beim Funktionsaufruf und in der Funktionsdefinition stimmen nicht überein. Solche Fehler kann C generell nicht feststellen (Prinzip der unabhängigen Übersetzung einzelner Module) und sind sehr schwer zu lokalisieren.

- Felder beginnen beim Index 0. Wenn "int a[13];" vereinbart wurde, dann ist a[i] für i=0,...,12 definiert!

- Operatoren wurden verwechselt, ihre Priorität nicht berücksichtigt. Vor allem betrifft das:
  = statt == bei logischer Äquivalenz,

& und | statt && und || bei logischem UND und ODER;
"c=*++p && c!='\n'" kann z.B. als "(c=*++p) && c!='\n'" gemeint sein;
Bitoperationen haben niedrigere Priorität als arithmetische, ebenso Verschiebeoperationen.

- Ausdrücke haben Nebeneffekte, d.h., ihr Wert hängt von der Reihenfolge der Compilierung ihrer Teilausdrücke ab. Beispiel:
  ```
  f(*p++,p);
  ```
  Solche Konstruktionen sind prinzipiell nicht auf andere Rechner übertragbar und daher unakzeptabel. Diese Fehler sind sehr schwer zu finden!

- In Funktionen wie scanf wurde:
  'scanf("%d",i);' statt 'scanf("%d",&i);' geschrieben.
  Dieser Fehler verursacht Abstürze, da scanf (genauer: _s) nicht mit Schalter /I+ übersetzt werden darf.
  Dieser Fehler ist sehr häufig!

- Bei einer return-Anweisung wird kein Wert angegeben, obwohl ein Funktionswert verwendet wird.
  Zur teilweisen Vermeidung solcher Fehler ist der Typ "void" eingeführt worden!

- Beim Funktionsaufruf wird ein mehrdimensionales Feld übergeben, in der aufgerufenen Funktion aber als Mehrfachpointer verwendet. Das wird meistens zu illegalen Adreßwerten führen.

- char *p; p="hello"; bedeutet nur, daß p jetzt auf die Anfangsadresse des Strings "hello" zeigt. Der String selbst wurde nicht zugewiesen. Ebenso werden bei Funktionsaufrufen nur Feldadressen übergeben, nie ganze Felder.

- Lokale Variable, die nicht als 'static' deklariert sind, verlieren i.a. ihren Wert nach Verlassen des Blocks, in dem sie definiert wurden.

- "x" ist etwas Anderes als 'x' !

- Zeichenketten wurden berechnet, aber nicht mit '\0' abgeschlossen.

- Groß- und Kleinbuchstaben werden in C unterschieden!

- Eine "else"-Anweisung gehört stets zum letzten "if"! Dieser Fehler ist oft äußerst schwer zu finden.

- Eine Anweisungsfolge nach "case" schließt nicht mit "break;" ab (das kann aber auch Absicht sein!).

- Nach "while,if,for" kein Semikolon, wenn Anweisungen ausgeführt werden sollen!

- Mittels "#define" definierte Macros dürfen fast nie ein Semikolon enthalten.

- Konvertierungsregeln werden nicht beachtet. Z.B. wird bei der Konvertierung von "int" auf "long unsigned" erst auf "unsigned", dann auf "long unsigned" konvertiert.

- Ist p ein Pointer, so heißt "++p" **n i c h t** Erhöhung von p um 1, sondern um die Bytelänge des Typs, auf den p zeigt!

- Überprüfen Sie, ob es "++i" oder "i++" heißen muß. Solche Denkfehler sind in C häufig (oft wegen der Schwierigkeit der in Angriff genommenen Probleme).

Kapitel 4: Der Kommandoprozessor C@

Der Kommandoprozessor ist sehr klein und einfach, aber so wichtig, daß ihm ein eigenes Kapitel gehören sollte.
Schon bei Übersetzung von gestreutem Quelltext, erst recht bei Anwendung des Debuggers, muß der Compiler bei jeder Neuübersetzung mindestens 2-3mal gestartet werden, jedesmal mit einer anderen Schalterkombination (der Weg über Schalter im Quelltext ist nach Ansicht des Autors umständlicher als die vorliegende mit C@).
Damit die Kommandozeilen nicht jedesmal neu eingetippt werden müssen (eine nicht zu vergessende Fehlerquelle!), können wir sie in einen Textpuffer mit dem Namen CMD schreiben.
Durch das Kommando

```
C@    ('@' = "Klammeraffe", Code 64)
```

wird ein Puffer mit dem Namen CMD gesucht. Ist er gefunden (sonst Fehlermeldung), so werden seine Zeilen nacheinander abgearbeitet:

- Beginnt eine Zeile mit '/' oder wird das Pufferende erreicht, so bricht C@ ab.
- Beginnt eine Zeile mit '.', so unterbricht C@ seine Arbeit mit der Meldung "delaying" und kann durch das Monitorkommando "R@" fortgesetzt werden. Das ist günstig, wenn z.B. zwischen einzelnen Compilerläufen Quelltext nachgeladen werden soll.
- Ansonsten muß die Zeile eine Kommandozeile zum Start von CC sein und wird so ausgeführt, als wäre sie direkt eingegeben worden.

Bricht CC wegen eines Fehlers ab, so bricht auch C@ ab, ebenso nach Ausführung des Schalters /DS.

Die gerade abgearbeitete Kommandozeile wird jeweils mit vorgesetztem "###" angezeigt.
Beachten Sie bitte, daß **nur** der Compiler von C@ aus gestartet werden kann, nicht der Editor oder GO!

Beispiel:

Wir wollen ein Programm mit dem Debugger testen. Das Programm stehe im Puffer CQ (vgl.a.Kap. 9 und 12):

```
EC /DI

A000 A080 CQ
A200 A437 debug
A600 AB18 printf

>EC CMD/CR
filename:CMD
*CC CQ/BP:2/LI:1/AS:100
*CC debug/BP:0/LI:0/CN
*CC printf/CN
*
E>
```

(Zweckmäßigerweise wird man hier mittels ESC+C Großbuchstaben als Standard angeben.)
Nun starten wir den Kommandoprozessor:

```
>C@
### CC CQ/BP:2/LI:1/AS:100

...

main({
"(global)" A012
```

```
      *** error: declaration syntax

      >
```

C@ hat abgebrochen. Wir geben "EC" (der aktuelle Filename ist ja noch CQ!), "ESC+A":

```
      >EC
      filename:CQ


      _

      A
      main( {
```

Wir korrigieren den Fehler und starten neu:

```
      ### CC CQ/BP:2/LI:1

      ....

      main() {
        int n; n = -3;
        if (n<0) n = -n;}

      * compilation ended - no errors *

      ### CC debug/BP:0/LI:0/CN
      ### CC printf/CN
      >
```

Kapitel 5: Laufzeitroutine GO

Fehlerfrei übersetzte C-Programme werden durch das Laufzeitsystem abgearbeitet. Es enthält alle im generierten Code gerufenen Unterprogramme einschließlich einfach- und doppeltgenauer Gleitkommaarithmetik sowie Hilfsroutinen zur Initialisierungsphase, Fehlerbehandlung und Programmunterbrechung, Funktionsaufruf und v.a.m.

Das Laufzeitsystem ist in vorliegender Form noch mit dem Compiler verbunden.

5.0 Kommandozeile, Startphase

In vorliegender Version lautet die Kommandozeile einfach

```
GO
```

- andere Bestandteile sind bedeutungslos. Daher darf die Funktion 'main' in Pretty C auch keine Argumente enthalten!

Nun

- wird geprüft, ob die Übersetzung ordnungsgemäß abgeschlossen wurde (sonst Gefahr des Systemabsturzes!),
- werden einige Arbeitszellen initialisiert und der Stackpointer umgesetzt,
- wird überprüft, ob alle verwendeten Funktionen tatsächlich definiert wurden,
- werden alle statischen Variablen mit dem Wert 0 initialisiert,
- wird die Funktion mit dem Namen "main" gesucht und gerufen.

Nach Beendigung des C-Programms bzw. nach Laufzeitfehler wird der Stackpointer wieder zurückgesetzt.

5.1 Fehlermeldungen

5.1.1 Fehlerbehandlung, Aufbau der Meldung

Es gibt zwei Arten von Fehlern:

1. Fehler während der Startphase.

   Das sind nur zwei Fehler:

   a) *** command illegal

      Trotz Abbruchs der Compilierung wegen Fehlers wurde GO gegeben.

   b) -> xxx not defined

      Die Funktion 'xxx' wurde verwendet, aber in keinem Puffer definiert.

   Nach diesen Fehlermeldungen bricht GO kommentarlos ab.

2. Fehler während der Laufzeit.

   Wie zur Compilezeit beginnen diese Fehler mit

   *** error:

   Danach wird eine backtrace-Liste ausgegeben, was am besten am Beispiel zu erklären ist:

```
*** error: integer zero division

   at PC=4E11   in:  descend
 from PC=4CF6   in:  descend
 from PC=4CF6   in:  descend
 from PC=4E1A   in:  test_chain
 from PC=4F28   in:  main
 from (sys)
```

Bedeutung: Vor der Maschinenadresse 4E11 im generierten Code wurde durch 0 dividiert, beide Operanden waren ganzzahlig. Die kritische Anweisung liegt in der Funktion "descend", die wiederum von "descend" gerufen wurde, und zwar vor der Adresse 4CF6 usw. (im Beispiel wurde "descend" rekursiv aufgerufen). Über den Schalter /EL (s.3.0.12) können die angegebenen PC sofort mit dem Quelltext in Verbindung gebracht werden.

Achtung!
Um ggf. die backtrace-Liste erstellen zu können, wird keinerlei zusätzliche Rechenzeit benötigt. Die Fehlerbehandlungsroutinen analysieren den Stack.
Dadurch ist es aber theoretisch möglich (und kommt auch sehr selten vor), daß die erste "PC="-Adresse unsinnig ist. In der Praxis entstehen dadurch kaum Probleme, wenn man weiß, daß dieser Fehler überhaupt auftreten kann.

Danach bricht GO ab.

In einem Fall wird keine backtrace-Liste ausgegeben, nämlich bei den Fehlern erster Art
-> _p not defined (bzw. __p, _s, __s).
Das liegt daran, daß printf und scanf zwar vordefiniert sind, aber erst nach ihrem Aufruf die C-Funktionen p_ bzw. s_ suchen (vgl. 3.3.1).

5.1.2 Liste der Fehlermeldungen

Es werden nur die reinen Laufzeitfehler angeführt (s.5.1.1).

***** error: floating overflow**

Bei einer Gleitkommaoperation trat Überlauf auf, d.h., das Endergebnis würde größer als etwa 1.e+38 werden.

***** error: floating zero division**

Bei einer Gleitkommadivision war der Devisor 0.

***** error: integer overflow**

Bei einer arithmetischen Operation mit int- oder long-Operanden trat Überlauf auf (bei vorzeichenlosen Operanden wird diese Meldung unterdrückt und entsprechend der Sprachdefinition modulo 2^16 bzw. 2^32 gerechnet).

***** error: integer zero division**

Bei einer Division mit ganzzahligen Operanden war der Divisor 0.

***** error: number of arguments**

Die bei einem Funktionsaufruf ermittelte Zahl von Argumenten ist verschieden von der Argumentzahl in der Funktionsdefinition. Beim Start mit GO wird 'main' als Funktion ohne Argumente gerufen, daher erzeugt auch 'main(argc,argv)' diesen Laufzeitfehler.
Ausnahme: (s)printf, (s)scanf, _bdos (diese Funktionen werden über eine spezielle Assembler-Schnittstelle gerufen).

***** error: pointer out of data space**

Dies ist die zum Schalter /I+ (vgl.3.0.10) gehörende Fehlermeldung.

break

Diese Meldung erscheint bei Abbruch mit der STOP-Taste. vgl. 5.2.

5.2 Programmunterbrechung und -abbruch

Wird beim Compilieren der Schalter /BP:1 gesetzt (Standard), so fügt CC an jedem Blockende einen Breakpoint ein, d.h., es wird überprüft, ob eine Taste gedrückt wurde.
Dabei wird nur auf zwei Tasten reagiert:

STOP Das Programm bricht mit der Meldung "*break*" (s.0.) und einem backtrace-Listing ab.
PAUSE Das Programm hält an (ohne Meldung) und kann durch Drücken einer beliebigen Taste fortgesetzt werden (mit STOP Abbruch).

PAUSE ist insbesondere wichtig, um längere Bildschirmausgaben anhalten zu können.
Durch die Unterbrechbarkeit werden die Programme in sehr unterschiedlichem Maße langsamer (unwesentlich, aber auch Faktor 2...5).

Mit /BP:0 übersetzte Programme können nicht mehr abgebrochen werden, d.h., nur ausgetestete Programme sollten mit dieser Option übersetzt werden!

Kapitel 6: Bibliotheksfunktionen

"C ist so gut wie seine Bibliothek."

In dieser Hinsicht ist Pretty C gegenüber größeren Compilern nicht konkurrenzfähig, denn seine mitgelieferten Bibliotheken sind keineswegs so umfangreich wie üblich. Jedoch ist dieser Nachteil nicht allzu groß, denn:

- Fast alle Bibliotheksfunktionen werden als C-Programme gerufen. Der Nutzer erhält die Quelltexte und kann sie beliebig verbessern und erweitern.

- Durch '_bdos' ev. auch '_user' und 'inline' hat der Nutzer vollen Zugriff zum Betriebssystem. Fehlende Funktionen sind oft leicht zu schreiben (außer 'fprintf' und 'fscanf', die wegen ihrer variablen Argumentzahl nicht ohne weiteres codiert werden können. Allerdings sind sie mit 'sprintf' und 'sscanf' zu umschreiben).

Auf Fileoperationen wurde völlig verzichtet - wegen der fehlenden Peripherie.

6.1 Aufbau der Bibliotheken

Pretty C wird mit 3 Bibliotheken geliefert: CLIB, CLIBM und CEXT.

- CLIB enthält die fast immer notwendigen Ein- und Ausgabefunktionen (s)printf, (s)scanf, putchar, getchar und die dafür benötigten Hilfsfunktionen. Diese Bibliothek ist für einfache Anwendungen gedacht und wird in der Regel als Ganzes verwendet. Der Quelltext ist ca. 9KBytes lang und erzeugt 7,3KBytes Code. Bis auf eine integer-Größe sind alle Variablen dynamisch (Speicherklasse 'auto').

- CLIBM ist eine reduzierte Version von CLIB, die keine Gleitkommaein- und -ausgabe unterstützt.

- CEXT enthält zahlreiche speziellere Funktionen wie Stringverarbeitung, Bandverwaltung, mathematische Funktionen. CEXT ist thematisch und innerhalb einer Gruppe alphabetisch geordnet. Vor jeder Funktion steht ein Kommentar in folgender Form:

  ```
  /*$name: Liste der benötigten Funktionen */
  ```

  Alle Hilfsfunktionen, die nur von dieser Funktion gerufen werden, sind in obiger Liste nicht enthalten, sondern noch vor dem nächsten Kommentar der Form '/*$...' eingefügt.

  Die benötigten Funktionen müssen nun mit EC wie folgt extrahiert werden:

  1. Einlesen der gesamten Bibliothek.
  2. Suchen einer benötigten Funktion mit 'TOP','FIND $name'.
  3. Markieren dieser Zeile als Blockanfang.
  4. Suchen von '/*$' mit FIND.
  5. Markieren der vorhergehenden Zeile als Blockende.
  6. Sprung zum Pufferende (BOTTOM), dort INSERT BLOCK, DELETE BLOCK.
  7. Wiederholung ab 2., bis alle gewünschten Funktionen einschließlich der von ihnen benötigten am Pufferende stehen. Die Extraktion kann natürlich auch ganze Gruppen von Funktionen umfassen.
  8. TOP, Blockanfang markieren, Suche von '$$$' (ursprüngliches Pufferende), Markieren als Blockende und Löschen des Blocks.
  9. Den erzeugten Textpuffer auf Band auslagern.

Selbstverständlich kann man sich auch ein "Linkerprogramm" schreiben, das diese Arbeit vereinfacht (aber lohnt sich das?).

Einheitlich sind die Namen aller nicht dokumentierten Hilfsfunktionen und globaler Variablen mit '_' am Beginn gekennzeichnet. Der Nutzer sollte solche Namen meiden, um nicht in Konflikte zu geraten.
Die Winkelfunktionen sin, cos, tan, arctan, fsin, fcos, ftan, und farctan benutzen das vor 'cos' definierte Macro PI.

Schließlich noch ein Hinweis: Aus Speicherplatzgründen werden Kommentare in den Bibliotheksfunktionen praktisch weggelassen, in CLIB(M) auch alle Einrückungen (Compilerlisting benutzen!). Oft wurde bewußt die Möglichkeit gewählt, die in Pretty C den kürzesten Code erzeugt.
Die Listings der Bibliotheksfunktionen sind also **kein** Beispiel für klaren, übersichtlichen Programmierstil!

6.2 Liste der Bibliotheksfunktionen

Funktionen, die üblicherweise nicht in C-Bibliotheken enthalten sind, sind mit "###" in der jeweiligen Überschrift markiert.

Bibliotheken CLIB/CLIBM

Diese Bibliothek enthält die für die Ein- und Ausgabe fast immer benötigten Funktionen printf (sprintf), scanf (sscanf), getchar und putchar sowie gets. Die Namen von Hilfsfunktionen beginnen mit '_' und sollten nicht verwendet werden.
Treten im Programm keine Gleitkommazahlen auf, so sollte man CLIBM verwenden und so Speicherplatz sparen.

***** putchar**

Syntax: int putchar(int c)
(c kann auch vom Typ 'char' sein.)

Wirkung: Ausgabe des Zeichens c auf dem Bildschirm. Dabei wird '\n' in "\r\n" umgewandelt (da es in CP/M kein Zeichen für "neue Zeile" wie in UNIX gibt).

***** getchar**

Syntax: int getchar()

Wirkung: Eingabe eines Zeichens über Tastatur. Das Zeichen wird gleichzeitig auf dem Bildschirm ausgegeben.

Besonderheiten: Bei Eingabe von STOP ('\3') bricht das Programm ab.
Bei Eingabe von ENTER ('\r') wird '\r\n' ausgegeben. und dann '\n' von getchar zurückgegeben.
Alle anderen Steuerzeichen werden zwar zurück-, aber nicht ausgegeben.

***** gets**

Syntax: char gets(char *buf)

Wirkung: Eingabe einer Zeichenkette von der Tastatur. Die Zeichenkette wird als Pointer über den Funktionswert zurückgegeben und im Feld buf gespeichert, das groß genug sein muß, diese Zeichenkette aufzunehmen.
Bei Eingabe von STOP Abbruch des Programms.

Besonderheiten: gets ruft über inline-Anweisungen Teile von Pretty C. Dadurch sind alle Zeichenkommandos des Editors wirksam, d.h., man kann bei Stringeingabe über gets wie bei der Eingabe einer neuen Zeile im Editor verfahren (vgl. 2.3.1).

***** printf, sprintf**

Syntax: printf(char *format, arg1,...,arg n)
sprintf(char *s, char *format, arg1,...,arg n)

Wirkung: Übliche formatierte Ausgabe. printf gibt auf dem Bildschirm aus, während sprintf die gleiche Zeichenkette wie printf in das Feld s schreibt, das groß genug sein muß.
Die Syntax hält sich an Kernighan/Ritchie, einige Abweichungen sind:
%u für vorzeichenlose Zahlen,
%h gibt es nicht;
%le, %lf, %lg für double-Größen (da in Pretty C ein float-Argument nicht automatisch auf 'double' konvertiert wird).
Return-Wert: Anzahl der geschriebenen Bytes.

Besonderheiten: printf und sprintf sind vordefiniert, CLIB enthält nur die Funktionen _p und __p, vgl. 3.3.1 .
Die Ausgabe von printf geschieht mittels putchar.

***** scanf, sscanf**

Syntax: int scanf(char *format, arg1,...,arg n)
int sscanf(char *s, char *format, arg1,...,arg n)

Wirkung: Übliche formatierte Eingabe. scanf liest von der Tastatur, sscanf nimmt den String s als Eingabe.
Auch hier ist die Arbeitsweise entsprechend Kernighan/Ritchie, aber mit Einschränkungen:
- Zwischen '%' und dem Typspezifikator darf nur 'l' stehen;
- ein ev. nicht mehr verarbeiteter Teil des Eingabestrings geht verloren.
- Zurückgegeben wird stets die Zahl der gelesenen Argumente, nie EOF (-1).
- Gleitkommazahlen müssen der Syntax von C genügen, d.h., '.' und ev. einen Exponentialteil enthalten. Bei Überlauf während der Eingabe bricht das Programm ab.

Besonderheiten: scanf und sscanf sind vordefiniert, CLIB enthält nur die Funktionen _s und __s, vgl. 3.3.1 .
'scanf' liest über 'gets' von der Tastatur, d.h., alle EC-Zeichenkommandos sind wirksam (vgl. 2.3.1).
Insbesondere wird durch STOP das Programm abgebrochen.

Bibliothek CEXT

Folgende 4 Gruppen von Funktionen werden beschrieben:

I/O - Ein- und Ausgabe
STRING - Zeichen- und Zeichenkettenverarbeitung
MATH - mathematische Funktionen
SYS - Systemfunktionen

Gruppe I/O

***** cload ###**

Syntax: int cload(char *name, char *buf, int nbytes)

Wirkung: Einlesen des Files 'name' vom Magnetband in einen Puffer 'buf'; es werden höchstens 'nbytes' Bytes gelesen.
Bei Lesefehlern kann wie üblich durch Drücken einer Taste der fehlerhafte Block nochmals gelesen werden (Abbruch der Funktion mit STOP).
Steht an Stelle von 'name' eine 0, so wird der nächste File gelesen und angezeigt.
Return-Wert:
0 - nbytes wurden gelesen;
1 - es wurden weniger als nbytes Bytes gelesen.

***** cls ###**

Syntax: int cls()

Wirkung: Schnelles Bildschirmlöschen, der Cursor wird in die linke obere Ecke gesetzt.

***** csave ###**

Syntax: int csave(char *name, char *buf, int nbytes)

Wirkung: Es wird das Feld buf[nbytes] unter dem Filenamen 'name' auf Magnetband geschrieben.
Return-Wert:
0 - Ausgabe verlief fehlerfrei;
1 - ein Fehler bzw. Abbruch mit STOP traten auf.

***** kbhit ###**

Syntax: int kbhit()

Wirkung: Liefert 0 zurück, wenn keine Taste gedrückt ist, sonst den entsprechenden Zeichencode. Das Zeichen erscheint nicht auf dem Bildschirm.

***** kbin ###**

Syntax: int kbin()

Wirkung: Eingabe eines Zeichens ohne Echo auf dem Bildschirm.
Bei Eingabe von STOP ('\3') Programmabbruch.

Gruppe STRING

***** strcat**

Syntax: char *strcat(char *a, char *b)

Wirkung: Anhängen des Strings b an den String a. Beide Strings müssen mit 0 abgeschlossen sein, und a muß groß genug sein, um b noch aufzunehmen.
Return-Wert: a.

***** strchr**

Syntax: `char *strchr(char *s, char c)`

Wirkung: Suchen des Zeichens c im String s.
Return-Wert:
Adresse des Zeichens im String s, falls das Zeichen gefunden wurde, 0 sonst.

***** strcmp**

Syntax: `int strcmp(char *a, char *b)`

Wirkung: Vergleich der beiden Strings a und b in lexikografischer Ordnung.
Return-Wert:
-1 bei a < b,
0 bei a = b,
1 bei a > b.

***** strcpy**

Syntax: `char *strcpy(char *out, char *in)`

Wirkung: Kopieren des Strings 'in' in den String 'out', der groß genug sein muß, 'in' aufzunehmen.

***** streq**

Syntax: `int streq(char *a, char *b)`

Wirkung: Return von 1, falls die Strings a und b gleich sind, 0 sonst.

***** strlen**

Syntax: `int strlen(char *s)`

Wirkung: Rückgabe der Stringlänge von s; '\0' wird nicht mitgezählt.

***** tolower**

Syntax: `int tolower(int c)`

Wirkung: Falls c ein Großbuchstabe ist, Rückgabe des entsprechenden Kleinbuchstabens, sonst von c.
c kann auch als char übergeben werden.

***** toupper**

Syntax: `int toupper(int c)`

Wirkung: wie tolower, nur ev. Konvertierung in Großbuchstaben.

Gruppe MATH

Bei fehlerhaften Argumenten erfolgen Fehleranzeige und Programmabbruch.
Die doppelt genauen Funktionen sind in jedem Fall mit /F- zu übersetzen!

***** abs**

Syntax: int abs(int a)

Wirkung: Rückgabe des Absolutwertes von a.

***** arcsin**

Syntax: double arcsin(double x)

Wirkung: Rückgabe von arcsin(x) im Bogenmaß.

***** arctan**

Syntax: double arctan(double x)

Wirkung: Rückgabe von arctan(x) im Bogenmaß.

***** cos**

Syntax: double cos(double x)

Wirkung: Rückgabe von cos(x), x im Bogenmaß.

***** exp**

Syntax: double exp (double x)

Wirkung: Rückgabe von e^x.

***** fabs**

Syntax: double fabs(double a)

Wirkung: Rückgabe des Absolutwertes von a.

***** farcsin** ###

Syntax: float farcsin(float z)

Wirkung: wie arcsin.

***** farctan** ###

Syntax: float farctan(float z)

Wirkung: wie arctan.

***** fcos**

Syntax: float fcos(float z)

Wirkung: wie cos.

***** fexp** ###

Syntax: float fexp(float z)

Wirkung: wie exp.

***** ffabs ###**

Syntax: float ffabs(float z)

Wirkung: wie fabs.

***** flog ###**

Syntax: float flog(float z)

Wirkung: wie log.

***** fsin ###**

Syntax: float fsin(float z)

Wirkung: wie sin.

***** fsqrt ###**

Syntax: float fsqrt(float z)

Wirkung: wie sqrt.

***** ftan ###**

Syntax: float ftan(float z)

Wirkung: wie tan.

***** log**

Syntax: double log(double x)

Wirkung: Rückgabe des natürlichen Logarithmus von x.

***** max**

Syntax: int max(int a, int b)

Wirkung: Rückgabe des größeren der beiden Werte a und b.

***** sin**

Syntax: double sin(double x)

Wirkung: Rückgabe von sin(x), x im Bogenmaß

***** sqrt**

Syntax: double sqrt(double x)

Wirkung: Rückgabe der Quadratwurzel von x.

***** tan**

Syntax: double tan(double x)

Wirkung: Rückgabe von tan(x), x im Bogenmaß.

Gruppe SYS

*** _bsav

Syntax: int _bsav(int n)

Wirkung: Retten bzw. Restaurieren der gemeinsamen Arbeitszellen von (s)printf, (s)scanf und _bdos.
Ist n>0, so werden n Zellen gerettet; ist n<0, so werden n Zellen wieder restauriert. Der Nutzer ist dafür verantwortlich, daß
- _bsav abwechselnd mit positivem und negativem Argument gerufen wird und
- in zwei zusammengehörigen Rufen die Argumente gleiche Absolutwerte haben.

Diese Funktion braucht der Nutzer nur zu kennen, wenn er eigene Routinen _p,_s,... schreibt und diese Routinen sich gegenseitig bzw. _bdos rufen. Zu retten sind:
- beim Ruf von bdos 4 Zellen,
- beim Ruf von _p,_s,... n+1 Zellen, wenn das übergebene Feld u[] aus n Elementen besteht.
  (Bemerkung: Falls bei einer Fehlermeldung 'printf' und sofort danach 'exit' gerufen werden, braucht nichts gerettet zu werden.)

*** rand

Syntax: long rand()
  extern long seed;

Wirkung: Erzeugen von Zufallszahlen. Durch Zuweisung eines beliebigen Wertes an 'seed' wird der Startwert der Folge beeinflußt.
Die Zufallszahlen liegen zwischen 0 und (2^31)-1.

*** sleep

Syntax: int sleep(int n)

Wirkung: Anhalten des Programms für n Sekunden.
Bemerkung: Durch einfache Veränderung des Programms kann auch ein Anhalten für n*0.1 Sekunden erreicht werden. Dann ist sleep aber nicht mehr UNIX-kompatibel.

*** time

Syntax: long time(int parm)

Wirkung: parm=0: Starten der Uhr, Einstellen der Zeit 0.
parm=1: Rückgabe der Zeit in Sekunden seit dem Start der Uhr. Die Uhr läuft weiter.
parm=2: Anhalten der Uhr (Fortsetzung bei parm=1). Rückgabe der Zeit in Sekunden seit Start der Uhr.
Die Zeit läuft auch während sleep() weiter!
Die Uhr beginnt nach 24 Stunden wieder bei 0.

Kapitel 7: Testprogramme

In diesem Kapitel soll einiges zum Vergleich von Pretty C mit anderen Compilern hinsichtlich der Effektivität gesagt werden.
I.a. verwendet man dazu sog. Benchmarkprogramme, das sind Testprogramme, die möglichst ohne Änderung von verschiedenen Compilern übersetzt und abgearbeitet werden.
Meistens vergleicht man nur die Laufzeit der übersetzten Programme, manchmal auch Compilezeit und Codelänge. Bei Gleitkommarechnungen sollte unbedingt die Genauigkeit betrachtet werden.

Solche Ergebnisse sind zwar wichtig, aber prinzipiell vorsichtig zu interpretieren, denn:

- Jeder Compiler hat Stärken und Schwächen. Bei einem Benchmarktest erzeugt er guten Code und arbeitet viel schneller als andere Compiler, während bei den restlichen Benchmarks hoffnungslos zurückbleibt. Z.B. kann ein schlechter Compiler gute Gleitkommeroutinen benutzen.
  Der Vergleich von Compilern mit nur einem Programm ist also sinnlos.

- Benchmarkprogramme dürfen nicht dem Compiler angepaßt werden. Oft wird gerade das stillschweigend getan.

- Die Compilezeit hängt normalerweise wesentlich von der Peripherie ab.

- Die Codelänge bezieht oft "printf" ein. Guter Service bringt hier scheinbar schlechtere Ergebnisse.

Warnungen dieser Art gelten auch für die folgenden Betrachtungen.

Allgemeines

Die Schwerpunkte bei Pretty C sind Benutzerfreundlichkeit, Flexibilität, Sprachumfang und nicht zuletzt effektive Nutzung des Speichers, insbesondere der sehr kompakte generierte Code. Die Laufzeit der Programme leidet etwas darunter, denn schnelle Programme werden oft lang oder erfordern große, optimierende Compiler.

Zum Code: Das übersetzte Programm ist i.a. kürzer als der Quelltext, auch wenn letzterer unkommentiert und "dicht" ist. Das erscheint vielleicht lang, aber z.B.

```
i = n++;
```

ist schnell geschrieben, erzeugt aber mindestens folgende Operationen (Speicherklasse "auto" vorausgesetzt):

1. Berechne die Adresse von n.
2. Berechne die Adresse von i.
3. Kopiere den Inhalt der n-Adresse zur i-Adresse.
4. Erhöhe den Inhalt der n-Adresse.

Pretty C benötigt dazu 20 Bytes Code.

Zur Geschwindigkeit: Wird nicht mit Gleitkommazahlen gearbeitet, so ist Pretty C in den bisher betrachteten Fällen etwa 30 mal schneller als BASIC, bei Gleitkommaberechnungen 3...5 mal schneller und wenigstens 3 mal genauer (einfache Genauigkeit).
Im Vergleich zu anderen CP/M-C-Compilern ist Pretty C 2-3 mal langsamer. Zwar sind seine Laufzeitroutinen sorgfältig programmiert, und auch der generierte Code ist teilweise optimiert (vgl. B.3), aber durch seine Kleinheit und den Zwang zum Einpaßcompiler sind Pretty C starke Grenzen gesetzt.
Die unten erwähnten Compiler von Aztec und Whitesmith beanspruchen z.B. 56k bzw. 60k Speicher und 134k bzw. 260k auf Diskette!

Nicht zu vergessen ist, daß viele CP/M-Compiler durch Weglassen von long- und Gleitkommetypen direkt mit der Registervariablen arbeiten können, während das bei Pretty C viel schwieriger wäre.

Zur Codelänge: Verglichen mit BDS- und Supersoft-C-Compiler für CP/M erzeugt Pretty C je nach Benchmark etwas längeren oder kürzeren Code (vgl. Benchmark 1 und 2).

Alle Benchmarks wurden in der Option /I-/BP:0 übersetzt.

Benchmark 1: SIEVE

Das ist sicher das bekannteste Benchmarkprogramm. Es berechnet zehnmal alle Primzahlen kleiner als 16384 (2^15) mit dem Sieb des Erastothenes und ist eine gute Mischung aus Feldberechnungen und Schleifen:

```
/* Sieve-Benchmark */

#define SIZE      8190
#define NTIMES      10
#define TRUE         1
#define FALSE        0

char flags[SIZE+1];

main()
  {int i,prime,k,count,iter;

   for(iter=1, iter<=NTIMES; iter++) {

      count=0;
      for(i=0;i<=SIZE;i++) flags[i]=TRUE;
      for(i=0;i<=SIZE;i++) {
        if(flags[i]) {prime=i+i+3;
          for(k=i+prime; k<=SIZE; k+=prime)
              flags[k]=FALSE;
              count++;
        }
      }
   }
      printf("%d primes found\7",count);
}
```

Quelle: [2]
Laufzeit: 134 sec. Codelänge: 299 Bytes (ohne printf-Zeile)

andere Compiler:
  8-Bit-Rechner unter CP/M:
    BDS V1.5: 64 sec. Codelänge: 239 Bytes
    Supersoft C: 55 sec. Codelänge: 356 Bytes
    Withesmith 2.1: 48 sec.
    Telcom 2.7: 70 sec.
    (Quelle: [5]/eigener Test)
  16-Bit-Rechner (z.B. Lattice C und Manx Aztec C86K für Amiga,
    Aztec C für Macintosh, Decus C für PDP-11):
    zwischen 2,8 und 11 sec.

Zum Vergleich: BASIC am KC 85/1: 2030 sec.

Benchmark 2: FIB

Das 24. Element der durch

$X_0 = 1,\ x_1 = 1,\ x_{n+2} = x_{n+1} + x_n\ (n=0,1,...)$

definierten Fibonacci-Zahlenfolge wird rekursiv berechnet. Auch dieser Benchmark ist sehr populär und testet die Zeit für Funktionsaufruf und Parametervermittlung.

```
/* Fibonacci-Benchmark */

#define NTIMES 10

main()
  {
   int i; unsigned value,fib();

   printf("\nStart Fib");

   for (i=1; i<=NTIMES; i++) value=fib(24);

   printf("ended after %d iterations\7",NTIMES);
   printf("\n value: %u",value);

  }

  unsigned fib(x)

     int x;
     {
      if (x>2 return fib(x-1)+fib(x-2);
        else return 1;
     }
```

Quelle:[2]
Laufzeit: 672 sec. Codelänge: 175 Bytes

Andere Compiler:
  8-Bit-Rechner unter CP/M:
    BDS 1.5 : 160 sec Codelänge: 127 Bytes
    Supersoft C: 100 sec Codelänge: 81 Bytes
    Withesmith 2.1: 123 sec
    Aztec V1.05G: 190 sec
    (Quelle: [5]/eigener Test)
  16-Bit-Rechner (s.o.): 29-47 sec.

Benchmark 3: STRLNG

Die Länge eines Strings wird mit einem C-Programm ermittelt.

```
/* strlng - Benchmark */

#define NTIMES 25000

main()
  {char *p; int n;

   n=NTIMES;
   p =
   "Now is the time for all good men to come to the aid of their party";

   while (n--) strlng(p);

   printf("\nfinished\7");
  }
```

```
strlng(s)
  char *s;
  {
   char *p;
   for(p=s; *s!='\0'; s++);
   return(s-p);
  }
```

```
Quelle:[5]
Laufzeit: 770 sec.
andere Compiler:
  8-Bit-Rechner unter CP/M:
    Withesmith 2.1:     110 sec
    BDS 1.5 :           158 sec
    C/80 2.0:           286 sec
    (Quelle: [5])
```

Bemerkung: Ersetzt man in der 'for'-Schleife in strlng() den Ausdruck *s!='\0' durch den äquivalenten *s, so reduziert sich die Laufzeit auf 306 sec. für Pretty C. Schreibt man statt der 'for'-Schleife

```
  p=s-1; while(*++s);
```

so läuft das Programm nur noch 223 sec. Mit der Funktion strlen() aus der Bibliothek CEXT läuft es nur noch 34 sec.

Benchmark 4: SAVAGE oder FLOAT

Das ist eigentlich ein Test für die Standardfunktionen aus der Bibliothek auf Geschwindigkeit und Genauigkeit:

```
main()
  {
   double a,sqrt(),log(),exp(),tan(),arctan();
   int i;

   a=1.;
   for (i=1; i<2500; i++)
     a = tan( arctan( exp( log( sqrt(a*a))))) + 1.;
   printf("\ndone - error: $le\7",(a-2500.)/2500.);
  }
```

```
Quelle:[7]
einfache Genauigkeit (zu Beginn einfügen:
#define double float
und mit /F+ übersetzen):
  Laufzeit:       sec.  Fehler
doppelte Genauigkeit:
  Laufzeit:       sec.  Fehler

andere Rechner:
  8-Bit-Rechner, CP/M:
    PC 1715, BASIC:          180 sec., Fehler -1 (!) (7 Stellen)
    KC 85/1, BASIC:          330 sec., Fehler -0,6   (7 Stellen)
    Commodore C64, BASIC:    514 sec., Fehler ca. 1.e-2 (1%)
    ZX 81, BASIC:            990 sec., Fehler -0.3
  16-Bit-Rechner:
    Macintosh. Aztec C       268 sec.
    IBM PC, DeSmet C:        244 sec., Fehler 5.e-4
            Aztec C:         353 sec.
            Turbo Pascal:    544 sec., 3.e-10       (11 Stellen)
            Assembler und Coprozessor 8087:
                             2.2 sec., 3.e-10       (12 Stellen)
  größere Rechner:
    SM 1420, FORTRAN:        5/8 sec., 0.1/7.e-11   (einfach/doppelt genau)
    VAX 11/780, C:             2 sec., 7.e-10       (11 Stellen)
    Cray 1S, FORTRAN:          3 ms,   7.e-10
    Cray-XMP, ?:             120 µs,   2.e-10
```

Kapitel 8: Der Debugger

8.0 Aufgabe und Arbeitsweise

Wenn der erste Teil einer Programmentwicklung abgeschlossen ist - das Entwerfen und Niederschreiben des Programms -, beginnt die Fehlersuche, oft der Hauptteil der Arbeit.
Die Beseitigung von Syntaxfehlern ist im Dialog kein Problem mehr und unter Pretty C besonders bequem. Die Probleme zeigen sich erst, wenn das Programm aus "absolut unerfindlichen Gründen" abstürzt oder wenigstens unsinnige Ergebnisse liefert.

Der Nachteil compilierter Programme ist, daß sie keine Bindung zum Quelltext mehr haben (wie etwa in BASIC, wo man ja jederzeit unterbrechen und alle Variablen unter ihrem Namen ansprechen kann). Eine gewisse Hilfe sind das backtrace-Listing (vgl.3.1.1) und der Schalter /EL in Pretty C. In schwierigeren (nach McMurphy also üblichen) Fällen werden oft nur "printf"-Anweisungen solange eingefügt (und übersetzt, und gestestet, und neu editiert), bis ein Fehler gefunden wurde.

Die großen Nachteile dieser Methode, die nur bei einfachen Anwendungen zu vertreten ist, sind offensichtlich. Hier hilft ein Debugger sehr viel Zeit und Mühe sparen. Dieses Werkzeug ist so wichtig,

- daß Sie unbedingt mit ihm vertraut sein sollten! -

Damit können Sie nämlich ihr Programm scheinbar interpretierend abarbeiten, Schritt für Schritt oder auch Anweisung für Anweisung seine Arbeit verfolgen und nötigenfalls eingreifen.

Der Pretty C - Debugger ist ein C-Programm (Quelltextfile DEBUG auf dem mitgelieferten Band), das über die Debuggerschnittstelle gerufen wird s.3.3.5). Damit können Sie den Debugger auch nach Wunsch verändern.
Die Rufe dieser Schnittstelle werden automatisch nach der Berechnung jedes Ausdrucks, jedes Arguments in einem Funktionsaufruf und jedes nicht konstanten Feldindex eingefügt.

- wenn Sie beim Übersetzen den Schalter /BP:2 setzen. -

(Damit ist klar, daß der Debugger selbst nie mit /BP:2 übersetzt werden kann, und CLIB auch nicht, da er diese Bibliothek benutzt.)
Der Debugger übernimmt dann die Steuerung Ihres Programms, sobald ein mit /BP:2 übersetzter Programmteil abgearbeitet wird.

Sie können u.a.:
- das Programm schrittweise abarbeiten oder abbrechen;
- Variablen abfragen und verändern und das entsprechende Format dabei selbst vergeben;
- einen Haltepunkt setzen, an dem der Debugger erst wieder aktiv werden soll.

Sie sehen einige Zeilen des Quelltextes auf dem Bildschirm; der gerade berechnete Ausdruck ist markiert.

8.1 Einbinden in ein Programm

Es folgen nur wenige Regeln, sie sind aber auf jeden Fall gewissenhaft zu beachten! Eine Absicherung von Bedienfehlern ist nicht möglich!

1. Folgende Funktionsnamen dürfen in Ihrem Programm nicht verwendet werden: debug, _deblin, _debc, _debcr, _debval, _debcmd, _debmv.

2. CLIB und DEBUG werden benötigt und dürfen nicht an die zu debuggenden Teile angehängt werden. Fehlen Gleitkommazahlen im Programm, kann auch CLIBM statt CLIB eingebunden werde.

3. Die Debug-Teile (die mit /BP:2 übersetzt wurden) müssen in einem Textpuffer stehen, der auch zur Laufzeit im Speicher verbleibt.

4. Übersetzen Sie den ersten Debug-Teil des Programms mit den Schaltern

   /AS:n00H/BP:2

   n sollte dabei nicht zu klein sein (n=1 nur bei kurzen Programmen). Pro automatisch eingefügtem Debuggerruf werden 4 Bytes benötigt.
   Kontrollieren Sie nach der Übersetzung mit CC /DS, ob Sie zuviel Platz reserviert haben.

5. Übersetzen Sie alle folgenden Debug-Teile mit
   /BP:2/CN

6. Übersetzen Sie die ev. restlichen Programmteile, CLIB(M) und DEBUG mit
   /BP:0/CN
   Die Files können auch (aneinandergehängt) mit /TP/BP:0/CN übersetzt werden.

Damit ist der Debugger eingebunden und wird automatisch nach dem Programmstart aktiviert, s.8.2 .
Die Reihenfolge der Übersetzungen kann auch verändert werden, jedoch ist der Schalter /AS stets beim ersten Compilerlauf (und nur dort!) anzugeben, und nur Debug-Teile dürfen mit /BP:2 übersetzt werden.

Sollen Variablen angezeigt und verändert werden, so ist zusätzlich zu /BP:2 noch der Schalter /LI:2 oder /LI:5 zu setzen, vgl. 8.2, Kommando 'f'.

Achtung! Vergessen Sie bitte niemals zwei Dinge:
- Der zu "debuggende" Quelltext muß in einem Textpuffer stehen, auch zur Laufzeit Ihres Programms (und zwar auf gleicher Adresse wie zur Compilezeit!);
- CLIB und DEBUG müssen mit /BP:0 übersetzt werden.

8.2 Bedienung

Bei der Steuerung Ihres Programms durch den Debugger hat der Bildschirm folgenden Aufbau:

Zeilen 1-8: "Quelltextfenster", in dem 7 Zeilen Ihres Quelltextes angezeigt werden. **Nach** dem Ausdruck, der gerade berechnet bzw. nach dem ein Haltepunkt gesetzt wurde, erscheint ein Sonderzeichen.

Zeile 9: Anforderung eines Kommandos.

folgende Zeilen: Anzeige der Namen, Worte und Ausgabeformate von Ihnen ausgewählter Variabler mit laufenden Nummern 0...9. Dabei ist der Wert Nr.0 der des gerade berechneten Ausdrucks, mit "*** value;" bezeichnet (anfängliches Ausgabeformat %d).

darunter: Ein- und Ausgaben Ihres Programms in einem rollenden Fenster.

Nach 'GO' wird der Debugger sofort initialisiert. Es erscheint "debug-init" in der letzten Zeile, und der Stop-Mode (s.u.) wird eingestellt. Nun beginnt Ihr Programm wie gewohnt zu arbeiten, bis der erste Haltepunkt erreicht wird. Der Bildschirm wird, wie oben beschrieben, aufgebaut und ein Kommando erwartet.

Sie bestimmen bei jedem Debug-Lauf selbst, welche Variablen angezeigt werden sollen (max. 9). Der Debugger benötigt dazu Namen und zugehörige Adressen (absolut oder offset, je nach Speicherklasse). Dazu müssen Sie das bei /LI:2 bzw. /LI:3 erzeugte Listing auswerten. Näheres beim 'f'-Kommando.

Kommandos

Ein Kommando ist ein Tastendruck, auf den der Debugger sofort reagiert. Es ist gleichgültig, ob Groß- oder Kleinbuchstaben eingegeben werden.
Illegale Kommandos werden mit '???' quittiert. Die Syntax von Formaten, Adressen usw. wird **nicht** geprüft!

Im folgenden beziehen sich "Ausdruck" und "Quelltext" immer auf mit /BP:2 übersetzte Teile.

Folgende Kommentare sind zulässig:

a - 'animation': Wie bei 'step' wird nach jedem Ausdruck angehalten, aber 1-2 Sekunden. Das Programm läuft also wie ein Trickfilm ab.
Mit einer anderen Taste, z.B. "b", wird das Programm angehalten und diese Taste als Kommando interpretiert.

b - 'breakpoint': Auswahl eines Haltepunktes. Nach "b" wird die Bewegungs-richtung der Veränderung angezeigt (+). Folgende Kommandos werden nun erwartet:

+ Änderung der Richtung in "vorwärts"
- Änderung der Richtung in "rückwärts"
1...9 In der festgelegten Richtung wird der Haltepunkt entsprechend 1...9 Ausdrücke weitergesetzt.
ENTER Der Haltepunkt wird gewählt. Ihr Programm arbeitet nun so lange weiter, bis es den mit dem Sonderzeichen markierten Ausdruck erreicht.
ESC Das Kommando 'b' wird verworfen und ein neues erwartet.
STOP Programmabbruch.

c - 'changed format': Das Format für die Anzeige und Eingabe einer Variablen bzw. des berechneten Ausdrucks wird verändert. Debug erwartet die Eingabe einer Ziffer (Nummer eines der Werte, aus der angezeigten Liste). Das Format wird wie in printf aufgefaßt, nur ohne vorgesetztes '%'. Es darf nur aus einem Buchstaben bestehen, ev. mit Zusatz 'I', d.h.: c,d,u,o,x,e,f,g oder ld,lu,lo,lx,le,lf,lg.

e oder STOP -
'exit': Programmabbruch.

f - 'new format': Die Liste der anzuzeigenden Variablen wird um 1 erweitert. Der Debugger fordert eine Eingabe der Form name:addr:format.
Dabei ist 'name' der Variablenname, 'addr' die dem /LI:2-Listing entnommene Adresse - auch mit ev. '$'! - und 'format' eine Formatangabe wie beim 'c'-Kommando.
Sind bereits 9 Variablen vorgegeben, ist das Kommando illegal.

g - 'go': Der Haltepunkt wird gelöscht. Das Programm arbeitet ungünstigstenfalls bis zu 25mal langsamer weiter.

c oder ENTER -
'step': Schrittbetrieb; nach dem nächsten berechneten Ausdruck hält Ihr Programm wieder an.

w - 'write': Veränderung eines Variablenwertes oder des Wertes des berechneten Ausdrucks.
Zunächst ist die Nummer des angezeigten Wertes einzugeben (0 entspricht dem berechneten Ausdruck), dann der neue Wert gemäß dem angegebenen Format.
Der Bildschirm wird nun aktualisiert und ein weiteres Kommando erwartet.
Achtung!
Der Debugger hält an, **bevor** der berechnete Ausdruck weiterverwendet wird (Ausnahme: konstante Feldindizes). Damit kann also auch der logische Wert eines Ausdrucks in Testbedingungen (wie 'if(expr)') verändert, d.h., der Programmablauf geändert werden.

Nach den Kommandos 'c', 'f' und 'w' erwartet Debug stets ein weiteres Kommando.

8.3 Zum Debugger-Quelltext

In DEBUG sind einige Macros definiert, die Sie bei Wunsch verändern können:

LINES Anzahl der angezeigten Quelltextzeilen.

GRAPH Der Code des Grafikzeichens, das den berechneten Ausdruck markiert.

MAXSW max. Anzahl anzuzeigender Variablen + 1 (sollte nur verkleinert werden).

MAXDS max. Länge eines beim 'f'-Kommando eingegebenen Formats.

Der Cursor wird in den Funktionen debug() und _deber() umgesetzt; die Kommandoeingabe erfolgt in _debcmd. Diese Funktion gibt 1 zurück, wenn ein weiteres Kommando erwartet wird, sonst 0.
Die Informationen über Variablennamen, Adressen und Formate im globalen Feld v aufbewahrt, die Anzahl der angezeigten Werte in der globalen Variablen actsw.

Kapitel 9: Das Installierungsprogramm INSTALL

Auf dem mitgelieferten Magnetband befinden sich 4 Files:

INSTALL - Der Masterfile
CLIB - und
CEXT - die Bibliotheken (vgl. Kap.6) sowie
DEBUG - der Quelltext des Debuggers (vgl. Kap.8).

Mit dem Masterfile können Sie den Compiler installieren, d.h., einige Parameter nach Ihren Wünschen festlegen und den so erzeugten Compiler auf Band speichern.

Die Bedienung ist wie folgt:

1. Einlesen des Files INSTALL; das Programm startet von selbst.

2. Beantworten der Anfragen. Davon gibt es zwei Arten:

   - Eingabe einer hexadezimalen Adresse (3- oder 4-stellig). Der Standardwert ist in eckigen Klammern angegeben
     Die Eingabe erfolgt wie im Monitor üblich (Korrektur durch "Cursor links") und ist mit ENTER abzuschließen. Illegale Werte führen zur Fehlermeldung und Programmabbruch.
     Der gelesene Wert wird zur Kontrolle nochmals ausgegeben.

   - Entscheidungsfragen. Sie sind mit 'Y' (yes) oder 'N' (no) zu beantworten. Die Bedeutung der Antworten ist in der Frage angegeben. Illegale Eingabezeichen werden ignoriert.

   Bei beiden Eingabeformen gilt:

   - Wird nur ENTER gedrückt, so wird der Standardwert bzw. 'Y' angenommen.
   - STOP bricht das Programm ab.

   Die Standardversion können Sie also installieren, indem Sie einfach immer nur ENTER drücken.

3. Auslagern auf Band. Nach Beendigung der Initialisierung können Sie Pretty C

   *** unter dem Namen PRETTYC ***

   auf Band so oft schreiben, bis Sie mit STOP abbrechen (Bedienung im Dialog).

Erläuterung der Anfragen:

Adreßeingabe:

RAMTP: Das ist die Obergrenze des RAM-Speichers, über den Pretty C verfügen darf. Die Speicherzellen hinter RAMTP sind logisch geschützt.
Sie können die physische Speichergrenze angeben:
32k = 1 RAM-Modul: 7FFFH, 48k = 2 RAM-Module = Standard: BFFFH,
aber auch einen kleineren Wert, um dahinter Maschinencodeprogramme unterzubringen. RAMTP muß die Form nFF haben.

Standardadresse für Textpuffer:
Auf dieser Adresse wird der erste Textpuffer durch den Schalter /CR mit EC erzeugt (vgl. 2.1.1).
Beachten Sie, daß diese Adresse kleiner als RAMTP ist, da Sie sonst ständig Fehlermeldungen erhalten.

Der Wert muß die Form n00 haben.

Konstantenpuffer:
Alle Konstanten (auch Strings) und Funktionsnamen werden in einen Puffer geschrieben (vgl. B.2). Diese Anfrage legt die Standardgröße für den Puffer fest (400 entspricht 1KBytes und reicht für viele Zwecke aus).
Der Wert muß die Form n00 haben.

Bei allen drei Anfragen wird nur geprüft, ob das niederwertige Byte 'FF' bzw. '00' ist. Seien Sie also vorsichtig!

Entscheidungsfragen:

24/20 Zeilen:
EC schaltet während seiner Arbeit den Bildschirm auf die vorgegebene Zeilenzahl um. Nach Verlassen eines Teilprogrammes von Pretty C (vgl. 1.0) wird der Bildschirm **stets** auf den 24-Zeilen-Mode umgeschaltet.

schnelle Tastenfunktionen:
Die in 2.6, letzter Punkt, beschriebene Veränderung der Tastenwiederholfrequenz kann durch 'N' unterdrückt werden; die entsprechenden Bemerkungen in 2.6 entfallen dann.

Einrücken von 4/2 Positionen:
Das in 3.0.1 unter "n-1:" erwähnte strukturierte Listing erfolgt mit Einrücken von 4 bzw. 2 Cursorpositionen je Blockniveau.

Anhang A: Die realisierte Sprache

Dieser Anhang enthält KEINE C-Sprachbeschreibung. Es werden nur (in komprimierter Form) alle Besonderheiten der von Pretty C verstandenen Sprache aufgeführt, die in Kernighan/Ritchie [1] (im folgenden "K&R" genannt) anders oder nicht festgelegt sind.
Der Leser kann sich also darauf verlassen, daß er hier nicht erwähnte Sprachelemente entsprechend K&R verwenden darf.

Wichtige Änderungen gegenüber K&R sind mit '***' markiert, implementationsabhängige Details mit 'ooo'.

Wie in K&R gilt hier folgender Sprachgebrauch: Eine Definition reserviert Speicherplatz für ein Objekt (und legt den Typ fest), eine Deklaration hingegen bestimmt nur den Typ.
Beispiel: int n; = Definition
           extern n; = Deklaration

Zeichen

In Zeichenketten dürfen alle Zeichen mit ASCII-Code größer oder gleich 32 vorkommen, auch Grafikzeichen; Kommentare dürfen selbstverständlich auch ENTER (13) enthalten, da sie über mehrere Zeilen gehen können (vgl. 2.6, Aufbau der Textpuffer).
Da Auftreten eines Grafik- oder Steuerzeichens - außer ETX (03) am Zeilenanfang - führt in allen anderen Fällen zur Fehlermeldung.

Namen

Namen können theoretisch beliebig lang sein (beschränkt durch EC auf 255 Zeichen, bei Übersetzung mit /TP auf 64 Zeichen. Alle Zeichen sind signifikant.

Schlüsselwörter

Nicht implementiert sind 'enum' und 'entry', dafür gibt es 'void', vgl. Abschnitt "Grundtypen".

Konstanten

Zeichenkonstanten:

Alle Nichtsteuerzeichen sind zulässig. Mit dem "backslash" '\' können folgende Zeichen dargestellt werden (ASCII-Code in Klammern):

```
\n     newline          = ^J  (10)
\r     return           = ^M  (13)
\b     backspace        = ^H  (08)
\f     formfeed         = ^L  (12)
\t     tabulator        = ^T  (09)
\'     Apostroph             (39)
\" Anführungsstriche         (34)
\\ backslash            (92)
```

Folgen nach '\' ein, zwei oder drei Oktalziffern: \ddd , so wird das Zeichen mit dem oktalen Code ddd eingesetzt.

Folgt nach '\' ein anderes Zeichen, werden dieses und '\' ignoriert.

Integerkonstanten:

ooo Alle Integerkonstanten, auch oktale und hexadezimale, sind vom Typ int bzw. long.
0xFFFF wird als -1 verstanden,
0x10000 als 65537.
*** '8' und '9' werden nicht als Oktalziffern verstanden.

Gleitkommakonstanten:

*** Der Dezimalpunkt darf nie fehlen.

ooo Es dürfen
*** höchstens 18 Stellen ***
angegeben werden, sonst erfolgt Fehlermeldung.

*** In Erweiterung der K&R-Sprache wird definiert:

Ist der Exponentialteil mit 'e' oder 'E' bezeichnet, so hat die Konstante den Typ 'double';
Ist der Exponentialteil mit 'f' oder 'F' bezeichnet, so hat die Konstante den Typ 'float';
fehlt der Exponentialteil, so wird der Typ der Konstanten durch den Compilerschalter /F+ bzw. /F- festgelegt.

Strings:

Es ist die gleiche Menge von Zeichen zulässig wie bei Zeichenkonstanten.

*** In vorliegender Version gibt es noch keine Initialisierungen, daher sind Strings char-Felder im Konstantnpuffer. Eine ev. Modifikation von Feldelementen ist möglich, aber permanent, d.h., sie bleibt über einen Programmlauf hinaus bestehen.
Es ist dringend anzuraten, diese Besonderheit von Pretty C **nicht** zu nutzen.

Grundtypen, Wertebereiche

Der Typ 'void' ist vorhanden und darf wie alle anderen Grundtypen verwendet werden, jedoch führt die Verwendung von Objekten vom Typ 'void' in einem (nicht nur aus dem Objekt selbst bestehenden) Ausdruck zur Fehlermeldung. Zeiger auf den Typ 'void' sind zulässig und dürfen konvertiert werden.

Folgende Datenbreiten gelten in Pretty C:

1 Byte = 8 Bits

```
char:       1 Byte, Wertebereich 0...255 (also nie negativ)
short,int:  2 Bytes, Wertebereiche:
              -32768...32767 bei int,
              0...65535 bei unsigned.
long,float: 4 Bytes, Wertebereiche:
              -2 147 483 648 ... 2 147 483 647 bei long,
              0 ... 4 294 967 295 bei unsigned long,
              etwa ±2.9e-39...±1.76e+38 und 0 bei float
              (etwa 7 signifikante Stellen).
double:     8 Bytes, Wertebereich wie bei 'float'; etwa 17 signifikante
            Stellen.
```

Die Adressen der Daten genügen keinen Einschränkungen bzgl. Wortgrenzen.

Deklarationen, abgeleitete Typen

*** Bitfelder gibt es nicht.

*** Deklarationen der Form
extern a[];
sind verboten.

*** 'typedef', darf nicht während der Deklaration von Funktionsargumenten vereinbart werden (sondern vorher).

ooo 'short', 'long' und 'unsigned' werden wie Attribute behandelt, z.B. sind 'unsigned long int' und 'long unsigned' gleichwertig. Weiter sind äquivalent:
'short int', 'short' und 'int';
'unsigned char' und 'char';
'long float' und 'double'.

- Strukturdeklarationen dürfen keine 'typedef'-Anweisung oder weitere Strukturdeklarationen enthalten, wie etwa in

```
struct {int a;    /* UNZULÄSSIG */
        struct {int x,y;} sx;
        int b;
       }
        x;
```

- Funktionsdeklarationen und -definitionen werden wie folgt unterschieden:

- Wenn die Deklaration mehr als einen Namen enthält, so muß es sich um eine Funktionsdefinition (mit Argumenten) handeln.
- Wenn auf eine Funktionsdeklaration kein ';' oder ',' folgt, so wird sie als Definition aufgefaßt.
  In beiden Fällen werden ev. Deklarationen von Argumenten und danach der Funktionsblock erwartet.
- Andernfalls ist es eine Funktionsdeklaration.
  Funktionen dürfen nur Werte von Grundtypen bzw. Pointer zurückgeben.

Lexikalischer Gültigkeitsbereich, Speicherklassen

Folgende Unterschiede zu K&R sind zu beachten:

Register

- Registervariablen dürfen vereinbart werden, sind jedoch mit 'auto'-Variablen völlig identisch. Der Nutzer hat selbst darauf zu achten, daß keine unsinnigen (und daher nicht portable) Konstruktionen wie "register float f;", "register a[3];" oder "register i; p=&i;" vorkommen.
  Als dynamische Variable führen Registervariable in Pretty C zu **langsameren** Programmen als mit statischen Variablen; ihre Verwendung ist nur aus Gründen der Portabilität zugelassen.

Funktionen und Variable

*** Funktionen sind immer 'extern', eine ev. Angabe der Speicherklasse wird ignoriert!
Insbesondere gibt es keine 'static'-Funktionen.

- Alle Objekte außer Funktionen dürfen pro File höchstens einmal auf gleicher Ebene deklariert werden (also z.B. keine Variable zweimal auf globalem Niveau).
  Zulässig ist aber z.B.
  f() {extern n; ... {extern;...}} ... int n;

- Die Namen von Funktionen sowie von Objekten, die nicht mit 'static' auf globaler Ebene oder aber als 'extern' deklariert wurden (im folgenden "externe Namen" genannt), werden zusammen mit Informationen zum Typ in eine spezielle Liste eingetragen. Für den Rest dieses Quelltextpuffers/-files sowie während aller mit /CN angehängten Compilerläufe kann man sich extern auf diese Namen beziehen.
  Pretty C prüft, daß diese Namen höchstens einmal definiert werden.
  Nicht definierte, aber deklarierte externe Namen werden bei "CC /DS" angezeigt.

Marken

*** Im Unterschied zu K&R sind Marken nur im kleinsten umfassenden Block gültig. Markennamen dürfen nicht mit Variablen- oder Funktionsnamen, die in einem umfassenden Block definiert wurden, übereinstimmen.
Insbesondere kann man zwar mit 'goto' aus einem Block heraus-, aber niemals hineinspringen.

Strukturen

ooo Die Namen von Exponenten sind nur innerhalb einer Struktur sichtbar und können beliebig an anderer Stelle verwendet werden.

Typvergleiche

Wird ein externer Name mehrfach deklariert (ev. auch in verschiedenen Files oder Textpuffern), so prüft Pretty C auf Übereinstimmung der Typen.

ooo Auch abgeleitete Typen müssen dabei völlig identisch sein, mit einer Ausnahme: Strukturen (und Unions) gleicher Größe gelten als gleich.
Beispiele:

```
  Zulässig -
    Puffer 1:
      extern struct {int a[3];} s[2];
    Puffer 2:
      extern struct {int a,b,c;} s[2];
  Unzulässig -
    Puffer 1:
      extern a[2][2];
    Puffer 2:
      extern a[3][2];
```

Konvertierungen

*** Variablen vom Typ 'float' werden in Pretty C **NICHT** automatisch auf 'double' konvertiert, insbesondere nicht bei Funktionsrufen. Eine Konvertierung auf 'double' erfolgt erst bei binärer Verknüpfung mit einem Operanden vom Typ 'double'.

ooo Variablen vom Typ 'char' haben nie negative Werte, ergeben also bei Konvertierung auf 'int' Werte von 0 bis 255.

ooo Liegt der Wert einer Gleitkommazahl
- bei Konvertierung auf 'unsigned long' nicht zwischen
  -(2^32-1) und (2^32-1) oder
- bei Konvertierung auf 'long' nicht zwischen
  -2^31 und (2^31-1),

so erfolgt eine Fehlermeldung.
Bei Konvertierung auf 'unsigned long' wird zu negativen Zahlen 2^32 addiert.
Bei Konvertierung auf 'int' bzw. 'unsigned' wird zunächst auf 'long' bzw. 'long unsigned' konvertiert und dann abgeschnitten.

ooo Werden 'unsigned long' oder 'long'-Größen auf 'float' konvertiert, so wird zunächst auf 'double' konvertiert und dann auf 'float' gerundet.

ooo Funktionsnamen werden, falls nach ihnen nicht '(' folgt, automatisch in einen Pointer umgewandelt. Das ist nur bei Anweisung und bei Parameterübergabe in Funktiosrufen erlaubt.
Beispiel:
```
int n,g(),{*f)(); f=g; n=(*f)();
```

Unäre Operatoren

ooo Der Adreßoperator '&' darf nicht auf Funktionsnamen angewendet werden.
Vgl. hierzu Abschnitt "Konvertierungen".

ooo 'sizeof' darf beliebige Typdeklarationen enthalten, auch Strukturdeklarationen, aber keine Speicherklasse.

*** Nach 'sizeof' muß stets ein geklammerter Ausdruck bzw. Typ folgen.

*** Inkrement- und Dekrementoperatoren dürfen nur auf integrale Typen angewendet werden.

ooo Bei 'int'- und 'long'-Variablen wird dabei auf Überlauf geprüft.

Binäre Operatoren

ooo Bei den arithmetischen Operatoren +,-,*,/ und % wird auf ev. Division durch 0 und Überlauf geprüft.

Subtraktion: ooo Es dürfen nur Pointer gleichen Typs subtrahiert werden. Felder und Pointer sind verschiedene Typen! Daher muß es heißen
```
int *p,a[13];
p=a; ... = p - &a[0];
```
und **nicht**
```
... = p-a;
```
(Fehler "type mismatch").

Division /: ooo Bei ganzzahligen Werten ist das Ergebnis genau dann negativ, wenn beide Operanden verschiedene Vorzeichen haben.

Divisionsrest %:
ooo Das Vorzeichen von a%b ist das von a.

Verschiebeoperationen <<,>>:
ooo Es wird stets nur logisch verschoben (Auffüllen mit Bit 0). Der rechte Operand muß vom Typ char, int oder unsigned sein, der linke integral.
Ist der Wert des rechten Operanden größer als die Bitbreite des linken, ist das Ergebnis undefiniert.

Bedingte Operation '?:'
ooo Der zweite und dritte Operand (also b und c in a?b:c) müssen gleichen Typ haben, mit Ausnahme der Kombinationen
int - unsigned und
long - unsigned long.
Notfalls sind cast-Operatoren anzuwenden.

Zuweisungsoperationen:
ooo Mit Ausnahme von '=' müssen beide Operanden den gleichen Typ haben, bis auf die beiden unter '?:' erwähnten Kombinationen.

Konstante Ausdrücke

*** Konstante Ausdrücke dürfen nicht den bedingten Operator '?:' enthalten.

Initialisierungen

*** Definitionen mit Initialisierungen sind in vorliegender Version nicht möglich.
Lediglich eine automatische Initialisierung von statischen Variablen mit 0 wird durchgeführt.

Präprozessor

*** Der Präprozessor versteht nur die Kommandos
#define und
#undef

*** Macros mit Argumenten werden von Pretty C nicht verstanden.

*** Eine Fortsetzung von Zeilen mittels '\' ist nicht möglich.

ooo Auf bereits substituierte Namen können keine weiteren Macros angewendet werden.

Anhang B: Zum Compiler selbst

B.0 Portabilität

Pretty C ist in U880-Assemblercode geschrieben und benutzt nur die vom Hersteller dokumentierten Befehle des Mikroprozessors. Weder Pretty c noch die übersetzten Programme sind selbstverändernd.
Es werden sämtliche Register benutzt, auch Schattenregister. Fast alle Systemrufe erfolgen n Form von BDOS-Rufen über eine einheitliche Schnittstelle.

Abweichungen gegenüber CP/M sind:

1. Zeichenketten, die mit dem BDOS-Ruf Nr.9 ausgegeben werden, sind mit einem Nullbyte abgeschlossen (das wird selten benutzt).

2. Die Magnetbandarbeit erfolgt entsprechend der CP/M-Modifizierung am KC 85/1.

3. In vorliegender Version greift der Editor noch direkt auf den Bildwiederholspeicher zu, setzt also dessen Existenz voraus. Es muß möglich sein, die aktuelle Cursorposition zu erfragen.

4. Des weiteren wird benutzt, daß die Systemrufe am KC 85/1 alle Register retten, mit Ausnahme zurückgelieferter Parameter und des Carryflags.

5. Bis auf RST 0 werden **alle** RST-Adressen benötigt (diese Adressen sind in manchen CP/M-Implementierungen nicht frei).

Bis auf 3. und ev. 5. bereitet die Übertragung von Pretty C auf CP/M-Rechner also keine wesentlichen Probleme. Auf die RST-Befehle kann allerdings in keinem Fall verzichtet werden.

B.1 Listen

Pretty C arbeitet mit folgenden Listen:

1. Liste der Funktionsnamen

2. Liste der Konstanten

3. Liste der externen Namen

4. Liste der Macrodefinitionen

5. Liste der deklarierten Namen

6. Liste der deklarierten struct- und union-Typen

Dabei sind die Listen 3 und 4 sowie 5 und 6 jeweils in einer Liste zusammengefaßt. Während die Listen 1 und 2 in einem Puffer ("Konstantenpuffer") fester Länge (die durch /CS:n00 bestimmt werden kann) stehen und sich aufeinander zu bewegen, legt Pretty C die beiden anderen Listen hinter den übersetzten Code und verschiebt sie von Zeit zu Zeit (vgl.a.B.2). Damit ist der Programmierer weitgehend davon befreit, für ihn uninteressante Listengrößen festzulegen, und gleichzeitig ist eine gute Auslastung des Speichers garantiert.

Zu 1: Die Liste der Funktionennamen steht auch zur Laufzeit im Speicher und ermöglicht das backtrace-Listing.

Zu 2: Konstanten, außer Strings, werden auf Wiederholung geprüft, auch bei Konstanten Ausdrücken. Daher beanspruchen z.B.
1+3*3; 7-8; 0xA; '0';
nur 4 Bytes Speicherplatz für Konstanten.

Zu 3: Diese Liste enthält zusätzlich noch Informationen zum Typ dieser Namen und Funktionen.

Zu 4: Die Macrodefinitionen bilden eine Teilliste der Liste 3. Wegen der Struktur dieser Liste kann das Löschen von Macrodefinitionen nur logisch, niemals physisch, erfolgen und daher in Extremfällen Speicherprobleme bedingen.

Zu 5: Diese Liste besteht aus einem Stack von Teillisten, der beim Erreichen von Blockgrenzen entsprechend verändert wird. Beim Löschen einer Teilliste werden verwendete, aber noch nicht definierte Marken in die nächsthöhere Liste übernommen.
Außerdem werden alle deklarierten zusammengesetzten Typen rekursiv beschrieben. Pro Rekursion sind 5 Bytes notwendig. 'typedef' für komplizierte Typen kann Speicherplatz sparen helfen, da immer auf den gleichen Pfad zurückgegriffen wird.
Die Deklaration eines Namens der Länge n beansprucht n+8 Bytes.

Zu 6: Diese Teilliste ist ebenfalls aus einem Stack von Teillisten aufgebaut, der analog wie Liste 5 verwaltet wird. Jedes Element verweist wiederum auf eine Teilliste mit den Komponentennamen und deren Typbeschreibungen.

Zu Beginn eines Compilerlaufes mit dem Schalter /CN werden die Listen 4, 5 und 6 initialisiert, sonst alle Listen.

B.2 Speicheraufteilung

Mit '*' bezeichnete Bereiche haben variable Adressen, die sich teilweise noch während der Compile- bzw. Laufzeit ändern.

a) Zur Compilezeit

```
|   90 -   18F    Stack für Syntaxanalyse
|  210 -   2D1    Puffer für /TP-Übersetzung
|  2D2 -   2FF    Zeiger auf Teillisten
|  300 -  4F00    Pretty C mit Arbeitszellen
|  [   *          ev. Adressenpuffer für /AS-Option]
|    ↓ *          Funktionsnamen
|    ↑ *          Konstanten
|      *          generierter Code
|      *          Liste 3/4
|      *          Liste 5/6
↓

↑ -------------- Stackpointer SP
|
|
---------------------------------------
logisches Speicherende (Textpuffer, oder mit /WE definiertes Speicherende,
                        oder RAMTP)
```

b) Zur Laufzeit

```
|  210 -  2FF     Puffer für gets() und vordefinierte Funktionen
|  300 - 4F00     Pretty C mit Arbeitszellen
|  [   *          ev. Adressenpuffer für /AS-Option]
|      *          Funktionen
|      *          Konstanten
|      *          generierter Code
```

```
|      *          Datenstack
|      *          Stack für Zwischenergebnisse
↓

↑ --------------- Stackpointer
|
|      *          statische Variable
|
----------------------------------------
logisches Speicherende (s."a)")
```

Der Fehler "data space overflow" erscheint, wenn entgegengesetzte "Pfeile" sich erreichen oder zu nahe kommen (bzw. der Compiler merkt, daß dies zur Laufzeit geschehen wird).

B.3 Generierter Code, Optimierungen

Im folgenden werden nicht alle Optimierungen beschrieben, sondern nur Hinweise zum Code, die die effektive Programmierung und die Anwendung der inline-Anweisung ermöglichen.

*** Alle Daten werden von hinten nach vorn in den Speicher geschrieben. Die Inkrementierung eines Pointers entspricht in Wirklichkeit also einer Dekrementierung. Das ist z.B. wichtig, wenn ein C-Programm über Pointer direkt auf den Bildwiederholspeicher zugreifen soll.
Niederwertige Bytes stehen auf der höheren Adresse. Für portable Programme hat das keine Bedeutung.

Der Offset 0 entspricht demnach der aktuellen Obergrenze des Datenstacks. Zwischenergebnisse (bis zu 64) haben noch größere Adressen.

Zur Übersetzung von Ausdrücken:

- Es werden niemals Operanden selbst in die Register geladen, sondern nur Zeiger auf sie. Das ermöglicht eine einheitliche Behandlung aller Datentypen ohne große Zeitverluste (während andere C-Compiler auf 8-Bit-Rechnern oft nur Daten bis zu einer Breite von 16 Bits verarbeiten, d.h., char, int und unsigned)

- Der Zeiger auf den Wert steht nach der Berechnung eines Ausdrucks stets in HL.

- Binäre Operationen werden wie folgt übersetzt:
  Aus

  ```
  a+b;
  ```

  wird

  ```
  LD HL,Adr. von a
  PUSH HL
  LD HL,Adr. von b
  POP DE
  EXX
  LD HL, Adr. des Zwischenergebnisses    (*)
  CALL Add
  ```

  Dabei sind die Befehle "LD HL,..." wie folgt zu verstehen:
  . Ist a eine statische Variable, so wird
    LD HL,nn
    geschrieben.
  . Ist a dynamisch, so wird
    RST 20H
    DA offset

geschrieben.

. Bei den ersten beiden Niveaus für Zwischenergebnisse wird in (*) codiert:
  LD HL,(Adresse)
  (Die Zelle "Adresse" wurde bereits während des Funktionsaufrufes belegt.)
  In anderen (selteneren) Fällen wird die Adresse wie bei dynamischen variablen berechnet.

. Wenn a oder b bereits in Zellen für Zwischenergebnisse stehen, wird geschrieben:
  PUSH HL oder PUSH DE
  EXX
  POP HL
  Dabei wird in der ersten Zeile der Operand mit der kleineren Adresse gewählt.

- Bei '=' wird nach POP DE geschrieben:
  RST 38H
  DB sizeof(result) ;
  beim Typ char dagegen
  LD A,(HL)
  LD (DE),A
  EX DE,HL

- Unäre Operationen:
  Bei der Übersetzung wird (außer bei *p, ++n/--n und n++/n--) geprüft, ob der Operand im Stack für Zwischenergebnisse steht. Wenn nicht, wird geschrieben
  LD HL,Adr.Operand (vgl. binäre Operationen)
  RST 28H
  DB offset
  DB sizeof(Operand)
  Dann wird das entsprechende Unterprogramm gerufen.

  . n++ wird wie folgt übersetzt:
    [LD HL,Adr.Operand]
    LD D,H
    LD E,L
    RST 28H
    DB offset
    DB sizeof(Operand)
    EX DE,HL
    CALL Incr
    EX DE,HL
    Wird der Wert von n++ nicht weiterverwendet (wie etwa in for(n=1;n<N;n++)), so wird n++ übersetzt (also nur 6 Bytes Code statt 13).

  . Wird der *-Operator auf ein Feld angewendet, so ändert der Compiler nur den Typ und schreibt keinen Code.
    Eindimensionale Felder, die als Funktionsargumente übergeben wurden, interpretiert der Compiler als Pointer.

- Feldelemente werden optimiert berechnet. Beispiele:

  static a[4]; a[3];
    ergibt
  LD HL,Adr.von a
  LD DE,-6
  ADD HL,DE

  static long a[4]; static i; a[i];
    ergibt
  LD HL,Adr.von a
  PUSH HL
  LD HL,Adr.von i
  CALL Indirect

```
ADD HL,HL
ADD HL,HL
CALL Substract

static struct {...} a{8]; int i; ... a[i];
  ergibt
LD HL,Adr.von a
PUSH HL
LD HL,Adr.von i
CALL Indxmul
DA sizeof(a[0])
```

- Strukturen werden wie Felder mit konstanten Indizes übersetzt. Ist der Offset 0, so wird nur die Strukturadresse geladen. Insbesondere kostet die Verwendung von Unions nicht mehr Zeit und Platz als die von einfachen Variablen.

- Anweisungen werden entsprechend obigen Prinzipien von links nach rechts übersetzt (mit ev. Zwischenspeicherung von Operationen im Stack 90H...190H). Konstante Teilausdrücke werden sofort aufgelöst, nur das Ergebnis belastet den Konstantenpuffer.
  Es ist garantiert, daß geklammerte konstante Teilausdrücke zusammengefaßt werden.

- Funktionsaufrufe werden wie folgt codiert:
  a) ohne Parameter
  ```
  LD HL,(Funktionsadresse)
  RST 10H
  DB offset (Stackzeiger für Zwischenergebnisse)
  DB 0
  ```

  b) mit Parametern
  ```
  LD HL,(Funktionsadresse)
  PUSH HL
  Berechnung des 1. Parameters
  EX (SP),HL
  PUSH HL
  ...
  Berechnung des n-ten Parameters
  EX (SP),HL
  RST 10H
  DB offset
  DB n
  ```

  Nach dem Aufruf wird, falls das Ergebnis weiterverwendet wird, der Funktionswert mit RST 28H... auf den aktuellen Stack für Zwischenergebnisse umkopiert.

- Funktionen werden wie folgt codiert:

  ```
  DA benötigter Datenstack
  DB Anzahl der Argumente
  DB Länge des 1.Argunments
  ...
  DB Länge des n-ten Arguments
  Code des Funktionsblocks
  ```

  Wie daraus folgt, wird dynamischer Speicherplatz nur beim Funktionsaufruf allociert, nicht bei Eintritt in einen Block.

- Bei Ausdrücken in if-, while- und for-Bedingungen sowie bei '&&', '||' und '?:' wird getestet, ob zuletzt eine Vergleichsoperation ausgeführt wurde. Wenn ja, ist das Z-Flag bereits gesetzt, und das Ergebnis des Vergleichs (0 oder 1) kann verworfen werden. Wenn nein, wird übersetzt.

  ```
  beim Typ char:          LD A,(HL)
                          OR A
  ```

```
beim Typ int/unsigned:    LD A,(HL)
                          DEC HL
                          OR (HL)

beim Typ [unsigned] long: CALL Testprogramm
```

- Die switch- und case-Anweisungen erfordern je 4 Bytes, bei 'case' werden diese mit JR 6 übersprungen.

B.4 Arithmetikroutinen, Zahlendarstellungen

Entsprechend der Anordnung aller Daten von hinten nach vorn im Speicher, hat das niederwertige Byte die höhere Adresse.
Integer- und long-Objekte werden im Zwischenkomplement dargestellt.

Gleitkommazahlen werden logarithmisch zur Basis 2 als Exponent und Mantisse dargestellt:

```
höchste Adresse      exp     Exponentenbyte
        |             m1     höchstwertiges Mantissenbyte
        |             m2
        |             m3     (float)
        |            ...
        ↓             m7     (double)
niedrigste Adresse
```

Dabei enthalten 'exp' den um 80H vergrößerten Exponenten.
m1...m3 (bei float) bzw. m1...m7 (bei double) die normalisierte Mantisse. Letztere hat 'hidden bit'-Format, d.h., Bit 7 von m1 ist bei einer normalisierten Mantisse immer 1. An Stelle dieser redundanten Information steht das Vorzeichen der Zahl (0=positiv, 1=negativ).
Ist 'exp' Null, so ist der Wert der Gleitkommazahl Null, und nur in diesem Fall.

Alle Arithmetikroutine, also für +,-,*,/,%,++ und -- prüfen auf ev. Division durch 0 und (außer bei 'unsigned') auf Überlauf. Bei Unterlauf einer Gleitkommaoperation wird das Ergebnis 0.

Gleitkommaoperationen rechnen bei

- einfacher Genauigkeit intern mit 25-Bit-Mantissen und bei
- doppelter Genauigkeit intern mit 64-Bit-Mantissen

und runden das Ergebnis.
Rechenzeiten für Gleitkommaoperationen sind effektiv:

	einfach	doppelt genau
+ -	160 µs	800...1340 µs
*	960 µs	5800 µs
/	1140 µs	10...12 ms

B.5 Compilerfehler, Weiterentwicklungen

Vorliegende Version 1.0 ist eine Testversion. Der Autor bittet alle Nutzer, den Compiler möglichst "auszureizen" und reproduzierbare Fehler mit genauen Angaben (Quelltext, Schalter, Adressen) zu übermitteln. Auch für Kritiken und Vorschläge ist er jederzeit dankbar.

Als Weiterentwicklung sind geplant:

- Initialisierungen in einfacher Form;

- ein Hilfsprogramm, das übersetzte Programme zusammen mit dem Laufzeitsystem als lauffähige Maschinenprogramme auf Band auslagert;

- Verbesserungen des Laufzeitsystems

u.a.

Anschrift des Autors:
Dr. Reinhard Wobst
Fritz-Hoffmann-Str.5
Dresden
8060

Anhang C: ASCII-Zeichencode

"^J" bedeutet beispielsweise, daß dieser Code durch gleichzeitiges Drücken der CONTROL-Taste und "J" erzeugt wird.
Zusätzlich wird noch die Codierung der Sondertasten am KC 85/1 angegeben.

Steuerzeichen:

dez	okt	hex	Code	Taste	dez	okt	hex	Code	Taste
0	0	0	(nicht darst.)		16	20	10	^P	
1	1	1	^A		17	21	11	^Q	
2	2	2	^B	CL LN	18	22	12	^R	
3	3	3	^C		19	23	13	^S	PAUSE
4	4	4	^D		20	24	14	^T	COLOR
5	5	5	^E		21	25	15	^U	COLOR/SHIFT
6	6	6	^F		22	26	16	^V	
7	7	7	^G		23	27	17	^W	
8	10	8	^H	←	24	30	18	^X	→|
9	11	9	^I	→	25	31	19	^Y	|←
10	12	A	^J	↓	26	32	1A	^Z	INS
11	13	B	^K	↑	27	33	1B	^[	ESC
12	14	C	^L		28	34	1C	^\	LIST
13	15	D	^M	ENTER	29	35	1D	^]	RUN
14	16	E	^N		30	36	1E		CONT
15	17	F	^O		31	37	1F		DEL

Nichtsteuerzeichen:

dez	okt	hex	Zeichen	dez	okt	hex	Zeichen	dez	okt	hex	Zeichen
32	40	20	space	64	100	40	@	96	140	60	`
33	41	21	!	65	101	41	A	97	141	61	a
34	42	22	"	66	102	42	B	98	142	62	b
35	43	23	#	67	103	43	C	99	143	63	c
36	44	24	$	68	104	44	D	100	144	64	d
37	45	25	%	69	105	45	E	101	145	65	e
38	46	26	&	70	106	46	F	102	146	66	f
39	47	27	'	71	107	47	G	103	147	67	g
40	50	28	(	72	110	48	H	104	150	68	h
41	51	29	)	73	111	49	I	105	151	69	i
42	52	2A	*	74	112	4A	J	106	152	6A	j
43	53	2B	+	75	113	4B	K	107	153	6B	k
44	54	2C	,	76	114	4C	L	108	154	6C	l
45	55	2D	-	77	115	4D	M	109	155	6D	m
46	56	2E	.	78	116	4E	N	110	156	6E	n
47	57	2F	/	79	117	4F	O	111	157	6F	o
48	60	30	0	80	120	50	P	112	160	70	p
49	61	31	1	81	121	51	Q	113	161	71	q
50	62	32	2	82	122	52	R	114	162	72	r
51	63	33	3	83	123	53	S	115	163	73	s
52	64	34	4	84	124	54	T	116	164	74	t
53	65	35	5	85	125	55	U	117	165	75	u
54	66	36	6	86	126	56	V	118	166	76	v
55	67	37	7	87	127	57	W	119	167	77	w
56	70	38	8	88	130	58	X	120	170	78	x
57	71	39	9	89	131	59	Y	121	171	79	y
58	72	3A	:	90	132	5A	Z	122	172	7A	z
59	73	3B	;	91	133	5B	[	123	173	7B	{
60	74	3C	<	92	134	5C	\	124	174	7C	|
61	75	3D	=	93	135	5D	]	125	175	7D	}
62	76	3E	>	94	136	5E	^	126	176	7E	~
63	77	3F	?	95	137	5F	_	127	177	7F	del

Anhang D: Literatur

[1] Kernighan,B.W.; Ritchie, D.M.: The C Programming Language;
Englewood Cliffs, Prentice Hall 1978
dt.: Programmieren in C; München/Wien, Carl Hanser Verlag 1983
russ.: Jazyk programirovan'a si;
Finansy i Statistika Moskau 1985
(diese Ausgabe enthält außerdem [9]!)

Die "Bibel" für C-Programmierer, ohne Frage das wichtigste Buch über diese Programmiersprache. Es wird oft mit "K&R" bezeichnet.
Trotz verschiedener Erweiterungen und Änderungen von C richten sich die meisten C-Programmierer nach dieser Beschreibung (bzw. dem UNIX-Compiler). Nicht zuletzt deswegen sind C-Programme so portabel.

[2] BYTE Nov.1985, S.275ff

[3] Claßen,L.; Oefler,U.: UNIX und C - Ein Anwenderhandbuch;
Verlag Technik Berlin 1987
Das Buch enthält eine recht nützliche C-Beschreibung, ist aber kein Lehrbuch.

[4] Gehani,N.: Advanced C: Food for the Educated Palate;
Rockville, Maryland; Computer Science Press 1985

Die Sprache wird nicht lückenlos dargelegt, aber der Autor geht auf viele interessante und wichtige Details ein, die in [1] zu kurz kommen.

[5] Kern,C.O.: Five C Compilers for CP/M 80; BYTE v8 (Sonderheft), 1985, S.130

[6] Relph,R., Hahn,S., Viles,F.: Benchmarking C Compilers;
Dr.Dobb's Journal of Software Tools 8/86, S.30

[7] Clark,D.D.: Benchmarks for C; BYTE 1/86 S.307

[8] Purdum,J.J.: Einführung in C; Haar bei München, Verlag Markt und Technik 1985
(engl. Original: C Programming Gide)

Eine verständliche Einführung in die C-Programmierung, aber sehr unvollständig und daher nicht als Handbuch zu benutzen.

[9] Feuer,A.: C Puzzle Book; Englewood Cliffs, Prentice Hall 1982.
dt.: Das C-Puzzle-Buch; München/Wien, Carl Hanser Verlag 1983.
russ.: In [1] (russ.) enthalten.

Eine sehr empfehlenswerte Anleitung, wie man in C programmiert und wo man Fehler macht. Die Kenntnis der Sprache ist Voraussetzung.
Das Buch enthält eine Sammlung von Aufgaben in Gestalt von Programmfragmenten, deren Verhalten am Rechner auszuprobieren und zu erklären ist. Die Lösungen sind ausführlich kommentiert.

[10] X/OPEN Portability Guide - Programming Language C. Juli 1985.
Eine ausgezeichnete, beinahe lückenlose Beschreibung von C unter UNIX.

[11] Saeltzer,G.; Wenlder,F.: Wissensspeicher Programmiersprache C; rechentechnik/datenverarbeitung 5/1987, S.24

Ein sehr kompakter Überblick, aber unvollständig.