Markt&Technik

128er-Software

DIGITAL RESEARCH®

CBASIC Compiler™

FROM DIGITAL RESEARCH® THE CREATORS OF CP/M

5¼"-Diskette im 1541-Format
für Commodore 128 (128 D)
Handbuch in englischer Sprache

Markt&Technik

128er-Software

DIGITAL
RESEARCH®

CBASIC Compiler™

FROM DIGITAL RESEARCH® THE CREATORS OF CP/M

5¼"-Diskette im 1541-Format
für Commodore 128 (128 D)
Handbuch in englischer Sprache

Markt&Technik Verlag Aktiengesellschaft · Hans-Pinsel-Straße 2 · 8013 Haar

Die Informationen in diesem Handbuch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können
für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine
Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

»Commodore 128 Personal Computer« ist eine Produktbezeichnung der Commodore Büromaschinen GmbH, Frankfurt,
die ebenso wie der Name »Commodore« Schutzrecht genießt. Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis
der Schutzrechtsinhaberin.

CBASIC, CP/M, CP/M-86 und CP/NET sind eingetragene Warenzeichen der Digital Research, Inc., USA

SID, CB 80, CB 86, Concurrent CP/M 86, MP/M, GSX, GSX-86, MP/M II und MP/M 86
sind Warenzeichen der Digital Research, Inc., USA

Intel ist ein eingetragenes Warenzeichen der Intel, Corp., USA

Zilog und Z 80 sind eingetragene Warenzeichen der Zilog, Inc., USA

© 1981, 1982, 1983 by Digital Research, Inc., USA
© 1986 by Markt&Technik, 8013 Haar bei München
Alle Rechte vorbehalten

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Der CBASIC-Compiler für den Commodore 128

Sehr geehrter CBASIC-Compiler-Anwender,

vielen Dank, daß Sie sich zum Kauf des CBASIC-Compilers von Digital Research für den Commodore 128 entschieden haben. Bitte nehmen Sie sich die Zeit, diese Seiten gründlich durchzulesen, da Sie die Bedienung des Compilers erleichtern und Sie vor dem Verlust Ihres neuen Programmes durch Fehlbedienung schützen.

Der CBASIC-Compiler ist für die meisten Computer mit den Betriebssystemen CP/M, CP/M 3.0, CP/M-86, MS-DOS und PC-DOS erhältlich. Das Handbuch ist deshalb nicht speziell auf den Commodore 128 ausgerichtet. Auf diesen Seiten stehen die Informationen, die Sie für den Einsatz des CBASIC-Compilers auf dem Commodore 128 zusätzlich benötigen.

Wichtig! Bitte denken Sie daran, daß Sie sich Kopien von der mitgelieferten Originaldiskette erstellen, bevor Sie mit dem Produkt arbeiten. So können Sie immer wieder auf die Originaldisketten zurückgreifen, wenn eine Arbeitsdiskette einmal defekt werden sollte. Informationen darüber, wie Sie eine Kopie einer Diskette erstellen, finden Sie im Bedienungshandbuch Ihres Computers.

Auf der 1541-Diskette befindet sich der komplette Compiler, der Linker und der Bibliotheks-Manager, sowie die Laufzeitbibliothek: ganz einfach alles, was Sie benötigen, um kleine oder große Programme mit Profi-Qualität zu entwickeln, die auf jedem CP/M 2.2 und CP/M 3.0-Computer lauffähig sind. Wenn Sie ein 1571-Laufwerk haben, sollten Sie sich die Diskette in das 1571-Format umkopieren.

Kompatibilität

Auch wenn dieses Produkt für sich komplett ist, benötigen Sie zum Erstellen von Arbeitsdisketten, zum Eingeben von Quelldateien und zum Testen Ihrer Programme einige Hilfsprogramme, die mit Ihrem Computer geliefert wurden. Dieses Produkt ist in bezug auf die Erfassung der Quelldateien voll kompatibel zu ED (dem Editor von CP/M), WordStar und vielen anderen Editoren und Textverarbeitungen. Ebenso besteht Kompatibilität zu RMAC zum Erstellen von Assembler-Unterroutinen und zu SID zum Test von Programmen. RMAC kann dazu verwendet werden, Spracherweiterungen zu erstellen, die jede beliebige Funktion ausführen, zum

Beispiel die Steuerung der Klang- und Grafik-Eigenschaften Ihres Computers.

Mit Ausnahme der von Ihnen eventuell erstellten Assembler-Unterroutinen können die CB-80-Programme auch mit dem CB-86-Compiler übersetzt werden, der für die Betriebssysteme CP/M-86, PC-DOS und MS-DOS erhältlich ist. Sie können sicher sein, daß Ihre CBASIC-Programme auf einer Vielzahl von Computern lauffähig sind, wobei höchsten eine Neukompilierung (z.B. für 8088- oder 8086-Computer) notwendig ist. Ihre Anwendungen sind also vollkommen portabel und kompatibel zur 16-Bit-Welt.

Weitere Programme für Ihren Computer

Wir glauben, daß sie viel Spaß am CBASIC-Compiler haben werden und daß er für Sie schnell zu einem leistungsfähigen - ja idealen - Werkzeug zur Erstellung Ihrer Anwendungen werden wird.

Wir möchten noch darauf hinweisen, daß Markt & Technik auch den Pascal/MT+-Compiler von Digital Research für den Commodore 128 anbietet. Diese Sprache bietet sich zur Programmierung einer Vielzahl von Anwendungen an und eignet sich besonders für solche Programme, die im Speicher oder auf der Diskette auf komplexe Datenstrukturen zugreifen müssen. Näheres zu diesem und weiteren Programmen für Ihren Commodore 128 aus dem Markt & Technik Verlag finden Sie ab der folgenden Seite.

Bücher zum Commodore 128 PC

Markt&Technik
Die faszinierende Welt der Grafik erklärt an zahlreichen Anwendungsbeispielen in BASIC und ASSEMBLER: Sprites ★ Shapes ★ Animation
GRAFIK-PROGRAMMIERUNG C128
Heimo Ponnath

H. Ponnath

Grafik-Programmierung C 128

1. Quartal 1986, ca. 250 Seiten
inkl. Beispieldiskette

Ein mächtiges Werkzeug hat der Anwender von Computergrafik mit dem Basic 7.0 des Commodore 128 PC in den Händen! Was man damit alles anfangen kann, soll Ihnen dieses Buch zeigen: hochauflösende Grafik, Multicolorbilder, Sprites und Shapes werden anhand von vielen Beispielprogrammen besprochen. Die Videochips und ihre Möglichkeiten sind ebenso Thema wie einige nützliche Assemblerroutinen, die Speicherorganisation, der 80-Zeichen-Bildschirm und vieles andere mehr. Außerdem enthält das Buch eine Diskette mit allen Programmen.

Best.-Nr. MT 90202
ISBN 3-89090-202-2
DM 52,-/sFr. 47,80/öS 405,60

P. Rosenbeck

Das Commodore 128-Handbuch

Juli 1985, 383 Seiten

Dieses Buch sagt Ihnen alles, was Sie über Ihren C128 wissen müssen: die Hardware, die drei Betriebssystem-Modi und was die CP/M-Fähigkeit für Ihren Computer bedeutet. Aber Sie werden irgendwann Lust verspüren, tiefer in Ihren C128 einzusteigen. Auch dafür ist gesorgt: an einen Assemblerkurs, der Ihnen zugleich die Funktionsweise des eingebauten Monitors nahebringt, schließen sich Kapitel an, die mit Ihnen auf Entdeckungsreise ins Innere der Maschine gehen. Daß die Reise spannend wird, dafür sorgen die Beispiele, aus denen Sie viel über die Interna des Systems lernen können - bis hin zur Grafik-Programmierung.

Best.-Nr. MT 809
ISBN 3-89090-195-6
DM 52,-/sFr. 47,80/öS 405,60

J. Hückstädt

BASIC 7.0 auf dem Commodore 128

Juli 1985, 239 Seiten

Das neue BASIC 7.0 des C128 eröffnet mit seinen ca. 150 Befehlen ganz neue Dimensionen der BASIC-Programmierung. Es ermöglicht dem Anfänger den einfachen und effektiven Zugriff auf die erstaunlichen Grafik- und Tonmöglichkeiten des C128; der Fortgeschrittene findet die nötigen Informationen für (auch systemnahe) Profi-Programmierung mit strukturierten Sprachmitteln.

An praxisnahen Beispielen (wie z.B. der Dateiverwaltung) zeigt der Autor auf, wie man die für den 128er typischen Merkmale und Eigenschaften (Sprites, Shapes, hochauflösende Grafik, Musikprogrammierung und Geräusche) optimal nutzt!.

Best.-Nr. MT 808
ISBN 3-89090-170-0
DM 52,-/sFr. 47,80/öS405,60

Markt&Technik
BUCHVERLAG

Hans-Pinsel-Str. 2, 8013 Haar bei München

Markt & Technik-Fachbücher erhalten Sie bei Ihrem Buchhändler

Spitzen-Software für Commodore 128 PC

dBASE II, Version 2.41

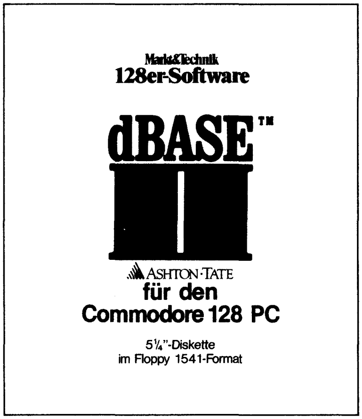
dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, eröffnet Ihnen optimale Möglichkeiten der Daten- u. Dateihandhabung. Einfach u. schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.

dBASE II für den Commodore 128 PC
Bestell-Nr. MS 303 (5¹/₄″-Diskette)

Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Für nur DM 199,–* (sFr. 178,–/öS 1890,–*)

***inkl. MwSt. Unverbindliche Preisempfehlung**



Und dazu die weiterführende Literatur:

Zu einem Weltbestseller unter den Datenbanksystemen gehört auch ein klassisches Einführungs- und Nachschlagewerk! Dieses Buch von dem deutschen Erfolgsautor Dr. Peter Albrecht begleitet Sie mit nützlichen Hinweisen bei Ihrer täglichen Arbeit mit **dBASE II.** Schon nach Beherrschung weniger Befehle ist der Einsteiger in der Lage, Dateien zu erstellen, mit Informationen zu laden und auszuwerten.

Best.-Nr. MT 838
ISBN 3-89090-189-1
DM 49,- (sFr. 45,10/öS 382,20)

Erhältlich bei Ihrem Buchhändler.

Sie erhalten jedes **WordStar-**, **dBASE II-** und **MULTIPLAN-Programm** für Ihren Commodore 128 PC fertig angepaßt (Bildschirmsteuerung und Druckerinstallation). **Jeweils Originalprodukte!** Jedes Programmpaket enthält außerdem ein ausführliches Handbuch mit kompakter Befehlsübersicht.

Diese Markt&Technik-Softwareprodukte erhalten Sie in den Computer-Abteilungen der Kaufhäuser oder bei Ihrem Computerhändler.

Markt&Technik
128er-Software
Hans-Pinsel-Straße 2, 8013 Haar bei München

Spitzen-Software für Commodore 128 PC

MULTIPLAN, Version 1.06

Wenn Sie die zeitraubende manuelle Verwaltung tabellarischer Aufstellungen mit Bleistift, Radiergummi und Rechenmaschine satt haben, dann ist MULTIPLAN, das System zur Bearbeitung »elektronischer Datenblätter«, genau das richtige für Sie! Das benutzerfreundliche und leistungsfähige Tabellenkalkulationsprogramm kann bei allen Analyse- und Planungsberechnungen eingesetzt werden wie z. B. Budgetplanungen, Produktkalkulationen, Personalkosten usw. Spezielle Formatierungs-, Aufbereitungs- und Druckanweisungen ermöglichen außerdem optimal aufbereitete Präsentationsunterlagen!

MULTIPLAN für den Commodore 128 PC
Bestell-Nr. MS 203 (5¹/₄″-Diskette)

Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Für nur DM 199,–* (sFr. 178,–/öS 1890,–*)
***inkl. MwSt. Unverbindliche Preisempfehlung**

Und dazu die weiterführende Literatur:

Dank seiner Menütechnik ist **MULTIPLAN** sehr schnell erlernbar. Mit diesem Buch von Dr. Peter Albrecht werden Sie Ihre Tabellenkalulation ohne Probleme in den Griff bekommen. Als Nachschlagewerk leistet es auch dem Profi nützliche Dienste.

Best.-Nr. MT 836
ISBN 3-89090-187-5
DM 49,- (sFr. 45,10/öS 382,20)

Erhältlich bei Ihrem Buchhändler.

Sie erhalten jedes **WordStar-**, **dBASE II-** und **MULTIPLAN-Programm** für Ihren Commodore 128 PC fertig angepaßt (Bildschirmsteuerung und Druckerinstallation). **Jeweils Originalprodukte!** Jedes Programmpaket enthält außerdem ein ausführliches Handbuch mit kompakter Befehlsübersicht.

Diese Markt&Technik-Softwareprodukte erhalten Sie in den Computer-Abteilungen der Kaufhäuser oder bei Ihrem Computerhändler.

Markt&Technik
128er-Software
Hans-Pinsel-Straße 2, 8013 Haar bei München

TOP ASS

Der ASE-Macro-Assembler für den Commodore 128 PC mit integriertem Editor und Linker.

Dieser 6502-Macroassembler setzt neue Maßstäbe. Seine Leistungsfähigkeit wird jeden CP/M-Assembler-verwöhnten Maschinenprogrammierer überzeugen:

- integrierter Editor, der schon bei der Eingabe des Quelltextes eine Syntaxüberprüfung vornimmt;
- integrierter Linker, mit dem quellgesteuertes Linken von relokatiblen Modulen möglich ist;
- assemblereigene schnelle und gleichzeitig sehr leistungsfähige Integerarithmetik;
- über 2000 Labels können gleichzeitig verwaltet werden, das heißt Maschinenprogramme bis zu einer Länge von ca. 25 KByte Objektcode können bei Bedarf in einem Rutsch assembliert werden;
- Macros mit beliebig vielen Parametern, Macrobibliotheken, Minimacs, bedingte Assemblierung, Labeleingabe im Dialog, Ausgabe formatierter Assemblerlistings, Ausgabe sortierter Symboltabellen und vieles andere mehr.

Außerdem wird der ASE-Macroassembler von einem sehr guten Monitor und einem Relativlader unterstützt, der relokatible Module an beliebige Speicheradressen laden kann und endlich Schluß macht mit den Dutzenden Maschinenprogrammen auf Diskette, die sich nur durch ihre Startadresse unterscheiden!

Lernen Sie es kennen, das TOPASS Assembler-Entwicklungssystem! Es lohnt sich!

Best.-Nr. MD 253A **DM 89,–**
inkl. MwSt. Unverbindliche Preisempfehlung.

PROTEXT

Die Profi-Textverarbeitung mit vollautomatischer Silbentrennung, integrierter Tabellenkalkulation und Zusatzprogramm zum Überprüfen der Rechtschreibung für den Commodore 128 PC.

PROTEXT ist ein leicht bedienbares Textprogramm mit hoher Leistungsfähigkeit. Eingebaute Hilfefunktionen ermöglichen eine schnelle Einarbeitung. Mit **PROTEXT** sind daher auch Anfänger in der Lage, alle Vorteile eines professionellen Textprogramms zu nutzen. Überzeugen Sie sich selbst!

Was **PROTEXT** alles kann:

- Farbkombination für Hintergrund und Schrift (Vordergrund) frei wählbar;
- formatierte Ausgabe auf Bildschirm und Drucker mit programmierbaren Haltepunkten über serielle, V24- oder zwei Software-Centronics-Schnittstellen;
- vielfältige Formatanweisungen: linker/rechter Rand, vollautomatische Silbentrennung, Kopf-/Fußzeilen, Fußnoten, Zentrieren usw.;
- schnelle selbstlernende Textkorrektur mit deutschem (ca. 25000 Worte) Grundwortschatz sowie neun Kundenbibliotheken, die in Text umgewandelt, bearbeitet, ergänzt, sortiert und ausdruckbar sind;
- Textübertragung per DFÜ mit Space-Optimierung und automatischer Fehlerkorrektur;
- leistungsfähige Rechenmöglichkeiten mit Zeilenmarkierung (Rechentabulator), Kolonnenverarbeitung, progr. Tabellenkalkulation und Taschenrechner.

Hardwareanforderungen: C128 oder C128D, 80-Zeichen-Monitor, Commodore-Drucker oder Drucker mit Centronics-Schnittstelle.

Best.-Nr. MD 254A **DM 89,–** inkl. MwSt. Unverbindliche Preisempfehlung.

TOPASS und PROTEXT erhalten Sie
in den Computer-Abteilungen der
Kaufhäuser und in Computershops.

Markt&Technik
128er-Software
Hans-Pinsel-Straße 2, 8013 Haar b. München

DIGITAL
RESEARCH™

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München

CBASIC® Compiler (CB80™)

Language

Programming Guide

DIGITAL
RESEARCH™

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München

CBASIC® Compiler (CB80™)
Language
Programming Guide

COPYRIGHT

Copyright © 1982 by Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

This manual is, however, tutorial in nature. Thus, the reader is granted permission to include the example programs, either in whole or in part, in his own programs.

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CBASIC, CP/M, CP/M-86, and CP/NET are registered trademarks of Digital Research. CB80, CB86, Concurrent CP/M-86, LK80, MP/M, MP/M II, MP/M-80, MP/M-86, RMAC, and SID are trademarks of Digital Research. Intel is a registered trademark of Intel Corporation. Z80 is a registered trademark of Zilog, Inc.

The *CBASIC Compiler (CB80) Language Programming Guide* was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

First Edition: January 1983

Foreword

The CBASIC® Compiler is a compiler version of the CBASIC programming language. For the software developer interested in maximizing the execution speed of commercial applications programs, the CBASIC Compiler is an excellent choice.

Digital Research designed the CBASIC Compiler for use under single-user, multiuser, and concurrent operating systems based on both 8-bit and 16-bit microprocessors.

- 8-bit CBASIC Compiler, CB80™, runs under the CP/M® versions 2 and 3, MP/M™, and CP/NET® operating systems based on the Intel® 8080, 8085, or Zilog Z80® microprocessor.
- 16-bit CBASIC Compiler, CB86™, runs under the CP/M-86®, MP/M-86™, and Concurrent CP/M-86™ operating systems based on the Intel 8086, 8088 family of microprocessors.

The *CBASIC Compiler Language Programming Guide* provides a short demonstration program to help you get your CBASIC Compiler system up and running. The manual is divided into five sections.

- Section 1 is an introduction and demonstration program.
- Section 2 describes the compiler, CB80.
- Section 3 describes the link editor, LK80™.
- Section 4 describes the indexed library and the library manager utility program.
- Section 5 explains the machine-level environment of the CBASIC Compiler.

Use this Programming Guide in conjunction with the *CBASIC Compiler Language Reference Manual.* Together, the manuals provide all the information you need to use the CBASIC Compiler to its full potential.

Digital Research is very interested in your comments on programs and documentation. Please use the Software Performance Reports and the Reader Comment card to help us provide you with the best microcomputer software and documentation.

Table of Contents

Table of Contents (continued)

Appendixes

Table of Contents (continued)

List of Tables

List of Figures

Section 1
Getting Started with the CBASIC Compiler

A compiler is a computer program that translates high-level programming language instructions into machine readable code. The compiler takes as input a user-written source program and produces as output a machine-level object program. Some compilers translate a user-written source program into a program that a computer can execute directly. The CBASIC Compiler system, however, uses a link editor and a library in addition to the compiler. Together, the three components translate your CBASIC source code file into a directly executable program using your microcomputer's memory space as efficiently as possible. The system enables you to modularize programs for quick and easy maintenance. The result is a programming system that rivals the performance of systems based on much larger machines.

The primary advantage that compilers provide over other methods of translation is speed. Compiled applications programs execute faster than interpreted programs because the compiler creates a program that the computer can execute directly.

1.1 Components

The three components that make up the CBASIC Compiler system are listed in the directory of your CBASIC product disk along with three compiler overlay files and the library manager utility:

- The compiler, CB80, translates CBASIC source code into relocatable machine code modules. Source programs default to a .BAS filetype unless specified otherwise. CB80 generates .REL files.
- The link editor, LK80, combines the relocatable object modules that the compiler creates and relocatable routines from the library into a directly executable program with optional overlays. LK80 generates .COM files.
- The library provides relocatable routines that allocate and release memory, determine available memory space, and perform arithmetic operations and input/output processing.

1.2 A Demonstration

The following demonstration program can help you learn how to compile, link, and run your first CBASIC program. The instructions are for the CBASIC Compiler on a CP/M-based system with two floppy-disk drives. You should already be familiar with CP/M and a text editor.

Make a back-up copy of your master CBASIC Compiler product disk. Place your operating system disk in drive A and a copy of your CBASIC Compiler disk in drive B.

1. Write the source program.

   Using your text editor, create a file named TEST.BAS on your CBASIC Compiler disk in drive B. Enter the following program into TEST.BAS exactly as it appears below:

   ```
   PRINT
   FOR I% = 1 TO 10
       PRINT I%;  "TESTING THE CBASIC COMPILER!"
   NEXT I%
   PRINT
   PRINT "FINISHED"
   END
   ```

2. Compile the program.

   To start CB80, enter the following command. Be sure drive B is the default drive.

   ```
   B>CB80 TEST
   ```

   CB80 assumes a filetype of .BAS for the file you specify in the compiler command line unless otherwise specified. A sign-on message, a listing of your source program, and several diagnostic messages display on your terminal.

```
----------------------------------------------
CBASIC Compiler CB80               Version 1.x
Ser. No. 000-0000          Copyright (c) 1982
Digital Research, Inc.  All rights reserved
----------------------------------------------
end of pass 1
end of pass 2
    1:  003eh PRINT
    2:  0041h FOR I% = 1 TO 10
    3:  004ah   PRINT I%; "TESTING THE CBASIC COMPILER!"
    4:  0056h NEXT I%
    5:  0064h PRINT
    6:  0067h PRINT "FINISHED"
    7:  0070h END
end of compilation
no errors detected
code area size:     112        0070h
data area size:     2          0002h
common area size:   0          0000h
symbol table space remaining: 31890
```

Section 2.1 describes the various parts of the listing. The message no errors detected indicates a successful compilation. CB80 creates a relocatable file for the TEST.BAS program. The directory for disk B should have the new file TEST.REL.

3. Link the program.

   To start LK80, enter the following command. Be sure drive B is the default drive.

   ```
   B>LK80 TEST
   ```

   LK80 assumes a filetype of .REL for the file you specify in the linker command line. A sign-on message and several diagnostic messages display on your terminal.

```
-----------------------------------------------
LK80                                 Version 1.x
Ser. No. 000-0000          Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-----------------------------------------------
code size:        1200 (0100-12FF)
common size:      0000
data size:        0166 (1300-1465)
symbol table space remaining:     124F
```

If you get no error messages, the program has been linked successfully. LK80 creates a directly executable program. The directory for disk B should have the new file TEST.COM.

4. Run the program.

   To run the TEST.COM program, enter the following command. Be sure drive B is the default drive.

   ```
   B>TEST
   ```

   The following output should appear on your terminal:

   ```
   1 TESTING THE CBASIC COMPILER!
   2 TESTING THE CBASIC COMPILER!
   3 TESTING THE CBASIC COMPILER!
   4 TESTING THE CBASIC COMPILER!
   5 TESTING THE CBASIC COMPILER!
   6 TESTING THE CBASIC COMPILER!
   7 TESTING THE CBASIC COMPILER!
   8 TESTING THE CBASIC COMPILER!
   9 TESTING THE CBASIC COMPILER!
   10 TESTING THE CBASIC COMPILER!

   FINISHED
   ```

End of Section 1

Section 2
The Compiler, CB80

CB80 consists of an executable file with filetype .COM and three overlay files. Your CBASIC Compiler product disk should contain the following four files:

- CB80.COM
- CB80.OV1
- CB80.OV2
- CB80.OV3

When compiling a CBASIC program, all four files must be on the logged-in drive. The source program file can be on any logical drive.

2.1 Compiling Programs

CB80 takes a CBASIC source program as input and generates a relocatable object file. During compilation, CB80 creates the following temporary work files with a .TMP filetype:

```
PA.TMP
QCODE.TMP
DATA.TMP
```

Unless compilation is unsuccessful, you never see these temporary files listed in a directory. CB80 erases the files automatically when compilation is finished. CB80 also erases the temporary files if they are on disk before you start the compiler.

The size of the .TMP files varies according to the size of the source program. The amount of temporary space required is approximately equal to the amount of space the source program occupies. If you do not have enough work space on disk for the compiler, you can break up large programs into modules and compile each module separately.

The following is an example of a CB80 listing:

```
-------------------------------------------
CBASIC Compiler CB80              Version 1.x
Ser. No.                    Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-------------------------------------------
end of pass 1
end of pass 2
    1:  003eh PRINT
    2:  0041h FOR I% = 1 TO 10
    3:  004ah   PRINT I%; "TESTING THE CBASIC COMPILER!"
    4:  0056h NEXT I%
    5:  0064h PRINT
    6:  0067h PRINT "FINISHED"
    7:  0070h END
end of compilation
no errors detected
code area size:     112        0070h
data area size:     2          0002h
common area size:  0          0000h
symbol table space remaining: 31890
```

Certain phases of the compilation process are combined into a module called a pass. CB80 is a three-pass compiler. Following the sign-on message, CB80 indicates the completion of the first two passes with a message. The program listing includes the line numbers, relative addresses for the code that each line generates, and the actual source code lines. In the preceding listing, 1: is an example of a line number. 003eh is a relative address for the relocatable code that the first PRINT statement generates.

CB80 prints the total number of compilation errors detected in the program following the message end of compilation. The message no errors detected, however, indicates a successful compilation. The last four messages indicate the amount of space CB80 allocates for certain segments of data. Refer to Section 5 for an explanation of memory allocation. If CB80 detects errors, the relative addresses and the memory allocation messages do not print.

To complete the compilation process, CB80 generates a relocatable object file. The relocatable file has the same filename as the source program and has a .REL filetype. The .REL file requires approximately the same amount of space as the source program. If the source program contains errors that prevent a successful compilation, CB80 does not generate the .REL file.

2.1.1 CB80 Command Lines

The command line starts CB80, specifies the file to compile, and passes special information in the form of compiler directives. The following command line compiles the source program in a file named TEST. CB80 assumes a filetype of .BAS unless otherwise specified.

```
CB80 TEST
```

Enter a complete file specification to override the .BAS filetype. The following command line compiles the source program in a file named TEST.PR1. Remember, source files cannot have a .REL filetype.

```
CB80 TEST.PR1
```

Source files can be on any logical disk drive. The following command line compiles the source program TEST.PR1 from drive D:

```
CB80 D:TEST.PR1
```

If you type an incorrect command line or neglect to enter a filename, CB80 indicates the error with the message invalid command line.

2.1.2 Compiler Errors

CB80 reports three different types of compiler errors. The first type, file system and memory space errors, includes mistakes such as invalid command lines, read errors, and out of memory conditions. CB80 indicates file system and memory space errors with literal messages such as disk full and symbol table overflow. Refer to Appendix B, Table B-1, for a complete listing of file system and memory space error messages.

The second type, compilation errors, includes misuses of the CBASIC language such as invalid characters, improper data type specifications, and missing delimiters. CB80 inserts an integer value in the compiler listing of the source program to indicate the occurrence of a compilation error. The integer corresponds to an error description listed in Appendix B, Table B-2.

The third type, fatal compiler errors, should never occur during your experience with the CBASIC Compiler. CB80 indicates a fatal compiler error with the following message. The XXX stands for a three-digit integer value.

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE XXXX
```

If this error message occurs during compilation of your CBASIC program, contact the Digital Research Technical Support Center. Please report the three-digit integer and the circumstances under which the error occurs.

2.2 Compiler Directives

Compiler directives are special instructions to CB80. The CBASIC Compiler supports two different ways to specify compiler directives: source code compiler directives and command-line toggles.

2.2.1 Source Code Compiler Directives

Source code compiler directives are special keywords that do not translate into executable code. All source code compiler directives begin with a percent sign. You cannot place blanks between the percent sign and the rest of the keyword. Only blanks and tab characters can precede a directive. Source code compiler directives cannot appear on the same line with CBASIC statements or functions. CB80 ignores all characters on the same line that are not part of the directive. A source code compiler directive cannot span more than one line with a continuation character. You cannot label source code compiler directives.

The CBASIC Compiler supports the following six source code compiler directives:

- %NOLIST
- %LIST
- %EJECT
- %PAGE
- %INCLUDE
- %DEBUG

Normally, CB80 generates a listing of the source program during compilation. The %NOLIST directive tells CB80 not to list anything that follows the %NOLIST in the program. The %LIST directive tells CB80 to resume the listing. Use %LIST and %NOLIST any number of times in a program. Toggle B described in Section 2.2.2 suppresses all listings regardless of any directives in the source code.

The %EJECT directive tells CB80 to continue the program listing at the top of the next page of paper. %EJECT works only when you direct the listing to a printer. CB80 ignores %EJECT if the %NOLIST directive is in effect, or if you direct the listing to the console or a disk file.

The %PAGE directive sets the page length for a listing directed to a printer. The page length you specify.must be an unsigned integer placed after the %PAGE keyword, as shown in the following example:

```
%PAGE 40
```

The %INCLUDE directive tells CB80 to include the code from a specified source file along with the original compiling program. The included source file is incorporated into the original program immediately following the %INCLUDE. Specify the filename, the filetype, and the drive that holds the file. CB80 assumes the default drive and a .BAS filetype if not specified otherwise. The following examples show three variations of %INCLUDE:

```
%INCLUDE CONDEF

%INCLUDE CONDEF.INC

%INCLUDE D:CONDEF.INC
```

You can nest included files six deep. The maximum nesting depth depends on your particular implementation of the CBASIC Compiler. Refer to Appendix A for current implementation dependent values.

The %DEBUG directive works with three command line toggles: the I, N, and V toggles. You can switch these three toggles on or off from within the program source code. To turn a toggle on, place the toggle letter after the %DEBUG keyword. To turn a toggle off, place the toggle letter preceded by a minus sign after the %DEBUG keyword. The following examples show variations of the %DEBUG directive:

```
%DEBUG I

%DEBUG -I

%DEBUG INV

%DEBUG -I-N-V
```

2.2.2 CB80 Command Line Toggles

Command line toggles are single-letter compiler directives that you specify in the CB80 command line instead of in the source program. Once a toggle is set, it normally remains set through the entire compilation process. The %DEBUG directive can change the I, N, and V toggles during compilation. Place letters within brackets following the file specification in a CB80 command line. Letters can be lower- or upper-case. If you enter conflicting toggles in a command line, the last one read from left to right takes effect. Certain toggles require an additional parameter enclosed in parentheses. The following examples show several ways to specify command line toggles:

```
CB80 TEST [B]

CB80 TEST.BAS [B, P, S]

CB80 FILE.DAT [BPW(72)]

CB80 CALCS.PRG [N] [O] [P]

CB80 DATA.OVL [ pon ]
```

CB80 supports the fifteen command line toggles listed in the following table:

Table 2-1. CB80 Toggles

Toggle	*Instruction*
B	Suppress listing of the source file.
C	Change the default %INCLUDE file disk.
F	Send the source listing to a disk file on the same drive as the source file.
I	Interlist the generated code with the source file.
L	Set the page length for printed listings.
N	Generate code for line numbers.
O	Suppress the generation of the object .REL file.
P	List the source file on the printer.
R	Change the disk that the .REL file is written to.
S	Include symbol name information in the .REL file.
T	List the symbol table following the source listing.
U	Generate error messages for undeclared variables.
V	Put source code line numbers into the .SYM file.
W	Set the page width for printed listings.
X	Change the disk used for the work files.

The B toggle tells CB80 not to list the source program on the console screen. However, compiler errors and statistical data concerning size of code and data areas display on the screen. The B toggle overrides other toggles that control compiler output.

The C toggle specifies the default drive for include files. Enclose the new drive specification in parentheses following the C. If a drive has been specified in the %INCLUDE directive, the C toggle has no effect. The C toggle allows program development to be independent of your hardware configuration.

The F toggle tells CB80 to send the source listing to a disk file that is on the same drive as the source file. The new file has the same filename as the source file and has a .LST filetype.

The I toggle interlists compiler-generated code with the original source statements. Compiler-generated code uses standard 8080 mnemonics.

The L toggle changes the page length for a listing directed to a printer. Enclose the new length in parentheses following the L. The length must be an unsigned integer, as in the following example:

```
CB80 TEST [L(50)]
```

The N toggle generates code that saves the current line number for each physical line in a source program. The code enables the ERRL function to return the line number when an execution error occurs.

The O toggle tells CB80 not to generate the relocatable object file. If a compiler error occurs, CB80 does not generate the .REL file.

The P toggle prints the program listing on the printer. CB80 sends a form-feed before printing the first page. CB80 prints the page number and the source filename at the top of each page.

The R toggle specifies which drive to place the .REL file on.

The S toggle places all information on program variables and line labels into the .REL file. The link editor uses the information to generate a .SYM file. You can use the .SYM file with the Digital Research Symbolic Instruction Debugger, SID™.

The T toggle lists the symbol table immediately following the source program listing.

The U toggle generates an error message if a variable name does not appear in an INTEGER, REAL, or STRING declaration. Use the U toggle to locate misspelled identifiers.

The V toggle places the source code line numbers into the .SYM file.

The W toggle changes the page width for a listing directed to the printer. Initially, the width is set to 80 columns. Enclose the new width in parentheses following the W. The width must be an unsigned integer.

```
CB80 TEST [W(70)]
```

The X toggle specifies a drive for the temporary work files. Normally, CB80 places the work files on the same drive as the source file. Enclose the new drive specification in parentheses following the X. The drive is specified by a single lower- or uppercase letter.

```
CB80 TEST [X(D)]
```

CB80 evaluates toggles from left to right. This means a subsequent directive can override any earlier one. In the following example, CB80 sends the listing to the printer.

```
CB80 TEST [BP]
```

In the following example, CB80 suppresses the listing.

```
CB80 TEST [PB]
```

End of Section 2

Section 3
The Link Editor, LK80

LK80 is a linkage editor designed specifically for use with CB80. LK80 combines relocatable object modules that CB80 generates with relocatable modules from the indexed library, CB80.IRL, into an executable program with optional overlay files. Your CBASIC Compiler product disk should contain the following two files:

```
LK80.COM
CB80.IRL
```

When linking a CBASIC program, both files must be on the default drive.

3.1 Linking Modules

LK80 converts a .REL file into an executable .COM file. During linking, LK80 automatically searches the default disk for the indexed library file. LK80 includes any library routines that the compiled program requires in the executable program.

Following a sign-on message, LK80 prints four messages that indicate the amount of space LK80 allocates for the program. Refer to Section 5 for an explanation of memory allocation. The following example shows the console display during linking:

```
-----------------------------------------------
LK80                                Version 1.x
Ser. No. 000-0000           Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-----------------------------------------------
code size:        1200 (0100-12FF)
common size:      0000
data size:        0166 (1300-1465)
symbol table space remaining:    124F
```

LK80 determines and displays the four values that follow the sign-on message as hexadecimal numbers. The values in parentheses are the memory location assigned to each area.

To complete the linking process, LK80 generates the executable program and a symbol location file with filetype .SYM. The executable program has the same filename as the first .REL file listed in the LK80 command line and has a filetype of .COM. You can specify a different filename for the executable program in the command line. You can use the .SYM file with the Digital Research symbolic debugging program, SID.

3.1.1 LK80 Command Lines

The command line starts LK80 and specifies the relocatable files to link. The following command line links the modules in a file named TEST.REL with the runtime subroutine library and generates an executable program file named TEST.COM. LK80 assumes a filetype of .REL if not specified otherwise.

```
LK80 TEST
```

You can rename a file in the LK80 command line using an equal sign. The following command line links the modules in the file named TEST.REL but generates an executable file named TESTPGM.COM.

```
LK80 TESTPGM=TEST
```

You can specify which drive holds the .REL file to link, and you can specify a drive for LK80 to write the executable file to. The following command line produces the same executable file as in the previous example, but LK80 links the TEST.REL file from drive D and writes the TESTPGM.COM file to disk A.

```
LK80 A:TESTPGM=D:TEST
```

LK80 can link any program that occupies less than 64K bytes of memory unless the length of symbols exhausts the space reserved for the symbol table. You can link several relocatable files into one executable program. However, when combining several files, only one file can contain executable statements. All other files must contain only multiple-line functions. In the following command line, TEST is the executable program, and ONE, TWO, and THREE contain multiple-line functions. LK80 links all four relocatable files into one executable program named TEST.COM.

```
LK80 TEST, ONE, TWO, THREE
```

You can specify a filetype other than .REL for files in the command line if the files are relocatable object files. LK80 aborts the linking process if any of the files are not relocatable object files. In the following command line, TEST.A is the executable program, and ONE.B, TWO.C, and THREE.D contain multiple-line functions. LK80 assumes all four files are relocatable object files and links them into one executable program named TESTPGM.COM.

```
LK80 TESTPGM=TEST.A,ONE.B,TWO.C,THREE.D
```

If you generate your own subroutine library using LIB86, you must specify the library in the LK80 command line. The following example links the library LIB1.IRL into TESTPGM, along with ONE.REL and TWO.REL.

```
LK80 TESTPGM=ONE,TWO,LIB1.IRL
```

Note that you must specify the .IRL filetype for libraries.

LK80 can link up to 60 relocatable files at one time. However, the total length of the command line cannot exceed 128 characters. In cases where a command line exceeds 128 characters, you can shorten filenames, or you can place the command line in a disk file. There is no limit on the length of a command line if LK80 reads it from a disk file.

Create the command line file using any text editor. Do not enter the characters LK80 in the disk file. List each relocatable object file as you would in an ordinary command line. You can place tab characters, carriage returns, and line-feeds anywhere in a command line file. Use the backslash to document large command line files. LK80 ignores all characters that follow a backslash on the same line. Then specify the disk file in an LK80 command line as shown in the following example:

```
LK80 $ CMDLINE.LIN
```

The preceding example tells LK80 to read the rest of the command line from a disk file named CMDLINE.LIN. The dollar sign must follow the LK80. At least one space must separate the dollar sign from the file specification. The command line file can have any filename and filetype.

3.1.2 LK80 Errors

LK80 reports two different types of errors. The first type includes mistakes such as improper command lines and out of memory conditions. LK80 indicates these errors with a literal message. Refer to Appendix C, Table C-1, for a complete listing of LK80 error messages and descriptions.

The second type, LK80 failures, should never occur during your experience with the CBASIC Compiler. LK80 indicates a failure with the following message. The N stands for an integer value.

```
LK80 FAILURE N
```

If the above error message occurs during linking of your CBASIC program, contact the Digital Research Technical Support Center. Please report the number and the circumstances under which the error occurs.

3.2 LK80 Toggles

LK80 toggles are single-letter directives that you specify in the LK80 command line. Once a toggle is set, it remains set through the entire linking process. Place the letters within brackets following the relocatable file specifications in the LK80 command line. Letters can be lower- or upper-case. The following examples show several ways to specify LK80 toggles:

```
LK80 TEST[Q]

LK80 TESTPGM=TEST,ONE,TWO,THREE[QL]

LK80 TEST [M,OB,L]

LK80 TEST [MOBL]

LK80 TEST [M] [OB] [L]
```

LK80 supports five toggles as listed in the following table.

Table 3-1. LK80 Toggles

Toggle	*Instruction*
L	Redirect console output from LK80 to the printer.
M	List all module names followed by an absolute starting address for each.
O	Write output files to a drive other than the default drive.
Q	Place all symbols beginning with a question mark into the .SYM file.
S	Save linking information for the overlays in a file for future short-links.

The L toggle directs all console messages output during linking to the printer.

The M toggle lists all module names and corresponding absolute addresses on the console. LK80 lists library modules first, followed by overlay modules. The M toggle displays a load map that you can use with the addresses provided in the CB80 listing to aid in debugging.

The O toggle directs all output files to a disk drive other than the default drive. Place the new drive specification immediately after the O. For example, the toggle OC writes all output files to drive C.

The Q toggle tells LK80 to place all symbols beginning with a question mark into the .SYM file. The toggle adds about 100 symbols to the .SYM file. If you do not specify the Q toggle, LK80 places only program defined symbols into the .SYM file.

The S toggle provides a way to short-link an overlay file independent of all other overlays in a program. Short-linking enables you to modify an overlay file and relink it into the original program without relinking the entire program. The S toggle saves linking information for the overlays in a disk file. LK80 saves the information in a file with the same name as the root program, but with a .LNK filetype. The following command line links the relocatable files TEST.REL, ONE.REL, and TWO.REL into an executable program TEST.COM and an overlay MSGS.OVL. The S toggle saves information for the overlay.

```
LK80 TEST,ONE,TWO,(MSGS)[S]
```

If you find an error in MSGS.OVL during execution of the program, you can correct the error in MSGS.BAS. Then recompile MSGS.BAS to generate MSGS.REL. Finally, you can short-link the new MSGS.REL into the TEST.COM program using the information saved in TEST.LNK. You do not have to repeat the whole link specified in the original command line. The following short-link command line relinks the new MSGS.REL overlay into TEST.COM. Note that you must not enclose MSGS in parentheses in a short-link command line.

```
LK80 TEST.LNK, MSGS
```

The following rules apply to short-linking with the S toggle:

- The new overlay must not require code, common, or data segment sizes larger than the segment sizes allocated in the original link.
- The new overlay must not reference library modules that are not included in the root. LK80 informs you if this occurs.
- You must place the saved .LNK file first in the LK80 short-link command line. LK80 assumes that all filenames after the .LNK filename constitute one overlay.
- LK80 can short-link only one overlay at a time.
- LK80 does not generate a symbol file during a short-link.

3.3 Producing Overlays

LK80 can produce overlay files that a CBASIC CHAIN statement can load into memory and execute. Overlay files have a .OVL filetype. LK80 overlay files preserve all variables declared in COMMON including variables stored dynamically, such as arrays and strings.

To generate an overlay, enclose the filename of the relocatable object file in parentheses within the LK80 command line. The following command line creates an executable file named TEST.COM and one overlay named ONE.OVL.

```
LK80 TEST(ONE)
```

The TEST.COM file is the root program. A CHAIN statement in the TEST.COM program loads the overlay ONE.OVL. When the root program chains to an overlay, the overlay actually replaces and overwrites the root in memory. This also occurs when an overlay chains to another overlay or back to the root.

The root program contains all library routines for the entire program. This reduces the size of an overlay file and the time required to load an overlay into memory.

LK80 can create up to 60 overlays at one time. However, the total number of relocatable modules in one link cannot exceed 60. The following command line generates an executable program named TESTPGM.COM and two overlays named ONE.OVL and TWO.OVL:

```
LK80 TESTPGM=TEST(ONE)(TWO)
```

You can combine several relocatable modules into one overlay. The following command line generates an executable program named TEST.COM and three overlays named A.OVL, C.OVL, and F.OVL:

```
LK80 TEST(A,B) (C,D,E) (F)
```

You can specify names for the overlay files in the command line. The following command line generates the TESTPGM.COM program and two overlays named FIRST.OVL and SECOND.OVL:

```
LK80 TESTPGM=TEST (FIRST=A) (SECOND=B,C)
```

3.4 Linking Assembly Language Routines

LK80 can link assembly language routines with relocatable modules that CB80 creates. You can use the Digital Research Relocating Macro Assembler, RMAC™, to convert your assembly language programs into relocatable modules that LK80 can link.

Assembly language routines linked into a CBASIC program must not contain initialized data. You can place all data that requires an initial value in the code segment. Refer to section 5.3 for information on parameter passing and returning values.

Note that using assembly language routines makes a program machine-dependent.

End of Section 3

Section 4
The Library

A library file consists of one or more relocatable modules. However, to be useful in the linking process, a library file must contain an index. The CBASIC Compiler provides an indexed library file for use with LK80 and a library manager utility program to create your own library files. Your CBASIC Compiler product disk should contain the following two files:

```
CB80.IRL
LIB.COM
```

4.1 CB80.IRL

The file CB80.IRL is an indexed library file that contains modules to allocate and release memory, determine available memory space, and perform arithmetic operations and input/output processing. All indexed library files have a .IRL filetype. An index precedes the group of modules and contains all the public symbols that are in each module. The index enables LK80 to determine which routines in CB80.IRL are required to create the executable program.

LK80 first reads the .REL files you specify in the command line. LK80 then searches the index of CB80.IRL for any symbols that remain unresolved. LK80 links only those modules from CB80.IRL that contain definitions of the unresolved symbols.

For example, if a module in one of your programs requires the square root subroutine, LK80 searches the index of the CB80.IRL file for the symbol ?RSQR. Assuming that this symbol is not defined anywhere in your program, LK80 links the module from CB80.IRL that contains the definition of ?RSQR. LK80 links any module from the indexed library that contains a required symbol definition.

4.1.1 Dynamic Storage Allocation Routines

The CBASIC Compiler indexed library file provides four routines for use in assembly modules that enable you to allocate and release memory, and to determine the amount of space that is available for allocation.

- The ?GETS routine allocates space. The routine requires that the number of bytes of memory to allocate pass in registers H and L. The maximum number of bytes the routine can allocate is 32,762. ?GETS returns a pointer to a contiguous block of memory in registers H and L. There is no restriction on what the allocated memory space can contain, if the adjacent space at either end of the allocated area is not modified.
- The ?RELS routine releases previously allocated memory. The routine requires that the address of the space to release passes in registers H and L. ?RELS does not return a value.
- The ?MFE routine returns the size of the largest contiguous area available for allocation using the ?GETS routine. The value returned is an integer placed in in registers H and L.
- The ?IFRE routine returns the total amount of unallocated dynamic memory. The returned value is an integer placed in registers H and L. A negative value indicates a number larger than 32,767.

4.1.2 Arithmetic Routines

The CBASIC Compiler indexed library file provides routines for signed integer multiplication and division for use in assembly modules.

- The ?MIDH routine multiplies the signed integer in registers D and E by the signed integer in registers H and L. The routine returns the result in registers H and L.
- The ?DIDH routine divides the signed integer in registers D and E by the signed integer in registers H and L. The routine returns the result in registers H and L.

4.2 The Library Manager Utility, LIB

The LIB.COM file is a versatile librarian program used to develop library files for use with LK80. LIB.COM can perform the following four tasks:

- concatenate a group of .REL files into a library
- create an indexed library, .IRL file
- select modules from a library
- print module names and public symbols from a library

The LIB.COM program supports three command line switches. A switch is a single lower- or upper-case letter that you specify in brackets following the first filename in the LIB command line.

- The I switch creates an indexed library, .IRL file.
- The M switch prints a listing of module names from a specified file.
- The P switch prints a listing of both module names and public symbols from a specified file.

4.2.1 LIB Manager Command Lines

You can concatenate a group of .REL files to unite modules that must execute in combination. Two separate modules might contain public functions that a third module needs for execution. Combining the three modules into one .REL file simplifies the LK80 command line that refers to them. The following librarian command line creates a file named TEST.REL by concatenating the files TEST1.REL, TEST2.REL, and TEST3.REL:

```
LIB TEST=TEST1,TEST2,TEST3
```

The librarian program only concatenates the three original files. The librarian does not modify the files in any other way.

Using the I switch, you can create an indexed library file for a group of modules that support the same types of applications, but are not interdependent for execution. Certain programs might require only one or two modules from a group. Generate an indexed library with the group of modules using LIB.COM. The following command line generates an indexed library named TEST.IRL from the three modules TEST1, TEST2, and TEST3:

```
LIB TEST[I]=TEST1,TEST2,TEST3
```

Specify the library name in the LK80 command line whenever the application program requires certain modules. LK80 searches the index of the library for the required routines and links the corresponding modules into the program. This procedure helps keep the executable program file as small as possible. You can specify up to ten indexed library files in an LK80 command line. LK80 processes library files in order of occurrence.

When you start the LIB.COM program, a sign-on message and some diagnostic messages display on the console. Using the M toggle, you can have LIB.COM generate a list of all module names in the specified files following the diagnostic messages. The following command line generates a list of all modules for the TEST.REL file:

```
LIB TEST[M]
```

Using the P toggle, you can have LIB.COM generate a list of all module names and public symbols in the specified files following the diagnostic messages. The following command line generates a list of all modules and public symbols for the TEST.REL file:

```
LIB TEST[P]
```

End of Section 4

Section 5
Machine-level Environment

To understand the CBASIC Compiler machine-level environment, you should have a working knowledge of CP/M and a familiarity with elementary computer architecture.

5.1 Memory Allocation

The operating system loads an executable CBASIC program into the CP/M Transient Program Area (TPA). Before a CBASIC program loads, the CP/M Console Command Processor (CCP) resides at the top of the TPA. The CBASIC program overwrites all of the CCP after it begins execution.

The following diagram shows memory allocation during execution of a compiled CBASIC program. The area extending from the base of memory at 0H to the base of the TPA at 100H is reserved for CP/M and remains fixed. The area extending from the top of the TPA to the top of memory at FFFFH is reserved for CP/M and remains fixed. When CP/M loads a CBASIC program, the memory available in the TPA is partitioned into six areas of varying size. The diagram shows the relative positions of the different areas in memory, but does not accurately represent relative sizes.

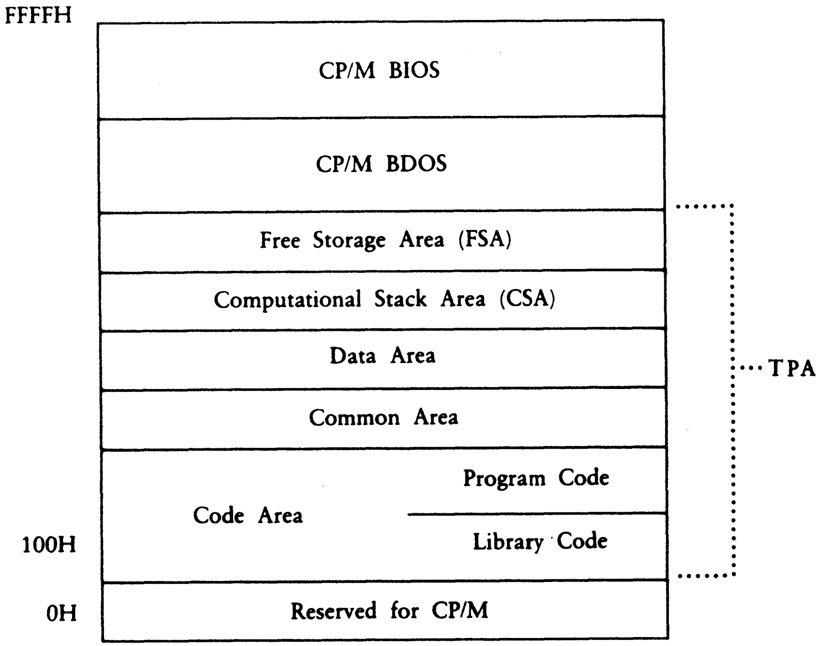


Figure 5-1. CP/M Memory Allocation

- The Code Area contains the actual computer instructions used during execution. The Code Area consists of two partitions: the Program Code and the Library Code. The Program Code section contains the root program. When you chain to an overlay, the overlay file overwrites the root program in this area. Likewise, when you chain back to the root programs, the root program overwrites the overlay file. The Library Code section contains the various routines from CB80.IRL and other indexed library modules that the program requires for execution.
- The Common Area contains all variables passed through COMMON statements to chained programs. The Common Area reserves eight bytes of storage space for each variable, regardless of data type. For array and string variables, the actual value is stored in the Free Storage Area. The value stored in the Common Area is an address in the FSA.
- The Data Area contains all variables that are not declared in COMMON. The Data Area is not preserved during chaining. The Data Area reserves two bytes for each integer and eight bytes for each real number. The Data Area stores the pointers that refer to strings and arrays in the FSA.
- The Computational Stack Area (CSA) is fixed at 100 bytes of memory. The CSA evaluates expressions and passes parameters to CBASIC predefined functions.
- The Free Storage Area (FSA) stores arrays, strings, and file buffers. Variably sized blocks of memory are allocated from the FSA as required and returned when no longer needed.

The starting and ending addresses for each partition in the TPA varies for different programs. Once allocated, however, the amount of memory each partition occupies remains fixed during program execution.

5.2 Internal Data Representation

CBASIC machine-level representation varies somewhat for real numbers, integers, strings, and arrays.

- REAL NUMBERS are stored in binary coded decimal (BCD) floating-point form. Each real number occupies eight bytes of memory storage space, as shown in Figure 5-2. The high-order bit in the first byte (byte 0) contains the sign of the number. The remaining seven bits in byte 0 contain a decimal exponent. Bytes 1 through 7 contain the mantissa. Two BCD digits occupy each of the seven bytes in the mantissa. The number's most significant digit is stored in byte 7, furthest from the exponent. The floating decimal point is always situated to the left of the most significant digit.

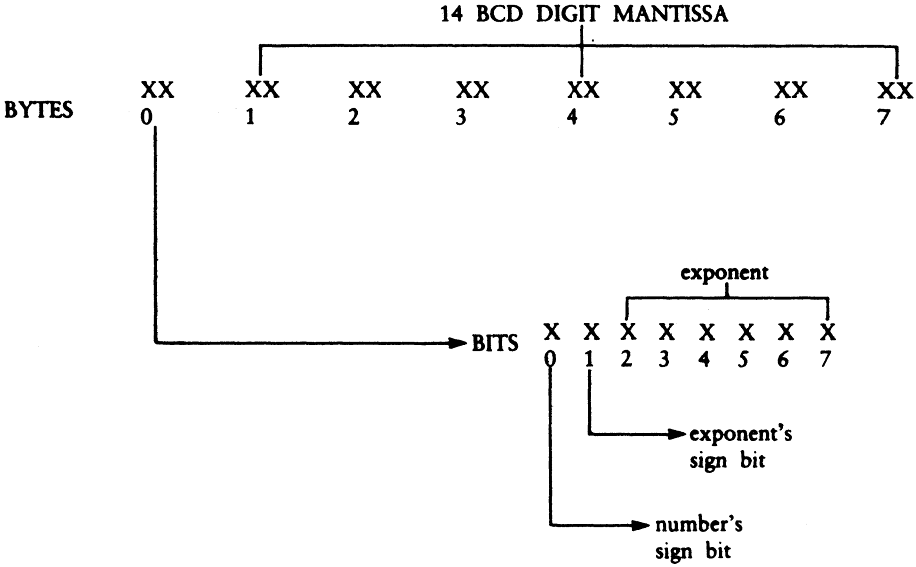


Figure 5-2. Real Number Storage

- INTEGERS are stored in two bytes of memory space with the low order byte first as shown in Figure 5-3. Integers are represented as 16-bit two's complement binary numbers. Integer values range from −32768 to +32767, inclusive.

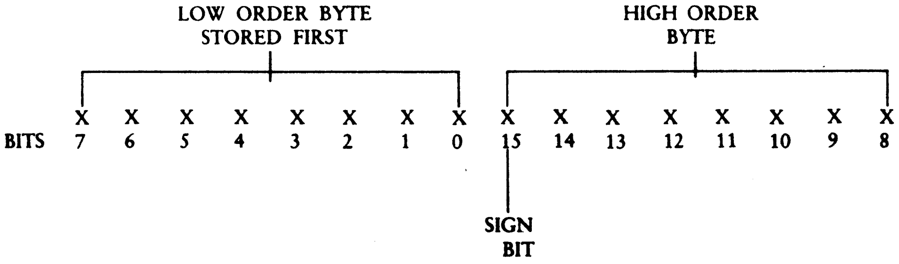

LOW ORDER BYTE STORED FIRST
HIGH ORDER BYTE
X X X X X X X X X X X X X X X X
BITS 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8
SIGN BIT

Figure 5-3. Integer Storage

- STRINGS are stored as a sequential list of ASCII representations. The length of a string is stored in the first two bytes followed by the actual ASCII values. The maximum number of characters in a string is 32,762. CBASIC Compiler allocates space in the Free Storage Area for strings. A pointer in the Data Area is an address in the FSA for the actual string.

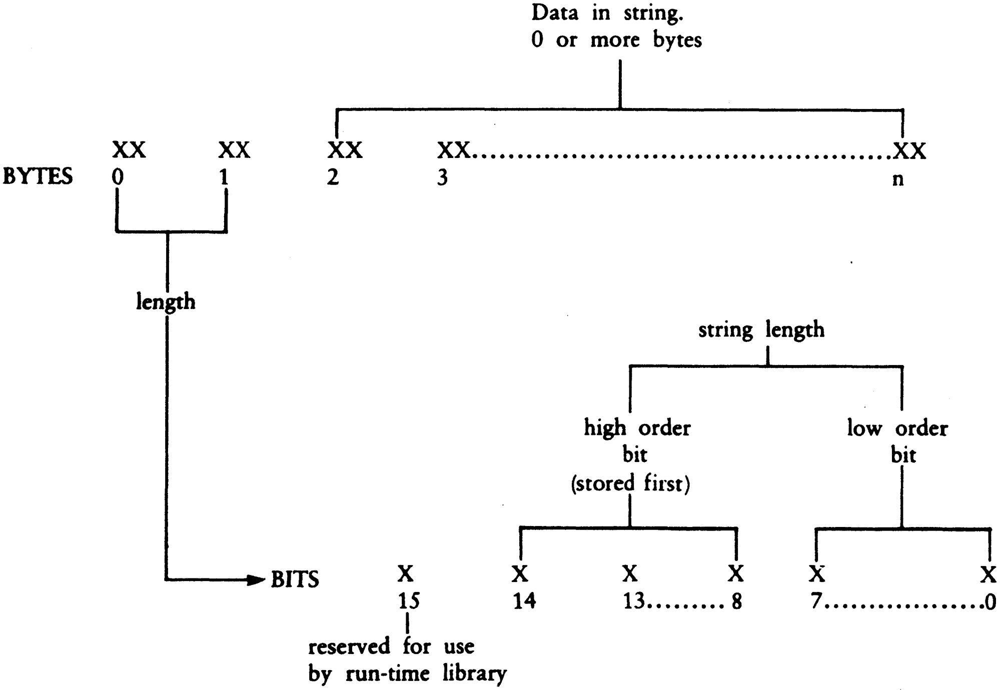


Figure 5-4. String Storage

Note: string lengths are stored high order and then low order. This is contrary to the normal 8080 convention for storing 16-bit quantities. The reserved bit, 15, is used to indicate that the string is temporary if the bit is a 1, and not temporary if it is a 0.

- ARRAYS, both numeric and string, are allocated space in the Free Storage Area as required. Eight bytes are reserved for each element of an array containing real numbers and two bytes for each element of an integer array.

At some point in a program, it might be necessary to free memory space allocated to arrays that are no longer needed in the program. Freeing numeric array space requires that you simply redimension the array to zero. However, freeing string array space is a two step process. First, you must set all string array elements to null. Set all string array elements equal to a string variable that has never been assigned a value. Use a variable such as NULL$. Be sure that NULL$ has never been assigned a value. Do not set NULL$ equal to "/". Second, you must redimension the string array to zero after assigning each element in the array to NULL$.

5.3 Parameter Passing and Returning Values

CBASIC Compiler passes all parameters on the hardware stack. When a program calls a routine, CBASIC places each parameter on the stack reading from left to right. The last entry on the stack is the return address. All values must conform to the format described in Section 5.2.

An assembly language routine can return integer, real, or string values to a CBASIC program. Before returning to the CBASIC program, all parameters passed on the stack must be removed and the stack pointer adjusted accordingly.

Integers return in registers H and L. Real numbers return using a pointer in registers H and L that points to an eight byte area containing the real value. The H and L registers contain the address of the exponent byte of the number being returned.

Strings return using a pointer in registers H and L. Strings must have been allocated using CBASIC Compiler dynamic storage management routines. The allocation bit of a returning string should be set to 1. This ensures that the space is initialized when no longer needed.

5.4 .REL File Format

CB80 and the Digital Research Relocating Macro Assembler (RMAC) create .REL files. A .REL file contains information encoded in a bit stream. You can interpret the information as described in the following list:

- If the first bit is a 0, the next 8 bits load according to the value of the location counter.
- If the first bit is a 1, the next two bits can be interpreted as follows:

  00 Special link item.

  01 Program relative. The next 16 bits load following an offset from the program segment origin.

  10 Data relative. The next 16 bits load following an offset from the data segment origin.

  11 Common relative. The next 16 bits load following an offset from the origin of the currently selected common block.

A special item consists of the following:

- 4-bit control field that selects one of 16 special link items described in Table 5-1.
- An optional value field that consists of a 2-bit address type field and a 16-bit address field. The address type field can be interpreted as follows:

  00 - absolute
  01 - program relative
  10 - data relative
  11 - common relative

- an optional name field consisting of a 3-bit name count followed by the name in 8-bit ASCII characters.

Table 5-1. Special Link Items

Control Field	*Meaning*
A name field follows the next five Link Items:	
0000	Entry symbol. This module describes the symbol indicated in the name field.
0001	Currently unassigned.
0010	Program name. This is the name of the relocatable module.
0011	Name field. Gives the name of default library file to use. LK80 assumes a filetype of .IRL.
0100	Currently unassigned.
A value field and a name field follow the next four link items:	
0101	Define common size. The value field determines the amount of memory to reserve for the common block indicated in the name field.
0110	Chain external. The value field contains the head of a chain that ends with an absolute 0. The value of the external symbol described in the name field replaces each element of the chain.
0111	Define entry point. The value field defines the value of the symbol in the name field. This link item puts local symbols in .REL files.
1000	Currently unassigned.

Table 5-1. (continued)

Control Field	*Meaning*
A value field follows the next six link items:	
1001	External plus offset. The value in the value field after all chains are processed must offset the following two bytes in the current segment.
1010	Define data size. The value field contains the number of bytes in the data segment of the current module.
1011	Set location counter. Set the location counter to the value indicated in the value field.
1100	Chain address. The value field contains the head of a chain that ends with an absolute 0. The current value of the location counter replaces each element of the chain.
1101	Define program size. The value field contains the number of bytes in the code segment of the current module.
1110	End module. Defines the end of the current module. If the value field contains a value other than absolute 0, the value is the start address for the linking program. The next item in the file will start at the next byte boundary.
The last item has no value field or name field:	
1111	End file. Follows the last module item for the last module in the file.

5.5 .IRL File Format

The CBASIC Compiler librarian utility, LIB.COM, creates .IRL files. An .IRL file consists of three parts: the header, the index, and the relocatable section. The header contains 128 bytes allocated as follows:

- byte 0 - extent number of first record of relocatable section
- byte 1 - record number of first record of relocatable section
- bytes 2 to 127 - currently unassigned

The index consists of entries that correspond to the entry symbol items listed in the relocatable section. Entries use the following form:

e	r	b	c1	c2	. . .	cn	d

e = Extent offset from start of relocatable section to start of module.

r = Record offset from start of extent to start of module.

b = Byte offset from start of record to start of module.

c1 - cn = Name of symbol.

d = End of symbol delimiter (0FEH).

When c1 equals 0FFH, the index terminates and the remainder of the record is not used.

The relocatable section contains relocatable object code as described in the Section 5.4.

End of Section 5

Appendix A
Implementation Dependent Values

The following implementation dependent values apply to CB80 version 1 for use with CP/M, versions 2 and 3, and MP/M-80™, versions 1 and 2:

Table A-1. Implementation Dependent Values

Parameter	*Value*	*Minimum*
Initial page width for compiler output	80	—
Initial page length for compiler output	66	—
Maximum number of errors maintained	95	—
Maximum nesting of include	6	4
Maximum number of formal parameters	15	15
Maximum number of subscripts in an array	15	15
Maximum unique identifier length	50	31
Maximum number of characters in string constant	255	255
Maximum length of Global and External names	6	6
Maximum nesting of FOR loops	13	—
Maximum nesting of WHILE loops	39	—
Number of files that can be open at one time	20	12
File buffer size in bytes	128	—

The minimum values are the minimum that are used in any CB80 implementation.

The following extensions exist in CB80, versions 1.3 and 1.4, to provide compatibility with CBASIC version 2. Note that future versions of CB80 might not support these extensions.

- The LPRINTER statement accepts a WIDTH option to be consistent with CBASIC. The width is ignored.
- Integer and real data is initialized to 0; strings are initialized to null strings. See Section 5.2.
- The INPUT prompt string can be any expression; the first operand must be a string constant.
- An OPEN or CREATE statement accepts a RECS field for compatibility with CBASIC. The expression is ignored.
- You can use the reserved words LT, GT, LE, GE, EQ, and NE in place of the relational operators <, >, <=, >=, =, and < >.
- CB80 supports the following form of an IF statement:

  IF *expression* THEN *label*

  but the *label* must be a numeric label.

End of Appendix A

Appendix B
Compiler Error Messages

The compiler prints the following messages when a file system error or memory space error occurs. In each case, control returns to the operating system.

Table B-1. File System and Memory Space Errors

Error	*Meaning*
COULD NOT OPEN FILE:*filename*	The filename following the message cannot be located in the file system directory.
%INCLUDES NESTED TOO DEEP:*filename*	The filename following the message occurred in an %INCLUDE directive that exceeds the allowed nesting of %INCLUDE directives.
SYMBOL TABLE OVERFLOW	The available memory for symbol table space has been exceeded. Break the program into modules or use shorter symbol names.
INVALID FILE NAME:*filename*	The filename is not valid for your operating system.
DISK READ ERROR	The operating system reports a disk read error.

Table B-1. (continued)

Error	*Meaning*
CREATE ERROR: *filename*	The file cannot be created. Normally this means there is no directory space on the disk.
DISK FULL	The operating system reports that no additional space is available to write temporary or output files. The directory may be full or the disk is out of space.
INVALID COMMAND LINE	The command line is incorrect. The compiler prints a greater-than sign, >, one blank space, and all command line characters beginning with the first character in error. If no characters remain in the command line when an error occurs, the compiler does not print the > or the space.
MISSING SOURCE FILE NAME	The command line processor reports that you did not specify a source file.
CLOSE OR DELETE ERROR	The operating system reports that it cannot close a file. This occurs if diskettes are switched during compilation.

If the compiler detects an internal failure, the following error message appears:

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE XXXX
```

where XXX is a three digit number. Please advise Digital Research of the error and the circumstances under which it occurs.

The following error messages indicate a compilation error occurred during compilation of a program. Compilation continues after the error is recorded. Compilation error messages display within the source code listing.

Table B-2. Compilation Error Messages

Error	*Meaning*
1	Invalid character in the source program. The character is ignored.
2	Invalid string constant. The string is too long or contains a carriage return.
3	Invalid numeric constant. An integer constant of zero is assumed.
4	Undefined compiler directive. This source line is ignored.
5	The %INCLUDE directive is missing a filename. This source line is ignored.
6	Statements found after an END.
7	Not used.
8	Variable used without being defined, and the U toggle used during compilation.
9	The DEF statement is not terminated by a carriage return. A carriage return is inserted.
10	A right parenthesis is missing from the parameter list. A right parenthesis is inserted.
11	A comma is missing in the parameter list. A comma is inserted.
12	An identifier is missing in the parameter list.
13	The same name is used twice in a parameter list.
14	A DEF statement occurs within a multiple-line function. Multiple-line functions cannot be nested. The statement is ignored.

Table B-2. (continued)

Error	*Meaning*
15	A variable is missing.
16	The function name is missing following the keyword DEF. The DEF statement is ignored.
17	A function name is used previously. The DEF statement is ignored.
18	A FEND statement is missing. A FEND is inserted.
19	There are too many parameters in a multiple line function.
20	Inconsistent identifier usage. An identifier cannot be used as both a label and a variable.
21	Additional data exists in the source file following an END statement. This is the logical end of the program.
22	Data statements must begin on a new line. The remainder of this statement is treated as a remark.
23	There are no variables or function names in a declaration statement; or, a reserved word appears in the list of identifiers.
24	A function name appears in a declaration within a multiple-line function other than the multiple-line function that defines this function name.
25	A function call has incorrect number of parameters.
26	A left parenthesis is missing. A left parenthesis is inserted.
27	Invalid mixed mode. The type of the expression is not permitted.
28	Unary operator cannot be used with this operand.
29	Function call has improper type of parameter.

Table B-2. (continued)

Error	*Meaning*
30	Invalid symbol follows a variable, constant, or function reference.
31	This symbol cannot occur at this location in an expression. The symbol is ignored.
32	Operator is missing. Multiplication operator inserted.
33	Invalid symbol encountered in an expression. The symbol is ignored.
34	A right parenthesis is missing. A right parenthesis is inserted.
35	A subscripted variable is used with the incorrect number of subscripts.
36	An identifier is used as a simple variable with previous usage as a subscripted variable.
37	An identifier is used as a subscripted variable with previous usage as an unsubscripted variable.
38	A string expression is used as a subscript in an array reference.
39	A constant is missing.
40	Invalid symbol found in declaration list. The symbol is skipped.
41	A carriage return is missing in a declaration statement. A carriage return is inserted.
42	Comma is missing in declaration list. A comma is inserted.
43	A common declaration cannot occur in a multiple-line function. The statement is ignored.
44	An identifier appears in a declaration twice in the main program or within the same multiple-line function.

Table B-2. (continued)

Error	*Meaning*
45	The number of dimensions specified for an array exceedᵉ the maximum number allowed. A value of one is used. This might generate additional errors in the program.
46	Right parenthesis is missing in the dimension specification within a declaration. A right parenthesis is inserted.
47	The same identifier is placed in COMMON twice.
48	An invalid subscripted variable reference encountered in a declaration statement. An integer constant is required. A value of 1 is used.
49	An invalid symbol found following a declaration, or the symbol in the first statement in the program is invalid. The symbol is ignored.
50	An invalid symbol encountered at the beginning of a statement or following a label.
51	An equal sign is missing in assignment. An equal sign is inserted.
52	A name used as a label previously used at this level as either a label or variable.
53	Unexpected symbol follows a simple statement. The symbol is ignored.
54	A statement is not terminated with a carriage return. Text is ignored until the next carriage return.
55	A function name is used in the left part of an assignment statement outside of a multiple-line function. Only when the function is being compiled can its name appear on the left of an assignment statement.
56	A predefined function name is used as the left part of an assignment statement.
57	In an IF statement, a THEN is missing. A THEN is inserted.

Table B-2. (continued)

Error	*Meaning*
58	A WEND statement is missing. A WEND is inserted.
59	A carriage return or colon is missing at the end of a WHILE loop header.
60	In a FOR loop header the index is missing. The compiler skips to end of this statement.
61	In a FOR loop header, a TO is missing. A TO is inserted.
62	An equal sign is missing in a FOR loop header assignment. An equal sign is inserted.
63	Carriage return or colon is missing at end of FOR loop header.
64	A NEXT statement is missing. A NEXT is inserted.
65	Not used.
66	The variable that follows NEXT does not match the FOR loop index.
67	NEXT statement encountered without a corresponding FOR loop header.
68	WEND statement encountered without a corresponding WHILE loop header.
69	FEND statement encountered without a corresponding DEF statement. This error indicates that the end of the source program was detected while within a multiple-line function.
70	The PRINT USING string is not of type string.
71	A delimiter is missing in a PRINT statement. A comma is inserted.
72	A semicolon is missing in an INPUT prompt. A semicolon is inserted.

Table B-2. (continued)

Error	*Meaning*
73	A delimiter is missing in an INPUT statement. A comma is inserted.
74	A semicolon is missing following a file reference. A semicolon is inserted.
75	The prompt in an INPUT statement is not of type string.
76	In an INPUT LINE statement, the variable following the keyword LINE is not a string variable.
77	In an INPUT statement, a comma is missing between variables. A comma is inserted.
78	The keyword AS is missing in an OPEN or CREATE statement. AS is inserted.
79	The filename in an OPEN or CREATE statement is not a string expression.
80	A delimiter is missing in a READ statement. A comma is inserted.
81	In a GOTO, GOSUB, or ON statement, a label is missing. This token can be an identifier previously used as a variable.
82	The label in a GOTO statement is not defined. If the label is used in a function, it must be defined in that function.
83	A delimiter is missing in a file READ statement. A comma is inserted.
84	In a READ LINE statement, the variable following the keyword LINE is not a string variable.
85	The label in an IF END statement is not defined.
86	A pound sign, #, is missing in an IF END statement. A pound sign is inserted.

Table B-2. (continued)

Error	*Meaning*
87	A THEN is missing in an IF END statement. A THEN is inserted.
88	In a PRINT statement, the semicolon is missing following a using string. A semicolon is inserted.
89	In an ON statement, a GOTO or GOSUB is missing. A GOTO is assumed.
90	The index of a FOR loop header is of type string. The index must be an integer or real value.
91	The expression following the keyword TO in a FOR loop header is of type string. The expression must be an integer or real value.
92	The expression following the keyword STEP in a FOR loop header is of type string. The expression must be an integer or real value.
93	A variable in a DIM statement is defined previously as other than a subscripted variable.
94	An identifier is missing as an array name in a DIM statement. The entire statement is ignored.
95	A left parenthesis is missing in a DIM statement. A left parenthesis is inserted.
96	A right parenthesis is missing in a DIM statement. A right parenthesis is inserted.
97	The maximum number of dimensions allowed with a subscripted variable is exceeded.
98	A comma is missing in a POKE statement. A comma is inserted.
99	The index of a FOR loop header is not a simple variable.
100	In a CALL statement, a multiple-line function name is missing.

Table B-2. (continued)

Error	*Meaning*
101	A file PRINT statement is terminated with a comma or se-nicolon.
102	A DIM statement is missing for this subscripted variable.
103	A comma is missing in the label list associated with an ON GOTO or ON GOSUB statement. A comma is inserted.
104	A GOTO is missing in an ON ERROR statement. A GOTO is inserted.
105	A comma is missing in a PUT statement. A comma is inserted.
106	The expression in an IF statement is of type string. An integer or real expression is required.
107	The expression in a WHILE loop header is of type string. An integer or real expression is required.
108	In an OPEN or CREATE statement, the filename is missing.
109	In an OPEN or CREATE statement, the expression following the reserved word AS is missing.

Table B-2. (continued)

Error	*Meaning*
110	A multiple-line function calls itself.
111	A semicolon separates expressions in a file PRINT statement. A comma is substituted for the semicolon.
112	A file PRINT statement does not have an expression list.
113	A TAB function is used in a file PRINT statement expression list.
114	Label used as a variable in a list of expressions.
115	A GO not followed by a TO or SUB. GOTO is assumed.
116	An OPEN or CREATE statement specifies both UNLOCKED and LOCKED access control.
117	A CREATE statement uses the READ ONLY access control.

End of Appendix B

Appendix C
LK80 Error Messages

LK80 prints the following messages to indicate the occurrence of an error during linking. Control returns to the operating system after the message is displayed.

Table C-1. LK80 Error Messages

Message	*Meaning*
`Unresolved external:` *symbol name*	You defined the symbol name as external but neglected to define the symbol as public.
`Out of Directory Space`	LK80 ran out of directory space while writing the root, overlay, symbol, or environment file.
`Disk Full`	LK80 ran out of disk space while writing the root, overlay, symbol, or environment file.
`Multiple Definition:` *symbol name*	You defined a symbol name more than once.
`Too many overlays`	You specified more than 60 overlays in the command line.
`Too many modules`	You specified more than 60 modules in the command line.

Table C-1. (continued)

Message	*Meaning*
`Symbol table overflow`	There is not enough memory for the symbol table.
`Cannot open source file`	A source file specified in the command line cannot be opened.
`Too many library files`	You cannot specify more than 10 indexed library files in the command line.
`Too many library modules`	You cannot extract more than 150 modules from all indexed library files.
`Index too big`	A library file index cannot exceed 16K.
`Too many external-plus-offsets`	The table that saves external-plus-offsets has overflowed. References to offsets from external symbols usually occur in assembly language programs.
`Code size exceeded. Short link aborted.`	The new overlay cannot require a code segment larger than the code segment in the original full link.
`Data size exceeded. Short link aborted.`	The new overlay cannot require a data segment larger than the data segment in the original full link.

Table C-1. (continued)

Message	*Meaning*
Common size exceeded. Short link aborted.	The new overlay cannot require a common segment larger than the common segment in the original full link.
Root has no entry point.	You did not specify the root program in the command line or your root program does not contain executable statements.
No entry point defined for overlay: *overlay*	The overlay file specified in the message does not contain executable statements.
Not enough memory	There is not enough memory for LK80 to complete linking the modules specified in the command line.
Cannot close file: *file*	LK80 cannot complete linking because it cannot close the module specified in the message.
Expected module name	You did not specify a module name in the command line.
Toggle not supported	You specified an invalid toggle letter in the command line.
Expected] at end of toggle definition	You omitted a closing square bracket in a command line toggle definition.

Table C-1. (continued)

Message	*Meaning*
`Unexpected ( ?`	You entered a left parenthesis without a matching right parenthesis in the command line, indicating an incomplete overlay specification.
`Unexpected ) ?`	You entered a right parenthesis without a matching left parenthesis in the command line, indicating an incomplete overlay specification.
`Invalid or unexpected character`	You entered a character in the command line that LK80 does not recognize or did not expect at a certain position in the command line.
`Module name or type too long`	Module names cannot exceed 8 characters and types cannot exceed 3 characters.
`Can only specify output name on first module`	Only one module name can precede the equal sign in a command line. If you do not use the equal sign, the first module listed becomes the name of the output file.
`Multiple entry points in:` *filespec*	More than one file specified in the command line contains executable statements. The file specified in the message contains executable statements.

End of Appendix C

Appendix D
Execution Error Messages

The following warning message might be printed during execution of a CB80 program:

```
IMPROPER INPUT - REENTER
```

This message occurs when the fields you enter from the console do not match the fields specified in the INPUT statement. Following this message, you must reenter all values required by the input statement.

Execution errors cause a two-letter code to be printed. The following table contains valid CB80 error codes.

If an error occurs with a code consisting of an asterisk followed by a letter, such as *R, a CB80 library has failed. Please notify Digital Research of the circumstances under which the error occurred.

Table D-1. CB80 Error Codes

Code	*Error*
AC	The argument in an ASC function is a null string.
BN	The value following the BUFF option in an OPEN or CREATE statement is less than 1 or greater than 128.
CE	The file being closed cannot be found in the directory. This occurs if the file has been changed by the RENAME function.
CM	The file specified in a CHAIN statement cannot be found in the selected directory. If no filetype is present, the compiler assumes a type of OVL.
CT	The filetype of the file specified in a CHAIN statement is other than COM or OVL.

Table D-1. (continued)

Code	*Error*
CU	A CLOSE statement specifies a file identification number that is not active.
DF	An OPEN or CREATE statement uses a file identification number that is already used.
DU	A DELETE statement specifies a file identification number that is not active.
DW	The operating system reports that there is no disk or directory space available for the file being written to, and no IF END statement is in effect for the file identification number.
DZ	Division by zero is attempted.
EF	Attempt to read past the end-of-file, and no IF END statement is in effect for the file identification number.
ER	Attempt to write a record of length greater than the maximum record size specified in the OPEN or CREATE statement for this file.
EX	Indicates MP/M II extended error.
FR	Attempt to rename a file to a filename that already exists.
FU	Attempt to access a file that was not open.
IF	A filename in an OPEN or CREATE statement or with the RENAME function is invalid for your operating system.
IR	A record number of zero is specified in a READ or PRINT statement.
LN	The argument in the LOG function is zero or negative.
ME	The operating system reports an error during an attempt to create or extend a file. Normally, this means the disk directory is full.

Table D-1. (continued)

Code	*Error*
MP	The third parameter in a MATCH function is zero or negative.
NE	A negative value is specified for the operand to the left of the power operator.
NF	A file identification is less than 1 or greater than the maximum number allowed. See Appendix E.
NN	An attempt to print a numeric expression with a PRINT USING statement fails because there is not a numeric field in the USING string.
NS	An attempt to print a string expression with a PRINT USING statement fails because there is not a string field in the USING string.
OD	A READ statement is executed but there are no DATA statements in the program, or all data items in all the DATA statements have been read.
OE	Attempt to OPEN a file that does not exist, and for which no IF END statement is in effect.
OF	An overflow occurs during a real arithmetic calculation.
OM	The program runs out of dynamically allocated memory during execution.
RB	Random access is attempted to a file activated with the BUFF option specifying more than one buffer.
RE	Attempt to read past the end of a record in a fixed file.
RU	A random read or print is attempted to a stream file.

Table D-1. (continued)

Code	*Error*
SL	A concatenation operation results in a string greater than the maximum allowed string length.
SQ	Attempt to calculate the square root of a negative number.
SS	The second parameter of a MID$ function is zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ is negative.
TL	A tab statement contains a parameter less than 1.
UN	A PRINT USING statement is executed with a null edit string, or the backslash escape character, \, is the last character in an edit string.
WR	Attempt to write to a stream file after it is read, but before it is read to the end-of-file.

End of Appendix D

Appendix E
LIB Error Messages

The following table presents the LIB.COM program error messages and descriptions.

Table E-1. LIB Error Messages

Message	*Meaning*
CANNOT CLOSE	LIB cannot close the output file. The diskette might be write-protected.
DIRECTORY FULL	There is no directory space for the output file.
DISK READ ERROR	LIB cannot read the specified file.
DISK WRITE ERROR	LIB cannot write the specified file; probably due to a full diskette.
FILE NAME ERROR	The form of a source filename is invalid.
NO FILE	LIB cannot find the file that is specified in the command line.
NO MODULE	LIB cannot find the module that is specified in the command line.
SYNTAX ERROR	You used an incorrect command line to start the LIB program.

End of Appendix E

Index

DIGITAL
RESEARCH™

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München

CBASIC® Compiler
Language
Reference Manual

DIGITAL
RESEARCH™

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München

CBASIC® Compiler
Language
Reference Manual

COPYRIGHT

Copyright © 1981, 1982 by Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

This manual is, however, tutorial in nature. Thus, the reader is granted permission to include the example programs, either in whole or in part, in his own programs.

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CBASIC, CP/M, CP/M-86, and CP/NET are registered trademarks of Digital Research. CB80, CB86, Concurrent CP/M-86, MP/M, MP/M II, and MP/M-86 are trademarks of Digital Research. Intel is a registered trademark of Intel Corporation. Zilog and Z80 are registered trademarks of Zilog, Inc.

The *CBASIC Compiler Language Reference Manual* was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

First Edition: September 1981
Second Edition: March 1982
Third Edition: December 1982

Foreword

CBASIC® is a comprehensive and versatile programming language for developing professional microcomputer software. Software developers worldwide have selected CBASIC for its capacity to quickly produce reliable, maintainable programs in a structured programming environment. CBASIC combines the power of a structured, high-level language with the simplicity of BASIC to provide a serious development tool that is easy to learn and easy to use.

The CBASIC® Compiler is a compiler version of the CBASIC programming language. The CBASIC Compiler is available for both 8-bit and 16-bit operating systems. Use the *CBASIC Compiler Language Reference Manual* with either version.

- The 8-bit version, CB80™, runs under CP/M®, MP/M™, and CP/NET® operating systems for microcomputers based on the Intel® 8080, 8085, or Zilog® Z80® microprocessor.
- The 16-bit version, CB86™, runs under the CP/M-86®, MP/M-86™,or Concurrent CP/M-86™ operating systems for computers based on the Intel 8086, 8088 family of microprocessors.

The *CBASIC Compiler Language Reference Manual* is for readers familiar with conventional BASIC terminology and programming concepts. The manual defines the structure, statements, and functions of the CBASIC language in Sections 1 through 4. Section 5 covers input and output, including the use of disk files. The *CBASIC Compiler (CB80) Language Programming Guide* and the *CBASIC Compiler (CB86) Language Programming Guide* provide in-depth discussions of the compiler, link editor, and library file for the respective versions of the CBASIC Compiler.

Programs written in other versions of CBASIC maintain compatibility with the CBASIC Compiler. You can convert existing CBASIC programs to the CBASIC Compiler with few modifications. The result is much faster execution and additional flexibility using assembly language routines. Appendix C explains the language enhancements made to implement the CBASIC Compiler version.

Digital Research is interested in your comments on programs and documentation. Please use the Software Performance Reports enclosed in each product package to help us provide you with better software products.

Table of Contents

Table of Contents (continued)

Appendixes

Table of Contents (continued)

List of Tables

List of Figures

Section 1
Introduction to CBASIC Compiler

1.1 CBASIC Compiler Components

The CBASIC Compiler system has three main components: a compiler, a link editor, and a library.

- The compiler translates CBASIC source code into relocatable machine code. Source programs default to a .BAS filetype unless otherwise specified. The compiler generates .REL files.
- The link editor combines relocatable object modules into an executable core-image file with optional overlays. The link editor generates executable files of type .COM for the 8-bit microprocessor family, and type .CMD for the 16-bit microprocessor family.
- The library provides relocatable modules that allocate memory, release memory, determine available space, and perform arithmetic operations and input/output processing.

1.2 Program Structure

CBASIC has features found in other high-level languages, such as structured control statements, functions, complex expressions, labels, data declarations, and a variety of data types. Other CBASIC features are parameter passing, local and global variables, easy access to the operating system, and chaining between programs.

CBASIC requires no line numbers and allows you to use commas, spaces, and tabs freely to make your programs more readable. You must use a statement number or label only when referencing a statement or module from another location in the program. CBASIC allows literal identifiers, integers, decimal fractions, and exponential numbers as labels, as in the following examples.

```
CALC.TOTAL: PRINT A% + B% + C%

1  PRINT "THESE ARE VALID LINE NUMBERS"

0  INPUT "ENTER A NUMBER:";N

100  GO TO 100.0

100.0   END

21.543  A$ = NAME$

7920E12  Y = 2.0 * X
```

Numeric statement labels do not have to be in order. The compiler treats the labels as strings of characters, not as numeric quantities. For example, the two labels 100 and 100.0 are distinct CBASIC statement labels. Only the first thirty-one characters are meaningful for distinguishing one label from another.

CBASIC statements can span more than one physical line. Use the backslash character, \, to continue a CBASIC statement on the next line. The compiler ignores any character that follows a backslash on the same line, thus providing a method of program documentation. The backslash does not work as a continuation character if used in a string constant. The following example demonstrates the continuation character:

```
IF X = 3  THEN \
   PRINT "THE VALUES ARE EQUAL" \
ELSE \
   GOSUB 1000
```

In most cases, you can write multiple statements on the same line. Use a colon, :, to separate each command that appears on one line. However, the statements DIM, IF, DATA, END, and declaration statements cannot appear on one line with other statements. The following example demonstrates multiple statements on one line:

```
PRINT TAB(10);"X": READ #1;NAME$: GOTO 1000
```

Use comments or remarks freely to document your programs. The REM statement allows unlimited program documentation. Use spaces freely to enhance readability of your programs. Comments, long variable names, and blank spaces do not affect the size of your compiled program.

End of Section 1

Section 2
Identifiers, Numbers, and Expressions

CBASIC has three data types: integers, real numbers, and strings. CBASIC also supports dynamic, multidimensional arrays of all three data types. Each data type has a distinct form for identifiers. Numeric constants have several forms.

CBASIC has a large set of operators for building expressions with variables, constants, and functions of the three data types. By converting from one type to another, where necessary, CBASIC allows you to mix real and integer numbers in most expressions.

2.1 Identifiers

An identifier is a string of characters that names an element in a program. Identifiers specify variable names and user-defined function names. An identifier can be any length. Only the first thirty-one characters are meaningful for distinguishing one name from another. The first character must be a letter or a question mark, the remaining characters can be letters, numerals, or periods. The last character determines a default data type for the individual identifier. Declarations can override the default data type (see Section 2.2).

- Identifiers ending with $ represent strings.
- Identifiers ending with % represent integers.
- Identifiers without a $ or % represent real numbers.

The compiler converts lower-case letters to upper-case unless you set toggle D.

The following are examples of valid CBASIC identifiers.

```
A%

NEW.SUM

file12.name$

Payroll.Identification.Number%
```

2.2 Declarations

Declarations enable you to specify the data type for a group of variables or function names. A declaration statement consists of a data type keyword followed by a space and a list of identifiers delimited with commas. The data type keywords are INTEGER, REAL, and STRING. The following are examples of valid declaration statements.

```
INTEGER I,J,LOOP.COUNT

REAL A, AMOUNT.DUE, C

STRING NAME, PART.DESCRIP
```

The three preceding examples, listed in a program, form a declaration group or block. A declaration group can contain blank lines, REM statements, COMMON statements, and DATA statements.

You can declare common variables with the COMMON statement allowing two or more programs to share data. Refer to the Programming Guide for instructions on chaining. The following COMMON statement declares three common variables.

```
COMMON X, Y%, Z$
```

You can list the same variable in a declaration statement and a COMMON statement as follows.

```
STRING X

COMMON X, Y(1)

REAL   Y(1)
```

You can place any number of COMMON statements in a declaration group. However, you cannot use COMMON statements in the declaration group of a multiple-line function.

To use an array identifier in a declaration statement, place the number of subscripts in parentheses after the array name, as shown in the following examples.

```
INTEGER COORDINATES(2), Y(1)
```

```
COMMON NAMES$(1)
```

The COORDINATES array is a two-dimensional integer array. Y is a one-dimensional integer array, and NAMES$ is a one-dimensional string array. For more information about arrays, see the DIM statement in Section 3.

2.3 Strings

Strings can contain ASCII characters or binary data. Some editors can even place control characters in strings. Delimit string constants with quotation marks. Zero or more characters placed between a pair of quotation marks make up a single string constant. A string constant must fit on a single physical line. A pair of adjacent quotation marks represents a null string. A null string contains no characters. The backslash, \, has no special meaning inside a string constant. You can embed quotation marks in a string constant by using two quotes to represent one, as in the following example.

The string constant

```
"""Hello,""  said Tom."
```

stores internally as the string:

```
"Hello," said Tom.
```

String constants must fit on one physical line. This means that a string constant cannot contain a carriage return, and cannot exceed 255 characters. String variables are more flexible. Internally, a string can have from 0 to 32,767 characters. Each character takes up one byte. The first two bytes in the string contain the length of the string. To build long strings, use string expressions (described later in this section), and string functions (described in Section 3).

The following are examples of valid CBASIC string constants:

```
"July 4, 1776"

"Enter your name please:"

"""\"" has no special meaning inside a string."
```

`""` (represents the null string)

2.4 Numbers

CBASIC supports two types of numeric quantities: real and integer. You can write a real constant in either fixed format or exponential notation. In both cases, the real number contains from one to fourteen digits, a sign, and a decimal point. In exponential notation, the exponent is of the form Esdd, where s, if present, is a valid sign, +, -, or blank, and where dd is one or two valid digits. The sign is the exponent sign. Do not confuse the exponent sign with the optional sign of the mantissa. The numbers range from 1.0E-64 to 9.9999999999999E62. Although CBASIC maintains only fourteen significant digits, you can include more digits in a real constant. Real constants round down to fourteen significant digits. The following are examples of real numbers.

```
25.00

-4529.78
```

`1.5E+3` (equals 1500.0)

`1.5E-3` (equals .0015)

CBASIC treats a constant as an integer if the constant does not contain an embedded decimal point, is not in exponential notation, and ranges from -32,768 to +32,767. The following are examples of integers.

```
1

-99

4E2

32767
```

You can express integer constants as hexadecimal or binary constants. The letter H terminates a hexadecimal constant. The letter B terminates a binary constant. The first digit of a hexadecimal constant must be numeric. For example, 255 in hexadecimal is 0FFH, not FFH. FFH is a valid identifier. The following are additional examples of hexadecimal and binary representations.

```
1ab0H

01011B

0FFFFH

10111110B
```

Hexadecimal and binary constants cannot contain a decimal point. The value retained is the sixteen least-significant bits of the number specified.

In this manual, the terms real number and floating-point number are interchangeable. The term numeric applies to either a real or integer quantity.

2.5 Variables and Array Variables

A variable in CBASIC represents an integer, a real number, or a string, depending on the type of the identifier.

Each variable always has a value associated with it. The value can change many times during program execution. A string variable does not have a fixed length associated with it. Rather, as different strings are assigned to the variable, the run-time system allocates storage dynamically. The maximum length allowed in a string variable is 32,767 characters. Numeric variables initialize to 0. String variables initialize to a null string.

A variable takes the general form:

identifier [(*subscript list*)]

The following are examples of variables:

```
X$

PAYMENT

day.of.deposit%
```

Array variables look like regular variables with an added subscript list. CBASIC arrays can hold strings, integers, or reals. As with regular variables, the type of identifier specifies the type of array. A subscript list specifies which element in the array to reference. The number of subscripts allowed in a variable is implementation dependent. See Appendix A of the Programming Guide for current values.

A subscript list takes the general form:

(*subscript* {,*subscript*})

The following examples show array variables:

```
y$(i%,j%,k%,l%)

COST(3,5)

POS%(XAXIS%,YAXIS%)

INCOME(AMT(CLIENT%),CURRENT.MONTH%)
```

The subscripts in a subscript list must be numeric expressions. Access to array elements is more efficient if you use integer expressions. If the expression is real, the value rounds to the nearest integer. The subscript list indicates that the variable is an array variable and indicates which element of the array to reference.

Before you reference an array variable in a program, dimension the array using the DIM statement. The DIM statement specifies the upper-bound of each subscript and allocates storage for the array. Section 3 describes the DIM statement.

You must dimension an array explicitly; no default options are available.

Use the subscript list to specify the number of dimensions and the extent of each dimension for the array that you declare. The subscript list cannot contain a reference to the array. All subscripts have an implied lower-bound of zero.

2.6 Expressions

Expressions consist of algebraic combinations of function references, variables, constants, and operators. Expressions evaluate to an integer, real, or string value. The following are examples of expressions.

```
cost + overhead * percent

a*b/c(1.2+xyz)

last.name$ + ", " + first.name$

index% + 1
```

Table 2-1. Hierarchy of Operators

Hierarchy	*Operator*	*Definition*
1	( )	balanced parentheses
2	^	power operator
	Arithmetic Operators	
3	*, /	multiply, divide
4	+, -	plus, minus
	Relational Operators	
5	<	LT (less than)
	<=	LE (less than/equal to)
	>	GT (greater than)
	>=	GE (greater than/equal to)
	=	EQ (equal to)
	< >	NE (not equal)
	Logical Operators	
6	NOT	
7	AND	
8	OR	
9	XOR	

Arithmetic and relational operations work with integers and real numbers. An integer value converts to a real number if the operation combines a real and integer value. The operation then uses the two real values, resulting in a real value. This is mixed-mode arithmetic.

Mixed-mode operations require more time to execute because the compiler generates more code. A mixed-mode expression always evaluates to a real value.

The power operator calculates the logarithm of the mantissa if the calculation uses real values. A warning results when the number to the left of the operator is negative because the logarithm of a negative number is undefined. The absolute value of the negative number is used to calculate the result. The exponent can be positive or negative.

If both values used with the power operator are either integer constants or integer variables, CBASIC calculates the result by successive multiplications. This allows you to raise a negative integer number to an integer power. With integers, if the exponent is negative, the result is zero. In all cases, 0 ^ 0 is 1, and 0 ^ X, where X is not equal to 0, is 0.

If the exponent is an integer but the base is real, the integer converts to a real value before calculating the result. Likewise, if the exponent is real but the base is an integer quantity, CBASIC calculates the result using real values.

Only the relational operators and +, the concatenation operator, work with string variables. CBASIC does not support mixed string and numeric operations. The mnemonic relational operators (LT, LE, etc.,) are interchangeable with the corresponding algebraic operators (<, < =, etc.). Relational operators result in integer values. A 0 is false and a -1 is true.

Logical operators AND, NOT, OR, and XOR operate on integer values and result in an integer number. The result is bitwise logical. If you use a real value with logical operators, it first converts to an integer.

If a numeric quantity exceeds the range from 32,767 to -32,768, you cannot represent it with a 16-bit two's complement binary number. Logical operations on such a number produce unpredictable results.

These are results of logical operations:

12 AND 3	= 0	1100B AND 0101B	= 4
NOT -1	= 0	NOT 3H	= -4
12 OR 3	= 15	0CH OR 5H	= 13
12.4 XOR 3.2	= 15	12.4 XOR 3.7	= 8

You can increase efficiency by using integer expressions instead of real expressions for relational tests and logical operations.

If a series of digits contains no decimal point or ends in a decimal point, the compiler attempts to store it as an integer. If the resulting number is in the range of CBASIC integers, the compiler treats it as an integer. If the constant is then required in an expression as a real number, the constant converts to a real number at run-time. For example,

```
X = X + 1.
```

causes the integer constant 1 to convert to a real value before adding it to X. To eliminate this extra conversion, embed the decimal in the number as shown:

```
X = X + 1.0
```

Actually, there is very little difference in execution speed. A similar situation exists in the following statement:

```
Y% = X% + 1.0
```

In this case, the X% converts to a real number before adding it to the real constant. The result then converts back to an integer prior to assignment to Y%.

Generally, you should avoid mixed-mode expressions whenever possible, and do not use real constants with integer variables. CBASIC stores most whole numbers used in a program as integers. This provides the most effective execution.

If an overflow occurs during an operation between real values, an execution error occurs.

End of Section 2

Section 3
Statements and Functions

The syntax notation in this section uses the following typographical conventions to highlight the various elements that make up each statement and function.

- CAPS designate CBASIC Compiler keywords.
- Lower-case letters indicate variables.
- Italics identify syntactic items, such as expressions.
- Items enclosed in square brackets [ ] are optional.
- Items enclosed in braces { } are optional and can be repeated.

All other punctuation, such as delimiters and parentheses, must be included. The glossary in Appendix D contains general definitions of syntactic items such as *expression*, *file specification*, and *label.*

ABS Function

The ABS function returns the absolute value of a number.

Syntax:

x = ABS(*numeric expression*)

Explanation:

The ABS function returns a real number. Integer expressions convert to real numbers.

Examples:

```
X = ABS(150)

Y = ABS(-150)

IF ABS(TEMP.A-TEMP.B) < SAFE.LIMIT  THEN CALL WARN.MSG
```

ASC Function

The ASC function returns the ASCII decimal value of the first character in a string.

Syntax:

i% = ASC(*string expression*)

Explanation:

ASC returns an integer between 0 and 255. The string must contain at least one character. An execution error occurs if the string expression evaluates to a null string.

Refer to Appendix B for a listing of ASCII symbols and corresponding numeric values. The inverse function of ASC is CHR$.

Examples:

```
PRINT ASC(A$ + B$)

SEND% = ASC(LAST.NAME$)

IF ASC(DIGIT$) > 47 AND ASC(DIGIT$) < 58 \
   THEN PRINT "VALID DIGITS"
```

ATN Function

The ATN function returns the arctangent of a number.

Syntax:

x = ATN(*numeric expression*)

Explanation:

The ATN function is the inverse of the TAN function. ATN returns the angle, expressed in radians, whose tangent is the expression. ATN returns a real number.

Examples:

```
X = ATN(.6494)

PI = 3.14159
IF ATN(N) < PI/2.0 THEN \
   PRINT "ANGLE LESS THAN 90 DEGREES"

PI = 3.14159
RADIANS = ATN(X)
DEGREES = RADIANS * 180/PI
```

ATTACH Function

The ATTACH function returns a Boolean integer value indicating whether or not a specified printer is available for program use. If the printer is available, the function attaches it to the program.

Syntax:

i% = ATTACH(*printer number*)

Explanation:

Use ATTACH with concurrent or multiuser operating systems. The ATTACH function returns the value that the operating system returns after attempting to attach a specified printer. A logical false, 0, indicates that the printer is attached for program use. ATTACH returns a logical false in systems that do not support multiple printers.

Once ATTACH attaches a printer to a program, no other program can use that printer.

Examples:

```
I% = ATTACH(4)

J% = ATTACH(PRINTER.NO%)

IF ATTACH(PRINTER.NO%) = TRUE%  THEN GOTO MESSAGE
   LPRINTER
   CALL  PRINT.TABLE.OF.VALUES
   CALL  PRINT.BAR.CHART
   DETACH
```

CALL Statement

The CALL statement transfers program control to a multiple-line function.

Syntax:

CALL *function name* {(*parameter list*)}

Explanation:

The CALL statement passes parameters to and starts execution of the specified function. The address of the statement following the CALL statement is placed on a stack. A RETURN or FEND statement in the function sends control back to the statement following the CALL statement.

The parameter list is a list of expressions, variables, or constants. You must separate the expressions with commas. The number of parameters specified in a CALL statement must match the number of formal parameters in the function definition. Parameter data types in the CALL statement and function definition must also match. Numeric parameters convert from integer to real, or real to integer, as required.

The CALL statement cannot reference a single-line function or a program label. Section 4 explains how to define and use functions.

Examples:

```
CALL CLEAR.SCREEN

CALL FN.CALC.TOTAL(SUB%)

CALL GET.REC(FILE.NM$, REC.NO%, AMOUNT)
```

CHAIN Statement

The CHAIN statement loads another program into memory and starts execution.

Syntax:

CHAIN *filespec*

Explanation:

The CHAIN statement can load two types of programs: an overlay program generated by the linker, or a directly executable file. CHAIN can load files generated by languages other than CBASIC. However, before you chain to an overlay file, the linker must create that overlay and the root program at the same time.

The filespec can be a string expression, a variable, or a constant. The compiler assumes a filetype of .OVL if not specified otherwise in the filespec. Refer to the Programming Guide for more information on chaining modules and programs.

Examples:

```
CHAIN "B:AVERAGES"
```

```
CHAIN NEW.PROG$
```

```
TOTALS$ = "ACCOUNTS.OVL"
CHAIN CDRIVE$ + TOTAL$
```

CHR$ Function

The CHR$ function returns a one character string. The string is a single ASCII character that has the specified ASCII decimal value.

Syntax:

a$ = CHR$(*numeric expression*)

Explanation:

The expression contains the ASCII decimal value of the character. If the expression is real, CHR$ converts it to an integer.

Refer to Appendix B for a listing of ASCII symbols and corresponding numeric values. The ASC function is the inverse function of CHR$.

Examples:

```
REM  BEEP THE TERMINAL
PRINT CHR$(7)

LINEFEED% = 10
PRINT CHR$(LINEFEED%)

IF CHR$(INP(IN.PORT%)) = "A" THEN GOSUB 100
```

CLOSE Statement

The CLOSE statement closes disk files.

Syntax:

CLOSE *file number{,file number}*

Explanation:

The CLOSE statement closes the files, releases the file numbers, and frees all buffer space that the files used. A file must first be activated with a CREATE or OPEN statement before using a CLOSE statement. An IF END statement assigned to a closed file has no further effect unless you reassign the file number in a CREATE or OPEN statement.

The file number is a unique identification number you assign to a file with the CREATE or OPEN statement. File numbers can be any numeric expression. If file numbers evaluate to real values, they convert to integers.

STOP statements automatically close all active files. A run-time error does not close files.

Examples:

```
CLOSE 2

CLOSE 5, 12, 20

CLOSE UPDATE.FILE%, OLD.MASTER.FILE%, NEW.MASTER.FILE%
```

COMMAND$ Function

The COMMAND$ function returns a string containing the command tail used to execute the program.

Syntax:

a$ = COMMAND$

Explanation:

A command line is the line that you enter at the keyboard telling the operating system to run a program. A command line consists of a command keyword and an optional command tail. The command keyword identifies the program to execute. The command tail can contain extra information for the program such as a filename, option, or parameter.

The COMMAND$ function does not return the command keyword. COMMAND$ eliminates all blanks preceding the first character in the command tail and converts all characters to upper-case.

You can use the COMMAND$ function anywhere, any number of times in a program. You can use COMMAND$ in any CBASIC program loaded with a CHAIN statement.

Examples:

```
IF COMMAND$ = "" THEN STOP
```

For the following command lines,

```
PAYROLL nochecks totals

PAYROLL NOCHECKS TOTALS

ACCOUNTS nochecks TOTALS
```

COMMAND$ returns the string:

```
NOCHECKS TOTALS
```

COMMON Statement

The COMMON statement specifies variables to retain in memory for use by chained programs.

Syntax:

COMMON *variable*{,*variable*}

Explanation:

Only blank lines, REM statements, and data type declaration statements can precede COMMON statements.

The compiler treats all COMMON statements in a program as one consecutive list of variables. Therefore, a program can contain any number of COMMON statements. All COMMON statements taken as a group must have the same number of variables in each chained program. Each COMMON statement in a chained program can contain a different number of variables if the total number of variables matches for all chained programs. The position of each variable and data type must match in each chained program. Dimensioned variables must have the same number of subscripts.

For array variables, place the number of subscripts in parentheses after the array name. The COMMON statement does not indicate the size of the subscript. Be sure to allocate array space with a DIM statement before referencing an array variable in COMMON. The first program requiring access to the array must contain the DIM statement. Subsequent programs can access the array without affecting the data.

If a DIM statement executes a second time for the same array, the original data is lost. However, elements in a string array are not released from memory. Set string array elements to null strings before reexecuting a DIM statement for the same string array. Refer to the DIM statement for information on setting array elements to null.

Examples:

```
COMMON X

COMMON X, I%, A$

COMMON A$(2), B$(3), Y, Z
```

CONCHAR% Function

The CONCHAR% function reads one character from the console keyboard and returns the decimal ASCII representation of that character.

Syntax:

i% = CONCHAR%

Explanation:

CONCHAR% waits for a character to be entered at the console keyboard, then displays the character on the console screen before returning the ASCII decimal value. However, if the ASCII decimal value is less than 32, CONCHAR% does not display the character.

The low-order eight bits of the returned value comprise the binary ASCII representation. The high-order eight bits are always zeros. The value returned is a decimal integer. Refer to Appendix B for a listing of ASCII symbols and corresponding numeric values. The INKEY function performs the same task that CONCHAR% performs except INKEY does not display the character on the console screen.

Examples:

```
ALPHA% = CONCHAR%

IF CONCHAR% = 48  THEN GOSUB 1000

PRINT CHR$(CONCHAR%)
```

CONSOLE Statement

The CONSOLE statement directs program output to the console screen.

Syntax:

CONSOLE

Explanation:

CBASIC maintains a special print control flag to determine whether output from a PRINT statement is displayed on a console screen or printer. The CONSOLE and LPRINTER statements set and reset the flag. You cannot access the print control flag directly.

Initially, the flag is set to logical false and output from PRINT statements displays on the console screen. LPRINTER sets the flag to logical true so information can be printed on a list device. The CONSOLE statement resets the flag to false and redirects output to the console screen.

The print control flag does not affect INPUT statement prompt strings. Prompt strings always print on the console screen.

If the current output column is not 1, both CONSOLE and LPRINTER send a carriage return line-feed prior to changing the print control flag. Refer to the LPRINTER statement for more information.

Examples:

```
CONSOLE

500 CONSOLE

IF LST.REQUEST  THEN LPRINTER \
   ELSE CONSOLE
```

CONSTAT% Function

The CONSTAT% function returns a logical value signifying console status.

Syntax:

```
i% = CONSTAT%
```

Explanation:

Use CONSTAT% to determine if the console has a ready status. Ready status means a character has been entered at the console keyboard but has not been read by the program. CONSTAT% returns a -1 or logical true if the console is ready. Otherwise, CONSTAT% returns a zero or logical false.

Examples:

```
IF CONSTAT%  THEN \
   GOSUB 95  REM PROCESS OPERATOR INTERRUPT

PRINT "PRESS ANY KEY TO CONTINUE"
      WHILE NOT CONSTAT%
      WEND
```

COS Function

The COS function returns the cosine of a number.

Syntax:

x = COS(*numeric expression*)

Explanation:

All CBASIC trigonometric functions require that you specify the numeric expression in radians. Integers convert to real numbers. The cosine value returned is a real number.

Examples:

```
I% = COS(3.14159)

IF COS(ANGLE) = 0.0  THEN VERTICAL% = TRUE%

PI = 3.14159
INPUT "ENTER DEGREE VALUE..."; DEGREES
RADIANS = DEGREES * PI/180.0
X = COS(RADIANS)
```

CREATE Statement

The CREATE statement creates a new disk file on disk with no information in it.

Syntax:

CREATE *filespec* [RECL *rec length*]
AS *file number* [BUFF *number of buffers*] [*mode*]

Explanation:

CREATE erases any preexisting file of the same name before creating the new file. Use CREATE statements to create either stream or fixed disk files.

To make a stream file, CREATE requires a filespec and a file number. The filespec can be a string expression, variable, or constant. The file number is a unique integer identification number ranging from 1 to the current implementation limit for the number of files accessible at one time. Refer to Appendix A of the Programming Guide for the current limit. Place the file number in a CREATE statement after the keyword AS.

To create a fixed file, specify the record length with the RECL parameter in addition to the filespec and file number.

The BUFF option assigns additional internal buffers. CREATE assumes a default value of 1 buffer if not specified otherwise. The BUFF parameter must specify 1 if you access the file randomly.

CBASIC supports three different modes for accessing files: LOCKED, UNLOCKED, and READONLY. Use the mode parameter under multiuser or concurrent operating systems. If you CREATE a file in LOCKED mode, no other program or user can access that file. UNLOCKED mode allows more than one program or user to access the file. READONLY files allow more than one program or user to read the file. Another program or user cannot modify the data inside a READONLY file. You cannot CREATE a file in READONLY mode. Use READONLY with the OPEN statement.

Examples:

```
CREATE "SALES.FEB"  AS 1

CREATE "B:TEST.DAT"  RECL 250  AS 20

CREATE ACCOUNT.MASTER$  RECL 500  AS 12  BUFF 4

CREATE "B:" + NAME$ + "." + LEFT$(STR$(CURR.WORK%), 3)\
       AS CURR.WORK%

CREATE "FILE.DAT"  AS NUM%  BUFF (MFRE/128)
```

DATA Statement

The DATA statement defines a list of constants that a READ statement can assign to variables.

Syntax:

DATA *constant*{,*constant*}

Explanation:

The constant list in a DATA statement can be any combination of integer, real, and string constants. However, data types for the constants in the DATA statements and the corresponding variables in the READ statements must match. Real constants assigned to integer variables by a READ statement are truncated to the integer portion of the real number.

DATA statements can span more than one physical line using the backslash continuation character, but cannot appear on the same line with other statements. The continuation character can appear in string constants enclosed in quotation marks. However, string constants do not require quotation marks. Delimit each constant with a comma or a carriage return line-feed.

DATA statements are nonexecutable statements that can appear anywhere in a program. CBASIC treats all DATA statements in a program as one consecutive list of constants.

See the READ and RESTORE statements for additional information.

Examples:

```
DATA 3, 25, 14, 8, 66, 181, 4

DATA one, two, three, 4, 5, 6.0
DATA 7.0, eight, 9, 10

DATA 331.5, "VIOLET", 456.2, "BLUE", \
     583.7, "YELLOW", 614.9, "RED"

DATA "ABC\DEF"
```

DEF Statement

The DEF statement defines both single-line and multiple-line functions.

Syntax:

Single-line:

DEF *function name*[(formal parameters)] = *expression*

Multiple-line:

```
DEF function name[(formal parameters)]\
      [EXTERNAL or PUBLIC]
      [declaration statements]
      .
      .
      .
      CBASIC statements
      .
      .
      .
      RETURN
FEND
```

Explanation:

A function definition must occur in a program before using the function. To define a function, the word DEF must precede the function name.

Single-line function definitions use an equal sign followed by an expression. The expression contains the actual process that the single-line function performs. The data types used in the expression must correspond to the data type used in the function name.

Multiple-line function definitions include optional data declarations and any number of statements. A DEF statement precedes the declaration group, and a FEND statement terminates the function. You can place any number of RETURN statements in the body of the function. Refer to Section 4.4 for information on PUBLIC and EXTERNAL functions.

In both cases, formal parameters hold a place for actual parameters specified in the function reference. A formal parameter is either a string variable or numeric variable; it is never a constant. If a formal parameter is a string variable, the actual parameter must be a string expression. If the formal parameter is numeric, the actual parameter must be numeric. However, real numbers convert to integers and integers convert to real as required.

All formal parameters and any variables declared in the declaration group are local to the function. Labels defined in a multiple-line function are local to that function. Refer to Section 4 for complete information on defining and using functions.

Examples:

Single-line:

```
DEF FN25 = RND * 25.0

DEF HYPOT(SIDE1,SIDE2)= \
    SQR((SIDE1 * SIDE1) + (SIDE2 * SIDE2))
```

Multiple-line:

```
DEF READ.INPUT(INPUT.NO%)
    READ # INPUT.NO%; CUSTNO%, AMOUNT
    RETURN
FEND

DEF TEST(A, B)
    INTEGER TEST, C
    C = A + B
    D = A / B
FEND

DEF COUNT%(INDEX1%)
    COUNT% = 0
    FOR I% =1 TO INDEX1%
        COUNT% = COUNT% + ARRAY(I%)
    NEXT I%
    COUNT% = COUNT%
    RETURN
FEND
```

DELETE Statement

The DELETE statement deactivates files from processing and erases them from the disk directory.

Syntax:

DELETE *file number*{,*file number*}

Explanation:

The DELETE statement erases the file, releases the file number, and reallocates all buffer space that the file used. An IF END statement assigned to the file number has no further effect unless you reassign the file number with a CREATE or OPEN statement.

The file number is the unique identification number you assign to a file with a CREATE or OPEN statement.

Examples:

```
DELETE 3

DELETE 6, 13, 18

DELETE UPDATE.FILE%, OLD.MASTER.FILE%
```

DETACH Statement

The DETACH statement deactivates a printer from program access.

Syntax:

DETACH

Explanation:

Use the DETACH statement with the ATTACH function under concurrent or multiuser operating systems. The DETACH statement has no effect in systems that do not support multiple printers.

Example:

```
IF ATTACH(PRINTER.NO%) = FALSE%  THEN GO TO MESSAGE
   LPRINTER
   CALL TABLE.OF.VALUES
   CALL BAR.CHART
DETACH
```

DIM Statement

The DIM statement dynamically allocates space for an array.

Syntax:

DIM *identifier*(*subscript list*)

Explanation:

The DIM statement reserves storage space for both numeric and string arrays and specifies the upper-bound of each subscript. Initially, the individual elements are set to zero in numeric arrays, and are set to null in string arrays.

The number of subscripts in the DIM statement determines the number of dimensions in the array. The number of subscripts is limited by current implementation values. Refer to Appendix A of the Programming Guide for the current limit. The value of each subscript plus 1 equals the number of elements in each dimension. All subscripts have an implied lower-bound of zero.

Each execution of a DIM statement allocates a new array. If a DIM statement for a numeric array executes a second time, data in the first allocation is lost. You should set each element in a string array to null before executing the DIM statement a second time. Set array elements to null by setting the elements equal to a string variable that is not assigned a value. Refer to the Programming Guide for information on the internal representation of arrays.

String array elements are limited to 32,760 bytes each.

Examples:

```
DIM A(10)

DIM B%(50, 50, 50)

DIM NAME$(300), ADDRESS$(300), PHONE(300)
```

END Statement

The END statement terminates a CBASIC program.

Syntax:

END

Explanation:

The END statement is a directive to the compiler indicating an end to the source program. The compiler reports an error if any statements follow the END statement.

An END statement cannot appear on the same line with other statements.

The compiler adds an END statement to a program automatically if you omit it in the source code file.

Examples:

```
500 END
```

```
END
```

ERR Function

The ERR function returns a two-character string signifying the last execution error to occur in a program.

Syntax:

a$ = ERR

Explanation:

Use the ERR function with the ON ERROR statement and ERRL function. The two-character string contains an execution error message as listed in Appendix D of the Programming Guide. The ERR function returns a null string if no error has occurred in the program at the time the ERR function executes.

You can use the ERR function any number of times in a program.

Examples:

```
IF ERR = "OM"  THEN \
   PRINT "OUT OF MEMORY"
IF ERR = "EX"  THEN \
   CALL EXTENDED.ERROR(ERR)

REM IF DATA.STRING$ IS NULL, ERROR AC OCCURS
    ON ERROR  GOTO 100
    ALPHA% = ASC(DATA.STRING$)
    PRINT ALPHA%
GOTO 200
    100 A$ = ERR
    PRINT A$
200 END
```

ERRL Function

The ERRL function returns the line number in which the last execution error occurred.

Syntax:

i% = ERRL

Explanation:

You can use the ERRL function with or without the ON ERROR statement. ERRL determines the line number of the last execution error.

ERRL returns an integer. You must compile the source program using the N toggle, or ERRL returns a zero.

Example:

```
REM IF DATA.STRING$ IS NULL, ERROR AC OCCURS
    ON ERROR  GOTO 100
    ALPHA% = ASC(DATA.STRING$)
    PRINT ALPHA%
STOP
    100 PRINT ERRL
END
```

ERRX Function

The ERRX function returns the sixteen-bit MP/M II™ extended error code.

Syntax:

```
i% = ERRX
```

Explanation:

Use ERRX with the ON ERROR statement and ERR function. Execution error EX indicates the occurrence of an MP/M II extended error. If the ERR function detects the error EX, you can use ERRX to determine which extended error occurred. ERRX returns an integer corresponding to an MP/M II extended error code. If an extended error has not occurred, ERRX returns a 0. Refer to the *MP/M II Operating System Programmer's Guide* for descriptions of extended error codes.

Example:

```
ON ERROR  GOTO CHECK.ERROR
OPEN "FILE.DAT"  AS 5  READONLY
 .
 .
 .
CHECK.ERROR:
   IF ERR ="EX"  THEN \
   PRINT "MP/M II EXTENDED ERROR..."; ERRX
CLOSE 5
END
```

EXP Function

The EXP function returns the constant e raised to an exponent.

Syntax:

x = EXP(*numeric expression*)

Explanation:

The constant e is the base of natural logarithms equal to 2.7182. Integers convert to real numbers. EXP returns a real number.

Examples:

```
X = DEVIANCE / EXP(2)

Z = EXP(SIN(X) * COS(Y))
```

FEND Statement

The FEND statement terminates multiple-line, user-defined functions.

Syntax:

FEND

Explanation:

Use one FEND statement to terminate each multiple-line function definition. FEND returns program control to the statement following the last function call or reference.

Examples:

```
DEF CALC.TOTAL(A%, B%)
    TOT% = A% + B%
FEND

DEF AREA.LAND(LENGTH, WIDTH)
    AREA = LENGTH * WIDTH
    PRINT "THE AREA IS..."; AREA
    RETURN
FEND
```

FLOAT Function

The FLOAT function converts a number to a floating-point real number.

Syntax:

x = FLOAT(*numeric expression*)

Explanation:

A real expression first converts to an integer, then back to floating-point form.

Examples:

```
X = FLOAT(360)

DOLLARS = FLOAT(DOLLARS%)

POSITION = COS(FLOAT(ANG%)) * OFFSET
```

FOR Statement

The FOR statement controls the execution of a FOR/NEXT loop.

Syntax:

> FOR *index variable = numeric expression*
> TO *numeric expression* [STEP *numeric expression*]

Explanation:

All statements between a FOR statement and a corresponding NEXT statement execute repeatedly, depending on the numeric expressions. The expressions before and after the keyword TO determine the number of loop executions. The first expression is the initial value and the second expression is the terminating value.

Each execution of the statements in the FOR/NEXT loop adds the value in the STEP expression to the index variable. If not specified, the STEP value defaults to 1. If the STEP expression is positive, the value of the index variable must exceed the expression following the keyword TO for the loop to terminate. If the STEP expression is negative, the value of the index variable must become less than the expression following the keyword TO for the loop to terminate.

The index variable must be a nonsubscripted numeric variable, either real or integer. The FOR statement converts all numeric expressions to real numbers if the index variable is real, and to integers if the index variable is an integer.

The sign of the STEP expression determines how the loop ends. If the STEP expression is positive, the loop executes as long as the index variable is less than or equal to the terminating value. If the STEP expression is negative, the loop executes as long as the index is greater than or equal to the terminating expression.

FOR/NEXT loops can contain any executable statement. You can nest FOR/NEXT loops. Refer to Appendix A of the Programming Guide for implementation limits on FOR/NEXT loop nesting. Refer to the NEXT statement for additional information.

Examples:

```
FOR I% = 1 TO 10
    PRINT I% ; "TESTING CBASIC!"
NEXT I%

FOR J = -1.0 TO -10.0  STEP -2.0
    PRINT J ; "TESTING CBASIC!"
NEXT J

FOR POSITION=MARGIN+TABS TO PAPER.WIDTH STEP TABS
    PRINT TAB(POSITION); SET.TAB$
NEXT POSITION
```

FRE Function

The FRE function returns the amount of space available in the Free Storage Area (FSA).

Syntax:

x = FRE

Explanation:

FRE returns an integer equal to the number of bytes available in the FSA. FRE actually returns an unsigned 16-bit binary number. Be sure to interpret the function correctly when free space is greater than 32,767 bytes. CBASIC treats a number greater than 32,767 as a negative number. Consequently, a negative number indicates a large positive value. A large amount of space remains when FRE returns a negative value.

See the MFRE function for more information.

Examples:

```
X = FRE
  PRINT X

IF FRE < 500.0  THEN PRINT "LOW MEMORY SPACE!"
```

GET Function

The GET function reads one byte of data from a specified disk file.

Syntax:

i% = GET(*file number*)

Explanation:

Each execution of the GET function reads the binary data from one byte in the file and returns an integer value between 0 and 255.

The file number is a unique identification number you assign to a file in a CREATE or OPEN statement.

Examples:

```
I% = GET(3)

IF END # FILE.NO%  THEN SET.EOF
WHILE NOT EOF%
      CALL PROCESS(GET(FILE.NO%))
WEND
STOP
SET.EOF: EOF% = TRUE%
RETURN
```

GOSUB Statement

The GOSUB statement execution to the subroutine identified with a statement label.

Syntax:

GOSUB *label*
GO SUB *label*

Explanation:

CBASIC saves the address of the statement following a GOSUB statement on a stack. This allows a RETURN statement to send control back to the statement following the GOSUB.

The label must be defined somewhere within the program. GOSUB statements inside multiple-line functions cannot reference a label outside the body of the function. Likewise, GOSUB statements outside of a given function cannot reference a label inside the function.

Do not place a colon after an alphabetic label reference in a GOSUB statement.

Examples:

```
GOSUB 10
```

```
GOSUB GET.NEXT.ONE
```

```
LPRINTER
PRINT "SPACE BEFORE TABLE OF VALUES"
GO SUB 200
PRINT "SPACE AFTER TABLE OF VALUES"
STOP
200 REM  PRINT THE TABLE
    FOR INDEX% = 1 TO TABLE.SIZE%
        PRINT TABLE(INDEX%)
    NEXT INDEX%
RETURN
```

GOTO Statement

The GOTO statement transfers execution to a statement identified with a label.

Syntax:

GOTO *label*

GO TO *label*

Explanation:

The GOTO statement continues execution at the statement label you specify. If the specified statement is not executable, execution continues with the next executable statement encountered.

The label must be defined somewhere within the program. Labels within multiple-line functions are local to the function. GOTO statements inside multiple-line functions cannot reference a label outside the body of the function. Likewise, GOTO statements outside of a given function cannot reference a label inside the function.

Do not place a colon after an alphabetic label reference in a GOTO statement.

Examples:

```
112 GOTO 1000
```

```
GO TO 2001.5
```

```
GOTO CALCULATIONS
```

```
X: GOTO X
```

IF Statement

The IF statement transfers execution to one of two statements or statement groups, depending on the value of a logical expression.

Syntax:

IF *logical expression* THEN *statement group* [ELSE *statement group*]

Explanation:

The IF statement determines whether the expression is true (-1) or false (0). Real numbers convert to integers. If the expression is true, execution passes to the statement group following the keyword THEN. If the expression is false, execution passes to the statement group following the keyword ELSE. If you omit the ELSE portion of the IF statement, execution falls through to the next executable statement when the logical expression evaluates to false.

A statement group can contain one or more executable CBASIC statements. Use the colon to group statements together and the backslash continuation character to continue a statement group over several lines.

You can nest IF statements. If required, you can use empty or null statements to force the proper pairing of an IF/THEN portion with the corresponding ELSE portion. An ELSE corresponds to the nearest unpaired IF.

Examples:

```
X% = 100
IF X% < 150  THEN GOSUB REPEAT

IF DIMENSIONS.WANTED%  THEN PRINT LENGTH, HEIGHT \
   ELSE GOTO 425

IF TIME > LIMIT  THEN PRINT TIME.OUT.MSG$ : \
   BAD. RESPONSE% = BAD.RESPONSE% + 1 : \
   QUESTION% = QUESTION% + 1 \
ELSE \
   PRINT THANK.MSG$ : \
   GOSUB 2000 : \  ANALYSE RESPONSE
   ON RESPONSE% GOSUB 2010, 2020 \
      2030, 2040, 2050
```

IF END Statement

The IF END statement transfers program execution to a specified label when a file access exception occurs.

Syntax:

IF END # *file number* THEN *label*

Explanation:

The IF END statement detects the following three file access exceptions:

- attempting to READ past an end-of-file
- disk or directory full when creating or writing to a file
- attempting to OPEN a file that does not exist

Control reverts to an IF END statement when one of three preceding exceptions occurs. Program execution transfers to the statement specified by the label that follows the keyword THEN. An IF END statement applies to the one file specified by the file number. The file number is a unique identification number assigned to a file in a CREATE or OPEN statement.

A program can have any number of IF END statements for the same file to transfer execution to different labels. The most recently executed IF END statement for a given file number is the one in effect when a file access exception occurs.

To detect access errors for a given series of statements, the IF END statement must execute before the statements.

You can execute an IF END statement for a file number before that file number is active. This procedure traps errors caused by opening a file that does not exist or creating a file when there is no directory space.

Examples:

```
IF END #7  THEN 500
OPEN "FILE.DAT"  AS 7

IF END #19  THEN 230.5
READ #19; FIRST$, SECOND$, THIRD$

IF END FILE.NO%  THEN MESSAGE
PRINT # FILE.NO%; FIRST%, SECOND%, THIRD%
```

INITIALIZE Statement

The INITIALIZE statement allows you to change diskettes and other removable storage media during program execution without restarting the operating system.

Syntax:

INITIALIZE [*numeric expression*]

Explanation:

Storage media must be changed before the INITIALIZE statement executes. Never change media while files are open on that media.

When using INITIALIZE under multiuser systems, use the numeric expression to specify which drives to reset. INITIALIZE treats the expression as a series of binary digits. You specify which drives to reset with a binary 1. Drives A through P correspond to digits from right to left.

Examples:

`INITIALIZE` resets all drives

`INITIALIZE 11B` resets drives A and B

`INITIALIZE 110B` resets drives B and C

`INITIALIZE 1000B` resets drive D

INKEY Function

The INKEY function returns the ASCII decimal value equal to a character entered at the console keyboard.

Syntax:

i% = INKEY

Explanation:

INKEY waits for a character to be entered at the console keyboard. Unlike the CONCHAR% function, INKEY does not display the character on the console screen.

The low-order eight bits of the returned value comprise the binary ASCII representation. The high-order eight bits are always zeros.

INKEY is useful to prevent passwords and other special characters from printing. INKEY accepts control characters.

Examples:

```
I% = INKEY

WHILE INKEY <> ESC%
WEND

REM  GET PASSWORD
  PRINT "ENTER PASSWORD..."
    PW$ = ""
FOR I% = 1 TO PW.LEN%
    PW$ = PW$ + CHR$(INKEY)
NEXT I%
```

INP Function

The INP function returns a value from a CPU input/output port.

Syntax:

i% = INP (*numeric expression*)

Explanation:

The INP function is hardware dependent and might not apply to certain microprocessors. The expression must specify a valid I/O port number. CBASIC does not check the validity of the port number.

Real numbers convert to integers. The function returns an eight-bit integer value.

Examples:

```
PRINT INP(ADDR%)

IF INP(255) > 0  THEN PRINT CHR$(7)

ON INP(INPUT.DEVICE.PORT%)  GOSUB \
   100, 200, 300, 400, 400, 400, 500
```

INPUT Statement

The INPUT statement accepts data from the console during program execution and assigns the data to program variables.

Syntax:

INPUT [*prompt string*;] *variable* {,*variable*}

Explanation:

The INPUT statement prompts you for response with a question mark during program execution. If you specify a literal prompt string, the INPUT statement prints the string on the console screen and waits for input from the keyboard. If you specify a null prompt string, the INPUT statement simply waits for input from the keyboard. One blank space prints after either prompt. A prompt string must be a string constant.

Each variable initiates a request from the console screen. Each response at the console corresponds to a variable in the INPUT statement. A warning message appears on screen if the number of response items you enter does not match the number of variables. You must separate individual response items with commas. However, you can enclose string responses in quotation marks, allowing commas to serve as literal characters. Press the carriage return key to complete a response.

All characters entered in response display on the console screen. The maximum number of characters you can enter in response is implementation dependent. CBASIC supports at least 255 characters in any implementation.

For numeric data entered in response to an INPUT statement, the data type converts to the assigned variable data type. Conversion terminates if INPUT encounters an unexpected character. INPUT does not print an error message to indicate integer overflow.

All CP/M line-editing functions remain in effect.

Examples:

```
INPUT PRICES

INPUT "Please enter your last name..."; LNAME$

INPUT "Enter three integer values."; INT1%, INT2%, INT3%
```

INPUT LINE Statement

The INPUT LINE statement accepts one line of data from the console and assigns it to a string variable.

Syntax:

INPUT [*prompt string*;] LINE *string variable*

Explanation:

The INPUT LINE statement is a special form of the INPUT statement. Only one variable can appear following the keyword LINE. INPUT LINE prompts you for response with a question mark during program execution unless you specify a literal prompt string.

INPUT LINE accepts all characters in response, including commas and spaces, until you press the carriage return key. If you enter only a carriage return in response, INPUT LINE assigns a null string to the variable.

The maximum length of a line is 255 characters. All CP/M line-editing functions remain in effect.

Examples:

```
INPUT LINE CHARACTERS$

INPUT "Please enter your address."; LINE ADDR$

INPUT "Type RETURN to continue..."; LINE DUMMY$
```

INT Function

The INT function returns the integer portion of a number as a floating-point number.

Syntax:

x = INT (*numeric expression*)

Explanation:

INT truncates the fractional portion of the expression. Integer numbers convert to real numbers. INT returns a real number.

Examples:

```
X = INT(322.50)

REFUND = INT(TAXES - CONSTANT)

IF (NUM/2) - INT(NUM/2) = 0  THEN \
   PRINT "EVEN"  ELSE PRINT "ODD"
```

INT% Function

The INT% function returns the integer portion of a number as an integer.

Syntax:

i% = INT% (*numeric expression*)

Explanation:

INT% truncates the fractional portion of the expression. Integers first convert to real numbers then back to integer form. INT% returns an integer.

Examples:

```
I% = INT%(452.25)

LENGTH% = 12 * INT%(FEET) + INCHES%

REFUND = INT%(TAXES - CONSTANT)
```

INTEGER Statement

The INTEGER statement is a declaration statement that specifies the integer data type for variables and function names.

Syntax:

INTEGER *identifier*[,*identifier*]

Explanation:

Use INTEGER statements in the declaration group of a program or multiple-line function. Declaration statements override the default data type specified with the last character in an identifier.

To use an array identifier in an INTEGER statement, place the number of subscripts in parentheses after the array name.

Refer to Section 2 for more information on declarations and identifiers.

Examples:

```
INTEGER I%
```

```
INTEGER I, J, K
```

```
INTEGER COORD(2), I(1)
STRING NAMES$(1)
```

LEFT$ Function

The LEFT$ function returns a string consisting of the leftmost characters in a string.

Syntax:

a$ = LEFT$ (*string expression, numeric expression*)

Explanation:

The numeric expression is a positive value specifying the number of characters to return. If the numeric expression is negative, an error occurs. Real expressions convert to integers. LEFT$ returns a null string if the numeric expression equals zero. LEFT$ returns the entire string if the numeric expression specifies more characters than the string contains.

Examples:

```
A$ = LEFT$("GOODXXXX", 4)

PRINT LEFT$(INPUT.DATA$, GOOD%)

IF LEFT$(ANSWER$, 1) = "Y"  THEN GOTO CONTINUE
```

LEN Function

The LEN function returns the length of a string.

Syntax:

i% = LEN (*string expression*)

Explanation:

The LEN function returns an integer. LEN returns zero if the expression is a null string.

Examples:

```
I% = LEN("645 BAYVIEW AVENUE")

IF LEN(TEMPORARY$) > 25  THEN \
   PRINT "LIMIT ENTRY TO 25 CHARACTERS"

FOR INDEX% = 1 TO LEN(OBJECT$)
    NUM%(INDEX%) = ASC(MID$(OBJECT$,INDEX%,1))
NEXT INDEX%
```

LET Statement

The LET statement assigns a value to a variable.

Syntax:

[LET] *variable* = *expression*

Explanation:

The keyword LET is optional.

Variables and expressions can be strings, real numbers, or integers. For numeric expressions and variables, the LET statement converts the data type of the expression to match the data type of the variable.

Examples:

```
100 LET A = B + C

SALARY = (HOURS.WORKED * RATE) - DEDUCTIONS

DATE$ = MONTH$ +  " "  + DAY$ + " " +YEAR$

S(I%) = T(I%) + U(I%) - W
```

LOCK Function

The LOCK function prevents any program from modifying the data in a record.

Syntax:

i% = LOCK(*file number, record number*)

Explanation:

The LOCK function returns the value that the operating system returns after attempting to lock a record. Normally, a zero indicates that the record is locked. LOCK returns a value of zero in systems that do not support record locking.

To LOCK a record, the file must be a fixed file accessed in the UNLOCKED mode. Refer to the UNLOCK function, CREATE statement, and OPEN statement for more information.

Examples:

```
I% = LOCK(20,3)

IF LOCK(6,30) > 0 THEN GOTO LOCK.ERROR.MSG

FOR J% = 1 TO 10
    K% = LOCK(11,J%)
    PRINT K%
NEXT J%
```

LOG Function

The LOG function returns the natural logarithm of a number.

Syntax:

x = LOG (*numeric expression*)

Explanation:

LOG returns the natural logarithm as a real number. Integer expressions convert to real numbers. The expression must be a positive value greater than zero.

Examples:

```
X = LOG(266.72)

PRINT "The logarithm ="; LOG(VALUE%)

IF LOG(VALUE) > TOLERANCE%  THEN GOSUB NEWDATA
```

LPRINTER Statement

The LPRINTER statement directs program output to a printer.

Syntax:

LPRINTER

Explanation:

CBASIC maintains a special print control flag to determine whether output from a PRINT statement displays on a console screen or printer. The LPRINTER and CONSOLE statements set and reset the flag. You cannot access the print control flag directly.

Initially, the flag is set to logical false and output from PRINT statements display on the console screen. LPRINTER sets the flag to logical true so information is sent to the printer. The CONSOLE statement resets the flag to false and redirects output to the console screen.

The print control flag does not affect INPUT statement prompt strings. Prompt strings always print on the console screen.

If the current output column is not 1, both LPRINTER and CONSOLE send a carriage return line-feed prior to changing the print control flag. See the CONSOLE statement.

Examples:

```
500 LPRINTER

LPRINTER
PRINT "A table of relative values follows."
PRINT
PRINT TABLE.VALUES

IF DOCUMENT.FILE%  THEN LPRINTER
```

MATCH Function

The MATCH function returns the position of the first occurrence of a specified character pattern in a string.

Syntax:

i% = MATCH (*pattern string, string expression, numeric expression*)

Explanation:

MATCH searches the string expression for a series of characters that matches the pattern defined in the pattern string. The numeric expression specifies a position in the string expression to begin searching.

The pattern string contains a series of letters and digits, plus the following wildcard matching characters, which represent different classes of characters.

# represents any digit
! represents any lower-case or upper-case letter
? represents any character

MATCH returns a zero if either the pattern string or string expression is a null string.

The backslash is an escape character in the pattern string. Any character after the backslash is literal, and does not serve as a wildcard character.

Examples:

`MATCH("is", "Now is the time!", 1)` returns the position 5

`MATCH(" ##", "October 9, 1982", 1)` returns 12

`MATCH("a?", "character", 4)` returns 5

`MATCH("\#", "1#2345#6789", 3)` returns 7

`MATCH("ABCD", "ABC", 1)` returns 0

MFRE Function

The MFRE function returns the largest contiguous area of available memory space in the Free Storage Area (FSA).

Syntax:

```
i% = MFRE
```

Explanation:

MFRE returns an integer equal to the largest number of contiguous bytes available in the FSA. MFRE returns an unsigned 16-bit binary number. Be sure you interpret the function correctly when the amount of contiguous free space is greater than 32,767 bytes. CBASIC treats a number greater than 32,767 as a negative number. Therefore, a negative number actually indicates a large positive value. When MFRE returns a negative value, a large contiguous segment of space remains in memory.

MFRE returns an integer that is less than or equal to the value returned by the FRE function. The FRE function returns the total amount of unallocated space in the FSA whether or not it is contiguous. Refer to the Programming Guide for a description of the Free Storage Area.

Examples:

```
PRINT "CHECK POINT #1"; MFRE

IF MFRE < A.SIZE%  THEN \
   PRINT "CANNOT DIMENSION ARRAY!"

WHILE MFRE > MIN%
      CALL ALLOCATE.MORE
WEND
```

MID$ Function

The MID$ function returns a segment of a string.

Syntax:

A$ = MID$ (*string expression, numeric expression,*
numeric expression)

Explanation:

The first numeric expression specifies a position that determines the first character to return from the original string. MID$ returns a null string if the first numeric expression is greater than the length of the string. The second numeric expression specifies the length of the string segment to return. MID$ returns all characters to the right of the first character specified in the first numeric expression, if the second numeric expression is greater than the number of characters to the right of the first character. The function converts real numbers to integers.

Examples:

```
DIGITS$ = MID$("TOMAHAWK2551K", 9, 4)

VALID$ = MID$(LISTING$, POS%, 1)

DAY$ = MID$("MONTUEWEDTHUFRISATSUN", DAY%*3-2, 3)
```

MOD Function

The MOD function returns the remainder from an integer division.

Syntax:

i% = MOD(*numeric expression, numeric expression*)

Explanation:

The MOD function divides the first expression by the second and returns the remainder. Real numbers convert to integers. MOD returns an integer value.

Examples:

```
I% = MOD(J%, K%)

IF MOD(L%, MAX%) <> 0  THEN \
   PRINT "NOT DIVISOR"
```

NEXT Statement

The NEXT statement denotes the end of a FOR/NEXT loop.

Syntax:

NEXT [*index variable*]

Explanation:

If specified, the index variables after the keyword NEXT must match the index variables in the corresponding FOR statement. The NEXT statement sends control to the beginning of the FOR/NEXT loop until the termination criteria for the loop is met. Refer to the FOR statement for additional information.

Examples:

```
FOR I% = 1 TO 10
    PRINT X(I%)
NEXT I%

FOR LOOP% = 1 TO ARRAY.SIZE%
    GO SUB 210
    GO SUB 410
NEXT
```

ON Statement

The ON statement transfers program execution to one of a number of labels. The ON statement has two forms.

Syntax:

ON *numeric expression* GOTO *label*{,*label*}
ON *numeric expression* GOSUB *label*{,*label*}

Explanation:

The numeric expression determines where to transfer program execution. If the expression evaluates to 1, ON branches to the first label. If the expression evaluates to 2, ON branches to the second label and so forth. However, if the numeric expression evaluates to a number less than one or greater than the number of labels, the results are unpredictable. Always test the value of the numeric expression before executing an ON statement. Real number expressions convert to integers.

When using the ON statement with a GOSUB, the RETURN statement in the subroutine returns execution to the first executable statement following the ON statement.

There is no limit to the number of labels allowed in an ON statement. A label can appear anywhere in a program in relation to the ON statement except in a multiple-line function.

Examples:

```
ON I%  GOTO 10, 20, 30

ON RESULT% - 1  GOSUB 290, 620, 1000, 110

WHILE TRUE%
    GOSUB 100    REM ENTER PROCESS DESIRED
    GOSUB 110    REM TRANSLATE PROCESS NUMBER
    IF PROCESS.DESIRED% = 0 THEN REPEAT
    IF PROCESS.DESIRED% < 6 THEN \
       ON PROCESS.DESIRED% GOSUB \
       1000, \    ADD A RECORD
       1010, \    ALTER NAME
       1020, \    UPDATE QUANTITY
       1030, \    DELETE A RECORD
       1040  \    CHANGE COMPANY CODE
       1050  \    GET PRINTOUT
    ELSE  GOSUB 400  REM ERROR - REPEAT
WEND
```

ON ERROR Statement

The ON ERROR statement branches execution to a label upon detection of an execution error.

Syntax:

ON ERROR GOTO *label*

Explanation:

Program control reverts to an ON ERROR statement when an execution error occurs in a program following the ON ERROR statement. If you use more than one ON ERROR statement in a program, the last one to execute remains in effect.

Do not use an ON ERROR statement in a multiple-line function. If you return from a multiple-line function using an ON ERROR statement, the return address is lost because the stack is reset. You can use the ON ERROR statement with the ERR and ERRL functions.

Example:

```
REM IF DATA.STRING$ IS NULL, ERROR AC OCCURS
    ON ERROR  GOTO 100
    ALPHA% = ASC(DATA.STRING$)
    PRINT ALPHA%
GOTO 200
    100 A$ = ERR
    PRINT A$
    I% = ERRL
    PRINT I%
200 END
```

OPEN Statement

The OPEN statement opens an existing disk file for reading or updating.

Syntax:

OPEN "*filespec*" [RECL *rec length*]
AS *file number* [BUFF*number of buffers*] [*mode*]

Explanation:

Use OPEN statements to open both stream and fixed disk files. If you specify a file that does not exist, the program detects an end-of-file condition.

To open an existing stream file, OPEN requires a filespec and a file number. The filespec can be a string expression, a variable, or a constant. The file number is a unique integer identification number ranging from 1 to the current implementation limit for the number of files accessible at one time. Refer to Appendix A of the Programming Guide for current limits. Place the file number in an OPEN statement after the keyword AS.

To access an existing fixed file, you must specify the fixed record length with the RECL parameter in addition to the filespec and file number. Assign the same record length that you assigned in the original CREATE statement.

The BUFF option assigns additional internal buffers. OPEN assumes a default value of one buffer if not specified otherwise. BUFF must specify 1 if you access a file randomly.

CBASIC supports three different modes for accessing files: LOCKED, UNLOCKED, and READONLY. Use the mode parameter under multiuser or concurrent operating systems. If you OPEN a file in LOCKED mode, no other program or user can access that file. UNLOCKED mode allows more than one program or user to access the file. READONLY files allow more than one program or user to read the file. Another program or user cannot modify the data inside a READONLY file.

Examples:

```
OPEN "SALES.APR"  AS 2

OPEN "B: QUESTION.DAT"  RECL 300  AS 18

OPEN ACCOUNT.MASTER$  AS 12  BUFF 4

OPEN "B:" + NAME$ + "." + LEFT$(STR$(CURR.WORK%), 3)
     AS CURR.WORK%
```

OUT Statement

The OUT statement sends an integer value to a specified CPU output port.

Syntax:

OUT *numeric expression, numeric expression*

Explanation:

The OUT function is hardware dependent and might not apply to certain microprocessors. The first expression must specify a valid output port number. CBASIC does not check the validity of the port number. The second expression specifies an eight-bit integer value to send.

Real numbers convert to integers.

Examples:

```
OUT 1, 58

OUT 3, 80H

OUT FRONT.PANEL%, RESULT%

OUT PORT%(SELECTED%), ASC("$")
```

PEEK Function

The PEEK function returns the contents of a memory location.

Syntax:

i% = PEEK (*numeric expression*)

Explanation:

The expression must evaluate to an absolute address for the computer you use. CBASIC does not check the validity of this memory address.

PEEK converts real-number expressions to integers.

Examples:

```
I% = PEEK(250)

CONTENTS% = PEEK(MEM.ADDR%)

FOR INDEX% = 1 TO PEEK%(BUFFER%)
    IN.BUFFER$(INDEX%) = CHR$(PEEK%(BUFFER%+INDEX%))
NEXT INDEX%
```

POKE Statement

The POKE statement stores one byte of data into a memory location.

Syntax:

POKE *numeric expression, numeric expression*

Explanation:

The first expression must evaluate to an absolute memory address for the computer you use. CBASIC does not check the validity of this memory address.

The second expression specifies the value to store. POKE converts this value into a one-byte integer.

Examples:

```
POKE 135, 54

POKE 1700, ASC("$")

FOR LOC% = 1 TO LEN(OUT.MSG$)
    POKE MSG.LOC%+LOC%, ASC(MID$(OUT.MSG$,LOC%,1))
NEXT LOC%
```

POS Function

The POS function returns the next column position to be printed on the console or printer.

Syntax:

i% = POS

Explanation:

POS returns the next output column for either the console or printer, depending on which output mode is in effect. POS determines the number of characters and spaces output to the console or printer since the last carriage return. POS returns that total plus 1 to indicate the next column available for output.

POS returns inaccurate values if you output cursor control characters or backspace characters.

Examples:

```
PRINT "The print head is at column: "; POS

IF (WIDTH.LINE - POS) < 15  THEN GOSUB LINEFEED
```

PRINT Statement

The PRINT statement outputs data to the console or printer.

Syntax:

PRINT [*expression*{*delimiter*;*expression*}*delimiter*]

Explanation:

PRINT outputs expressions to the console unless the LPRINTER statement is in effect.

Use any number of expressions with the PRINT statement; delimit each expression with a comma or semicolon. The comma tabs to the next column that is a multiple of 20 before the next expression prints. The semicolon allows expressions to print continuously on a line with no spaces in between. However, numeric expressions are always separated by one space.

The keyword PRINT with no expression list outputs a carriage return line-feed. The PRINT statement sends a carriage return line-feed after each execution unless a comma or semicolon follows the last expression.

Refer to Section 5 for more information on input and output.

Examples:

```
PRINT "This program calculates total profits."

PRINT QUANTITY%, PRICE, QUANTITY% * PRICE

PRINT "Today's date is: "; MONTH$; " "; DAY%; " "; YEAR%
```

PRINT USING Statement

The PRINT USING statement allows you to specify special formats for output data. The PRINT USING # variation directs formatted output to a disk file.

Syntax:

PRINT USING *format string*;[#*file number*,[*rec number*];]
expression {,*expression*}

Explanation:

The format string is a model for the output. A format string contains data fields and literal data. Data fields can be numeric or string types. Any character in the format string that is not part of a data field is a literal character. Format strings cannot be null strings. Table 3-1 describes characters that have special meaning in format strings.

Table 3-1. Special Characters in Format Strings

Character	*Meaning*
!	single-character string field
&	variable-length string field
/	fixed-length string field delimiter
#	digit position in numeric field
**	asterisk fill in numeric field
$$	float a $ in numeric field
.	decimal point position in numeric field
-	leading or trailing sign in numeric field
^	exponential position in numeric field
,	place comma every third digit before decimal point
\	escape character

The expression list tells which variables hold the data to be formatted. Separate each variable with a comma or semicolon. The comma does not cause automatic tabbing as it does with unformatted printing. PRINT USING matches each variable in the list with a data field in the format string. If there are more expressions than there are fields in the format string, execution is reset to the beginning of the format string.

While searching the format string for a data field, the type of the next expression in the list, either string or numeric, determines which data field to use. Section 5.3 has additional information on formatted printing.

Examples:

```
PRINT USING "###"; I%

ST$= "Total amount due is $$#,###.##"

PRINT USING ST$; TOTAL.DUE

PRINT USING "! ! !"; #15; "ALPHA", "BETA", "GAMMA"
```

PRINT # Statement

The PRINT # statement outputs data to a disk file.

Syntax:

PRINT # *file number*[,*rec number*];*expression*
{,*expression*}

Explanation:

The PRINT # statement writes expressions to the file specified by the file number. Each PRINT # statement executed creates a single record. Each expression used in the PRINT # statement creates a single field.

Use any number of expressions with the PRINT # statement and separate each one with a comma.

You can specify a random access record number for files that have a fixed record length. However, the amount of data written to fixed-length records must not exceed the record length specified in the RECL parameter in the CREATE or OPEN statement. You must add two bytes for the carriage return line-feed when determining the amount of data you can print to a record. Record numbers start with one, not zero.

Refer to Section 5 for more information on using disk files.

Examples:

```
CREATE "FILE.1"  AS 1
       A$ = "FIELD.ONE"
       B% = "22222"
PRINT #1; A$, B%

REM  STORE CURRENT VALUE IN RECORD 5
OPEN "UPDATE.DAT"  RECL 10  AS 15
     INPUT "Enter current value."; VALUE%
PRINT #15,5; VALUE
```

PUT Statement

The PUT statement writes one byte of data to a specified disk file.

Syntax:

PUT *file number expression*

Explanation:

Each execution of the PUT statement writes binary data for one byte to the file.

The expression can be any value between 0 and 255. Real expressions convert to integers. The file number is a unique identification number you assign to a file in a CREATE or OPEN statement.

Examples:

```
PUT 3, 255

PUT 20, ALPHA%
```

RANDOMIZE Statement

The RANDOMIZE statement seeds the random number generator for use with the RND function.

Syntax:

RANDOMIZE

Explanation:

An INPUT statement must precede any RANDOMIZE statement if your operating system does not support a time-of-day function. During program execution, the amount of time it takes a user to respond to the INPUT statement serves as a variable to seed the random number generator.

See the RND function for more information.

Examples:

```
INPUT "Type any character to continue."; LINE A$
RANDOMIZE
```

READ Statement

The READ statement sequentially assigns the constants in a DATA statement to variables.

Syntax:

READ *variable* {,*variable*}

Explanation:

CBASIC maintains a pointer to keep track of the next constant in the DATA statement constant list. Each time a READ statement executes, READ assigns a constant in the DATA statement to the next variable in the READ statement. Then, READ sets the pointer to the next constant in the DATA statement. A compiler error occurs if the READ statement attempts to read past the last constant.

READ statements must assign each constant to a variable with a matching data type. If the data types do not correspond, the READ statement might assign an unexpected value to a variable.

Refer to the DATA and RESTORE statements for further information.

Examples:

```
DATA 1, 2, 3.0
READ FIRST%, SECOND%, THIRD

FOR I% = 1 TO 5
    READ NAMES$
NEXT I%
DATA "BROWN", "BAILEY", "JOHNSON"
DATA "ERICSON", "PRINCE"
```

READ # Statement

The READ # statement reads data fields from a specified disk file into variables.

Syntax:

> READ # *file number*[,*rec number*];*variable*
> {,*variable*}

Explanation:

The READ # statement reads expressions from a disk file specified by the file number. The file number is a unique identification number assigned to a file in the CREATE or OPEN statement. [File numbers are limited by the current implementation value for the number of files allowed open at one time.] Each READ # statement executed reads data sequentially, field by field, into the variables. READ # assigns one field of data to each variable. When reading a fixed file, the number of variables in the READ # statement must be less than or equal to the number of fields in each record.

You can specify a random access record number for files that have a fixed record length. Record numbers start with one, not zero.

Refer to Section 5 for more information on using disk files.

Examples:

```
OPEN "B: FILE.DAT"  AS 8
WHILE NUMBER.OF.FIELDS%
     READ #8; FIELDS$
     PRINT FIELDS%
     NUMBER.OF.FIELDS% = NUMBER.OF.FIELDS% - 1
WEND

REM  READ RECORD 3...FIELDS ONE AND TWO
IF END # 15  THEN 700
OPEN "FILE.1"  AS 15
     READ #15, 3; FIELD1$, FIELD2%
```

READ # LINE Statement

The READ # LINE statement reads one complete line of data from a file and assigns the information to a string variable.

Syntax:

READ # *file number*, [*record number*]; LINE *string variable*

Explanation:

You can use only one variable after the keyword LINE. The variable must be a string variable.

The READ # LINE statement can read records accessed sequentially or randomly.

Examples:

```
READ #FILE.NO%; LINE D$

READ #F%, REC%; LINE X$
```

REAL Statement

The REAL statement is a declaration statement that specifies a real number data type for variables and function names.

Syntax:

REAL *identifier* [,*identifier*]

Explanation:

Use REAL statements in the declaration group of a program or multiple-line function. Declaration statements override the default data types specified with the last character in an identifier.

To use an array identifier in a REAL statement, place the number of subscripts in parentheses after the array name.

Refer to Section 2 for more information on declarations and identifiers.

Examples:

```
REAL X

REAL X, Y, Z

REAL COORD(2), X
STRING NAMES$(1)
```

REM Statement

The REM statement documents a source program to improve readability.

Syntax:

REM *any characters carriage return*

REMARK *any characters carriage return*

Explanation:

The REM statement allows program documentation. REM statements do not affect the size of a compiled program. Adding comments to a program with the REM statement makes the program easier to understand and maintain. The compiler ignores everything that follows the keywords REM or REMARK on a physical line.

The continuation character allows a remark to span more than one physical line. The REM statement can appear on the same line with other statements, but must be the last statement on a logical line. A colon is not required to separate the REM from executable statements.

Examples:

```
REM  THIS IS A REMARK

remark  this is also a remark

TAX = 0.15 * INCOME   REM LOWEST TAX RATE
            REM THIS SECTION CONTAINS THE \
            TAX TABLES FOR CALIFORNIA
```

RENAME Function

The RENAME function allows you to change the name of a disk file during program execution.

Syntax:

i% = RENAME (*filespec,filespec*)

Explanation:

The first filespec is the new name assigned to the file. The second filespec is the file to rename. The RENAME function returns an integer value. RENAME returns a -1 if the function is successful and a 0 if the function fails. Assigning a file a name that already exists causes an execution error.

Be sure to close a file before renaming it. Otherwise, when files automatically close at the end of processing, CBASIC tries to close the renamed file under the old name but cannot find it.

Examples:

```
DUMMY% = RENAME("PAYROLL.MST", "PAYROLL.$$$")

IF RENAME(NEWFILE$, OLDFILE$) THEN RETURN
```

RESTORE Statement

A RESTORE statement allows rereading of the constants in DATA statements.

Syntax:

RESTORE

Explanation:

A RESTORE statement repositions the DATA statement pointer to the beginning of the DATA statement constant list. Use a RESTORE statement to reread the constants in the DATA statements. The CHAIN statement automatically executes a RESTORE statement.

Refer to the DATA and READ statements for more information.

Examples:

```
500 RESTORE

IF  END.OF.DATA%  THEN RESTORE
```

RETURN Statement

The RETURN statement transfers control from a subroutine back to the calling program.

Syntax:

```
    RETURN
```

Explanation:

The RETURN statement transfers execution of a program to the location saved on top of the return stack. The subroutine call can be a GOSUB statement, an ON...GOSUB statement, or a call to a multiple-line function.

RETURN passes a value back to the main program when returning from a multiple-line function.

Examples:

```
500 RETURN

IF ANSWER.VALID%  THEN RETURN
```

RIGHT$ Function

The RIGHT$ function returns a string consisting of the rightmost characters in a string.

Syntax:

a$ = RIGHT$(*string expression,numeric expression*)

Explanation:

The numeric expression is a positive value specifying the number of characters from the string expression to return. If the numeric expression is negative, an execution error occurs. Real expressions convert to integers. RIGHT$ returns a null string if the numeric expression equals zero. RIGHT$ returns the entire string if the numeric expression specifies more characters than the string contains.

Examples:

```
IF RIGHT$(ACCOUNT.NO$,1) = "0" THEN \
      TITLE.ACCT% = TRUE

NAME$ = RIGHT$(NAME$,LEN(NAME$)-LEN(FIRST.NAME$))
```

RND Function

The RND function generates and returns a random number.

Syntax:

x = RND

Explanation:

RND returns a uniformly distributed random number between 0 and 1. The RANDOMIZE statement seeds a random number generator to avoid identical sequences of random numbers. RND returns a real number.

Refer to the RANDOMIZE statement for further information.

Examples:

```
DIE%=INT%(RND*6.)+1

IF RND > .5 THEN \
   HEADS% = TRUE%\
ELSE \
   TAILS% = TRUE%
```

SADD Function

The SADD function returns the address of a specified string.

Syntax:

i% = SADD$(*string variable*)

Explanation:

Strings are stored as a sequential list of ASCII characters. The first two bytes hold the length of the string followed by the actual ASCII values. The length is stored as an unsigned binary integer. SADD returns an integer equal to the address of the first byte of the length.

If the expression is a null string, SADD returns a zero.

Examples:

The following statements place the address of STRING$ into the address stored in PARM.LOC%:

```
POKE PARM.LOC%,SADD(STRING$) AND OFFH
POKE PARM.LOC%+1,SADD(STRING$)/256
```

SGN Function

The SGN function returns an integer value representing the algebraic sign of a number.

Syntax:

i% = SGN(*numeric expression*)

Explanation:

SGN returns a -1 if the expression is negative, a 0 if the expression is zero, and a +1 if the expression is greater than zero.

Real number expressions convert to integers.

Examples:

```
IF SGN(BALANCE) <>  0 THEN \
   OUTSTANDINGBAL% = TRUE%

IF SGN(BALANCE) = -1 THEN \
   OVERDRAWN% = TRUE%
```

SHIFT Function

The SHIFT function returns an integer that is arithmetically shifted a specified number of positions to the right.

Syntax:

i% = SHIFT(*numeric expression, numeric expression*)

Explanation:

The first expression specifies the value that the function shifts. The second expression specifies the number of positions to shift the value in the first expression to the right. SHIFT returns a 0 if the second expression is greater than 15.

The function shifts arithmetically. SHIFT divides the value in the first expression by 2 for each position shifted to the right. The function retains the arithmetic sign of the first expression after a shift. Therefore, if the first expression is positive, zeros shift into the high-order positions. When the value is negative, ones shift into the high-order positions.

Examples:

```
SHIFT (12345, 3)

SHIFT (FFH, 2)

SHIFT (1011011B, 1)
```

SIN Function

The SIN function returns the sine of a number.

Syntax:

x = SIN(*numeric expression*)

Explanation:

The SIN function assumes the expression is an angle in radians. Integers convert to real numbers. The sine value returned is a real number.

Examples:

```
FACTOR(Z) = SIN(A - B/C)

IF SIN(ANGLE/(2.0 * PI)) = 0.0 THEN \
   PRINT "HORIZONTAL"
```

SIZE Function

The SIZE function returns the number of 1-kilobyte blocks in a specified file.

Syntax:

i% = SIZE(*filespec*)

Explanation:

The filespec can specify ambiguous filenames if your operating system supports ambiguous references.

The SIZE function returns the number of bytes allocated for all files specified in the filespec, divided by 1024.

The SIZE function returns an unsigned integer.

Examples:

```
SIZE("NAMES.BAK")
```

```
SIZE(COMPANY$ + DEPT$ + ".NEW")
```

```
SIZE("*.*")
```

```
SIZE("*.BAS")
```

```
70   REM  TESTING FOR ENOUGH SPACE
SIZE.OF.OUTPUT% = 1.25 * SIZE("A:INPUT")
FREE.BLOCKS% = 241  - SIZE("B:*.*")
IF FREE.BLOCKS% < SIZE.OF.OUTPUT%  THEN \
   ENOUGH.ROOM% = FALSE% \
   ELSE ENOUGH.ROOM% = TRUE%
RETURN
```

SQR Function

The SQR function returns the square root of a number.

Syntax:

x = SQR(*numeric expression*)

Explanation:

SQR returns a real number. Integers convert to real numbers. If the expression is negative, an execution error occurs.

Examples:

```
HYPOT = SQR((SIDE1^2.0)+(SIDE2^2.0))

PRINT USING \
      "THE SQR ROOT OF X IS: ####.##"; SQR(X)
```

STOP Statement

A STOP statement terminates program execution and returns control to the operating system.

Syntax:

STOP

Explanation:

The STOP statement closes all open files and returns control to the operating system. Any number of STOP statements can appear in a program.

Examples:

```
400 STOP

IF STOP.REQUESTED THEN STOP
```

STR$ Function

The STR$ function converts a number to a string.

Syntax:

a$ = STR$(*numeric expression*)

Explanation:

STR$ converts the expression to a string of characters identical to the digits in the expression. STR$ deletes the blank space that follows a number. Integers convert to real numbers.

Examples:

```
PRINT STR$(NUMBER)
```

```
IF LEN(STR$(VALUE))>5 THEN ED$="#######"
```

STRING Statement

The STRING statement is a declaration statement that specifies a string data type for variables and function names.

Syntax:

STRING *identifier* [,*identifier*]

Explanation:

Use STRING statements in the declaration group of a program or multiple-line function. Declaration statements override the default data type specified with the last character in an identifier.

To use an array identifier in a STRING statement, place the number of subscripts in parentheses following the array name.

Refer to Section 2 for more information on declarations and identifiers.

Examples:

```
STRING A$

STRING A, B, C

STRING NAMES$, TITLE$
REAL SALARY$
```

STRING$ Function

The STRING$ function returns a string that consists of one string copied a specified number of times.

Syntax:

a$ = STRING$ (*numeric expression, string expression*)

Explanation:

The numeric expression specifies the number of times to copy the string in the string expression. The length of the returned string equals the length of the string in the string expression, multiplied by the numeric expression.

STRING$ reduces memory fragmentation when building large strings and executes significantly faster than building a string using concatenation.

Numeric expressions that evaluate to real numbers convert to integers.

Examples:

`STRING$ (3, "AB")` returns ABABAB

`STRING$ (0, A$)` returns a null string

`STRING$ (I%, "")` returns a null string

TAB Function

The TAB function moves the cursor or print head to a specified column.

Syntax:

TAB (*numeric expression*)

Explanation:

The expression specifies a column number. If the value of the expression is less than or equal to the current print position, TAB sends a carriage return line-feed before tabbing to the specified column. The expression cannot exceed the line width.

TAB outputs blank characters until the cursor or print head reaches the desired position. An incorrect TAB column might result if a program outputs cursor or printer control characters.

Do not use TAB with PRINT # statements. Use the TAB function in PRINT or PRINT USING statements only. TAB rounds real expressions to the nearest integer.

Examples:

```
PRINT TAB(15);"X"

PRINT "THIS IS COL. 1";TAB(50);"THIS IS COL. 50"

PRINT TAB(X%+Y%/Z%);"!";TAB(POS%+OFFSET%);

PRINT TAB(LEN(STR$(NUMBER)));"*"
```

TAN Function

The TAN function returns the tangent of a number.

Syntax:

x = TAN(*numeric expression*)

Explanation:

The TAN function assumes the expression is a value in radians. Integers convert to real numbers. The tangent value is a real number.

Examples:

```
POWER.FACTOR = TAN(PHASE.ANGLE)

QUIRK = TAN(X - 3.0 * COS(Y))
```

UCASE$ Function

The UCASE$ function translates lower-case characters to upper-case.

Syntax:

a$ = UCASE$(*string expression*)

Explanation:

UCASE$ returns a string with all of its lower-case characters converted to upper-case. The function does not change the original string.

Examples:

```
IF UCASE$(ANS$) = "YES" THEN\
   RETURN \
ELSE STOP

NAME$ = UCASE$(NAME$)
```

UNLOCK Function

The UNLOCK function unlocks a record, allowing modification of the data in the record.

Syntax:

i% = UNLOCK(*file number, record number*)

Explanation:

The UNLOCK function returns the value that the operating system returns after attempting to unlock a record. Normally, a zero indicates that the record is unlocked. UNLOCK returns a value of zero in systems that do not support record locking.

To LOCK or UNLOCK a record, the file must be a fixed file accessed in the UNLOCKED mode. Refer to the LOCK function, CREATE statement, and OPEN statement for more information.

Examples:

```
IF UNLOCK(1, REC%)  THEN CALL ERROR.MSG
```

VAL Function

The VAL function converts a digit string to a real number.

Syntax:

x = VAL(*digit string expression*)

Explanation:

VAL processes from left to right until it reaches the end of the string or until it encounters a character that is not a digit.

VAL returns a zero if the string is null. A plus or minus sign can precede the digit string.

Examples:

```
PRINT ARRAY$(VAL(IN.STRING$))

ON VAL(PROG.SEL$) GOSUB 10, 20, 30, 40, 50
```

VARPTR Function

The VARPTR function returns the address of a variable.

Syntax:

i% = VARPTR (*variable*)

Explanation:

VARPTR returns the actual address for an unsubscripted numeric variable. For string variables, however, VARPTR returns the address of a 16-bit pointer. The actual location of the string varies because strings are allocated dynamically, but the value that VARPTR returns does not change during program execution. If a variable is in COMMON, the VARPTR value location does not change after chaining.

For subscripted variables, VARPTR returns the address of a pointer to an array in the Free Storage Area. Refer to the Programming Guide for a description of the Free Storage Area.

Examples:

```
A% = VARPTR(X)

PRINT VARPTR (I%)

DIM A$(10)
CALL PROCESS (VARPTR (A$))
```

WEND Statement

A WEND statement denotes the end of a WHILE/WEND loop.

Syntax:

WEND

Explanation:

The WEND statement sends control to the beginning of the WHILE/WEND loop until the WHILE expression evaluates to logical false (0).

Branching to a WEND statement sends control to the corresponding WHILE statement.

Examples:

```
WHILE VALUE > 1
      PRINT "X"
WEND

WHILE ACCOUNT.IS.ACTIVE%
      GOSUB 100   REM ACCUMULATE INTEREST
WEND

TIME = 0.0
TIME.EXPIRED% = FALSE%
WHILE TIME < LIMIT
      TIME = TIME + 1.0
      IF CONSTAT% THEN \
         RETURN REM ANSWERED IN TIME
WEND
TIME.EXPIRED% = TRUE%
RETURN
```

WHILE Statement

The WHILE statement specifies the conditional expression that controls a WHILE/WEND loop.

Syntax:

WHILE *logical expression*

Explanation:

All statements between a WHILE statement and a corresponding WEND statement execute until the value of the expression following the keyword WHILE evaluates to logical false (0).

Real expressions convert to integers. Integer expressions reduce execution time. WHILE/WEND loops can be nested.

Examples:

```
PRINT "PRESS ANY KEY TO CONTINUE"
      WHILE NOT CONSTAT%
      WEND

WHILE NUMBER.OF.FIELDS%
      READ #FILE.DAT; FIELD$
      PRINT FIELD$
      NUMBER.OF.FIELDS% = NUMBER.OF.FIELDS% - 1
WEND

WHILE FILE.EXISTS%
      WHILE TRUE%
            IF ARG$ = ACCT$  THEN \
               ACTIVITY% = TRUE% :\
               RETURN
            IF ARG$ < ACCT$  THEN \
               ACTIVITY% = FALSE% :\
               RETURN
            GOSUB 3000  REM READ ACCT$ REC
      WEND
WEND
ACTIVITY% = FALSE%
```

Section 4
Defining and Using Functions

A function is a named, isolated portion of a program that other parts of the program can invoke to compute a value or perform some operation. To execute, a function must be referenced by name. You cannot call a function with a GOSUB or GOTO statement. There are only two ways to invoke a function:

- with a CALL statement
- in an expression

When a function's name is in an expression, the function returns a value that the expression uses as if the function were a variable. Functions can have parameters that are like variables whose values you specify in the function call.

COS, MATCH, and INP are examples of functions that are predefined in the CBASIC language. You can define your own functions to perform tasks in a program that predefined functions cannot perform. A user-defined function passes parameters to one or more executable statements. The function name serves as a variable to pass a computed value back to the calling statement. Once defined, you can call a user-defined function any number of times in the program.

4.1 Function Names

A function name is a valid CBASIC identifier. However, only the first six characters distinguish one function name from another. The function name identifies a function and serves as a variable to hold the value that the function passes back to the calling statement. The form of the function name determines which type of value the function returns.

- Names for string functions end with $.
- Names for integer functions end with %.
- Names for real number functions do not end with $ or %.

You must use a function name to define a function and to call a function from another location in a program. The following examples are valid function names:

```
PROPER.FUNCTION.NAMES

TRUNCATE$

W1234%
```

4.2 Function Definitions

A function definition must occur in a program before making a function call. Use the DEF statement to define a function. CBASIC supports two types of function definitions: single-line and multiple-line.

4.2.1 Single-line Functions

A single-line function performs computations that do not span more than one logical line. You can accomplish more complex programming tasks with multiple-line functions. Single-line function definitions use an equal sign followed by an expression. The expression contains the actual process that the single-line function performs. The data types in the expression must match the data type in the function name. Use the following format when defining single-line functions:

DEF *function.name* [(formal parameters)] = *expression*

A formal parameter holds a place for an actual parameter that you specify in a function reference. A formal parameter is either a string variable or a numeric variable; it is never a constant. Formal parameters must have the same data type as the actual parameters in the function reference. CBASIC considers formal parameters local to the function. Local variables are independent of the rest of a program. CBASIC functions pass parameters by value.

The following examples show single-line function definitions:

```
DEF 25 = RND * 25.0

100 DEF CALC.HYPOT(SIDE1,SIDE2)= \
    SQR((SIDE1 * SIDE1) + (SIDE2 * SIDE2))

DEF LEFT.JUSTIFY$(A$,LEN%)=LEFT$(A$+BLNKS$,LEN%)
```

4.2.2 Multiple-line Functions

Multiple-line function definitions contain data declarations and executable statements. The function definition begins with a DEF statement and ends with a FEND statement. You can use RETURN statements in the body of the function. Both RETURN and FEND stop function execution, pass the function value, and send control back to the statement following the calling statement. Use any number of RETURN statements, but be sure one FEND statement appears last in a multiple-line function definition. Use the following format when defining multiple-line functions:

```
DEF function.name [(formal parameters)]
   data declarations
   .
   .
   CBASIC statements
   .
   .
   RETURN
FEND
```

The function reference assigns the value of the actual parameters to the formal parameters in the function definition. All formal parameters and variables in a declaration are local to the function. All labels defined in a function are local to the function. The following two examples show multiple-line function definitions:

```
DEF GET.AMOUNT(CUST.NO%)
    READ # CUSTOMER.INF; CUSTNO%, AMOUNT
    GET.AMOUNT = AMOUNT
FEND
```

```
DEF AREA.CIR(DIA, MIN)
  REAL  AREA.CIR, DIA, MIN
    IF DIA < MIN  THEN \
RETURN \
    ELSE RAD = D/2
AREA.CIR = (R * 3.14159)^2
FEND
```

The following rules apply to multiple-line functions:

- Function definitions cannot be nested. However, a function can call another function.
- COMMON statements cannot appear in a function definition.
- Functions cannot have GOTO statements that reference a line outside the function.
- A DIM statement in a function allocates a new array on each execution of the function. Data stored in an array from a previous execution is lost. Arrays in multiple-line functions are global to an entire program.

4.3 Function References

Reference user-defined functions in any CBASIC statement or expression. The CALL statement can reference multiple-line functions only. The number of actual parameters in the function reference must match the number of formal parameters in the function definition. The function substitutes the current value of each actual parameter in the reference statement for the formal parameters in the function definition. The following are examples of function references:

```
300 PRINT FUNCTION(250, 1429)

CALL FUNCTION.NUMBER.ONE(WHOLESALE, RETAIL, DIFF)

IF  LENGTH%("INPUT DATA",X$,Q) < LIMIT%  THEN
          GOSUB 3000

WHILE FN.ALTITUDE(CURR.ALT%) > MINIMUM.SAFE
          CURR.ALT%=INP(ALT.PORT%)
WEND
```

4.4 Public and External Functions

Multiple-line functions can compile separately to form individual program modules. The PUBLIC and EXTERNAL keywords provide a method to access the same multiple-line function from different modules.

To execute a multiple-line function from a different program module, declare the function public in the definition. Enter the keyword PUBLIC after the list of formal parameters in the DEF statement, as shown in the following example.

DEF *function.name* [(formal parameters)] PUBLIC
    data declarations
    .
    .
    CBASIC statements
    .
    .
FEND

The complete definition for the public function appears in one module. However, any module that references a public function in another module must contain an abbreviated definition of that function specified as external. An external function definition does not contain the executable statement group. Other modules can reference the public function in an expression or with a CALL statement. Enter the keyword EXTERNAL after the list of formal parameters in the abbreviated definition, as shown in the following example.

```
DEF function.name [(formal parameters)] EXTERNAL
     data declarations
FEND
CALL function.name [(actual parameters)]
```

The link editor links the external function call to the public function for execution. CBASIC does not check parameter data types between modules. Be sure that corresponding parameter data types match. Parameter names do not have to match. However, the function names used in external functions must match the name of the corresponding public function.

End of Section 4

Section 5
Input and Output

CBASIC uses the operating system to control input and output for interaction between programs, consoles or terminals, printers, and disk drives.

5.1 Console Input and Output

CBASIC reads input from the console one line at a time. Therefore, all CP/M line-editing functions, such as CTRL-U and DELETE, remain in effect. CTRL-C entered from the keyboard terminates a program but does not close files being accessed. Three functions, CONCHAR%, INKEY, and CONSTAT%, use direct console input and output. Use the following statements and predefined functions to input data from a console device. Refer to Section 3 for more detailed descriptions of statements and functions.

- INPUT statements query the user for information during program execution. You can enter any number of input values with an INPUT statement. You can have a prompt message displayed if you desire.
- INPUT LINE works like an INPUT statement, but accepts only one variable for data to be entered. All characters entered in response to INPUT LINE are interpreted as one string.
- CONSTAT% is a predefined function that determines console status. The function returns a logical true value (-1) if a character is ready at the console, and a logical false value (0) if a character is not ready.
- CONCHAR% is a function that waits for an entry from the keyboard and returns an eight-bit ASCII representation of the character entered. CONCHAR% echoes characters of ASCII decimal value greater than 31.

The following CBASIC statements and predefined functions control console output.

- The CONSOLE statement restores printed output to the console device.
- The TAB predefined function moves the console cursor to a specified position on the screen. TAB also works with printers. Use TAB only in the PRINT statement.
- The POS predefined function returns the next available position on the console screen to be printed.

5.2 Printing

CBASIC provides three statements to control output to a line printer.

- PRINT outputs data to a console or printer.
- LPRINTER directs all output PRINT statements to the line printer or list device.
- PRINT USING allows formatting of printed output to the console or printer.

5.3 Formatted Printing

The PRINT USING statement allows you to specify special formats for output data. You can output formatted data to the console or line printer with CONSOLE or LPRINTER. The PRINT USING # variation directs formatted output to a disk file. Write a PRINT USING statement as follows:

```
PRINT USING format string.[#file number[,rec number];]
                <expression list>
```

The format string is a model or image for the output. A format string contains data fields and literal data. Data fields can be numeric or string type. Any character in the format string that is not part of a data field is a literal character. Format strings cannot be null string expressions. Table 5-1 describes characters that have special meanings in format strings:

Table 5-1. Special Characters in Format Strings

Character	*Meaning*
!	single-character string field
&	variable-length string field
/	fixed-length string field delimiter
#	digit position in numeric field
**	asterisk fill in numeric field
$$	float a $ in numeric field
.	decimal point position in numeric field
-	leading or trailing sign in numeric field
^	exponential position in numeric field
,	place comma every third digit before decimal point
\	escape character

The expression list tells which variables hold the data to format. Separate each variable with a comma or semicolon. The comma does not cause automatic tabbing as it does with unformatted printing. PRINT USING matches each variable in the list with a data field in the format string. If there are more expressions than there are fields in the format string, execution resets to the beginning of the format string.

While searching the format string for a data field, the type of the next expression in the list, either string or numeric, determines which data field to use. For example, if PRINT USING encounters a numeric data field while outputting a string, the statement treats characters in the numeric data field as literal data. An error occurs if there is no data field in the format string of the type required.

5.3.1 String Character Fields

Specify a one-character string data field with an exclamation point, !. PRINT USING outputs the first character of the next expression statement list.

For example,

```
F.NAME$="Lynn":M.NAME$ = "Marion":L.NAME$= "Kobi"
PRINT USING "!. !. &"; F.NAME$,M.NAME$,L.NAME$
```

outputs

```
L. M. Kobi
```

In this example, PRINT USING treats the period as literal data.

5.3.2 Fixed-length String Fields

Specify a fixed-length string field of more than one position with a string of characters enclosed between a pair of slashes. The width of the field is equal to the number of characters between the slashes, plus two. Place any characters between the slashes to serve as fill. PRINT USING ignores fill characters for fixed-length string fields.

A string expression from the print list is left-justified in the fixed field and, if necessary, padded on the right with blanks. PRINT USING truncates a string longer than the data field on the right.

For example,

```
FOR1$ = "THE PART REQUIRED IS /...5....0....5/"
PART.DESCRP$ = "GLOBE VALVE, ANGLE"
PRINT USING FOR1$; PART.DESCRP$
```

outputs

```
THE PART REQUIRED IS GLOBE VALVE, ANG
```

Using periods and numbers between the slashes makes it easy to verify that the data field is 16 characters long. Periods and numbers do not effect the output.

5.3.3 Variable-length String Fields

Specify a variable-length string field with an ampersand, &. This results in a string output exactly as defined.

For example,

```
COMPANY$ = "SMITH INC."
PRINT USING "& &"; "THIS REPORT IS FOR",COMPANY$
```

outputs

```
THIS REPORT IS FOR SMITH INC.
```

The following example shows how a string can be right-justified in a fixed-length string field using a variable-length string field.

```
FLD.S% = 20
BLK$ = "                   "
PHONE$ = "408-649-3896"
PRINT USING "#&"; RIGHT$(BLK$ + PHONE$, FLD.S$)
```

outputs

```
#        408-649-3896
```

The preceding example uses the # as a literal character because the print list contains only a string expression. A # can also indicate a numeric data field.

5.3.4 Numeric Data Fields

Specify a numeric data field with a pound sign, #, to indicate each digit required in the resulting number. One decimal point can also be included in the field. Values are rounded to fit the data field. Leading zeros are replaced with blanks. When the number is negative, PRINT USING prints a minus sign to the left of the most significant digit. A single zero prints to the left of the decimal point for numbers less than one if you provide a position in the data field.

For example,

```
X = 123.7546
Y = -21.0
FOR$ = "####.####   ###.# ###"
PRINT USING FOR$; X, X, X
PRINT USING FOR$; Y, Y, Y
```

outputs

```
123.7546   123.8   124
-21.0000   -21.0   -21
```

Tell PRINT USING to print numbers in exponential format by appending one to four up arrows, ˆ, to the end of the numeric data field. For example,

```
X = 12.345
PRINT USING "#.###^^    "; X, -X
```

outputs

```
1.235E 01   -.123E 02
```

PRINT USING reserves four positions for the exponent regardless of the number of up arrows used in the field.

If one or more commas appear embedded in a numeric data field, the number prints with commas between each group of three digits that precede the decimal point. For example,

```
PRINT USING  "##,###    "; 100, 1000, 10000
```

outputs

```
100     1,000    10,000
```

PRINT USING includes each comma that appears in the data field in the width of the field. You need only one comma to obtain embedded commas in the output; however, placing each comma in the data field at the specified position clarifies the formatting statement.

For example, the following data fields produce the same results, except the width of the first field allows only nine output digits. The second field allows ten digits.

```
#,########
#,###,###,###
```

Commas do not print if you use the exponent option. In this case, PRINT USING treats commas as pound signs, #.

You can use asterisk filling in a numeric data field by appending two asterisks to the beginning of the data field. You can float a dollar sign by appending two dollar signs to the beginning of the data field. Do not use the exponential format with either asterisk filling or floating dollar signs. PRINT USING includes a pair of asterisks or dollar signs in the count of digit positions available for the field. The asterisks or dollar signs appear in the output if there is enough space. The dollar sign does not print if the number is negative.

For example,

```
COST = 8742937.56
PRINT USING "****,######.##   "; COST, -COST
PRINT USING "$$##,######.##   "; COST, -COST
```

outputs

```
**8,742,937.56   *-8,742,937.56
 $8,742,937.56    -8,742,937.56
```

PRINT USING outputs a number with a trailing sign instead of a leading sign if the last character in the data field is a minus sign. A blank replaces the minus sign in the output if the number is positive. For example,

```
PRINT USING  "###-###^^^^- "; 10, 10, -10, -10
```

outputs

```
10  100E-01    10- 100E-01-
```

PRINT USING fixes the sign position as the next output position if a minus sign is the first character in a numeric data field. If the number is positive, a blank prints instead of the minus sign. For example,

```
PRINT USING  "-####    "; 10, -10
```

outputs

```
10   -   10
```

If a number does not fit in a numeric data field without truncating digits before the decimal point, a percent sign, %, precedes the number in the standard format. For example,

```
X = 132.71
PRINT USING  "##.#    ###.#"; X,X
```

outputs

```
% 132.71  132.7
```

5.3.5 Escape Character

You can use a special format string character as literal data in a data field with the escape character. A backslash, \ , signals PRINT USING to treat the next consecutive character as a literal character. For example, a pound sign, #, can precede a number. For example,

```
ITEM.NUMBER = 31
PRINT USING "THE ITEM NUMBER IS\ ###"; ITEM.NUMBER
```

outputs

```
THE ITEM NUMBER IS #31
```

Two consecutive backslashes cause a one backslash to print as a literal character. An escape character cannot be the last character in a format string.

5.4 File Input and Output

CBASIC uses the operating system file accessing routines to store and retrieve data in disk files. All data is represented in character format using the ASCII code. Programs can create, open, read, write, and close data files with the following CBASIC statements and functions. Each statement is described in more detail in Section 3.

- CREATE creates a new file on disk with no information in it. The CREATE statement erases a preexisting file of the same name before creating the new file.
- OPEN opens an existing disk file for reading or updating. If the file does not exist, the program detects an end-of-file condition.
- IF END transfers program execution to a specified label when a file access exception occurs.
- READ # accesses a specified file and assigns the data sequentially, field by field, into specified variables. Data can also be accessed from a specified record.
- READ LINE reads one complete record of data from a file and assigns the information to a string variable.
- PRINT # outputs data to a specified file and assigns the data sequentially into fields from specified expressions. Data can also be output to a specified record.
- PRINT USING # outputs data to a specified file using formatted printing options.
- CLOSE closes disk files. The specified files are no longer available for input or output until reopened.
- DELETE deactivates a file from processing and erases it from the disk directory.
- GET reads one byte of data from a specified disk file.
- PUT writes one byte of data to a specified disk file.
- SIZE returns the number of 1-kilobyte blocks in a specified file.
- RENAME allows you to change the name of a disk file during program execution.

5.5 File Organization

CBASIC organizes information on a disk surface into three levels: files, records, and fields.

- Files consist of one or more records.
- Records are groups of fields. Each record is delimited by a carriage return linefeed.
- Fields are the individual data items. Each field in a record is delimited by a comma.

CBASIC supports two types of data files on disk: stream and fixed.

5.5.1 Stream Files

Sequential or stream organization is performed on a strict field-by-field basis. The PRINT # statement writes each field to the disk in a continuous stream. Each data item uses only as much space as needed. Each PRINT # statement executed creates a single record. Each variable used in the PRINT # statement creates a single field. Individual record lengths vary according to the amount of space occupied by the fields. There is no padding of data space. The following diagram shows a stream file composed of three records.

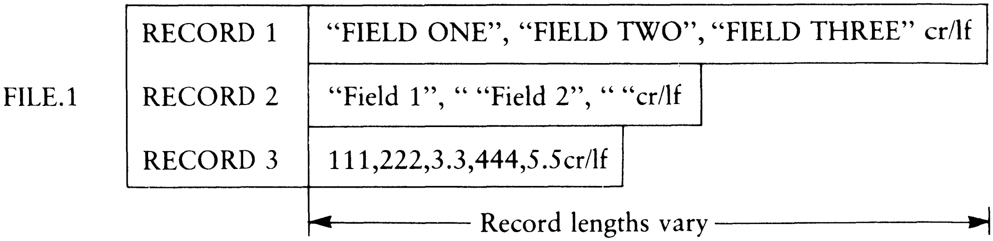

FILE.1
RECORD 1
“FIELD ONE”, “FIELD TWO”, “FIELD THREE” cr/lf
RECORD 2
“Field 1”, “ “Field 2”, “ “cr/lf
RECORD 3
111,222,3.3,444,5.5cr/lf
Record lengths vary

Figure 5-1. Sequential File

Field three in record two is a null string. Commas serve as delimiters, but are considered string characters when embedded in a pair of quotation marks. Quotation marks are not considered string characters when embedded in a string. Quotation marks are always considered as string delimiters in files.

The following CBASIC program creates the stream file diagramed in Figure 5-1.

```
CREATE  "FILE.1"  AS 1
       A$ = "FIELD ONE"
       B$ = "FIELD TWO"
       C$ = "FIELD THREE"
       D$ = "FIELD 1"
       E$ = "FIELD 2"
       F$ = ""
       G% = 111
       H% = 222
       I  = 3.3
       J% = 444
       K  = 5.5
PRINT #1; A$, B$, C$
PRINT #1; D$, E$, F$
PRINT #1; G%, H%, I, J%, K
CLOSE 1
END
```

The three PRINT statements correspond to the three records, and each variable corresponds to a field.

When you access stream files, each field is read sequentially, one at a time, from the first to the last. The READ # statement considers a field complete when it encounters a comma, a terminating quotation mark for string fields, or a carriage return line-feed. The following program reads the fields in FILE.1 sequentially and prints them on the console screen.

```
IF END #19  THEN 100
OPEN "FILE.1"  AS 19
    FOR I% = 1 TO 11
      READ #19; FIELDS$
      PRINT FIELDS$
    NEXT I%
100 END
```

Any type of variable can be used in the READ # statement in a sequential access. Executing the preceding program outputs the following display:

```
FIELD ONE
FIELD TWO
FIELD THREE
FIELD 1
FIELD 2

111
222
3.3
444
5.5
```

5.5.2 Fixed Files

Fixed files offer the advantage of random access, which is the ability to access any record in a file directly. Record lengths are fixed. Data space between the end of the last field and the carriage return line-feed is padded with blanks. The carriage return line-feed occupies the last two bytes of the record. The number of bytes occupied by the fields, field delimiters, and the carriage return line-feed cannot exceed the specified record length. Figure 5-2 shows a fixed file composed of three records.

	RECORD 1	“FIELD ONE”, “FIELD TWO”, “FIELD THREE”	cr/lf
FILE.2	RECORD 2	“FIELD 1,” “FIELD TWO”,” ”	cr/lf
	RECORD 3	111,222,3.3,444,5.5	cr/lf
		◄—— Record lengths fixed ——►	

Figure 5-2. Relative File

The same rules regarding commas, quotation marks, and null strings in stream files apply to fixed files. The following program creates the fixed file diagramed in Figure 5-2.

```
CREATE  FILE.2  RECL 40  AS 2
      A$ = "FIELD ONE"
      B$ = "FIELD TWO"
      C$ = "FIELD THREE"
      D$ = "FIELD 1
      E$ = "FIELD 2
      F$ = " "
      G% = 111
      H% = 222
      I  = 3.3
      J% = 444
      K  = 5.5
PRINT #2,1; A$, B$, C$
PRINT #2,2; D$, E$, F$
PRINT #2,3; G%, H%, I, J%, K
LOSE 2
END
```

To access a fixed file randomly, specify an actual record number. Enter the record number in all PRINT # and READ # statements after the file identification number. Separate the two numbers with a comma. In the following example, 5 is the record number.

```
PRINT #2,5; VARIABLE1%, VARIABLE2%
```

CBASIC locates each record on a randomly accessed file by taking the record number, subtracting 1, and multiplying that difference by the record length. The result is a byte displacement value for the desired record measured from the beginning of the file. The record to be accessed must be specified in each READ # or PRINT # statement executed. Each READ # and PRINT # statement executed accesses the next specified record. The following program reads the first three fields from record three in FILE.2.

```
IF END #20  THEN 200
OPEN "FILE.2"  RECL 40  AS 20
     READ #20,3; FIELD1$, FIELD2$, FIELD3
     PRINT FIELD1$, FIELD2$, FIELD3
200 END
```

The data types of the variables in the READ # statement must match the data contained in the fields being read. Executing the above program outputs the following display on screen.

```
111                222                 3.3
```

End of Section 5

Appendix A
CBASIC Compiler Reserved Words

CBASIC Compiler Reserved Words

ABS
AND
AS
ASC
ATTACH
ATN
BUFF
CALL
CHAIN
CHR$
CLOSE
COMMAND$
COMMON
CONCHAR%
ERR
ERRL
ERROR
ERRX
EQ
EXP
EXTERNAL
FEND
FLOAT
FOR
FRE
GE
GET
GO
LE
LEFT$
LEN
LET
LINE
LOCK
LOCKED
LOG
LPRINTER
LT
MATCH
MFRE
MID$
MOD
PUBLIC
PUT
RANDOMIZE
READ
READONLY
REAL
RECL
RECS
REM
REMARK
RENAME
RESTORE
RETURN
RIGHT$
STRING$
SUB
TAB
TAN
THEN
TO
UCASE$
UNLOCK
UNLOCKED
USING
VAL
VARPTR
WEND
WHILE

CBASIC Compiler Reserved Words **(continued)**

CONSOLE

CONSTAT%

COS

CREATE

DATA

DEF

DELETE

DETACH

DIM

ELSE

END

GOSUB

GOTO

GT

IF

INITIALIZE

INKEY

INP

INPUT

INT

INT%

INTEGER

NE

NEXT

NOT

ON

OPEN

OR

OUT

PEEK

POKE

POS

PRINT

RND

SADD

SGN

SHIFT

SIN

SIZE

SQR

STEP

STOP

STR$

STRING

WIDTH

XOR

%CHAIN

%DEBUG

%EJECT

%INCLUDE

%LIST

%NOLIST

%PAGE

End of Appendix A

Appendix B
Decimal-ASCII-Hex Table

Decimal-ASCII-Hex Table

Decimal	*ASCII*	*Hex*	*Decimal*	*ASCII*	*Hex*	*Decimal*	*ASCII*	*Hex*
0	NUL	00	44	,	2C	88	X	58
1	SOH	01	45	-	2D	89	Y	59
2	STX	02	46	.	2E	90	Z	5A
3	ETX	03	47	/	2F	91	[	5B
4	EOT	04	48	0	30	92	\	5C
5	ENQ	05	49	1	31	93	]	5D
6	ACK	06	50	2	32	94	ˆ	5E
7	BEL	07	51	3	33	95	—	5F
8	BS	08	52	4	34	96	‘	60
9	HT	09	53	5	35	97	a	61
10	LF	0A	54	6	36	98	b	62
11	VT	0B	55	7	37	99	c	63
12	FF	0C	56	8	38	100	d	64
13	CR	0D	57	9	39	101	e	65
14	SO	0E	58	:	3A	102	f	66
15	SI	0F	59	;	3B	103	g	67
16	DLE	10	60	<	3C	104	h	68
17	DC1	11	61	=	3D	105	i	69
18	DC2	12	62	>	3E	106	j	6A
19	DC3	13	63	?	3F	107	k	6B
20	DC4	14	64	@	40	108	l	6C
21	NAK	15	65	A	41	109	m	6D
22	SYN	16	66	B	42	110	n	6E
23	ETB	17	67	C	43	111	o	6F
24	CAN	18	68	D	44	112	p	70
25	CR	19	69	E	45	113	q	71
26	SUB	1A	70	F	46	114	r	72
27	ESC	1B	71	G	47	115	s	73
28	FS	1C	72	H	48	116	t	74
29	GS	1D	73	I	49	117	u	75

Decimal-ASCII-Hex Table **(continued)**

30	RS	1E	74	J	4A	118	v	76
31	US	1F	75	K	4B	119	w	77
32	SP	20	76	L	4C	120	x	78
33	!	21	77	M	4D	121	y	79
34	"	22	78	N	4E	122	z	7A
35	#	23	79	O	4F	123	{	7B
36	$	24	80	P	50	124	|	7C
37	%	25	81	Q	51	125	}	7D
38	&	26	82	R	52	126		7E
39	'	27	83	S	53	127	DEL	7F
40	(	28	84	T	54			
41	)	29	85	U	55			
42	*	2A	86	V	56			
43	+	2B	87	W	57			

End of Appendix B

Appendix C
CBASIC to CBASIC Compiler Conversion Aid

This conversion aid helps you convert your CBASIC programs to CBASIC Compiler. When you compile your source code in CBASIC Compiler, pay close attention to all error messages. This is the fastest way to determine any necessary changes. Most programs recompile with no conversion. If any problems arise, call the Digital Research Support Line (408-375-6262) for assistance.

In this appendix, CBASIC refers to the compiled/interpreted version of CBASIC, and CBASIC Compiler refers to the compiled version of CBASIC defined in this manual.

C.1 Subscripted Variables (Arrays)

CBASIC allows you to use a dimensioned variable name (an array) as a simple or unsubscripted variable. CBASIC treats these as separate and distinct variables. CBASIC Compiler does not allow a dimensioned variable without the array index.

```
CBASIC                     CBASIC Compiler

DIM A% (20)                DIM A% (20)

FOR I% = 1 to 20           FOR I% = 1 to 20
   A% (I%) = 0                A% (I%) = 0
NEXT I%                    NEXT I%

A% = 100                   A% = 100
                            (error message #36)
```

CBASIC Compiler issues error message #36 for the statement A% = 100 because the statement uses an identifier as a simple variable that was previously used as a subscripted variable.

```
CBASIC                   CBASIC Compiler

A% = 100                 A% = 100

DIM A% (20)              DIM A% (20)
                          (error message #93)

For I% = 1 to 20         For I% = 1 to 20
  A% (I%) =0               A% (I%) = 0
NEXT I%                   (error message #37)
                         NEXT I%
END
```

CBASIC Compiler issues error message #93 for the statement DIM A% (20) because a variable in a DIM statement is previously defined as other than a subscripted variable. CBASIC Compiler issues error message #37 for the statement A% = 100 because an identifier used as a subscripted variable was previously used as an unsubscripted variable.

To correct the error, change the unsubscripted variable to a different variable name of the same type. Choose a new variable that differs from all other variable names in your program.

C.2 FILE Statement

The FILE statement in CBASIC opens a file present on the referenced disk. The FILE statement can also create a file of the name you specify. However, CBASIC Compiler does not use the FILE statement. Use the OPEN, SIZE, and CREATE statements to open and create files.

```
CBASIC           CBASIC Compiler

FILE NAME$       IF SIZE (NAME$) <> 0 \
                   THEN OPEN NAME$ AS FILE.NO% \
                   ELSE CREATE NAME$ AS FILE.NO%
```

In the CBASIC Compiler example, if there is a file NAME$, the file is opened as usual. If there is no file NAME$, or the length of the file is zero (determined by the SIZE statement), the IF statement passes control to the CREATE statement, which creates the file NAME$. Both the OPEN and CREATE statements require a file reference number (FILE.NO%). However, the FILE statement does not need a file reference number.

When you convert a FILE statement, choose a file number that does not conflict with any other file reference numbers already in your program. Remember to modify the PRINT and READ statements that access the file to reflect the new file number.

C.3 SAVEMEM

The SAVEMEM statement, which executes routines written to the assembler in CBASIC, has no meaning in CBASIC Compiler. The *CBASIC Compiler (CB80) Language Programming Guide* and *CBASIC Compiler (CB86) Language Programming Guide* tell how to use assembler routines and explain how to link the routines to CBASIC Compiler programs.

C.4 CHAIN Statement

The CHAIN statement in CBASIC and CBASIC Compiler passes control from the program executing in memory to the program selected in the CHAIN statement. The CHAIN statement format is the same in both CBASIC Compiler and CBASIC.

CBASIC	*CBASIC Compiler*
CHAIN <expression>	CHAIN <expression>

The expression must evaluate to an unambiguous filename on the disk. If the filename in the expression does not include the filetype, CBASIC assumes an .INT filetype; however, CBASIC Compiler assumes an .OVL (overlay) filetype.

In CBASIC Compiler, the .OVL filetype is not the root of a chaining sequence. The root program has a .COM filetype. If your program chains back to the original root (.COM file) or a different root, the expression in the CHAIN statement must evaluate to a filename with .COM filetype. A CBASIC Compiler program can chain to a .COM file other than the one generated by the link editor.

C.5 String Lengths

CBASIC Compiler allows string lengths up to 32K. CBASIC Compiler uses two bytes to give this expanded string length; CBASIC uses one byte. To set strings to null in CBASIC Compiler, see the Programming Guide.

If your program uses the SADD function with PEEK and POKE to pass a string to an assembly language routine, you must change your program to accommodate the two-byte length indicator in CBASIC Compiler.

CBASIC

```
LEN% = PEEK (SADD(STRING$))
END
```

CBASIC Compiler

```
LEN% = (PEEK (SADD(STRING$)) AND 07FH \
      + PEEK (SADD(STRING$) + 1)) * 256
```

C.6 PEEK and POKE

The PEEK function in CBASIC and CBASIC Compiler returns the contents of the memory location specified in the PEEK function call. Memory locations in CBASIC Compiler might not contain the same information that CBASIC programs expect. You might have to change the memory location your program is examining, or remove the PEEK statement from your program.

The POKE statement behaves the same in CBASIC Compiler as it does in CBASIC. However, the memory locations in CBASIC Compiler differ from the memory locations in CBASIC. If your program contains a POKE statement to a location in a CBASIC program, it might insert the value at the wrong address when used in a CBASIC Compiler program. In particular, the statements,

```
POKE 0110H, 0
```

or

```
POKE 272, 0
```

used in CBASIC to adjust the console width, must be removed. Use the POKE statement carefully because the actual location of code is determined by the link editor.

C.7 FOR-NEXT Loops

When using FOR-NEXT loops in CBASIC, the NEXT statement can terminate more than one loop. CBASIC Compiler does not allow this construct. You must use a separate NEXT statement for each FOR statement that begins a loop.

```
CBASIC                         CBASIC Compiler

FOR I% = 1 TO 100              FOR I% = 1 TO 100
FOR J% = 1 TO 100              FOR J% = 1 TO 100
    .                              .
    . (statements)                 . (statements)
    .                              .
NEXT J%, I%                    NEXT J%
                               NEXT I%
```

Also, CBASIC executes all statements in the FOR-NEXT loop at least once. CBASIC Compiler executes the statements in a FOR-NEXT loop zero or more times, depending on the values of the loop indexes. This is potentially troublesome. Examine the logic of your programs, and make any necessary changes.

C.8 Console Width

To facilitate cursor addressing, CBASIC Compiler generates a carriage return only upon executing a PRINT statement not terminated by a comma or semicolon. This is analogous to setting the CBASIC console width to zero by a POKE to 272. CBASIC automatically generates a carriage return when the console width has been exceeded. Therefore, CBASIC programs that assume the cursor returns when the console width is exceeded might not execute correctly in CBASIC Compiler.

C.9 FRE

In CBASIC Compiler, FRE returns a binary value that represents the number of bytes of available memory. In CBASIC, the binary value represents a real value. Programs that use FRE must interpret negative values correctly, because CBASIC Compiler arithmetic routines interpret binary values in excess of 32,767 as negative numbers. In general, negative values indicate ample available memory.

The following statement can determine whether adequate memory is available.

```
IF (FRE > 0) AND (FRE < MIN.MEMORY%) THEN \
   CALL LOW.MEMORY.WARNING
```

C.10 READ and INPUT Statements for Integers

READ and INPUT statements handle integers differently in the two languages. CBASIC accepts all numeric values as real numbers, and then converts to integers if required. CBASIC Compiler accepts integers directly.

CBASIC	*CBASIC Compiler*
`DATA 10.7, 1E2`	`DATA 10.7, 1E2`
`READ A%,B%`	`READ A%,B%`
The values of A% and B% after the READ are:	The values of A% and B% after the READ are:
`A% = 11   B% = 100`	`A% = 10   B% = 1`

With CBASIC Compiler, conversion stops at the first character not a part of a valid integer.

C.11 Function and Variable Names

CBASIC Compiler requires that function names, variables, and statement labels be unique. In CBASIC, all functions must start with the letters FN, and labels must be numeric constants. Thus, no problems should occur when you convert programs from CBASIC to CBASIC Compiler. Remember that variables and arrays might conflict as described in Section C.1.

C.12 Labels

CBASIC Compiler places all program labels, including unreferenced labels, in a symbol table. CBASIC does not put unreferenced labels in the symbol table.

A label in a multiple-line function is local to the function. This is not the same in CBASIC.

```
CBASIC                     CBASIC Compiler

DEF FN.A                   DEF FN.A
100 PRINT "HELLO"          100 PRINT "HELLO"
FEND                       FEND
GOTO 100                   GOTO 100
                           (error message #82)
```

CBASIC Compiler issues error message #82 because the label in a GOTO statement is undefined. The label used in a function must be defined in that function.

C.13 Warning Messages

CBASIC Compiler produces no warning messages during the execution of a program. All errors are fatal and execution terminates unless you use an ON ERROR GOTO statement to trap the error.

C.14 New Reserved Words

CBASIC Compiler incorporates new reserved words with some of the newly implemented features. If your CBASIC programs use these words as variables, rename them to a different variable name. The following is a list of reserved words unique to CBASIC Compiler. Appendix A contains a complete list of all CBASIC Compiler reserved words.

ATTACH
%DEBUG
DETACH
ERR
ERRL
ERROR
ERRX
EXTERNAL
GET
INITIALIZE
INKEY
INTEGER
LOCK
LOCKED
MOD
PUBLIC
PUT
READONLY
REAL
SHIFT
STRING
STRING$
UNLOCK
UNLOCKED

End of Appendix C

Appendix D
Glossary

address: Location in memory.

ambiguous file specification: File specification that contains either of the Digital Research wildcard characters, ? or *, in the filename, filetype, or both. When you replace characters in a file specification with these wildcard characters, you create an ambiguous filespec and can reference more than one file in a single command line.

applications program: Program that needs an operating system to provide an environment in which to execute. Typical applications programs are business accounting packages, word processing, and mailing list programs.

argument: Variable or expression value that is passed to a procedure or function and substituted for the dummy argument in the function. Same as actual argument or calling argument. Used interchangeably with parameter.

array: Data type that is a collection of individual data items of the same data type. Term that describes a form of storing and accessing data in memory, visualized as matrices. The number of extents of an array is the number of dimensions of the array. A one-dimensional array is essentially a list.

ASCII: Acronym for American Standard Code for Information Interchange. ASCII is a standard code for representation of the numbers, letters, and symbols that appear on most keyboards.

assembler: Language translator that translates assembly language statements into machine code.

assignment statement: Statement that assigns the value of an expression on the right side of an equal sign to the variable name on the left side of the equal sign.

back-up: Copy of a file or disk made for safekeeping, or the creation of the file or disk.

binary: Base two numbering system containing the two symbols zero and one.

bit: Common contraction for binary digit. Switch in memory that can be set to on (1) or off (0). Eight bits grouped together comprise a byte.

buffer: Area of memory that temporarily stores data during the transfer of information.

byte: Unit of memory or disk storage containing eight bits.

call: Transfer of control to a computer program subroutine.

chain: Transfer of control from the currently executing program to another named program without returning to the system prompt or invoking the run-time monitor.

code: Sequence of statements of a given language that make up a program.

command: Instruction or request for the operating system or a system program to perform a particular action. Generally, a Digital Research command line consists of a command keyword, a command tail usually specifying a file to be processed, and a carriage return.

common: Variables used by a main program and all programs executed through a chain statement.

compiler: Language translator that translates the text of a high-level language into machine code.

compiler directive: Reserved words that modify the action of the compiler.

compiler error: Error detected by the compiler during compilation; usually caused by improper formation of language statement.

compiler toggle: Switch that modifies the output of the compiler.

concatenate: Join one string to another or one file to another.

concatenation operator: Symbol peculiar to a given language that instructs the compiler to combine two unique data items into one.

console: Primary input/output device. The console consists of a listing device such as a screen and a keyboard through which the user communicates with the operating system or the applications program.

constant: String or numeric value that does not change throughout program execution.

control character: Nonprinting character combination that sends a simple command to the operating system or applications program. To enter a control character, press the control (CTRL) key on your terminal and strike the character key specified.

control statement: Language statement that transfers control or directs the order of execution of instructions by the processor.

cursor: One-character symbol that can appear anywhere on the video screen. The cursor indicates the position where the next keystroke at the console will have an effect.

data: Information; numbers, figures, names, and so forth.

data base: Large collection of information, usually covering various aspects of related subject matter.

data file: Nonexecutable file of similar information that generally requires a command file to process it.

data structure: Mechanism, including both storage layout and access rules, by which information can be stored and retrieved in a computer system. Data structures can reside in memory or on secondary storage. System tables such as symbol tables, matrices of numerical data, and data files are examples of data structures.

data type: Class or use of the data; for example, integer, real, or string.

debug: Remove errors from a program.

default: Values, parameters, or options a given command assumes if not otherwise specified.

delimiter: Special characters or punctuation that separate different items in a command line or language statement.

dimension: Refers to the number of extents of an array. A one-dimensional array is essentially a list of the elements of the array. A two-dimensional array can be visualized as a matrix of rows and columns of storage space for the elements of the array. A three-dimensional array can be thought of as a geometric solid having volume, and so forth.

directory: Portion of a disk that contains entries for each file on the disk. In response to the DIR command, CP/M and MP/M systems display the file specifications stored in the directory.

disk, diskette: Magnetic media used to store information. Programs and data are recorded on the disk in the same way that music is recorded on a cassette tape. The term diskette refers to smaller capacity removable floppy diskettes. The term disk can refer to a diskette, a removable cartridge disk, or a fixed hard disk.

disk drive: Peripheral device that reads and writes on hard or floppy disks. CP/M and MP/M systems assign a letter to each drive under their control.

drive specification: Alpha character A through P followed by a colon that indicates the CP/M or MP/M drive reference for the default or specified drive.

editor: Utility program that creates and modifies text files. An editor can be used to create documents or code for computer programs.

element: Individual data item in an array.

executable: Ready to run on the processor. Executable code is a series of instructions that can be carried out on the processor. For example, the computer cannot execute names and addresses, but it can execute a program that prints names and addresses on mailing labels.

execute a program: Start a program running. When the program is executing, a process is executing a sequence of instructions.

expression: Algebraic combination of variables, constants, operators, and function references that evaluates to an integer, real, or string value.

FCB: File Control Block. Structure used for accessing files on disk. Contains the drive, filename, filetype, and other information describing a file to be accessed or created on the disk.

field: Portion of a record; length and type are defined by the programmer. One or more fields comprise a record.

file: Collection of related records containing characters, instructions or data; usually stored on a disk under a unique file specification.

file number: Unique identification number you assign to a file with the CREATE or OPEN statement. File numbers can be any numeric expression. If the expression evaluates to a real number, it converts to an integer. File numbers range from 1 to the current implementation limit for the number of files accessible at one time.

filename: Name assigned to a file. The filename can include 1 to 8 alpha, numeric, and/or some special characters. The filename should tell something about the file.

filetype: Extension to a filename. A filetype is optional, and can contain from 0 to 3 alpha, numeric, and/or some special characters. The filetype must be separated from the filename by a period. Certain programs require that files to be processed have specific filetypes.

file access: Refers to methods of entering a file to retrieve the information stored in the file.

file specification: Unique file identifier. A Digital Research file specification includes an optional drive specification followed by a colon, a primary filename of 1 to 8 characters, and an optional period and filetype of 0 to 3 characters. Some Digital Research operating systems allow an optional semicolon and password of 1 to 8 characters following the filename or filetype. All alpha and numeric characters and some special characters are allowed in Digital Research file specifications.

fixed: Type of file organization used when data is to be accessed randomly—not in sequential order. Refers generally to the nonvarying lengths of the records composing the file.

floating point: Value expressed in decimal notation that can include exponential notation; a real number.

floppy disk: Flexible magnetic disk used to store information. Floppy disks are manufactured in 5 1/4- and 8-inch diameters.

flowchart: Graphic diagram that uses special symbols to indicate the input, output, and flow of control of part or all of a program.

flow of control: Order of the execution of statements in a program.

formal parameter: Holds a place for an actual parameter that you specify in a user-defined function reference.

format: System utility that writes a known pattern of information on a disk so a given hardware configuration can properly support reading and writing on that disk.

formatted printing: Output specifically designed in a certain pattern and achieved through particular coded language statements.

fragmentation: Division of storage area in a way that causes areas to be wasted.

function: Subroutine to which you can pass values and which returns a value. Useful when the same code is required repeatedly, because the program can call the function at any time.

global: Relevant throughout an entire program.

hex file: ASCII-printable representation of a code or data file in hexadecimal notation.

hexadecimal notation: Notation for the base 16 number system using the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F to represent the sixteen digits. Machine code is often converted to hexadecimal notation because it can be more easily understood.

high bound: Upper limit of one dimension of an array.

high-level language: Set of special words and punctuation that allows a programmer to code software without being concerned with internal memory management.

identifier: String of characters used to name elements of a program, such as variable names, reserved words, and user-defined function names. Commonly used synonymously with variable name.

include: Call an external file into the code sequence of a program at the point where the include statement is executed.

initialize: Set a disk system or one or more variables to initial values.

I/O: Abbreviation for input/output.

input: Data entered to an executing program, usually from an operator typing at the terminal or by the program reading data from a disk.

instruction: Set of characters that defines an operation.

integer: Positive or negative nonexponential whole number that does not contain a decimal point.

interface: Object that allows two independent systems to communicate with each other, as an interface between the hardware and software in a microcomputer.

intermediate code: Code generated by the syntactical and semantic analyzer portions of a compiler.

interpreter: Computer program that translates and executes each source language statement before translating and executing the next one.

ISAM: Abbreviation for Indexed Sequential Access Method.

key: Particular field of a record on which the processing is performed.

keyword: Reserved word with special meaning for statements or commands.

kilobyte: 1024 bytes denoted as 1K. 32 kilobytes equal 32K. 1024 kilobtyes equal one megabyte, or over one million bytes.

label: Constant, either numeric or literal, that references a statement or function. Labels in the main executable block of a program must be unique. All labels in a function must also be unique. However, a label in a function can be the same as a label in the main executable block of a program or in another function.

linker: System software module that connects previously assembled or compiled programs or program modules into a unit that can be loaded into memory and executed.

linked list: Data structure in which each element contains a pointer to its predecessor or successor (singly-linked list) or both (double-linked list).

list device: Device, such as a printer, onto which data can be listed or printed.

listing: Output file created by the compiler that lists the statements in the source program, the line numbers it has assigned to them, and possibly other optional information.

literal data: Verbatim translation of characters in the code, such as in screen prompts, report titles, and column headings.

load: To move code from storage into memory for execution.

local variable: Relevant only in a specific portion of a program, such as in a function.

logged-in: Made known to the operating system, in reference to drives. A drive is logged-in when it is selected by the user or an executing process.

logical: Representation of something, such as a console, memory, or disk drive, that might or might not be the same in its actual physical form. For example, a hard disk can occupy one physical drive, and yet you can divide the available storage on it to appear to the user as if there were several different drives. These apparent drives are the logical drives.

logical device: Reference to an I/O device by the name or number assigned to the physical device.

logical expression: Expression that evaluates to either true or false.

logical operator: NOT, AND, OR, and XOR.

lower bound: Lower limit of one dimension of an array.

machine code: Output of an assembler or compiler to be executed directly on the target processor.

machine language: Instructions directly executable by the processor.

memory: Storage area in and/or attached to a computer system.

microprocessor: Silicon chip that is the CPU of the microcomputer system.

mixed mode: Combination of integer, real or numeric, string values in an expression. Mixed string and numeric operations are generally not allowed in high-level languages.

mnemonic operator: Alphabetical symbol for algebraic operator: LT, LE, GT, GE, NE, and EQ.

module: Section of software having well-defined input and output that can be tested independently of other software.

multiple-line function: Function composed of a function definition statement and one or more additional statements.

numeric constant: Real or integer quantity that does not vary in the program.

numeric variable: Real or integer identifier to which varying numeric quantities can be assigned during program execution.

null string: String that contains no character; essentially an empty string.

object code: Output of an assembler or compiler that executes on the target processor.

open: System service that informs the operating system of the manner in which a given resource, usually a disk file, is intended to be used.

operating system: Collection of programs that supervises the execution of other programs and the management of computer resources. An operating system provides an orderly input/output environment between the computer and its peripheral devices, enabling user programs to execute safely.

operation: Execution of a piece of code.

operator: Symbol that represents an arithmetic operation or comparison such as +, -, =, or <.

option: One of a set of parameters that can be part of a command or language statement. Options are used to modify the output of an executing process.

output: Data that the processor sends to the console, printer, disk, or other storage media.

parameter: Value supplied to a command or language statement that provides additional information for the command or statement. Used interchangeably with argument. An actual parameter is a value that is substituted for a dummy or formal argument in a given procedure or function.

peripheral device: Devices external to the CPU. For example, terminals, printers, and disk drives are common peripheral devices that are not part of the processor, but are used in conjunction with it.

pointer: Data item whose value is the address of a location in memory.

primitive: Most basic or fundamental unit of data such as a single digit or letter.

process: Program that is actually executing, as opposed to being in a static state of storage on disk.

program: Series of specially coded instructions that performs specific tasks when executed on a computer.

prompt: Characters displayed on the input terminal to help the user decide what the next appropriate action is. A system prompt is a special prompt displayed by the operating system, indicating to the user that it is ready to accept input.

random access: Method of entering a file at any record number, not necessarily the first record in the file.

random access file: File structure in which data can be accessed in a random manner, irrespective of its position in the file.

random number: Number selected at random from a set of numbers.

real number: Numeric value specified with a decimal point; same as floating-point notation.

record: One or more fields usually containing associated information in numerical or textual form. A file is composed of one or more records and generally stored on disk.

record number: Position of a specific record in a fixed-length file, relative to record number 1. A key by which a specific record in a fixed file is accessed randomly.

recursive: Code that calls itself.

relational operator: Comparison operator. A relational operator states a relationship between two expressions. The following symbols are CBASIC relational operators: LT, LE, NE, EQ, GT, GE, EQ.

reserved word: Keyword that has a special meaning to a given language or operating system.

return value: Value returned by a function.

row-major order: Order of assignment of values to array elements in which the first item of the subscript list indicates the number of rows in the array.

run a program: Start a program executing. When a program is running, the microprocessor chip is executing a series of instructions.

run-time error: Error occurring during program execution.

run-time monitor: Program that directly executes the coded instructions generated by a compiler/interpreter.

sequential access: Type of file structure in which data can only be accessed serially, one record at a time. Data can be added only to the end of the file and cannot be deleted. An example of a sequential access media is magnetic tape.

source program: Text file that is an input file for a processing program, such as an editor, text formatter, assembler, or compiler.

statement: Defined way of coding an instruction or data definition using specific keywords in a specific format.

storage: Place for keeping data temporarily in memory or permanently on disk.

stream organization: Type of file organization used when data is to be accessed sequentially. Can contain variable length records.

string constant: Literal data, as in a screen prompt, column heading, or title of a report.

string variable: Identifier of type string to which varying strings can be assigned during program execution.

subroutine: Section of code that performs a specific task, is logically separate from the rest of the program, and can be prewritten. A subroutine is invoked by another statement and returns to the place of invocation after executing. Subroutines are useful when the same sequence of code is used more than once in a program.

subscript: Integer expression that specifies the position of an element in an array.

subscript list: Numeric value appended to a variable name that indicates the number of elements in each dimension in the array of that name. Each dimension must have a value in the subscript list indicating the number of elements for which to allocate storage space.

syntax: Rules for structuring statements for an operating system or programming language.

toggle: Switch enabled by a special code in the command line that modifies the output of the executing program.

trace: Option used for run-time debugging. The trace option generally lists each line of code as it executes to enable the programmer to note where a problem occurs.

upward-compatible: Term meaning that a program created for the previously released operating system or compiler runs under a later release of the same software program.

user-defined function: Set of statements created and given a function name by the user. The function performs a specific task and is called into action by referencing the function by name.

utility: Tool; a program or module that facilitates certain operations, such as copying, erasing, and editing files, or controlling the cursor positioning on the video screen from within a program. Utilities are created for the convenience of programmers and applications operators.

value: Quantity expressed by an integer or real number.

variable: Name to which the program can assign a numerical value or string.

variable length: Usually refers to records, where each record in a file is not necessarily the same length as another.

variable name: Same as variable.

wildcard characters: Special characters, ? and *, that can be included in a Digital Research filename and/or filetype to identify more than one file in a single file specification.

End of Appendix D

Index

DIGITAL RESEARCH®

CBASIC Compiler™

Handbuch in englischer Sprache

Der Hochleistungs-BASIC-Compiler für Softwareprofis zur Erstellung kommerzieller Anwendungen.
Der CBASIC-Compiler ist ein erweitertes BASIC mit wichtigen Vorteilen für Softwareprofis. Er ist ein Compiler, der Maschinencode erzeugt und die Programmierung und den Test separater Module erlaubt, die später ein komplettes Programm ergeben sollen.

Schnelle Ausführung
Der CBASIC-Compiler kombiniert die Geschwindigkeit von Maschinencode mit der leichten Verständlichkeit der Sprache BASIC. Ein mit dem CBASIC-Compiler kompiliertes Programm wird acht- bis zehnmal schneller ausgeführt als das gleiche interpretierte Programm.

Dezimal-Arithmetik
Die 14stellige Dezimal-Arithmetik gewährleistet höchste Genauigkeit bei Berechnungen und stellt sicher, daß alle Geldbeträge auf den Pfennig genau stimmen. Rundungsfehler, wie sie bei binärer Arithmetik möglich sind, können nicht auftreten.
Der CBASIC-Compiler unterstützt auch echte Integer-Arithmetik, so daß zur Erhöhung der Geschwindigkeit auch Integer-Variablen verwendet werden können.

Mehrzeilige Funktionen
Durch die Möglichkeit, mehrzeilige Funktionen zu erstellen, verfügt der CBASIC-Compiler über Fähigkeiten, die sich sonst nur in strukturierten Programmiersprachen wie PL/I oder Pascal finden. Innerhalb einer mehrzeiligen Funktion können lokale Variablen verwendet werden.

Für professionellen Einsatz
Der CBASIC-Compiler wurde für die hohen Ansprüche professioneller Softwareentwickler und erfahrener Anwender entwickelt. Der CBASIC-Compiler wird mit ausführlicher Dokumentation in englischer Sprache geliefert.

Hardwarevoraussetzungen
Der CBASIC-Compiler läuft auf dem Commodore 128 mit einem Diskettenlaufwerk. Der CBASIC-Compiler wird im 1541-Format ausgeliefert.

Die Vorteile des CBASIC-Compilers auf einen Blick:

- hohe Geschwindigkeit der erzeugten Programme
- Grafikerweiterungen
- Dezimal-Arithmetik mit hoher Genauigkeit
- umfangreiche Stringverarbeitung
- Stringlänge bis 32 KByte
- mehrzeilige Funktionen
- keine Zeilennummern erforderlich
- Overlays durch CHAIN-Befehl

Lieferumfang:

- 5¼-Zoll-Diskette
- Bedienerhandbuch in englischer Sprache
- Beschreibung der Commodore-spezifischen Version

Markt&Technik
128er-Software
Hans-Pinsel-Straße 2
8013 Haar