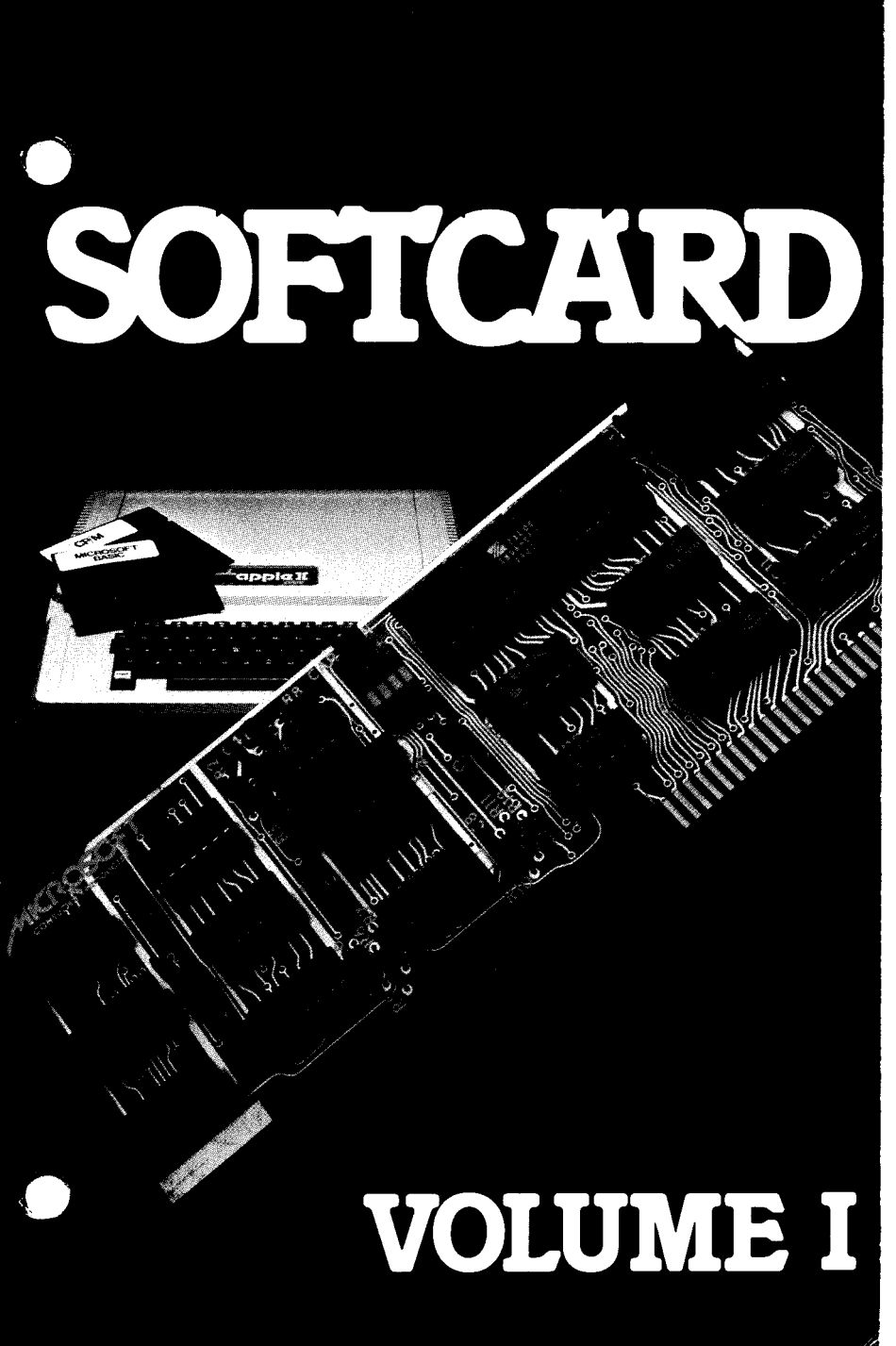
SOFTCARD



VOLUME I

SoftCard™

A Peripheral for the Apple II®
With CP/M® and Microsoft BASIC on diskette.

Produced by Microsoft

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

Copyright and Trademark Notices

If you have any questions on these copyrights, please contact:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

TABLE OF CONTENTS

INTRODUCTION

PART I: Installation and Operation

PART II: Software and Hardware Details

PART III: CP/M Reference Manual

PART V: Software Utilities Manual

Introduction

The SoftCard Explained

The Circuit Card

The Microsoft SoftCard is a plug-in card for the Apple II microcom-
modification, but be sure to read the Installation and Operation Manual to ensure that you do it correctly.

Once you have installed the SoftCard, you will be able to operate your Apple in either 6502 or Z-80 mode, using software commands to switch between the two. Whenever you are in 6502 mode, the SoftCard in no way affects operation of your Apple.

When in Z-80 mode, you can run both the CP/M operating system from Digital Research and Microsoft's BASIC interpreter, Version 5.0, which are included in the SoftCard package.

The SoftCard is easy to install and requires no hardware or software puter that greatly enhances the software capability of the Apple. The SoftCard actually contains a Z-80A microprocessor, allowing the Apple to run software that was written for Z-80 based microcomputers.

CP/M Operating System

Next to the circuit card itself, CP/M is the most important key to allowing a wide variety of Z-80 software to run on the Apple. Version 2.2 of CP/M is included in the SoftCard package.

CP/M (which stands for Control Program/Microprocessors) is an operating system designed for use with 8080 and Z-80 microprocessors. It is composed of many small programs whose collective function is to write information to, and retrieve information from, microcomputer floppy disks. CP/M has been adapted to run on almost all computers using the 8080 or Z-80 families of microprocessors and because of its widespread use, a very large group of high-level languages and application software has been written to operate in the CP/M environment.

With the advent of the SoftCard, Apple owners are now able to take advantage of the CP/M Operating System. Microsoft has implemented CP/M on the Apple II, making all modifications needed to make CP/M run on the Apple.

Standard CP/M programs will be compatible with Apple CP/M. There is just one difficulty in loading them on the Apple: Apple disks have a physically different format than CP/M disks. Before a CP/M program written for another type of computer can be run on the Apple, it must be downloaded from a standard CP/M system to the Apple. This process is described *in detail* in the Software Utilities Manual.

In addition to supporting a wider variety of software, CP/M offers several convenient features not found in Apple DOS. These include easy interface to machine language programs; faster disk I/O; simple file transfer; and wild card file-naming conventions that allow you to refer to multiple files with one name.

Microsoft BASIC

Microsoft's ANSI-standard BASIC interpreter, in its fifth major release, is also included as part of the SoftCard package. Microsoft BASIC has many features not found in Applesoft. Among these are PRINT USING, CALL, WHILE/WEND, CHAIN and COMMON and built-in Disk I/O statements. In addition, most of the graphics features of Applesoft have been incorporated into Microsoft BASIC to take advantage of the Apple's special capabilities. A complete list of the differences between Microsoft BASIC and Applesoft can be found in Part 4, the Microsoft BASIC Reference Manual.

The Diskettes

Two diskettes, each containing CP/M and Microsoft BASIC plus several utility programs, are provided. One of the disks is in 13-Sector format and should be used if you don't have a Language Card or DOS 3.3. The other disk is in 16-Sector format and should be used with systems that have the Apple Language Card and/or DOS 3.3. The 16-Sector disk also contains an enhanced version of Microsoft BASIC with high-resolution graphics capabilities.

Designers and Manufacturer

The Softcard Circuit Board

Designer: The SoftCard circuit board was designed by Don Burtis of Burtronix, Villa Park, California. Microsoft Consumer Products is grateful to Burtronix for its contribution to making the SoftCard a reality.

Manufacturer: The SoftCard circuit board is manufactured for Microsoft Consumer Products by Vista Computer Co. of Santa Ana, California.

SoftCard Software

The CP/M operating system, Version 2.0, is licensed by Microsoft from Digital Research, Inc., of Pacific Grove, California. The BASIC interpreter included in this package is Microsoft's ANSI-standard BASIC-80, Version 5.0, with additional enhancements to take advantage of the Apple's special capabilities. Neil Konzen, of Microsoft Consumer Products, was instrumental in implementing all of the SoftCard software on the Apple II.

System Requirements

The SoftCard will operate on an Apple II or Apple II Plus microcomputer with a minimum of 48K RAM and one disk drive.

The SoftCard supports the Apple Language Card system and can utilize 12K of the 16K RAM on the Language Card when in Z-80 mode.

CP/M occupies 7K of RAM, only 5K of which is needed during the execution of user programs. CP/M and MBASIC together occupy just over 29K RAM. CP/M and GBASIC (BASIC with high-resolution graphics, found only on the 16-Sector disk) occupy just over 37K RAM.

When you are in 6502 mode, the SoftCard in no way affects operation of the Apple II.

When in Z-80 mode, all standard Apple I/O peripheral cards and some independent peripherals are supported.

SoftCard Terminology

There are several terms we use throughout this documentation that you may not understand at first glance. These terms, their definitions, and the reasons we have adopted them are listed below.

44K System — Refers to an Apple II or Apple II Plus that has 48K RAM installed. We call it a 44K System, because when you are using the SoftCard (in Z-80 mode), you can address 44K of the 48K total. The 4K you lose is used to handle the Apple screen and CP/M sector read and write routines.

56K System — Refers to an Apple II or Apple II Plus *with* Language Card (an Apple with 64K RAM installed). As with a 48K system, 4K of the 64K is dedicated to the Apple screen and CP/M sector read and write routines. And since only 12K of the 16K RAM on the Language Card is addressable, you have, in effect, a 56K system.

13-Sector Disk — Refers to one of the disks in the SoftCard package. This disk should be used if you have Apple DOS 3.2 or earlier and no Language Card.

16-Sector Disk — Refers to the other disk in the SoftCard package. This disk should be used if you have an Apple Language Card or DOS 3.3. In addition to all of the software on the 13-Sector disk, the 16-Sector disk includes a second version of BASIC, called GBASIC, that includes high-resolution graphics features.

A:-F: — The names A:, B:, C:, D:, E: and F: refer to disk drives. This is the standard CP/M drive naming convention and since we are using CP/M, it is used throughout this manual. For the relationship of drive names to drives, see the Installation and Operations Manual.

External Terminal Refers to two types of devices. An external terminal can be a 24 × 80 character video card (such as the Videx Videoterm), or it can actually be a second terminal (such as a Hazeltine or SOROC) that you are using with your system.

RETURN vs. ⟨cr⟩ vs. carriage return
All of these mean to press the RETURN key on the Apple keyboard.

Digital Research License Information

IMPORTANT: Our license with Digital Research for the CP/M Operating System requires that each purchaser of the SoftCard with CP/M register with Microsoft Consumer Products so that records can be maintained of all CP/M owners. This requirement is made by Digital Research, not by Microsoft, and a post card is enclosed for reply. The serial number requested on the card is the number stamped on the disk labels. The registration card also specifies agreement to Digital Research's software license agreement. Before signing the card and returning it to Microsoft, read the software license agreement below carefully.

DIGITAL RESEARCH

Box 579 Pacific Grove, California, 93950

SOFTWARE LICENSE AGREEMENT

IMPORTANT: All Digital Research programs are sold only on the condition that the purchaser agrees to the following license. READ THIS LICENSE CAREFULLY. If you do not agree to the terms contained in this license, return the packaged diskette UNOPENED to your distributor and your purchase price will be refunded. If you agree to the terms contained in this license, fill out the REGISTRATION information and RETURN by mail to Microsoft Consumer Products.

DIGITAL RESEARCH agrees to grant and the Customer agrees to accept on the following terms and conditions nontransferable and nonexclusive licenses to use the software program(s) (Licensed Programs) herein delivered with this agreement.

Term:
This agreement is effective from the date of receipt of the above-referenced program(s) and shall remain in force until terminated by the Customer upon one month's prior written notice, or by Digital Research as provided below.

Any license under this Agreement may be discontinued by the Customer at any time upon one month's prior written notice. Digital Research may discontinue any license or terminate this Agreement if the Customer fails to comply with any of the terms and conditions of this Agreement.

License:
Each program license granted under this Agreement authorizes the Customer to use the Licensed Program in any machine readable form on any single computer system (referred to as System). A separate license is required for each System on which the Licensed Program will be used.

This Agreement and any of the licenses, programs or materials to which it applies may not be assigned, sublicensed or otherwise transferred by the Customer without prior written consent from Digital Research. No right to print or copy, in whole or in part, the Licensed Programs is granted except as hereinafter expressly provided.

Permission To Copy or Modify Licensed Programs:
The customer shall not copy, in whole or in part, any Licensed Programs which are provided by Digital Research in printed form under this Agreement. Additional copies of printed materials may be acquired from Digital Research.

Any Licensed Programs which are provided by Digital Research in machine readable form may be copied, in whole or in part, in printed or machine readable form in sufficient number for use by the Customer with the designated System, to understand the contents of such machine readable material, to modify the Licensed Program as provided below, for back-up purposes, or for archive purposes, provided, however, that no more than five (5) printed copies will be in existence under any license at any one time without prior written consent from Digital Research. The Customer agrees to maintain appropriate records of the number and location of all such copies of Licensed Programs. The original, and any copies of the Licensed Programs, in whole or in part, which are made by the Customer shall be the property of Digital Research. This does not imply, of course, that Digital Research owns the media on which the Licensed Programs are recorded. The Customer may modify any machine readable form of the Licensed Programs for his own use and merge it into other program material to form an updated work, provided that, upon discontinuance of the license for such Licensed Program, the Licensed Program supplied by Digital Research will be completely removed from the updated work. Any portion of the Licensed Program included in an updated work shall be used only if on the designated System and shall remain subject to all other terms of this Agreement.

The Customer agrees to reproduce and include the copyright notice of Digital Research on all copies, in whole or in part, in any form, including partial copies of modifications, of Licensed Programs made hereunder.

Protection and Security:
The customer agrees not to provide or otherwise make available any Licensed Program including but not limited to program listings, object code and source code, in any form, to any person other than Customer or Digital Research employees, without prior written consent from Digital Research, except with the Customer's permission for purposes specifically related to the Customer's use of the Licensed Program.

Discontinuance:
Within one month after the date of discontinuance of any license under this Agreement, the Customer will furnish Digital Research a certificate certifying that through his best effort, and to the best of his knowledge, the original and all copies, in whole or in part, in any form, including partial copies in modifications, of the Licensed Program received from Digital Research or made in connection with such license have been destroyed, except that, upon prior written authorization from Digital Research, the Customer may retain a copy for archive purposes.

Disclaimer of Warranty:
Digital Research makes no warranties with respect to the Licensed Programs. The sole obligation of Digital Research shall be to make available all published modifications or updates made by Digital Research to Licensed Programs which are published within one (1) year from date of purchase, provided Customer has returned the Registration Card delivered with the Licensed Program.

Limitation of Liability:
THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL DIGITAL RESEARCH BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF DIGITAL RESEARCH HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

General
If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted.

Microsoft Consumer Products Registration Information

Please fill out the SoftCard registration card that is enclosed and return it to us so that we may provide you with information about updates and about new products. The serial number requested on the card is the number printed on the disk labels.

SoftCard Warranty

Microsoft Consumer Products ("MCP") warrants to the original user of this product that it shall be free of defects resulting from faulty manufacture of the product or its components for a period of ninety (90) days from the date of sale. MCP MAKES NO WARRANTIES REGARDING EITHER THE SATISFACTORY PERFORMANCE (i.e. MERCHANTABILITY) OF THE SOFTWARE ENCODED ON THIS PRODUCT OR THE FITNESS OF THE SOFTWARE FOR ANY PARTICULAR PURPOSE. Defects covered by this Warranty shall be corrected either by repair or, at MCP's election, by replacement. In the event of replacement, the replacement unit will be warranted for the remainder of the original ninety (90) day period or 30 days, whichever is longer.

If this product should require service, return it to Microsoft Consumer Products, 400 108th Ave. NE, Suite 200, Bellevue, Washington 98004, postage prepaid, along with an explanation of the suspected defect. MCP will promptly handle all warranty claims.

THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THOSE OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, WHICH EXTEND BEYOND THE DESCRIPTION AND DURATION SET FORTH HEREIN.

MCP's SOLE OBLIGATION UNDER THIS WARRANTY IS LIMITED TO THE REPAIR OR REPLACEMENT OF A DEFECTIVE PRODUCT AND MCP SHALL NOT, IN ANY EVENT, BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY KIND RESULTING FROM USE OR POSSESSION OF THIS PRODUCT.

Some states do not allow 1) limitations on how long an implied warranty lasts, or 2) the exclusion or limitation of incidental or consequential damages, so the above limitations or exclusions may not apply to you.

This Warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Service Information

If your SoftCard requires repair, please return it to the dealer from whom it was purchased. If it is not possible to return the SoftCard to your dealer, you may send it directly to Microsoft Consumer Products.

If the repair is required during the warranty period, please enclose proof of purchase. During warranty, we will replace or repair your SoftCard without charge. See page I-10 for more details regarding warranty coverage.

If the SoftCard requires service after the warranty period expires, it will be repaired for a flat fee of $39.50. This service charge does not cover damage due to negligence, misuse or inadequate packaging on return to MCP.

To return your SoftCard for service, please mail it post-paid to Microsoft Consumer Products. Package the card securely as we cannot be responsible for damage due to shipping. BE SURE to enclose proof of purchase for warranty work or a check or money order in the amount of $39.50 for non-warranty repairs.

Mail post-paid to:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

SoftCard Installation and Operations

PART I: INSTALLATION AND OPERATION

Chapter 1
How To Install the SoftCard

Chapter 2
Getting Started with Apple CP/M

Chapter 3
An Introduction to Apple CP/M

Chapter 1
How To Install the SoftCard

- **Apple Peripheral Cards: What Goes Where**
- **Interface Cards Compatible with CP/M**
- **Placement of Apple Disk Drives**
- **Printer Interface Installation**
- **General Purpose I/O Installation**
- **Using an External Terminal Interface**
- **Installation of the SoftCard**

Installation of the SoftCard is easy, but there are some things you should know before you install it. Improper installation can damage both the SoftCard and the rest of your Apple system. So . . .

READ THESE INSTRUCTIONS CAREFULLY BEFORE INSTALLING THE SOFTCARD!!

Apple Peripheral Cards: What Goes Where

Before you install the SoftCard, you must make sure that your other peripheral cards are installed in the correct peripheral slots in your Apple to insure proper operation with Apple CP/M.

This is necessary because unlike Applesoft and Integer BASIC (but similar to Apple PASCAL), Apple CP/M requires that peripheral I/O cards be plugged into specific slots depending on their intended use. For instance, if you have a printer interface, it should be installed in slot one. This allows you to refer to the printer without specifying a slot number, as is necessary with Applesoft and Integer BASIC. Use the information below as a guide for installing any other peripheral interface cards you might own.

NOTE to Apple Language Card users:
The peripheral card slot assignments for Apple CP/M are exactly the same as for Apple Pascal. Therefore, if you have your system set up for use with Apple Pascal, no rearrangement is necessary.

Interface Cards Directly Compatible With CP/M:

Below is a list of the I/O peripheral card types that are known to be directly compatible with Apple CP/M. The cards listed below, when installed in the appropriate Apple peripheral slot, will work without any software modifications.

TYPE	CARD NAME
1	Apple Disk II Controller
*2	Apple Communications Interface
	California Computer Systems 7710A Serial Interface
3	Apple High Speed Serial Interface
	Apple Silentype Printer Interface
	Videx Videoterm 24×80 Video Terminal Card
	M&R Enterprises Sup-R-Term 24×80 Video Terminal Card
4	Apple Parallel Printer Card

*The CCS 7710A serial interface card is the preferred card of type 2 as it supports hardware handshaking and variable baud rates from 110-19200 baud. The Apple Communications Interface card requires hardware modification for use with data rates other than 110 or 300 baud.

There are some interface cards not listed above that may work with Apple CP/M. As a general rule, any card that is directly compatible with Apple Pascal without requiring any software modifications will probably be directly compatible with Apple CP/M as well. Other peripheral cards may be used if software supplied by the card manufacturer is bound to your Apple CP/M system using the CONFIGIO program. See the Software Details section and the CONFIGIO utility for more information on the implementation of non-standard peripheral cards.

Below is a table of the assigned functions for each of the Apple slots, along with the card types (see above) that are recognized when installed in each. Unless otherwise noted below, unrecognized cards or empty slots are ignored.

IMPORTANT: MAKE SURE your Apple is TURNED OFF before you attempt to rearrange your peripheral cards or serious damage may result to your Apple.

SLOT	VALID CARD TYPES	PURPOSE
0	Not used for I/O	This slot may contain a Language Card or an Applesoft or Integer BASIC ROM card. (The latter are not used by CP/M.)
1	types 2,3,4	Line printer interface (CP/M LST: device)
2	input: 2, 3,4 output: 1,2,3,4	General purpose I/O (CP/M PUN: and RDR: devices)
3	types 2,3,4	Console output device (CRT: or TTY:) The normal Apple 24×40 screen is used as the TTY: device if no card is present.
4	type 1	Disk controller for drives E: and F:. The SoftCard may be installed here if not occupied by a Disk controller card.
5	type 1	Disk controller for drives C: and D:.

SLOT	VALID CARD TYPES	PURPOSE
6	type 1	Disk controller for drives A: and B:. (must be present)
7	any type	No assigned purpose. The SoftCard may be installed in slot 7.

Placement of Apple Disk Drives

As indicated in the table above, Apple Disk II controller cards may be installed in slots 6, 5 or 4. You must have at least one disk drive installed in slot six. Disk controller cards are installed in order downward from slot 6, i.e., your second controller should be installed in slot 5, and the third in slot 4.

In CP/M, each of the drives is assigned a letter name, followed by a colon. For instance, the disk in slot 6, drive 1, is CP/M drive A:. (See table below.) This is the way we will refer to your disk drives throughout this documentation. You may want to label each disk drive according to its assigned CP/M name and it is for just that purpose that we enclosed the package of self-adhesive disk drive labels.

	CP/M name	Slot #	Drive #
1st drive:	A:	6	1
2nd drive:	B:	6	2
3rd drive:	C:	5	1
4th drive:	D:	5	2
5th drive:	E:	4	1
6th drive:	F:	4	2

NOTE for DOS 3.3 or Apple Pascal users:
Apple CP/M supports the large-capacity 16-Sector disk format used by DOS 3.3 and Apple Pascal, in addition to standard Apple II 13-Sector format.

Printer Interface Installation

If you own a printer, its interface card must be installed into slot 1. Most interface cards designed to work with Apple Pascal will work with Apple CP/M as well.

General Purpose I/O Installation

General purpose I/O (such as modems, paper tape readers and punches, etc.) must be installed in slot 2. Only those cards noted in Table 1 will be recognized, although other types of cards may be used with interface software supplied by the manufacturer of the card. For more details on interfacing foreign hardware, see the Software Details section, and the CONFIGIO program in the Software Utilities Manual.

Using an External Terminal Interface

Any of the type 2, 3, or 4 cards of Table 1 can be used to interface an external terminal to Apple CP/M. The terminal interface card must be installed in slot 3.

The SoftCard supports both the Videx Videoterm and M&R Sup-R-Term 24 × 80 character video cards. Other plug-in video boards may be used with interface software supplied by the board manufacturer.

If an interface card is plugged into slot 3, the I/O interface card is used as the terminal device, rather than the Apple 24 × 40 screen and keyboard. If you do have an external terminal interface, we suggest that you remove it from slot 3 and use the normal Apple screen and keyboard until you have configured Apple CP/M for use with your terminal. See CONFIGIO in the Software Utilities Manual.

If you are using an *external* terminal, we suggest that you use either a California Computer Systems 7710A Serial interface or a modified Apple Communications Interface to interface the terminal to your Apple CP/M system. The Apple High Speed Serial Interface will be tolerated, but is not recommended because there is no way for CP/M to check the "status" of this device (i.e., you won't be able to "Ctrl-C" out of a BASIC program).

Installation of the SoftCard

Now you are ready to install the SoftCard. First,

MAKE SURE THAT YOUR APPLE IS TURNED OFF!!

Serious damage to your Apple and to the SoftCard will result if your Apple is left on during installation.

1. With the card laying component-side up in front of you, notice the four small switches on the Apple SoftCard. Make sure that all of these

switches are OFF. The side of the switch nearest the gold-plated edge connector is DOWN when in the off position. This is the standard operating position for Apple CP/M.

2. With your Apple computer positioned with the keyboard directly in front of you, clear the top of the Apple of miscellaneous monitors, disks, old coffee cups and any other junk. Now remove the top by grasping the cover under its rear lip at each corner with one hand at each corner, pulling up gently till the cover pops loose from its fasteners. Then pull the cover directly out toward the rear of the machine and remove it from your Apple. (The power is off isn't it?)

3. Now you must decide into which slot to install the SoftCard. You may plug the card into any unused slot (except slot zero), but we suggest you install it in slot 4. If slot 4 is occupied by a disk controller card, simply choose any other unused slot.

4. Position the SoftCard's connectors directly over the chosen expansion slot on the Apple's circuit board. Holding the board firmly and vertically, push the connector down into the expansion slot. Insure that the SoftCard is inserted all the way by rocking it gently fore and aft while applying downward pressure. Insure that the SoftCard is not tilted down toward the front of the Apple as this could cause the rear connector fingers to not be fully seated in the expansion slot (which would lead to results that are best not thought about).

5. Make sure that all of your peripheral cards are installed correctly as per the instructions on page 1-2.

6. Carefully replace the Apple's cover. Be sure that the corners pop into place and secure the lid. Now you can rearrange all of your junk just as before.

Now your SoftCard system is ready for use. Checkout of the system consists of bringing up CP/M and using it. BUT, before you turn on your Apple, please read the sections on "Bringing Up Apple CP/M" and "How To Copy Your SoftCard Disk." It is possible to destroy your disks if you do not follow the information in the two sections. So, KEEP READING AND DON'T TURN ON ANYTHING YET.

Chapter 2
Getting Started with Apple CP/M

- **Bringing Up Apple CP/M**
- **How to copy your SoftCard Disk**
- **Creating CP/M System Disks**
- **Using Apple CP/M with the Language Card.**
- **I/O Configuration**

In the pages to follow, we will show you how to bring up Apple CP/M. PLEASE read this section CAREFULLY and COMPLETELY before you power up your Apple!! You should read completely and understand all the information on pages 1-2 to 1-7 before proceeding.

Bringing up Apple CP/M

Starting Apple CP/M is simple, but first you must be sure you are using the correct disk.

Two disks are included in the SoftCard package – one in 16-Sector format and the other in 13-Sector format. If you are currently using Apple DOS version 3.3 or Apple Pascal with the Language Card, you must use the 16-Sector disk. If you are currently using DOS 3.2 or earlier, you must use the 13-Sector disk. A 16-Sector disk will NOT boot on a drive set up for 13-Sector disks, and vice-versa.

Select the disk appropriate for your system and insert it into drive A:. (You'll have to start getting used to these drive names – A: is slot 6, drive 1.)

If you have an Apple II Plus or an Apple II with an Autostart ROM installed, simply turn the Apple's power on, which will automatically boot the disk.

If you have a standard Apple II without an Autostart ROM, boot the disk by first turning the Apple's power on, hitting the RESET key, and then typing 6 Ctrl-K RETURN. Ctrl-K is typed by first pressing the key marked CTRL and holding it down while you press the K key.

After a few seconds, the computer will display

```
APPLE II CP/M
44K vers. 2.2X
(C) 1980 MICROSOFT

A>
```

NOTE: If the sign-on message above is not displayed, check to be sure you are using the correct SoftCard disk. Also check to make sure that you have inserted all of your peripheral cards properly.

The "A>" prompt means that CP/M is ready for your command. To see that CP/M is really working, type

```
DIR
```

and press RETURN to display the names of all of the programs on your SoftCard Master disk. The DIR command and the rest of the CP/M commands will be explained in detail later.

But first, you should . . .

MAKE A BACKUP COPY OF YOUR SOFTCARD CP/M MASTER DISK!

and save the original in a nice, safe, dry, non-magnetic place. In fact, it is a good idea to make more than one copy!

How To Copy Your SoftCard Disk

NOTE: The process below works with both single- and multiple-drive systems. For more information on the use of the FORMAT and COPY programs, see the "Software Utilities Manual."

Copying a CP/M disk is a two-step process. The first step is to use the FORMAT program to "format" a blank disk to use as the backup disk. This process initializes the disk so that it can accept data. Next, you use the COPY program to COPY the master disk onto the newly formatted backup disk.

NOTE: CP/M, unlike Apple DOS, does not place the system software on each disk. This means that there are not "slave" or "master" disks in the same sense as with Apple DOS. We refer to the disks shipped with your SoftCard system as "Master Disks" only in the sense that you should save and protect these disks, and not in an operational sense. Also, CP/M disks will not boot up unless the system software is on that particular disk. You must first load CP/M from the system disk before you use any standard CP/M disk.

Formatting the backup disk

Assuming CP/M is up and running (you should see the A> prompt), and you still have the SoftCard disk in drive A:, type:

FORMAT A:

and press RETURN. Soon, the Apple will respond by printing:

```
          APPLE II CP/M
 xx SECTOR DISK FORMATTER    (xx 13 or 16 Sector depending
(C) COPYRIGHT MICROSOFT 1980  on which disk you are using)

INSERT DISK TO BE FORMATTED INTO DRIVE A:
```

Now remove the SoftCard system disk and insert your blank disk. When you are ready to begin, just hit RETURN. Make sure that you have the blank backup disk in the drive when you press RETURN.

The formatting process takes about 30 seconds. The disk drive will remain on during the entire process (you should be able to hear it operating).

When the FORMAT process is complete, the disk will stop and the Apple will type:

```
FORMAT COMPLETED
INSERT SYSTEM DISK AND PRESS RETURN
```

When the red light on the disk drive goes out, remove the newly formatted disk and re-insert the SoftCard system disk. Then press RETURN to return to CP/M. After a second or two, the A> prompt will reappear, letting you know that you have returned to CP/M.

Copying the backup disk

Now you are ready to copy your SoftCard system disk with the COPY program.
Type:

```
COPY A:=A:
```

After a few seconds, the Apple will display:

```
          APPLE II CP/M                (xx is 13 or 16 Sector
xx SECTOR DISK COPY PROGRAM            depending on which disk you
      (C) MICROSOFT 1980               are using)

INSERT MASTER DISK    PRESS RETURN
```

Since you want to copy the disk that is already in drive A:, just press RETURN to begin the COPY process. The disk will whirr for a few seconds, then the computer will print:

```
INSERT SLAVE DISK     PRESS RETURN
```

Remove the SoftCard Master disk and insert your freshly formatted backup disk into drive A: and hit RETURN. Again after a few seconds, the computer will prompt:

INSERT MASTER DISK
PRESS RETURN

Now remove the backup disk and re-insert the master disk, and hit RETURN.

Finally, the computer will ask you to re-insert the slave disk. This process will be repeated three times.

After you have inserted the slave disk into drive A: for the last time, the computer will display:

COPY COMPLETE
DO YOU WISH TO MAKE ANOTHER COPY? (Y/N)
PRESS RETURN

Since the disk in the drive is an exact copy of the SoftCard disk, you do not need to re-insert the SoftCard Master Disk. You should now store the SoftCard Master Disk away in a nice, safe, dry, non-magnetic place for safekeeping.

It is a good idea to make at least two backup copies of your SoftCard Master Disk. If you ever have problems that are not immediately identifiable as hardware or software, having a second backup will allow you to test your system without risking your SoftCard Master Disk.

If you have a Language Card, you should definitely make at least three copies as you will need to modify CP/M to take advantage of the additional Language Card memory. We strongly recommend that you do this modification on backup disks and not on your SoftCard Master Disk.

Creating CP/M System Disks

A CP/M System disk is a disk that will load and initialize CP/M when booted. Creation of CP/M System disks is a two step process: first you must FORMAT the disk, then you must use the COPY program to write the CP/M system onto the disk that will load and run when the system is booted. Below is outlined the process for creating system disks:

1. Use the FORMAT program to format a blank disk. This process is exactly the same as the FORMAT process that you used to copy your SoftCard Master disks earlier.

2. Next, you must use the COPY program to write the CP/M system onto the disk. This is done using the "/S" option as shown below:

Use of the COPY program

1. Insert a CP/M system disk that contains the COPY program into drive A: and boot your system. When you see the A> prompt, type

   COPY A:=A:/S

   The "/S" means that you only want to copy the CP/M system, not the entire disk. After a second, the computer will display

   INSERT MASTER DISK PRESS RETURN

   Since your disk containing COPY also contains CP/M, just leave the current disk in drive A and press RETURN.

   The disk will whirr for a few seconds then the computer will display the message:

   INSERT SLAVE DISK PRESS RETURN

   Then, insert the disk you want to write the CP/M system onto, and hit RETURN. After a few seconds, the disk will stop and the computer will display

   INSERT CP/M SYSTEM DISK INTO DRIVE A:
   PRESS RETURN

   Since the disk in the drive is now a CP/M system disk, you can just hit RETURN to return to CP/M.

   Your new CP/M system disk will now boot and can be used to store programs and data.

   If you have more than one disk drive, or if you wish to create more than one system disk at a time, you should read the "Software Utilities Manual" for more complete information on the use of FORMAT and COPY.

Using Apple CP/M with the Apple Language Card

If you are using the Apple Language Card, it is possible to take advantage of the extra 12K of addressable memory contained on the card. This extra 12K of memory makes 56K of contiguous memory space available for use with CP/M. First, however, you must update your CP/M system disk so that 56K CP/M, rather than 44K CP/M, will be invoked when the disk is booted. This is done with the CPM56 utility.

NOTE: Updating your CP/M disks in this way does not affect the operation of CP/M. However, a 56K CP/M disk will *NOT BOOT* on a system that is not equipped with a Language Card. We suggest that you do NOT update your SoftCard CP/M Master disk to 56K CP/M. Instead, use one of the backup copies you have just finished making.

To use the CPM56 utility, first make sure CP/M is up and running, (you should see the "A>" prompt) and insert your backup copy of the SoftCard system disk. Then, type:

CPM56 A:

and hit RETURN. Once you press RETURN, the computer will automatically update your disk. When the conversion is complete, the computer will display the message

DISK IN DRIVE A: HAS BEEN UPDATED TO 56K

You now have a diskette containing CP/M configured for a 56K system. To load this new version, RE-BOOT your system by either hitting RESET, 6, Ctrl-K (if you don't have an Autostart ROM), or by turning your Apple off and back on again. Soon, the prompt message will re-appear, this time displaying "56K CP/M" instead of ""44K CP/M."

I/O Configuration

I/O Configuration is the last step in setting CP/M up for your system. This step is not necessary on all systems but you will need to perform it *IF*:

1. You are using an external terminal

2. You wish to patch non-standard I/O software to the CP/M system

The CONFIGIO program is used to perform all of the system configuration process described below. Read the section on CONFIGIO in the "Software Utilities Manual" carefully for more information on the use of CONFIGIO.

Here are the final configurations that may be performed:

Redefining Keyboard Characters – If you wish to make it possible to type a character that is not normally available on the Apple keyboard (or on your external terminal if you use one), you can use the CONFIGIO utility of Apple CP/M to redefine the ASCII value that is assigned to any particular key on the keyboard. Since many CP/M programs use characters not found on the Apple keyboard, you will probably find it valuable to use this option. See both Chapter 2 of the "Software and Hardware Details Manual," and CONFIGIO in the "Software Utilities Manual" for complete information on redefining keyboard characters.

Loading User I/O Driver Software – The I/O Configuration Block also provides for the support of non-standard Apple peripherals and I/O software. To interface a non-standard peripheral (i.e., a peripheral that the SoftCard does not normally support, see list on page 1-2), you must load the interface software provided by the peripheral manufacturer into the I/O Block. There are specific restrictions regarding the software that can be loaded. For a complete description of these restrictions and for the actual loading process, see both Chapter 2 of the "Software and Hardware Details Manual" and "CONFIGIO" in the "Software Utilities Manual."

Configuring Apple CP/M for use with an External Terminal – If you are using an external terminal, you must configure Apple CP/M for use with your terminal. This configuration process is necessary because Apple CP/M supports a number of special screen and cursor control functions (e.g. Clear Screen and Address Cursor) that are used by a number of CP/M programs, such as Microsoft BASIC and the many CP/M word processors. These screen functions are invoked on most terminals by sending a sequence of characters to the terminal, which then performs the appropriate function. So, Apple CP/M must be made to recognize the particular screen function command sequences for your terminal.

Apple CP/M supports most popular video terminals, including the SOROC IQ 120/140, the Hazeltine 1500/1510, and the popular 24 × 80 plug-in video boards, such as the Videx Videoterm and the M&R Sup-R-Term.

As mentioned earlier in the section on installation of the SoftCard, the terminal interface card must be installed in slot 3 of your Apple. "See Apple Peripheral Cards: What Goes Where," page 1-2 for more information on the types of terminal interface cards supported by Apple CP/M.

Terminal configuration is done using a program written in Microsoft BASIC: CONFIGIO. The use of this program, and the procedure for configuring Apple CP/M to your system can be found in the "Software Utilities Manual."

Chapter 3
An Introduction to Apple CP/M

- Typing at the Keyboard
- Output Control
- CP/M Warm Boot: Ctrl-C
- Changing CP/M Disks
- CP/M Command Structure
- CP/M File Naming Conventions
- File Name Specification
- Some CP/M Commands: DIR, ERA, REN, TYPE
- CP/M Error Messages
- Definitions of Programs Included on the SoftCard Disk

The information presented in this section is intended to be used as a short introduction to CP/M on the Apple II. It will help you get started using CP/M but is in no way intended to replace the standard CP/M documentation as a guide to the complete usage of CP/M. Read the CP/M Reference Manual carefully.

The heart of the CP/M operating system does not lie in the power of its built-in keyboard commands. Instead, CP/M was designed as a link between a computer's hardware and its software. This is the reason for its wide popularity – a program written for CP/M on one machine can be easily transported to another.

Most CP/M "commands" (with the exception of a few such as DIR) are actually *programs* on a disk and so are extensible. To invoke commands of this type, the appropriate disk must be in your drive. Commands executed by loading their program code from the disk in this way are called "transient commands." The COPY and FORMAT commands you used to back up your system disk are transient commands.

Typing at the Keyboard

Typing at the keyboard with CP/M is quite a bit different than with Integer BASIC or Applesoft. The backspace key deletes the character under the cursor as it moves, and the forward arrow key doesn't work. None of the ESCape key cursor movement/editing features are supported.

However, CP/M supports a few line editing features that are useful when typing at the keyboard. There are also some other important control characters that can be used to perform other useful functions. (Remember: Control characters (denoted by "Ctrl-") are typed by first hitting the CTRL key and holding it down while you type the indicated character).

<--	Backspaces one character position. The backspace key deletes the character under the cursor. (Also invoked with Ctrl-H)
Ctrl-X	Backspaces up to the beginning of the line, deleting the line.
Ctrl-R	Retypes current line.
Ctrl-J	Terminates input same as RETURN key. (Also invoked with LINE FEED)
Ctrl-E	Physical end of line. Cursor is moved to beginning of next line, but line is not terminated until RETURN is typed.
RUBOUT	Deletes and "echos" (reprints) the last character typed. Also referred to as DEL or DELETE. (Type Ctrl-@ to get RUBOUT on the Apple keyboard – see below)

memory at all times which is used to allocate space on the disk. When you change disks, this information must be replaced with the directory information of the newly inserted disk.

To let CP/M know that you have changed disks, type Ctrl-C to execute a CP/M "Warm Boot." Make sure you do this AFTER you have changed disks. This will cause the disk directory information in the drive to be updated. You should get used to typing Ctrl-C often.

If you do not type Ctrl-C before changing disks and a WRITE is attempted to the changed disk, the computer will display

BDOS ERR ON x:Disk R/O (Where x: is a disk drive A:-F:)

(R/O stands for Read Only) When you receive this error message, hit RETURN. This will perform a CP/M warm boot and return you to CP/M. The above error condition applies only to changed disks that are to be WRITTEN. No error will result if you attempt to READ from the changed disk.

Many CP/M programs perform a warm boot upon termination. So, you need not type Ctrl-C to change disks after execution of programs of this type. After a while you will probably recognize the sound of your Apple disk drive during a CP/M warm boot. This is one way to know whether a program performs a warm boot upon completion.

For more information, read the "CP/M Reference Manual – An Introduction to CP/M Features and Facilities." Also see "CP/M Error Messages" later in this section.

The RESET Key

Pressing the RESET will have different effects, depending on whether your system has an Autostart ROM or not.

On a system that has an Autostart ROM. Pressing the RESET key while in CP/M will cause a CP/M warm boot, and you will return to CP/M. Pressing the RESET key while in either MBASIC or GBASIC will result in a "Reset Error," which can be trapped using ON ERROR GOTO, etc.

On a system that does not have an Autostart ROM. You can recover from hitting RESET by typing Ctrl-Y then pressing RETURN. You will then either re-boot CP/M (if you hit RESET while in CP/M) or return to BASIC with a "Reset Error" (if you hit RESET while in MBASIC or GBASIC).

There are a few characters that are normally unavailable on the Apple's keyboard. These have been assigned to certain control characters so that they are available to you:

Type:	To get:
Ctrl-K	[(Left Bracket)
Ctrl-@	RUBOUT
Ctrl-B	\ (Backslash)

These characters are often required by CP/M commands and programs. To change (or do away with) these assignments, or add additional ones, see the CONFIGIO program in the "Software Utilities Manual."

Output Control

There are two control characters that are used to control character output to the screen and printer:

Ctrl-S	Temporarily stops character output to the terminal. Program execution and character output resume when any character is typed.
Ctrl-P	Sends all character output to the line printer device as well as to the terminal. This "printer echo" mode remains in effect until the next Ctrl-P is typed.

CP/M Warm Boot: Ctrl-C

There is also another important control character: Ctrl-C. When typed as the first character of a line, Ctrl-C is used to perform a CP/M "Warm Boot," causing CP/M to be reloaded from the disk to insure that the CP/M in memory is in working order. (This is NOT the same as a *Cold* Boot. A Cold Boot is the act of booting the CP/M disk for the first time.) You should ALWAYS type Ctrl-C whenever you change disks. (See "Changing CP/M Disks," below.)

Ctrl-C	Perform a CP/M warm boot.

Changing CP/M Disks

Unlike Apple DOS, you cannot indiscriminately change disks in drives with CP/M. When you change disks, you must let CP/M know that you have done so. This is because there is certain disk directory information stored in

CP/M Command Structure

When you see the "A>" prompt, you know that CP/M is ready for your command. The "A" in the prompt means that drive A: is the "currently logged drive." The "currently logged drive" is the default drive that is used in a file specification if another drive is not specified. It is also the drive that CP/M searches for transient commands if a drive is not specified in the command.

CP/M commands themselves are generally very simple. There are only a handful of non-transient commands, the most useful of which are DIR, ERA, and REN. The DIR command is used to display a disk directory, the ERA command is used to erase disk files, and the REN command is used to rename disk files.

CP/M File Naming Conventions

Before you are introduced to these CP/M commands, you should become familiar with CP/M disk file naming conventions. CP/M file names are very different than those used with Apple DOS. A file name may be up to 8 characters long, with an optional 3 character "extension." This is a handy construct that lets you identify related files on the disk.

File Name Specification

The CP/M file specification structure allows you to refer to one *or more* files with a single specification. Files are usually specified in a command by typing the name (up to 8 characters), followed by a period (".") and the 3 character extension. It is also possible to specify the drive in which the file is located. This is done by preceding the file name with the drive name. If no drive is specified, the currently logged drive is assumed. Below are some examples of valid CP/M file name specifications:

A:FNAME.EXT	Refers to file FNAME.EXT on drive A:
TEMP.OLD	Refers to file TEMP.OLD on the currently logged drive
B:TEMP.NEW	Refers to file TEMP.NEW on drive B:

The 3-character extension usually provides information about the internal format of a file. The most important of these common extensions is COM, which stands for COMMAND. Any file with an extension of COM is a transient command type file and can be invoked by simply typing its name (without the .COM). Other common extensions are BAS, used for BASIC programs; and HEX, ASM, and PRN, which are used (and produced) by the ASM program, which is the CP/M 8080 assembler.

File specifications can also be used to refer to more than one file at a time. This is done by the use of "wild card" file name specifications. A question mark used in a file name is a "wild card" character, that is, it will match any character in that position when searching the directory for the file name match. An asterisk ("*") is used to match any string of characters. For instance,

```
B:TEMP.???
or
B:TEMP.*
```

refer to both TEMP.OLD and TEMP.NEW on drive B:, if they exist. Below are some more examples of "wild card" file specifications:

A:*.COM	Refers to all files on drive A: with an extension of COM
B:*.*	Refers to all files on drive B:
B:????????.???	Exactly the same as B:*.* above.
DUMP.*	Refers to all files on the currently logged disk beginning with "DUMP"
C*.*	Refers to any file on the currently logged disk beginning with the letter "C"

Note that an "*" is actually an abbreviation of a string of "?"s.

Some CP/M Commands: DIR, ERA, REN, TYPE

These are the four most commonly used built-in CP/M commands. DIR is used to display the directory of all files on a disk; ERA is used to erase disk files; REN is used to rename disk files; and TYPE is used to display a text file on the terminal. Below is a short introduction to each.

NOTE: The information below is meant only as an introduction to a few of the CP/M commands. For more complete information about these and other CP/M commands, see the "CP/M Reference Manual – An Introduction to CP/M Features and Facilities."

The DIR Command

The DIR command is used to display the names of the files on a disk. To display the directory of all the files on the currently logged disk, type

```
DIR
```

and press RETURN. To display a directory of the disk in another drive, just include the drive name. For instance,

```
DIR B:
```

will display the directory of the disk in drive B:.

If a file specification is included with the DIR command, only those files whose names match the file specification will be displayed. Here are some examples of the DIR command used with file specifications:

DIR MBASIC.COM	Displays MBASIC.COM if the file exists on the currently logged disk.
DIR A:*.COM	Displays all files with an extension of COM on drive A:
DIR B:	Displays all files on drive B:
DIR A:A*.*	Displays all files on drive A: whose name begins with the letter "A"

If there are no files on the disk, or if no files match the file specification, CP/M will respond

```
NO FILE
```

The ERA Command

The ERA command is used to erase files on the disk. You must always include a file specification with this command.

NOTE: *Don't* delete any of the files on your CP/M disk! If you do, you'll have to make another backup copy of the SoftCard Master disk.

Here are a few examples of the use of the ERA command:

ERA B:TEMP.OLD	Erase the file TEMP.OLD on drive B:
ERA C:*.BAK	Erase all files on drive C: with extension BAK
ERA *.*	Erase all of the files on the currently logged disk. If you attempt to erase all of the files on a disk, CP/M will ask ALL (Y/N)?. If you don't want to delete all the files on the disk, respond by typing "N"

Notice that you can erase more than one file at a time with ERA by using the wild card naming convention.

The REN Command

The REN command is used to rename files. Here is the general format of this command:

REN newname = oldname

where "newname" and oldname" are file specifications. You *cannot* use wild card file specifications with the REN command. You *can* precede the first file specification with a drive name. Below are some examples of the use of the REN command:

REN TEMP.NEW=TEMP.OLD	Rename TEMP.OLD as TEMP.NEW
REN B:PEAR.COM=APPLE.COM	Rename APPLE.COM on drive B: as PEAR.COM

NOTE: Unlike Apple DOS, the new file name *precedes* the existing file name (as in algebra, what's on the left side of the " = " becomes what's on the right).

The TYPE Command

The TYPE command is used to display the contents of a text file on the terminal. You must include a file specification. (Wild card file specifications are not allowed.)

For example, to display the contents of the file DUMP.ASM on the screen, type

TYPE DUMP.ASM

and press RETURN. If you attempt to TYPE a file that is not a text file, only junk will appear.

NOTE: DUMP.ASM is the only text file on the SoftCard Master disk.

CP/M Error Messages

There are four possible CP/M error messages. Below is listed each message, followed by a list of the possible causes, in the order of their likelihood:

BDOS ERR ON x:BAD SECTOR

(Where x: is a disk drive A:-F:)

This error message can mean any number of things – it does NOT necessarily mean that there is a bad sector on your disk (but it could!). This error message is roughly equivalent to the Apple DOS "DISK I/O ERROR" message. Possible causes:

1. No disk in drive
2. Drive door not closed
3. Disk inserted improperly
4. An attempt was made to access a drive not installed in a controller card (See SELECT error below)
5. A bad disk

When you receive a BAD SECTOR error, CP/M waits for you to type a character from the keyboard. If you type Ctrl-C, a Warm Boot will be performed and you will return to CP/M command mode. Type R to retry the read or write and continue execution. Any other character will cause the error to be ignored and resume execution of the program or operation.

BDOS ERR ON x:R/O

(where x: is a disk drive A:-F:)

This error message usually means one of two things:

1. You have changed the disk in a drive without typing Ctrl-C
2. There is a write-protect tab covering the notch in the side of your disk

When you receive this error message, CP/M will wait for you to type a character at the keyboard. After you do so, a warm boot will be performed and you will be returned to CP/M.

BDOS ERR ON x:FILE R/O

(where x: is a disk drive A:-F:)

This error message can mean only one thing:

1. A write was attempted to a file that was marked Read Only with the STAT program

When you receive this error message, CP/M will wait for you to type a character at the keyboard. Type any key to perform a warm boot and return to CP/M.

For more information on write protection of files with STAT, consult the "CP/M Reference Manual – An Introduction to CP/M Features and Facilities."

BDOS ERR ON x:SELECT
(where x: is a disk drive A:–F:)

This error message can mean only one thing:

1. An attempt was made to access a non-existent disk drive

When you receive this error message, CP/M will wait for you to enter a character from the keyboard. Type any character to perform a CP/M warm boot and return to CP/M.

NOTE: If you only have one drive attached to a disk controller card in your Apple (as is the case with a single-drive system), attempting to access the drive that is not installed will result in a BAD SECTOR error instead of a SELECT error.

Description of Programs Included on the SoftCard Disk

MBASIC, GBASIC and a number of utility programs are found on the SoftCard disk. All of these programs are described in detail in other sections of the SoftCard Documentation package. Below is a synopsis of the purpose of each program, followed by a reference stating where the complete program documentation can be found.

APDOS

This utility program allows you to transfer data from your Apple DOS disks to CP/M disks. APDOS may be used to transfer text and binary files only. (Requires 2 or more disk drives.)

See the "Software Utilities Manual."

ASM

ASM is the CP/M 8080 assembler. ASM can be used along with DDT to write and debug 8080 assembly language programs.

See the "CP/M Reference Manual," Chapter 4.

CONFIGIO

The CONFIGIO utility is used to configure the Apple CP/M operating environment to your particular system configuration. It has four major functions – to configure I/O for an external terminal, to redefine keyboard characters, to load user I/O software, and to read and write to the I/O Configuration Block. For more information about the function of I/O Configuration, see Chapter 2 of the "Software and Hardware Details Manual" in addition to the "Software Utilities Manual."

See "Software Utilities Manual."

COPY

The COPY program is used to copy CP/M disks, or to create blank CP/M system disks from a newly formatted disk.

See the "Software Utilities Manual."

DDT

DDT is the CP/M Dynamic Debugging Tool. It allows dynamic interactive testing and debugging of 8080 assembly language programs.

See the "CP/M Reference Manual," Chapter 5.

DOWNLOAD

The DOWNLOAD and UPLOAD utilities enable the user to transfer CP/M files from another CP/M machine to the Apple by means of an RS-232 serial data link. UPLOAD is not included on either of the Apple CP/M disks. Use of these programs requires a working knowledge of 8080 assembly language programming and thus are intended for experienced programmers only.

See the "Software Utilities Manual."

DUMP

DUMP displays the contents of a disk file in hexadecimal form. DUMP.ASM is the source listing of the DUMP program, given in Chapter 2 of the CP/M Interface Guide, as an example of an 8080 assembly language program written for the CP/M environment.

See the "CP/M Reference Manual," Chapter 1 and Chapter 2.

ED

ED is the CP/M text editor. It is used to create and edit CP/M text files.

See the "CP/M Reference Manual," Chapter 3.

FORMAT

FORMAT formats a blank disk so that it can accept data. A freshly formatted disk will not boot, but it can be used to store programs and data. Use COPY to make a newly formatted disk into a CP/M system disk.

See the "Software Utilities Manual."

LOAD

LOAD is used to convert a disk file of extension .HEX into a machine-executable .COM file. LOAD can be used to convert output from the assembler into machine executable code.

See "CP/M Reference Manual," Chapter 1, "Introduction to CP/M Features and Facilities."

MBASIC

This is Microsoft BASIC. This version of BASIC is disk BASIC that supports low-resolution graphics, sound, and game controls in addition to many features not found in Applesoft. This version does not support high-resolution graphics.

See the "Microsoft BASIC Reference Manual" for more information.

PIP

PIP is one of the most frequently used CP/M programs. It is used to transfer files from one disk to another. It is also used to copy and append disk files. PIP may also be used to transfer files to the terminal devices and to the printer.

See the "CP/M Reference Manual," Chapter 1. For copying an entire disk, or for copying the CP/M system itself to another disk, see COPY in the "Software Utilities Manual."

STAT STAT provides general status information about disk capacity, file sizes, file indicators and device assignments. File indicators and device assignments can also be altered using this program.

See the "CP/M Reference Manual," Chapter 1.

SUBMIT SUBMIT allows CP/M commands and program input lines to be executed from a disk file rather than from the keyboard for automatic processing.

See the "CP/M Reference Manual," Chapter 1.

XSUB XSUB, when used with SUBMIT, allows character input from a disk file *at all times during* execution of programs.

See the "CP/M Reference Manual," Chapter 1.

The following three programs are found only on the 16-Sector SoftCard disk:

CPM56 CPM56 is used to update a 44K CP/M system disk to a 56K system disk for use with the Apple Language Card. CPM56 cannot be used with 48K Apple systems.

See the "Software Utilities Manual."

GBASIC GBASIC is the same as MBASIC except that it also supports high-resolution graphics.

See the "BASIC Reference Manual" for more information.

RW13 RW13 is used to allow 16-Sector CP/M to access files on a 13-Sector CP/M disk. Used with PIP, RW13 is especially useful for transferring files from a 13-Sector to a 16-Sector diskette. (Requires 2 or more disk drives.)

See the "Software Utilities Manual."

Chapter 4
Getting Started with Microsoft BASIC

Once you have made backup copies of your SoftCard disk, you'll be ready to begin exploring Microsoft BASIC. As mentioned previously, two versions of BASIC are included in the SoftCard package.

MBASIC Includes all of Microsoft BASIC, Version 5.0, plus low-resolution graphics and some other Applesoft extensions. (A comparision of MBASIC with Applesoft is included in the "BASIC Reference Manual.") MBASIC is found on both the 13-Sector and 16-Sector disks. The name of the file is MBASIC.COM.

GBASIC Includes all of the features of MBASIC *plus* high-resolution graphics. GBASIC is found only on the 16-Sector disk and its filename is GBASIC.COM.

To bring up either MBASIC or GBASIC, you must first be at CP/M command level as indicated on the screen by the A> prompt. If you don't see the prompt, return to page 1-8 Loading CP/M.

The initialization instructions below refer to MBASIC, but may also be used for loading GBASIC simply by substituting GBASIC where MBASIC is typed. Use of the two BASICs is identical except that in GBASIC you also have high-resolution graphics commands available to you.

Once you see the A> prompt, simply type:

MBASIC

then press RETURN. The computer will reply:

```
BASIC-80 Version 5.xx
Apple CP/M Version
Copyright © 1980 by Microsoft
Created: dd-mm-yy
xxxx Bytes Free
Ok
```

and BASIC is ready to accept commands.

Initialized in this way, BASIC sets certain default parameters: 3 files may be open at any one time during execution of a BASIC program; all the memory up to the start of FDOS in CP/M may be used and the maximum record size is set at 128.

If you wish to set these parameters (which are explained further in the "Microsoft BASIC Reference Manual") yourself, you can set certain "switches" when you type in the initialization command. You can also specify a program in the command line to be automatically run when the command is entered. This extended command line format is:

MBASIC [⟨filename⟩] [/F:⟨number of files⟩] [/M:⟨highest memory location⟩] [/S:⟨maximum record⟩]

(The square brackets ([]) indicate items that are optional and the angle brackets (⟨⟩) indicate items to be specified by you.)

The ⟨filename⟩ option allows you to RUN a program automatically after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long.

The /F:⟨number of files⟩ option sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes plus 128 (or number specified by /S:) bytes of memory. The ⟨number of files⟩ may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /M; ⟨highest memory location⟩ option sets the highest memory location that will be used by MBASIC. In some cases, it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, the highest memory location should be below the start of FDOS (whose address is contained in locations 6 and 7). The ⟨highest memory location⟩ may be decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /S:⟨maximum record size⟩ option sets the maximum size to be allowed by random files. Any integer may be specified, including integers larger than 128.

Here are a few examples of the different initialization options:

A>MBASIC PAYROLL.BAS	Use all memory and 3 files; load and execute PAYROLL.BAS
A>MBASIC INVENT/F:6	Use all memory and 6 files; load and execute INVENT.BAS

A>MBASIC/M;32768	Use first 32K of memory and 3 files
A>MBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files and execute DATACK.BAS

When BASIC is initialized, it types the prompt "Ok." "Ok" means BASIC is at command level, that is, it is ready to accept commands. At this point, it may be used in either direct or indirect mode.

You can now write programs in either MBASIC or GBASIC, depending on which you initialized. Programming in Microsoft BASIC is like programming in Applesoft, but with significantly more power. See the "Microsoft BASIC Reference Manual" for complete documentation on programming in Microsoft BASIC.

This completes the Installation and Operations portion of this manual. At this point, you should have the SoftCard installed and have both CP/M and BASIC up and running. Throughout this section, we have referred you to other sections of the manual for more detailed information. These other sections *are very detailed* and should contain all the information that you need. If after searching carefully, you still cannot find some information, contact your dealer or write a letter to Microsoft Consumer Products. Enjoy yourself! We sincerely hope you will find the SoftCard an exciting and useful addition to your Apple.

Software and Hardware Details

PART 2
SOFTWARE AND HARDWARE DETAILS

Chapter 1
Apple II CP/M Software Details

Chapter 2
Apple II CP/M I/O Configuration Block

Chapter 3
Hardware Description

CHAPTER 1
APPLE II CP/M SOFTWARE DETAILS

- **Introduction**
- **I/O Hardware Conventions**
- **6502/Z-80 Address Translation**
- **Apple II CP/M Memory Usage**
- **Assembly Language Programming with the SoftCard**
- **ASCII Character Codes**
- **Interrupt Handling**

Introduction

This chapter deals with the software features that are peculiar to Apple II CP/M, and how these features relate to the I/O hardware installed in the different slots of the Apple. First we will discuss the hardware I/O protocol supported by Apple CP/M. Then we will examine the software support of this hardware protocol: the I/O Configuration Block. For more information on the use of the CP/M operating system, see the "CP/M Reference Manual."

I/O Hardware Conventions

The I/O hardware protocol is identical to that supported by the initial release of Apple PASCAL, with a few exceptions. All standard Apple I/O peripherals are supported, as well as a few others, such as California Computer Systems' 7710A Asynchronous Serial Interface, the Videx Videoterm, and M&R Enterprises Sup-R-Term. Apple CP/M does not support horizontal scrolling on the Apple 24 × 40 video screen.

Apple Peripheral Cards: What Goes Where
Unlike Applesoft and Integer BASIC (but similar to Apple PASCAL), Apple CP/M requires that peripheral I/O cards be plugged into specific slots depending on their functions. For instance, a printer interface card must be plugged into slot one in order to use a printer. When the system is booted, CP/M is able to recognize the presence or absence of certain standard Apple peripheral interface cards. Once the system is booted, I/O is performed by using either the hardware directly or by calling the 6502 software on the card.

Below is a table of the assigned functions for each of the Apple slots, along with the card types that are recognized when plugged into each. (See the list of recognized card types following the table.) Note that unless otherwise noted below, unrecognized cards or empty slots are ignored.

SLOT	VALID CARD TYPES	PURPOSE
0	Not used for I/O	This slot may contain a Language Card or an Applesoft or Integer BASIC ROM card. (the latter are not used by CP/M)
1	types 2,3,4	Line printer interface (CP/M LST: device)
2	input: 2,3,4 output: 1,2,3,4	General purpose I/O (CP/M PUN: and RDR: devices)

3	types 2,3,4	Console output device (CRT: or TTY:) The normal Apple 24 × 40 screen is used as the TTY: device if no card is present.
4	type 1	Disk controller for drives E: and F:
5	type 1	Disk controller for drives C: and D:
6	type 1	Disk controller for drives A: and B: (must be present)
7	any type	No assigned purpose. The SoftCard may be installed in slot 7.

NOTE: The SoftCard may be installed in any empty slot except slot zero.

Below is a list of the I/O peripheral card types that are currently recognized by Apple CP/M.

TYPE	CARD NAME
1	Apple Disk II Controller
2	Apple Communications Interface
	*California Computer Systems 7710A Serial Interface
3	Apple High Speed Serial Interface
	Videx Videoterm 24 × 80 Video Terminal Card
	M&R Enterprises Sup-R-Term 24 × 80 Video Terminal Card
4	Apple Parallel Printer Card

*The CCS 7710A serial interface card is the preferred type 2 card as it supports hardware handshaking and variable baud rates from 110-19200 baud.

6502/Z-80 Address Translation

Because of the memory address translation performed by the hardware on the SoftCard, a particular data byte is not accessed at the same address for both processors. The correspondence of memory addresses between the Z-80 and 6502 is shown below (All addresses are hexadecimal). Use of this table is necessary when translating 6502 BASIC or assembly language software for use with the SoftCard.

Z-80 ADDRESS	6502 ADDRESS	
0000H-0FFFH	$1000-$1FFF	Z-80 location zero
1000H-1FFFH	$2000-$2FFF	
2000H-2FFFH	$3000-$3FFF	
3000H-3FFFH	$4000-$4FFF	
4000H-4FFFH	$5000-$5FFF	
5000H-5FFFH	$6000-$6FFF	
6000H-6FFFH	$7000-$7FFF	
7000H-7FFFH	$8000-$8FFF	
8000H-8FFFH	$9000-$9FFF	
9000H-9FFFH	$A000-$AFFF	
0A000H-0AFFFH	$B000-$BFFF	
0B000H-0BFFFH	$D000-$DFFF	
0C000H-0CFFFH	$E000-$EFFF	
0D000H-0DFFFH	$F000-$FFFF	6502 RESET, NMI, BREAK vectors
0E000H-0EFFFH	$C000-$CFFF	6502 memory mapped I/O
0F000H-0FFFFH	$0000-0FFF	6502 zero page, stack, Apple screen

Apple II CP/M Memory Usage

Here is how the Apple memory is used by Apple CP/M:

6502 ADDRESS	Z-80 ADDRESS	PURPOSE
$800-$FFF	0F800-0FFFF	Apple disk drivers and disk buffers
$400-$7FF	0F400-0F7FF	Apple screen memory
$200-$3FF	0F200H-0F3FFH	I/O Configuration Block.
$000-$1FF	0F000H-0F1FFH	Reserved 6502 memory area – 6502 stack and zero page.
$C000-$CFFF	0E000H-0EFFFH	Apple memory mapped I/O
$FFFA-$FFFF	0DFFAH-0DFFFH	6502 RESET, NMI, and BREAK vectors.
$D400-$FFF9	0C400H-0DFF9H	56K Language Card CP/M (if Language Card installed)
$D000-$D3FF	0C000H-0C3FFH	Top 1K of free RAM space with 56K Language Card CP/M
$A400-$BFFF	9400H-0AFFFH	44K CP/M. (Free memory with 56K CP/M)
$1000-$A3FF	0000H-093FFH	Free RAM (CP/M uses lowest 256 bytes)

Assembly Language Programming with the SoftCard

The Z-80 processor executes all of the 8080 instruction set plus its own set of instructions. You can run software written for either the 8080 or Z-80 processor on the SoftCard. There is, however, a different set of instruction mnemonics for each of the processors.

Included with the standard CP/M utilities are ED, a line oriented text editor; ASM, an 8080 assembler; and DDT, an 8080 machine language debugger. These programs can be used to write and debug 8080 programs.

It is also possible to write 6502 subroutines for use with the SoftCard. The Microsoft Assembly Language Development System is available separately for the development of both Z-80 and 6502 software.

ASCII Character Codes

DEC = ASCII decimal code
HEX = ASCII hexadecimal code
CHAR = ASCII character name

DEC	HEX	CHAR	WHAT TO TYPE
0	00	NULL	ctrl @
1	01	SOH	ctrl A
2	02	STX	ctrl B
3	03	ETX	ctrl C
4	04	ET	ctrl D
5	05	ENQ	ctrl E
6	06	ACK	ctrl F
7	07	BEL	ctrl G
8	08	BS	ctrl H *or* ←
9	09	HT	ctrl I
10	0A	LF	ctrl J
11	0B	VT	ctrl K
12	0C	FF	ctrl L
13	0D	CR	ctrl M *or* RETURN
14	0E	SO	ctrl N
15	0F	SI	ctrl O
16	10	DLE	ctrl P
17	11	DC1	ctrl Q
18	12	DC2	ctrl R

19	13	DC3	ctrl S
20	14	DC4	ctrl T
21	15	NAK	ctrl U *or* →
22	16	SYN	ctrl V
23	17	ETB	ctrl W
24	18	CAN	ctrl X
25	19	EM	ctrl Y
26	1A	SUB	ctrl Z
27	1B	ESCAPE	ESC
28	1C	FS	ctrl [
29	1D	GS	ctrl shift-M
30	1E	RS	ctrl ^
31	1F	US	ctrl _
32	20	SPACE	space
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C

68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	[	[
92	5C	\	\
93	5D	]	] (shift-M)
94	5E	^	^
95	5F	_	_
96	60	'	'
97	61	a	a
98	62	b	b
99	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q
114	72	r	r
115	73	s	s
116	74	t	t

117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	{	{
124	7C	|	|
125	7D	}	}
126	7E	~	~
127	7F	RUB	

Interrupt Handling

Because of the way the 6502 is "put to sleep" by the SoftCard using the DMA line on the Apple bus, ALL interrupt processing must be handled by the 6502. An interrupt can occur at two times: while in Z-80 mode and while in 6502 mode.

Handling an interrupt in 6502 mode:
Handle the interrupt in the usual way – simply end the interrupt processing routine with an RTI instruction.

Handling an interrupt in Z-80 mode:
Both processors are interrupted when an interrupt occurs in Z-80 mode. Here is the step-by-step process for handling an interrupt while in Z-80 mode:

1. Save any registers that are destroyed on the stack.

2. Save the contents of the 6502 subroutine call address (See Calling of 6502 Subroutines above) in case an interrupt has occurred during a 6502 subroutine call.

3. Set up the 6502 subroutine call address to $FF58, which is the address of a 6502 RTS instruction in the Apple Monitor ROM.

4. Return control to the 6502 by performing a write to the address of the SoftCard (again see Calling of 6502 Subroutines).

5. When control is returned to the Z-80, restore the previous 6502 subroutine call address.

6. Restore all used Z-80 registers from the stack.

7. Enable interrupts with an EI instruction.

8. Return with a RET instruction.

CHAPTER 2
APPLE II CP/M
I/O CONFIGURATION BLOCK

- Introduction
- Console Cursor Addressing/Screen Control
  The Hardware/Software Screen Function Table
  Terminal Independent Screen Functions/Cursor Addressing
- Redefinition of Keyboard Characters
- Support of Non-Standard Peripherals and I/O Software
  Assigning Logical to Physical I/O Devices: the IOBYTE
  Patching User Software Via the I/O Vector Table
- Calling of 6502 Subroutines
- Indication of Presence and Location of Peripheral Cards

Introduction

The I/O Configuration Block contains the information necessary to interface Apple CP/M to the various hardware and software configurations available to the Apple CP/M user. Every Apple CP/M system disk has its own I/O Configuration Block, which is loaded and initialized when the system is booted.

There are five primary functions of the I/O Configuration Block:

1. Console cursor addressing/screen function interface
2. Redefinition of keyboard characters
3. Support of non-standard peripheral devices and I/O software
4. Calling of 6502 subroutines
5. Indication of the presence and location of peripheral cards

Each is detailed in its own section in the following pages.

Note: The CONFIGIO program is used to examine and modify the I/O Configuration Block – See Part 5, "Software Utilities Manual" for more information.

Console Cursor Addressing/Screen Control

Most popular video terminals, including the normal 24×40 Apple screen, can support special features such as direct cursor addressing, screen clear, highlighted text, etc. Apple CP/M applications software such as word processors and business software can easily take advantage of these features.

These advanced screen functions are usually initiated by sending a certain sequence of characters to the terminal. The sequences required to perform a specific screen function are often different for different terminals. Most applications software designed to take advantage of these screen functions can be configured for a number of popular terminals. However, if your terminal is NOT compatible with your software, you must usually write some specialized machine language subroutines to take care of the problem. Since the Datamedia terminal screen function sequences supported by Apple PASCAL and the popular 24×80 plug-in video boards are not considered "popular" by many CP/M applications programmers, they are rarely supported.

Under Apple CP/M, these problems are solved in most cases by translating the functions as they are received, into the corresponding function expected by the terminal hardware. This is achieved by two translation tables: the Software Screen Function Table and the Hardware Screen Function Table, both part of the I/O Configuration Block. Apple CP/M uses the Software Screen Function Table to recognize an incoming screen function sequence, which is then translated to the corresponding sequence found in the Hardware Screen Function Table. This sequence is then sent to the terminal device.

For example: Suppose that you want to use a CP/M screen-oriented word processor (designed to work with a SOROC IQ 120 terminal) with a Videx Videoterm 24×80 video board. The problem: Since the Videoterm board recognizes only the Datamedia type terminal character sequences, it does not recognize the screen function character sequences (meant for the SOROC) that the word processor sends.

To solve this problem, you would use the CONFIGIO utility (see the Software Utilities Manual) to encode the SOROC screen function sequences into the Software Screen Function Table and encode the Datamedia sequences into the Hardware Table. Now when your word processor sends characters to the terminal, they are compared to the SOROC function sequences that have been placed in the Software Screen Function Table. A match means that your word processor is attempting to perform a screen function. Next, the corresponding Datamedia character sequence is taken from the Hardware Screen Function Table and sent to the terminal, where the function is actually performed.

The Hardware/Software Screen Function Table

There are nine screen functions supported by Apple CP/M:

1. Clear Screen
2. Clear to End of Page
3. Clear to End of Line
4. Set Normal (lowlight) Text Mode
5. Set Inverse (highlight) Text Mode
6. Home Cursor
7. Address Cursor
8. Move Cursor Up
9. Non-destructively Move Cursor Forward

The Backspace character (ASCII 8) is assumed to move the cursor backwards, and the Line Feed character (ASCII 10) is assumed to move the cursor down one line.

Screen function character sequences supported by Apple CP/M may be of two forms:

1. A single control character, or
2. Any ASCII character preceded by a single character lead-in.

Screen function sequences longer than two characters are not supported.

The internal format of each of the two 11-byte tables is identical. Below are listed the function number, the hexadecimal address and a description of each table entry.

FUNC. #	SOFTWARE	HARDWARE	DESCRIPTION
	0F396H	0F3A1H	Cursor address coordinate offset. Range: 0-127. If the high order is Ø, the X and Y coordinates are expected to be transmitted Y first, X last. If the high order bit is 1, the coordinates are sent X first, Y last.
	0F397H	0F3A2H	Lead-in character. This byte is zero if there is no lead-in.
	NOTE: The following rules apply to the screen function table entries below: If the table entry is zero, the function is not implemented. If the entry has the high order bit set, the function requires a lead-in. An entry with the high order bit clear means the function does not require a lead-in.		
1	0F398H	0F3A3H	Clear screen
2	0F399H	0F3A4H	Clear to End of Page
3	0F39AH	0F3A5H	Clear to End of Line
4	0F39BH	0F3A6H	Set Normal (low-light) Text Mode
5	0F39CH	0F3A7H	Set Inverse (high-light) Text Mode
6	0F39DH	0F3A8H	Home Cursor

7	0F39EH	0F3A9H	Address Cursor (See above)
8	0F39FH	0F3AAH	Move Cursor Up One Line
9	0F39FH	0F3AAH	Non-destructively Move Cursor Forward

The standard 24 × 40 Apple screen supports all nine functions independent of the Hardware Screen Function Table. However, if a Software Screen Function Table entry is zero, that function will be disabled.

The Hardware and Software Screen Function Tables can be examined and modified with the CONFIGIO program. Use of this program and more information concerning terminal configuration can be found in the Apple CP/M Utilities Reference Manual.

Terminal Independent Screen Functions/Cursor Addressing

Because of the general-purpose nature of the Hardware and Software Screen Function Tables, it is possible to write programs that use the information contained in these tables to perform screen functions. These programs would work with *any* terminal, as long as the Hardware Screen Function Table was set up correctly for the particular terminal. Below is a short segment of 8080 assembly language code that illustrates the use of the Screen Function Tables for terminal-independent screen programming:

```
;
;               Terminal Independent Screen I/O
;
;               This routine will execute the screen function
;               specified by E, where E contains the screen function
;               number from one to nine. If the function is not implemented,
;               the subroutine simply returns. All registers are destroyed.
;               (NK 5/80)
;
;               Equates:
;
BDOS            EQU  0005H          ;CP/M function call address
SXYOFF          EQU  0F396H         ;Software cursor address XY coord.
                                    offset
SFLDIN          EQU  0F397H         ;Software function lead-in character
SSFTAB          EQU  0F398H         ;Software screen functions
;
SCRFUN:         MVI  D,0            ;Prepare for index
                LXI  H,SSFTAB-1     ;Point to Software Screen
                                    Function table minus one
                DAD  D              ;Index to desired function char.
```

```
            MOV   A,M           ;Get the char.
            ORA   A             ;See if a Lead-in is required
            RZ                  ;If the function isn't there, quit
            JP    CONOUA        ;If pos., no
            PUSH  PSW           ;Save char.
            LDA   SFLDIN        ;Get software lead-in char.
            CALL  CONOUA        ;Output char. in A
            POP   PSW           ;Re-get char.
CONOUA:     MOV   E,A           ;Put char. in its place
CONOUE:     MVI   C,2           ;Console output function
            JMP   BDOS          ;Call CP/M BDOS at 0005H
;
;           This routine will position the cursor at the X,Y coords
;           in [HL].
;
GOTOXY:     PUSH  H             ;Save coords while we do seq.
            MVI   E,7           ;Do a Cursor Address function
            CALL  SCRFUN
            POP   H             ;Get coordinates back
            LDA   SXYOFF        ;Get software XY coordinate offset
            ORA   A             ;Set CC's on [A]
            JP    NORVS         ;Reverse coords if neg.
            MOV   E,L           :Reverse H&L
            MOV   L,H
            MOV   H,E
NORVS:      MOV   E,A           ;Save offset
            ADD   H             ;Add offset
            MOV   H,A           ;Save for later
            MOV   A,E           ;Re-get offset
            ADD   L
            PUSH  H             ;Save all this
            CALL  CONOUA        ;Output first coord.
            POP   H             ;Restore coords.
            MOV   E,H           ;Output second coordinate
            JMP   CONOUE        ;And return.
```

Notice that the screen function character sequences are determined by the *Software* Screen Function Table in the subroutines above. This is *necessary* for these subroutines to work with the normal Apple screen. Also note that a NUL entry in either Screen Function Table will disable that function on the Apple's 24 × 40 screen.

Redefinition of Keyboard Characters

Some CP/M software requires specific keys for proper operation that are normally unavailable on some keyboards. The Apple keyboard is particularly deficient in this respect. Common characters such as the left square bracket ([), and RUBOUT simply cannot be typed. This problem is solved by the Keyboard Character Redefinition Table found in the I/O Configuration Block.

The function of the Keyboard Character Redefinition table is simple: it redefines any key on the keyboard as any of the ASCII character codes. For example, Ctrl-K could be redefined as the left square bracket. Then when Ctrl-K is typed, the [character appears.

Another somewhat tricky use of Keyboard Character Redefinition is to disable BASIC program termination with Ctrl-C by redefining Ctrl-C as some other character such as NUL. Thus it would be impossible to break out of a BASIC program because it is impossible to type Ctrl-C. (It is also clear from this example that messing around with this table can cause some annoying problems.)

Keyboard redefinition takes place only during input from the TTY: and CRT: devices. (See Assigning Logical to Physical I/O devices below.)

The Keyboard Character Redefinition Table

The Keyboard Character Redefinition Table will support up to six character redefinitions. The table is located at 0F3ACH from the Z-80. Entries in the table are two bytes: the first is the ASCII value of the keyboard character to be redefined, and the second is the desired ASCII value of the character. Both bytes must have their high order bits cleared.

If there are less than six entries in the Keyboard Character Redefinition Table, the end of the table is denoted by a byte with the high order bit set.

Modifications to the Keyboard Character Redefinition Table may be made using the CONFIGIO program. See the "Software Utilities Manual."

Support of Non-Standard Peripherals and I/O Software

The I/O Information Block also provides for the support of non-standard Apple peripherals and I/O software. All of the primitive character I/O functions are vectored through the I/O Vector Table which is contained in the I/O Configuration Block. These vectors normally point to the standard I/O routine located in the CP/M BIOS, but they can be altered by the user to point to his own drivers. Three blocks of 128 bytes each are provided within

the I/O Configuration Block for user I/O driver software. Each of the three 128-byte blocks is allocated to a specific device, and thus to a specific slot, in order to prevent memory conflicts.

ADDR	ASSIGNED SLOT	ASSIGNED LOGICAL DEVICE
0F200H-0F27FH	Slot 1	LST: – line printer device
0F280H-0F300H	Slot 2	PUN: and RDR: – general purpose I/O
0F300H-0F37FH	Slot 3	TTY: – the console device

Most Apple I/O interface cards have 6502 ROM drivers on the card. The easiest way to interface these types of cards to Apple CP/M is to write Z-80 code to call the 6502 subroutines on the ROM. This should be sufficient to interface most common I/O devices to Apple CP/M. (See Calling of 6502 Subroutines below.)

If no card is installed in a particular slot, its allocated 128-byte space can be used for other purposes relating to its assigned logical device. These include lower-case-input drivers for the Apple keyboard, cassette tape interface, etc.

I/O driver subroutines are patched to CP/M by patching the appropriate I/O vector to point to the subroutine. A table of vector locations and their purposes is shown below:

VEC #	ADDR	VECTOR NAME	DESCRIPTION
1	0F380H	Console Status	Returns 0FFH in register A if a character is ready to read, 00H in register A otherwise.
2	0F382H	Console Input vector #1	Reads a character from the console into the A register with the high order bit clear.
3	0F384H	Console Input vector #2	
4	0F386H	Console Output vector #1	Sends the ASCII character in register C to the console device.
5	0F388H	Console Output vector #2	

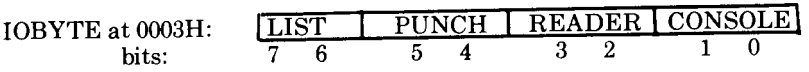
6	0F38AH	Reader Input vector #1	Reads a character from the "paper tape reader" device into register A.
7	0F38CH	Reader Input vector #2	
8	0F38EH	Punch Output vector #1	Sends the character in register C to the "paper tape punch" device.
9	0F390H	Punch Output vector #2	
10	0F392H	List Output vector #1	Sends the character in register C to the line printer device.
11	0F394H	List Output vector #2	

NOTE: During Console Output, the B register contains a number corresponding to one of the nine supported screen functions during output of a screen function. B contains zero during normal character output. B is also non-zero during the output of the Cursor Address X Y coords after executing screen function #7.

Assigning Logical to Physical I/O Devices: the IOBYTE

As explained in the CP/M reference documentation, the IOBYTE can be used to assign logical I/O devices to physical devices. The IOBYTE is changed with the STAT program. See the "CP/M Reference Manual" for more information on changing and using the IOBYTE.

The IOBYTE function creates a mapping of logical and physical devices which can be altered by CP/M programs or with the STAT utility. The mapping is performed by splitting the IOBYTE into four bit fields, as shown below:



The value in each field can be in the range 0-3. The meaning of the values that can be assigned to each field is outlined below:

CONSOLE field (bits 0,1)
0 - CONSOLE is the TTY: device
1 - CONSOLE is the CRT: device
2 - Batch mode – Uses the RDR: device as the CONSOLE input, and the LST: device as the CONSOLE output (BAT:)
3 - User defined CONSOLE device (UC1:)

READER field (bits 2,3)
0 - READER is the TTY: device
1 - READER is the CRT: device
2 - READER is the "paper tape reader" device (PTR:)
3 - User defined READER device #2 (UR2:)

PUNCH field (bits 4,5)
0 - PUNCH is the TTY: device
1 - PUNCH is the "paper tape punch" device (PTP:)
2 - User defined PUNCH #1 (UP1:)
3 - User defined PUNCH #1 (UP2:)

LIST field (bits 6,7)
0 - LIST is the TTY: device
1 - LIST is the CRT: device
2 - LIST is the line printer device (LPT:)
3 - User defined LIST device (UL1:)

Below is a description of the Apple CP/M implementation of the physical devices mentioned above:

TTY: Either the standard Apple screen and keyboard or an external terminal installed in slot 3. This routine vectors through Console Input Vector #1 and Console Output #1. The Console status is always vectored through the Console Status vector.

CRT: Same as TTY:

UC1: User defined console device. This device is vectored through Console Input #2 and Console Output #2.

PTR: A standard Apple interface capable of doing *input* installed into slot 2. If no card is plugged into slot 2, the PTR: device always returns a 1AH end-of-file character. Input from the PTR: device is vectored through Reader Input vector #1. Characters are returned in the A register.

UR1: User defined reader #1. A character read from this device is returned in the A register. This input device is vectored through Reader Input vector #2.

UR2: User defined reader #2. This device is physically the same as UR2:.

PTP: Any standard Apple interface capable of doing *output* installed into slot 2. If no card is plugged into slot 2, the PTP: device does nothing. Output to the PTP: device is vectored through Punch Output vector #1.

UP1: User defined punch #1. The character in register C is output through Reader Input vector #2.

UP2: User defined punch #2. This device is physically the same as UP1:

LPT: The LPT: device is any standard Apple interface card installed into slot 1 capable of doing output. The character in register C is output through List Output vector #1.

ULl: User defined list device. The character in register C is output via List Output vector #2.

The IOBYTE can be changed with the STAT program, or it may be modified from an assembly language program using the CP/M Get IOBYTE and Set IOBYTE (#7 & #8) functions. See "An Introduction to CP/M Features and Facilities" and the "CP/M Interface Guide" in the "CP/M Reference Manual" for more information.

Patching User Software Via the I/O Vector Table

User subroutines can be patched into the I/O Configuration Block with the CONFIGIO program. Any patches made can also be permanently saved onto a CP/M system disk as well as with CONFIGIO.

To create a code file, use ASM to write the driver software, and then use LOAD to create a COM file from the HEX file produced by ASM.

The code file loaded by CONFIGIO must be of a certain internal format. Only one code segment may be patched into the I/O Configuration Block per code file. However, as many vectors in the I/O Vector Table may be patched as desired.

Below is outlined the format of a disk code file to be loaded with CONFIGIO and patched to the I/O Configuration Block:

First byte:	No. of patches to I/O Vector Table to be made.
Next 2 bytes:	Destination address of program code.
Next 2 bytes:	Length of program code.

Repeat for each I/O vector patch to be made:

Next byte:	Vector Patch type – either 1 or 2.
If Vector Patch type = 1 :	
Next byte:	Vector number to be patched. May be from 1-11. (See vector location definitions above)
Next 2 bytes:	Address to be patched into the vector referred by the previous byte. Points into the user's code.
If Vector Patch type = 2 :	
Next byte:	Vector number to be patched. May range from 0-11. (See vector location definitions above)

Next 2 bytes:	Address in which to place the current contents of the specified vector. (May be the address field of a JMP, etc.)
Next 2 bytes:	New address to be placed in the specified vector.
Next:	The actual program code is located after the patch information above. Convention restricts the size of the program code to 128 bytes per slot-dependent block. Use the block appropriate for your application and slot use. (See above)

Below is an example of a program that could be patched into the I/O Configuration Block using CONFIGIO. While it is listed here primarily as a model for writing your own programs, it is useful in its own right with a 24x80 video card or standard Apple video and keyboard, so you may want to enter it for your own use.

Notice how OFFSET is used to allow the program to be ORGed at 0100H.

To patch this program to the I/O Configuration Block, you would:

1. Use the DDT "S" command to enter the program into memory at 100 hex.

2. Use the CP/M SAVE command to save it to disk.

3. Use CONFIGIO option #3 to load the lower case driver into the I/O Configuration Block.

4. Use CONFIGIO option #4 to save the patched I/O Configuration Block to the disk.

If you patch this lower case input routine for your own use, note the following:

This driver defaults in upper case shift lock. The forward-arrow key is used as the shift key. Hit the arrow key once to enter lower-case input mode. Now, all characters typed will be entered in lower case. To shift a letter, hit the arrow key once—don't hold it down. The next character typed will be shifted. To enter shift-lock mode, hit the arrow key twice in a row.

```
;     APPLE CP/M LOWER CASE INPUT ROUTINE
;
; This routine can be assembled using ASM and
; LOAD to produce a file that can be loaded and
; patched into the I/O Confisuration Block with
; CONFIGIO.  It is also intended to be used as
; a model for your own programs.
;
```

```
0015 =          SHFCHR  EQU     21              ;Shift key is the forward-arrow
F3B9 =          SLTTYP  EQU     0F3B9H          ;Slot types table
E000 =          KEYBD   EQU     0E000H          ;Address of Apple keyboard
                ;
0100                    ORG     0100H           ;This is so LOAD will load at 100h
F300 =          ORIGIN  EQU     0F300H          ;Real origin of program
                OFFSET  SET     ORIGIN-LWRCASE  ;Must be added to 16-bit addresses
                ;
0100 01                 DB      1               ;Make one patch
0101 00F3               DW      ORIGIN          ;Destination address of program
0103 3E00               DW      PRGEND-LWRCASE  ;Length of program
                ;
0105 02                 DB      2               ;Patch type 2
                ;
0106 02                 DB      2               ;Patch Console Input vector #1
0107 06F3               DW      OLDINP+OFFSET   ;Place to put current contents of vector
0109 00F3               DW      LWRCASE+OFFSET  ;New contents of vector
                ;
                ; Check to make sure he isn't using an external terminal:
                ;
010B 3ABBF3     LWRCASE:LDA     SLTTYP+2        ;Is there a card in slot 3?
010E FE03               CPI     3               ;Is he using a Com Card as a terminal?
0110 CA0000             JZ      0000            ;Dummy address
0111 =          OLDINP  EQU     $-2             ;Place to put normal input routine addr
                ;
                ; Get a character from the Apple keyboard:
                ;
0113 3A00E0     KBLOOP: LDA     KEYBD           ;See if char available at keyboard
0116 B7                 ORA     A               ;Set condition codes on keybd loc
0117 F208F3             JP      KBLOOP+OFFSET   ;Loop if char not available
011A 3210E0             STA     KEYBD+10H       ;Clear keyboard strobe
011D E67F               ANI     7FH             ;Mask high bit of char
011F 4F                 MOV     C,A             ;Save character in [C]
                ;
0120 0615               MVI     B,SHFCHR        ;Shift character into [B]
0122 213DF3             LXI     H,STATE+OFFSET  ;Point to shift state
0125 7E                 MOV     A,M             ;Get state.
0126 FE01               CPI     1               ;Determine state
0128 79                 MOV     A,C             ;Get typed character into [A]
0129 DA36F3             JC      STATE0+OFFSET   ;Carry set - state 0
012C CA2EF3             JZ      STATE1+OFFSET   ;State 1
                ;
                ; Here if in lower case input mode.
                ; All alphabetic characters are converted
                ; to lower case, unless the shift character is
                ; typed, which enters 'shift next character' mode
                ;
012F B8         STATE2: CMP     B               ;for shift char.
0130 CA32F3             JZ      SETONE+OFFSET   ;It was, set state = 1
0133 FE40               CPI     64              ;It wasn't, so convert all
0135 D8                 RC                      ;alphabetic chars to lower case
0136 EE20               XRI     00100000B       ;This does the conversion
0138 C9                 RET                     ;All done
                ;
                ; Here if in 'shift next character' mode, entered
                ; by typing the shift char once in lower case
                ; input mode.  If shift character is typed again,
                ; upper case shift lock mode will be entered.
                ;
0139 34         STATE1: INR     M               ;Reset state = 2 = lower case mode
013A B8                 CMP     B               ;Hit shift character?
013B C0                 RNZ                     ;Let upper case character go.
013C 35                 DCR     M               ;set state to zero: upper shift lock
013D 35         SETONE: DCR     M
013E C300F3             JMP     LWRCASE+OFFSET  ;Get another character
                ;
                ; Here if in upper case shift lock mode.
                ; Shift character must be typed once to enter lower
                ; case input mode.
                ;
```

```
0141 B8         STATE0: CMP     B               ;Did he type shift char?
0142 C0                 RNZ                     ;Not shift, return upper case char.
0143 3602               MVI     M,2             ;Set state = 2 = lower case input mode
0145 C300F3             JMP     LWRCASE+OFFSET  ;and get another character
                ;
0148 00         STATE:  DB      0               ;Shift state.  Default = upper lock
                ;
                PRGEND:
0149                    END

0100 01 00 F3 3E 00 02 02 06 F3 00 F3 3A BB F3 FE 03
0110 CA 00 00 3A 00 E0 B7 F2 08 F3 32 10 E0 E6 7F 4F
0120 06 15 21 3D F3 7E FE 01 79 DA 36 F3 CA 2E F3 B8
0130 CA 32 F3 FE 40 D8 EE 20 C9 34 B8 C0 35 35 C3 00
0140 F3 B8 C0 36 02 C3 00 F3 00
-
-
-
-
```

Calling of 6502 Subroutines

As discussed in the Hardware Details section of this manual, the 6502 processor is enabled from Z-80 mode by a *write* to the slot-dependent location 0EN00H, where N is the slot location of the SoftCard, Z-80 mode is selected from 6502 mode with a *write* to the same slot dependent location, which is addressed at $CN00 in 6502 mode. (See the 6502 / Z-80 address translation table on page 2-5). Since the SoftCard may be plugged into any unused slot except zero, the location of the SoftCard will vary from system to system.

However, when the system is booted, the location of the SoftCard is determined by CP/M and its address is stored in the I/O Configuration Block. This address is thus available to CP/M software for calling 6502 subroutines. See the "Hardware Details" section of this manual.

Calling 6502 subroutines is a simple matter. The programmer simply sets up the address of the subroutine to be called, and then does a write to the address of the SoftCard explained above. It is also possible to pass parameters to and from 6502 subroutines through the 6502 A, X, Y, and P (status) registers. The 6502 stack pointer is also available after a 6502 subroutine call. Remember that 6502 and Z-80 addresses are not equivalent – See the 6502/Z-80 Address Translation Table on page 2-30.

Z-80 ADDR	6502 ADDR	PURPOSE
0F045H	$45	6502 A register pass area
0F046H	$46	6502 Y register pass area
0F047H	$47	6502 X register pass area
0F048H	$48	6502 P (status) register pass area
0F049H	$49	Contains 6502 stack pointer on exit from subroutine
0F3DEH		Address of SoftCard held here–low byte = 0 followed by high byte of form 0ENH where N is the slot occupied by the SoftCard.

0F3D0H	Address of 6502 subroutine to be called is stored here in low-high order.
$3C0	Start address of 6502 to Z-80 mode switching routine. 6502 RESET, NMI, and BREAK vectors point here. A JMP to this address puts the 6502 on "hold" and returns to Z-80 mode.

NOTE: Locations $800-$FFF are NOT available for use by a 6502 subroutine. The Apple disk driver software and disk buffers reside here.

Special Note for Language Card Users:
When in Z-80 mode, the Language Card RAM is both read- and write-enabled. When a 6502 subroutine is called, the Apple's on-board ROM is automatically enabled, making the Apple Monitor available to the 6502 subroutine. However, the Language Card RAM is write-enabled during a 6502 call, which means that a write to any location above 6502 $D000 will write in the Language Card RAM.

A side effect of read-enabling the on-board Apple ROMs is that the Z-80 memory from 0C000H to 0EFFFH ($D000-$FFFF on 6502) cannot be *read* by the 6502 unless the appropriate Language Card addresses are accessed.

The first of the two available 4K banks for the 6502 $D000-$DFFF area is not used by Apple CP/M.

Below is a short segment of 8080 assembly language code to illustrate the use of the above addresses to call a 6502 subroutine:

```
;
; Subroutine to read the value of
; Paddle zero into register A.
; Demonstrates 6502 subroutine
; calling conventions and parameter
; passing. (NK 5/80)
;
;          Equates
Z$CPU    EQU     0F3DEH    ;Location of SoftCard stored here
A$VEC    EQU     0F3D0H    ;Addr of 6502 sub. to call goes here
A$ACC    EQU     0F045H    ;6502 A register goes here
A$XREG   EQU     0F046H    ;6502 Y register pass area
PREAD    EQU     0FB1EH    ;Apple Monitor paddle read routine
;
PDL:     XRA     A         ;Clear A register
         STA     A$XREG    ;Read paddle zero
         LXI     H,PREAD   ;Get addr of subroutine
         SHLD    A$VEC     ;And store it for 6502 caller
```

```
        LHLD    Z$CPU       ;Get SoftCard addr...
        MOV     M,A         ;Go do it! (Must be a write)
;
;       Execution resumes here after 6502 does a RTS
        LDA     A$ACC       ;A = paddle value.
        RET                 ;All done – return
```

Indication of Presence and Location of Peripheral Cards.

The Card Type Table

When Apple CP/M is booted, each of the slots of the Apple is checked to see if a standard Apple I/O card is installed. This is done by checking to see if there is ROM present in the slot-dependent memory space allocated to peripheral card driver ROMs, and then comparing two signature bytes to those of the standard Apple I/O peripheral cards.

This information is then stored in the Card Type Table, which is located in the I/O Configuration Block. There are seven bytes in the Card Type Table, each corresponding to the seven slots from 1 to 7.

The value of a table entry may range from 0 to 5. The meaning of each value is as follows:

VALUE	EXPLANATION
0	No peripheral card ROM was detected (Usually means that no card is installed in the slot)
1	A peripheral card ROM was detected, but it was of an unknown type.
2	An Apple Disk II Controller card is installed in the slot.
3	An Apple Communications Interface or CCS 7710A Serial Interface is installed in the slot.
4	An Apple High-Speed Serial Interface, Videx Videoterm, M&R Sup-R-Term or Apple Silentype printer interface is installed in the slot.
5	An Apple Parallel Printer Interface is installed in the slot.

This information can be useful to the programmer. For instance, if the third entry (slot 3 – console device) of the Card Type Table is either 3 or 4, a program can assume that the user is using an 80 column external terminal of some kind. In this way, it is possible to write software that configures itself for 40 or 80 column terminals automatically.

The Card Type Table is located at 0F3B9H. The entry for a given slot is located at 3B8H + S, where S is an integer from 1 to 7.

Disk Count Byte
The Disk Count Byte is a single byte equal to the number of disk controller cards in the system times two. This value does not reflect an odd number of disk drives (i.e., only one drive plugged into a controller card).

The Disk Count Byte is located at 0F3B8H.

To Boot a Diskette Without Powering Down
The following program will allow you to boot diskettes from CP/M without having to turn the Apple's power off. This program *is not necessary;* it simply bypasses the power-off step.

1. Use the DDT "S" command to enter the following data at 100 hex.

```
0100 0E 01 CD 05 00 21 77 C7 22 00 30 21 00 C6 22 D0
0110 F3 2A DE F3 C3 00 30
-
```

2. Type Control-C to exit DDT.

3. Type SAVE 1 BOOT.COM

The program is now saved on disk. To use it, just type BOOT and press RETURN. Wait a few seconds, then insert the disk you wish to boot. Press any key to reboot the disk. Your system will reboot exactly as if you had typed PR#6 in Applesoft or Integer BASIC.

CHAPTER 3
HARDWARE DESCRIPTION

- **Introduction**
- **Timing Scheme**
- **SoftCard Control**
- **Address Bus Interface**
- **Data Bus Interface**
- **6502 Refresh**
- **DMA Daisy Chain**
- **Interrupts**
- **SoftCard Parts List**
- **SoftCard Schematic**

This chapter describes the SoftCard itself, both physically and operationally. You won't need this information for normal use of the SoftCard; it is included here to satisfy your curiosity and in case you have an unusual application in which this information would be needed.

Introduction

The Microsoft SoftCard is a peripheral card for the Apple family of computers. The SoftCard contains the necessary hardware to interface a Z-80 microprocessor (contained on the card) to the Apple bus. This permits the direct execution of 8080 and Z-80 programs, including Digital Research's CP/M operating system and all of the programs written to execute in the CP/M software environment.

The SoftCard plugs into any Apple slot except slot zero, and will work in the Apple II, Apple II Plus, or either machine with the Apple Language System. When the Language System is used, the additional memory of the Language Card is made available for use by CP/M or any program operating under CP/M.

Timing Scheme

The Z-80 microprocessor on the SoftCard is synchronized and phase locked to the Apple clocks. This is accomplished by generating a syncopated clock for the Z-80 from the Apple clocks.

During each video refresh period (01), the seven MHz Apple clock is divided down to provide three half clock periods of 135 nsec. The first half-clock is always high, the second always low, and the third always high again. After the end of the third half clock, the signal goes low and stays low until the start of the next 01. This means that the Z-80 clock is low during all of 02 plus a small part of 01. This fourth half-cycle is typically 563 nsec long. (This time is stretched by 69 nsec at the end of each video line.) The effective Z-80 clock rate is 2.041 MHz.

Each kind of machine cycle always contains one memory access period (02). The read/write line is constructed by synchronizing the leading edge of the write transition to the SoftCard clock, thus ensuring that write will only go low during the time that the SoftCard clock is high.

Because all address transitions from the Z-80 occur when its clock is high, they all must occur during 01, when the video update accesses are occurring. Therefore, each 02 cycle has stable addresses for the entire duration of the cycle.

The clock generation is performed by U4 and parts of U1 and U9. The circuit is arranged so that it will still work if the seven MHz clock occurs just prior to the start of 01, or vice-versa. Q1 and the associated components form an analog buffer to provide the high speed switching to within a few tenths of a volt of the supply voltage.

SoftCard Control

The SoftCard is controlled by write commands to the area of memory that normally contains peripheral read-only-memory. It is important to use a write instruction to ensure that the 6502 will not perform two accesses in succession (which would prevent switching back to the 6502).

When the Apple is powered up, the Apple reset signal forces the SoftCard to the off state. The reset signal is synchronized to the Apple clocks to ensure that a write operation cannot be interrupted. The Z-80 is immediately placed in a wait mode, and remains there until the SoftCard is activated.

Upon receipt of a write to the proper area of memory, the SoftCard is activated, and the red LED is turned on. The Z-80 remains in a wait mode until one memory cycle occurs with SoftCard address information. At this point, the Z-80 is released from the wait mode and allowed to run with no further wait cycles required.

Receipt of another write to the same area of memory (this time from the SoftCard itself) will de-activate the SoftCard.

The table below shows the memory addresses used to control the SoftCard as a function of slot location:

SLOT	CONTROL ADDRESSES
1	$C100-$C1FF
2	$C200-$C2FF
3	$C300-$C3FF
4	$C400-$C4FF
5	$C500-$C5FF
6	$C600-$C6FF
7	$C700-$C7FF

Address Bus Interface

The SoftCard address bus is interfaced to the Apple I/O bus through a bank translation circuit. This circuit, consisting of U7, U8, U11, and half of U12, resolves the memory address conflicts that exist between the 6502

architecture and the conventions used by both CP/M and the Z-80 microprocessor. When enabled by S1-1 turned off, the translator adds $1000 to all addresses. This effectively shifts the Z-80 interrupt addresses and CP/M starting addresses out of the 6502 zero page of memory. In addition, addresses in the range of $C000-$EFFF are shifted to allow apparent contiguous memory for CP/M. The table below shows exactly how the translator functions:

Z-80 ADDRESS	APPLE ADDRESS
$0000-$0FFF	$1000-$1FFF
$1000-$1FFF	$2000-$2FFF
$2000-$2FFF	$3000-$3FFF
$3000-$3FFF	$4000-$4FFF
$4000-$4FFF	$5000-$5FFF
$5000-$5FFF	$6000-$6FFF
$6000-$6FFF	$7000-$7FFF
$7000-$7FFF	$8000-$8FFF
$8000-$8FFF	$9000-$9FFF
$9000-$9FFF	$A000-$AFFF
$A000-$AFFF	$B000-$BFFF
$B000-$BFFF	$D000-$DFFF
$C000-$CFFF	$E000-$EFFF
$D000-$DFFF	$F000-$FFFF
$E000-$EFFF	$C000-$CFFF
$F000-$FFFF	$0000-$0FFF

Notice that when the Language Card is installed, the Z-80 can address contiguous memory from $0000-$DFFF, without accessing the 6502 zero page of memory or the Apple peripheral area.

When the translator is disabled (S1-1 turned on) addresses presented by the Z-80 are buffered and appear at the Apple I/O bus unchanged.

All of the address buffers are tri-state buffers capable of sinking or sourcing 24 mA of current. All of the buffers are turned off whenever the SoftCard relinquishes control of the bus. The timing at turn-on and turn-off is arranged to prevent the SoftCard buffers from driving the address bus when the Apple is driving the bus.

The timing of the SoftCard forces all address transitions to occur during the time that the video display (and dynamic memory) is being refreshed by the Apple. Because for each memory access the address lines are stable at the start of the cycle, no wait states are used for memory accesses.

Data Bus Interface

The data from the SoftCard to the Apple (memory writes) is buffered by the same high current driver type as used by the address bus interface. It is only enabled when the following two conditions occur:

1. The SoftCard has control of the bus
2. The SoftCard is attempting to write

When the SoftCard is reading memory, the data is buffered and latched by U15. The outputs of U15 are tri-state, and only enabled when the SoftCard is performing a read. The latch is needed to save data not latched by the Apple (such as the keyboard characters) until the Z-80 can look at it.

Because the SoftCard timing is synchronous and phase locked with the Apple, the timing signals generated by the Z-80 can be used to drive the buffers and the latch.

When an interrupt is recognized by the Z-80 (assuming they are enabled in both hardware and software) the pull-up resistors guarantee that a predictable response is generated for any of the interrupt modes of the Z-80. The byte of data read during an interrupt sequence will be $FF.

6502 Refresh

The 6502 is a dynamic microprocessor, meaning that it requires clock cycles to maintain the contents of its internal registers. The Apple DMA circuitry interrupts operation of the 6502 by turning its clock off. Occasionally, this clock must be turned back on if the 6502 is to remain ready to operate.

This is accomplished by holding the 6502 in a non-ready state (by holding the "RDY" line low) and allowing one memory fetch to be controlled by the 6502. The data fetched is not used by the 6502, and control of the bus reverts back to the SoftCard immediately after the "refresh" memory cycle.

The Z-80 dynamic refresh control lines are used to implement this function. Therefore, the 6502 "refresh" occurs immediately after an op code fetch, and is thus transparent to the SoftCard and the user. No wait cycles have to be added to any Z-80 machine cycles, because the 6502 refresh time is used by the Z-80 to decode the op code. While the 6502 has control of the bus again, the SoftCard address and data buffers are placed in the tri-state mode.

If higher priority DMA devices are allowed to interrupt operation of the SoftCard, the 6502 refresh does not continue. Therefore if it is important to retain the register contents of the 6502 during a DMA cycle, the length of the cycle must be limited to a few microseconds (less than 5).

During a normal mixture of instructions, the 6502 refresh occurs every 4-5 microseconds, well under the data sheet maximum of 40 microseconds. The longest instruction will allow 11.25 microseconds to elapse between refreshes.

DMA Daisy Chain

The Apple DMA daisy chain is fully supported, to the extent that a higher priority DMA device may cause the SoftCard to relinquish control of the bus. Switch S1-2 (when on) enables DMA requests to interrupt the SoftCard. If this switch is on, and the DMA daisy chain input (pin 27) is driven low, the Z-80 will finish the current machine cycle, then the SoftCard will give up control of the bus by raising the DMA control line on pin 22 of the I/O bus. At this time another device may assume control by lowering pin 22. Control must not commence sooner, because the SoftCard buffers will still be driving the bus.

If S1-2 is off, the daisy chain is preserved if the SoftCard is off. When the SoftCard is turned on, the daisy chain output (pin 24) indicates to lower priority devices that DMA activity is in progress. The lower priority devices are therefore locked out of doing any DMA. Likewise, the higher priority devices are also locked out because the SoftCard will not relinquish control of the bus.

Interrupts

Hardware has been included to allow interrupts to be recognized by the Z-80 on the SoftCard as well as by the 6502 microprocessor. When S1-4 is on, the Z-80 will respond to interrupts occurring in the Apple. The interrupt handler program should not attempt to service the interrupt. Instead, control should be passed back to the 6502 for the actual processing. This permits the 6502, which also sees the interrupt, to clear itself of the interrupt status.

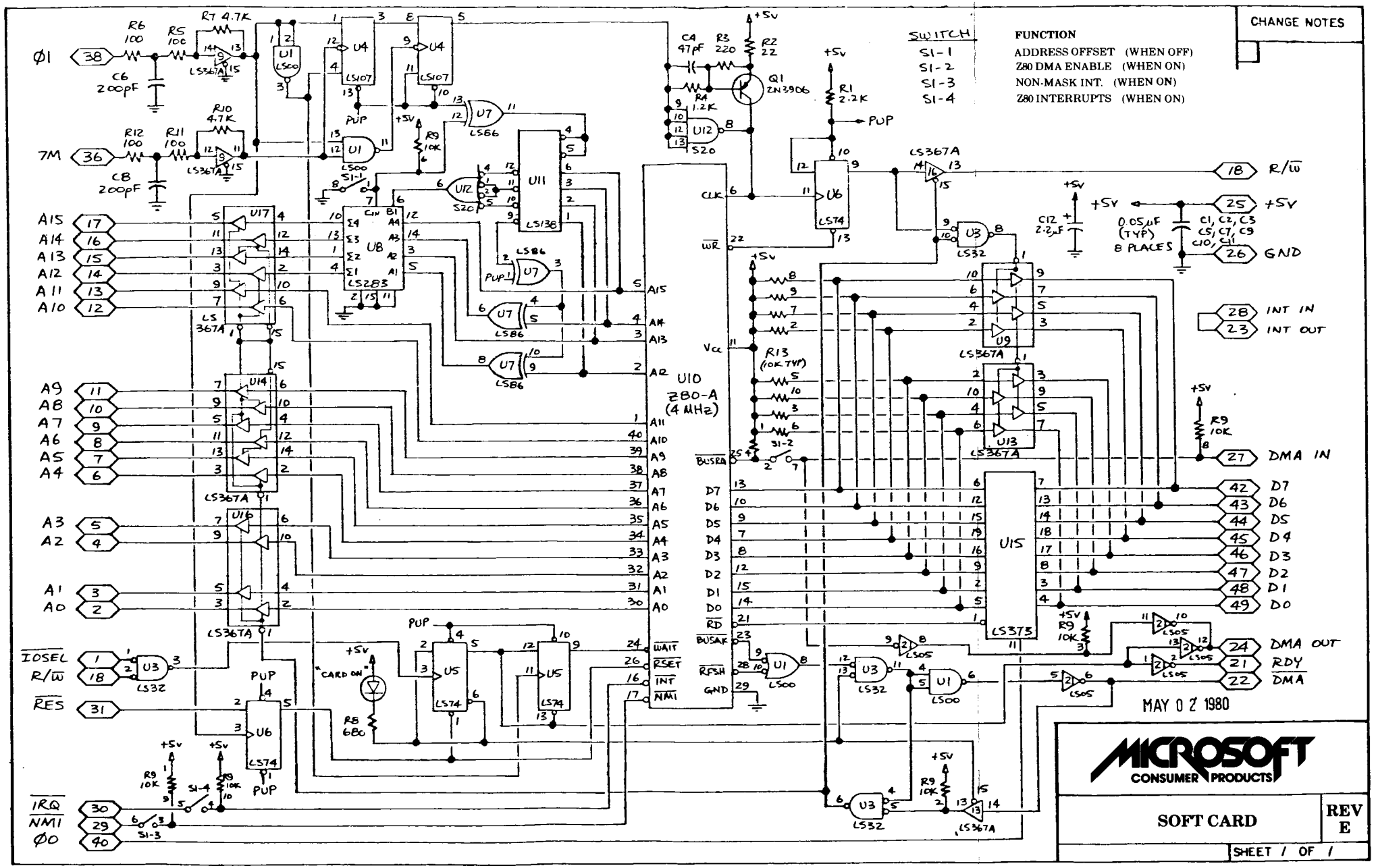
Regardless of the interrupt mode selected for the Z-80, the data byte read during the interrupt sequence will always be $FF. This may be used to vector to a particular memory location for the interrupt handling routine.

Switch S1-3 performs the same function for the non-maskable interrupt.

Parts List SoftCard

Component Identifier	Part No.	Description
U1	74LS00	Quad Nand
U2	74LS05	Hex Inverter
U3	74LS32	Quad Or

U4	74LS107	Dual Flip-Flop
U5	74LS74A	Dual Flip-Flop
U6	74LS74A	Dual Flip-Flop
U7	74LS86	Quad Ex-Or
U8	74LS283	4 Bit Adder
U9	74LS367A	Hex Buffer
U10	Z-80A	Z-80A (4 MHz)
U11	74LS138	Octal Decoder
U12	74S20 (must be "S" part)	Dual Nand
U13	74LS367A	Hex Buffer
U14	74LS367A	Hex Buffer
U15	74LS373	Octal Latch
U16	74LS367A	Hex Buffer
U17	74LS367A	Hex Buffer
Q1	2N3906	PNP Transistor
R1		2.2KΩ, 5%, ¼ watt
R2		22Ω, 5%, ¼ watt
R3		220Ω, 5%, ¼ watt
R4		1.2KΩ, 5%, ¼ watt
R5		100Ω, 5%, ¼ watt
R6		100Ω, 5%, ¼ watt
R7		4.7KΩ, 5%, ¼ watt
R8		680Ω , 5%, ¼ watt
R9		Resistor Pack, 10KΩ,
R10		4.7KΩ, 5%, ¼ watt
R11		100Ω, 5%, ¼ watt
R12		100Ω, 5%, ¼ watt
R13		Resistor Pack, 10KΩ
C1		Capacitor, 0.05 μF
C2		Capacitor, 0.05 μF
C3		Capacitor, 0.05 μF
C4		Capacitor, 47 ρF, 10%, 1000V
C5		Capacitor, 0.05 μF
C6		Capacitor, 200 ρF, 10%, 1000V
C7		Capacitor, 0.05 μF
C8		Capacitor, 200 ρF, 10%, 1000V
C9		Capacitor, 0.05 μF
C10		Capacitor, 0.05μF
C11		Capacitor, 0.05μF
C12		Capacitor, Solid Tant., 2.2 μF, 20%, 35V
"Card On"		Light Emitting Diode
S1		Dip Switch – Quad
	-1	Printed Circuit Card


CHANGE NOTES
SWITCH
SI-1
SI-2
SI-3
SI-4
FUNCTION
ADDRESS OFFSET (WHEN OFF)
Z80 DMA ENABLE (WHEN ON)
NON-MASK INT. (WHEN ON)
Z80 INTERRUPTS (WHEN ON)
U10
Z80-A
(4 MHz)
MAY 0 2 1980
MICROSOFT
CONSUMER PRODUCTS
SOFT CARD
REV
E
SHEET 1 OF 1

CP/M Reference Manual

Written by Digital Research

Digital Research, Post Office Box 579
Pacific Grove, CA 93950

PART 3: CP/M REFERENCE MANUAL

Chapter 1
Introduction to CP/M Features and Facilities

Chapter 2
CP/M 2.0 Interface Guide

Chapter 3
CP/M Editor

Chapter 4
CP/M Assembler

Chapter 5
CP/M Dynamic Debugging Tool

Copyright Notice

Disclaimer

CHAPTER 1
INTRODUCTION TO CP/M FEATURES AND FACILITIES

- Introduction
- Overview of CP/M 2.0 Facilities
- Functional Description of CP/M
- General Command Structure
- Switching Disks
- Form of Built-in Commands
  - ERAse Command
  - DIRectory Command
  - REName Command
  - SAVE Command
  - TYPE Command
  - USER Command
- Line Editing and Output Control
- Transient Commands
  - STAT
  - ASM
  - LOAD
  - DDT
  - PIP
  - ED
  - SUBMIT
  - DUMP
- BDOS Error Messages

Introduction

CP/M is a monitor control program for microcomputer system development which uses IBM-compatible flexible disks for backup storage. Using a computer mainframe based upon Intel's 8080 microcomputer, CP/M provides a general environment for program construction, storage, and editing, along with assembly and program check-out facilities. An important feature of CP/M is that it can be easily altered to execute with any computer configuration which uses an Intel 8080 (or Zilog Z-80) Central Processing Unit, and has at least 16K bytes of main memory with up to four IBM-compatible diskette drives. Although the standard Digital Research version operates on a single-density Intel MDS 800, several different hardware manufacturers support their own input-output drivers for CP/M.

The CP/M monitor provides rapid access to programs through a comprehensive file management package. The file subsystem supports a named file structure, allowing dynamic allocation of file space as well as sequential and random file access. Using this file system, a large number of distinct programs can be stored in both source and machine executable form.

CP/M also supports a powerful context editor, Intel-compatible assembler, and debugger subsystem. Optional software includes a powerful Intel-compatible macro assembler, symbolic debugger, along with various high-level languages. When coupled with CP/M's Console Command Processor, the resulting facilities equal or excel similar large computer facilities.

CP/M is logically divided into several distinct parts:

BIOS	Basic I/O System (hardware dependent)
BDOS	Basic Disk Operating System
CCP	Console Command Processor
TPA	Transient Program Area

The BIOS provides the primitive operations necessary to access the diskette drives and to interface standard peripherals (teletype, CRT, Paper Tape Reader/Punch, and user-defined peripherals), and can be tailored by the user for any particular hardware environment by "patching" this portion of CP/M.

The BDOS provides disk management by controlling one or more disk drives containing independent file directories. The BDOS implements disk allocation strategies which provide fully dynamic file construction while

minimizing head movement across the disk during access. Any particular file may contain any number of records, not exceeding the size of any single disk. In a standard CP/M system, each disk can contain up to 64 distinct files. The BDOS has entry points which include the following primitive operations which can be programmatically accessed:

SEARCH	Look for a particular disk file by name.
OPEN	Open a file for further operations.
CLOSE	Close a file after processing.
RENAME	Change the name of a particular file.
READ	Read a record from a particular file.
WRITE	Write a record onto the disk.
SELECT	Select a particular disk drive for further operations.

The CCP provides symbolic interface between the user's console and the remainder of the CP/M system. The CCP reads the console device and processes commands which include listing the file directory, printing the contents of files, and controlling the operation of transient programs, such as assemblers, editors, and debuggers. The standard commands which are available in the CCP are listed in a following section.

The last segment of CP/M is the area called the Transient Program Area (TPA). **The TPA** holds programs which are loaded from the disk under command of the CCP. During program editing, for example, the TPA holds the CP/M text editor machine code and data areas. Similarly, programs created under CP/M can be checked out by loading and executing these programs in the TPA.

It should be mentioned that any or all of the CP/M component subsystems can be "overlayed" by an executing program. That is, once a user's program is loaded into the TPA, the CCP, BDOS, and BIOS areas can be used as the program's data area. A "bootstrap" loader is programmatically accessible whenever the BIOS portion is not overlayed; thus, the user program need only branch to the bootstrap loader at the end of execution, and the complete CP/M monitor is reloaded from disk.

It should be reiterated that the CP/M operating system is partitioned into distinct modules, including the BIOS portion which defines the hardware environment in which CP/M is executing. Thus, the standard system can be

easily modified to any non-standard environment by changing the peripheral drivers to handle the custom system.

An Overview of CP/M 2.0 Facilities

CP/M 2.0 is a high-performance single-console operating system which uses table driven techniques to allow field configuration to match a wide variety of disk capacities. All of the fundamental file restrictions are removed, while maintaining upward compatibility from previous versions of release 1. Features of CP/M 2.0 include field specification of one to sixteen logical drives, each containing up to eight megabytes. Any particular file can reach the full drive size with the capability to expand to thirty-two megabytes in future releases. The directory size can be field configured to contain any reasonable number of entries, and each file is optionally tagged with read/only and system attributes. Users of CP/M 2.0 are physically separated by user numbers, with facilities for file copy operations from one user area to another. Powerful relative-record random access functions are present in CP/M 2.0 which provide direct access to any of the 65536 records of an eight megabyte file.

All disk-dependent portions of CP/M 2.0 are placed into a BIOS-resident "disk parameter block" which is either hand coded or produced automatically using the disk definition macro library provided with CP/M 2.0. The end user need only specify the maximum number of active disks, the starting and ending sector numbers, the data allocation size, the maximum extent of the logical disk, directory size information, and reserved track values. The macros use this information to generate the appropriate tables and table references for use during CP/M 2.0 operation. Deblocking information is also provided which aids in assembly or disassembly of sector sizes which are multiples of the fundamental 128 byte data unit, and the system alteration manual includes general-purpose subroutines which use this deblocking information to take advantage of larger sector sizes. Use of these subroutines, together with the table driven data access algorithms, make CP/M 2.0 truly a universal data management system.

File expansion is achieved by providing up to 512 logical file extents, where each logical extent contains 16K bytes of data. CP/M 2.0 is structured, however, so that as much as 128K bytes of data is addressed by a single physical extent (corresponding to a single directory entry), thus maintaining compatibility with previous versions while taking full advantage of directory space.

Random access facilities are present in CP/M 2.0 which allow immediate reference to any record of an eight megabyte file. Using CP/M's unique data organization, data blocks are only allocated when actually required and movement to a record position requires little search time. Sequential file access is upwardly compatible from earlier versions to the full eight

megabytes, while random access compatibility stops at 512K byte files. Due to CP/M 2.0's simpler and faster random access, application programmers are encouraged to alter their programs to take full advantage of the 2.0 facilities.

Several CP/M 2.0 modules and utilities have improvements which correspond to the enhanced file system. STAT and PIP both account for file attributes and user areas, while the CCP provides a "login" function to change from one user area to another. The CCP also formats directory displays in a more convenient manner and accounts for both CRT and hard-copy devices in its enhanced line editing functions.

Functional Description of CP/M

The user interacts with CP/M primarily through the CCP, which reads and interprets commands entered through the console. In general, the CCP addresses one of several disks which are online (the standard system addresses up to four different disk drives). These disk drives are labelled A, B, C, and D. A disk is "logged in" if the CCP is currently addressing the disk. In order to clearly indicate which disk is the currently logged disk, the CCP always prompts the operator with the disk name followed by the symbol ">" indicating that the CCP is ready for another command. Upon initial start up, the CP/M system is brought in from disk A, and the CCP displays the message

xxK CP/M VER m.m

where xx is the memory size (in kilobytes) which this CP/M system manages, and m.m is the CP/M version number. All CP/M systems are initially set to operate in a 16K memory space, but can be easily reconfigured to fit any memory size on the host system. Following system signon, CP/M automatically logs in disk A, prompts the user with the symbol "A>" (indicating that CP/M is currently addressing disk "A"), and waits for a command. The commands are implemented at two levels: built-in commands and transient commands.

General Command Structure

Built-in commands are a part of the CCP program itself, while transient commands are loaded into the TPA from disk and executed. The built-in commands are

ERA	Erase specified files.
DIR	Displays file names in the directory.

REN	Rename the specified file.
SAVE	Save memory contents in a file.
TYPE	Type the contents of a file on the logged disk.
USER	Move to another area within the same directory.

Nearly all of the commands reference a particular file or group of files. The form of a file reference is specified below.

File References

A file reference identifies a particular file or group of files on a particular disk attached to CP/M. These file references can be either "unambiguous" (ufn) or "ambiguous" (afn). An unambiguous file reference uniquely identifies a single file, while an ambiguous file reference may be satisfied by a number of different files.

File references consist of two parts: the primary name and the secondary name. Although the secondary name is optional, it usually is generic; that is, the secondary name "ASM," for example, is used to denote that the file is an assembly language source file, while the primary name distinguishes each particular source file. The two names are separated by a "." as shown below:

pppppppp.sss

where pppppppp represents the primary name of eight characters or less, and sss is the secondary name of no more than three characters. As mentioned above, the name

pppppppp

is also allowed and is equivalent to a secondary name consisting of three blanks. The characters used in specifying an unambiguous file reference cannot contain any of the special characters

< > . , ; : = ? * [ ]

while all alphanumerics and remaining special characters are allowed.

An ambiguous file reference is used for directory search and pattern matching. The form of an ambiguous file reference is similar to an unambiguous reference, except the symbol "?" may be interspersed throughout the primary and secondary names. In various commands throughout CP/M, the "?" symbol matches any character of a file name in the "?" position. Thus, the ambiguous reference

X?Z.C?M

is satisfied by the unambiguous file names

XYZ.COM

and

X3Z.CAM

Note that the ambiguous reference

.

is equivalent to the ambiguous file reference

????????.???

while

pppppppp.*

and

*.sss

are abbreviations for

pppppppp.???

and

????????.sss

respectively. As an example,

DIR *.*

is interpreted by the CCP as a command to list the names of all disk files in the directory, while

DIR X.Y

searches only for a file by the name X.Y. Similarly, the command

DIR X?Y.C?M

causes a search for all (unambiguous) file names on the disk which satisfy this ambiguous reference.

The following file names are valid unambiguous file references:

```
X          XYZ        GAMMA
X.Y        XYZ.COM    GAMMA.1
```

As an added convenience, the programmer can generally specify the disk drive name along with the file name. In this case, the drive name is given as a letter A through Z followed by a colon (:). The specified drive is then "logged in" before the file operation occurs. Thus, the following are valid file names with disk name prefixes:

```
A:X.Y      B:XYZ      C:GAMMA
Z:XYZ.COM  B:X.A?M    C:*.ASM
```

It should also be noted that all alphabetic lower case letters in file and drive names are always translated to upper case when they are processed by the CCP.

Switching Disks

The operator can switch the currently logged disk by typing the disk drive name (A, B, C, or D) followed by a colon (:) when the CCP is waiting for console input. Thus, the sequence of prompts and commands shown below might occur after the CP/M system is loaded from disk A:

```
16K CP/M VER 1.4
A>DIR                  List all files on disk A.
SAMPLE   ASM
SAMPLE   PRN
A>B:                   Switch to disk B.
B>Dir *.ASM            List all "ASM" files on B.
DUMP     ASM
FILES    ASM
B>A:                   Switch back to A.
```

Form of Built-In Commands

The file and device reference forms described above can now be used to fully specify the structure of the built-in commands. In the description below, assume the following abbreviations:

ufn	unambiguous file reference
afn	ambiguous file reference
cr	carriage return

Further, recall that the CCP always translates lower case characters to

upper case characters internally. Thus, lower case alphabetics are treated as if they are upper case in command names and file references.

ERAse Command

ERA afn

The ERA (erase) command removes files from the currently logged-in disk (i.e., the disk name currently prompted by CP/M preceding the ">"). The files which are erased are those which satisfy the ambiguous file reference afn. The following examples illustrate the use of ERA:

ERA X.Y	The file named X.Y on the currently logged disk is removed from the disk directory, and the space is returned.
ERA X.*	All files with primary name X are removed from the current disk.
ERA *.ASM	All files with secondary name ASM are removed from the current disk.
ERA X?Y.C?M	All files on the current disk which satisfy the ambiguous reference X?Y.C?M are deleted.
ERA *.*	Erase all files in the current user's directory. (See USER n, page 13.) The CCP prompts with the message ALL (Y/N)? which requires a Y response before files are actually removed.
ERA B:*.PRN	All files on drive B which satisfy the ambiguous reference ????????.PRN are deleted, independently of the currently logged disk.

DIRectory Command

DIR afn

The DIR (directory) command causes the names of all files which satisfy the ambiguous file name afn to be listed at the console device. As a special case, the command

DIR

lists the files on the currently logged disk (the command "DIR" is equivalent to the command "DIR *.*"). Valid DIR commands are shown below.

DIR X.Y

DIR X?Z.C?M

DIR ??.Y

Similar to other CCP commands, the afn can be preceded by a drive name. The following DIR commands cause the selected drive to be addressed before the directory search takes place.

DIR B:

DIR B:X.Y

DIR B:*.A?M

If no files can be found on the selected diskette which satisfy the directory request, then the message "NOT FOUND" is typed at the console.

RENName Command

REN ufn2 = ufn1

The REN (rename) command allows the user to change the names of files on disk. The file satisfying ufn2 is changed to ufn1. The currently logged disk is assumed to contain the file to rename (ufn1). The CCP also allows the user to type a left-directed arrow instead of the equal sign, if the user's console supports this graphic character. Examples of the REN command are

REN X.Y=Q.R — The file Q.R is changed to X.Y.

REN XYZ.COM=XYZ.XXX — The file XYZ.XXX is changed to XYZ.COM.

The operator can precede either ufn1 or ufn2 (or both) by an optional drive address. Given that ufn1 is preceded by a drive name, then ufn2 is assumed to exist on the same drive as ufn1. Similarly, if ufn2 is preceded by a drive name, then ufn1 is assumed to reside on that drive as well. If both ufn1 and ufn2 are preceded by drive names, then the same drive must be specified in both cases. The following REN commands illustrate this format.

REN A:X.ASM = Y.ASM — The file Y.ASM is changed to X.ASM on drive A.

REN B:ZAP.BAS=ZOT.BAS — The file ZOT.BAS is changed to ZAP.BAS on drive B.

REN B:A.ASM = B:A.BAK	The file A.BAK is renamed to A.ASM on drive B.

If the file ufn1 is already present, the REN command will respond with the error "FILE EXISTS" and not perform the change. If ufn2 does not exist on the specified diskette, then the message "NOT FOUND" is printed at the console.

SAVE Command

SAVE n ufn

The SAVE command places n pages (256-byte blocks) onto disk from the TPA and names this file ufn. In the CP/M distribution system, the TPA starts at 100H (hexadecimal), which is the second page of memory. Thus, if the user's program occupies the area from 100H through 2FFH, the SAVE command must specify two pages of memory. The machine code file can be subsequently loaded and executed. Examples are:

SAVE 3 X.COM	Copies 100H through 3FFH to X.COM.
SAVE 40 Q	Copies 100H through 28FFH to Q (note that 28 is the page count in 28FFH, and that 28H = 2*16+8 = 40 decimal).
SAVE 4 X.Y	Copies 100H through 4FFH to X.Y.

The SAVE command can also specify a disk drive in the afn portion of the command, as shown below.

SAVE 10 B:ZOT.COM	Copies 10 pages (100H through 0AFFH) to the file ZOT.COM on drive B.

The SAVE operation can be used any number of times without altering the memory image.

TYPE Command

TYPE ufn

The TYPE command displays the contents of the ASCII source file ufn on the currently logged disk at the console device. Valid TYPE commands are

TYPE X.Y

TYPE X.PLM

TYPE XXX

The TYPE command expands tabs (clt-I characters), assuming tab positions are set at every eighth column. The ufn can also reference a drive name as shown below.

TYPE	B:X.PRN	The file X.PRN from drive B is displayed.

USER Command

USER n

Where n is an integer value in the range 0 to 15.

Upon cold start, the operator is automatically "logged" into user area number 0. The operator may issue the USER command at any time to move to another logical area within the same directory.

Drives which are logged in while addressing one user number are automatically active when the operator moves to another user number since a user number is simply a prefix which accesses particular directory entries on the active disks.

The active user number is maintained until changed by a subsequent USER command, or until a cold start operation when user 0 is again assumed.

Line Editing and Output Control

The CCP allows certain line editing functions while typing command lines. "Control" indicates that the Control key and the indicated key are to be pressed simultaneously. CCP commands can generally be up to 255 characters in length; they are not acted upon until the carriage return key is pressed.

rubout/delete	Remove and echo last character typed
Control C	Reboot CP/M when at beginning of line
Control E	Physical end of line: carriage is returned, but line is not sent until the carriage return key is depressed.

Control H	Backspace one character position. Produces the backspace overwrite function. Can be changed internally to another character, such as delete, through a simple single byte change.
Control J	Line feed. Terminates current input.
Control M	Carriage return. Terminates input.
Control R	Retype current command line after new line.
Control X	Backspace to beginning of current line.

The line editor keeps track of the current prompt column position so that the operator can properly align data input following a Control R or Control X command.

The control functions Control P and Control S affect console output as shown below.

Control P	Copy all subsequent console output to the currently assigned list device (see the STAT command). Output is sent to both the list device and the console device until the next Control P is typed.
Control S	Stop the console output temporarily. Program execution and output continue when the next character is typed at the console (e.g., another Control S). This feature is used to stop output on high speed consoles, such as CRT's, in order to view a segment of output before continuing.

Transient Commands

Transient commands are loaded from the currently logged disk and executed in the TPA. The transient commands defined for execution under the CCP are shown below. Additional functions can easily be defined by the user (see the LOAD command definition).

STAT	List the number of bytes of storage remaining on the currently logged disk, provide statistical information about particular files, and display or alter device assignment.
ASM	Load the CP/M assembler and assemble the specified program from disk.

LOAD	Load the file in Intel "hex" machine code format and produce a file in machine executable form which can be loaded into the TPA (this loaded program becomes a new command under the CCP).
DDT	Load the CP/M debugger into TPA and start execution.
PIP	Load the Peripheral Interchange Program for subsequent disk file and peripheral transfer operations.
ED	Load and execute the CP/M text editor program.
SUBMIT	Submit a file of commands for batch processing.
DUMP	Dump the contents of a file in hex.

Transient commands are specified in the same manner as built-in commands, and additional commands can be easily defined by the user. As an added convenience, the transient command can be preceded by a drive name, which causes the transient to be loaded from the specified drive into the TPA for execution. Thus, the command

B:STAT

causes CP/M to temporarily "log in" drive B for the source of the STAT transient, and then return to the original logged disk for subsequent processing.

The basic transient commands are listed in detail below.

STAT

The STAT command provides general statistical information about file storage and device assignment. It is initiated by typing one of the following forms:

STAT
STAT "command line"

Special forms of the "command line" allow the current device assignment to be examined and altered as well. The various command lines which can be specified are shown below, with an explanation of each form shown to the right.

STAT ⟨cr⟩

If the user types an empty command line, the STAT transient calculates the storage remaining on all active drives, and prints a message

```
    x:  R/W,  SPACE:  nnnK
or
    x:  R/O,  SPACE:  nnnK
```

for each active drive x, where R/W indicates the drive may be read or written, and R/O indicates the drive is read only (a drive becomes R/O by explicitly setting it to read only, as shown below, or by inadvertently changing diskettes without performing a warm start). The space remaining on the diskette in drive x is given in kilobytes by nnn.

STAT x: ⟨cr⟩

If a drive name is given, then the drive is selected before the storage is computed. Thus, the command "STAT B:" could be issued while logged into drive A, resulting in the message

```
BYTES REMAINING ON B: nnnK
```

STAT afn ⟨cr⟩

The command line can also specify a set of files to be scanned by STAT. The files which satisfy afn are listed in alphabetical order, with storage requirements for each file under the heading

```
RECS  BYTS  EX    D:FILENAME.TYP
rrrr  bbbK  ee    d:pppppppp.sss
```

where rrrr is the number of 128-byte records allocated to the file, bbb is the number of kilobytes allocated to the file (bbb = rrrr*128/1024), ee is the number of 16K extensions (ee = bbb/16), d is the drive name containing the file (A...Z), pppppppp is the (up to) eight-character primary file name, and sss is the (up to) three-character secondary name. After listing the individual files, the storage usage is summarized.

STAT x:afn ⟨cr⟩

As a convenience, the drive name can be given ahead of the afn. In this case, the specified drive is first selected, and the form "STAT afn" is executed.

STAT d:filename.typ $S ⟨cr⟩

("d:" is optional drive name and "filename.typ" is an unambiguous or ambiguous file name)

Produces the output display format:

```
 Size  Recs Bytes  Ext  Acc
   48    48   6K    1   R/O A:ED.COM
   55    55  12K    1   R/O (A:PIP.COM)
65536  128   2K    2   R/W A:X.DAT
```

The $S parameter causes the "Size" field to be displayed. (The command may be used without the $S if desired.) The Size field lists the virtual file size in records, while the "Recs" field sums the number of virtual records in each extent. For files constructed sequentially, the Size and Recs fields are identical. The "Bytes" field lists the actual number of bytes allocated to the corresponding file. The minimum allocation unit is determined at configuration time, and thus the number of bytes corresponds to the record count plus the remaining unused space in the last allocated block for sequential files. Random access files are given data areas only when written, so the Bytes field contains the only accurate allocation figure. In the case of random access, the Size field gives the logical end-of-file record position and the Recs field counts the logical records of each extent (each of these extents, however, may contain unallocated "holes" even though they are added into the record count). The "Ext" field counts the number of local 16K extents allocated to the file. The "Acc" field gives the R/O or R/W access mode, which is changed using the commands shown below. The parentheses shown around the PIP.COM file name indicate that it has the "system" indicator set, so that it will not be listed in DIR commands.

STAT d:filename.typ $R/O ⟨cr⟩

Places the file or set of files in a read-only status until changed by a subsequent STAT command. The R/O status is recorded in the directory with the file so that it remains R/O through intervening cold start operations. When a file is marked R/O, attempts to erase or write into the file result in a terminal BDOS message: Bdos Err on D: File R/O.

STAT d:filename.typ $R/W ⟨cr⟩

Places the file in a permanent read/write status.

STAT d:filename.typ $SYS ⟨cr⟩

Attaches the system indicator to the file.

STAT d:filename.typ $DIR ⟨cr⟩

Removes the system indicator from the file.

STAT d:DSK: ⟨cr⟩

Lists the drive characteristics of the disk named by "d:" which is in the range A:, B:, ..., P:. The drive characteristics are listed in the format:

d:	Drive Characteristics
65536:	128 Byte Record Capacity
8192:	Kilobyte Drive Capacity
128:	32 Byte Directory Entries
0:	Checked Directory Entries
1024:	Records/Extent
128:	Records/Block
58:	Sectors/Track
2:	Reserved Tracks

The total record capacity is listed, followed by the total drive capacity listed in Kbytes. The number of checked entries is usually identical to the directory size for removable media, since this mechanism is used to detect changed media during CP/M operation without an intervening warm start. The number of records per extent determines the addressing capacity of each directory entry (1024 times 128 bytes, or 128K in the example above). The number of records per block shows the basic allocation size (in the example, 128 records/block times 128 bytes per record, or 16K bytes per block). The listing is then followed by the number of physical sectors per track and the number of reserved tracks.

STAT DSK: ⟨cr⟩

Lists drive characteristics as above for all currently active drives.

STAT USR: ⟨cr⟩

Produces a list of the user numbers which have files on the currently addressed disk. The display format is:

```
Active User : 0
Active Files: 0 1 3
```

where the first line lists the currently addressed user number, as set by the last CCP USER command, followed by a list of user numbers scanned from the current directory. In the above case, the active user number is 0 (default at cold start), with three user numbers which have

active files on the current disk. The operator can subsequently examine the directories of the other user numbers by logging in with USER 1, USER 2, or USER 3 commands, followed by a DIR command at the CCP level.

The STAT command also allows control over the physical to logical device assignment (see the IOBYTE function described in the "CP/M Interface Guide." In general, there are four logical peripheral devices which are, at any particular instant, each assigned to one of several physical peripheral devices. The four logical devices are named:

CON:	The system console device (used by CCP for communication with the operator)
RDR:	The paper tape reader device
PUN:	The paper tape punch device
LST:	The output list device

The actual devices attached to any particular computer system are driven by subroutines in the BIOS portion of CP/M. Thus, the logical RDR: device, for example, could actually be a high speed reader, Teletype reader, or cassette tape. In order to allow some flexibility in device naming and assignment, several physical devices are defined, as shown below:

TTY:	Teletype device (slow speed console)
CRT:	Cathode ray tube device (high speed console)
BAT:	Batch processing (console is current RDR:, output goes to current LST: device)
UC1:	User-defined console
PTR:	Paper tape reader (high speed reader)
UR1:	User-defined reader #1
UR2:	User-defined reader #2
PTP:	Paper tape punch (high speed punch)
UP1:	User-defined punch #1

UP2:	User-defined punch #2
LPT:	Line printer
UL1:	User-defined list device #1

It must be emphasized that the physical device names may or may not actually correspond to devices which the names imply. That is, the PTP: device may be implemented as a cassette write operation, if the user wishes. The exact correspondence and driving subroutine is defined in the BIOS portion of CP/M. In the standard distribution version of CP/M, these devices correspond to their names on the MDS 800 development system.

The command:

STAT VAL: ⟨cr⟩

produces a summary of the available status commands, resulting in the output:

Temp R/O Disk: d:=R/O

Set Indicator: d:filename.typ $R/O $R/W $SYS $DIR

Disk Status: DSK: d:DSK:

User Status: USR:

Iobyte Assign:

CON. = TTY:	CRT:	BAT:	UC1:
RDR: = TTY:	PTR:	UR1:	UR2:
PUN: = TTY:	PTP:	UP1:	UP2:
LST: = TTY:	CRT:	LPT:	UL1:

In each case, the logical device shown to the left can take any of the four physical assignments shown to the right on each line. The current logical to physical mapping is displayed by typing the command

STAT DEV: ⟨cr⟩

which produces a listing of each logical device to the left, and the current corresponding physical device to the right. For example, the list might appear as follows:

```
CON: = CRT:
RDR: = UR1:
PUN: = PTP:
LST: = TTY:
```

The current logical to physical device assignment can be changed by typing a STAT command of the form

STAT ld1 = pd1, ld2 = pd2 , ... , ldn = pdn ⟨cr⟩

where ld1 through ldn are logical device names, and pd1 through pdn are compatible physical device names (i.e., ldi and pdi appear on the same line in the "VAL:" command shown above). The following are valid STAT commands which change the current logical to physical device assignments:

```
STAT CON: = CRT: ⟨cr⟩
STAT PUN: = TTY:,LST: = LPT:, RDR: = TTY: ⟨cr⟩
```

ASM ufn

The ASM command loads and executes the CP/M 8080 assembler. The ufn specifies a source file containing assembly language statements where the secondary name is assumed to be ASM, and thus is not specified. The following ASM commands are valid:

```
ASM X

ASM GAMMA
```

The two-pass assembler is automatically executed. If assembly errors occur during the second pass, the errors are printed at the console.

The assembler produces a file

```
x.PRN
```

where x is the primary name specified in the ASM command. The PRN file contains a listing of the source program (with imbedded tab characters if present in the source program), along with the machine code generated for each statement and diagnostic error messages, if any. The PRN file can be listed at the console using the TYPE command, or sent to a peripheral device using PIP (see the PIP command structure below). Note also that the PRN file contains the original source program, augmented by miscellaneous assembly information in the leftmost 16 columns (program addresses and hexadecimal machine code, for example). Thus, the PRN file can serve as a

backup for the original source file: if the source file is accidentally removed or destroyed, the PRN file can be edited (see the ED operator's guide) by removing the leftmost 16 characters of each line (this can be done by issuing a single editor "macro" command). The resulting file is identical to the original source file and can be renamed (REN) from PRN to ASM for subsequent editing and assembly. The file

x.HEX

is also produced which contains 8080 machine language in Intel "hex" format suitable for subsequent loading and execution (see the LOAD command). For complete details of CP/M's assembly language program, see the "CP/M Assembler Language (ASM) User's Guide."

Similar to other transient commands, the source file for assembly can be taken from an alternate disk by prefixing the assembly language file name by a disk drive name. Thus, the command

ASM B:ALPHA ⟨cr⟩

loads the assembler from the currently logged drive and operates upon the source program ALPHA.ASM on drive B. The HEX and PRN files are also placed on drive B in this case.

LOAD ufn cr

The LOAD command reads the file ufn, which is assumed to contain "hex" format machine code, and produces a memory image file which can be subsequently executed. The file name ufn is assumed to be of the form

x.HEX

and thus only the name x need be specified in the command. The LOAD command creates a file named

x.COM

which marks it as containing machine executable code. The file is actually loaded into memory and executed when the user types the file name x immediately after the prompting character ">" printed by the CCP.

In general, the CCP reads the name x following the prompting character and looks for a built-in function name. If no function name is found, the CCP searches the system disk directory for a file by the name

If found, the machine code is loaded into the TPA, and the program executes. Thus, the user need only LOAD a hex file once; it can be subsequently executed any number of times by simply typing the primary name. In this way, the user can "invent" new commands in the CCP. (Initialized disks contain the transient commands as COM files, which can be deleted at the user's option.) The operation can take place on an alternate drive if the file name is prefixed by a drive name. Thus

```
LOAD B:BETA
```

brings the LOAD program into the TPA from the currently logged disk and operates upon drive B after execution begins.

It must be noted that the BETA.HEX file must contain valid Intel format hexadecimal machine code records (as produced by the ASM program, for example) which begin at 100H, the beginning of the TPA. Further, the addresses in the hex records must be in ascending order; gaps in unfilled memory regions are filled with zeroes by the LOAD command as the hex records are read. Thus, LOAD must be used only for creating CP/M standard "COM" files which operate in the TPA. Programs which occupy regions of memory other than the TPA can be loaded under DDT.

PIP

PIP is the CP/M Peripheral Interchange Program which implements the basic media conversion operations necessary to load, print, punch, copy, and combine disk files. The PIP program is initiated by typing one of the following forms

```
PIP ⟨cr⟩
PIP "command line" ⟨cr⟩
```

In both cases, PIP is loaded into the TPA and executed. In case 1, PIP reads command lines directly from the console, prompted with the "*" character, until an empty command line is typed (i.e., a single carriage return is issued by the operator). Each successive command line causes some media conversion to take place according to the rules shown below. Form 2 of the PIP command is equivalent to the first, except that the single command line given with the PIP command is automatically executed, and PIP terminates immediately with no further prompting of the console for input command lines. The form of each command line is

```
destination = source #1, source #2, ... , source #n ⟨cr⟩
```

where "destination" is the file or peripheral device to receive the data, and "source#1, ..., source#n" represents a series of one or more files or devices which are copied from left to right to the destination.

When multiple files are given in the command line (i.e., $n > 1$), the individual files are assumed to contain ASCII characters, with an assumed CP/M end-of-file character (ctl-Z) at the end of each file (see the O parameter to override this assumption). The equal symbol (=) can be replaced by a left-oriented arrow, if your console supports this ASCII character, to improve readability. Lower case ASCII alphabetics are internally translated to upper case to be consistent with CP/M file and device name conventions. Finally, the total command line length cannot exceed 255 characters (ctl-E can be used to force a physical carriage return for lines which exceed the console width).

The destination and source elements can be unambiguous references to CP/M source files, with or without a preceding disk drive name. That is, any file can be referenced with a preceding drive name (A:, B:, C:, or D:) which defines the particular drive where the file may be obtained or stored. When the drive name is not included, the currently logged disk is assumed. Further, the destination file can also appear as one or more of the source files, in which case the source file is not altered until the entire concatenation is complete. If the destination file already exists, it is removed if the command line is properly formed (it is not removed if an error condition arises). The following command lines (with explanations to the right) are valid as input to PIP:

X = Y ⟨cr⟩	Copy to file X from file Y, where X and Y are unambiguous file names; Y remains unchanged.
X = Y, Z ⟨cr⟩	Concatenate files Y and Z and copy to file X, with Y and Z unchanged.
X.ASM = Y.ASM,Z.ASM,FIN.ASM ⟨cr⟩	Create the file X.ASM from the concatenation of the Y, Z, and FIN files with type ASM.
NEW.ZOT = B:OLD.ZAP ⟨cr⟩	Move a copy of OLD.ZAP from drive B to the currently logged disk; name the file NEW.ZOT.
B:A.U. = B:B.V,A:C.W,D.X ⟨cr⟩	Concatenate file B.V from drive B with C.W from drive A and D.X. from the logged disk; create the file A.U on drive B.

For more convenient use, PIP allows abbreviated commands for transferring files between disk drives. The abbreviated forms are

PIP x: = afn ⟨cr⟩

PIP x: = y:afn ⟨cr⟩

PIP ufn = y: ⟨cr⟩

PIP x:ufn = y: ⟨cr⟩

The first form copies all files from the currently logged disk which satisfy the afn to the same file names on drive x (x = A...Z). The second form is equivalent to the first, where the source for the copy is drive y (y = A...Z). The third form is equivalent to the command "PIP ufn = y:ufn ⟨cr⟩" which copies the file given by ufn from drive y to the file ufn on drive x. The fourth form is equivalent to the third, where the source disk is explicitly given by y.

Note that the source and destination disks must be different in all of these cases. If an afn is specified, PIP lists each ufn which satisfies the afn as it is being copied. If a file exists by the same name as the destination file, it is removed upon successful completion of the copy, and replaced by the copied file.

The following PIP commands give examples of valid disk-to-disk copy operations:

B: = *.COM ⟨cr⟩	Copy all files which have the secondary name "COM" to drive B from the current drive.
A: = B:ZAP.* ⟨cr⟩	Copy all files which have the primary name "ZAP" to drive A from drive B.
ZAP.ASM = B: ⟨cr⟩	Equivalent to ZAP.ASM = B:ZAP.ASM
B:ZOT.COM = A: ⟨cr⟩	Equivalent to B:ZOT.COM = A:ZOT.COM
B: = GAMMA.BAS ⟨cr⟩	Same as B:GAMMA.BAS = GAMMA.BAS
B: = A:GAMMA.BAS ⟨cr⟩	Same as B:GAMMA.BAS = A:GAMMA.BAS

PIP also allows reference to physical and logical devices which are attached to the CP/M system. The device names are the same as given under the STAT command, along with a number of specially named devices. The logical

devices given in the STAT command are

CON: (console), RDR: (reader), PUN: (punch), and LST: (list)

while the physical devices are

TTY: (console, reader, punch, or list)		
CRT: (console, or list),		UC1: (console
PTR: (reader),	UR1: (reader),	UR2: (reader)
PTP: (punch),	UP1: (punch),	UP2: (punch)
LPT: (list),	UL1: (list)	

(Note that the "BAT:" physical device is not included, since this assignment is used only to indicate that the RDR: and LST: devices are to be used for console input/output.)

The RDR, LST, PUN, and CON devices are all defined within the BIOS portion of CP/M, and thus are easily altered for any particular I/O system. (The current physical device mapping is defined by IOBYTE; see the "CP/M Interface Guide" for a discussion of this function). The destination device must be capable of receiving data (i.e., data cannot be sent to the punch), and the source devices must be capable of generating data (i.e., the LST: device cannot be read).

The additional device names which can be used in PIP commands are

NUL: Send 40 "nulls" (ASCII 0's) to the device (this can be issued at the end of punched output).

EOF: Send a CP/M end-of-file (ASCII ctl-Z) to the destination device (sent automatically at the end of all ASCII data transfers through PIP).

INP: Special PIP input source which can be "patched" into the PIP program itself: PIP gets the input data character-by-character by CALLing location 103H, with data returned in location 109H (parity bit must be zero).

OUT: Special PIP output destination which can be patched into the PIP program: PIP CALLs location 106H with data in register C for each character to transmit. Note that locations 109H through 1FFH of the PIP memory image are not used and can be replaced by special purpose drivers using DDT (see the DDT operator's manual).

PRN: Same as LST:, except that tabs are expanded at every eighth

character position, lines are numbered, and page ejects are inserted every 60 lines, with an initial eject (same as [t8np]).

File and device names can be interspersed in the PIP commands. In each case, the specific device is read until end-of-file (ctl-Z for ASCII files, and a real end of file for non-ASCII disk files). Data from each device or file is concatenated from left to right until the last data source has been read. The destination device or file is written using the data from the source files, and an end-of-file character (ctl-Z) is appended to the result for ASCII files. Note that if the destination is a disk file, a temporary file is created ($$$ secondary name) which is changed to the actual file name only upon successful completion of the copy. Files with the extension "COM" are always assumed to be non-ASCII.

The copy operation can be aborted at any time by depressing any key on the keyboard (a rubout suffices). PIP will respond with the message "ABORTED" to indicate that the operation was not completed. Note that if any operation is aborted, or if an error occurs during processing, PIP removes any pending commands which were set up while using the SUBMIT command.

It should also be noted that PIP performs a special function if the destination is a disk file with type "HEX" (an Intel hex formatted machine code file), and the source is an external peripheral device, such as a paper tape reader. In this case, the PIP program checks to ensure that the source file contains a properly formed hex file, with legal hexadecimal values and checksum records. When an invalid input record is found, PIP reports an error message at the console and waits for corrective action. It is usually sufficient to open the reader and rerun a section of the tape (pull the tape about 20 inches). When the tape is ready for the re-read, type a single carriage return at the console, and PIP will attempt another read. If the tape position cannot be properly read, simply continue the read (by typing a return following the error message), and enter the record manually with the ED program after the disk file is constructed. For convenience, PIP allows the end-of-file to be entered from the console if the source file is a RDR: device. In this case, the PIP program reads the device and monitors the keyboard. If ctl-Z is typed at the keyboard, then the read operation is terminated normally.

Valid PIP commands are shown below.

PIP LST: = X.PRN ⟨cr⟩	Copy X.PRN to the LST device and terminate the PIP program.
PIP ⟨cr⟩	Start PIP for a sequence of commands (PIP prompts with "*").

*CON: = X.ASM,Y.ASM,Z.ASM ⟨cr⟩

Concatenate three ASM files and copy to the CON device.

*X.HEX = CON:,Y.HEX,PTR: ⟨cr⟩

Create a HEX file by reading the CON (until a ctl-Z is typed), followed by data from Y.HEX, followed by data from PTR until a ctl-Z is encountered.

*⟨cr⟩

Single carriage return stops PIP.

PIP PUN: = NUL:,X.ASM,EOF:,NUL: ⟨cr⟩

Send 40 nulls to the punch device; then copy the X.ASM file to the punch, followed by an end-of-file (ctl-Z) and 40 more null characters.

The user can also specify one or more PIP parameters, enclosed in left and right square brackets, separated by zero or more blanks. Each parameter affects the copy operation, and the enclosed list of parameters must immediately follow the affected file or device. Generally, each parameter can be followed by an optional decimal integer value (the S and Q parameters are exceptions). The valid PIP parameters are listed below.

B Block mode transfer: data is buffered by PIP until an ASCII x-off character (ctl-S) is received from the source device. This allows transfer of data to a disk file from a continuous reading device, such as a cassette reader. Upon receipt of the x-off, PIP clears the disk buffers and returns for more input data. The amount of data which can be buffered is dependent upon the memory size of the host system (PIP will issue an error message if the buffers overflow).

Dn Delete characters which extend past column n in the transfer of data to the destination from the character source. This parameter is used most often to truncate long lines which are sent to a (narrow) printer or console device.

E Echo all transfer operations to the console as they are being performed.

F Filter form feeds from the file. All imbedded form feeds are removed. The P parameter can be used simultaneously to insert new form feeds.

Gn Get file from user number n. (n is the range 0-15.) Allows one user area to receive data files from another. If the operator has issued the

USER 4 command at the CCP level, the PIP statement

```
PIP X.Y = X.Y[G2]
```

reads file X.Y from user number 2 into user area number 4. You cannot copy files into a different area than the one which is currently addressed by the USER command.

H — Hex data transfer: all data is checked for proper Intel hex file format. Non-essential characters between hex records are removed during the copy operation. The console will be prompted for corrective action in case errors occur.

I — Ignore ":00" records in the transfer of Intel hex format file (the I parameter automatically sets the H parameter).

L — Translate upper case alphabetics to lower case.

N — Add line numbers to each line transferred to the destination, starting at one, and incrementing by 1. Leading zeroes are suppressed, and the number is followed by a colon. If N2 is specified, then leading zeroes are included, and a tab is inserted following the number. The tab is expanded if T is set.

O — Object file (non-ASCII) transfer: the normal CP/M end of file is ignored.

Pn — Include page ejects at every n lines (with an initial page eject). If n = 1 or is excluded altogether, page ejects occur every 60 lines. If the F parameter is used, form feed suppression takes place before the new page ejects are inserted.

Qs↑z — Quit copying from the source device or file when the string s (terminated by ctl-Z) is encountered.

R — Read system files. Allows files with the system attribute to be included in PIP transfers. Otherwise, system files are not recognized.

Ss↑z — Start copying from the source device when the string s is encountered (terminated by ctl-Z). The S and Q parameters can be used to "abstract" a particular section of a file (such as a subroutine). The start and quit strings are always included in the copy operation.

NOTE – the strings following the s and q parameters are translated to upper case by the CCP if form (2) of the PIP command is used. Form (1) of the PIP invocation, however, does not perform the

automatic upper case translation.

(1) PIP ⟨cr⟩
(2) PIP "command line" ⟨cr⟩

Tn Expand tabs (ctl-I characters) to every nth column during the transfer of characters to the destination from the source.

U Translate lower case alphabetics to upper case during the copy operation.

V Verify that data has been copied correctly by rereading after the write operation (the destination must be a disk file).

W Write over R/O files without console interrogation. Under normal operation, PIP will not automatically overwrite a file which is set to a permanent R/O status. It advises the user of the R/O status and waits for overwrite approval. W allows the user to bypass this interrogation process.

Z Zero the parity bit on input for each ASCII character.

The following are valid PIP commands which specify parameters in the file transfer:

PIP X.ASM = B:[v] ⟨cr⟩ Copy X.ASM from drive B to the current drive and verify that the data was properly copied.

PIP LPT: = X.ASM[nt8u] ⟨cr⟩
Copy X.ASM to the LPT: device; number each line, expand tabs to every eighth column, and translate lower case alphabetics to upper case.

PIP PUN: = X.HEX[i],Y.ZOT[h] ⟨cr⟩
First copy X.HEX to the PUN: device and ignore the trailing ":00" record in X.HEX; then continue the transfer of data by reading Y.ZOT, which contains hex records, including any ":00" records which it contains.

PIP X.LIB = Y.ASM [sSUBR1:↑z qJMP L3↑z] ⟨cr⟩
Copy from the file Y.ASM into the file X.LIB. Start the copy when the string "SUBR1:" has been found, and quit copying after the string "JMP L3" is encountered.

PIP PRN: = X.ASM[p50]	Send X.ASM to the LST: device, with line numbers, tabs expanded to every eighth column, and page ejects at every 50th line. Note that nt8p60 is the assumed parameter list for a PRN file; p50 overrides the default value.

Note that the PIP program itself is initially copied to a user area (so that subsequent files can be copied) using the SAVE command. The sequence of operations shown below effectively moves PIP from one user area to the next.

```
USER 0                  login user 0
DDT PIP.COM             load PIP in memory
(note PIP size s)
G0                      return to CCP
USER 3                  login user 3
SAVE s PIP.com
```

where s is the integral number of memory "pages" (256 byte segments) occupied by PIP. The number s can be determined when PIP.COM is located under DDT, by referring to the value under the "NEXT" display. If for example, the next available address is 1D00, then PIP.COM requires 1C hexadecimal pages (or 1 times 16 + 12 = 28 pages), and thus the value of s is 28 in the subsequent save. Once PIP is copied in this manner, it can then be copied to another disk belonging to the same user number through normal PIP transfers.

ED

The ED program is the CP/M system context editor, which allows creation and alteration of ASCII files in the CP/M environment. Complete details of operation are given in Chapter 3 CP/M ED. In general, ED allows the operator to create and operate upon source files which are organized as a sequence of ASCII characters, separated by end-of-line characters (a carriage-return line-feed sequence). There is no practical restriction on line length (no single line can exceed the size of the working memory), which is instead defined by the number of characters typed between ⟨cr⟩'s. The ED program has a number of commands for character string searching, replacement, and insertion, which are useful in the creation and correction of programs or text files under CP/M. Although the CP/M has a limited memory work space area (approximately 5000 characters in a 16K CP/M system), the file size which can be edited is not limited, since data is easily "paged" through this work area.

Upon initiation, ED creates the specified source file, if it does not exist, and opens the file for access. The programmer then "appends" data from the

source file into the work area, if the source file already exists (see the A command), for editing. The appended data can then be displayed, altered, and written from the work area back to the disk (see the W command). Particular points in the program can be automatically paged and located by context (see the N command), allowing easy access to particular portions of a large file.

Given that the operator has typed

ED X.ASM ⟨cr⟩

the ED program creates an intermediate work file with the name

X.$$$

to hold the edited data during the ED run. Upon completion of ED, the X.ASM file (original file) is renamed to X.BAK, and the edited work file is renamed to X.ASM. Thus, the X.BAK file contains the original (unedited) file, and the X.ASM file contains the newly edited file. The operator can always return to the previous version of a file by removing the most recent version, and renaming the previous version. Suppose, for example, that the current X.ASM file was improperly edited; the sequence of CCP commands shown below would reclaim the backup file.

DIR X.*	Check to see that BAK file is available.
ERA X.ASM	Erase most recent version.
REN X.ASM = X.BAK	Rename the BAK file to ASM.

Note that the operator can abort the edit at any point (reboot, power failure, ctl-C, or Q command) without destroying the original file. In this case, the BAK file is not created, and the original file is always intact.

The ED program also allows the user to "ping-pong" the source and create backup files between two disks. The form of the ED command in this case is

ED ufn d:

where ufn is the name of a file to edit on the currently logged disk and d is the name of an alternate drive. The ED program reads and processes the source file, and writes the new file to drive d, using the name ufn. Upon completion of processing, the original file becomes the backup file. Thus, if the operator is addressing disk A, the following command is valid:

ED X.ASM B:

which edits the file X.ASM on drive A, creating the new file X.$$$ on drive B. Upon completion of a successful edit, A:X.ASM is renamed to A:X.BAK, and B:X.$$$ is renamed to B:X.ASM. For user convenience, the currently logged disk becomes drive B at the end of the edit. Note that if a file by the name B:X.ASM exists before the editing begins, the message

FILE EXISTS

is printed at the console as a precaution against accidentally destroying a source file. In this case, the operator must first ERAse the existing file and then restart the edit operation.

Similar to other transient commands, editing can take place on a drive different from the currently logged disk by preceding the source file name by a drive name. Examples of valid edit requests are shown below

ED A:X.ASM	Edit the file X.ASM on drive A, with new file and backup on drive A.
ED B:X.ASM A:	Edit the file X.ASM on drive B to the temporary file X.$$$ on drive A. On termination of editing, change X.ASM on drive B to X.BAK, and change X.$$$ on drive A to X.ASM.

ED takes file attributes into account. If the operator attempts to edit a read/only file, the message

FILE IS READ/ONLY

appears at the console. The file can be loaded and examined, but cannot be altered in any way. Normally the operator simply ends the edit session, and uses STAT to change the file attribute to R/W. If the edited file has the system attribute set, the message

"SYSTEM" FILE NOT ACCESSIBLE

is displayed at the console, and the edit session is aborted. Again, the STAT program can be used to change the system attribute if desired.

SUBMIT

The SUBMIT command allows CP/M commands to be batched together for

automatic processing. The format of SUBMIT is: SUBMIT ufn parm #1...parm #n⟨cr⟩.

The ufn given in the SUBMIT command must be the filename of a file which exists on the currently logged disk, with an assumed file type of "SUB." The SUB file contains CP/M prototype commands, with possible parameter substitution. The actual parameters parm #1 ... parm #n are substituted into the prototype commands, and, if no errors occur, the file of substituted commands is processed sequentially by CP/M.

The prototype command file is created using the ED program, with interspersed "$" parameters of the form

$1 $2 $3 ... $n

corresponding to the number of actual parameters which will be included when the file is submitted for execution. When the SUBMIT transient is executed, the actual parameters parm #1 ... parm #n are paired with the formal parameters $1 ...$n in the prototype commands. If the number of formal and actual parameters does not correspond, then the submit function is aborted with an error message at the console. The SUBMIT function creates a file of substituted commands with the name

$$$.SUB

on the logged disk. When the system reboots (at the termination of the SUBMIT), this command file is read by the CCP as a source of input, rather than the console. If the SUBMIT function is performed on any disk other than drive A, the commands are not processed until the disk is inserted into drive A and the system reboots. Further, the user can abort command processing at any time by typing a rubout when the command is read and echoed. In this case, the $$$.SUB file is removed, and the subsequent commands come from the console. Command processing is also aborted if the CCP detects an error in any of the commands. Programs which execute under CP/M can abort processing of command files when error conditions occur by simply erasing any existing $$$.SUB file.

In order to introduce dollar signs into a SUBMIT file, the user may type a "$$" which reduces to a single "$" within the command file. Further, an up-arrow symbol "↑" may precede an alphabetic character x, which produces a single ctl-x character within the file.

The last command in a SUB file can initiate another SUB file, thus allowing chained batch commands.

Suppose the file ASMBL.SUB exists on disk and contains the prototype

commands

```
ASM $1
DIR $1.*
ERA *.BAK
PIP $2:=$1.PRN
ERA $1.PRN
```

and the command

```
SUBMIT ASMBL X PRN ⟨cr⟩
```

is issued by the operator. The SUBMIT program reads the ASMBL.SUB file, substituting "X" for all occurrences of $1 and "PRN" for all occurrences of $2, resulting in a $$$.SUB file containing the commands

```
ASM X
DIR X.*
ERA *.BAK
PIP PRN:=X.PRN
ERA X.PRN
```

which are executed in sequence by the CCP.

The SUBMIT function can access a SUB file which is on an alternate drive by preceding the file name by a drive name. Submitted files are only acted upon, however, when they appear on drive A. Thus, it is possible to create a submitted file on drive B which is executed at a later time when it is inserted in drive A.

XSUB

XSUB extends the power of the SUBMIT facility to include character input during program execution as well as entering command lines. The XSUB command is included as the first line of your submit file and, when executed, self-relocates directly below the CCP.

All subsequent submit command lines are processed by XSUB, so that programs which read buffered console input (BDOS function 10) receive their input directly from the submit file. For example, the file SAVER.SUB could contain the submit lines:

```
XSUB
DDT
I$1.HEX
R
G0
SAVE 1 $2.COM
```

with a subsequent SUBMIT command:

```
SUBMIT SAVER X Y
```

which substitutes X for $1 and Y for $2 in the command stream. The XSUB program loads, followed by DDT which is sent the command lines "IX.HEX" "R" and "G0", thus returning to the CCP. The final command "SAVE 1 Y.COM" is processed by the CCP.

The XSUB program remains in memory, and prints the message

```
(xsub active)
```

on each warm start operation to indicate its presence. Subsequent submit command streams do not require the XSUB, unless an intervening cold start has occurred. Note that XSUB must be loaded after DESPOOL, if both are to run simultaneously.

DUMP

The DUMP program types the contents of the disk file (ufn) at the console in hexadecimal form. The file contents are listed sixteen bytes at a time, with the absolute byte address listed to the left of each line in hexadecimal. Long typeouts can be aborted by pushing the rubout key during printout. (The source listing of the DUMP program is given in the "CP/M Interface Guide" as an example of a program written for the CP/M environment.)

BDOS Error Messages

There are three error situations which the Basic Disk Operating System intercepts during file processing. When one of these conditions is detected, the BDOS prints the message:

```
BDOS ERR ON x: error
```

where x is the drive name, and "error" is one of the three error messages:

```
BAD SECTOR
SELECT
R/O
```

The "BAD SECTOR" message indicates that the disk controller electronics has detected an error condition in reading or writing the diskette. This condition is generally due to a malfunctioning disk controller, or an extremely worn diskette. If you find that your system reports this error more than once a month, you should check the state of your controller electronics, and the condition of your media. You may also encounter this condition in reading files generated by a controller produced by a different manufacturer. Even though controllers are claimed to be IBM-compatible, one often finds small differences in recording formats. The MDS-800 controller, for example, requires two bytes of one's following the data CRC byte, which is not required in the IBM format. As a result, diskettes generated by the Intel MDS can be read by almost all other IBM-compatible systems, while disk files generated on other manufacturers' equipment will produce the "BAD SECTOR" message when read by the MDS. In any case, recovery from this condition is accomplished by typing a ctl-C to reboot (this is the safest!), or a return, which simply ignores the bad sector in the file operation. Note, however, that typing a return may destroy your diskette integrity if the operation is a directory write, so make sure you have adequate backups in this case.

The "SELECT" error occurs when there is an attempt to address a drive beyond the A through D range. In this case, the value of x in the error message gives the selected drive. The system reboots following any input from the console.

The R/O (read only) message occurs when there is an attempt to write to a diskette which has been designated as read-only in a STAT command, or has been set to read-only by the BDOS. In general, the operator should reboot CP/M either by using the warm start procedure ctl-C or by performing a cold start whenever the diskettes are changed. If a changed diskette is to be read but not written, BDOS allows the diskette to be changed without the warm or cold start, but internally marks the drive as read-only. The status of the drive is subsequently changed to read/write if a warm or cold start occurs. Upon issuing this message, CP/M waits for input from the console. An automatic warm start takes place following any input.

CHAPTER 2
CP/M 2.0 INTERFACE GUIDE

- **Introduction**
- **Operating System Call Conventions**
- **Sample File-to-File Copy Program**
- **Sample File Dump Utility**
- **Sample Random Access Program**
- **System Function Summary**

The transient program may use the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the FDOS entry point at BOOT + 0005H. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB to the CP/M FDOS. The FDOS, in turn, performs the operation and returns with either a disk read completion indication or an error number indicating that the disk read was unsuccessful. The function numbers and error indicators are given below.

Operating System Call Conventions

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs.

CP/M facilities which are available for access by transient programs fall into two general categories: simple device I/O, and disk file I/O. The simple device operations include:

Read a Console Character
Write a Console Character
Read a Sequential Tape Character
Write a Sequential Tape Character
Write a List Device Character
Get or Set I/O Status
Print Console Buffer
Read Console Buffer
Interrogate Console Ready

The FDOS operations which perform disk Input/Output are

Disk System Reset
Drive Selection
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Random or Sequential Read
Random or Sequential Write
Interrogate Available Disks
Interrogate Selected Disk
Set DMA Address
Set/Reset File Indicators

As mentioned above, access to the FDOS functions is accomplished by passing a function number and information address through the primary entry point at location BOOT + 0005H. In general, the function number is passed in register C with the information address in the double byte pair DE. Single byte values are returned in register A, with double byte values returned in HL (a zero value is returned when the function number is out of range). For reasons of compatibility, register A = L and register B = H upon return in all cases. Note that the register passing conventions of CP/M agree with those of Intel's PL/M systems programming language. The list of CP/M function numbers is given below.

0 System Reset
1 Console Input
2 Console Output
3 Reader Input
4 Punch Output
5 List Output
6 Direct Console I/O
7 Get I/O Byte
8 Set I/O Byte
9 Print String
10 Read Console Buffer
11 Get Console Status
12 Return Version Number
13 Reset Disk System
14 Select Disk
15 Open File
16 Close File
17 Search for First
18 Search for Next
19 Delete File
20 Read Sequential
21 Write Sequential
22 Make File
23 Rename File
24 Return Login Vector
25 Return Current Disk
26 Set DMA Address
27 Get Addr (Alloc)
28 Write Protect Disk
29 Get R/O Vector
30 Set File Attributes
31 Get Addr (Disk Parms)
32 Set/Get User Code
33 Read Random
34 Write Random
35 Compute File Size
36 Set Random Record

(Functions 28 and 32 should be avoided in application programs to maintain upward compatibility with MP/M.)

Upon entry to a transient program, the CCP leaves the stack pointer set to an eight level stack area with the CCP return address pushed onto the stack, leaving seven levels before overflow occurs. Although this stack is usually not used by a transient program (i.e., most transients return to the CCP through a jump to location 0000H), it is sufficiently large to make CP/M system calls since the FDOS switches to a local stack at system entry. The following assembly language program segment, for example, reads characters continuously until an asterisk is encountered, at which time control returns to the CCP (assuming a standard CP/M system with BOOT + 0000H):

```
BDOS    EQU     0005H   ;STANDARD CP/M ENTRY
CONIN   EQU     1       ;CONSOLE INPUT FUNCTION
;
        ORG     0100H   ;BASE OF TPA
NEXTC:  MVI     C,CONIN ;READ NEXT CHARACTER
        CALL    BDOS    ;RETURN CHARACTER IN <A>
        CPI     '*'     ;END OF PROCESSING?
        JNZ     NEXTC   ;LOOP IF NOT
        RET             ;RETURN TO CCP
        END
```

CP/M implements a named file structure on each disk, providing a logical organization which allows any particular file to contain any number of records from completely empty, to the full capacity of the drive. Each drive is logically distinct with a disk directory and file data area. The disk file names are in three parts: the drive select code, the file name consisting of one to eight non-blank characters, and the file type consisting of zero to three non-blank characters. The file type names the generic category of a particular file, while the file name distinguishes individual files in each category. The file types listed below name a few generic categories which have been established, although they are generally arbitrary:

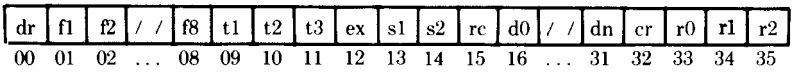
ASM	Assembler Source	PLI	PL/I Source File
PRN	Printer Listing	REL	Relocatable Module
HEX	Hex Machine Code	TEX	TEX Formatter Source
BAS	Basic Source File	BAK	ED Source Backup
INT	Intermediate Code	SYM	SID Symbol File
COM	CCP Command File	$$$	Temporary File

Source files are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (0DH followed by 0AH). Thus one 128 byte CP/M record could contain several lines of source text. The end of an ASCII file is denoted by a control-Z character (1AH) or a real end of file, returned by the CP/M read operation. Control-Z characters embedded within machine code files (e.g., COM files) are ignored, however, and the end of file condition returned by CP/M is used to terminate read operations.

Files in CP/M can be thought of as a sequence of up to 65536 records of 128 bytes each, numbered from 0 through 65535, thus allowing a maximum of 8 megabytes per file. Note, however, that although the records may be considered logically contiguous, they may not be physically contiguous in the disk data area. Internally, all files are broken into 16K byte segments called logical extents, so that counters are easily maintained as 8-bit values. Although the decomposition into extents is discussed in the paragraphs which follow, they are of no particular consequence to the programmer since each extent is automatically accessed in both sequential and random access modes.

In the file operations starting with function number 15, DE usually addresses a file control block (FCB). Transient programs often use the default file control block area reserved by CP/M at location BOOT + 005CH (normally 005CH) for simple file operations. The basic unit of file information is a 128 byte record used for all file operations, thus a default location for disk I/O is provided by CP/M at location BOOT + 0080H (normally 0080H) which is the initial default DMA address (see function 26). All directory operations take place in a reserved area which does not affect write buffers as was the case in release 1, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at 005CH can be used for random access files, since the three bytes starting at BOOT + 007DH are available for this purpose. The FCB format is shown with the following fields:



where

dr — drive code (0 - 16)
0 = >use default drive for file
1 = >auto disk select drive A,
2 = >auto disk select drive B,
...
16 = >auto disk select drive P.

f1. . .f8 — contain the file name in ASCII upper case, with high bit = 0

t1,t2,t3 — contain the file type in ASCII upper case, with high bit = 0
t1', t2', and t3' denote the bit of these positions,
t1' = 1 = >Read/Only file,
t2' = 1 = >SYS file, no DIR list

ex — contains the current extent number, normally set to 00 by the user, but in range 0 - 31 during file I/O

s1 — reserved for internal system use

s2 — reserved for internal system use, set to zero on call to OPEN, MAKE, SEARCH

rc — record count for extent "ex," takes on values from 0 - 128

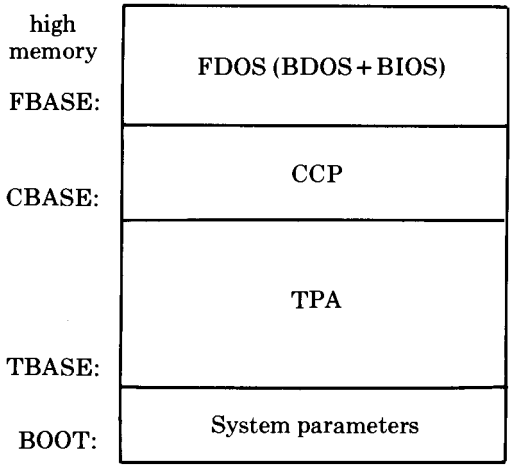
CHAPTER 2
CP/M 2.0 INTERFACE GUIDE

- **Introduction**
- **Operating System Call Conventions**
- **Sample File-to-File Copy Program**
- **Sample File Dump Utility**
- **Sample Random Access Program**
- **System Function Summary**

Introduction

This manual describes CP/M, release 2, system organization including the structure of memory and system entry points. The intention is to provide the necessary information required to write programs which operate under CP/M, and which use the peripheral and disk I/O facilities of the system.

CP/M is logically divided into four parts, called the Basic I/O System (BIOS), the Basic Disk Operating System (BDOS), the Console command processor (CCP), and the Transient Program Area (TPA). The BIOS is a hardware-dependent module which defines the exact low level interface to a particular computer system which is necessary for peripheral device I/O. The BIOS and BDOS are logically combined into a single module with a common entry point, and referred to as the FDOS. The CCP is a distinct program which uses the FDOS to provide a human-oriented interface to the information which is cataloged on the backup storage device. The TPA is an area of memory (i.e., the portion which is not used by the FDOS and CCP) where various non-resistant operating system commands and user programs are executed. The lower portion of memory is reserved for system information and is detailed in later sections. Memory organization of the CP/M system is shown below:



Base addresses for the two Apple memory configurations that can be used with CP/M are shown in the table below:

Module	**44K**	**56K (Language Card)**
CCP	9400H	C400H
BDOS	9C00H	CC00H

BIOS	AA00H	DA00H
Top of RAM	AFFFH	DFFFH

All standard CP/M versions assume BOOT = 0000H, which is the base of random access memory. The machine code found at location BOOT performs a system "warm start" which loads and initializes the programs and variables necessary to return control to the CCP. Thus, transient programs need only jump to location BOOT to return control to CP/M at the command level. Further, the standard versions assume TBASE = BOOT + 0100H which is normally location 0100H. The principal entry point to the FDOS is at location BOOT + 0005H (normally 0005H) where a jump to FBASE is found. The address field at BOOT + 0006H (normally 0006H) contains the value of FBASE and can be used to determine the size of available memory, assuming the CCP is being overlayed by a transient program.

Transient programs are loaded into the TPA and executed as follows. The operator communicates with the CCP by typing command lines following each prompt. Each command line takes one of the forms:

```
command
command file1
command file1 file2
```

where "command" is either a built-in function such as DIR or TYPE, or the name of a transient command or program. If the command is a built-in function of CP/M, it is executed immediately. Otherwise, the CCP searches the currently addressed disk for a file by the name

command.COM

If the file is found, it is assumed to be a memory image of a program which executes in the TPA, and thus implicitly originates at TBASE in memory. The CCP loads the COM file from the disk into memory starting at TBASE and possibly extending up to CBASE.

If the command is followed by one or two file specifications, the CCP prepares one or two file control block (FCB) names in the system parameter area. These optional FCB's are in the form necessary to access files through the FDOS, and are described in the next section.

The transient program receives control from the CCP and begins execution, perhaps using the I/O facilities of the FDOS. The transient program is "called" from the CCP, and thus can simply return to the CCP upon completion of its processing, or can jump to BOOT to pass control back to CP/M. In the first case, the transient program must not use memory above CBASE, while in the latter case, memory up through FBASE-1 is free.

The transient program may use the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the FDOS entry point at BOOT + 0005H. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB to the CP/M FDOS. The FDOS, in turn, performs the operation and returns with either a disk read completion indication or an error number indicating that the disk read was unsuccessful. The function numbers and error indicators are given below.

Operating System Call Conventions

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs.

CP/M facilities which are available for access by transient programs fall into two general categories: simple device I/O, and disk file I/O. The simple device operations include:

Read a Console Character
Write a Console Character
Read a Sequential Tape Character
Write a Sequential Tape Character
Write a List Device Character
Get or Set I/O Status
Print Console Buffer
Read Console Buffer
Interrogate Console Ready

The FDOS operations which perform disk Input/Output are

Disk System Reset
Drive Selection
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Random or Sequential Read
Random or Sequential Write
Interrogate Available Disks
Interrogate Selected Disk
Set DMA Address
Set/Reset File Indicators

As mentioned above, access to the FDOS functions is accomplished by passing a function number and information address through the primary entry point at location BOOT + 0005H. In general, the function number is passed in register C with the information address in the double byte pair DE. Single byte values are returned in register A, with double byte values returned in HL (a zero value is returned when the function number is out of range). For reasons of compatibility, register A = L and register B = H upon return in all cases. Note that the register passing conventions of CP/M agree with those of Intel's PL/M systems programming language. The list of CP/M function numbers is given below.

0 System Reset
1 Console Input
2 Console Output
3 Reader Input
4 Punch Output
5 List Output
6 Direct Console I/O
7 Get I/O Byte
8 Set I/O Byte
9 Print String
10 Read Console Buffer
11 Get Console Status
12 Return Version Number
13 Reset Disk System
14 Select Disk
15 Open File
16 Close File
17 Search for First
18 Search for Next
19 Delete File
20 Read Sequential
21 Write Sequential
22 Make File
23 Rename File
24 Return Login Vector
25 Return Current Disk
26 Set DMA Address
27 Get Addr (Alloc)
28 Write Protect Disk
29 Get R/O Vector
30 Set File Attributes
31 Get Addr (Disk Parms)
32 Set/Get User Code
33 Read Random
34 Write Random
35 Compute File Size
36 Set Random Record

(Functions 28 and 32 should be avoided in application programs to maintain upward compatibility with MP/M.)

Upon entry to a transient program, the CCP leaves the stack pointer set to an eight level stack area with the CCP return address pushed onto the stack, leaving seven levels before overflow occurs. Although this stack is usually not used by a transient program (i.e., most transients return to the CCP through a jump to location 0000H), it is sufficiently large to make CP/M system calls since the FDOS switches to a local stack at system entry. The following assembly language program segment, for example, reads characters continuously until an asterisk is encountered, at which time control returns to the CCP (assuming a standard CP/M system with BOOT + 0000H):

```
BDOS    EQU     0005H   ;STANDARD CP/M ENTRY
CONIN   EQU     1       ;CONSOLE INPUT FUNCTION
;
        ORG     0100H   ;BASE OF TPA
NEXTC:  MVI     C,CONIN ;READ NEXT CHARACTER
        CALL    BDOS    ;RETURN CHARACTER IN <A>
        CPI     '*'     ;END OF PROCESSING?
        JNZ     NEXTC   ;LOOP IF NOT
        RET             ;RETURN TO CCP
        END
```

CP/M implements a named file structure on each disk, providing a logical organization which allows any particular file to contain any number of records from completely empty, to the full capacity of the drive. Each drive is logically distinct with a disk directory and file data area. The disk file names are in three parts: the drive select code, the file name consisting of one to eight non-blank characters, and the file type consisting of zero to three non-blank characters. The file type names the generic category of a particular file, while the file name distinguishes individual files in each category. The file types listed below name a few generic categories which have been established, although they are generally arbitrary:

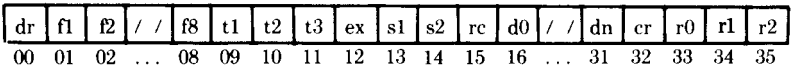
ASM	Assembler Source	PLI	PL/I Source File
PRN	Printer Listing	REL	Relocatable Module
HEX	Hex Machine Code	TEX	TEX Formatter Source
BAS	Basic Source File	BAK	ED Source Backup
INT	Intermediate Code	SYM	SID Symbol File
COM	CCP Command File	$$$	Temporary File

Source files are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (0DH followed by 0AH). Thus one 128 byte CP/M record could contain several lines of source text. The end of an ASCII file is denoted by a control-Z character (1AH) or a real end of file, returned by the CP/M read operation. Control-Z characters embedded within machine code files (e.g., COM files) are ignored, however, and the end of file condition returned by CP/M is used to terminate read operations.

Files in CP/M can be thought of as a sequence of up to 65536 records of 128 bytes each, numbered from 0 through 65535, thus allowing a maximum of 8 megabytes per file. Note, however, that although the records may be considered logically contiguous, they may not be physically contiguous in the disk data area. Internally, all files are broken into 16K byte segments called logical extents, so that counters are easily maintained as 8-bit values. Although the decomposition into extents is discussed in the paragraphs which follow, they are of no particular consequence to the programmer since each extent is automatically accessed in both sequential and random access modes.

In the file operations starting with function number 15, DE usually addresses a file control block (FCB). Transient programs often use the default file control block area reserved by CP/M at location BOOT + 005CH (normally 005CH) for simple file operations. The basic unit of file information is a 128 byte record used for all file operations, thus a default location for disk I/O is provided by CP/M at location BOOT + 0080H (normally 0080H) which is the initial default DMA address (see function 26). All directory operations take place in a reserved area which does not affect write buffers as was the case in release 1, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at 005CH can be used for random access files, since the three bytes starting at BOOT + 007DH are available for this purpose. The FCB format is shown with the following fields:



where

dr — drive code (0 - 16)
0 = >use default drive for file
1 = >auto disk select drive A,
2 = >auto disk select drive B,
...
16 = >auto disk select drive P.

f1...f8 — contain the file name in ASCII upper case, with high bit = 0

t1,t2,t3 — contain the file type in ASCII upper case, with high bit = 0
t1', t2', and t3' denote the bit of these positions,
t1' = 1 = >Read/Only file,
t2' = 1 = >SYS file, no DIR list

ex — contains the current extent number, normally set to 00 by the user, but in range 0 - 31 during file I/O

s1 — reserved for internal system use

s2 — reserved for internal system use, set to zero on call to OPEN, MAKE, SEARCH

rc — record count for extent "ex," takes on values from 0 - 128

d0. . .dn	filled-in by CP/M, reserved for system use
cr	current record to read or write in a sequential file operation, normally set to zero by user
r0,r1,r2	optional random record number in the range 0-65535, with overflow to r2, r0,r1 constitute a 16-bit value with low byte r0, and high byte r1

Each file being accessed through CP/M must have a corresponding FCB which provides the name and allocation information for all subsequent file operations. When accessing files, it is the programmer's responsibility to fill the lower sixteen bytes of the FCB and initialize the "cr" field. Normally, bytes 1 through 11 are set to the ASCII character values for the file name and file type, while all other fields are zero.

FCB's are stored in a directory area of the disk, and are brought into central memory before proceeding with file operations (see the OPEN and MAKE functions). The memory copy of the FCB is updated as file operations take place and later recorded permanently on disk at the termination of the file operation (see the CLOSE command).

The CCP constructs the first sixteen bytes of two optional FCB's for a transient by scanning the remainder of the line following the transient name, denoted by "file1" and "file2" in the prototype command line described above, with unspecified fields set to ASCII blanks. The first FCB is constructed at location BOOT + 005CH, and can be used as-is for subsequent file operations. The second FCB occupies the d0 . . . dn portion of the first FCB, and must be moved to another area of memory before use. If, for example, the operator types

PROGNAME B:X.ZOT Y.ZAP

the file PROGNAME. COM is loaded into the TPA, and the default FCB at BOOT + 005CH is initialized to drive code 2, file name "X" and file type "ZOT". The second drive code takes the default value 0, which is placed at BOOT + 006CH, with the file name "Y" placed into location BOOT + 006DH and file type "ZAP" located 8 bytes later at BOOT + 0075H. All remaining fields through "cr" are set to zero. Note again that it is the programmer's responsibility to move this second file name and type to another area, usually a separate file control block, before opening the file which begins at BOOT + 005CH, due to the fact that the open operation will overwrite the second name and type.

If no file names are specified in the original command, then the fields beginning at BOOT + 005DH and BOOT + 006DH contain blanks. In all

cases, the CCP translates lower case alphabetics to upper case to be consistent with the CP/M file naming conventions.

As an added convenience, the default buffer area at location BOOT + 0080H is initialized to the command line tail typed by the operator following the program name. The first position contains the number of characters, with the characters themselves following the character count. Given the above command line, the area beginning at BOOT + 0080H is initialized as follows:

BOOT + 0080H:

+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+10	+11	+12	+13	+14
14	" "	"B"	":"	"X"	"."	"Z"	"O"	"T"	" "	"Y"	"."	"Z"	"A"	"P"

where the characters are translated to upper case ASCII with unintialized memory following the last valid character. Again, it is the responsibility of the programmer to extract the information from this buffer before any file operations are performed, unless the default DMA address is explicitly changed.

The individual functions are described in detail in the pages which follow.

FUNCTION 0: System Reset

Entry Parameters:
Register C: 00H

The system reset function returns control to the CP/M operating system at the CCP level. The CCP re-initializes the disk subsystem by selecting and logging-in disk drive A. This function has exactly the same effect as a jump to location BOOT.

FUNCTION 1: CONSOLE INPUT

Entry Parameters:
Register C: 01H

Returned Value :
Register A: ASCII Character

The console input function reads the next console character to register A. Graphic characters, along with carriage return, line feed, and backspace (ctl-H) are echoed to the console. Tab characters (ctl-I) are expanded in columns of eight characters. A check is made for start/stop scroll (ctl-S) and start/stop printer echo (ctl-P). The FDOS does not return to the calling program until a character has been typed, thus suspending execution of a character if not ready.

FUNCTION 2: CONSOLE OUTPUT

Entry Parameters:
Register C: 02H
Register E: ASCII Character

The ASCII character from register E is sent to the console device. Similar to function 1, tabs are expanded and checks are made for start/stop scroll and printer echo.

FUNCTION 3: READER INPUT

Entry Parameters:
Register C: 03H

Returned Value :
Register A: ASCII Character

The Reader Input function reads the next character from the logical reader into register A. Control does not return until the character has been read.

FUNCTION 4: PUNCH OUTPUT

Entry Parameters:

Register C: 04H
Register E: ASCII Character

The Punch Output function sends the character from register E to the logical punch device.

FUNCTION 5: LIST OUTPUT

Entry Parameters:
Register C: 05H
Register E: ASCII Character

The List Output function sends the ASCII character in register E to the logical listing device.

FUNCTION 6: DIRECT CONSOLE I/O

Entry Parameters:
Register C: 06H
Register E: 0FFH (input) or
char (output)

Returned Value :
Register A: char or status
(no value)

Direct console I/O is supported under CP/M for those specialized applications where unadorned console input and output is required. Use of this function should, in general, be avoided since it bypasses all of CP/M's normal control character functions (e.g., control-S and control-P). Programs which perform direct I/O through the BIOS under previous releases of CP/M, however, should be changed to use direct I/O under BDOS so that they can be fully supported under future releases of MP/M and CP/M.

Upon entry to function 6, register E either contains hexadecimal FF, denoting a console input request, or register E contains an ASCII character. If the input value is FF, then function 6 returns A = 00 if no character is ready, otherwise A contains the next console input character.

If the input value in E is not FF, then function 6 assumes that E contains a valid ASCII character which is sent to the console.

FUNCTION 7: GET I/O BYTE

Entry Parameters:
Register C: 07H

Returned Value:
Register A: I/O Byte Value

The Get I/O Byte function returns the current value of IOBYTE in register A.

FUNCTION 8: SET I/O BYTE

Entry Parameters:
Register C: 08H
Register E: I/O Byte Value

The Set I/O Byte function changes the system IOBYTE value to that given in register E.

FUNCTION 9: PRINT STRING

Entry Parameters:
Register C: 09H
Registers DE: String Address

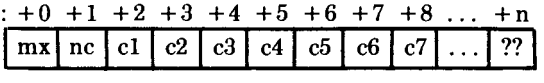
The Print String function sends the character string stored in memory at the location given by DE to the console device, until a "$" is encountered in the string. Tabs are expanded as in function 2, and checks are made for start/stop scroll and printer echo.

FUNCTION 10: READ CONSOLE BUFFER

Entry Parameters:
Register C: 0AH
Registers DE: Buffer Address

Returned Value :
Console Characters in Buffer

The Read Buffer function reads a line of edited console input into a buffer addressed by registers DE. Console input is terminated when either the input buffer overflows. The Read Buffer takes the form:



where "mx" is the maximum number of characters which the buffer will hold (1 to 255), "nc" is the number of characters read (set by FDOS upon return), followed by the characters read from the console. If nc < mx, then uninitialized positions follow the last character, denoted by "??" in the above figure. A number of control functions are recognized during line editing:

rub/del	removes the echoes the last character
ctl-C	reboots when at the beginning of line
ctl-E	causes physical end of line
ctl-H	backspaces one character position
ctl-J	(line feed) terminates input line
ctl-M	(return) terminates input line
ctl-R	retypes the current line after new line
ctl-X	backspaces to beginning of current line

Note also that certain functions which return the carriage to the leftmost

position (e.g., ctl-X) do so only to the column position where the prompt ended (in earlier releases, the carriage returned to the extreme left margin). This convention makes operator data input and line correction more legible.

FUNCTION 11: GET CONSOLE STATUS

Entry Parameters:
Register C: 0BH

Return Value :
Register A: Console Status

The Console Status function checks to see if a character has been typed at the console. If a character is ready, the value 0FFH is returned in register A. Otherwise a 00H value is returned.

FUNCTION 12: RETURN VERSION NUMBER

Entry Parameters:
Register C: 0CH

Returned Value :
Registers HL: Version Number

Function 12 provides information which allows version independent programming. A two-byte value is returned, with H=00 designating the CP/M release (H=01 for MP/M), and L=00 for all releases previous to 2.0. CP/M 2.0 returns a hexadecimal 20 in register L, with subsequent version 2 releases in the hexadecimal range 21, 22, through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions, with random access disabled when operating under early releases of CP/M.

FUNCTION 13: RESET DISK SYSTEM

Entry Parameters:
Register C: 0DH

The Reset Disk Function is used to programmatically restore the file system to a reset state where all disks are set to read/write (see functions 28 and 29), only disk drive A is selected, and the default DMA address is reset to BOOT+0080H. This function can be used, for example, by an application program which requires a disk change without a system reboot.

FUNCTION 14: SELECT DISK

Entry Parameters:
Register C: 0EH
Register E: Selected Disk

The Select Disk function designates the disk drive named in register E as the default disk for subsequent file operations, with E = 0 for drive A, 1 for drive B, and so-forth through 15 corresponding to drive P in a full sixteen drive system. The drive is placed in an "on-line" status which, in particular, activates its directory until the next cold start, warm start, or disk system reset operation. If the disk media is changed while it is on-line, the drive automatically goes to a read/only status in a standard CP/M environment (see function 28). FCB's which specify drive code zero (dr = 00H) automatically reference the currently selected default drive. Drive code values between 1 and 16, however, ignore the selected default drive and directly reference drives A through P.

FUNCTION 15: OPEN FILE

Entry Parameters:
Register C: 0FH
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Open File operation is used to activate a file which currently exists in the disk directory for the currently active user number. The FDOS scans the referenced disk directory for a match in positions 1 through 14 of the FCB referenced by DE (byte s1 is automatically zeroed), where an ASCII question mark (3FH) matches any directory character in any of these positions. Normally, no question marks are included and, further, bytes "ex" and "s2" of the FCB are zero.

If a directory element is matched, the relevant directory information is copied into bytes d0 through dn of the FCB, thus allowing access to the files through subsequent read and write operations. Note that an existing file must not be accessed until a successful open operation is completed. Upon return, the open function returns a "directory code" with the value 0 through 3 if the open was successful, or 0FFH (255 decimal) if the file cannot be found. If question marks occur in the FCB then the first matching FCB is activated. Note that the current record ("cr") must be zeroed by the program if the file is to be accessed sequentially from the first record.

FUNCTION 16: CLOSE FILE

Entry Parameters:
Register C: 10H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Close File function performs the inverse of the open file function. Given that the FCB addressed by DE has been previously activated through an open or make function (see functions 15 and 22), the close function permanently records the new FCB in the referenced disk directory. The FCB matching process for the close is identical to the open function. The directory code returned for a successful close operation is 0, 1, 2, or 3, while a 0FFH (255 decimal) is returned if the file name cannot be found in the directory. A file need not be closed if only read operations have taken place. If write operations have occurred, however, the close operation is necessary to permanently record the new directory information.

FUNCTION 17: SEARCH FOR FIRST

Entry Parameters:
Register C: 11H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

Search First scans the directory for a match with the file given by the FCB addressed by DE. The value 255 (hexadecimal FF) is returned if the file is not found, otherwise 0, 1, 2, or 3 is returned indicating the file is present. In the case that the file is found, the current DMA address is filled with the record containing the directory entry, and the relative starting position is A * 32 (i.e., rotate the A register left 5 bits, or ADD A five times). Although not normally required for application programs, the directory information can be extracted from the buffer at this position.

An ASCII question mark (63 decimal, 3F hexadecimal) in any position from "f1" through "ex" matches the corresponding field of any directory entry on the default or auto-selected disk drive. If the "dr" field contains an ASCII question mark, then the auto disk selected function is disabled, the default disk is searched, with the search function returning any matched entry, allocated or free, belonging to any user number. This latter function is not normally used by application programs, but does allow complete flexibility to scan all current directory values. If the "dr" field is not a question mark, the "s2" byte is automatically zeroed.

FUNCTION 18: SEARCH FOR NEXT

Entry Parameters:
Register C: 12H

Returned Value :
Register A: Directory Code

The Search Next function is similar to the Search First function, except that the directory scan continues from the last matched entry. Similar to function 17, function 18 returns the decimal value 255 in A when no more directory items match.

FUNCTION 19: DELETE FILE

Entry Parameters:
Register C: 13H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Delete File function removes files which match the FCB addresses by DE. The filename and type may contain ambiguous references (i.e., question marks in various positions), but the drive select code cannot be ambiguous, as in the Search and Search Next functions.

Function 19 returns a decimal 255 if the referenced file or files cannot be found, otherwise a value in the range 0 to 3 is returned.

FUNCTION 20: READ SEQUENTIAL

Entry Parameters:
Register C: 14H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

Given that the FCB addressed by DE has been activated through an open or make function (numbers 15 and 22), the Read Sequential function reads the next 128 byte record from the file into memory at the current DMA address. The record is read from position "cr" of the extent, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next read operation. The value 00H

is returned in the A register if the read operation was successful, while a non-zero value is returned if no data exists at the next record position (e.g., end of file occurs).

FUNCTION 21: WRITE SEQUENTIAL

Entry Parameters:
Register C: 15H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

Given that the FCB addressed by DE has been activated through an open or make function (numbers 15 and 22), the Write Sequential function writes the 128 byte data record at the current DMA address to the file named by the FCB. The record is placed at position "cr" of the file, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next write operation. Write operations can take place into an existing file, in which case newly written records overlay those which already exist in the file. Register A = 00H upon return from a successful write operation, while a non-zero value indicates an unsuccessful write due to a full disk.

FUNCTION 22: MAKE FILE

Entry Parameters:
Register C: 16H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Make File operation is similar to the open file operation except that the FCB must name a file which does not exist in the currently referenced disk directory (i.e., the one named explicitly by a non-zero "dr" code, or the default disk if "dr" is zero). The FDOS creates the file and initializes both the directory and main memory value to an empty file. The programmer must ensure that no duplicate file names occur, and a preceding delete operation is sufficient if there is any possibility of duplication. Upon return, register A = 0, 1, 2, or 3 if the operation was successful and 0FFH (255 decimal) if no more directory space is available. The make function has the side-effect of activating the FCB and thus a subsequent open is not necessary.

FUNCTION 23: RENAME FILE

Entry Parameters:
Register C: 17H
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Rename function uses the FCB addressed by DE to change all occurrences of the file named in the first 16 bytes to the file named in the second 16 bytes. The drive code "dr" at position 0 is used to select the drive, while the drive code for the new file name at position 16 of the FCB is assumed to be zero. Upon return, register A is set to a value between 0 and 3 if the rename was successful, and 0FFH (255 decimal) if the first file name could not be found in the directory scan.

FUNCTION 24: RETURN LOGIN VECTOR

Entry Parameters:
Register C: 18H

Returned Value :
Registers HL: Login Vector

The login vector value returned by CP/M is a 16-bit value in HL, where the least significant bit of L corresponds to the first drive A, and the high order bit of H corresponds to the sixteenth drive, labelled P. A "0" bit indicates that the drive is not on-line, while a "1" bit marks a drive that is actively on-line due to an explicit disk drive selection, or an implicit drive select caused by a file operation which specified a non-zero "dr" field. Note that compatibility is maintained with earlier releases, since registers A and L contain the same values upon return.

FUNCTION 25: RETURN CURRENT DISK

Entry Parameters:
Register C: 19H

Returned Value :
Register A: Current Disk

Function 25 returns the currently selected default disk number in register A. The disk numbers range from 0 through 15 corresponding to drives A through P.

FUNCTION 26: SET DMA ADDRESS

Entry Parameters:
Regular C: 1AH
Registers DE: DMA Address

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers which directly access the memory of the mainframe computer to transfer data to and from the disk subsystem. Although many computer systems use non-DMA access (i.e., the data is transferred through programmed I/O operations), the DMA address has, in CP/M, come to mean the address at which the 128 byte data record resides before a disk write and after a disk read. Upon cold start, warm start, or disk system reset, the DMA address is automatically set to BOOT + 0080H. The Set DMA function, however, can be used to change this default value to address another area of memory where the data records reside. Thus, the DMA address becomes the value specified by DE until it is changed by a subsequent Set DMA function, cold start, warm start, or disk system reset.

FUNCTION 27: GET ADDR (ALLOC)

Entry Parameters:
Register C: 1BH

Returned Value :
Registers HL: ALLOC Address

An "allocation vector" is maintained in main memory for each on-line disk drive. Various system programs use the information provided by the allocation vector to determine the amount of remaining storage (see the STAT program). Function 27 returns the base address of the allocation vector for the currently selected disk drive. The allocation information may, however, be invalid if the selected disk has been marked read/only. Although this function is not normally used by application programs, additional details of the allocation vector are found in the "CP/M Alteration Guide."

FUNCTION 28: WRITE PROTECT DISK

Entry Parameters:
Register C: 1CH

The disk write protect function provides temporary write protection for the currently selected disk. Any attempt to write to the disk, before the next cold or warm start operation produces the message

Bdos Err on d: R/O

FUNCTION 29: GET READ/ONLY VECTOR

Entry Parameters:
Register C: 1DH

Returned Value :
Registers HL: R/O Vector Value

Function 29 returns a bit vector in register pair HL which indicates drives which have the temporary read/only bit set. Similar to function 24, the least significant bit corresponds to drive A, while the most significant bit corresponds to drive P. The R/O bit is set either by the explicit call to function 28, or by the automatic software mechanisms within CP/M which detect changed disks.

FUNCTION 30: SET FILE ATTRIBUTES

Entry Parameters:
Register C: 1EH
Registers DE: FCB Address

Returned Value :
Register A: Directory Code

The Set File Attributes function allows programmatic manipulation of permanent indicators attached to files. In particular, the R/O and System attributes (t1' and t2') can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 30 searches for a match, and changes the matched directory entry to contain the selected indicators. Indicators f1' through f4' are not presently used, but may be useful for applications programs, since they are not involved in the matching process during file open and close operations. Indicators f5' through f8' and t3' are reserved for future system expansion.

FUNCTION 31: GET ADDR (DISK PARMS)

Entry Parameters:
Register C: 1FH

Returned Value :
Registers HL: DPB Address

The address of the BIOS resident disk parameter block is returned in HL as a result of this function call. This address can be used for either of two purposes. First, the disk parameter values can be extracted for display and

space computation purposes, or transient programs can dynamically change the values of current disk parameters when the disk environment changes, if required. Normally, application programs will not require this facility.

FUNCTION 32: SET/GET USER CODE

Entry Parameters:
Register C: 20H
Register E: 0FFH (get or User Code (set)

Returned Value :
Register A: Current Code or (no value)

An application program can change or interrogate the currently active user number by calling function 32. If register E = 0FFH, then the value of the current user number is returned in register A, where the value is in the range 0 to 31. If register E is not 0FFH, then the current user number is changed to the value of E (modulo 32).

FUNCTION 33: READ RANDOM

Entry Parameters:
Register C: 21H
Registers DE: FCB Address

Returned Value :
Register A: Return Code

The Read Random function is similar to the sequential file read operation of previous releases, except that the read operation takes place at a particular record number, selected by the 24-bit value constructed from the three byte field following the FCB (byte positions r0 at 33, r1 at 34, and r2 at 35). Note that the sequence of 24 bits is stored with least significant byte first (r0), middle byte next (r1), and high byte last (r2). CP/M does not reference byte r2, except in computing the size of a file (function 35). Byte r2 must be zero, however, since a non-zero value indicates overflow past the end of file.

Thus, the r0,r1 byte pair is treated as a double-byte, or "word" value, which contains the record to read. This value ranges from 0 to 65535, providing access to any particular record of the 8 megabyte file. In order to process a file using random access, the base extent (extent 0) must first be opened. Although the base extent may or may not contain any allocated data, this ensures that the file is properly recorded in the directory, and is visible in DIR requests. The selected record number is then stored into the random record field (r0,r1), and the BDOS is called to read the record. Upon return from the

call, register A either contains an error code, as listed below, or the value 00 indicating the operation was successful. In the latter case, the current DMA address contains the randomly accessed record. Note that contrary to the sequential read operation, the record number is not advanced. Thus, subsequent random read operations continue to read the same record.

Upon each random read operation, the logical extent and current record values are automatically set. Thus, the file can be sequentially read or written, starting from the current randomly accessed position. Note, however, that in this case, the last randomly read record will be re-read as you switch from random mode to sequential read, and the last record will be re-written as you switch to a sequential write operation. You can, of course, simply advance the random record position following each random read or write to obtain the effect of a sequential I/O operation.

Error codes returned in register A following a random read are listed below.

01 reading unwritten data
02 (not returning in random mode)
03 cannot close current extent
04 seek to unwritten extent
05 (not returned in read mode)
06 seek past physical end of disk

Error code 01 and 04 occur when a random read operation accesses a data block which has not been previously written, or an extent which has not been created, which are equivalent conditions. Error 3 does not normally occur under proper system operation, but can be cleared by simply re-reading, or re-opening extent zero as long as the disk is not physically write protected. Error code 06 occurs whenever byte r2 is non-zero under the current 2.0 release. Normally, non-zero return codes can be treated as missing data, with zero return codes indicating operation complete.

FUNCTION 34: WRITE RANDOM

Entry Parameters:
Register C: 22H
Registers DE: FCB Address

Returned Value :
Register A: Return Code

The Write Random operation is initiated similar to the Read Random call, except that data is written to the disk from the current DMA address. Further, if the disk extent or data block which is the target of the write has not yet been allocated, the allocation is performed before the write operation

continues. As in the Read Random operation, the random record number is not changed as a result of the write. The logical extent number and current record positions of the file control block are set to correspond to the random record which is being written. Again, sequential read or write operations can commence following a random write, with the notation that the currently addressed record is either read or rewritten again as the sequential operation begins. You can also simply advance the random record position following each write to get the effect of a sequential write operation. Note that in particular, reading or writing the last record of an extent in random mode does not cause an automatic extent switch as it does in sequential mode.

The error codes returned by a random write are identical to the random read operation with the addition of error code 05, which indicates that a new extent cannot be created due to directory overflow.

FUNCTION 35: COMPUTE FILE SIZE

Entry Parameters:
Register C: 23H
Registers DE: FCB Address

Returned Value :
Random Record Field Set

When computing the size of a file, the DE register pair addresses an FCB in random mode format (bytes r0, r1, and r2 are present). The FCB contains an unambiguous file name which is used in the directory scan. Upon return, the random record bytes contain the "virtual" file size which is, in effect, the record address of the record following the end of the file. If, following a call to function 35, the high record byte r2 is 01, then the file contains the maximum record count 65536. Otherwise, bytes r0 and r1 constitute a 16-bit value (r0 is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file by simply calling function 35 to set the random record position to the end of file, then performing a sequence of random writes starting at the preset record address

The virtual size of a file corresponds to the physical size when the file is written sequentially. If, instead, the file was created in random mode and "holes" exist in the allocation, then the file may in fact contain fewer records than the size indicates. If, for example, only the last record of an eight megabyte file is written in random mode (i.e., record number 65535), then the virtual size is 65536 records, although only one block of data is actually allocated.

FUNCTION 36: SET RANDOM RECORD

Entry Parameters:
Register C: 24H
Registers DE: FCB Address

Returned Value :
Random Record Field Set

The Set Random Record function causes the BDOS to automatically produce the random record position from a file which has been read or written sequentially to a particular point. The function can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the position of various "key" fields. As each key is encountered, function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record position is placed into a table with the key for later retrieval. After scaning the entire file and tabularizing the keys and their record numbers, you can move instantly to a particular keyed record by performing a random read using the corresponding random record number which was saved earlier. The scheme is easily generated when variable record lengths are involved since the program need only store the buffer-relative byte position along with the key and record number in order to find the exact starting position of the keyed data at a later time.

A second use of function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, function 36 is called which sets the record number, and subsequent random read and write operations continue from the selected point in the file.

Sample File-to-File Copy Program

The program shown below provides a relatively simple example of file operations. The program source file is created as COPY.ASM using the CP/M ED program and then assembled using ASM or MAC, resulting in a "HEX" file. The LOAD program is then used to produce a COPY.COM file which executes directly under the CCP. The program begins by setting the stack pointer to a local area, and then proceeds to move the second name from the default area at 006CH to a 33-byte file control block called DFCB. The DFCB is then prepared for file operations by clearing the current record field. At this point, the source and destination FCB's are ready for processing since the SFCB at 005CH is properly set-up by the CCP upon entry to the COPY program. That is, the first name is placed into the default FCB, with

the proper fields zeroed, including the current record field at 007CH. The program continues by opening the source file, deleting any existing destination file, and then creating the destination file. If all this is successful, the program loops at the label COPY until each record has been read from the source file and placed into the destination file. Upon completion of the data transfer, the destination file is closed and the program returns to the CCP command level by jumping to BOOT.

```
                ;       sample file-to-file copy program
                ;
                ;       at the ccp level, the command
                ;
                ;               copy a:x.y b:u.v
                ;
                ;       copies the file named x.y from drive
                ;       a to a file named u.v on drive b.
                ;
0000 =          boot    equ     0000h   ; system reboot
0005 =          bdos    equ     0005h   ; bdos entry point
005c =          fcbl    equ     005ch   ; first file name
005c =          sfcb    equ     fcbl    ; source fcb
006c =          fcb2    equ     006ch   ; second file name
0080 =          dbuff   equ     0080h   ; default buffer
0100 =          tpa     equ     0100h   ; beginning of tpa
                ;
0009 =          printf  equ     9       ; print buffer func#
000f =          openf   equ     15      ; open file func#
0010 =          closef  equ     16      ; close file func#
0013 =          deletef equ     19      ; delete file func#
0014 =          readf   equ     20      ; sequential read
0015 =          writef  equ     21      ; sequential write
0016 =          makef   equ     22      ; make file func#
                ;
0100                    org     tpa     ; beginning of tpa
0100 311b02             lxi     sp,stack; local stack
                ;
                ;       move second file name to dfcb
0103 0e10               mvi     c,16    ; half an fcb
0105 116c00             lxi     d,fcb2  ; source of move
0108 21da01             lxi     h,dfcb  ; destination fcb
010b 1a         mfcb:   ldax    d       ; source fcb
010c 13                 inx     d       ; ready next
010d 77                 mov     m,a     ; dest fcb
010e 23                 inx     h       ; ready next
010f 0d                 dcr     c       ; count 16...0
0110 c20b01             jnz     mfcb    ; loop 16 times
                ;
                ;       name has been moved, zero cr
0113 af                 xra     a       ; a = 00h
0114 32fa01             sta     dfcbcr  ; current rec = 0
                ;
                ;       source and destination fcb's ready
                ;
0117 115c00             lxi     d,sfcb  ; source file
011a cd6901             call    open    ; error if 255
011d 118701             lxi     d,nofile; ready message
0120 3c                 inr     a       ; 255 becomes 0
0121 cc6101             cz      finis   ; done if no file
                ;
                ;       source file open, prep destination
0124 11da01             lxi     d,dfcb  ; destination
0127 cd7301             call    delete  ; remove if present
                ;
012a 11da01             lxi     d,dfcb  ; destination
012d cd8201             call    make    ; create the file
0130 119601             lxi     d,nodir ; ready message
```

```
0133 3c             inr     a       ; 255 becomes 0
0134 cc6101         cz      finis   ; done if no dir space
                ;
                ;       source file open, dest file open
                ;       copy until end of file on source
                ;
0137 115c00 copy:   lxi     d,sfcb  ; source
013a cd7801         call    read    ; read next record
013d b7             ora     a       ; end of file?
013e c25101         jnz     eofile  ; skip write if so
                ;
                ;       not end of file, write the record
0141 11da01         lxi     d,dfcb  ; destination
0144 cd7d01         call    write   ; write record
0147 11a901         lxi     d,space ; ready message
014a b7             ora     a       ; 00 if write ok
014b c46101         cnz     finis   ; end if so
014e c33701         jmp     copy    ; loop until eof
                ;
                eofile: ; end of file, close destination
0151 11da01         lxi     d,dfcb  ; destination
0154 cd6e01         call    close   ; 255 if error
0157 21bb01         lxi     h,wrprot; ready message
015a 3c             inr     a       ; 255 becomes 00
015b cc6101         cz      finis   ; shouldn't happen
                ;
                ;       copy operation complete, end
015e 11cc01         lxi     d,normal; ready message
                ;
                finis:  ; write message given by de, reboot
0161 0e09           mvi     c,printf
0163 cd0500         call    bdos    ; write message
0166 c30000         jmp     boot    ; reboot system
                ;
                ;       system interface subroutines
                ;       (all return directly from bdos)
                ;
0169 0e0f   open:   mvi     c,openf
016b c30500         jmp     bdos
                ;
016e 0e10   close:  mvi     c,closef
0170 c30500         jmp     bdos
                ;
0173 0e13   delete: mvi     c,deletef
0175 c30500         jmp     bdos
                ;
0178 0e14   read:   mvi     c,readf
017a c30500         jmp     bdos
                ;
017d 0e15   write:  mvi     c,writef
017f c30500         jmp     bdos
                ;
0182 0e16   make:   mvi     c,makef
0184 c30500         jmp     bdos
                ;
                ;       console messages
0187 6e6f20fnofile: db      'no source file$'
0196 6e6f209nodir:  db      'no directory space$'
01a9 6f7574fspace:  db      'out of data space$'
01bb 7772695wrprot: db      'write protected?$'
01cc 636f700normal: db      'copy complete$'
                ;
                ;       data areas
01da        dfcb:   ds      33      ; destination fcb
01fa =      dfcbcr  equ     dfcb+32 ; current record
                ;
01fb                os      32      ; 16 level stack
                stack:
021b                end
```

Note that there are several simplifications in this particular program. First, there are no checks for invalid file names which could, for example, contain ambiguous references. This situation could be detected by scanning the 32 byte default area starting at location 005CH for ASCII question marks. A check should also be made to ensure that the file names have, in fact, been included (check locations 005DH and 006DH for non-blank ASCII characters). Finally, a check should be made to ensure that the source and destination file names are different. A speed improvement could be made by buffering more data on each read operation. One could, for example, determine the size of memory by fetching FBASE from location 0006H and use the entire remaining portion of memory for a data buffer. In this case, the programmer simply resets the DMA address to the next successive 128 byte area before each read. Upon writing to the destination file, the DMA address is reset to the beginning of the buffer and incremented by 128 bytes to the end as each record is transferred to the destination file.

Sample File Dump Utility.

The file dump program shown below is slightly more complex than the single copy program given in the previous section. The dump program reads an input file, specified in the CCP command line, and displays the content of each record in hexadecimal format at the console. Note that the dump program saves the CCP's stack upon entry, resets the stack to a local area, and restores the CCP's stack before returning directly to the CCP. Thus, the dump program does not perform warm start at the end of processing.

```
                ; DUMP program reads input file and displays hex data
                ;
0100                    org     100h
0005 =          bdos    equ     0005h   ;dos entry point
0001 =          cons    equ     1       ;read console
0002 =          typef   equ     2       ;type function
0009 =          printf  equ     9       ;buffer print entry
000b =          brkf    equ     11      ;break key function (true if char
000f =          openf   equ     15      ;file open
0014 =          readf   equ     20      ;read function
                ;
005c =          fcb     equ     5ch     ;file control block address
0080 =          buff    equ     80h     ;input disk buffer address
                ;
                ;       non graphic characters
000d =          cr      equ     0dh     ;carriage return
000a =          lf      equ     0ah     ;line feed
                ;
                ;       file control block definitions
005c =          fcbdn   equ     fcb+0   ;disk name
005d =          fcbfn   equ     fcb+1   ;file name
0065 =          fcbft   equ     fcb+9   ;disk file type (3 characters)
0068 =          fcbrl   equ     fcb+12  ;file's current reel number
006b =          fcbrc   equ     fcb+15  ;file's record count (0 to 128)
007c =          fcber   equ     fcb+32  ;current (next) record number (0
007d =          fcbln   equ     fcb+33  ;fcb length
                ;
                ;       set up stack
0100 210000             lxi     h,0
0103 39                 dad     sp
                ;       entry stack pointer in hl from the ccp
```

```
0104 221502             shld    oldsp
                ;       set sp to local stack area (restored at finis)
0107 315702             lxi     sp,stktop
                ;       read and print successive buffers
010a cdcl01             call    setup   ;set up input file
010d feff               cpi     255     ;255 if file not present
010f c21b01             jnz     openok  ;skip if open is ok
                ;
                ;       file not there, give error message and return
0112 11f301             lxi     d,opnmsg
0115 cd9c01             call    err
0118 c35101             jmp     finis   ;to return
                ;
                openok: ;open operation ok, set buffer index to end
011b 3e80               mvi     a,80h
011d 321302             sta     ibp     ;set buffer pointer to 80h
                ;       hl contains next address to print
0120 210000             lxi     h,0     ;start with 0000
                ;
                gloop:
0123 e5                 push    h       ;save line position
0124 cda201             call    gnb
0127 e1                 pop     h       ;recall line position
0128 da5101             jc      finis   ;carry set by gnb if end file
012b 47                 mov     b,a
                ;       print hex values
                ;       check for line fold
012c 7d                 mov     a,l
012d e60f               ani     0fh     ;check low 4 bits
012f c24401             jnz     nonum
                ;       print line number
0132 cd7201             call    crlf
                ;
                ;       check for break key
0135 cd5901             call    break
                ;       accum lsb = 1 if character ready
0138 0f                 rrc             ;into carry
0139 da5101             jc      finis   ;don't print any more
                ;
013c 7c                 mov     a,h
013d cd8f01             call    phex
0140 7d                 mov     a,l
0141 cd8f01             call    phex
                nonum:
0144 23                 inx     h       ;to next line number
0145 3e20               mvi     a,' '
0147 cd6501             call    pchar
014a 78                 mov     a,b
014b cd8f01             call    phex
014e c32301             jmp     gloop
                ;
                finis:
                ;       end of dump, return to ccp
                ;       (note that a jmp to 0000h reboots)
0151 cd7201             call    crlf
0154 2a1502             lhld    oldsp
0157 f9                 sphl
                ;       stack pointer contains ccp's stack location
0158 c9                 ret             ;to the ccp
                ;
                ;
                ;       subroutines
                ;
                break:  ;check break key (actually any key will do)
0159 e5d5c5             push h! push d! push b; environment saved
015c 0e0b               mvi     c,brkf
015e cd0500             call    bdos
0161 c1d1e1             pop b! pop d! pop h; environment restored
0164 c9                 ret
                ;
                pchar:  ;print a character
```

```
0165 e5d5c5          push h! push d! push b; saved
0168 0e02            mvi     c,typef
016a 5f              mov     e,a
016b cd0500          call    bdos
016e c1d1e1          pop b! pop d! pop h; restored
0171 c9              ret
                ;
                crlf:
0172 3e0d            mvi     a,cr
0174 cd6501          call    pchar
0177 3e0a            mvi     a,lf
0179 cd6501          call    pchar
017c c9              ret
                ;
                ;
                pnib:   ;print nibble in reg a
017d e60f            ani     0fh     ;low 4 bits
017f fe0a            cpi     10
0181 d28901          jnc     pl0
                ;       less than or equal to 9
0184 c630            adi     '0'
0186 c38b01          jmp     prn
                ;
                ;       greater or equal to 10
0189 c637       pl0:    adi     'a' - 10
018b cd6501     prn:    call    pchar
018e c9              ret
                ;
                phex:   ;print hex char in reg a
018f f5              push    psw
0190 0f              rrc
0191 0f              rrc
0192 0f              rrc
0193 0f              rrc
0194 cd7d01          call    pnib    ;print nibble
0197 f1              pop     psw
0198 cd7d01          call    pnib
019b c9              ret
                ;
                err:    ;print error message
                ;       d,e addresses message ending with "$"
019c 0e09            mvi     c,printf        ;print buffer function
019e cd0500          call    bdos
01a1 c9              ret
                ;
                ;
                gnb:    ;get next byte
01a2 3a1302          lda     ibp
01a5 fe80            cpi     80h
01a7 c2b301          jnz     g0
                ;       read another buffer
                ;
                ;
01aa cdce01          call    diskr
01ad b7              ora     a       ;zero value if read ok
01ae cab301          jz      g0      ;for another byte
                ;       end of data, return with carry set for eof
01b1 37              stc
01b2 c9              ret
                ;
                g0:     ;read the byte at buff+reg a
01b3 5f              mov     e,a     ;ls byte of buffer index
01b4 1600            mvi     d,0     ;double precision index to de
01b6 3c              inr     a       ;index=index+1
01b7 321302          sta     ibp     ;back to memory
                ;       pointer is incremented
                ;       save the current file address
01ba 218000          lxi     h,buff
01bd 19              dad     d
                ;       absolute character address is in hl
01be 7e              mov     a,m
```

```
                ;       byte is in the accumulator
01bf b7                 ora     a       ;reset carry bit
01c0 c9                 ret
                ;
                setup:  ;set up file
                ;       open the file for input
01c1 af                 xra     a       ;zero to accum
01c2 327c00             sta     fcbcr   ;clear current record
                ;
01c5 115c00             lxi     d,fcb
01c8 0e0f               mvi     c,openf
01ca cd0500             call    bdos
                ;       255 in accum if open error
01cd c9                 ret
                ;
                diskr:  ;read disk file record
01ce e5d5c5             push h! push d! push b
01d1 115c00             lxi     d,fcb
01d4 0e14               mvi     c,readf
01d6 cd0500             call    bdos
01d9 c1d1e1             pop b! pop d! pop h
01dc c9                 ret
                ;
                ;       fixed message area
01dd 46494c0signon: db      'file du p version 2.0$'
01f3 0d0a4e0opnmsg: db      cr,lf,'no input file present on disk$'

                ;       variable area
0213            ibp:    ds      2       ;input buffer pointer
0215            oldsp:  ds      2       ;entry sp value from ccp
                ;
                ;       stack area
0217                    ds      64      ;reserve 32 level stack
                stktop:
                ;
0257                    end
```

Sample Random Access Program.

This manual is concluded with a rather extensive, but complete example of random access operation. The program listed below performs the simple function of reading or writing random records upon command from the terminal. Given that the program has been created, assembled, and placed into a file labelled RANDOM.COM, the CCP level command:

RANDOM X.DAT

starts the test program. The program looks for a file by the name X.DAT (in this particular case) and, if found, proceeds to prompt the console for input. If not found, the file is created before the prompt is given. Each prompt takes the form

next command?

and is followed by operator input, terminated by a carriage return. The input commands take the form

nW nR Q

where n is an integer value in the range 0 to 65535, and W, R, and Q are simple command characters corresponding to random write, random read, and quit processing, respectively. If the W command is issued, the RANDOM program issues the prompt

type data:

The operator then responds by typing up to 127 characters, followed by a carriage return. RANDOM then writes the character string into the X.DAT file at record n. If the R command is issued, RANDOM reads record number n and displays the string value at the console. If the Q command is issued, the X.DAT file is closed, and the program returns to the console command processor. In the interest of brevity, the only error message is

error, try again

The program begins with an initialization section where the input file is opened or created, followed by a continuous loop at the label "ready" where the individual commands are interpreted. The default file control block at 005CH and the default buffer at 0080H are used in all disk operations. The utility subroutines then follow, which contain the principal input line processor, called "readc." This particular program shows the elements of random access processing, and can be used as the basis for further program development.

```
                ;*****************************************************
                ;*                                                   *
                ;* sample random access program for cp/m 2.0         *
                ;*                                                   *
                ;*****************************************************
0100                    org     100h    ;base of tpa
                ;
0000 =          reboot  equ     0000h   ;system reboot
0005 =          bdos    equ     0005h   ;bdos entry point
                ;
0001 =          coninp  equ     1       ;console input function
0002 =          conout  equ     2       ;console output function
0009 =          pstring equ     9       ;print string until '$'
000a =          rstring equ     10      ;read console buffer
000c =          version equ     12      ;return version number
000f =          openf   equ     15      ;file open function
0010 =          closef  equ     16      ;close function
0016 =          makef   equ     22      ;make file function
0021 =          readr   equ     33      ;read random
0022 =          writer  equ     34      ;write random
                ;
005c =          fcb     equ     005ch   ;default file control block
007d =          ranrec  equ     fcb+33  ;random record position
007f =          ranovf  equ     fcb+35  ;high order (overflow) byte
0080 =          buff    equ     0080h   ;buffer address
                ;
000d =          cr      equ     0dh     ;carriage return
000a =          lf      equ     0ah     ;line feed
                ;
                ;*****************************************************
                ;*                                                   *
                ;* load SP, set-up file for random access            *
                ;*                                                   *
                ;*****************************************************
```

```
0100 31bc0          lxi     sp,stack
                    ;
                    ;       version 2.0?
0103 0e0c           mvi     c,version
0105 cd050          call    bdos
0108 fe20           cpi     20h     ;version 2.0 or better?
010a d2160          jnc     versok
                    ;       bad version, message and go back
010d 111b0          lxi     d,badver
0110 cdda0          call    print
0113 c3000          jmp     reboot
                    ;
                    versok:
                    ;       correct version for random access
0116 0e0f           mvi     c,openf ;open default fcb
0118 115c0          lxi     d,fcb
011b cd050          call    bdos
011e 3c             inr     a       ;err 255 becomes zero
011f c2370          jnz     ready
                    ;
                    ;       cannot open file, so create it
0122 0e16           mvi     c,makef
0124 115c0          lxi     d,fcb
0127 cd050          call    bdos
012a 3c             inr     a       ;err 255 becomes zero
012b c2370          jnz     ready
                    ;
                    ;       cannot create file, directory full
012e 113a0          lxi     d,nospace
0131 cdda0          call    print
0134 c3000          jmp     reboot  ;back to ccp
                    ;
                    ;*****************************************************
                    ;*                                                   *
                    ;*  loop back to "ready" after each command          *
                    ;*                                                   *
                    ;*****************************************************
                    ;
                    ready:
                    ;       file is ready for processing
                    ;
0137 cde50          call    readcom ;read next command
013a 227d0          shld    ranrec  ;store input record#
013d 217f0          lxi     h,ranovf
0140 3600           mvi     m,0     ;clear high byte if set
0142 fe51           cpi     'Q'     ;quit?
0144 c2560          jnz     notq
                    ;
                    ;       quit processing, close file
0147 0e10           mvi     c,closef
0149 115c0          lxi     d,fcb
014c cd050          call    bdos
014f 3c             inr     a       ;err 255 becomes 0
0150 cab90          jz      error   ;error message, retry
0153 c3000          jmp     reboot  ;back to ccp
                    ;
                    ;*****************************************************
                    ;*                                                   *
                    ;* end of quit command, process write                *
                    ;*                                                   *
                    ;*****************************************************
                    notq:
                    ;       not the quit command, random write?
0156 fe57           cpi     'W'
0158 c2890          jnz     notw
                    ;
                    ;       this is a random write, fill buffer until cr
015b 114d0          lxi     d,datmsg
015e cdda0          call    print   ;data prompt
0161 0e7f           mvi     c,127   ;up to 127 characters
0163 21800          lxi     h,buff  ;destination
```

```
                rloop:  ;read next character to buff
0166 c5                 push    b       ;save counter
0167 e5                 push    h       ;next destination
0168 cdc20              call    getchr  ;character to a
016b e1                 pop     h       ;restore counter
016c c1                 pop     b       ;restore next to fill
016d fe0d               cpi     cr      ;end of line?
016f ca780              jz      erloop
                ;       not end, store character
0172 77                 mov     m,a
0173 23                 inx     h       ;next to fill
0174 0d                 dcr     c       ;counter goes down
0175 c2660              jnz.    rloop   ;end of buffer?
                erloop:
                ;       end of read loop, store 00
0178 3600               mvi     m,0
                ;
                ;       write the record to selected record number
017a 0e22               mvi     c,writer
017c 115c0              lxi     d,fcb
017f cd050              call    bdos
0182 b7                 ora     a       ;error code zero?
0183 c2b90              jnz     error   ;message if not
0186 c3370              jmp     ready   ;for another record
                ;
                ;*************************************************
                ;*                                               *
                ;* end of write command, process read            *
                ;*                                               *
                ;*************************************************
                notw:
                ;       not a write command, read record?
0189 fe52               cpi     'R'
018b c2b90              jnz     error   ;skip if not
                ;
                ;       read random record
018e 0e21               mvi     c,readr
0190 115c0              lxi     d,fcb
0193 cd050              call    bdos
0196 b7                 ora     a       ;return code 00?
0197 c2b90              jnz     error
                ;
                ;       read was successful, write to console
019a cdcf0              call    crlf    ;new line
019d 0e80               mvi     c,128   ;max 128 characters
019f 21800              lxi     h,buff  ;next to get
                wloop:
01a2 7e                 mov     a,m     ;next character
01a3 23                 inx     h       ;next to get
01a4 e67f               ani     7fh     ;mask parity
01a6 ca370              jz      ready   ;for another command if 00
01a9 c5                 push    b       ;save counter
01aa e5                 push    h       ;save next to get
01ab fe20               cpi     ' '     ;graphic?
01ad d4c80              cnc     putchr  ;skip output if not
01b0 e1                 pop     h
01b1 c1                 pop     b
01b2 0d                 dcr     c       ;count=count-1
01b3 c2a20              jnz     wloop
01b6 c3370              jmp     ready
                ;
                ;*************************************************
                ;*                                               *
                ;* end of read command, all errors end-up here   *
                ;*                                               *
                ;*************************************************
                ;
                error:
01b9 11590              lxi     d,errmsg
01bc cdda0              call    print
01bf c3370              jmp     ready
```

```
                ;
                ;*************************************************
                ;*                                               *
                ;* utility subroutines for console i/o           *
                ;*                                               *
                ;*************************************************
                getchr:
                        ;read next console character to a
0lc2 0e01               mvi     c,coninp
0lc4 cd0050             call    bdos
0lc7 c9                 ret
                ;
                putchr:
                        ;write character from a to console
0lc8 0e02               mvi     c,conout
0lca 5f                 mov     e,a     ;character to send
0lcb cd0050             call    bdos    ;send character
0lce c9                 ret
                ;
                crlf:
                        ;send carriage return line feed
0lcf 3e0d               mvi     a,cr    ;carriage return
0ld1 cdc80              call    putchr
0ld4 3e0a               mvi     a,lf    ;line feed
0ld6 cdc80              call    putchr
0ld9 c9                 ret
                ;
                print:
                        ;print the buffer addressed by de until $
0lda d5                 push    d
0ldb cdcf0              call    crlf
0lde d1                 pop     d       ;new line
0ldf 0e09               mvi     c,pstring
0le1 cd0050             call    bdos    ;print the string
0le4 c9                 ret
                ;
                readcom:
                        ;read the next command line to the conbuf
0le5 116b0              lxi     d,prompt
0le8 cdda0              call    print   ;command?
0leb 0e0a               mvi     c,rstring
0led 117a0              lxi     d,conbuf
01f0 cd0050             call    bdos    ;read command line
                ;       command line is present, scan it
01f3 21000              lxi     h,0     ;start with 0000
01f6 117c0              lxi     d,conlin;command line
01f9 1a         readc:  ldax    d       ;next command character
01fa 13                 inx     d       ;to next command position
01fb b7                 ora     a       ;cannot be end of command
01fc c8                 rz
                ;       not zero, numeric?
01fd d630               sui     '0'
01ff fe0a               cpi     10      ;carry if numeric
0201 d2130              jnc     endrd
                ;       add-in next digit
0204 29                 dad     h       ;*2
0205 4d                 mov     c,l
0206 44                 mov     b,h     ;bc = value * 2
0207 29                 dad     h       ;*4
0208 29                 dad     h       ;*8
0209 09                 dad     b       ;*2 + *8 = *10
020a 85                 add     l       ;+digit
020b 6f                 mov     l,a
020c d2f90              jnc     readc   ;for another char
020f 24                 inr     h       ;overflow
0210 c3f90              jmp     readc   ;for another char
                endrd:
                ;       end of read, restore value in a
0213 c630               adi     '0'     ;command
0215 fe61               cpi     'a'     ;translate case?
```

```
0217 d8             rc
                ;   lower case, mask lower case bits
0218 e65f           ani     101$1111b
021a c9             ret
                ;
                ;***************************************************
                ;*                                                 *
                ;* string data area for console messages           *
                ;*                                                 *
                ;***************************************************
                badver:
021b 536f79         db      'sorry, you need cp/m version 2$'
                nospace:
023a 4e6f29         db      'no directory space$'
                datmsg:
024d 547970         db      'type data: $'
                errmsg:
0259 457272         db      'error, try again.$'
                prompt:
026b 4e6570         db      'next command? $'
                ;
                ;***************************************************
                ;*                                                 *
                ;* fixed and variable data area                    *
                ;*                                                 *
                ;***************************************************
027a 21     conbuf: db      conlen  ;length of console buffer
027b        consiz: ds      1       ;resulting size after read
027c        conlin: ds      32      ;length 32 buffer
0021 =      conlen  equ     $-consiz
                ;
029c                ds      32      ;16 level stack
                stack:
02bc                end
```

Again, major improvements could be made to this particular program to enhance its operation. In fact, with some work, this program could evolve into a simple data base management system. One could, for example, assume a standard record size of 128 bytes, consisting of arbitrary fields within the record. A program, called GETKEY, could be developed which first reads a sequential file and extracts a specific field defined by the operator. For example, the command

GETKEY NAMES.DAT LASTNAME 10 20

would cause GETKEY to read the data base file NAMES.DAT and extract the "LASTNAME" field from each record, starting at position 10 and ending at character 20. GETKEY builds a table in memory consisting of each particular LASTNAME field, along with its 16-bit record number location within the file. The GETKEY program then sorts this list, and writes a new file, called LASTNAME.KEY, which is an alphabetical list of LASTNAME fields with their corresponding record numbers. (This list is called an "inverted index" in information retrieval parlance.)

Rename the program shown above as QUERY, and massage it a bit so that it reads a sorted key file into memory. The command line might appear as:

QUERY NAMES.DAT LASTNAME.KEY

Instead of reading a number, the QUERY program reads an alphanumeric string which is a particular key to find in the NAMES.DAT data base. Since the LASTNAME.KEY list is sorted, you can find a particular entry quite rapidly by performing a "binary search," similar to looking up a name in the telephone book. That is, starting at both ends of the list, you examine the entry halfway in between and, if not matched, split either the upper half or the lower half for the next search. You'll quickly reach the item you're looking for (in log2(n) steps) where you'll find the corresponding record number. Fetch and display this record at the console, just as we have done in the program shown above.

At this point you're just getting started. With a little more work, you can allow a fixed grouping size which differs from the 128 byte record shown above. This is accomplished by keeping track of the record number as well as the byte offset within the record. Knowing the group size, you randomly access the record containing the proper group, offset to the beginning of the group within the record read sequentially until the group size has been exhausted.

Finally, you can improve QUERY considerably by allowing boolean expressions which compute the set of records which satisfy several relationships, such as a LASTNAME between HARDY and LAUREL, and an AGE less than 45. Display all the records which fit this description. Finally, if your lists are getting too big to fit into memory, randomly access your key files from the disk as well. One note of consolation after all this work: if you make it through the project, you'll have no more need for this manual!

System Function Summary

FUNC	FUNCTION NAME	INPUT PARAMETERS	OUTPUT RESULTS
0	System Reset	none	none
1	Console Input	none	A = char
2	Console Output	E = char	none
3	Reader Input	none	A = char
4	Punch Output	E = char	none
5	List Output	E = char	none
6	Direct Console I/O	see def	see def
7	Get I/O Byte	none	A = IOBYTE
8	Set I/O Byte	E = IOBYTE	none
9	Print String	DE = .Buffer	none
10	Read Console Buffer	DE = .Buffer	see def
11	Get Console Status	none	A = 00/FF
12	Return Version Number	none	HL = Version*
13	Reset Disk System	none	see def
14	Select Disk	E = Disk Number	see def
15	Open File	DE = .FCB	A = Dir Code
16	Close File	DE = .FCB	A = Dir Code
17	Search for First	DE = .FCB	A = Dir Code
18	Search for Next	none	A = Dir Code
19	Delete File	DE = .FCB	A = Dir Code
20	Read Sequential	DE = .FCB	A = Err Code
21	Write Sequential	DE = .FCB	A = Err Code
22	Make File	DE = .FCB	A = Dir Code
23	Rename File	DE = .FCB	A = Dir Code
24	Return Login Vector	none	HL = Login Vect*
25	Return Current Disk	none	A = Cur Disk #
26	Set DMA Address	DE = .DMA	none
27	Get Addr(Alloc)	none	HL = .Alloc
28	Write Protect Disk	none	see def
29	Get R/O Vector	none	HL = R/O Vect*
30	Set File Attributes	DE = .FCB	see def
31	Get Addr (disk parms)	none	HL = .DPB
32	Set/Get User Code	see def	see def
33	Read Random	DE = .FCB	A = Err Code
34	Write Random	DE = .FCB	A = Err Code
35	Compute File Size	DE = .FCB	r0, r1, r2
36	Set Random Record	DE = .FCB	r0, r1, r2

*Note that A = L, and B = H upon return

CHAPTER 3
CP/M EDITOR

- **Introduction to ED**
- **ED Operation**
- **Text Transfer Functions**
- **Memory Buffer Organization**
- **Memory Buffer Operation**
- **Command Strings**
- **Text Search and Alteration**
- **Source Libraries**
- **Repetitive Command Execution**
- **ED Error Conditions**
- **Summary of Control Characters**
- **Summary of ED Commands**
- **ED Text Editing Commands**

Introduction to ED

ED is the context editor for CP/M, and is used to create and alter CP/M source files. ED is initiated in CP/M by typing

$$\text{ED} \left\{ \begin{array}{l} \langle \text{filename} \rangle \\ \langle \text{filename} \rangle \cdot \langle \text{filetype} \rangle \end{array} \right\}$$

In general, ED reads segments of the source file given by ⟨filename⟩ or ⟨filename⟩ • ⟨filetype⟩ into central memory, where the file is manipulated by the operator, and subsequently written back to disk after alterations. If the source file does not exist before editing, it is created by ED and initialized to empty. The overall operation of ED is shown in Figure 1.

ED Operation

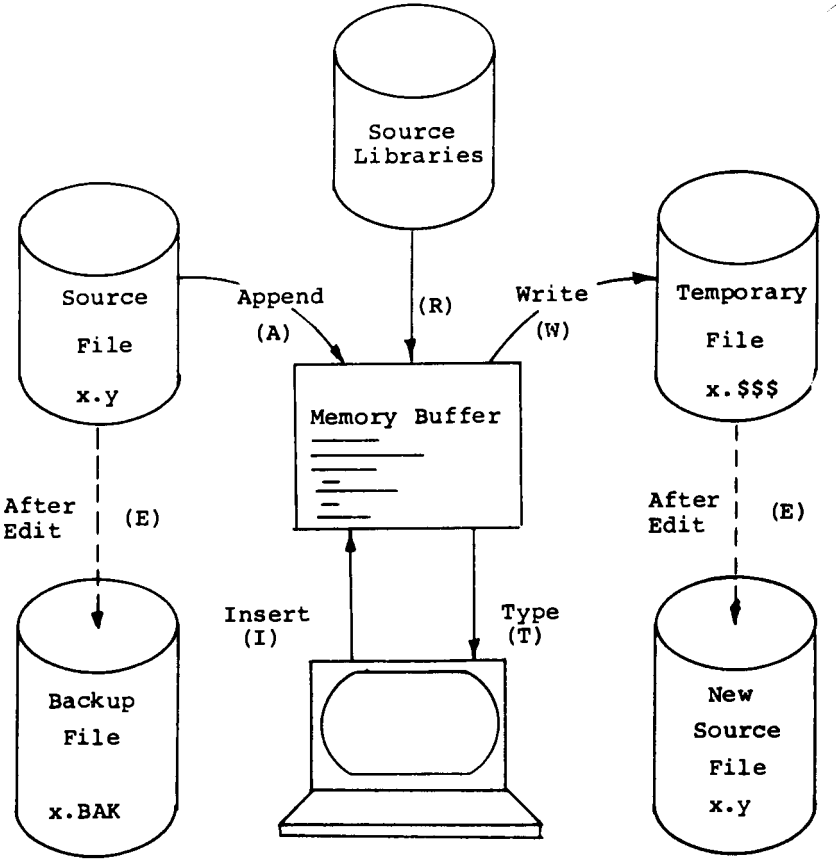
ED operates upon the source file, denoted in Figure 1 by x.y, and passes all text through a memory buffer where the text can be viewed or altered (the number of lines which can be maintained in the memory buffer varies with the line length, but has a total capacity of about 6000 characters in a 16K CP/M system). Text material which has been edited is written onto a temporary work file under command of the operator. Upon termination of the edit, the memory buffer is written to the temporary file, followed by any remaining (unread) text in the source file. The name of the original file is changed from x.y to x.BAK so that the most recent previously edited source file can be reclaimed if necessary (see the CP/M commands ERASE and RENAME). The temporary file is changed from x.$$$ to x.y which becomes the resulting edited file.

The memory buffer is logically between the source file and working file as shown in Figure 2.

Text Transfer Functions

Given that n is an integer value in the range 0 through 65535, the following ED commands transfer lines of text from the source file through the memory buffer to the temporary (and eventually final) file:

Figure 1. Overall ED Operation



Note: the ED program accepts both lower and upper case ASCII characters as input from the console. Single letter commands can be typed in either case. The U command can be issued to cause ED to translate lower case alphabetics to upper case as characters are filled to the memory buffer from the console. Characters are echoed as typed without translation, however. The -U command causes ED to revert to "no translation" mode. ED starts with an assumed -U in effect.

Figure 2. Memory Buffer Organization

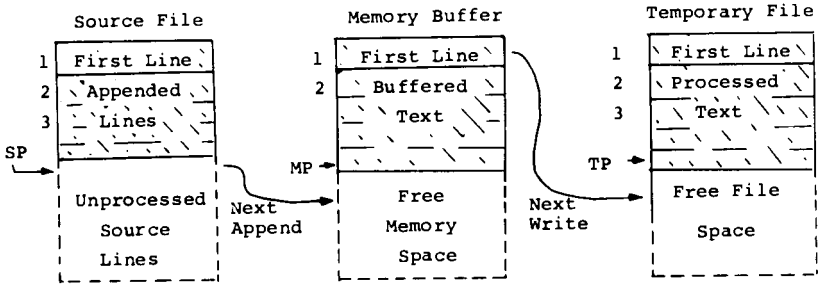
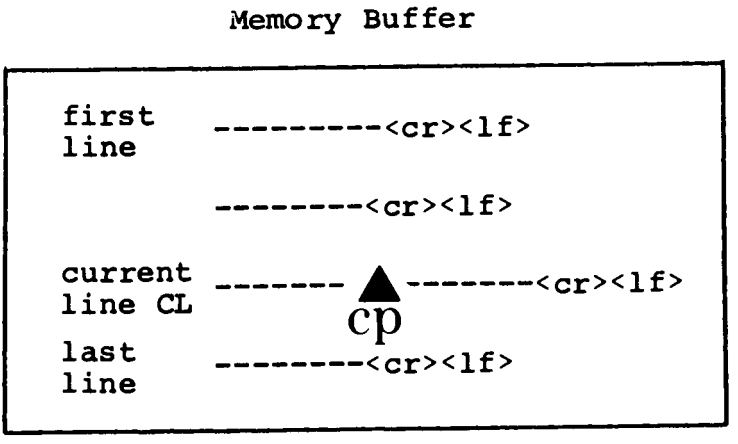


Figure 3. Logical Organization of Memory Buffer



nA⟨cr⟩* Append the next n unprocessed source lines from the source file at SP to the end of the memory buffer at MP. Increment SP and MP by n.

nW⟨cr⟩ Write the first n lines of the memory buffer to the temporary file free space. Shift the remaining lines n + 1 through MP to the top of the memory buffer. Increment TP by n.

E⟨cr⟩ End the edit. Copy all buffered text to temporary file, and copy all unprocessed source lines to the temporary file. Rename files as described previously.

H⟨cr⟩ Move to head of new file by performing automatic E command. Temporary file becomes the new source file, the memory buffer is emptied, and a new temporary file is created (equivalent to issuing an E command, followed by a reinvocation of ED using x.y as the file to edit).

O⟨cr⟩ Return to original file. The memory buffer is emptied, the temporary file is deleted, and the SP is returned to position 1 of the source file. The effects of the previous editing commands are thus nullified.

Q⟨cr⟩ Quit edit with no file alterations, return to CP/M.

There are a number of special cases to consider. If the integer n is omitted in any ED command where an integer is allowed, then 1 is assumed. Thus, the commands A and W append one line and write 1 line, respectively. In addition, if a pound sign (#) is given in the place of n, then the integer 65535 is assumed (the largest value for n which is allowed). Since most reasonably sized source files can be contained entirely in the memory buffer, the command #A is often issued at the beginning of the edit to read the entire source file to memory. Similarly, the command #W writes the entire buffer to the temporary file. Two special forms of the A and W commands are provided as a convenience. The command 0A fills the current memory buffer to at least half-full, while 0W writes lines until the buffer is at least half empty. It should also be noted that an error is issued if the memory buffer size is exceeded. The operator may then enter any command (such as W) which does not increase memory requirements. The remainder of any partial line read during the overflow will be brought into memory on the next successful append.

*⟨cr⟩represents the carriage-return key

Memory Buffer Organization

The memory buffer can be considered a sequence of source lines brought in with the A command from a source file. The memory buffer has an associated (imaginary) character pointer (CP) which moves throughout the memory buffer under command of the operator. The memory buffer appears logically as shown in Figure 3 where the dashes represent characters of the source line of indefinite length, terminated by carriage return (⟨cr⟩) and line feed (⟨lf⟩) characters, and ▲cp represents the imaginary character pointer. Note that the CP is always located ahead of the first character of the first line, behind the last character of the last line, or between two characters. The current line CL is the source line which contains the CP.

Memory Buffer Operation

Upon initiation of ED, the memory buffer is empty (i.e., CP is both ahead and behind the first and last character). The operator may either append lines (A command) from the source file, or enter the lines directly from the console with the insert command

I⟨cr⟩

ED then accepts any number of input lines, where each line terminates with a ⟨cr⟩ (the ⟨lf⟩ is supplied automatically), until a control-z (denoted by ↑z) is typed by the operator. The CP is positioned after the last character entered. The sequence

```
I⟨cr⟩
NOW IS THE⟨cr⟩
TIME FOR⟨cr⟩
ALL GOOD MEN⟨cr⟩
↑z
```

leaves the memory buffer as shown below

```
NOW IS THE⟨cr⟩⟨lf⟩
TIME FOR⟨cr⟩⟨lf⟩
ALL GOOD MEN⟨cr⟩⟨lf⟩▲cp
```

Various commands can then be issued which manipulate the CP or display source text in the vicinity of the CP. The commands shown below with a preceding n indicate that an optional unsigned value can be specified. When preceded by ±, the command can be unsigned, or have an optional preceding plus or minus sign. As before, the pound sign (#) is replaced by 65535. If an integer n is optional, but not supplied, then n = 1 is assumed. Finally, if a plus sign is optional, but none is specified, then + is assumed.

±B⟨cr⟩ move CP to beginning of memory buffer if +, and to bottom if -.

±nC⟨cr⟩ move CP by ±n characters (toward front of buffer if +), counting the ⟨cr⟩⟨lf⟩ as two distinct characters.

±nD⟨cr⟩ delete n characters ahead of CP if plus and behind CP if minus.

±nK⟨cr⟩ kill (i.e. remove) ±n lines of source text using CP as the current reference. If CP is not at the beginning of the current line when K is issued, then the characters before CP remain if + is specified, while the characters after CP remain if - is given in the command.

±nL⟨cr⟩ if n = 0, move CP to the beginning of the current line (if it is not already there). If n ≠ 0, first move the CP to the beginning of the current line, and then move it to the beginning of the line which is n lines down (if +) or up (if -). The CP will stop at the top or bottom of the memory buffer if too large a value is specified.

±nT⟨cr⟩ If n = 0 then type the contents of the current line up to CP. If n = 1 then type the contents of the current line from CP to the end of the line. If n > 1 then type the current line along with n-1 lines which follow, if + is specified. Similarly, if n > 1 and - is given, type the previous n lines, up to the CP. The break key can be depressed to abort long type-outs.

±n⟨cr⟩ equivalent to ±nLT, which moves up or down and types a single line.

Command Strings

Any number of commands can be typed contiguously (up to the capacity of the CP/M console buffer), and are executed only after the ⟨cr⟩ is typed. Thus, the operator may use the CP/M console command functions to manipulate the input command.

Rubout remove the last character

Control-X delete the entire line

Control-C re-initialize the CP/M System

Control-E return carriage for long lines without transmitting buffer (max 128 chars)

Suppose the memory buffer contains the characters shown in the previous section, with the CP following the last character of the buffer. The command strings shown below produce the results shown to the right.

Command String	Effect	Resulting Memory Buffer
B2T⟨cr⟩	move to beginning of buffer and type 2 lines: "NOW IS THE TIME FOR"	▲cp NOW IS THE⟨cr⟩⟨lf⟩ TIME FOR⟨cr⟩⟨lf⟩ ALL GOOD MEN⟨cr⟩⟨lf⟩
5C0T⟨cr⟩	move CP 5 characters and type the beginning of the line "NOW I"	NOW I ▲cp S THE ⟨cr⟩⟨lf⟩
2L-T⟨cr⟩	move two lines down and type previous line "TIME FOR"	NOW IS THE ⟨cr⟩⟨lf⟩ TIME FOR⟨cr⟩⟨lf⟩ ▲cp ALL GOOD MEN⟨cr⟩⟨lf⟩
-L#K⟨cr⟩	move up one line, delete 65535 lines which follow	NOW IS THE⟨cr⟩⟨lf⟩▲cp
I⟨cr⟩ TIME TO⟨cr⟩ INSERT⟨cr⟩ ↑z	insert two lines of text	NOW IS THE⟨cr⟩⟨lf⟩ TIME TO⟨cr⟩⟨lf⟩ INSERT⟨cr⟩⟨lf⟩▲cp
-2L#T⟨cr⟩	move up two lines, and type 65535 lines ahead of CP "NOW IS THE"	NOW IS THE⟨cr⟩⟨lf⟩▲cp TIME TO⟨cr⟩⟨lf⟩ INSERT⟨cr⟩⟨lf⟩
⟨cr⟩	move down one line and type one line "INSERT"	NOW IS THE⟨cr⟩⟨lf⟩ TIME TO⟨cr⟩⟨lf⟩▲cp INSERT ⟨cr⟩⟨lf⟩

Text Search and Alteration

ED also has a command which locates strings within the memory buffer. The command takes the form

$$nF\ c_1c_2 \ldots c_k \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

where c_1 through c_k represent the characters to match followed by either a ⟨cr⟩ or control -z.* ED starts at the current position of CP and attempts to match all k characters. The match is attempted n times, and if successful, the CP is moved directly after the character c_k. If the n matches are not successful, the CP is not moved from its initial position. Search strings can include ↑l (control-l), which is replaced by the pair of symbols ⟨cr⟩ ⟨lf⟩.

The following commands illustrate the use of the F command:

Command String	Effect	Resulting Memory Buffer
B#T⟨cr⟩	move to beginning and type entire buffer	▲cp NOW IS THE ⟨cr⟩⟨lf⟩ TIME FOR⟨cr⟩⟨lf⟩ ALL GOOD MEN⟨cr⟩⟨lf⟩
FS T⟨cr⟩	find the end of the string "S T"	NOW IS T▲cpHE⟨cr⟩⟨lf⟩
FI↑z0TT	find the next "I" and type to the CP then type the remainder of the current line: "TIME FOR"	NOW IS THE⟨cr⟩⟨lf⟩ TI▲cpME FOR⟨cr⟩⟨lf⟩ ALL GOOD MEN⟨cr⟩⟨lf⟩

An abbreviated form of the insert command is also allowed, which is often used in conjunction with the F command to make simple textual changes. The form is:

$$I\ c_1c_2 \ldots c_n\uparrow z \quad \text{or}$$
$$I\ c_1c_2 \ldots c_n\langle cr \rangle$$

where c_1 through c_n are characters to insert. If the insertion string is terminated by a ↑z, the characters c_1 through c_n are inserted directly following the CP, and the CP is moved directly after character c_n. The action is the same if the command is followed by a ⟨cr⟩ except that a ⟨cr⟩⟨lf⟩ is automatically inserted into the text following character c_n. Consider the following command sequences as examples of the F and I commands:

*The control-z is used if additional commands will be typed following the ↑z.

Command String	Effect	Resulting Memory Buffer
BITHIS IS ↑z⟨cr⟩	Insert "THIS IS" at the beginning of the text	THIS IS ▲cp NOW THE⟨cr⟩⟨lf⟩ TIME FOR ⟨cr⟩⟨lf⟩ ALL GOOD MEN⟨cr⟩⟨lf⟩
FTIME↑z-4DIPLACE↑z⟨cr⟩	find "TIME" and delete it; then insert "PLACE"	THIS IS NOW THE⟨cr⟩⟨lf⟩ PLACE ▲cp FOR⟨cr⟩⟨lf⟩ ALL GOOD MEN⟨cr⟩⟨lf⟩
3FO↑z-3D5DICHANGES↑⟨cr⟩	find third occurrence of "O" (i.e. the second "O" in GOOD), delete previous 3 characters; then insert "CHANGES"	THIS IS NOW THE⟨cr⟩⟨lf⟩ PLACE FOR⟨cr⟩⟨lf⟩ ALL CHANGES ▲cp ⟨cr⟩⟨lf⟩
-8CISOURCE⟨cr⟩	move back 8 characters and insert the line "SOURCE⟨cr⟩⟨lf⟩"	THIS IS NOW THE⟨cr⟩⟨lf⟩ PLACE FOR⟨cr⟩⟨lf⟩ ALL SOURCE⟨cr⟩⟨lf⟩ ▲cp CHANGES⟨cr⟩⟨lf⟩

ED also provides a single command which combines the F and I commands to perform simple string substitutions. The command takes the form

$$n\ S\ c_1c_2 \ldots c_k \uparrow z\ d_1d_2 \ldots d_m \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

and has exactly the same effect as applying the command string

$$F\ c_1c_2 \ldots c_k \uparrow z\text{-}kDId_1d_2 \ldots d_m \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

a total of n times. That is, ED searches the memory buffer starting at the current position of CP and successively substitutes the second string for the first string until the end of buffer, or until the substitution has been performed n times.

As a convenience, a command similar to F is provided by ED which automatically appends and writes lines as the search proceeds. The form is

$$n\ N\ c_1c_2 \ldots c_k \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

which searches the entire source file for the nth occurrence of the string $c_1c_2 \ldots c_k$ (recall that F fails if the string cannot be found in the current buffer). The operation of the N command is precisely the same as F except in the case that the string cannot be found within the current memory buffer. In this case, the entire memory contents is written (i.e., an automatic # W is issued). Input lines are then read until the buffer is at least half full, or the entire source file is exhausted. The search continues in this manner until the string has been found n times, or until the source file has been completely transferred to the temporary file.

A final line editing function, called the juxtaposition command takes the form

$$n\ J\ c_1c_2 \ldots c_k\ \uparrow z\ \ d_1d_2 \ldots d_m \uparrow z\ \ e_1e_2 \ldots e_q \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

with the following action applied n times to the memory buffer: search from the current CP for the next occurrence of the string $c_1c_2 \ldots c_k$. If found, insert the string $d_1,d_2 \ldots,d_m$, and move CP to follow d_m. Then delete all characters following CP up to (but not including) the string $e_1, e_2, \ldots e_q$, leaving CP directly after d_m. If $e_1, e_e, \ldots e_q$ cannot be found, then no deletion is made. If the current line is

$$\underset{cp}{\blacktriangle} \text{NOW IS THE TIME}\langle cr \rangle\langle lf \rangle$$

Then the command

$$J W\ \uparrow z \text{WHAT} \uparrow z \uparrow l\ \langle cr \rangle$$

Results in

$$\text{NOW WHAT}\ \underset{cp}{\blacktriangle}\ \langle cr \rangle\langle lf \rangle$$

(Recall that ↑l represents the pair ⟨cr⟩⟨lf⟩ in search and substitution strings).

It should be noted that the number of characters allowed by ED in the F, S, N, and J commands is limited to 100 symbols.

Source Libraries

ED also allows the inclusion of source libraries during the editing process with the R command. The form of this command is

R $f_1f_2 \ldots f_n$↑z or

R $f_1f_2 \ldots f_n$⟨cr⟩

where $f_1f_2 \ldots f_n$ is the name of a source file on the disk with an assumed filetype of 'LIB'. ED reads the specified file, and places the characters into the memory buffer after CP, in a manner similar to the I command. Thus, if the command

RMACRO⟨cr⟩

is issued by the operator, ED reads from the file MACRO.LIB until the end-of-file, and automatically inserts the characters into the memory buffer.

Repetitive Command Execution

The macro command M allows the ED user to group ED commands together for repeated evaluation. The M command takes the form:

$$n\ M\ c_1c_2 \ldots c_k \left\{ \begin{matrix} \langle cr \rangle \\ \uparrow z \end{matrix} \right\}$$

where $c_1c_2 \ldots c_k$ represent a string of ED commands, not including another M command. ED executes the command string n times if n⟩1. If n = 0 or 1, the command string is executed repetitively until an error condition is encountered (e.g., the end of the memory buffer is reached with an F command).

As an example, the following macro changes all occurrences of GAMMA to DELTA within the current buffer, and types each line which is changed:

MFGAMMA↑z-5DIDELTA↑z0TT⟨cr⟩

or equivalently

MSGAMMA↑zDELTA↑z0TT⟨cr⟩

ED Error Conditions

On error conditions, ED prints the last character read before the error, along with an error indicator:

? unrecognized command

>	memory buffer full (use one of the commands D, K, N, S, or W to remove characters), F, N, or S strings too long.
#	cannot apply command the number of times specified (e.g., in F command)
O	cannot open LIB file in R command

Cyclic redundancy check (CRC) information is written with each output record under CP/M in order to detect errors on subsequent read operations. If a CRC error is detected, CP/M will type

PERM ERR DISK d

where d is the currently selected drive (A, B, . . .). The operator can choose to ignore the error by typing any character at the console (in this case, the memory buffer data should be examined to see if it was incorrectly read), or the user can reset the system and reclaim the backup file, if it exists. The file can be reclaimed by first typing the contents of the BAK file to ensure that it contains the proper information:

TYPE x.BAK⟨cr⟩

where x is the file being edited. Then remove the primary file:

ERA x.y⟨cr⟩

and rename the BAK file:

REN x.y=x.BAK⟨cr⟩

The file can then be re-edited, starting with the previous version.

Summary of Control Characters

The following table summarizes the Control characters and commands available in ED:

Control Character	Function
↑c	system reboot
↑e	physical ⟨cr⟩⟨lf⟩ (not actually entered in command)

↑i	logical tab (cols 1, 8, 15, . . .)
↑l	logical ⟨cr⟩⟨lf⟩ in search and substitute strings
↑x	line delete
↑z	string terminator
rubout	character delete
break	discontinue command (e.g., stop typing)

Summary of ED Commands

Command	Function
nA	append lines
± B	begin bottom of buffer
± nC	move character positions
± nD	delete characters
E	end edit and close files (normal end)
nF	find string
H	end edit, close and reopen files
I	insert characters
nJ	place strings in juxtaposition
± nK	kill lines
± nL	move down/up lines
nM	macro definition
nN	find next occurrence with autoscan

O	return to original file
± nP	move and print pages
Q	quit with no file changes
R	read library file
nS	substitute strings
± nT	type lines
_ U	translate lower to upper case if U, no translation if -U
nW	write lines
nZ	sleep
± n⟨cr⟩	move and type (± nLT)

ED Text Editing Commands

The ED context editor contains a number of commands which enhance its usefulness in text editing. The improvements are found in the addition of line numbers, free space interrogation, and improved error reporting.

The context editor issued with CP/M produces absolute line number prefixes when the "V" (Verify Line Numbers) command is issued. Following the V command, the line number is displayed ahead of each line in the format:

nnnnn:

where nnnnn is an absolute line number in the range 1 to 65535. If the memory buffer is empty, or if the current line is at the end of the memory buffer, then nnnnn appears as 5 blanks.

The user may reference an absolute line number by preceding any command by a number followed by a colon, in the same format as the line number display. In this case, the ED program moves the current line reference to the absolute line number, if the line exists in the current memory buffer. Thus the command

345:T

is interpreted as "move to absolute line 345, and type the line." Note that absolute line numbers are produced only during the editing process, and are not recorded with the file. In particular, the line numbers will change following a deleted or expanded section of text.

The user may also reference an absolute line number as a backward or forward distance from the current line by preceding the absolute line number by a colon. Thus, the command

:400T

is interpreted as "type from the current line number through the line whose absolute number is 400." Combining the two line reference forms, the command

345::400T

for example, is interpreted as "move to absolute line 345, then type through absolute line 400." Note that absolute line references of this sort can precede any of the standard ED commands.

A special case of the V command, "0V," prints the memory buffer statistics in the form:

free/total

where "free" is the number of free bytes in the memory buffer (in decimal), and "total" is the size of the memory buffer.

ED also includes a "block move" facility implemented through the "X" (Xfer) command. The form

nX

transfers the next n lines from the current line to a temporary file called

X$$$$$$$.LIB

which is active only during the editing process. In general, the user can reposition the current line reference to any portion of the source file and transfer lines to the temporary file. The transferred lines accumulate one after another in this file, and can be retrieved by simply typing:

which is the trivial case of the library read command. In this case, the entire transferred set of lines is read into the memory buffer. Note that the X command does not remove the transferred lines from the memory buffer, although a K command can be used directly after the X, and the R command does not empty the transferred line file. That is, given that a set of lines has been transferred with the X command, they can be re-read any number of times back into the source file. The command

0X

is provided, however, to empty the transferred line file.

Note that upon normal completion of the ED program through Q or E, the temporary LIB file is removed. If ED is aborted through Control-C, the LIB file will exist if lines have been transferred, but will generally be empty (a subsequent ED invocation will erase the temporary file).

Due to common typographical errors, ED requires several potentially disastrous commands to be typed as single letters, rather than in composite commands. The commands

E (end), H (head), O (original), Q (quit)

must be typed as single letter commands.

ED also prints error messages in the form

BREAK "x" AT c

where x is the error character, and c is the command where the error occurred.

CHAPTER 4
CP/M Assembler

- Introduction
- Program Format
- Forming the Operand
  - Labels
  - Numeric Constants
  - Reserved Words
  - String Constants
  - Arithmetic and Logical Operators
  - Precedence of Operators
- Assembler Directives
  - The ORG Directive
  - The END Directive
  - The EQU Directive
  - The Set Directive
  - The IF and ENDIF Directives
  - The DB Directive
  - The DW Directive
- Operation Codes
  - Jumps, Calls and Returns
  - Immediate Operand Instructions
  - Increment and Decrement Instructions
  - Data Movement Instructions
  - Arithmetic Logic Unit Operations
  - Control Instructions
- Error Messages
- A Sample Session

Introduction

The CP/M assembler reads assembly language source files from the diskette, and produces 8080 machine language in Intel hex format. The CP/M assembler is initiated by typing

ASM filename

or

ASM filename.parms

In both cases, the assembler assumes there is a file on the diskette with the name

filename.ASM

which contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form allows parameters to be passed to the assembler to control source file access and hex and print file destinations.

In either case, the CP/M assembler loads, and prints the message

CP/M ASSEMBLER VER n.n

where n.n is the current version number. In the case of the first command, the assembler reads the source file with assumed file type "ASM" and creates two output files.

filename.HEX

and

filename.PRN

The "HEX" file contains the machine code corresponding to the original program in Intel hex format, and the "PRN" file contains an annotated listing showing generated machine code, error flags, and source lines. If errors occur during translation, they will be listed in the PRN file as well as at the console.

The second command form can be used to redirect input and output files from their defaults. In this case, the "parms" portion of the command is a three letter group which specifies the origin of the source file, the destination of the hex file, and the destination of the print file. The form is

filename.p1p2p3

where p1, p2, and p3 are single letters

p1: A,B, ..., Y	designates the disk name which contains the source file
p2: A,B, ..., Y	designates the disk name which will receive the hex file
Z	skips the generation of the hex file
p3: A,B, ..., Y	designates the disk name which will receive the print file
X	places the listing at the console
Z	skips generation of the print file

Thus, the command

```
ASM  X.AAA
```

indicates that the source file (X.ASM) is to be taken from disk A, and that the hex (X.HEX) and the print (X.PRN) files are to be created also on disk A. This form of the command is implied if the assembler is run from disk A. That is, given that the operator is currently addressing disk A, the above command is equivalent to

```
ASM X
```

The command

```
ASM X.ABX
```

indicates that the source file is to be taken from disk A, the hex file is placed on disk B, and the listing file is to be sent to the console. The command

```
ASM X.BZZ
```

takes the source file from disk B, and skips the generation of the hex and print files. (This command is useful for fast execution of the assembler to check program syntax.)

The source program format is compatible with both the Intel 8080 assembler (macros are not currently implemented in the CP/M assembler, however), as well as the Processor Technology Software Package #1 assembler. That is, the CP/M assembler accepts source programs written in either format. There are certain extensions in the CP/M assembler which make it somewhat easier to use. These extensions are described below.

Program Format

An assembly language program acceptable as input to the assembler consists of a sequence of statements of the form

```
line#   label   operation   operand   ;comment
```

where any or all of the fields may be present in a particular instance. Each assembly language statement is terminated with a carriage return and line feed (the line feed is inserted automatically by the ED program), or with the character "!" which is treated as an end-of-line by the assembler (thus, multiple assembly language statements can be written on the same physical line if separated by exclamation symbols).

The line# is an optional decimal integer value representing the source program line number, which is allowed on any source line to maintain compatibility with the Processor Technology format. In general, these line numbers will be inserted if a line-oriented editor is used to construct the original program, and thus ASM ignores this field if present.

The label field takes the form

```
        identifier
```

or

```
        identifier:
```

and is optional, except where noted in particular statement types. The identifier is a sequence of alphanumeric characters (alphabetics and numbers), where the first character is alphabetic. Identifiers can be freely used by the programmer to label elements such as program steps and assembler directives, but cannot exceed 16 characters in length. All characters are significant in an identifier, except for the embedded dollar symbol ($) which can be used to improve readability of the name. Further, all lower case alphabetics are treated as if they were upper case. Note that the ":" following the identifier in a label is optional (to maintain compatibility between Intel and Processor Technology). Thus, the following are all valid instances of labels

```
x          x y        long$name
x:         y x 1:     longer$named$data:
X 1 Y 2    X 1 x 2    x234$5678$9012$3456:
```

The operation field contains either an assembler directive, or pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation codes are described below.

The operand field of the statement, in general, contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. Again, the complete details of properly formed expressions are given below.

The comment field contains arbitrary characters following the ";" symbol until the next real or logical end-of-line. These characters are read, listed, and otherwise ignored by the assembler. In order to maintain compatibility with the Processor Technology assembler, the CP/M assembler also treats statements which begin with a "*" in column one as comment statements, which are listed and ignored in the assembly process. Note that the Processor Technology assembler has the side effect in its operation of ignoring the characters after the operand field has been scanned. This causes an ambiguous situation when attempting to be compatible with Intel's language, since arbitrary expressions are allowed in this case. Hence, programs which use this side effect to introduce comments, must be edited to place a ";" before these fields in order to assemble correctly.

The assembly language program is formulated as a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

Forming the Operand

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression must produce a 16-bit value during the assembly. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance are given with the individual instructions.

Labels

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g, a MOV instruction, or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label

is given the value which results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, its value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

Numeric Constants

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are

B	binary constant (base 2)
O	octal constant (base 8)
Q	octal constant (base 8)
D	decimal constant (base 10)
H	hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers since the letter O is easily confused with the digit 0. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is binary constants must be composed of 0 and 1 digits, octal constants can contain digits in the range 0 - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading 0 will always suffice). A constant composed in this manner must evaluate to a binary number which can be contained within a 16-bit counter, otherwise it is truncated on the right by the assembler. Similar to identifiers, imbedded "$" are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants

```
1234    1234D   1100B   1111$0000$1111$0000B
1234H   0FFEH   3377O   33$77$22Q
3377o   0fe3h   1234d   0ffffh
```

Reserved Words

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8080 registers are given below, which, when encountered, produce the value shown to the right.

A	7
B	0
C	1
D	2
E	3
H	4
L	5
M	6
SP	6
PSW	6

(Again, lower case names have the same values as their upper case equivalents.) Machine instructions can also be used in the operand field, and evaluate to their internal codes. In the case of instructions which require operands, where the specific operand becomes a part of the binary bit pattern of the instruction (e.g, MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g, MOV produces 40H).

When the symbol "$" occurs in the operand field (not imbedded within identifiers and numeric constants) its value becomes the address of the next instruction to generate, not including the instruction contained within the current logical line.

String Constants

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols ('). All strings must be fully contained within the current physical line (thus allowing "!" symbols within strings), and must not exceed 64 characters in length. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes ''), which becomes a single apostrophe when read by the assembler. In most cases, the string length is restricted to either one or two characters (the DB pseudo operation is an exception), in which case the string becomes an 8 or 16 bit value, respectively. Two character strings become a 16-bit constant, with the second character as the low order byte, and the first character as the high order byte.

The value of a character is its corresponding ASCII code. There is no case translation within strings, and thus both upper and lower case characters can be represented. Note however, that only graphic (printing) ASCII characters are allowed within strings. Valid strings are

```
'A'     'AB'    'ab'    'c'
''''    'a'''   ''''''  ''''''
'Walla Walla Wash.'
'She said "Hello" to me.'
'I said "Hello" to her.'
```

Arithmetic and Logical Operators

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are

a + b	unsigned arithmetic sum of a and b
a - b	unsigned arithmetic difference between a and b
+ b	unary plus (produces b)
- b	unary minus (identical to 0 - b)
a * b	unsigned magnitude multiplication of a and b
a / b	unsigned magnitude division of a by b
a MOD b	remainder after a / b
NOT b	logical inverse of b (all 0's become 1's, 1's become 0's), where b is considered a 16-bit value
a AND b	bit-by-bit logical and of a and b
a OR b	bit-by-bit logical or of a and b
a XOR b	bit-by-bit logical exclusive or of a and b
a SHL b	the value which results from shifting a to the left by an amount b, with zero fill
a SHR b	the value which results from shifting a to the right by an amount b, with zero fill

In each case, a and b represent simple operands (labels, numeric constants, reserved words, and one or two character strings), or fully enclosed parenthesized subexpressions such as

```
10+20          10h+37Q          L1 / 3   (L2+4) SHR 3
('a' and 5fh) + '0'          ('B'+B) OR (PSW+M)
(1+(2+c)) shr (A-(B+1))
```

Note that all computations are performed at assembly time as 16-bit unsigned operations. Thus, -1 is computed as 0-1 which results in the value 0ffffh (i.e., all 1's). The resulting expression must fit the operation code in which it is used. If, for example, the expression is used in a ADI (add

immediate) instruction, then the high order eight bits of the expression must be zero. As a result, the operation "ADI -1" produces an error message (-1 becomes 0ffffh which cannot be represented as an 8 bit value), while "ADI (-1) AND 0FFH" is accepted by the assembler since the "AND" operation zeroes the high order bits of the expression.

Precedence of Operators

As a convenience to the programmer, ASM assumes that operators have a relative precedence of application which allows the programmer to write expressions without nested levels of parentheses. The resulting expression has assumed parentheses which are defined by the relative precedence. The order of application of operators in unparenthesized expressions is listed below. Operators listed first have highest precedence (they are applied first in an unparenthesized expression), while operators listed last have lowest precedence. Operators listed on the same line have equal precedence, and are applied from left to right as they are encountered in an expression

```
* / MOD SHL SHR
- +
NOT
AND
OR XOR
```

Thus, the expressions shown to the left below are interpreted by the assembler as the fully parenthesized expressions shown to the right below

```
a * b + c                    (a * b) + c
a + b * c                    a + (b * c)
a MOD b * c SHL d            ((a MOD b) * c) SHL d
a OR b AND NOT c + d SHL e   a OR (b AND (NOT (c + (d SHL e))))
```

Balanced parenthesized subexpressions can always be used to override the assumed parentheses, and thus the last expression above could be rewritten to force application of operators in a different order as

```
(a OR b) AND (NOT c) + d SHL e
```

resulting in the assumed parentheses

```
(a OR b) AND ((NOT c) + (d SHL e))
```

Note that an unparenthesized expression is well-formed only if the expression which results from inserting the assumed parentheses is well-formed.

Assembler Directives

Assembler directives are used to set labels to specific values during the assembly, perform conditional assembly, define storage areas, and specify starting addresses in the program. Each assembler directive is denoted by a "pseudo operation" which appears in the operation field of the line. The acceptable pseudo operations are

ORG	set the program or data origin
END	end program, optional start address
EQU	numeric "equate"
SET	numeric "set"
IF	begin conditional assembly
ENDIF	end of conditional assembly
DB	define data bytes
DW	define data words
DS	define data storage area

The ORG Directive

The ORG statement takes the form

```
label    ORG    expression
```

where "label" is an optional program label, and expression is a 16-bit expression, consisting of operands which are defined previous to the ORG statement. The assembler begins machine code generation at the location specified in the expression. There can be any number of ORG statements within a particular program, and there are no checks to ensure that the programmer is not defining overlapping memory areas. Note that most programs written for the CP/M system begin with an ORG statement of the form

```
ORG 100H
```

which causes machine code generation to begin at the base of the CP/M transient program area. If a label is specified in the ORG statement, then the label is given the value of the expression (this label can then be used in the operand field of other statements to represent this expression).

The END Directive

The END statement is optional in an assembly language program, but if it is present it must be the last statement (all subsequent statements are ignored in the assembly). The two forms of the END directive are

```
label    END
label    END    expression
```

where the label is again optional. If the first form is used, the assembly process stops, and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated, and becomes the program starting address (this starting address is included in the last record of the Intel formatted machine code "hex" file which results from the assembly). Thus, most CP/M assembly language programs end with the statement

```
END 100H
```

resulting in the default starting address of 100H (beginning of the transient program area).

The EQU Directive

The EQU (equate) statement is used to set up synonyms for particular numeric values. The form is

```
label    EQU    expression
```

where the label must be present, and must not label any other statement. The assembler evaluates the expression, and assigns this value to the identifier given in the label field. The identifier is usually a name which describes the value in a more human-oriented manner. Further, this name is used throughout the program to "parameterize" certain functions. Suppose for example, that data received from a Teletype appears on a particular input port, and data is sent to the Teletype through the next output port in sequence. The series of equate statements could be used to define these ports for a particular hardware environment

```
TTYBASE EQU 10H          ;BASE PORT NUMBER FOR TTY
TTYIN   EQU TTYBASE      ;TTY DATA IN
TTYOUT  EQU TTYBASE+1;TTY DATA OUT
```

At a later point in the program, the statements which access the Teletype could appear as

```
IN    TTYIN       ;READ TTY DATA TO REG - A
...
OUT TTYOUT        ;WRITE DATA TO TTY FROM REG-A
```

making the program more readable than if the absolute I/O ports had been used. Further, if the hardware environment is redefined to start the Teletype communications ports at 7FH instead of 10H, the first statement need only be changed to

```
TTYBASE EQU 7FH          ;BASE PORT NUMBER FOR TTY
```

and the program can be reassembled without changing any other statements.

The SET Directive

The SET statement is similar to the EQU, taking the form

```
label    SET    expression
```

except that the label can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, the EQU statement defines a label with a single value, while the SET statement defines a value which is valid from the current SET statement to the point where the label occurs on the next SET statement. The use of the SET is similar to the EQU statement, but is used most often in controlling conditional assembly.

The IF and ENDIF Directives

The IF and ENDIF statements define a range of assembly language statements which are to be included or excluded during the assembly process. The form is

```
IF   expression
statement#1
statement#2
   ...
statement#n
ENDIF
```

Upon encountering the IF statement, the assembler evaluates the expression following the IF (all operands in the expression must be defined ahead of the IF statement). If the expression evaluates to a non-zero value, then statement#1 through statement#n are assembled; if the expression

evaluates to zero, then the statements are listed but not assembled. Conditional assembly is often used to write a single "generic" program which includes a number of possible run-time environments, with only a few specific portions of the program selected for any particular assembly. The following program segments for example, might be part of a program which communicates with either a Teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins

```
TRUE     EQU  0FFFFH       ;DEFINE VALUE OF TRUE
FALSE    EQU  NOT TRUE     ;DEFINE VALUE OF FALSE
;
TTY      EQU  TRUE         ;TRUE IF TTY, FALSE IF CRT
;
TTYBASE  EQU  10H          ;BASE OF TTY I/O PORTS
CRTBASE  EQU  20H          ;BASE OF CRT I/O PORTS
         IF   TTY          ;ASSEMBLE RELATIVE TO
                           TTYBASE
CONIN    EQU  TTYBASE      ;CONSOLE INPUT
CONOUT   EQU  TTYBASE+1;CONSOLE OUTPUT
         ENDIF
;
         IF   NOT TTY      ;ASSEMBLE RELATIVE TO
                           CRTBASE
CONIN    EQU  CRTBASE      ;CONSOLE INPUT
CONOUT   EQU  CRTBASE+1;CONSOLE OUTPUT
         ENDIF
         ...
         IN   CONIN        ;READ CONSOLE DATA
         ...
         OUT  CONOUT       ;WRITE CONSOLE DATA
```

In this case, the program would assemble for an environment where a Teletype is connected, based at port 10H. The statement defining TTY could be changed to

```
TTY      EQU  FALSE
```

and, in this case, the program would assemble for a CRT based at port 20H.

The DB Directive

The DB directive allows the programmer to define initialized storage areas in single precision (byte) format. The statement form is

```
label    DB    e#1, e#2, ..., e#n
```

where e#1 through e#n are either expressions which evaluate to 8-bit values (the high order eight bits must be zero), or are ASCII strings of length no greater than 64 characters. There is no practical restriction on the number of expressions included on a single source line. The expressions are evaluated and placed sequentially into the machine code file following the last program address generated by the assembler. String characters are similarly placed into memory starting with the first character and ending with the last character. Strings of length greater than two characters cannot be used as operands in more complicated expressions (i.e., they must stand alone between the commas). Note that ASCII characters are always placed in memory with the parity bit reset (0). Further, recall that there is no translation from lower to upper case within strings. The optional label can be used to reference the data area throughout the remainder of the program. Examples of valid DB statements are

```
data:   DB  0,1,2,3,4,5
        DB  data and 0ffh,5,377Q,1+2+3+4
signon: DB  'please type your name',cr,lf,0
        DB  'AB' SHR 8, 'C', 'DE' AND 7FH
```

The DW Directive

The DW statement is similar to the DB statement except double precision (two byte) words of storage are initialized. The form is

```
label   DW      e#1, e#2, ..., e#n
```

where e#1 through e#n are expressions which evaluate to 16-bit results. Note that ASCII strings of length one or two characters are allowed, but strings longer than two characters disallowed. In all cases, the data storage is consistent with the 8080 processor: the least significant byte of the expression is stored first in memory, followed by the most significant byte. Examples are

```
doub:   DW      0ffefh,doub+4,signon-$,255+255
        DW      'a', 5, 'ab', 'CD', 6 shl 8 or 11b
```

The DS Directive

The DS statement is used to reserve an area of uninitialized memory, and takes the form

```
label   DS      expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the statement

```
label:  EQU    $  ;LABEL VALUE IS CURRENT CODE LOCATION
        ORG    $ + expression   ;MOVE PAST RESERVED AREA
```

Operation Codes

Assembly language operation codes form the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual *8080 Assembly Language Programming Manual.* Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. The individual operators are listed briefly in the following sections for completeness, although it is understood that the Intel manuals should be referenced for exact operator details. In each case,

e3 — represents a 3-bit value in the range of 0-7 which can be one of the predefined registers A, B, C, D, E, H, L, M, SP, or PSW.

e8 — represents an 8-bit value in the range 0–255

e16 — represents a 16-bit value in the range 0–65535

which can themselves be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. These cases will be noted as they are encountered.

In the sections which follow, each operation code is listed in its most general form, along with a specific example, with a short explanation and special restrictions.

Jumps, Calls and Returns

The Jump, Call and Return instructions allow several different forms which test the condition flags set in the 8080 microcomputer CPU. The forms are

```
JMB     e16     JMP   L1        Jump unconditionally to label
JNZ     e16     JMP   L2        Jump on non zero condition to label
JZ      e16     JMP   100H      Jump on zero condition to label
JNC     e16     JNC   L1+4      Jump no carry to label
JC      e16     JC    L3        Jump on carry to label
JPO     e16     JPO   $+8       Jump on parity odd to label
JPE     e16     JPE   L4        Jump on even parity to label
JP      e16     JP    GAMMA     Jump on positive result to label
```

JM	e16	JM	al	Jump on minus to label
CALL	e16	CALL	S1	Call subroutine unconditionally
CNZ	e16	CNZ	S2	Call subroutine if non zero flag
CZ	e16	CZ	100H	Call subroutine on zero flag
CNC	e16	CNC	S1+4	Call subroutine if no carry set
CC	e16	CC	S3	Call subroutine if carry set
CPO	e16	CPO	$+8	Call subroutine if parity odd
CPE	e16	CPE	S4	Call subroutine if parity even
CP	e16	CP	GAMMA	Call subroutine if positive result
CM	e16	CM	b1$c2	Call subroutine if minus flag
RST	e3	RST	0	Programmed "restart," equivalent to CALL 8*e3, except one byte call
RET				Return from subroutine
RNZ				Return if non zero flag set
RZ				Return if zero flag set
RNC				Return if no carry
RC				Return if carry flag set
RPO				Return if parity is odd
RPE				Return if parity is even
RP				Return if positive result
RM				Return if minus flag is set

Immediate Operand Instructions

Several instructions are available which load single or double precision registers, or single precision memory cells, with constant values, along with instructions which perform immediate arithmetic or logical operations on the accumulator (register A).

MVI e3,e8	MVI	B,255	Move immediate data to register A, B, C, D, E, H, L, or M (memory)
ADI e8	ADI	1	Add immediate operand to A without carry
ACI e8	ACI	0FFH	Add immediate operand to A with carry
SUI e8	SUI	L + 3	Subtract from A without borrow (carry)
SBI e8	SBI	L AND 11B	Subtract from A with borrow (carry)
ANI e8	ANI	$ AND 7FH	Logical "and" A with immediate data
XRI e8	XRI	1111$0000B	"Exclusive or" A with immediate data
ORI e8	ORI	L AND 1+1	Logical "or" A with immediate data

CPI e8	CPI 'a'	Compare A with immediate data (same as SUI except register A not changed)
LXI e3,e16	LXI B,100H	Load extended immediate to register pair (e3 must be equivalent to B,D,H, or SP)

Increment and Decrement Instructions

Instructions are provided in the 8080 repertoire for incrementing or decrementing single and double precision registers. The instructions are

INR e3	INR E	Single precision increment register (e3 produces one of A, B, C, D, E, H, L, M)
DCR e3	DCR A	Single precision decrement register (e3 produces one of A, B, C, D, E, H, L, M)
INX e3	INX SP	Double precision increment register pair (e3 must be equivalent to B,D,H, or SP)
DCX e3	DCX B	Double precision decrement register pair (e3 must be equivalent to B,D,H, or SP)

Data Movement Instructions

Instructions which move data from memory to the CPU and from CPU to memory are given below

MOV e3,e3	MOV A,B	Move data to leftmost element from rightmost element (e3 produces one of A, B, C, D, E, H, L, or M). MOV M,M is disallowed
LDAX e3	LDAX B	Load register A from computed address (e3 must produce either B or D)
STAX e3	STAX D	Store register A to computed address (e3 must produce either B or D)
LHLD e16	LHLD L1	Load HL direct from location e16 (double precision load to H and L)
SHLD e16	SHLD L5+x	Store HL direct to location e16 (double precision store from H and L to memory)

LDA e16	LDA Gamma	Load register A from address e16
STA e16	STA X3-5	Store register A into memory at e16
POP e3	POP PSW	Load register pair from stack, set SP (e3 must produce one of B, D, H, or PSW)
PUSH e3	PUSH B	Store register pair into stack, set SP (e3 must produce one of B, D, H, or PSW)
IN e8	IN 0	Load register A with data from port e8
OUT e8	OUT 255	Send data from register A to port e8
XTHL		Exchange data from top of stack with HL
PCHL		Fill program counter with data from HL
SPHL		Fill stack pointer with data from HL
XCHG		Exchange DE pair with HL pair

Arithmetic Logic Unit Operations

Instructions which act upon the single precision accumulator to perform arithmetic and logic operations are

ADD e3	ADD B	Add register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L)
ADC e3	ADC L	Add register to A with carry, e3 as above
SUB e3	SUB H	Subtract reg e3 from A without carry, e3 is defined as above
SBB e3	SBB 2	Subtract register e3 from A with carry, e3 defined as above
ANA e3	ANA 1+1	Logical "and" reg with A, e3 as above
XRA e3	XRA A	"Exclusive or" with A, e3 as above
ORA e3	ORA B	Logical "or" with A, e3 defined as above
CMP e3	CMP H	Compare register with A, e3 as above
DAA		Decimal adjust register A based upon last arithmetic logic unit operation
CMA		Complement the bits in register A

STC		Set the carry flag to 1
CMC		Complement the carry flag
RLC		Rotate bits left, (re)set carry as a side effect (high order A bit becomes carry)
RRC		Rotate bits right, (re)set carry as side effect (low order A bit becomes carry)
RAL		Rotate carry/A register to left (carry is involved in the rotate)
RAR		Rotate carry/A register to right (carry is involved in the rotate)
DAD e3	DAD B	Double precision add register pair e3 to HL (e3 must produce B, D, H, or SP)

Control Instructions

The four remaining instructions are categorized as control instructions, and are listed below

HLT	Halt the 8080 processor
DI	Disable the interrupt system
EI	Enable the interrupt system
NOP	No operation

Error Messages

When errors occur within the assembly language program, they are listed as single character flags in the leftmost position of the source listing. The line in error is also echoed at the console so that the source listing need not be examined to determine if errors are present. The error codes are

D	Data error: element in data statement cannot be placed in the specified data area
E	Expression error: expression is ill-formed and cannot be computed at assembly time
L	Label error: label cannot appear in this context (may be duplicate label)
N	Not implemented: features which will appear in future ASM versions (e.g., macros) are recognized, but flagged in this version

O	Overflow: expression is too complicated (i.e., too many pending operators) to compute; simplify it
P	Phase error: label does not have the same value on two subsequent passes through the program
R	Register error: the value specified as a register is not compatible with the operation code
V	Value error: operand encountered in expression is improperly formed

Several error messages are printed which are due to terminal error conditions

NO SOURCE FILE PRESENT	The file specified in the ASM command does not exist on disk
NO DIRECTORY SPACE	The disk directory is full; erase files which are not needed, and retry
SOURCE FILE NAME ERROR	Improperly formed ASM file name (e.g., it is specified with "?" fields)
SOURCE FILE READ ERROR	Source file cannot be read properly by the assembler, execute a TYPE to determine the point of error
OUTPUT FILE WRITE ERROR	Output files cannot be written properly, most likely cause is a full disk; erase and retry
CANNOT CLOSE FILE	Output file cannot be closed, check to see if disk is write protected

A Sample Session

The following session shows interaction with the assembler and debugger in the development of a simple assembly language program.

```
ASM SORT      Assemble SORT.ASM

CP/M ASSEMBLER - VER 1.0

015C  next free address
003H USE FACTOR  % of table used 00 to FF (hexadecimal)
END OF ASSEMBLY

DIR SORT.*

SORT      ASM  source file
SORT      BAK  backup from last edit
SORT      PRN  print file (contains tab characters)
SORT      HEX  machine code file
A>TYPE SORT.PRN
```

Source line

machine code location

generated machine code

truncated

```
                        ;       SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
                        ;       START AT THE BEGINNING OF THE TRANSIENT PROGRAM AR
0100                    SORT:   ORG     100H
0100 214601             SORT:   LXI     H,SW    ;ADDRESS SWITCH TOGGLE
0103 3601               MVI     M,1     ;SET TO 1 FOR FIRST ITERATION
0105 214701             LXI     H,I     ;ADDRESS INDEX
0108 3600               MVI     M,0     ;I = 0
                        ;
                        ;       COMPARE I WITH ARRAY SIZE
010A 7E                 COMP:   MOV     A,M     ;A REGISTER = I
010B FE09               CPI     N-1     ;CY SET IF I < (N-1)
010D D21981             JNC     CONT    ;CONTINUE IF I <= (N-2)
                        ;
                        ;       END OF ONE PASS THROUGH DATA
0110 214601             LXI     H,SW    ;CHECK FOR ZERO SWITCHES
0113 7EB7C20001         MOV A,M! ORA A! JNZ SORT ;END OF SORT IF SW=0
                        ;
0118 FF                 RST 7           ;GO TO THE DEBUGGER INSTEAD OF RE
                        ;
                        ;       CONTINUE THIS PASS
                        ;       ADDRESSING I, SO LOAD AV(I) INTO REGISTERS
0119 5F16002148CONT:    MOV E,A! MVI D,0! LXI H,AV! DAD D! DAD D
0121 4E792346           MOV C,M! MOV A,C! INX H! MOV B,M
                        ;       LOW ORDER BYTE IN A AND C, HIGH ORDER BYTE IN B
                        ;
                        ;       MOV H AND L TO ADDRESS AV(I+1)
0125 23                 INX     H
                        ;
                        ;       COMPARE VALUE WITH REGS CONTAINING AV(I)
0126 9657782396E        SUB M! MOV D,A! MOV A,B! INX H! SBB M    ;SUBTRACT
                        ;
                        ;       BORROW SET IF AV(I+1) > AV(I)
0128 DA3F01             JC      INCI    ;SKIP IF IN PROPER ORDER
                        ;
                        ;       CHECK FOR EQUAL VALUES
012E B2CA3F01           ORA D! JZ INCI  ;SKIP IF AV(I) = AV(I+1)
0132 56702B5E           MOV D,M! MOV M,B! DCX H! MOV E,M
0136 712B722B73         MOV M,C! DCX H! MOV M,D! DCX H! MOV M,E
                        ;
                        ;       INCREMENT SWITCH COUNT
013B 21460134           LXI H,SW! INR M
                        ;
```

```
                    ;       INCREMENT I
013F 21470134C3INCI:    LXI H,I! INR M! JMP COMP
                    ;
                    ;       DATA DEFINITION SECTION
0146 00             SW:     DB      0           ;RESERVE SPACE FOR SWITCH COUNT
0147                I:      DS      1           ;SPACE FOR INDEX
0148 05006400 1EAV: DW      5,100,30,50,20,7,1000,300,100,-32767
000A =              N       EQU     ($-AV)/2            ;COMPUTE N INSTEAD OF PRE
015C                        END
A>TYPE SORT.HEX
```

equate value

```
:10010000214601360121470136007EFE09D2190140
:10011000214601 7EB7C20001FF5F16002148011983
:1001200019 4E7923462396577823 9EDA3F01B2CAA7
:10013000 3F0156702B5E712B722B7321460134 21C7
:0701400047 0134C30A01006E
:1001480005006400 1E0032B0140007 00E0032C01BB
:0401580064000100BE
:0000000000
```

machine code in HEX format

```
A>DDT SORT.HEX      start debug run

16K DDT VER 1.0
NEXT  PC
015C 0000  default address (no address on END statement)
-XP

P=0000 100  change PC to 100

-UFFFF      untrace for 65535 steps                                  abort with rubout

C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI   H,0146*0100
-T10        trace 10(16) steps

C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI   H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI   M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI   H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI   M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV   A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI   09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JNC   0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0110 LXI   H,0146
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0113 MOV   A,M
C1Z0M1E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0114 ORA   A
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0115 JNZ   0100
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI   H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI   M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI   H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI   M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV   A,M
-A10D

010D  JC 119  change to a jump on carry                  stopped at 10BH
0110

-XP

P=010B 100  reset program counter back to beginning of program

-T10  trace execution for 10H steps

C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0100 LXI   H,0146
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0103 MVI   M,01
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0105 LXI   H,0147
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0108 MVI   M,00
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010A MOV   A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI   09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JC    0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0119 MOV   E,A
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011A MVI   D,00
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011C LXI   H,0148
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=011F DAD   D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0120 DAD   D
```

altered instruction

```
C0Z0M1E010 A=00 B=6000 D=0000 H=0148 S=0100 P=0121 MOV  C,M
C0Z0M1E010 A=00 B=0005 D=0000 H=0148 S=0100 P=0122 MOV  A,C
C0Z0M1E010 A=05 B=0005 D=0000 H=0148 S=0100 P=0123 INX  H
C0Z0M1E010 A=05 B=0005 D=0000 H=0149 S=0100 P=0124 MOV  B,M*0125
-L100
```

Automatic breakpoint

```
0100  LXI  H,0146
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
010A  MOV  A,M
010B  CPI  09
010D  JC   0119
0110  LXI  H,0146
0113  MOV  A,M
0114  ORA  A
0115  JNZ  0100
-L
```

list some code from 100H

```
0116  RST  07
0119  MOV  E,A
011A  MVI  D,00
011C  LXI  H,0148
```

list more

^ abort list with rubout

-G,118 start program from current PC (0125H) and run in real time to 11BH

*0127 stopped with an external interrupt 7 from front panel (program was looping indefinitely)

-T4 look at looping program in trade mode

```
C0Z0M8E010 A=30 B=0064 D=3006 H=0156 S=0100 P=0127 MOV  D,A
C0Z0M8E010 A=30 B=0064 D=3006 H=0156 S=0100 P=0128 MOV  A,B
C0Z0M8E010 A=00 B=0064 D=3006 H=0156 S=0100 P=0129 INX  H
C0Z0M8E010 A=00 B=0064 D=3006 H=0157 S=0100 P=012A SBB  M*012B
-D140
```

data is sorted, but program doesn't stop

```
0140 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 00 00 00 00 00 2 D D , . . . . . .
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . .
```

- G0 return to CP/M

```
DDT SORT.HEX
```

reload the memory image

```
16K DDT VER 1.0
NEXT  PC
015C 0000
-XP
```

P=0000 100 Set PC to beginning of program

```
-L10D
```

list bad opcode

```
010D  JNC  0119
0110  LXI  H,0146
```

^ abort list with rubout

-A10D assemble new opcode

```
010D  JC 119

0110
```

-L100 list starting section of program

```
0100  LXI  H,0146
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
```

^ abort list with rubout

-A103 change "switch" initialization to 00

```
0103  MVI M,0

0105

-^C  return to CP/M with ctl-c (G0 works as well)

SAVE 1 SORT.COM    save 1 page (256 bytes, from 100H to 1FFMH) on disk in case
                                                    we have to reload later
A>DDT SORT.COM   restart DDT with
                 saved memory image
16K DDT VER 1.0
NEXT  PC
0200 0100  "COM" file always starts with address 100H
-G  run the program from PC=100H

*0110  programmed stop (RST 7) encountered
-D148
                                                  data properly sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 00 00 00 00 00 2 D D , ........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

- G0 return to CP/M
ED SORT.ASM    make changes to original program
 ctl-Z
*M,0^Z#TT   find next ".0"
        MVI     M,0       ;I = 0
*-  up one line in text
        LXI     H,1       ;ADDRESS INDEX
*-  up another line
        MVI     M,1       ;SET TO 1 FOR FIRST ITERATION
*KT kill line and type next line
        LXI     H,I       ;ADDRESS INDEX
*I  insert new line
        MVI     M,0       ;ZERO SW
*T
        LXI     H,I       ;ADDRESS INDEX
*MJNC^Z0T
        JNC*T
        CONT    ;CONTINUE IF I <= (N-2)
*-2DIC^Z0LT
        JC      CONT      ;CONTINUE IF I <= (N-2)
*E
                source from disk A
                hex to disk A
ASM SORT.AAZ<-- skip prn file

CP/M ASSEMBLER - VER 1.0

015C next address to assemble
003H USE FACTOR
END OF ASSEMBLY

DDT SORT.HEX   test program changes

16K DDT VER 1.0
NEXT  PC
015C 0000
-G100

*0118
-D148
                                        data sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 00 00 00 00 00 2 D D , ........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

- abort with rubout

-G0  return to CP/M — program checks OK.
```

CHAPTER 5
CP/M Dynamic Debugging Tool

- Introduction
- DDT Commands
- The A (Assemble) Command
- The D (Display) Command
- The F (Fill) Command
- The G (Go) Command
- The I (Input) Command
- The L (List) Command
- The M (Move) Command
- The R (Read) Command
- The S (Set) Command
- The T (Trace) Command
- The U (Untrace) Command
- The X (Examine) Command
- Implementation Notes
- Sample Session

Introduction

The DDT program allows dynamic interactive testing and debugging of programs generated in the CP/M environment. The debugger is initiated by typing one of the following commands at the CP/M Console Command level

```
DDT
DDT filename.HEX
DDT filename.COM
```

where "filename" is the name of the program to be loaded and tested. In both cases, the DDT program is brought into main memory in the place of the Console Command Processor (refer to the CP/M Interface Guide for standard memory organization), and thus resides directly below the Basic Disk Operating System portion of CP/M. The BDOS starting address, which is located in the address field of the JMP instruction at location 5H, is altered to reflect the reduced Transient Program Area size.

The second and third forms of the DDT command shown above perform the same actions as the first, except there is a subsequent automatic load of the specified HEX or COM file. The action is identical to the sequence of commands

```
DDT
Ifilename.HEX or Ifilename.COM
R
```

where the I and R commands set up and read the specified program to test. (See the explanation of the I and R commands below for exact details.)

Upon initiation, DDT prints a sign-on message in the format

```
nnK DDT-s VER m.m
```

where nn is the memory size (which must match the CP/M system being used), s is the hardware system which is assumed, corresponding to the codes

D	Digital Research standard version
M	MDS version
I	IMSAI standard version
O	Omron systems
S	Digital Systems standard version

and m.m is the revision number.

Following the sign on message, DDT prompts the operator with the character "-" and waits for input commands from the console. The operator can type any of several single character commands, terminated by a carriage return to execute the command. Each line of input can be line-edited using the standard CP/M controls

rubout	remove the last character typed
Control-X	remove the entire line, ready for re-typing
Control-C	system reboot

Any command can be up to 32 characters in length (an automatic carriage return is inserted as the 33rd character), where the first character determines the command type

A	enter assembly language mnemonics with operands
D	display memory in hexadecimal and ASCII
F	fill memory with constant data
G	begin execution with optional breakpoints
I	set up a standard input file control block
L	list memory using assembler mnemonics
M	move a memory segment from source to destination
R	read program for subsequent testing
S	substitute memory values
T	trace program execution
U	untraced program monitoring
X	examine and optionally alter the CPU state

The command character, in some cases, is followed by zero, one, two, or three hexadecimal values which are separated by commas or single blank characters. All DDT numeric output is in hexadecimal form. In all cases, the commands are not executed until the carriage return is typed at the end of the command.

At any point in the debug run, the operator can stop execution of DDT using either a Control-C or G0 (jmp to location 0000H), and save the current memory image using a SAVE command of the form

SAVE n filename.COM

where n is the number of pages (256 byte blocks) to be saved on disk. The number of blocks can be determined by taking the high order byte of the top load address and converting this number to decimal. For example, if the highest address in the Transient Program Area is 1234H then the number of pages is 12H, or 18 in decimal. Thus the operator could type a Control-C during the debug run, returning to the Console Processor level, followed by

SAVE 18 X.COM

The memory image is saved as X.COM on the diskette, and can be directly executed by simply typing the name X. If further testing is required, the memory image can be recalled by typing

DDT X.COM

which reloads the previously saved program from location 100H through page 18 (12FFH). The machine state is not a part of the COM file, and thus the program must be restarted from the beginning in order to properly test it.

DDT Commands

The individual commands are given below in some detail. In each case, the operator must wait for the prompt character (-) before entering the command. If control is passed to a program under test, and the program has not reached a breakpoint, control can be returned to DDT by executing a RST 7 from the front panel (note that the rubout key should be used instead if the program is executing a T or U command). In the explanation of each command, the command letter is shown in some cases with numbers separated by commas, where the numbers are represented by lower case letters. These numbers are always assumed to be in a hexadecimal radix, and from one to four digits in length (longer numbers will be automatically truncated on the right).

Many of the commands operate upon a "CPU state" which corresponds to the program under test. The CPU state holds the registers of the program being debugged, and initially contains zeroes for all registers and flags except for the program counter (P) and stack pointer (S), which default to 100H. The program counter is subsequently set to the starting address given in the last record of a HEX file if a file of this form is loaded (see the I and R commands).

The A (Assemble) Command

DDT allows inline assembly language to be inserted into the current memory image using the A command which takes the form

```
As
```

where s is the hexadecimal starting address for the inline assembly. DDT prompts the console with the address of the next instruction to fill, and reads the console, looking for assembly language mnemonics (see the Intel 8080 Assembly Language Reference Card for a list of mnemonics), followed by register references and operands in absolute hexadecimal form. Each successive load address is printed before reading the console. The A command terminates when the first empty line is input from the console.

Upon completion of assembly language input, the operator can review the memory segment using the DDT disassembler. (See the L command.)

Note that the assembler/disassembler portion of DDT can be overlayed by the transient program being tested, in which case the DDT program responds with an error condition when the A and L commands are used.

The D (Display) Command

The D command allows the operator to view the contents of memory in hexadecimal and ASCII formats. The forms are

```
D
Ds
Ds,f
```

In the first case, memory is displayed from the current display address (initially 100H), and continues for 16 display lines. Each display line takes the form shown below

```
aaaa bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb ccccccccccccccc
```

where aaaa is the display address in hexadecimal, and bb represents data present in memory starting at aaaa. The ASCII characters starting at aaaa are given to the right (represented by the sequence of c's), where non-graphic characters are printed as a period (.) symbol. Note that both upper and lower case alphabetics are displayed, and thus will appear as upper case symbols on a console device that supports only upper case. Each display line gives the values of 16 bytes of data, except that the first line displayed is truncated so that the next line begins at an address which is the multiple of 16.

The second form of the D command shown above is similar to the first, except that the display address is first set to address s. The third form causes the display to continue from address s through address f. In all cases, the display address is set to the first address not displayed in this command, so that a continuing display can be accomplished by issuing successive D commands with no explicit addresses.

Excessively long displays can be aborted by pushing the rubout key.

The F (Fill) Command

The F command takes the form

```
Fs,f,c
```

where s is the starting address, f is the final address, and c is a hexadecimal byte constant. The effect is as follows: DDT stores the constant c at address s, increments the value of s and tests against f. If s exceeds f then the operation terminates, otherwise the operation is repeated. Thus, the fill command can be used to set a memory block to a specific constant value.

The G (Go) Command

Program execution is started using the G command, with up to two optional breakpoint addresses. The G command takes one of the forms

```
G
Gs
Gs,b
Gs,b,c
G,b
G,b,c
```

The first form starts execution of the program under test at the current value of the program counter in the current machine state, with no breakpoints set (the only way to regain control in DDT is through a RST 7 execution). The current program counter can be viewed by typing an X or XP command. The second form is similar to the first except that the program counter in the current machine state is set to address s before execution begins. The third form is the same as the second, except that program execution stops when address b is encountered (b must be in the area of the program under test). The instruction at location b is not executed when the breakpoint is encountered. The fourth form is identical to the third, except that two breakpoints are specified, one at b and the other at c. Encountering either breakpoint causes execution to stop, and both breakpoints are subsequently

cleared. The last two forms take the program counter from the current machine state, and set one and two breakpoints, respectively.

Execution continues from the starting address in real-time to the next breakpoint. That is, there is no intervention between the starting address and the break address by DDT. Thus, if the program under test does not reach a breakpoint, control cannot return to DDT without executing a RST 7 instruction. Upon encountering a breakpoint, DDT stops execution and types

```
*d
```

where d is the stop address. The machine state can be examined at this point using the X (Examine) command. The operator must specify breakpoints which differ from the program counter address at the beginning of the G command. Thus, if the current program counter is 1234H, then the commands

```
G,1234
```

and

```
G400,400
```

both produce an immediate breakpoint, without executing any instructions whatsoever.

The I (Input) Command

The I command allows the operator to insert a file name into the default file control block at 5CH (the file control block created by CP/M for transient programs is placed at this location; see the CP/M Interface Guide). The default FCB can be used by the program under test as if it had been passed by the CP/M Console Processor. Note that this file name is also used by DDT for reading additional HEX and COM files. The form of the I command is

```
Ifilename
```

or

```
Ifilename.filetype
```

If the second form is used, and the filetype is either HEX or COM, then subsequent R commands can be used to read the pure binary or hex format machine code (see the R command for further details).

The L (List) Command

The L command is used to list assembly language mnemonics in a particular program region. The forms are

```
L
Ls
Ls,f
```

The first command lists twelve lines of disassembled machine code from the current list address. The second form sets the list address to s, and then lists twelve lines of code. The last form lists disassembled code from s through address f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. Upon encountering an execution breakpoint, the list address is set to the current value of the program counter (see the G and T commands). Again, long typeouts can be aborted using the rubout key during the list process.

The M (Move) Command

The M command allows block movement of program or data areas from one location to another in memory. The form is

```
Ms,f,d
```

where s is the start address of the move, f is the final address of the move, and d is the destination address. Data is first moved from s to d, and both addresses are incremented. If s exceeds f then the move operation stops, otherwise the move operation is repeated.

The R (Read) Command

The R command is used in conjunction with the I command to read COM and HEX files from the diskette into the transient program area in preparation for the debut run. The forms are

```
R
Rb
```

where b is an optional bias address which is added to each program or data address as it is loaded. The load operation must not overwrite any of the system parameters from 000H through 0FFH (i.e., the first page of memory). If b is omitted, then b = 0000 is assumed. The R command requires a previous I command, specifying the name of a HEX or COM file. The load address for

each record is obtained from each individual HEX record, while an assumed load address of 100H is taken for COM files. Note that any number of R commands can be issued following the I command to re-read the program under test, assuming the tested program does not destroy the default area at 5CH. Further, any file specified with the filetype "COM" is assumed to contain machine code in pure binary form (created with the LOAD or SAVE command), and all others are assumed to contain machine code in Intel hex format (produced, for example, with the ASM command).

Recall that the command

```
DDT filename.filetype
```

which initiates the DDT program is equivalent to the commands

```
DDT
-Ifilename.filetype
-R
```

Whenever the R command is issued, DDT responds with either the error indicator "?" (file cannot be opened, or a checksum error occurred in a HEX file), or with a load message taking the form

```
NEXT PC
nnnn pppp
```

where nnnn is the next address following the loaded program, and pppp is the assumed program counter (100H for COM files, or taken from the last record if a HEX file is specified).

The S (Set) Command

The S command allows memory locations to be examined and optionally altered. The form of the command is

```
Ss
```

where s is the hexadecimal starting address for examination and alteration of memory. DDT responds with a numeric prompt, giving the memory location, along with the data currently held in the memory location. If the operator types a carriage return, then the data is not altered. If a byte value is typed, then the value is stored at the prompted address. In either case, DDT continues to prompt with successive addresses and values until either a period (.) is typed by the operator, or an invalid input value is detected.

The T (Trace) Command

The T command allows selective tracing of program execution for 1 to 65535 program steps. The forms are

```
T
Tn
```

In the first case, the CPU state is displayed, and the next program step is executed. The program terminates immediately, with the termination address displayed as

```
*hhhh
```

where hhhh is the next address to execute. The display address (used in the D command) is set to the value of H and L, and the list address (used in the L command) is set to hhhh. The CPU state at program termination can then be examined using the X command.

The second form of the T command is similar to the first, except that execution is traced for n steps (n is a hexadecimal value) before a program breakpoint occurs. A breakpoint can be forced in the trace mode by typing a rubout character. The CPU state is displayed before each program step is taken in trace mode. The format of the display is the same as described in the X command.

Note that program tracing is discontinued at the interface to CP/M, and resumes after return from CP/M to the program under test. Thus, CP/M functions which access I/O devices, such as the diskette drive, run in real-time, avoiding I/O timing problems. Programs running in trace mode execute approximately 500 times slower than real time since DDT gets control after each user instruction is executed. Interrupt processing routines can be traced, but it must be noted that commands which use the breakpoint facility (G, T, and U) accomplish the break using a RST 7 instruction, which means that the tested program cannot use this interrupt location. Further, the trace mode always runs the tested program with interrupts enabled, which may cause problems if asynchronous interrupts are received during tracing.

Note also that the operator should use the rubout key to get control back to DDT during trace, rather than executing a RST 7, in order to ensure that the trace for the current instruction is completed before interruption.

The U (Untrace) Command

The U command is identical to the T command except that intermediate program steps are not displayed. The untrace mode allows from 1 to 65535 (0FFFFH) steps to be executed in monitored mode, and is used principally to retain control of an executing program while it reaches steady state conditions. All conditions of the T command apply to the U command.

The X (Examine) Command

The X command allows selective display and alteration of the current CPU state for the program under test. The forms are

```
X
Xr
```

where r is one of the 8080 CPU registers

C	Carry Flag	(0/1)
Z	Zero Flag	(0/1)
M	Minus Flag	(0/1)
E	Even Parity Flag	(0/1)
I	Interdigit Carry	(0/1)
A	Accumulator	(0-FF)
B	BC register pair	(0-FFFF)
D	DE register pair	(0-FFFF)
H	HL register pair	(0-FFFF)
S	Stack Pointer	(0-FFFF)
P	Program Counter	(0-FFFF)

In the first case, the CPU register state is displayed in the format

```
CfZfMfEfIf A=bb B=dddd D=dddd H=dddd S=dddd P=dddd inst
```

where f is a 0 or 1 flag value, bb is a byte value, and dddd is a double byte quantity corresponding to the register pair. The "inst" field contains the disassembled instruction which occurs at the location addressed by the CPU state's program counter.

The second form allows display and optional alteration of register values, where r is one of the registers given above (C, Z, M, E, I, A, B, D, H, S, or P). In each case, the flag or register value is first displayed at the console. The DDT program then accepts input from the console. If a carriage return is typed, then the flag or register value is not altered. If a value in the proper range is typed, then the flag or register value is altered. Note that BC, DE,

and HL are displayed as register pairs. Thus, the operator types the entire register pair when B, C, or the BC pair is altered.

Implementation Notes

The organization of DDT allows certain non-essential portions to be overlayed in order to gain a larger transient program area for debugging large programs. The DDT program consists of two parts: the DDT nucleus and the assembler/disassembler module. The DDT nucleus is loaded over the Console Command Processor, and, although loaded with the DDT nucleus, the assembler/disassembler is overlayable unless used to assemble or disassemble.

In particular, the BDOS address at location 6H (address field of the JMP instruction at location 5H) is modified by DDT to address the base location of the DDT nucleus which, in turn, contains a JMP instruction to the BDOS. Thus, programs which use this address field to size memory see the logical end of memory at the base of the DDT nucleus rather than the base of the BDOS.

The assembler/disassembler module resides directly below the DDT nucleus in the transient program area. If the A, L, T, or X commands are used during the debugging process then the DDT program again alters the address field at 6H to include this module, thus further reducing the logical end of memory. If a program loads beyond the beginning of the assembler/disassembler module, the A and L commands are lost (their use produces a "?" in response), and the trace and display (T and X) commands list the "inst" field of the display in hexadecimal, rather than as a decoded instruction.

Sample Session

The following example shows an edit, assemble, and debug for a simple program which reads a set of data values and determines the largest value in the set. The largest value is taken from the vector, and stored into "LARGE" at the termination of the program

```
ED SCAN.ASM
*I
     ↑I   ORG     ↑I   100H      ;START OF TRANSIENT AREA
          MVI          B,LEN     ;LENGTH OF VECTOR TO SCAN
          MVI          C,0       ;LARGER-YET VALUE SO FAR
LOOP--P-0-0-L  LXI     H,VECT    ;BASE OF VECTOR
LOOP:     MOV          A,M       ;GET VALUE
 rubout   SUB          C         ;LARGER VALUE IN C?
 deletes  JNC          NFOUND    ;JUMP IF LARGER VALUE NOT FOUND
;characters NEW LARGEST VALUE, STORE IT TO C
          MOV          C,A
```

tab character

rubout

rubout echo

```
NFOUND: INX     H       ;TO NEXT ELEMENT
        DCR     B       ;MORE TO SCAN?
        JNZ     LOOP    ;FOR ANOTHER
;
;       END OF SCAN, STORE C
        MOV     A,C     ;GET LARGEST VALUE
        STA     LARGE
        JMP     0       ;REBOOT
;
;       TEST DATA
VECT:   DB      2,0,4,3,5,6,1,5
LEN     EQU     $-VECT  ;LENGTH
LARGE:  DS      1       ;LARGEST VALUE ON EXIT
        END
^Z*00P
        ORG     100H    ;START OF TRANSIENT AREA
        MVI     B,LEN   ;LENGTH OF VECTOR TO SCAN
        MVI     C,0     ;LARGEST VALUE SO FAR
        LXI     H,VECT  ;BASE OF VECTOR
LOOP:   MOV     A,M     ;GET VALUE
        SUB     C       ;LARGER VALUE IN C?
        JNC     NFOUND  ;JUMP IF LARGER VALUE NOT FOUND
;       NEW LARGEST VALUE, STORE IT TO C
        MOV     C,A
NFOUND: INX     H       ;TO NEXT ELEMENT
        DCR     B       ;MORE TO SCAN?
        JNZ     LOOP    ;FOR ANOTHER
;       END OF SCAN, STORE C
        MOV     A,C     ;GET LARGEST VALUE
        STA     LARGE
        JMP     0       ;REBOOT
;
;       TEST DATA
VECT:   DB      2,0,4,3,5,6,1,5
LEN     EQU     $-VECT  ;LENGTH
LARGE:  DS      1       ;LARGEST VALUE ON EXIT
        END
*E      <-- End of Edit
```

Create Source Program — underlined characters typed by programmer.

```
ASM SCAN
```

Start Assembler

```
CP/M ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY
```

Assembly Complete — Look at Program Listing

```
TYPE SCAN.PRN
```

Code Address / Machine Code / Source Program

```
0100            ORG     100H    ;START OF TRANSIENT AREA
0100 0608       MVI     B,LEN   ;LENGTH OF VECTOR TO SCAN
0102 0E00       MVI     C,0     ;LARGEST VALUE SO FAR
0104 211901     LXI     H,VECT  ;BASE OF VECTOR
0107 7E         LOOP:   MOV     A,M     ;GET VALUE
0108 91         SUB     C       ;LARGER VALUE IN C?
0109 D20D01     JNC     NFOUND  ;JUMP IF LARGER VALUE NOT FOUND
                ;       NEW LARGEST VALUE, STORE IT TO C
010C 4F         MOV     C,A
010D 23         NFOUND: INX     H       ;TO NEXT ELEMENT
010E 05         DCR     B       ;MORE TO SCAN?
010F C20701     JNZ     LOOP    ;FOR ANOTHER
                ;
                ;       END OF SCAN, STORE C
0112 79         MOV     A,C     ;GET LARGEST VALUE
0113 322101     STA     LARGE
0116 C30000     JMP     0       ;REBOOT
                ;
Code/data listing truncated -->
                ;       TEST DATA
0119 020004030VECT:     DB      2,0,4,3,5,6,1,5
```

```
0000 =          LEN     EQU     $-VECT  ;LENGTH
0121  Value of  LARGE:  DS      1       ;LARGEST VALUE ON EXIT
0122  Equate    END
A>
```

DDT SCAN.HEX — Start Debugger using hex format machine code

```
16K DDT VER 1.0
NEXT  PC
0121 0000
-X
```

X — last load address + 1

next instruction to execute at

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0000 OUT  7F  PC=0
-XP
```

XP — Examine registers before debug run

```
P=0000 100
```

Change PC to 100

```
-X
```

X — Look at registers again

PC changed

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,00
-L100
```

Next instruction to execute at PC=100

```
0100  MVI  B,00
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JNC  010D
010C  MOV  C,A
010D  INX  H
010E  DCR  B
010F  JNZ  0107
0112  MOV  A,C
-L
```

Disassembled Machine Code at 100H (See Source Listing for comparison)

```
0113  STA  0121
0116  JMP  0000
0119  STAX B
011A  NOP
011B  INR  B
011C  INX  B
011D  DCR  B
011E  MVI  B,01
0120  DCR  B
0121  LXI  D,2200
0124  LXI  H,0200
-A116
```

A little more machine code (note that Program ends at location 116 with a JMP to 0000)

A116 — enter inline assembly mode to change the JMP to 0000 into a RST 7, which will cause the program under test to return to DDT if 116H is ever executed.

```
0116  RST 7
0117
```

(single carriage return stops assembly mode)

```
-L113
```

L113 — List Code at 113H to check that RST 7 was properly inserted

```
0113  STA  0121
0116  RST  07
0117  NOP
0118  NOP
0119  STAX B
011A  NOP
011B  INR  B
011C  INX  B
-
```

in place of JMP

```
-X
```

X — Look at registers

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,00
-T
```

T — Execute Program for one step.

initial CPU state, before / is executed

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,00*0102
-T
```

T — Trace one step again (note 08H in B)

automatic breakpoint

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0102 MVI  C,00*0104
```

```
-T
```
Trace again (Register C is cleared)

```
C0Z0M0E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0104 LXI   H,0119*0107
```

```
-T3
```
Trace three steps

```
C0Z0M0E010 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 MOV   A,M
C0Z0M0E010 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB   C
C0Z0M0E011 A=02 B=0000 D=0000 H=0119 S=0100 P=0109 JNC   010D*010D
```

```
-D119
```
Display memory starting at 119H.

Automatic breakpoint at 10DH

Program data

Lower case x

```
0119 02 00 04 03 05 06 01 . . . . . .
0120 05 11 00 22 21 00 02 7E EB 77 13 23 EB 0B 78 B1 . . . " ! . . ~ . w . # . . . x .
0130 C2 27 01 C3 03 29 00 00 00 00 00 00 00 00 00 00 . ' . . . ) . . . . . .
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
0190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . .
```

Data is displayed in ASCII with a "0" in the position of non-graphic characters

```
-X
```
Current CPU state

```
C0Z0M0E011 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX   H
```

```
-T5
```
Trace 5 steps from current CPU state

```
C0Z0M0E011 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX   H
C0Z0M0E011 A=02 B=0000 D=0000 H=011A S=0100 P=010E DCR   B
C0Z0M0E011 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ   0107
C0Z0M0E011 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV   A,M
C0Z0M0E011 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB   C*0109
```

Automatic Breakpoint

```
-U5
```
Trace without listing intermediate states

```
C0Z1M0E111 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC   010D*0108
```

```
-X
```
CPU State at end of U5

```
C0Z0M0E111 A=04 B=0600 D=0000 H=0110 S=0100 P=0108 SUB   C
```

```
-G
```
Run program from current PC until completion (in real-time)
breakpoint at 116H, caused by executing RST 7 in machine code

```
*0116
```

```
-X
```
CPU state at end of program

```
C0Z1M0E111 A=00 B=0000 D=0000 H=0121 S=0100 P=0116 RST   07
```

```
-XP
```
examine and change program counter

```
P=0116 100
```

```
-X
```

```
C0Z1M0E111 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 MVI   B,00
```

```
-T10
```
Trace 10 (hexadecimal) steps

subtext for comparison A<C

first data element

current largest value

```
C0Z1M0E111 A=00 B=0000 D=0000 H=0121 S=0100 P=0102 MVI   C,00
C0Z1M0E111 A=00 B=0000 D=0000 H=0121 S=0100 P=0104 LXI   H,0119
C0Z1M0E111 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 MOV   A,M
C0Z1M0E111 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB   C
```

```
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JNC  010D
C0Z0M0E0I1 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX  H
C0Z0M0E0I1 A=02 B=0000 D=0000 H=011A S=0100 P=010E DCR  B
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ  0107
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV  A,M
C0Z0M0E0I1 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB  C
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC  010D
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=010D INX  H
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011B S=0100 P=010E DCR  B
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=010F JNZ  0107
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=0107 MOV  A,M*0100
-A109
0109  JC 10D
010C
```

Insert a "hot patch" into the machine code to change the JNC to JC

Program should have moved the value from A into C since A>C. Since this code was not executed, it appears that the JNC should have been a JC instruction

```
-G0
```

Stop DDT so that a version of the patched program can be saved

```
SAVE 1 SCAN.COM
```

Program resides on first page, so save 1 page.

```
A>DDT SCAN.COM
```

Restart DDT with the saved memory image to continue testing

```
16K DDT VER 1.0
NEXT  PC
0200 0100
-L100
```

List some code

```
0100  MVI   B,00
0102  MVI   C,00
0104  LXI   H,0119
0107  MOV   A,M
0108  SUB   C
0109  JC    010D
010C  MOV   C,A
010D  INX   H
010E  DCR   B
010F  JNZ   0107
0112  MOV   A,C
-XP
```

Previous patch is present in X-COM

```
P=0100
-T10
```

Trace to see how patched version operates

Data is moved from A to C

```
C0Z0M0E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI   B,00
C0Z0M0E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0102 MVI   C,00
C0Z0M0E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0104 LXI   H,0119
C0Z0M0E010 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 MOV   A,M
C0Z0M0E010 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB   C
C0Z0M0E0I1 A=02 B=0000 D=0000 H=0119 S=0100 P=0109 JC    010D
C0Z0M0E0I1 A=02 B=0000 D=0000 H=0119 S=0100 P=010C MOV   C,A
C0Z0M0E0I1 A=02 B=0002 D=0000 H=0119 S=0100 P=010D INX   H
C0Z0M0E0I1 A=02 B=0002 D=0000 H=011A S=0100 P=010E DCR   B
C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ   0107
C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=0107 MOV   A,M
C0Z0M0E0I1 A=00 B=0702 D=0000 H=011A S=0100 P=0108 SUB   C
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC    010D
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX   H
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011B S=0100 P=010E DCR   B
C1Z0M0E1I1 A=FE B=0602 D=0000 H=011B S=0100 P=010F JNZ   0107*0107
-X
```

breakpoint after 16 steps

```
C1Z0M0E1I1 A=FE B=0602 D=0000 H=011B S=0100 P=0107 MOV   A,M
-G,108
```

Run from current PC and breakpoint at 108H

```
*0108
-X
```

next data item

```
C1Z0M0E1I1 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB   C
```

```
-I                          Single step for a few cycles
C1Z0M0E1I1 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB  C*0109
-I
C0Z0M0E0I1 A=02 B=0602 D=0000 H=011B S=0100 P=0109 JC   010D*010C
-X
C0Z0M0E0I1 A=02 B=0602 D=0000 H=011B S=0100 P=010C MOV  C,A
-G      Run to completion
*0116
-X
C0Z1M0E1I1 A=03 B=0003 D=0000 H=0121 S=0100 P=0116 RST  07
-S121       look at the value of "LARGE"
0121 03  Wrong Value!
0122 00
0123 22
0124 21
0125 00
0126 02          End of the S command
0127 7E .

-L100

0100  MVI  B,03
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JC   010D
010C  MOV  C,A
010D  INX  H
010E  DCR  B
010F  JNZ  0107
0112  MOV  A,C      } Review the code
-L

0113  STA  0121
0116  RST  07
0117  NOP
0118  NOP
0119  STAX B
011A  NOP
011B  INR  B
011C  INX  B
011D  DCR  B
011E  MVI  B,01
0120  DCR  B
-XP

P=0116 100    Reset the PC

-I   Single step, and watch data values

C0Z1M0E1I1 A=03 B=0003 D=0000 H=0121 S=0100 P=0100 MVI  B,03*0102
-I

C0Z1M0E1I1 A=03 B=0003 D=0000 H=0121 S=0100 P=0102 MVI  C,00*0104
-I                          count set
                                "largest" set
C0Z1M0E1I1 A=03 B=0000 D=0000 H=0121 S=0100 P=0104 LXI  H,0119*0107
-I
```

```
                                                    base address of data set
C0Z1M0E111 A=03 B=0000 D=0000 H=0119 S=0100 P=0107 MOV   A,M*0108
-T
                                  first data item brought to A
C0Z1M0E111 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB   C*0109
-T

C0Z0M0E011 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JC    010D*010C
-T

C0Z0M0E011 A=02 B=0800 D=0000 H=0119 S=0100 P=010C MOV   C,A*010D
-T
                                  first data item moved to C correctly
C0Z0M0E011 A=02 B=0802 D=0000 H=0119 S=0100 P=010D INX   H*010E
-T

C0Z0M0E011 A=02 B=0802 D=0000 H=011A S=0100 P=010E DCR   B*010F
-T

C0Z0M0E011 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ   0107*0107
-T

C0Z0M0E011 A=02 B=0702 D=0000 H=011A S=0100 P=0107 MOV   A,M*0108
-T
                                  second data item brought to A
C0Z0M0E011 A=00 B=0702 D=0000 H=011A S=0100 P=0108 SUB   C*0109
-T
                                  subtract destroys data value which was loaded!!!
C1Z0M1E010 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC    010D*010D
-T

C1Z0M1E010 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX   H*010E
-L100

0100  MVI   B,00
0102  MVI   C,00
0104  LXI   H,0119
0107  MOV   A,M
0108  SUB   C          <-- This should have been a CMP so that register A
0109  JC    010D           would not be destroyed.
010C  MOV   C,A
010D  INX   H
010E  DCR   B
010F  JNZ   0107
0112  MOV   A,C
-H108

0108  CMP C      hot patch at 108H changes SUB to CMP

0109

-G0       stop DDT for SAVE
SAVE 1 SCAN.COM
                  save memory image
A>DDT SCAN.COM    Restart DDT

16K DDT VER 1.0
NEXT  PC
0200 0100
-XP

P=0100
```

```
-L116
0116  RST  07
0117  NOP
0118  NOP
0119  STAX B
011A  NOP
-  (rubout)
```

Look at code to see if it was properly loaded (long typeout aborted with rubout)

```
-G,116
```

Run from 100H to completion

```
*0116
-XC
```

Look at Carry (accidental typo)

```
C1
```

```
-X
```

Look at CPU state

```
C121M0E111 A=06 B=0006 D=0000 H=0121 S=0100 P=0116 RST  07
-S121
```

Look at "Large" — it appears to be correct.

```
0121 06

0122 00

0123 22 .
```

```
-G0
```

stop DDT

```
ED SCAN.ASM
```

Re-edit the source program, and make both changes

```
*NSUB
*0LT
```

ctl-Z

```
        SUB   C         ;LARGER VALUE IN C?
*SSUB*ZCMP*Z0LT
        CMP   C         ;LARGER VALUE IN C?
*
        JNC   NFOUND    ;JUMP IF LARGER VALUE NOT FOUND
*SNC*ZJC*Z0LT
        JC    NFOUND    ;JUMP IF LARGER VALUE NOT FOUND
*E
```

```
ASM SCAN.AAZ
```

Re-assemble, selecting source from disk A, hex to disk A, print to Z (selects no print file)

```
CP/M ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY
```

```
DDT SCAN.HEX
```

Re-run debugger to check changes

```
DDT VER 1.0
NEXT  PC
0121 0000
-L116
```

```
0116  JMP  0000
0119  STAX B
011A  NOP
011B  INR  B
-  (rubout)
```

check to ensure end is still at 116H

```
-G100,116
```

Go from beginning with breakpoint at end

```
*0116
-D121
0120 00 00 22 21 00 02 7E EB 77 13 23 EB 08 79 B1    "!  ~ w # y
0130 C2 27 01 C3 03 23 00 00 00 00 00 00 00 00 00  '  #
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

correct value computed

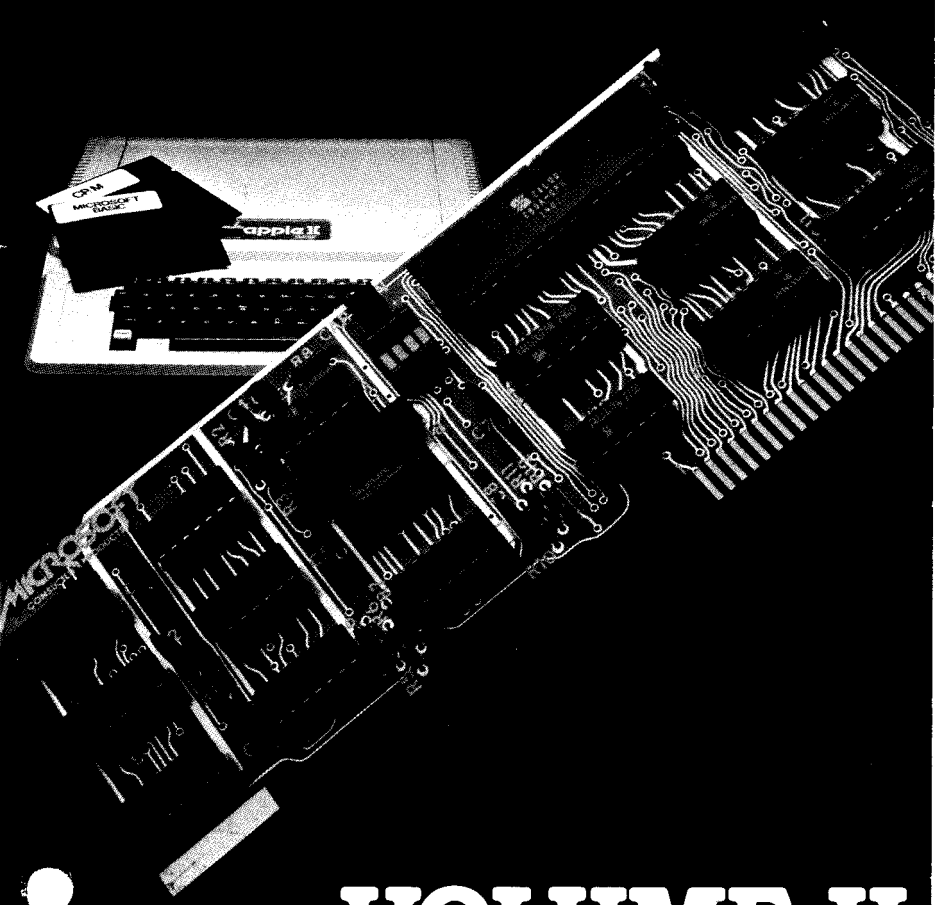
breakpoint reached

Look at "LARGE"

- (rubout) aborts long typeout

-GO stop DDT. debug session complete

SOFTCARD



VOLUME II

SoftCard™

A Peripheral for the Apple II®
With CP/M® and Microsoft BASIC on diskette.

Produced by Microsoft

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

Copyright and Trademark Notices

If you have any questions on these copyrights, please contact:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

TABLE OF CONTENTS

INTRODUCTION

PART I: Installation and Operation

PART II: Software and Hardware Details

PART III: CP/M Reference Manual

PART V: Software Utilities Manual

Microsoft BASIC Reference Manual

PART IV: Microsoft BASIC Reference Manual

INTRODUCTION

Microsoft BASIC, written for Z-80 and 8080 microprocessors, is the most extensive implementation of BASIC available for microcomputers today. Now in its fifth major release, Microsoft BASIC (or BASIC-80) meets the ANSI qualifications for BASIC as set forth in document BSRX3.60-1978. It is upwardly compatible with Applesoft BASIC.

With the Microsoft SoftCard, the most recent version of BASIC-80, Version 5.0, is available to Apple owners for the first time. It brings new power to the Apple, adding major features such as PRINT USING, 16-digit precision, CALL, CHAIN and COMMON, WHILE/WEND and improved disk I/O.

The SoftCard package includes two versions of Microsoft BASIC. MBASIC, which is found on both disks, includes all standard Applesoft extensions from low-resolution graphics to sound and cursor control. These features plus high-resolution graphics are included in GBASIC, which is found on the 16-sector disk only.

The reference guide is divided into five chapters plus a number of appendices. Chapter 1 is a short section covering differences between Microsoft BASIC and Applesoft, especially important for persons used to programming in Applesoft. Chapter 2 includes instructions for initialization of either version of Microsoft BASIC, (referred to throughout this manual as BASIC-80), and explains details of information representation when using Microsoft BASIC. Chapter 3 contains the syntax and semantics of every command and statement in BASIC-80 for the Apple, ordered alphabetically. Chapter 5 pertains to GBASIC only, describing all features found exclusively in GBASIC. The appendices contain lists of error messages, ASCII codes and math functions plus helpful information on the use of assembly language subroutines and disk I/O.

This manual is not intended as a tutorial on the BASIC language. It is a reference manual for the specific features of Microsoft BASIC. If you need instructional material regarding the BASIC language, we suggest you read one of the following:

BASIC by Robert L. Albrecht, LeRoy Finkel, Jerry Brown (John Wiley & Sons, 1973)

BASIC and the Personal Computer by Thomas A. Dwyer and Margot Critchfield (Addison-Wesley Publishing Co., 1978)

BASIC From the Ground Up by David E. Simon (Hayden, 1978)

CHAPTER 1

Microsoft BASIC-80 and Applesoft: A Comparison

Microsoft BASIC-80, Version 5.0, includes many features not found in Applesoft and also uses some features differently than Applesoft. Realizing that most SoftCard buyers have previously written BASIC programs in Applesoft, we include here a listing of the differences between the two versions of BASIC.

By making note of these differences and using the new features provided by BASIC-80, you can take advantage of increased BASIC programming power.

Features of Microsoft BASIC not found in Applesoft

The following features are found in Microsoft BASIC only. A brief description of these features is given here; for more information on the syntax, purpose and peculiarities of each, see Chapters 2 and 3 of this manual.

CHAIN and COMMON	Used to call in another BASIC program from disk and pass variables to it. This feature allows the disk to be used as program memory.
CALL	Used to call 6502 or Z-80 assembly language subroutine or FORTRAN subroutine.
PRINT USING	Greatly enhances programming convenience by making it easy to format output. It includes asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.
Built-in Disk I/O Statements	Since standard Applesoft BASIC and integer BASIC were not designed for a disk environment, Disk I/O commands have to be included in PRINT statements. With Microsoft BASIC 5.0's built-in disk I/O statements, this process is eliminated (no more PRINT "ctrl D").
WHILE/WEND	Gives BASIC a more structured flavor. By putting a WHILE statement in front of a loop and

	the WEND statement at the end, BASIC 5.0 will continuously execute the loop as long as a given condition is true.
EDIT Commands	Let you edit individual program lines easily and efficiently without re-entering the whole line.
AUTO and RENUM	RENUM makes it easier to edit and debug programs by letting you automatically renumber lines in user-specified increments. AUTO is a convenience feature that generates line numbers automatically after every carriage return.
IF ... THEN ... ELSE	Extends the IF statement in Applesoft to provide for handling the negative case of IF.
ANSI Compatibility	Microsoft 5.0 BASIC meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. That means any program you write on your Apple in Microsoft BASIC can be run on any other machine that has an ANSI standard BASIC.
Compilability	Microsoft has developed a BASIC compiler that compiles MBASIC and GBASIC programs into directly executable Z-80 machine code. The compiler is available separately to SoftCard owners.
Powerful Data Types	BASIC 5.0 has three variable types — fast two-byte true integer variables, single precision variables and double precision variables — to give it 16-digit precision, as opposed to 9-digit precision on the Apple. Also, hexadecimal and octal constants may be used.
Added String Functions	INSTR, HEX$, OCT$, STRING$, and direct assignment of substrings with MID$ are implemented.

Added Operators	New boolean operators AND, OR, XOR, IMP, and EQV are provided. True Integer arithmetic is supported with an Integer divide and MOD operators.
User-Defined Functions	BASIC-80 user-defined functions allow multiple arguments.
Protected Files	BASIC programs may be saved in a protected binary format. See SAVE, Chapter 3.

We have also included four new features to Microsoft BASIC, especially to take advantage of the Apple's unique characteristics. They are:

BUTTON(0)	A function used to determine whether a paddle button has been pressed.
BEEP	A statement that generates a tone of specified pitch and duration.
HSCRN(X,Y)	A function used to determine if a point has been plotted on the high-resolution screen at a specified point.
VPOS(0)	A function that returns the vertical cursor position.

Applesoft Enhancements

Both versions of BASIC support low-resolution graphics, sound, cursor control and other Applesoft BASIC features. The version of Microsoft BASIC included on the 16-sector disk also supports all of the Applesoft high-resolution graphics features except DRAW, XDRAW, SCALE and ROT.

Applesoft-compatible statements and functions found in MBASIC and GBASIC are shown below. Those features available only in GBASIC are indicated with an asterisk.

GR
COLOR
PLOT
VLIN
HLIN
SCRN
POP
HGR*
HCOLOR*
HPLOT*

TEXT
HTAB
VTAB
INVERSE
NORMAL
PDL(0)

Features Used Differently in Microsoft BASIC Than in Applesoft

Certain statements and commands found in Microsoft BASIC and Applesoft have slightly different uses. You should be aware of these differences when writing BASIC-80 programs. Those statements that differ are listed below; for more information see Chapters 2 and 3 of this manual.

FOR ... NEXT
INPUT
ON ERROR GOTO
RESUME
TEXT
GR
HGR
IF ... THEN ... ELSE
CALL

Changes in BASIC-80 Features

For the SoftCard version of BASIC-80, we have made a few very minor changes to normal CP/M Microsoft BASIC features. If you are accustomed to programming in Microsoft BASIC under CP/M, you will want to note the following changes:

TRON/TROFF	Statement name has been changed to TRACE/NOTRACE. Operation of this statement remains the same.
DELETE	Statement name has been changed to DEL. Operation of this statement remains the same.
WIDTH	You now have the option to specify screen height in addition to line width. Also, default width is 40 columns for Apple video and 80 columns for external terminals.
WAIT	WAIT now monitors the status of an address rather than of a machine input port. The effect, however, remains the same.
CLOAD	Not implemented.

CSAVE	Not implemented.
NULL	Not implemented.
INP	Not implemented.
OUT	Not implemented.

NOTE: BASIC-80 Version 5.0 programs transferred to the Apple must be in ASCII format (i.e., saved with the A option). They may not be in binary format.

Applesoft Features Not Supported

The following features found in Applesoft BASIC are not found in Microsoft BASIC.

FLASH	SHLOAD
ESC A, B, C, D screen editing	XDRAW
STORE	DRAW
RECALL	SCALE
IN#	cassette LOAD
PR#	cassette SAVE
HIMEM ... LOMEM	ROT

CHAPTER 2

GENERAL INFORMATION ABOUT BASIC-80

INITIALIZATION

MBASIC is the CP/M version of Microsoft BASIC that includes all standard Applesoft extensions except high-resolution graphics. It is supplied on both the 13-sector and the 16-sector disks in the SoftCard package. The name of the file on both disks is MBASIC.COM. These initialization instructions refer to MBASIC but may be used for GBASIC simply by substituting GBASIC where MBASIC is typed. (For specific instructions for initializing GBASIC, see Chapter 5.)

To load and run Microsoft BASIC-80, simply bring up the CP/M operating system in the usual manner (See Operations Manual). After the A> prompt appears type:

```
MBASIC
```

and press the RETURN key. In a few seconds, a copyright notice will appear, indicating BASIC-80 is ready for your command.

This sets at 3 the number of files that may be open at any one time during the execution of a BASIC program (see /F option below), allows all memory up to the start of FDOS in CP/M to be used (see /M option below) and sets the maximum record size at 128.

The command line format below can be used in place of the simple MBASIC command if you wish to set these options and/or automatically RUN any program after initialization:

```
MBASIC [<filename>] [/F:<number of files>] [/M:<highest memory location>]
[/S:<maximum record size>]    Press RETURN
```

The <filename> option allows you to RUN a program automatically after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include the SYSTEM statement (See Chapter 3) to return to CP/M when they have finished, allowing the next program in the batch stream to execute.

The /F:<number of files> option sets the number of disk files that may

be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes plus 128 (or number specified by /S:) bytes of memory. If the /F option is omitted, the number of files defaults to 3. Number of files may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases, it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used. The <highest memory location> number may be decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /S:<maximum record size> option sets the maximum size to be allowed for random files. Any integer may be specified, including integers larger than 128.

When BASIC-80 is initialized, the system will reply:

```
BASIC-80 Version 5.xx
(Apple CP/M Version)
Copyright 1980 (c) by Microsoft
Created: dd-Mmm-yy
xxxx Bytes free
Ok
```

Here are a few examples of the different initialization options:

A>MBASIC PAYROLL.BAS	Use all memory and 3 files; load and execute PAYROLL.BAS
A>MBASIC INVENT/F:6	Use all memory and 6 files; load and execute INVENT.BAS
A>MBASIC /M:32768	Use first 32K of memory and 3 files
A>MBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files and execute DATACK.BAS

MODES OF OPERATION

When BASIC-80 is initialized, it types the prompt "Ok." "Ok" means BASIC-80 is at command level, that is, it is ready to accept commands. At this point, BASIC-80 may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC commands and statements are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and

stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering RUN command.

DISK FILES

Disk filenames follow the normal CP/M naming conventions. All filenames may include A:, B:, C:, D:, E: or F: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. The drive name, if specified, must be upper case (i.e., A: not a:). A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN filename commands if no "." appears on the filename and the filename is less than 9 characters long.

LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

```
nnnnn BASIC statement [:BASIC statement...] <carriage return>
```

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the <line feed> or Control J. <Control J> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

CHARACTER SET

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters. The alphabetic characters in BASIC-80 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC-80 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC-80:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<carriage return>	Terminates input of a line.

Control Characters

The following control characters are in BASIC-80:

Control @	Rubout
Control-A	Enters Edit Mode on the line being typed.
Control-B	Backslash
Control-C	Interrupts program execution and returns to BASIC-80 command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspace. Deletes the last character typed. Same as ←
Control-I	Tab. Tab stops are every eight columns. Same as →
Control-J	Line feed. Moves to next physical line.
Control-K	Right square bracket
Control-O	Halts program output while execution continues. A second Control-O restarts output.
Control-R	Retypes the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-X	Deletes the line that is currently being typed.
Control-Y	Permits recovery from pressing RESET on a system with an Autostart ROM.
→	Tab. Same as Control-I.
←	Backspace. Same as Control-H.

NOTE: Control-@ Control-B, Control-K and Control-U may be redefined using the CONFIGIO utility.

CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants — Whole numbers between −32768 and +32767. Integer constants do not have decimal points.

2. Fixed Point constants — Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants — Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent must be in the range −38 to +38.
Examples:
```
235.988E-7 = .0000235988
2359E6 = 2359000000
```
(Double precision floating point constants use the letter D instead of E. See "Single and Double Precision Form for Numeric Constants.")

4. Hex constants — Hexadecimal numbers with the prefix &H. Examples:
```
&H76
&H32F
```

5. Octal constants — Octal numbers with the prefix &O or &. Examples:
```
&O347
&1234
```

Single And Double Precision Form For Numeric Constants

Numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
−7.09E−06	−1.09432D−06
3489.0	3489.0#
22.5!	7654321.1234

VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

Variable Names And Declaration Characters

BASIC-80 variable names may be any length; up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed — see below.

A variable name may not be a reserved word unless the reserved word is embedded. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string. Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of BASIC-80 variable names follow.

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single precision value

There is a second method by which variable types may be declared. The BASIC-80 statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 3.

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
```

The arithmetic was performed in double precision and the result was returned to D (single

.857143	precision variable), rounded and printed as a single precision value.

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range −32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.
Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E−8 times the original single precision value.
Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value. Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
↑	Exponentiation	X↑Y
–	Negation	–X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained. Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
$X-\frac{Y}{Z}$	X–Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)Y$	(X↑2)↑Y
X^{YZ}	X↑(Y↑Z)
X(–Y)	X*(–Y) Two consecutive operators must be separated by parentheses.

Integer Division And Modulus Arithmetic

Two additional operators are available in BASIC-80: Integer division and modulus arithmetic.

Integer division is denoted by the backslash or Control-B on the Apple

keyboard. (\). The operands are rounded to integers (must be in the range −32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

Overflow And Division By Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (−1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Chapter 3.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Chapter 3.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Chapter 3). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range −32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands. Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
−1 AND 8=8	−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)

10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
−1 OR −2=−1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X=−(X+1)	The two's complement of any integer is the bit complement plus one.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter 4. BASIC-80 also allows "user defined" functions that are written by the programmer. See DEF FN, Chapter 3.

String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=   <>   <   >   <=   >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
```

```
"SMYTH" < "SMYTHE"
B$ < "9/12/78" where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT (Control-A) key or with Control-H. Rubout or Control-A surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-X. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC-80 will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided. See EDIT, Chapter 3.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Chapter 3) NEW is usually used to clear memory prior to entering a new program.

ERROR MESSAGES

If BASIC-80 detects an error that causes program execution to terminate, an error message is printed. For a complete list of BASIC-80 error codes and error messages, see Appendix E.

CHAPTER 3

BASIC-80 COMMANDS AND STATEMENTS

All of the BASIC-80 commands and statements are described in this chapter. Each description is formatted as follows:

Syntax: Shows the correct syntax for the instruction. See below for syntax notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Syntax Notation

Wherever the syntax for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).
6. Items separated by a vertical bar (|) are mutually exclusive; choose one.
7. All reserved words must be preceded by and followed by a space.

AUTO

Syntax: AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example:

```
AUTO 100,50     Generates line numbers 100, 150, 200 ...
AUTO            Generates line numbers 10, 20, 30, 40 ...
```

BEEP

Syntax: BEEP <pitch>, <duration>

Purpose: To create a tone of specified pitch and duration.

Remarks: 0 is the highest <pitch>; 255 is the lowest.

0 is the shortest <duration>; 255 is the longest. A <duration> of 255 lasts approximately 1 second.

BEEP is intended for sound effects purposes. No attempt has been made to match specific <pitches> or <durations> with specific musical notes or note lengths.

Example:

```
10 BEEP PDL(0), PDL(1): GOTO 10
```

CALL

Syntax: CALL <variable name>[(<argument list>)]

Purpose: To call a Z-80 assembly language subroutine.

Syntax 2: CALL %<variable name>[(<argument>)]

Purpose: To call a 6502 assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an assembly language subroutine. (See also the USR function, Chapter 4)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains

the arguments that are passed to the assembly language subroutine.

In Syntax 2, the per cent symbol (%) preceding the <variable name> allows the CALL statement to call a 6502 assembly language subroutine.

A 6502 subroutine call may have up to three parameters of one byte each. The first (if any) value is placed in the 6502 A register, the next in the X register and the last in the Y register.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:
```
110 MYROUT=&HD000
120 CALL MYROUT
130 BELL=&HFF3A
140 CALL % BELL
```

CHAIN

Syntax: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:
```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:
```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See COMMON statement. Example:
```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is,

a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

CLEAR

Syntax: CLEAR [,[<expression1>][,<expression2>]]

Purpose: To set all numeric variables to zero and all string variables to null; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC-80.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: In previous versions of BASIC-80, <expression1> set the amount of string space, and <expression2> set the end of memory. BASIC-80, release 5.0 and later, allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for BASIC to use.

Examples:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR,32768,2000
```

CLOSE

Syntax: CLOSE[#]<file number>[,[#]<file number...>]

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Appendix B.

COLOR

Syntax: COLOR=<color number>
where <color number> is an integer in the range 0-15.

Purpose: To set the color for plotting in low resolution graphics mode.

Remarks: The colors available and their numbers are:

0 black	8 brown
1 magenta	9 orange
2 dark blue	10 gray
3 purple	11 pink
4 dark green	12 green
5 gray	13 yellow
6 medium blue	14 aqua
7 light blue	15 white

<color number> may be specified in the GR statement. (See GR). If it is not specified in GR it is set to zero when GR is set until another color is specified with the COLOR statement.

To find out the COLOR of a given point on the screen, use the SCRN function.

COLOR may be used in low resolution graphics mode only.

Example:
```
10 GR
20 COLOR=13
```

COMMON

Syntax: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:
```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
    .
    .
    .
```

CONT

Syntax: CONT

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may also be used to continue execution after an error.

CONT is invalid if the program has been edited during the break.

Example: See example for STOP statement.

DATA

Syntax: DATA <list of constants>

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ.)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

Example: See examples for READ statement.

DEF FN

Syntax: DEF FN<name>[(<parameter list>)] = <function definition>

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
410 DEF FNAB(X,Y)=X↑3/Y↑2
420 T=FNAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

DEFINT/SNG/DBL/STR

Syntax: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC-80 assumes all variables without declaration characters are single precision variables.

Examples:

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.

```
10 DEFINT I-N,W-Z
```

All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

DEF USR

Syntax: DEF USR[<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y↑2/2.89)
.
.
.
```

DEL

Syntax: DEL[<line number>][-<line number>]

Purpose: To delete program lines.

Remarks: BASIC-80 always returns to command level after a DEL is executed. If <line number> does not exist, an "Illegal function call" error occurs.

DELETE may be used in place of DEL. DEL entered in a program will list as DELETE.

Examples:

DEL 40	Deletes line 40
DEL 40-100	Deletes lines 40 through 100, inclusive
DEL-40	Deletes all lines up to and including line 40

DIM

Syntax: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see OPTION BASE).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
    .
    .
    .
```

EDIT

Syntax: EDIT <line number>

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

← — In Edit Mode, [i]moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout (Control-A) or left arrow (←) key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The

deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. **Finding Text**

S — The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K — The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. **Replacing Text**

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. **Ending and Restarting Edit Mode**

<cr> — Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E — The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q — The Q subcommand returns to BASIC-80 command level, without saving any of the changes that were made to the line during Edit Mode.

L — The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A — The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC-80 automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC-80 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination , first exit Edit Mode with the Q subcommand. BASIC-80 will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC-80 responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

END

Syntax: END

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC-80

always returns to command level after an END is executed.

Example: 520 IF K>1000 THEN END ELSE GOTO 20

ERASE

Syntax: ERASE <list of array variables>

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ... Otherwise, use

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC-80's error codes are listed in Appendix E.

ERROR

Syntax: ERROR <integer expression>

Purpose: 1) To simulate the occurrence of a BASIC-80 error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC-80 (see Appendix E), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC-80 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC-80 types this line)
Ok
```

Example 2:

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
```

```
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
.
.
.
```

FIELD

Syntax: FIELD[#]<file number>,<field width> AS <string variable>...

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed. <file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.) Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: **Do not use a FIELDed variable name in an INPUT or LET statement.** Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

FILES

Syntax: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:
```
FILES

FILES "*.BAS"

FILES "B:*.*"

FILES "TEST?.BAS"
```

FOR...NEXT

Syntax:
```
FOR <variable>=x TO y [STEP z]
.
.
.
NEXT [<variable>][,<variable>...]
```
where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC-80 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

There must be one and only one NEXT for every FOR.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them if the variable for each FOR is specified in the NEXT.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:
```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:
```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```
In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:
```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```
In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value

is set. (Note: Previous versions of BASIC-80 set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

Example 4:

```
10 FOR I=1 TO 20
20 IF I 10 GOTO 100
30 NEXT
40 GOTO 110
100 NEXT
110 END
```

This program would result in a NEXT without FOR error. There may be one and only one NEXT for every FOR.

GET

Syntax 1: GET [#]<file number>[,<record number>]

Purpose 1: To read a record from a random disk file into a random buffer.

Syntax 2: GET <keyboard character>

Purpose 2: To read a single character from the keyboard.

Remarks: Syntax 1: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number 32767. After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

Syntax 2: <keyboard character> is not displayed on the screen. It is not necessary to press the RETURN key. If Control @ is the <keyboard character>, it returns the null character. The result of GETting a left-arrow or Control H may also PRINT as if the null character were being returned.

Examples: For examples of syntax 1, see Appendix B.

Syntax 2:

```
10 GET A$:PRINT A$;
20 GOTO 10
```

GOSUB...RETURN

Syntax:

```
GOSUB <line number>
   .
   .
   .
RETURN
```

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. Also, see POP.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO

Syntax: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
```

```
30 A = 3.14*R↑2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5        AREA = 78.5
R = 7        AREA = 153.86
R = 12       AREA = 452.16
?Out of data in 10
Ok
```

GR

Syntax: GR <screen number>, <color number>
where <screen number> is an integer in the range 0-1 and <color number> is an integer in the range 0-15.

Purpose: To initialize low-resolution graphics mode.

Remarks: <screen number> specifies the mode to be used as follows:

number	screen mode
0	40x40 graphics + 4 lines text
1	40x48 graphics with no lines text

If <screen number> is not specified, <screen number> = 0 is assumed.

GR clears the screen when it initializes low-resolution graphics mode.

<color number> specifies the color to be used and is optional. If <color number> is not specified, color is set to black. <color number> will fill the screen with the color specified by <color number>. See COLOR for a list of color names and their associated numbers.

Examples:
```
GR      Same as Applesoft GR statement
GR 1,15 Fill screen with white and set 40x48 mode
```

NOTE THAT THIS STATEMENT MAY BE USED DIFFERENTLY IN MBASIC THAN IN APPLESOFT.

HLIN

Syntax: HLIN <x1 coordinate>, <x2 coordinate> AT <y coordinate>
where x1 and x2 are integers in the range 0-39 and y is an integer in the range 0-47.

Purpose: In low resolution graphics mode, to draw a horizontal line from point (x1,y) to point (x2,y).

Remarks: <x1 coordinate> must be less than or equal to <x2 coordinate>.

The color of the line is specified by the most recently executed COLOR statement.

If any of the coordinates are not in the required range as specified above, an ILLEGAL FUNCTION CALL error results.

The HLIN statement normally draws a line composed of dots from x1 to x2 at the vertical coordinate y. However, if used when in TEXT mode, or when in mixed graphics and text mode with y in the range 40-47, a line of characters is displayed instead of the line of dots.

Example:
```
10 GR
20 COLOR=3
30 HLIN 14,20 AT 39
```

HOME

Syntax: HOME

Purpose: To clear the screen of all text and move the cursor to the upper left corner of the screen.

Remarks: When used with an external terminal, HOME sends a "clear screen" character sequence to terminals that support this feature.

Example:
```
10 HOME
20 VTAB 12
30 PRINT "A CLEAN SCREEN"
```

HTAB

Syntax: HTAB <screen position number>

Purpose: To move the cursor to the screen position that is <screen position number> spaces from the left edge of the current screen line.

Remarks: The first (left-most) position on the line is 1, the last (right-most) position on the line is 40.

HTAB uses absolute moves, not relative moves. For instance, if the cursor was at position 10, and the command

HTAB 13 was executed, the cursor would be moved to position 13, not position 23.

If a <screen position number> greater than 40 but less than 255 is specified, it will be treated modulo 40. The command HTAB 60 would place the cursor at position 20 on the current line. A <screen position number> greater than 255 results in an ILLEGAL FUNCTION CALL error.

IF...THEN ...ELSE AND IF...GOTO

Syntax: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Syntax: IF <expression> GOTO <line number>
[ELSE <statement(s)> | <line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma may be used before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
    ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:
```
200 IF I THEN GET#1,I
```
This statement GETs record number I if I is not zero.

Example 2:
```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
    .
    .
    .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:
```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```
This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

INPUT

Syntax:
```
INPUT[;][<"prompt string">;]<list of variables>
INPUT[;][<"prompt string">,]<list of variables>
```

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and the program waits for information to be typed in at the terminal. If <"prompt string"> is included, the string is printed. The required data items are then entered at the terminal.

Note that unlike Applesoft, you have the option of entering either a semicolon or comma after the <"prompt string">. Like Applesoft, a semicolon causes a question mark to be printed after the <"prompt string">. A comma after the <"prompt string"> causes the question mark to be suppressed.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data items that are entered are assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

NOTE THAT THIS STATEMENT IS USED DIFFERENTLY IN MBASIC THAN IN APPLESOFT.

Example 1:
```
10 INPUT X
20 PRINT X "SQUARED IS" X↑2
30 END
RUN
? 5 (The 5 was typed in by the user
      in response to the question mark.)
 5 SQUARED IS 25
Ok
```

Example 2:
```
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R↑2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

INPUT#

Syntax: INPUT#<file number>,<variable list>

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC-80 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

After a GET statement INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

Example: See Appendix B.

INVERSE

Syntax: INVERSE

Purpose: To set the video output mode so that the screen displays black characters on a white background.

Remarks: When using an external terminal, INVERSE sends a "Hi-

lite" character sequence to those terminals that support this feature. (See "Installation and Operations Manual.")

INVERSE does not affect characters that are already on the screen when INVERSE is executed.

The NORMAL command restores the mode to the usual white letters on black background. (See NORMAL.)

Example:
```
10 PRINT "THESE ARE WHITE CHARACTERS"
20 INVERSE
30 PRINT "THESE ARE BLACK CHARACTERS"
```

KILL

Syntax: KILL <filename>

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs. KILL is used for all types of disk files: program files, random data files and sequential data files.

Example:
```
200 KILL "DATA1.TXT"
```
See also Appendix B.

LET

Syntax: [LET] <variable>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:
```
110 LET D=12
120 LET E=12↑2
130 LET F=12↑4
140 LET SUM=D+E+F
    .
    .
    .
```
or
```
110 D=12
120 E=12↑2
130 F=12↑4
140 SUM=D+E+F
    .
    .
```

LINE INPUT

Syntax: LINE INPUT[;][<"prompt string">;]<string variable>

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC-80 will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See Example, for LINE INPUT#.

LINE INPUT#

Syntax: LINE INPUT#<file number>,<string variable>

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
```

```
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES     234,4     MEMPHIS
LINDA JONES     234,4     MEMPHIS
Ok
```

LIST

Syntax 1: LIST [<line number>]

Syntax 2: LIST [<line number>[-[<line number>]]]

or LIST [<line number>[,[<line number>]]]

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC-80 always returns to command level after a LIST is executed.

Syntax 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, only the specified line is listed.

Syntax 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

Examples:

Syntax 1:

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Format 2:

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

LLIST

Syntax: LLIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character-wide printer.

BASIC-80 always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Syntax 2.

NOTE: Use of LLIST requires that a printer card be plugged into slot 1 of the Apple.

Example: See the examples for LIST, Syntax 2.

LOAD

Syntax: LOAD <filename>[,R]

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example: LOAD "STRTRK",R

LPRINT AND LPRINT USING

Syntax: LPRINT [<list of expressions>]
LPRINT USING <string exp>;<list of expressions>

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See PRINT and PRINT USING.

LPRINT assumes a 132-character-wide printer.

NOTE: Use of LPRINT requires that a printer card be plugged into slot 1 of the Apple

LSET AND RSET

Syntax: LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions.

Examples:
```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```
See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

MERGE

Syntax: MERGE <filename>

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC-80 always returns to command level after executing a MERGE command.

Example: MERGE "NUMBRS"

MID$

Syntax: MID$(<string exp1>,n[,m])=<string exp2>
where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example:
```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ may also be used as a function that returns a substring of a given string. See MID$ in Chapter 4.

NAME

Syntax: NAME <old filename> AS <new filename>

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:
```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```
In this example, the file that was formerly named ACCTS will now be named LEDGER.

NEW

Syntax: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC-80 always returns to command level after a NEW is executed.

NORMAL

Syntax: NORMAL

Purpose: To restore the video output mode to the usual white characters on black background.

Remarks: NORMAL is used in conjunction with the INVERSE command. (See INVERSE.)

NORMAL does not affect characters already on the screen in INVERSE mode when the NORMAL command is executed.

For external terminals that support the "Hi-lite" feature for INVERSE, NORMAL sends a "Lo-lite" character sequence. (See "Installation and Operations Manual.")

Example:
```
10 INVERSE
20 PRINT "THIS IS INVERSE MODE"
30 NORMAL
40 PRINT "THIS IS NOT"
```

ON ERROR GOTO

Syntax: ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example: 10 ON ERROR GOTO 1000

NOTE THAT THIS STATEMENT IS USED DIFFERENTLY IN MBASIC THAN IN APPLESOFT

ON...GOSUB AND ON...GOTO

Syntax: ON <expression> GOTO <list of line numbers>
ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example: 100 ON L-1 GOTO 150,300,320,390

OPEN

Syntax: OPEN <mode>,[#]<file number>,<filename>[,<reclen>]

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes. See also Appendix A

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example:
```
10 OPEN "I",2,"INVEN"
```
See also Appendix B.

OPTION BASE

Syntax: OPTION BASE n
where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

PLOT

Syntax: PLOT <x coordinate>, <y coordinate>
where <x coordinate> is an integer in the range 0-39 and <y coordinate> is an integer in the range 0-47.

Purpose: In low resolution graphics mode, to place a dot with <x coordinate> and <y coordinate>.

Remarks: The point (0,0) is in the upper left corner of the screen.

The color of the dot placed by PLOT is determined by the most recently executed COLOR or GR statement.

PLOT normally places a dot at (x,y). However, if PLOT is used while in TEXT mode, or while in mixed graphics and text mode with y in the range 40-47, a character is displayed instead of a dot.

If either <x coordinate> or <y coordinate> is not in the required range as specified above, an ILLEGAL FUNCTION CALL error results.

Example:
```
GR
COLOR=9
PLOT 24,37
```

POKE

Syntax: POKE I,J
where I and J are integer expressions

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range 0 to 65536. Refer to the 6502 to Z-80 Memory Map in the Hardware Details section of this manual.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See PEEK, Chapter 4.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

NOTE: PEEKs and POKEs used in Applesoft will not work unless they are first converted to use Z-80 addresses. Refer to the 6502 To Z-80 Memory Map in the Hardware Details section of this manual.

Example:
```
10 POKE &H5A00,&HFF
```

POP

Syntax: POP

Purpose: To return from a subroutine that was branched to by a GOSUB without branching back to the statement following the most recent GOSUB.

Remarks: POP is used instead of RETURN to nullify a GOSUB. Like RETURN, it nullifies the last GOSUB in effect, but it does not return to the statement following the GOSUB. After a POP, the next RETURN encountered will branch to one statement beyond the second most recently executed

GOSUB. Thus POP, in effect, takes one address off the top of the "stack" of RETURN addresses.

See also GOSUB ... RETURN.

Example:

```
5 PRINT "HERE WE GO"
10 GOSUB 100
20 PRINT "XYZ"
30 END
100 GOSUB 200
110 PRINT "HELLO"
120 RETURN
200 POP
210 RETURN

RUN
HERE WE GO
XYZ
```

PRINT

Syntax: PRINT [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon. If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly.

If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10↑(−6) is output as .000001 and 10↑(−7) is output as 1E−7. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D−16 is output as .0000000000000001 and 1D−17 is output as 1D−17.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:
```
10 X=5
20 PRINT X+5, X−5, X*(−5), X↑5
30 END
RUN
 10          0          −25          3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:
```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X↑2 "AND";
30 PRINT X "CUBED IS" X↑3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and

line 40 causes a blank line to be printed before the next prompt.

Example 3:
```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

PRINT USING

Syntax: PRINT USING <string exp>;<list of expressions>

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\   \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

\# A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##    ";10.2,5.3,66.789,.234
10.20     5.30     66.79     0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "  ##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##-    ";-68.95,22.449,-7.01
68.95-    22.45    7.01-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4    *-0.9    765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
  $456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (↑↑↑↑) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

↑↑↑↑ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or − is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##↑↑↑↑";234.56
 2.35E+02

PRINT USING ".####↑↑↑↑-";888888
 .8889E+06

PRINT USING "+.##↑↑↑↑";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "_ _" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

PRINT# AND PRINT# USING

Syntax: PRINT#<filenumber>,[USING<string exp>;]<list of exps>

Purpose: To write data to a sequential disk file.

Remarks: <filenumber> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described for PRINT USING. The expressions in <list of expressions> are the numeric and/or

string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT# 1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT# 1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT# 1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34). For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT# 1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT# 1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT # 1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""  93604-1"
```

and the statement

```
INPUT # 1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT # 1,USING"$$# # #.# #,";J;K;L
```

PRINT#, PRINT# USING and WRITE# may also be used to put characters in the random file buffer before a PUT statement

For more examples using PRINT#, see Appendix B. See also WRITE#.

PUT

Syntax: PUT [#] <file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767.

Example: See Appendix B.

RANDOMIZE

Syntax: RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC-80 suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768- + 32767)? 3 (user types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (-32768- + 32767)? 4 (user types 4 for
new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (-32768- + 32767)? 3 (same sequence as
first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

NOTE: With the BASIC Compiler, the prompt given by RANDOMIZE is:

```
Random Number Seed (−32768 to 32767)?
```

READ

Syntax: READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables. (See DATA.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA

statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE).

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", "  ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,",  COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE         ZIP
DENVER,       COLORADO      80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

REM

Syntax: REM <remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Example:

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
.
.
.
```

or

```
.
.
.
120 FOR I=1 TO 20 'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
.
.
.
```

RENUM

Syntax: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed.

The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET

Syntax: RESET

Purpose: To reset the CP/M directory allocation information after you have switched disks.

Use: The procedure for changing disks is as follows: First, type CLOSE to close any data files that may be open at the time. Then, remove the old disk and insert the new disk. Finally, *after* you have inserted the new disk, type RESET. Failure to follow this procedure when changing disks may cause loss of data, resulting in a "Disk Read Only" error.

RESTORE

Syntax: RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:
```
10 READ A,B,C
20 RESTORE
```

```
30 READ D,E,F
40 DATA 57, 68, 79
   .
   .
   .
```

RESUME

Syntax:
```
RESUME
RESUME 0
RESUME NEXT
RESUME <line number>
```

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:
```
10 ON ERROR GOTO 900
   .
   .
   .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
   .
   .
   .
```

NOTE THAT THIS STATEMENT IS USED DIFFERENTLY IN MBASIC THAN IN APPLESOFT.

RUN

Syntax 1: RUN [<line number>]

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC-80 always returns to command level after a RUN is executed.

Example: RUN

Syntax 2: RUN <filename>[,R]

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M the default extension .BAS is supplied.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R
See also Appendix B.

SAVE

Syntax: SAVE <filename>[,A | ,P]

Purpose: To save a program file on disk.

Remarks: <filename> is a quoted string with the default extension .BAS. If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

In addition, programs written in 5.0 BASIC that you wish to transfer to your Apple SoftCard system must be saved in ASCII format.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples: SAVE"COM2",A
SAVE"PROG",P
See also Appendix B.

STOP

Syntax: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC-80 always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see CONT).

Example:

```
10 INPUT A,B,C
20 K=A↑2*5.3:L=B↑3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

SWAP

Syntax: SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE ": B$=" ALL ": C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
```

```
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

SYSTEM

Syntax: SYSTEM

Purpose: To close all files and return to CP/M

Remarks: You cannot use Control-C to return to CP/M; it always returns to BASIC.

Example:
```
SYSTEM
A>
```

TEXT

Syntax: TEXT

Purpose: To reset the screen to normal full Apple text mode (24 lines x 40 characters) from low-resolution graphics (in either MBASIC or GBASIC) or high-resolution graphics (GBASIC only).

Remarks: TEXT will clear the screen if it is used to return from low-resolution graphics. It will not clear the screen from high-resolution graphics.

If used while in Text mode, TEXT has the same effect as VTAB 24.

Example:
```
10 HGR
20 COLOR=5
30 VLIN 24,30 AT 35
40 TEXT
50 PRINT "THIS IS A VERTICAL LINE"
```

TRACE/NOTRACE

Syntax:
```
TRACE
NOTRACE
```

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRACE statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the NOTRACE statement (or when a NEW command is executed).

Example:

```
TRACE
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
Ok
NOTRACE
Ok
```

VLIN

Syntax: VLIN <y1 coordinate>, <y2 coordinate> AT <x coordinate>
where <y1 coordinate> and <y2 coordinate> are integers in the range 0-47 and <x coordinate> is an integer in the range 0-39

Purpose: In low-resolution graphics mode, to draw a vertical line from the point at (x,y1) to the point at (x,y2).

Remarks: <y1 coordinate> must be less than or equal to <y2 coordinate>.

If any of the coordinates are not in the required range as specified above, an ILLEGAL FUNCTION CALL error results.

The color of the line is determined by the most recent COLOR statement.

The VLIN statement normally draws a line composed of dots from y1 to y2 at the horizontal coordinate x. However, if used when in Text mode, or when in mixed graphics and text mode with y2 in the range 40-47, the part of the line that falls in the text area will be displayed as a line of characters.

Example:

```
10 GR
20 COLOR=3
30 VLIN 20,45 AT 12
```

VTAB

Syntax: VTAB <screen line number>

Purpose: To move the cursor to the line on the screen that corresponds to the specified <screen line number>.

Remarks: The first line (top line) on the screen is line 1; the last line or bottom line on the screen is line 24.

VTAB uses absolute moves. For instance, if the cursor was on line 10 of the screen, then the command VTAB 13 was executed, the cursor would be moved to line 13, not line 23.

If a <screen line number> greater than 24 is specified, it will be treated modulo 24. The command VTAB 26 would place the cursor on screen line 2. If a <screen line number> greater than 255 is specified, it results in an ILLEGAL FUNCTION CALL error.

VTAB can move the cursor either up or down.

When used with an external terminal, VTAB sends a "cursor address" character sequence to terminals that address this feature.

Example:
```
10 VTAB 12: PRINT "MIDDLE OF SCREEN"
```

WAIT

Syntax: WAIT <address>, I[,J]
where I and J are integer expressions

Purpose: To suspend program execution while monitoring the status of an address.

Remarks: The WAIT statement causes execution to be suspended until a specified address develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC-80 loops back and reads the data at the address again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:
```
100 WAIT &HE000,128
200 PRINT "KEYPRESS!":GOTO 100
```

WHILE...WEND

Syntax:

```
WHILE <expression>
  .
  .
[<loop statements>]
  .
  .
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS=0
120         FOR I=1 TO J-1
130                 IF A$(I)>A$(I+1) THEN
                            SWAP A$(I),A$(I+1):FLIPS=1
140         NEXT I
150 WEND
```

WIDTH

Syntax 1: WIDTH [LPRINT] <line width>

Purpose 1: To set the printed line width in number of characters for the terminal or line printer.

Syntax 2: WIDTH [<line width>],[<screen height>]

Purpose 2: To set the printed line width in number of characters and/or screen height in number of lines for the terminal.

Remarks: <line width> must be an integer in the range 15-255. <screen height> must be an integer in the range 1-24. If you are using 40-column Apple video, the default line length is 40, and the default screen height is 24. If you are using an external terminal with 80 columns, the default line width is 80 and the default screen height is 24.

In Syntax 1, if the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

In Syntax 2, one or both of the parameters may be specified, but at least one must be specified.

If <line width> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOs function, returns to zero after position 255. Ok

WRITE

Syntax: WRITE [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of te expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement. (See PRINT.)

Example:
```
10 A=80:B=90:C$=THAT'S ALL
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

WRITE#

Syntax: WRITE # <file number>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

WRITE#, PRINT#, and PRINT# USING may also be used to put characters in the random file buffer before a PUT statement. In the case of WRITE#, BASIC-80 pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

CHAPTER 4

BASIC-80 FUNCTIONS

The intrinsic functions provided by BASIC-80 are presented in this chapter. The functions may be called from any program without further definition. Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X and Y	Represent plotting coordinates for graphics functions.
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC-80 will round the fractional portion and use the resulting integer.

NOTE

With the BASIC-80 interpreter, only integer and single precision results are returned by functions. Double precision functions are supported only by the BASIC compiler.

ABS

Syntax: ABS(X)

Action: Returns the absolute value of the expression X.

Example:
```
PRINT ABS(7*(-5))
 35
Ok
```

ASC

Syntax: ASC(X$)

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix L for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:
```
10 X$ = "TEST"
20 PRINT ASC(X$)
```

```
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

ATN

Syntax: ATN(X)

Action: Returns the arctangent of X in radians. Result is in the range −pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:
```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

BUTTON

Syntax: BUTTON(I)

Action: Returns the current value of the push button on the game controller, specified by I.

Remarks: I is in the range 0-3.

The returned value is either 0, if the button is not currently depressed, or -1 if the button is currently depressed.

Example:
```
10 IF BUTTON (0) THEN PRINT "BOOM"
```

CDBL

Syntax: CDBL(X)

Action: Converts X to a double precision number.

Example:
```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

CHR$

Syntax: CHR$(I)

Action: Returns a string whose one element has ASCII code I. x(ASCII codes are listed in Appendix G.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

Example:
```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

CINT

Syntax: CINT(X)

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range −32768 to 32767, an "Overflow" error occurs.

Example:
```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

COS

Syntax: COS(X)

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:
```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

CSNG

Syntax: CSNG(X)

Action: Converts X to a single precision number.

Example:
```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
```

```
RUN
 975.3421   975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

CVI, CVS, CVD

Syntax: CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)

Action: Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

See also MKI$, MKS$, MKD$, in this Chapter and Appendix B.

EOF

Syntax: EOF(<file number>)

Action: Returns −1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

The EOF function may also be used with random files. If a GET is done past the end of the file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
```

```
50 C=C+1:GOTO 30
.
.
.
```

EXP

Syntax: EXP(X)

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

FIX

Syntax: FIX(X)

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:
```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

FRE

Syntax: FRE(0)
FRE(X$)

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC-80.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up.

Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:
```
PRINT FRE(0)
 14542
Ok
```

HEX$

Syntax: HEX$(X)

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:
```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS  " A$ "  HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

INKEY$

Syntax: INKEY$

Action: Returns either a one character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through to the program except for Control-C which terminates the program.

Example:
```
1000 'Timed Input Subroutine
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

INPUT$

Syntax: INPUT$(X[,[#]Y])

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are

passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:
```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:
```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

INSTR

Syntax: INSTR([I,]X$,Y$)

Action: Searches for the first occurrence of string Y in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or ifY$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:
```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2   6
Ok
```

INT

Syntax: INT(X)

Action: Returns the largest integer <=X.

Examples:
```
PRINT INT(99.89)
 99
Ok
```

```
PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

LEFT$

Syntax: LEFT$(X$,I)

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:
```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

LEN

Syntax: LEN(X$)

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:
```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

LOC

Syntax: LOC(<file number>)

Action: With random disk files, LOC returns the next record number to be used if a GET or PUT (without a record number) is executed. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:
```
200 IF LOC(1)>50 THEN STOP
```

LOF

Syntax: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example:
```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

LOG

Syntax: LOG(X)

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:
```
PRINT LOG(45/7)
 1.86075
Ok
```

LPOS

Syntax: LPOS(X)

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:
```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

MID$

Syntax: MID$(X$,I[,J])

Action: Returns a string of length J characters from X beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:
```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

MKI$, MKS$, MKD$

Syntax: MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:
```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
.
.
```

See also CVI, CVS, CVD, in this Chapter and Appendix B.

OCT$

Syntax: OCT$(X)

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:
```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

PDL

Syntax: PDL(I)

Action: Returns the current value, in the range 0-255, of the game control specified by I.

Remarks: I is an integer in the range 0-3.

The value of two game controls should not be read in consecutive instructions, as the reading from the first may affect the second. A delay such as 10 FOR X=1 TO 10:

NEXT X between the two provides sufficient separation for a correct reading.

Example:
```
10 PRINT PDL(0): GOTO 10
RUN
0
23
79
100
190
255
C
BREAK IN
OK
```

PEEK

Syntax: PEEK(I)

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, See POKE, Chapter 3

NOTE: PEEKs and POKEs used in Applesoft and Integer BASIC will not work with BASIC-80 unless they are first converted to use Z-80 addresses. Refer to the 6502 To Z-80 memory map in the "Software and Hardware Details" section of this manual.

Example: A=PEEK(&H5A00)

POS

Syntax: POS(I)

Action: Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example: IF POS(X)>60 THEN PRINT CHR$(13)

Also see the LPOS function.

RIGHT$

Syntax: RIGHT$(X$,I)

Action: Returns the rightmost I characters of string X$. If I= LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:
```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

Also see the MID$ and LEFT$ functions.

RND

Syntax: RND[(X)]

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded. (See RANDOMIZE.) However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:
```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

SCRN

Syntax: SCRN(X,Y)
where X is an integer in the range 0-39 and Y is an integer in the range 0-47.

Action: Returns the color of the point at (X,Y).

Example:
```
10 GR
20 COLOR=13
30 PLOT 10,15
PRINT SCRN(10,15)
RUN
13
```

SGN

Syntax: SGN(X)

Action: If X>0, SGN(X) returns 1. If X=0, SGN(X) returns 0. If X<0, SGN(X) returns −1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

SIN

Syntax: SIN(X)

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example:
```
PRINT SIN(1.5)
 .997495
Ok
```

SPACE$

Syntax: SPACE$(X)

Action: Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:
```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

SPC

Syntax: SPC(I)

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

Example:
```
PRINT "OVER" SPC(15) "THERE"
OVER                    THERE
Ok
```

Also see the SPACE$ function.

SQR

Syntax: SQR(X)

Action: Returns the square root of X. X must be >=0.

Example:
```
10 FOR X = 0 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10          3.16228
 15          3.87298
 20          4.47214
 25          5
Ok
```

STR$

Syntax: STR$(X)

Action: Returns a string representation of the value of X.

Example:
```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

STRING$

Syntax: STRING$(I,J)
STRING$(I,X$)

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example:
```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

TAB

Syntax: TAB(I)

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that posi-

tion on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                    AMOUNT
G. T. JONES             $25.00
Ok
```

TAN

Syntax: TAN(X)

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

USR

Syntax : USR[<digit>](X)

Action: Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix C.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
    .
    .
    .
```

VAL

Syntax: VAL(X$)

Action: Returns the numerical value of string X$. If the first character of X$ is not +, −, &, or a digit, VAL(X$) = 0. VAL (X$)

will, however, strip blanks, tabs, and linefeeds from the argument string.

Example:
```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
.
.
.
```

See the STR$ function for numeric to string conversion.

VARPTR

Syntax 1: VARPTR(<variable name>)

Syntax 2: VARPTR(#<file number>)

Action: Syntax 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to −32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2: Returns the starting address of the disk I/O xbuffer assigned to <file number>.

For random files, VARPTR (#<file number>) returns the address of the FIELD buffer

Example:
```
100 X=USR(VARPTR(Y))
```

VPOS

Syntax: VPOS(I)

Action: Returns the current vertical position of the cursor. The topmost position is 1. X is a dummy argument.

Example:

```
10 PRINT "NOW YOU SEE IT."
20 FOR T=0 TO 1000: NEXT T
30 VTAB VPOS(0) -1
40 PRINT "NOW YOU DON'T "
```

CHAPTER 5

HIGH-RESOLUTION GRAPHICS: GBASIC

INITIALIZATION

GBASIC is the CP/M version of Microsoft BASIC that includes high-resolution graphics capability in addition to all of the features of MBASIC. It is supplied only on the 16-sector disk. The name of the file is GBASIC.COM.

To load and run GBASIC, simply bring up the CP/M operating system in the usual manner (See "Installation and Operations Manual." After the A> prompt appears, type:

```
GBASIC
```

and press the RETURN key. In a few seconds, a copyright notice will appear, indicating GBASIC is ready for your command.

This initialization process sets at 3 the number of files that may be open at any one time during the execution of a BASIC program (see /F option below), allows all memory up to the start of FDOS in CP/M to be used (see /M option below) and sets maximum record size at 128.

The command line format below allows you to set these options and/or automatically RUN any program after initialization:

```
GBASIC <filename> [/F:<number of files>] [/M:<highest
memory location>][/S:<maximum record size>] Press RETURN
```

The <filename> option allows you to RUN a program automatically after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include the SYSTEM statement (See Chapter 3) to return to CP/M when they have finished, allowing the next program in the batch stream to execute.

The /F:<number of files> option sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes plus 128 (or number specified by /S:) bytes of memory. If the /F option is omitted,

the number of files defaults to 3. <number of files> may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases, it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, highest memory location should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used. The <highest memory location> number may be decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

The /S:<maximum record size> option sets the maximum size to be allowed by random files. Any integer may be specified, including integers larger than 128. If the /S option is omitted, maximum record size is set at 128.

When BASIC-80 is initialized, the system will reply:

```
BASIC-80 Version 5.xx
(Apple CP/M Version)
Copyright 1980 (c) by Microsoft
Created: dd-Mmm-yy
xxxx Bytes free
Ok
```

Here are a few examples of the different initialization options:

A>GBASIC PAYROLL.BAS	Use all memory and 3 files; load and execute PAYROLL.BAS
A>GBASIC INVENT/F:6	Use all memory and 6 files; load and execute INVENT.BAS
A>GBASIC /M:32768	Use first 32K of memory and 3 files
A>GBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files and execute DATACK.BAS

All other information regarding GBASIC, with the exception of high-resolution graphics commands, is the same as that for MBASIC and can be found in Chapters 1-4. High-resolution graphics are the added attraction of GBASIC and are described below:

HGR High-Resolution Statements and Commands

<color number> is an integer in the range 0-12.

Syntax: HGR <screen number>, <color number>
where <screen number> is an integer in the range 0-3 and <color number> is an integer in the range 0-7.

Purpose: To initialize high-resolution graphics mode.

Remarks: <screen number> specifies the display mode to be used as follows:

Screen #	Clear Screen	Screen Mode
0	yes	280 x 160 graphics + 4 lines text
1	yes	280 x 192 graphics, no lines text
2	no	280 x 160 graphics + 4 lines text
3	no	280 x 192 graphics, no lines text

If <screen number> is not specified, <screen number> = 0 is assumed.

<color number> specifies the color to be used and is optional. If <color number> is not specified, color is set to 0. When used with modes 0 and 1 above, <color number> will fill the screen with the color specified by <color number>. See HCOLOR for a list of color names and their associated numbers.

Examples:

```
10 HGR        Same as Applesoft HGR statement
10 HGR 1,2    Fill screen with violet, set 280 x 192 mode
10 HGR 3      Set 280 x 192 mode, don't clear screen
```

NOTE THAT THIS STATEMENT CAN BE USED DIFFERENTLY IN GBASIC THAN IN APPLESOFT.

HCOLOR

Syntax: HCOLOR=<color number>
where <color number> is an integer in the range 0-12.

Purpose: To set the color for plotting in high resolution graphics mode.

Remarks: The colors available and their numbers are:

0 black	4 black	8 black1
1 green	5 orange	9 white1
2 violet	6 blue	10 black 2
3 white	7 white	11 white 2
		12 reverse

To distinguish between the different whites and blacks. Numbers 0, 3, 4 and 7 plot a very fine line. Black1, white1, black2 and white2 (8, 9, 10 and 11) plot a larger dot or thicker line that is equal in size (width) with dots or lines plotted with green, violet, orange or blue. Black1 and whi-

tel should be used with green or violet if you want dots or lines of the same position and width. Black2 and white2 should be used with orange or blue.

If you are using a black and white monitor, just use 0, 3, 4 and 7.

<color number> may be specified in the HGR statement (See HGR.) If is not specified in HGR, it is set to zero by HGR until another color is specified with the HCOLOR statement.

HCOLOR may be used in high-resolution graphics mode only.

Note that because of the way in which home TVs work, a high-resolution dot plotted with HCOLOR=3 (white) or HCOLOR=7 (white) will be white only if both (x,y) and (x+1,y) are plotted. If only (x,y) is plotted, the dot will be blue when x is even and green when x is odd.

HPLOT

Syntax 1: HPLOT [<x1>, <y1>] [TO <x2>, <y2> ... [TO <xn>, <yn>]]

Purpose: To plot a point or draw a line(s) on the high-resolution screen, using the points specified by (x1,y1), (x2,y2) etc.

Syntax 2: HPLOT TO <x2>, <y2>

Purpose: To draw a line from the last dot plotted to the point at (x2,y2).

Remarks: In Syntax 1, HPLOT <x1>, <y1> plots a single point.

HPLOT <x1>, <y1> TO <x2>, <y2> TO ... <xn>, <yn> plots a line starting a (x1, y1) and proceeding to each of the points specified. The plotted line may be extended from point to point in the same statement by specifying additional points, limited only by screen limits and the 239 character limit.

In Syntax 1, the color of the dot or line is determined by the most recent HCOLOR statement. If no color has been specified, the default color 0 will be assigned.

In Syntax 2, the color of the line is determined by the last HCOLOR executed. Syntax 2 cannot be used if no dot has previously been plotted.

HPLOT may be used in high-resolution graphics mode only.

Example:
```
10 HGR
20 COLOR=2
30 HPLOT 24,125 TO 100,12 TO 270,1
```

HSCRN

Syntax: HSCRN (X,Y)

Action: In high-resolution graphics mode, returns -1 if a dot exists at point (X,Y).

Remarks: Note that unlike SCRN, HSCRN does not recognize color.

X must be in the range 0-279 and Y must be in the range 0-191.

Example:
```
10 HGR: COLOR=3
20 HPLOT 0,100 TO 279,100
30 PRINT HSCRN (46,100), HSCRN (20,20)
RUN
-1            0
```

APPENDIX A

New Features in BASIC-80, Release 5.0

The execution of BASIC programs written under Microsoft BASIC, release 4.51 and earlier may be affected by some of the new features in release 5.0. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE.
2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I% = 2.5 results in I% = 3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the 5th position, A(1.5) yields A(2), and X = 11.5 MOD 4 yields 0 for X).
3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.
4. Division by zero and overflow no longer produce fatal errors. See Chapter 2.
5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used. See RND in Chapters 3 and 4.
6. The rules for PRINTing single precision and double precision numbers have been changed. See PRINT, in Chapter 3.
7. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space. See CLEAR, Chapter 3.
8. Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with a carriage return causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.
9. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string

fields, and an underscore signifies a literal character in a format string.

10. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer. See WIDTH, Chapter 3.
11. The at-sign and underscore are no longer used as editing characters.
12. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.
13. BASIC programs may be saved in a protected binary format. See SAVE, Chapter 3.
14. Reserved words must be preceded by and followed by a space.

Loading and Saving HIRES Pictures with GBASIC

Below is a short GBASIC program demonstrating the use of random disk I/O statements to load and save Hires Pictures to the disk. Loading and saving Hires pictures in this way is as fast as or faster than using Apple DOS BSAVE and BLOAD statements. This program also creates some pretty Hires pictures.

```
10 DEFINT A-Z:DEFSNG A,F,R
20 DIM X(23),Y(23)
30 GOTO 4000


1000 HGR 1,3:HCOLOR=0
1010 HPLOT 140,96
1020 FOR A=0 TO 3.14159*20 STEP .05
1030 R=SIN(A*2.9)
1040 HPLOT TO 140+107*R*COS(A),96+95*R*SIN(A)
1050 NEXT
1060 HGR 1,12:FOR T=0 TO 500:NEXT:HGR 1,12
1070 GET A$:GOTO 4000


2000 N=INT(RND*14)+13
2010 PI=6.28318/N:FOR I=0 TO N-1:A=PI*I
2020 X(I)=COS(A)*107+140:Y(I)=SIN(A)*95+96
2030 NEXT
2040 HGR 1,0:HCOLOR=3
2050 FOR I=0 TO N-1:FOR J=I TO N-1:HPLOT X(I),Y(I) TO X(J),Y(J):NEXT:NEXT
2060 HGR 1,12:FOR T=0 TO 200:NEXT T:HGR 1,12
2070 GET A$:GOTO 4000
```

```
3000 HOME:VTAB 4
3010 ON ERROR GOTO 3020:FILES "*.PIC":PRINT:PRINT:ON ERROR GOTO 0
3020 PRINT "Load or Save (L/S)? ";:GET D$
3030 IF D$="S" OR D$="s" THEN D=1:HGR 3:PRINT "Save"
       ELSE D=0:HGR 1,0:PRINT "Load"
3040 PRINT:INPUT "File name? ",F$:IF F$="" THEN 3040
3050 IF INSTR(F$,".")=0 THEN F$=F$+".PIC"

3060 OPEN "R",1,F$
3070 FIELD#1,128 AS A$:B$=A$
3080 H=16:L=0:P=VARPTR(B$)+1
3090 FOR I=1 TO 64
3100 POKE P,L:POKE P+1,H
3110 IF D=1 THEN LSET A$=B$:PUT 1 ELSE GET 1:LSET B$=A$
3120 L=L XOR 128:IF L=0 THEN H=H+1
3130 NEXT:CLOSE
3140 FOR T=0 TO 1500:NEXT T


4000 TEXT:HOME:VTAB 4:PRINT TAB(5)"*** HIRES GRAPHICS DEMO ***"
4010 VTAB 7:PRINT TAB(5)"1. Rose":PRINT:PRINT TAB(5)"2. Polygon":PRINT
4020 PRINT TAB(5)"3. Load/Save Hires Picture":PRINT:PRINT TAB(5)"Which - ";
4030 A$=INPUT$(1):A=VAL(A$):IF A>3 OR A<1 THEN 4030
4040 PRINT A$:PRINT:PRINT TAB(5)"Working..."
4050 ON A GOTO 1000,2000,3000
```

NOTE: Hires pictures transferred to CP/M from Apple DOS with APDOS will not load correctly with the above program. The first four bytes of these files contain the destination address and length of the picture, which must be removed. This will also save 1K of disk space. Follow the procedure below to fix transferred Apple DOS pictures (you type the *underlined* characters):

```
DDT
DDT VERS 2.2
Ifilename.ext
RFC
M200,2200,100
G0
A>SAVE 32 filename.ext
```

APPENDIX B

BASIC-80 Disk I/O

Disk I/O procedures for the beginning BASIC-80 user are examined in this appendix. If you are new to BASIC-80 or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The CP/M operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE "filename"[,A] — Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD "filename"[,R] — Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

RUN "filename"[,R] — RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

MERGE "filename" — Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE com-

mand, the "merged" program resides in memory, and BASIC returns to command level.

KILL "filename" — Deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file.

NAME — To change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited.

DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC-80 program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#		INPUT#	WRITE#
	PRINT# USING		LINE INPUT#	
CLOSE	EOF	LOC		

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode. — `OPEN "O",#1,"DATA"`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.) — `PRINT#1,A$;B$;C$`
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode. —
```
CLOSE #1
OPEN "I",#1,"DATA"
```

4. Use the INPUT# statement to read data from the sequential file into the program.

```
INPUT#1,X$,Y$,Z$
```

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN

NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
```

```
Input past end in 20
Ok
```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

Adding Data To A Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file.

Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME
       120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible

because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

Creating a Random File

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD #1 20 AS N$,
 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R"#1,"FILE"
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

Access a Random File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$,
4 AS A$, 8 AS P$
```

NOTE:

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

```
PRINT N$
PRINT CVS(A$)
```

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE"
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

PROGRAM B-6 - INVENTORY

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
```

```
225 IF (FUNCTION<1) OR (FUNCTION>6) THEN PRINT "BAD FUNCTION
      NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:IF A$ <>"Y" THEN
      RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=55 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=KI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;" REORDER
      LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";CVI(Q$) TAB(50)
      "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1) OR (PART%>100) THEN PRINT "BAD PART NUM-
      BER":GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM B-6 - INVENTORY

Sequential I/O to and from Random Files

It is also possible to perform sequential I/O operations to and from random disk files. Although it is generally slower, it *is* very similar to the Apple DOS random file I/O and may be useful.

```
20
30 OPEN "R",1,"RND.TXT"
40 FOR I=1 TO 20
50 WRITE#1,"RECORD ",I," *** DCJCJC ***"
60 PUT 1,I
70 NEXT
80 CLOSE
90 '
100 '
110 OPEN "R",1,"RND.TXT"
120 FOR I=20 TO 1 STEP -1
130 GET 1,I
140 INPUT#1,A$,R,B$
150 PRINT A$;R;B$
160 NEXT
170 CLOSE
```

APPENDIX C

Assembly Language Subroutines

All versions of BASIC-80 have provisions for interfacing with assembly language subroutines. The USR function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

The address of FRCINT is 103 hex and the address of MAKINT is 105 hex.

Location 107 Hex contains the high byte of the CP/M BIOS entry for use with direct calls to the BIOS. While BASIC is up, the JMP at location zero is a JMP to the "Reset error" of BASIC, not to the BIOS warm boot routine.

MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the system monitor, or the BASIC POKE statement, or (if the user has the MACRO-80 or FORTRAN-80 package) routines may be assembled with MACRO-80 and loaded using LINK-80.

USR FUNCTION CALLS - EXTENDED AND DISK BASIC

The syntax of the USR function is

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in

the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and
FAC-2 contains the middle 8 bits of mantissa and
FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).
FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "BASIC-80"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
PUSH    H       ;save value to be returned
LHLD    xxx     ;get address of MAKINT routine
XTHL            ;save return on stack and
                ;get back [H,L]
RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
    LXI     H       ;get address of subroutine
                    ;continuation
    PUSH    H       ;place on stack
    LHLD    xxx     ;get address of FRCINT
    PCHL
SUB1: .....
```

CALL STATEMENT

User function calls to either Z-80 assembly language subroutines or 6502 assembly language subroutines may be made with the CALL statement. (See CALL, Chapter 3.)

Calling a Z-80 Subroutine

The calling sequence used is the same as that in Microsoft's FORTRAN, COBOL and BASIC compilers.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).
2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter 1 in HL.
    2. Parameter 2 in DE.
    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT (located in the FORTRAN library, FORLIB.REL), and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like:

```
SUBR:   SHLD    P1          ;SAVE PARAMETER 1
        XCHG
        SHLD    P2          ;SAVE PARAMETER 2
        MVI     A,3         ;NO. OF PARAMETERS LEFT
        LXI     H,P3        ;POINTER TO LOCAL AREA
        CALL    $AT         ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subroutine]
        .
        .
        RET                 ;RETURN TO CALLER
P1:     DS      2           ;SPACE FOR PARAMETER 1
P2:     DS      2           ;SPACE FOR PARAMETER 2
P3:     DS      6           ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine $AT follows.

```
00100   ;       ARGUMENT TRANSFER
00200   ;[B,C]  POINTS TO 3RD PARAM.
00300   ;[H,L]  POINTS TO LOCAL STORAGE FOR PARAM 3
00400   ;[A]    CONTAINS THE # OF PARAMS TO XFER(TOTAL-2)
00500
00600
00700           ENTRY   $AT     ;SAVE [H,L] IN [D,E]
00800   $AT:    XCHG
00900           MOV     H,B
01000           MOV     L,C     ;[H,L] = PTR TO PARAMS
01100   AT1:    MOV     C,M
01200           INX     H
01300           MOV     B,M
01400           INX     H       ;[B,C] = PARAM ADR
01500           XCHG            ;[H,L] POINTS TO LOCAL STORAGE
01600           MOV     M,C
01700           INX     H
01800           MOV     M,B
01900           INX     H       ;STORE PARAM IN LOCAL AREA
02000           XCHG            ;SINCE GOING BACK TO AT1
02100           DCR     A       ;TRANSFERRED ALL PARAMS?
02200           JNZ     AT1     ;NO, COPY MORE
02300           RET             ;YES, RETURN
```

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

Calling a 6502 Subroutine

The syntax of a CALL statement to a 6502 subroutine differs from that of a CALL to a Z-80 subroutine in requiring that a percent symbol (%) be used before the variable name.

Up to three parameters may be used when calling a 6502 assembly language subroutine. All of the parameters must be single byte parameters.

They are passed as follows:

1. Parameter 1 in 6502 A register

2. Parameter 2 in 6502 X register
3. Parameter 3 in 6502 Y register

For example:

```
CALL % ROUTINE (10,20,30)
```

10 would be passed in the A register, 20 in the X register and 30 in the Y register. All or some of the parameters may be omitted.

INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. In CP/M BASIC, all interrupt vectors are free.

See "Software and Hardware Details" section of the SoftCard documentation.

APPENDIX D

Converting Programs to BASIC-80 From BASICs other Than Applesoft

If you have programs written in a BASIC other than BASIC-80, some minor adjustments may be necessary before running them with BASIC-80. Here are some specific things to look for when converting BASIC programs.

STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC-80 statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC-80 string concatenation.

In BASIC-80, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	BASIC-80
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	BASIC-80
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC-80 would interpret the second equal sign as a logical operator and set B equal to −1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With BASIC-80, be sure all statements on a line are separated by a colon (:).

MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

APPENDIX E

Summary of Error Codes and Error Messages

Number	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent 5. a call to a USR function for which the starting address has not yet been given 6. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

6 Overflow
The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 Undefined line
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

9 Subscript out of range
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Redimensioned array
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

15 String too long
An attempt is made to create a string more than 255 characters long.

16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,
2. has been modified during a break in execution, or
3. does not exist.

18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

31 Reset error
The RESET key on the Apple keyboard has been pressed.

32 Graphics statement not implemented
Graphics statement is not implemented in MBASIC. It may be used with GBASIC only.

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Disk BASIC-80. Report to Microsoft the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either

greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

68 Disk read only
Disk is write protected, or disk was changed without using RESET first. This error message will usually appear twice. This is normal and should not be cause for concern.

69 Drive select error.
A non-existent drive was selected.

70 File read only
A write was attempted to a file that has been set to "Read Only" with the STAT program.

APPENDIX F

Mathematical Functions

Derived Functions

Functions that are not intrinsic to BASIC-80 may be calculated as follows.

Function	BASIC-80 Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR (-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+ EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X) -EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR (X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1) +1)/X)

INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR (X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX G

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	036	$	072	H
001	SOH	037	%	073	I
002	STX	038	&	074	J
003	ETX	039	'	075	K
004	EOT	040	(	076	L
005	ENQ	041	)	077	M
006	ACK	042	*	078	N
007	BEL	043	+	079	O
008	BS	044	,	080	P
009	HT	045	-	081	Q
010	LF	046	.	082	R
011	VT	047	/	083	S
012	FF	048	0	084	T
013	CR	049	1	085	U
014	SO	050	2	086	V
015	SI	051	3	087	W
016	DLE	052	4	088	X
017	DC1	053	5	089	Y
018	DC2	054	6	090	Z
019	DC3	055	7	091	[
020	DC4	056	8	092	\
021	NAK	057	9	093	]
022	SYN	058	:	094	↑
023	ETB	059	;	095	<
024	CAN	060	<	096	'
025	EM	061	=	097	a
026	SUB	062	>	098	b
027	ESCAPE	063	?	099	c
028	FS	064	@	100	d
029	GS	065	A	101	e
030	RS	066	B	102	f
031	US	067	C	103	g
032	SPACE	068	D	104	h
033	!	069	E	105	i
034	"	070	F	106	j
035	#	071	G	107	k

108	l	115	s	122	z
109	m	116	t	123	{
110	n	117	u	124	|
111	o	118	v	125	}
112	p	119	w	126	~
113	q	120	x	127	DEL
114	r	121	y		

ASCII codes are in decimal.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

INDEX

Software Utilities Manual

PART 5
SOFTWARE UTILITIES MANUAL

Introduction

Several utility programs are provided in the SoftCard package to help you accomplish certain tasks associated with using CP/M on an Apple II computer. The utilities provided are:

FORMAT	Formats blank disks for use with the SoftCard system.
COPY	Makes duplicate copies of disks.
CONFIGIO	Configures I/O for different hardware and software combinations
RW13	Accesses 13-Sector CP/M files from 16-Sector CP/M
CPM56	Configures CP/M for a 56K Language Card System
APDOS	Transfers text files and binary files from Apple DOS to CP/M
UPLOAD/DOWNLOAD	Transfers files from a standard CP/M machine to the Apple CP/M system.

Instructions for using each of these programs are included on the following pages.

Format Notation:

Wherever the format for a statement or command is given, the following rules apply:

1. d:, d_1:, d_2: etc. are disk drives to be specified by you. Acceptable drive names are A:, B:, C:, D:, E: and F:

2. n is an integer 0-9 that will be displayed by the computer according to the particular software you are using.

3. Items in square brackets ([ ]) are optional.

To Prepare Diskettes for Reading and Writing: FORMAT

Command Format:

FORMAT
or
FORMAT d:

Purpose:
The FORMAT program allows you to prepare blank diskettes for reading and writing. You will need to format all blank diskettes before you use them with the Apple SoftCard system.

Instructions:
To format a blank diskette, first insert a diskette that contains CP/M and the FORMAT utility into your disk drive. (These programs are contained on both diskettes in the SoftCard package.) Bring up CP/M as usual. (See "Installation and Operations Manual.") Once you see the CP/M prompt A>, you are ready to begin.

The FORMAT program can be initialized in either of two ways, depending on whether you plan to format *one* or *multiple* diskettes.

Option 1: Use This Method if You Wish To Format Just One Diskette.

1. First, type:

   FORMAT d:

   And press RETURN. If you have two or more disk drives, make sure you have inserted a blank diskette in the specified drive before you press RETURN. If you have a single-drive system, indicate drive A: but leave the disk containing FORMAT in the drive for now.

2. The screen will display the program copyright notice:

   APPLE II CP/M
   nn SECTOR DISK FORMATTER
   (C) 1980 MICROSOFT

   INSERT DISK TO BE FORMATTED IN DRIVE d:

Do so, then press RETURN to start the formatting process.

4. If you have a multi-drive system, the computer will automatically return to CP/M when the formatting is completed. If you have a single-drive system, the computer will indicate:

   FORMAT COMPLETED
   INSERT CP/M SYSTEM DISK IN DRIVE A: AND PRESS RETURN

   Insert a diskette that contains CP/M, then press RETURN. You now have one formatted diskette which is ready to receive CP/M or may be used to store programs and data.

WARNING

Newly formatted disks do *not* have the CP/M operating system on them and *WILL NOT BOOT.* To create CP/M system diskettes from formatted diskettes, use the COPY utility. (See COPY, page 5-7.)

Option 2: Use This Method if You Plan To Format More Than One Disk

1. Type:

   FORMAT

   and press RETURN. The computer will indicate:

   APPLE II CP/M
   nn-SECTOR DISK FORMATTER
   (C) MICROSOFT 1980

   FORMAT DISK IN WHICH DISK DRIVE?

2. Indicate the desired disk drive by typing:

   d:

   then pressing RETURN. If you have two or more disk drives, make sure you have inserted a blank disk in the specified drive before you press RETURN. If you have a single-drive system, indicate drive A: but leave the disk containing FORMAT in the drive for now. If you press RETURN without specifying a drive, the computer will return to CP/M.

3. On a multi-drive system, the computer will begin formatting the diskette in the specified drive. On a single-drive system, the computer will display the message:

   INSERT DISK TO BE FORMATTED IN DRIVE A:
   PRESS RETURN TO BEGIN

   Insert the diskette to be formatted, then press RETURN to begin formatting the diskette.

4. When the formatting is complete, the computer will indicate:

   FORMAT COMPLETE
   FORMAT DISK IN WHICH DRIVE?

   You can continue formatting diskettes in this fashion indefinitely, inserting a blank diskette in the appropriate drive each time. When you have finished formatting diskettes, press RETURN in response to FORMAT DISK IN WHICH DRIVE? to return to CP/M. For a single-drive system, be sure to reinsert a diskette containing CP/M before pressing RETURN.

WARNING

Newly formatted disks do *not* have the CP/M operating system on them and they *WILL NOT BOOT.* To create CP/M system diskettes from formatted diskettes, use the COPY utility. (See COPY, page 5-7.)

NOTE: If you attempt to format a disk that already contains data, the computer will display this message:

DISK IN DRIVE d: WILL BE ERASED.
CONTINUE(Y/N)?

If you answer Y, the computer will re-format the disk, completely erasing it.

If you answer N, the computer will again ask:

FORMAT DISK IN WHICH DRIVE?

allowing you to insert another diskette or specify another drive.

If you simply press RETURN, the program will be terminated and the computer will return to CP/M.

Error Messages

If the FORMAT is not successful, the computer will indicate one of three error messages.

DISK WRITE PROTECTED	There is a write protect tab on the diskette you want to format. Remove the write protect tab and repeat the FORMAT process.
DISK I/O ERROR	The computer is unable to format the diskette for some reason. Check to be sure you have a diskette in the disk drive and that the disk drive door is closed.
COMMAND ERROR	The command could not be understood. Retype the command line, making sure it is in the correct format.

After an error is encountered, the computer returns to "FORMAT DISK IN WHICH DRIVE?"

To Make Copies of Diskettes: COPY

Command Format:

COPY d_1: = d_2:
Option: /S allows you to copy CP/M (tracks 0-2) only.

Purpose:
The copy utility allows you to make copies of Apple CP/M disks. Copy is also used to create CP/M system disks from newly formatted disks.

Instructions:
To make a copy of a diskette onto a blank, formatted diskette, first insert a diskette containing the COPY utility and CP/M (these programs are contained on the SoftCard diskettes) in your disk drive and bring up CP/M as usual. (See "Installation and Operations Manual.") Once you see the CP/M prompt A>, you are ready to begin.

1. Type:

   COPY d_1: = d_2:

   d_1: is the drive to which you wish to copy, and d_2: is the drive from which you wish to copy. If, for example, you indicate A: = B:, the computer will copy from drive B: and write to drive A:. If you have a single-drive system, type A: = A:.

   If you just typed COPY, the computer will return an asterisk (*) prompt and wait for you to enter a command line (d_1: = d_2:) before proceeding further.

   After the command line is typed, the computer will display the message:

   APPLE II CP/M
   xx-SECTOR DISK DUPLICATION PROGRAM
   (C) 1980 MICROSOFT

   (If you have a single-drive system, proceed to step 3.)

2. On a multi-drive system, the computer will also display the message:

   INSERT MASTER DISK IN d_1:
   INSERT SLAVE DISK IN d_2:
   PRESS RETURN TO BEGIN

Insert the disk from which you wish to copy in drive d_1: and the diskette to which you wish to copy in drive d_2:. Press RETURN to begin copying. (Proceed to Step 4.)

3. For a single-drive system, the computer will display the message:

   INSERT MASTER DISK
   PRESS RETURN TO CONTINUE

   Remove the SoftCard diskette and insert the diskette of which you wish to make a copy. Press RETURN. After some diskette activity, the computer will display the message:

   INSERT SLAVE DISK PRESS RETURN

   Remove the disk you wish to copy and insert a blank formatted disk, then press RETURN. After some disk activity, the above message will be displayed again. Repeat Step 3, until the COPY COMPLETE message is displayed. (See Step 4.)

4. When a copy is completed, the computer will display the message:

   COPY COMPLETE
   DO YOU WISH TO MAKE ANOTHER COPY? (Y/N)
   PRESS RETURN

   Press Y to make another copy. Insert a new blank (formatted) disk in drive d_2 before pressing RETURN. (Also, see "To Create CP/M System Disks.")

   Press N to return to CP/M command level.

To Create CP/M System Disks:

The /S option allows you to *copy the CP/M operating system only* from one diskette to another. Other files on either disk are not affected. *You will need to copy CP/M onto each disk* you wish to use with the SoftCard system. (Diskettes must be formatted before CP/M can be copied to them. See FORMAT, page 5-3.) The command format for initiating the program is:

COPY d_1:=d_2:/S

d_1: is the drive from which you want to copy and d_2: is the drive to which you want to copy. /S is the switch that tells the computer to only copy CP/M. Otherwise, follow the instructions above for copying disks.

WARNING:

Unless you use the /S option, all files on the destination disk will be erased. Also, the diskette onto which you wish to copy *must be formatted* before it may be copied.

Error Messages

If the COPY is not successful, the computer will indicate one of three error messages.

DISK WRITE PROTECTED	There is a write protect tab on the diskette you want to copy. Remove the write protect tab and repeat the COPY process.
DISK I/O ERROR	The computer is unable to access the diskette for some reason. Check to be sure you have diskettes in the specified disk drives and that the disk drive doors are closed.
COMMAND ERROR	The command could not be understood. Retype the command line, making sure it is in the correct format.

To Access 13-Sector CP/M Files from 16-Sector CP/M: RW13

Command Format:

RW13 d_1:
and
RW13 X (To convert drive back to 16-Sector)

Purpose:
RW13 allows a 16-Sector system to Read and Write to a 13-Sector diskette. When RW13 is run, 13-Sector files can be accessed by 16-Sector CP/M. Used with PIP, RW13 is especially useful for transferring files from a 13-Sector to a 16-Sector diskette. The RW13 X command is used to convert the drive back to 16-Sector. RW13 is found only on the 16-Sector SoftCard diskette and requires a system with two or more disk drives. Drive A: cannot be converted to 13-Sector operation.

Instructions:
Insert a diskette that contains RW13 and CP/M into your disk drive (these programs are contained on the diskettes in the SoftCard package) and bring up CP/M as usual. (See "Installation and Operations Manual.") When you see the CP/M prompt A>, you are ready to begin.

1. Type:

   RW13 d_1:

   where d_1 is a disk drive B:-F:. You may specify any drive *except drive A:*. Press RETURN.

2. The computer will display the message:

   APPLE II CP/M
   13-SECTOR DISK CONVERSION
   (C) 1980 MICROSOFT

   DRIVE D_1: CONVERTED TO 13 SECTOR OPERATION.

3. Any 13-Sector diskette inserted into the "converted" drive can now be read from or written to by any CP/M program. In this mode, you can use PIP (See "Installation and Operations Manual") to transfer files from a 13-Sector diskette in a converted drive to a 16-Sector diskette in a non-converted drive. Or you can use any other CP/M software for a 13-Sector disk system. NOTE: Do not use the COPY program.

4. When you are finished, convert all of the drives back to 16-Sector by typing the command:

   RW13 X

   and pressing RETURN. The drive will be returned to 16-Sector operation.

NOTE: RW13 occupies 4K of memory, so while it is in effect, there is 4K less memory available to programs.

To Configure CP/M for a 56K System: CPM56

Command Format:

CPM56 d:

Purpose:
To update CP/M for use with a 56K Language Card System. If you have a 56K system, you will want to perform this conversion before using CP/M, to take advantage of your system's additional memory. If you have a 48K system, you will not need this utility.

Instructions:
1. Insert a diskette containing CP/M and CPM56 into one of your Apple disk drives. (These programs are contained on the 16-Sector disk in the SoftCard package.) Boot up CP/M as usual. (See "Installation and Operations Manual.") When you see the CP/M prompt A>, you are ready to begin.

2. Type:

   CPM56 d:

   Insert a diskette that contains a copy of CP/M into the specified drive. (See the COPY utility for instructions for copying CP/M.) Press RETURN.

3. Once you press RETURN, the computer will automatically configure the copy of CP/M in the specified drive for a 56K system. When the conversion is complete, the computer will display the message:

   DISK IN DRIVE d: HAS BEEN UPDATED TO 56K.

   You now have a diskette containing CP/M for a 56K system.

NOTE: If you have used the CONFIGIO utility to define special characters, those characters will be preserved when CP/M is updated.

Error Messages

If the 56K configuration is not successful, one of the following error messages will be displayed.

DISK I/O ERROR	The drive cannot access the diskette for some reason. Check to be sure there is a diskette in the drive and the drive door is closed.
DISK WRITE PROTECTED	There is a write protect tab on the disk you wish to configure. Remove the write protect tab and repeat the CPM56 process.
COMMAND ERROR	The command could not be understood. Retype the command line, making sure it is in the correct format.

To Transfer Files from Apple DOS to CP/M: APDOS

Command Format:

APDOS d_1:filename.typ = d_2:filename
or
APDOS d_2:

Purpose:
To transfer text files and binary files from Apple DOS to CP/M. APDOS cannot read BASIC files and it cannot write to an Applesoft diskette. If you want to transfer a DOS 3.2 file to a 16-sector CP/M disk, you must first use RW13 (see page 5-10) to convert the drive to 13-sector operation.

Instructions:

1. Insert a diskette containing both APDOS and CP/M into your Apple disk drive. (Both programs are contained on the diskettes in the SoftCard package.) Bring up CP/M as usual. (See "Installation and Operations Manual.") When you see the CP/M prompt A>, you are ready to begin.

2. Type:

   APDOS

   and hit RETURN. The computer will print

   APPLE II CP/M
   APPLE DOS CP/M FILE TRANSFER
   (C) 1980 MICROSOFT

   and then print a colon prompt. If you type CAT d: the catalog of the Apple DOS disk in drive d: will be displayed.

3. Type

   [d_1:] Fname.typ = [d_2:] Filename

   to transfer the Apple DOS file "Filename" (in drive d_2) to the CP/M file Fname.typ in drive d_1. If drives are not specified, d_1: defaults to A: and d_2: to B:.

4. To continue copying files from the Apple DOS to the CP/M diskette type:

   Fname.typ = Filename

   The computer will assume the same disk drives as previously specified. If you wish to change disk drives, type the APDOS command in its original format.

All characters of an Apple DOS text file transferred using APDOS have their high order bits cleared. Apple DOS binary files retain the four bytes of address and file-length information at the beginning of the file. Actual data begins therefore at the *fifth* byte of the file. See the Apple DOS 3.2 or DOS 3.3 manual for details on the format of text and binary files.

Use the following procedure for transferring either Applesoft or Integer BASIC programs under Apple DOS to CP/M. This procedure converts the Integer BASIC or Applesoft program into a textfile which can be transferred using APDOS:

1. Boot an Apple DOS 3.2 or 3.3 disk that contains the program you wish to transfer, and LOAD the program as usual.
2. Enter the following program line as the first line of the program:
   0 PRINT "ctrl-D OPEN APPLEPROG": PRINT "ctrl-D WRITE APPLEPROG": POKE 33,33: LIST: PRINT "ctrl-D CLOSE": END
   (ctrl-D is an embedded ctrl-D character typed by you.)
3. RUN the program. When the program ends, you will have a text file on your Apple DOS disk called APPLEPROG that is actually a text copy of your program.
4. Boot your CP/M disk.
5. Type APDOS.
6. Insert the Apple DOS disk into drive B: (or into A: with a single-drive system).
7. If you have a multi-drive system, type APPLE.BAS = APPLEPROG and press RETURN. If you have a single-drive system, type APPLE.BAS = A:APPLEPROG and press RETURN.
8. Exit APDOS by typing ctrl-C.
9. Enter BASIC by typing MBASIC or GBASIC.
10. Type LOAD "APPLE" and press RETURN.
11. Delete line zero (the line entered by step 2).
12. You have now transferred a copy of your Applesoft or Integer BASIC program to Apple CP/M, which probably *will not run* at first try. You will probably be required to edit the program, changing the POKEs, PEEKs, CALLs, and disk file statements into their equivalent Microsoft BASIC statements. Note that

most POKEs, PEEKs and CALLs simply will not work with Microsoft BASIC. They can, however, usually be replaced. See the Microsoft BASIC Reference Manual for more information on converting programs to Microsoft BASIC.

Error Messages
If the Apple DOS to CP/M transfer is not successful, one of the following error messages will be displayed:

DISK I/O ERROR	The drive cannot access the diskette for some reason. Check to be sure there is a diskette in the drive and the drive door is closed.
DISK WRITE PROTECTED	There is a write protect tab on one of the diskettes. Remove the write protect tab and repeat the APDOS process.
COMMAND ERROR	The command could not be understood. Retype the command line, making sure it is in the correct format.

To Configure the Apple CP/M Operating Environment: CONFIGIO

Purpose:
The CONFIGIO utility is used to configure the Apple CP/M operating environment to the user's particular system configuration.

Instructions:
Insert a CP/M system disk that contains MBASIC (or GBASIC) and CONFIGIO into a disk drive. (These programs can be found on either of the SoftCard disks). Bring up CP/M as usual. (See "Installation and Operations Manual.") When you see the CP/M prompt A>, type:

MBASIC CONFIGIO

and press RETURN.

If you are using the standard Apple, (i.e., no external terminal), the computer will ask

CAN YOUR APPLE DISPLAY LOWER CASE (Y/N)?

If your Apple is equipped with hardware that allows the direct display of lower case text on the Apple screen, respond with Y. Otherwise, answer with

an N. An N response causes lower case characters to be converted to upper case before they are printed on the Apple screen. (This can be made permanent with option 4 below.)

The computer will then display the menu:

```
++ I/O CONFIGURATION PROGRAM ++

1. CONFIGURE CP/M FOR EXTERNAL TERMINAL

2. REDEFINE KEYBOARD CHARACTERS

3. LOAD USER I/O DRIVER SOFTWARE

4. READ/WRITE I/O CONFIGURATION BLOCK

Q. QUIT PROGRAM

SELECT -
```

Select 1,2,3,4 or Q to perform the following functions:

1. Configure CP/M for External Terminal – Allows you to specify the character sequences required by your particular software or hardware to execute a particular screen function. Once these sequences are set up properly, your system can translate these character sequences between your terminal and your software. See page 5-17.

2. Redefine Keyboard Characters – Allows you to redefine the ASCII value that is assigned to any particular key on the keyboard. Using this option you can make one key (for example the 3 key) generate a character not usually associated with it (for example an ! mark). Or more usefully, you can make pressing Ctrl-V generate a [. This option is especially useful for making characters available that are not normally found on the Apple keyboard. See page 5-23.

3. Load User I/O Driver Software – Allows you to load and bind I/O driver software into the I/O Configuration Block for use with non-standard Apple peripherals, etc. See page 5-25.

4. Read/Write I/O Configuration Block – Allows you to read or write the I/O Configuration Block from or to the disk. Changes made using options 1-3 of CONFIGIO are made permanent by writing the I/O Configuration Block to the disk. See page 5-26.

Q Quit program – Exits program and returns to BASIC.

More information about each of these functions can be found in the Software Details Manual, pages 2-1 to 2-34. Below is an explanation of the use of each of the four functions:

1. Configure CP/M for External Terminal

Introduction
Most video terminals (including the Apple 24 × 40 screen) support a number of special screen functions such as Clear Screen, Highlight Text, and Address Cursor. This is done by sending a special character sequence to the terminal to perform a particular function. Most applications software (such as screen-oriented word processors), however, are usually only capable of working with a small number of terminals – those that "understand" the screen character sequences sent by the software.

Apple CP/M provides you with translation tables for handling the screen function character sequence requirements of your hardware and software. The procedure for setting up Apple CP/M for your particular system configuration is outlined below.

NOTE: See the "Software and Hardware Details," page 2-12 for more information regarding terminal configuration.

After you select number 1 from the main menu, the computer will display another menu:

+ TERMINAL SCREEN FUNCTION DEFINITION +

FUNCTION		SOFTWARE	HARDWARE
CLEAR SCREEN		ESC *	FF
CLR TO EOS		ESC Y	VT
CLR TO EOL		ESC T	GS
LO-LITE TEXT		ESC)	SO
HI-LITE TEXT		ESC (	SI
HOME CURSOR		RS	EM
ADDRESS CURSOR		ESC =	RS
XY COORD OFFST		32	32
XY XMIT ORDER	:	YX	XY
CURSOR UP		VT	US
CURSOR FORWARD		FF	FS

1. SOROC IQ 120/IQ 140
2. HAZELTINE 1500/1510

3. DATAMEDIA
4. OTHER
Q. QUIT

SELECT –

These are the Hardware and Software Screen Function Tables.

NOTE: When configuring CP/M for an external terminal, you should remove the interface card from slot 3 and use the standard Apple video. Once the configuration process is complete, you can reinsert the card.

The contents of the Hardware and Software Screen Function Tables are displayed using standard ASCII character names. A NUL entry in either table means that the function is not available.

Tables set up for certain other common terminals are available and can be selected by typing the appropriate number as indicated below:

1. SOROC IQ 120/IQ 140 – Type 1 to configure either the Software or Hardware Screen Function Table for a SOROC IQ 120/IQ 140 video terminal. If you type 1, you will then be asked which table (hardware or software) is to be reconfigured.

Since the Screen Function Tables are initially set up for use with a SOROC IQ 120/IQ 140 video terminal, you will not need to change them unless you wish to redefine the Software Screen Function Table. NOTE: When the SOROC terminal is powered on, it defaults to "Hi-lite" text mode. CP/M sends the "Lo-lite" character sequence when the system is booted.

2. Hazeltine 1500/1510 – Press 2 to configure the Hardware Screen Function table for use with a Hazeltine 1500/1510 video terminal.

Selection 2 should only be used to set up the Hardware Screen Function Table. Because of the non-standard way in which Apple CP/M handles the Hazeltine cursor addressing function (no X-Y coordinate offset is used), it is NOT advisable to use the Hazeltine screen function sequences in the Software table. Set up the Hardware table for the Hazeltine, and the Software table for some other common terminal, such as the SOROC IQ 120/140 (#1).

3. Datamedia – Type 3 to configure the Hardware Screen Function Table for use with a Datamedia-style terminal. This is the configuration used for the 24 × 80 video terminal boards such as the Videoterm or the Sup-R-Term.

Selection 3 should be used to set up the Hardware Screen Function Table only, because the Datamedia Terminal sequences are not usually supported by CP/M *software*. You should set up the hardware table for use with the 24 × 80 video board, and the Software Table for some other common terminal

such as the SOROC IQ 120/140 (#1). Hi-lite text and Lo-lite text (INVERSE and NORMAL) are not supported by all Datamedia-type terminals, thus the table entries we've specified for these functions are arbitrary. This was done so that these entries would be non-zero.

4. Other –Type 4 if you want to set up either the Software or Hardware tables for any terminal not accounted for by the other menu selections. This selection is used to change one or all of the screen function character sequences. When you type 4, the computer will display yet another menu:

```
+ + SCREEN FUNCTION DEFINITION + +

 1 - LEAD-IN CHARACTER
 2 - CLEAR SCREEN
 3 - CLR TO EOS
 4 - CLR TO EOL
 5 - LO-LITE TEXT
 6 - HI-LITE TEXT
 7 - HOME CURSOR
 8 - ADDRESS CURSOR
 9 - CURSOR UP
10 - CURSOR FORWARD
 Q - QUIT

 SELECT –
```

You can now change any of the values in the "Terminal Screen Function Definition" Table.

NOTE: The appropriate screen function command characters for your terminal can be found in the manual for that terminal. To find out which codes are transmitted by a particular program (i.e., a word processor), consult the manual for the particular program.

Select a number 1 through 10 to define the character sequences for any of the following functions:

Number	Title	Description
1	Lead-in char	Defines the Lead-in character – the character (usually an ESC) that precedes the screen function command character. A particular screen function may or may not require a lead-in character.
2	Clear screen	Clears the screen and places the cursor at the top left corner of the screen.

3	Clear to EOS	Clears the screen from the cursor to the end of the screen
4	Clear to EOL	Clears the screen from the cursor to the end of the line.
5	Lo-lite text	Sets the normal video mode for displaying text.
6	Hi-lite text	Sets inverse or double intensity video mode depending on which of these your terminal supports.
7	Home cursor	Puts the cursor at the top left corner of the screen but does not clear the screen.
8	Address cursor	Tells the terminal to go to a certain cursor address that is defined by the next two characters entered.
	XY Coord. Offset	Defined as part of #8. The XY coordinate offset is the number that is added to the X and Y coordinates when they are sent to the terminal (Usually 32).
	XY Xmit Order	Also defined as part of #8. Establishes the order that coordinates are transmitted. Must be either XY or YX (Usually YX).
9	Cursor Up	Moves the cursor up one line on the screen.
10	Cursor Forward	Moves the cursor forward on a line without deleting the character under the cursor.

To assign the appropriate character sequences to any of these functions, just type its corresponding number and hit RETURN.

Choose number 1 if you wish to specify a screen function lead-in character. The computer will display:

LEAD-IN CHAR :

Enter the lead-in character required. Characters may be entered in any one of the following formats:

2 or 3-character ASCII name

CTRL-ch — where ch is any character

ch — where ch is any keyboard character

LC-ch — LC- denotes that the following character is to be lower case. This can be used in place of the lower case character if your keyboard has no lower case.

ASCII hexadecimal code (preceded by &H)
May be used if the character cannot be typed. (See the ASCII Code Chart in the "Software Details Manual.")

After you have entered the lead-in character, the computer will respond:

SOFTWARE OR HARDWARE (S/H)?

Press S or H according to whether the lead-in character is to be used in the Software Screen Function Table or the Hardware Table.

To define any of the other screen functions, simply type the corresponding number for that function and the computer will prompt you to input the command character for that particular function. For instance, if you typed 2, the computer would prompt:

CLEAR SCREEN :

Enter the character to be used for that particular function (in any of the formats listed above), then press RETURN. Do not include the lead-in character if it is required. If the function is not available, enter NUL (ASCII 00). Characters may be entered in any of the formats shown above.

After you enter in the character, you will be asked:

REQUIRE LEAD-IN (Y/N)?

Type Y if a lead-in character is required for execution of this function. Type N if none is required.

Next, the computer will ask:

SOFTWARE OR HARDWARE (S/H)?

Type S to make the change you've indicated to the Software Screen Function Table. Type H to modify the Hardware Table.

The computer will return to the "SCREEN FUNCTION DEFINITION" menu and wait for you to select another number or Quit. You may make as many changes to the tables as you wish in this way. (The process for changing 8, Address Cursor, differs somewhat. See below.) Typing Q from this menu will redisplay the Screen Function Tables.

If you select 8, Address Cursor, you will be lead through the process as above up to:

Require Lead-in (Y/N)?

After you answer this question by pressing Y or N, the computer will print:

XY COORD OFFST :

Enter a number to indicate the number of spaces that is to be added to the X/Y coordinates before they are transmitted. Finally, you will be asked

XY XMIT ORDER :

If the X and Y coordinates are transmitted Y first then X, enter YX. If the coordinates are transmitted X then Y, enter XY.

The computer will then pick back up with the questions:

SOFTWARE OR HARDWARE (S/H)?

and continue as with any of the other functions.

Notes on CP/M Terminal Configuration
Limitations: The Screen Function Tables may only be used with one or two character sequences: a single control character, or any character preceded by a lead-in character. Longer sequences can be implemented with a special purpose I/O driver. See the Software and Hardware Details (Part 2) for more information.

In order to make changes to the Screen Function Tables permanent, you must use option 4 of the "I/O Configuration Program" menu. If you don't write the I/O Configuration Block onto a CP/M disk, the changes you've made will be "forgotten" the next time your system is re-booted.

No matter what values you've inserted in the Tables, they will work with the normal Apple 24×40 screen **if and only if ALL table entries are non-zero.**

The Software Screen Function Table must match the sequences the software will send to perform screen functions, and the Hardware Screen Function Table must match the sequences expected by the hardware device.

Microsoft BASIC will work with any terminal as long as the Hardware and Software Screen Function Tables are set up with non-zero entries in all of the nine functions.

It is usually a good idea to set up the Software Screen Function Table to emulate a SOROC IQ 120/140 type terminal. This is a common configuration that is supported by a majority of CP/M software.

2. Redefine Keyboard Characters

Keyboard Character Redefinition is used to make available characters to the user that are not normally available.

If you select number 2, the computer will display:

```
        + + KEYBOARD CHARACTER DEFINITION + +

Ctrl-K  ->      [
Ctrl-@  ->      RUB
Ctrl-U ->       Ctrl-I
Ctrl-B  ->      \
ADD/DELETE/QUIT (A/D/Q) -
```

Shown in the table are three characters that have already been redefined: Ctrl-K, Ctrl-@, and Ctrl-B. These characters have been redefined to be often used characters that are normally unavailable on the Apple keyboard – "[", RUBOUT, and "\".

You can define additional characters, delete characters or return to the main menu by selecting A, D or Q, respectively.

If you type A to add to the table, the computer will display:

```
CHAR:
```

Enter the character to be redefined. A character may be entered in any one of several formats:

ch — where ch is any character

2 or 3-character ASCII name

Ctrl-ch — where ch is any character

LC-ch — The LC- prefix is used to enter lower case characters when lower case is not available.

ASCII hexadecimal code (preceded by &H)
(may be used if the character cannot be typed. See the ASCII Code chart in the "Software and Hardware Details" section of this manual.)

If, for example, you wanted to redefine Ctrl-C as a NUL (ASCII 00) in order to prevent a user's ability to break out of a BASIC program by typing Ctrl-C, you would first type:

CTRL-C

after the CHAR: prompt.

If the character you have typed in is acceptable, the computer will prompt you to enter the new definition of the character with an arrow. With the example above:

CTRL-C -> NUL

where you type in NUL.

If your response is not acceptable, the computer will erase your previous input and wait for you to type an acceptable character entry.

Once you have hit RETURN, the list of redefined keyboard characters will again be displayed with the new redefinition added to the list. Now, every time you type Ctrl-C, a NUL character is actually entered. (Oh, by the way, try to Ctrl-C out of the CONFIGIO program!)

You can delete a keyboard character redefinition from the table by typing D. For example, to delete the entry made in the example above, type D. The computer will prompt you for the keyboard character redefinition to be deleted (CHAR:), to which you type: CTRL-C and hit RETURN. The list will be displayed with the Ctrl-C -> NUL entry deleted.

Type Q to return to the main menu.

Notes on Keyboard Character Redefinition

It is usually a good idea to delete keyboard character redefinitions if they do not apply to your keyboard. If for example, your keyboard has a RUBOUT key, you should delete the Ctrl-@ redefinition entry.

Redefining Ctrl-C as a NUL to prevent breakout out of BASIC programs with Ctrl-C is a useful idea, but it can present problems when in CP/M command mode. Ctrl-C is usually used by CP/M to re-initialize the system.

Some terminal devices do some redefinition of their own. For instance, with the Videx Videoterm, Ctrl-A is used to toggle upper and lower case input mode. Since Ctrl-A is also used in BASIC to enter EDIT mode, you may want to redefine some other character as Ctrl-A (such as Ctrl-W).

3. Load User I/O Driver Software

I/O software intended for use with non-standard hardware, etc., must be loaded and patched into the I/O Configuration Block. This is done with option 3. The program data that is loaded from disk must be of a special internal format. See the "Software and Hardware Details" section for more information.

If you type 3, the computer will display:

```
+ + LOAD USER I/O DRIVER SOFTWARE + +

OBJECT FILE NAME?
```

Type the name of the data file that contains the program to be loaded into the I/O Configuration Block and press RETURN. The computer will display the message:

```
LOADING...
```

as it loads and patches the routines from disk into the I/O Configuration Block. After the patches have been made, the computer will return to the main menu.

4. Read/Write I/O Configuration Block

This function allows you to write the I/O Configuration Block to disk or read the I/O Configuration Block from a disk into memory. This allows you to examine and modify the I/O Configuration Block on any CP/M disk and then save it to as many disks as desired. Writing the I/O Configuration Block

to a CP/M system disk makes all changes made by the CONFIGIO program permanent.

If you type 4, the computer will display:

+ READ/WRITE I/O CONFIGURATION BLOCK +

READ OR WRITE (R/W)?

If you type W, the computer will display:

DESTINATION DRIVE (A:-F:)?

Insert a CP/M system disk and select the appropriate drive. The I/O Configuration Block on the disk will be replaced with the one currently in memory. Use W to make permanent any changes you've made under options 1-3. As soon as the process is complete, you will be returned to the main menu.

If you type R, the computer will ask

SOURCE DRIVE (A:-F:)?

Insert a CP/M system disk and select the appropriate drive name. The I/O Configuration Block will be read from the CP/M disk and loaded into memory. Once the operation is complete, you will be returned to the main menu.

NEVER attempt to read or write the I/O Configuration Block on a disk that has CP/M configured for a different memory size than the system on which it is running. (i.e., don't try to read a 44K I/O Configuration Block using a system that runs 56K CP/M). Always make sure that the disk you which to read or write has the same CP/M configuration as the disk in drive A:.

To Transfer CP/M Files from Another Computer: Download and Upload

WARNING: USE OF THESE PROGRAMS ASSUMES FAMILIARITY WITH 8080 ASSEMBLY LANGUAGE PROGRAMMING. THESE PROGRAMS ARE INTENDED FOR EXPERIENCED PROGRAMMERS ONLY!

Purpose:
The DOWNLOAD and UPLOAD utilities enable the user to transfer CP/M files from another CP/M machine to the Apple by means of an RS-232 serial data link. The UPLOAD utility is intended to be typed into the non-Apple CP/M system (referred to as the "source" machine) and configured for the source machine's particular I/O environment using the DDT utility of CP/M. To use DOWNLOAD, you must have an Apple Communications Interface or CCS 7710A serial card plugged into slot 2. DOWNLOAD is found on both of the supplied SoftCard disks.

To use the Download and Upload Utilities you need:

1. A working knowledge of the CP/M DDT program and 8080 assembly language programming.

2. A CP/M based computer system (in addition to your Apple II) with an RS-232 Serial I/O port other than the port used for console I/O.

3. Either an Apple Communications Interface or California Computer Systems 7710A Serial Interface installed in slot 2 of the Apple.

Instructions:

Step 1
Using DDT, enter the following machine language program, UPLOAD, into the source machine starting at location 0163H:

```
0163 3A 80 00 B7 11 D7 01 CA CC 01 CD 03 01
0170 0E 0F 11 5C 00 CD 05 00 3C 11 E5 01 CA CC 01 3E
0180 52 CD 43 01 CD 23 01 FE 53 C2 7F 01 3E 47 CD 43
0190 01 11 06 02 CD D2 01 0E 14 11 5C 00 CD 05 00 B7
01A0 C2 C9 01 21 80 00 0E 00 16 80 7E CD 43 01 A9 4F
01B0 23 15 C2 AA 01 79 CD 43 01 CD 23 01 FE 42 CA A3
01C0 01 FE 47 CA 97 01 C3 B9 01 11 F4 01 CD D2 01 C3
01D0 00 00 0E 09 C3 05 00 43 6F 6D 6D 61 6E 64 20 45
01E0 72 72 6F 72 24 46 69 6C 65 20 6E 6F 74 20 66 6F
01F0 75 6E 64 24 0D 0A 55 50 4C 4F 41 44 20 43 6F 6D
0200 70 6C 65 74 65 24 55 70 6C 6F 61 64 69 6E 67 2E
0210 2E 2E 24 FE
```

Enter the following three bytes at location 0100H:

C3 63 01 (This is a JMP 0163H)

When you're finished, double and triple check that you have entered the data correctly. Use the DDT "L" command to list the program and compare the listing with the listing of UPLOAD on page 5-31. Before you attempt the patches below, you should save the program by exiting DDT and typing SAVE 2 UPLOAD.COM. (For more information on the use of DDT, see the "CP/M Reference Manual").

Step 2:

The next step is to patch the UPLOAD program to recognize the serial I/O port on the source machine. This is done by using DDT to write the following three subroutines. Each routine must begin at the address listed next to the subroutine description below. 32 bytes are allocated for each.

1. 0103H — Initialize Serial Port – This routine must initialize the serial port on the source machine. (Baud rate, data format, etc.) The data format should be set up for 8 data bits, 1 stop bit, no parity, for compatibility with the Apple Com Card and the CCS 7710A card.

2. 0123H — Return Serial Port Status – If no character is available at the serial port, this subroutine must return A = 00. If a byte is available, the routine should read the byte and return it in the A register.

3. 0143H — Write to Serial Port – Output a byte to the serial port. Must save all registers including A.

Once these routines have been written and patched into the UPLOAD program, it should again be saved using the SAVE command.

Step Three:

Next, wire up a connecting cable from the Apple to the source machine. One port must be wired up as a send (DTE) device, and the other a receiver (DCE). Sometimes the Xmit and Rcve lines (Pins 2 & 3) need to be reversed, or certain handshaking lines need to be wired together. If you are using a CCS 7710A serial card, wire pins 4, 6 and 20 together on the Apple end. Make sure that the data formats expected by the two serial ports are the same.

Step Four:

Once UPLOAD has been patched and the cable has been made up, you are ready to begin.

On the Apple, type

DOWNLOAD fname.typ

where fname.typ is the name the transferred program will be saved under.

Over on the source machine, type

UPLOAD fname.typ_2

where fname.typ_2 is the name of the file you want to transfer. As soon as communication is established, the Apple will display .

DOWNLOADING

and the source machine will display:

UPLOADING...

As each 128 byte record is transferred successfully, a period (".") is printed on the Apple screen. If an error is detected during transfer of a 128 byte record, a "B" is printed, and the record is retried.

When the transfer is complete, the source machine will display, appropriately,

UPLOAD COMPLETE

and return to CP/M.

When this message appears on the source machine, type Ctrl-C from the Apple keyboard. The disk will whir a bit, and soon the Apple will display

DOWNLOAD COMPLETE

and return to CP/M.

This process must be repeated for each file to be transferred. To transfer more than one file at a time, use the CP/M program, SUBMIT. You might also want to modify these programs to allow the use of a non-standard interface card, etc.

SOURCE LISTING: UPLOAD

```
                        ;
                        ;       UPLOAD
                        ;
                        ; WRITTEN 5/80 BY NEIL KONZEN
                        ;    (C) 1980 MICROSOFT
                        ;
0000 =          BOOT    EQU     0000H           ;BOOT SYSTEM
0005 =          BDOS    EQU     0005H           ;BDOS ENTRY POINT
005C =          FCB     EQU     005CH           ;DEFAULT FCB
0080 =          BUFFER  EQU     0080H           ;DEFAULT BUFFER ADDR
                        ;
0100                    ORG     0100H           ;START AT TPA
                        ;
0100 C36301     UPLOAD: JMP     ENTRY           ;JUMP AROUND ALL THESE
                        ;
                        ;
                        INIT:                   ;INITIALIZE SERIAL PORT
                        ;
                        ; THIS SUBROUTINE SHOULD DO ANY INITIALIZATION
                        ; OF THE SERIAL PORT THAT MAY BE REQUIRED.  IF
                        ; NONE IS REQUIRED, A 'RET' WILL DO.
                        ;
0103                    DS      32              ;SPACE FOR ROUTINE
                        INPSTS:                 ;INPUT STATUS/READ
                        ;
                        ; THE INPUT STATUS/READ ROUTINE RETURNS ZERO IN [A] IF NO
                        ; BYTE IS AVAILABLE.  IF A BYTE IS AVAILABLE,
                        ; THE BYTE SHOULD BE READ AND RETURNED IN THE
                        ; [A] REGISTER.
                        ;
0123                    DS      32              ;SPACE FOR ROUTINE
                        OUTPUT:                 ;SEND A BYTE TO APPLE
                        ;
                        ; THE OUTPUT ROUTINE SHOULD TRANSMIT THE BYTE IN THE
                        ; [A] REGISTER TO THE APPLE VIA THE SERIAL
                        ; PORT.  ALL REGS INCLUDING [A] SHOULD BE SAVED.
                        ;
0143                    DS      32              ;SPACE FOR ROUTINE
                        ;
                        ;
0163 3A8000     ENTRY:  LDA     BUFFER          ;MAKE SURE HE TYPED SOME SORT OF FILE NAME
0166 B7                 ORA     A               ;A NON-ZERO NO. OF CHARS IN CMD LINE?
0167 11D701             LXI     D,CMDMSG        ;DEFAULT TO COMMAND ERROR MESSAGE
016A CACC01             JZ      EXIT            ;QUIT.
016D CD0301             CALL    INIT            ;INITIALIZE SERIAL PORT
0170 0E0F               MVI     C,15            ;OPEN FILE COMMAND
0172 115C00             LXI     D,FCB           ;POINT TO FCB
0175 CD0500             CALL    BDOS            ;OPEN IT UP
                        ;
0178 3C                 INR     A               ;FF BECOMES ZERO
0179 11E501             LXI     D,FNFMSG        ;DEFAULT TO FILE NOT FOUND MSG
017C CACC01             JZ      EXIT            ;NO FILE
                        ;
                        ;  SEND A BUNCH OF 'R'S UNTIL DOWNLOAD ANSWERS
                        ;
017F 3E52       RDYLP:  MVI     A,'R'           ;SEND 'R' FOR 'READY'
0181 CD4301             CALL    OUTPUT          ;SEND VIA SERIAL LINE
0184 CD2301             CALL    INPSTS          ;DID HE RESPOND?
0187 FE53               CPI     'S'             ;'S' FOR 'SET'
0189 C27F01             JNZ     RDYLP           ;THEN TRY AGAIN
                        ;
018C 3E47               MVI     A,'G'
018E CD4301             CALL    OUTPUT          ;SEND TO DOWNLOAD
                        ;
0191 110602             LXI     D,WRKMSG        ;TELL HIM WE'RE DONG IT
0194 CDD201             CALL    PRMSG
                        ;
0197 0E14       READ:   MVI     C,20            ;READ SEQUENTIAL FUNCTION
0199 115C00             LXI     D,FCB
019C CD0500             CALL    BDOS            ;GO READ 128 BYTES
019F B7                 ORA     A               ;ERROR?
01A0 C2C901             JNZ     EOF             ;END OF FILE
01A3 218000     TRYAGN: LXI     H,BUFFER        ;POINT TO THE 128 BYTES
01A6 0E00               MVI     C,0             ;CHECKSUM = 0
01A8 1680               MVI     D,80H           ;BYTE COUNT = 128
```

```
                        ;
01AA 7E         LOOP1:  MOV     A,M             ;GET CHAR
01AB CD4301             CALL    OUTPUT          ;SEND IT (MUST SAVE [A])
01AE A9                 XRA     C               ;CALCULATE CHKSUM
01AF 4F                 MOV     C,A             ;AND UPDATE IT
01B0 23                 INX     H               ;PT. TO NEXT BYTE
01B1 15                 DCR     D               ;DEC BYTE COUNT
01B2 C2AA01             JNZ     LOOP1           ;KEEP GOING UNTILL ALL 128 SENT
01B5 79                 MOV     A,C             ;NOW SEND CHECKSUM BYTE
01B6 CD4301             CALL    OUTPUT          ;SEND IT
                        ;
                        ;       WAIT FOR VERIFICATION: 'G'=GOOD, 'B'=BAD
                        ;
01B9 CD2301     VFYLP:  CALL    INPSTS          ;GET A CHAR
01BC FE42               CPI     'B'             ;A BAD READ?
01BE CAA301             JZ      TRYAGN          ;START AGAIN
01C1 FE47               CPI     'G'             ;A GOOD READ?
01C3 CA9701             JZ      READ            ;GO GET NEXT RECORD THEN
01C6 C3B901             JMP     VFYLP           ;CHAR MUST NOT HAVE BEEN READY
                        ;
01C9 11F401     EOF:    LXI     D,DONMSG
01CC CDD201     EXIT:   CALL    PRMSG           ;OUTPUT MESSAGE
01CF C30000             JMP     BOOT            ;ALL FINISHED
                        ;
01D2 0E09       PRMSG:  MVI     C,9             ;PRINT MESSAGE FUNCTION
01D4 C30500             JMP     BDOS
                        ;
                        ;
01D7 436F6D6D61CMDMSG:  DB      'COMMAND ERROR$'
01E5 46696C6520FNFMSG:  DB      'FILE NOT FOUND$'
01F4 0D0A55504CDONMSG:  DB      13,10,'UPLOAD COMPLETE$'
0206 55706C6F61WRKMSG:  DB      'UPLOADING...$'
                        ;
0213                    END
```

SOURCE LISTING: DOWNLOAD

```
                        ;
                        ;       DOWNLOAD
                        ;
                        ;  THIS PROGRAM WORKS WITH AN APPLE COMMUNICATIONS
                        ;  INTERFACE OR A CCS 7710A SERIAL INTERFACE IN SLOT
                        ;  TWO.
                        ;
                        ;  WRITTEN 5/80 BY NEIL KONZEN
                        ;    (C) 1980 MICROSOFT
                        ;
0000 =          BOOT    EQU     0000H           ;BOOT SYSTEM
0005 =          BDOS    EQU     0005H           ;BDOS ENTRY POINT
005C =          FCB     EQU     005CH           ;DEFAULT FCB
0080 =          BUFFER  EQU     0080H           ;DEFAULT BUFFER ADDR
                        ;
E0AE =          COMSTS  EQU     0E0AEH          ;COM OR CCS CARD STATUS LOC
E0AF =          COMDAT  EQU     0E0AFH          ;COM CARD DATA - SLOT 2
E000 =          APPKBD  EQU     0E000H          ;APPLE KEYBOARD
                        ;
                        ;
0100                    ORG     0100H           ;START AT TPA
                        ;
                        ;
                        ;
0100 3A8000     DWNLOD: LDA     BUFFER          ;MAKE SURE THERE'S A FILE NAME
0103 B7                 ORA     A               ;ANY CHARS IN CMD LINE?
0104 11C001             LXI     D,CMDMSG        ;POINT TO CMD ERROR MSG
0107 CA8D01             JZ      EXIT            ;QUIT
010A 0E13               MVI     C,19            ;DELETE FILE
010C 115C00             LXI     D,FCB
010F D5                 PUSH    D               ;SAVE PTR TO FCB
0110 CD0500             CALL    BDOS
0113 D1                 POP     D               ;REGET PTR TO FCB
0114 0E16               MVI     C,22            ;MAKE FILE
0116 CD0500             CALL    BDOS
                        ;
0119 3C                 INR     A               ;CHECK FOR ERROR
011A 11CE01             LXI     D,NDSMSG        ;GET READY TO PRINT 'NO DIR. SPACE'
011D CA8D01             JZ      EXIT
                        ;
                        ;       WAIT TILL UPLOAD SENDS AN 'R'
                        ;
0120 CDA001     RDYLP:  CALL    RDCOM           ;GET A CHAR FROM COM CARD
0123 FE52               CPI     'R'             ;'R' FOR 'READY'?
0125 C22001             JNZ     RDYLP
                        ;
0128 1E53               MVI     E,'S'           ;GET 'S' FOR 'SET'
012A CD9301             CALL    WRCOM
                        ;
012D CDA001     GETGEE: CALL    RDCOM           ;WAIT FOR 'G' FOR 'GO'
0130 FE47               CPI     'G'
0132 C22D01             JNZ     GETGEE
                        ;
0135 21F501             LXI     H,WRKMSG        ;POINT TO 'DOWNLOING' MSG
0138 7E         PRLP:   MOV     A,M             ;GET CHR
0139 B7                 ORA     A               ;SET CC'S
                        ;
013A CA4701             JZ      TRYAGN          ;GO DO DOWNLOAD
013D E5                 PUSH    H
013E 5F                 MOV     E,A             ;CHAR TO LEI FOR CONOUT
013F CD8B01             CALL    CONOUT
0142 E1                 POP     H
0143 23                 INX     H
0144 C33801             JMP     PRLP
```

```
                ;
                ;
0147 218000     TRYAGN: LXI     H,BUFFER        ;POINT TO 128 BYTE BUFFER
014A 0E00               MVI     C,0             ;CLEAR CHECKSUM
014C 0E81               MVI     C,81H           ;READ 128 BYTES + 1 CHKSUM
                ;
014E CDA001     LOOP1:  CALL    RDCOM           ;READ A BYTE
0151 77                 MOV     M,A             ;STORE IT
0152 A9                 XRA     C               ;CALC CHKSUM
0153 4F                 MOV     C,A             ;UPDATE IT
0154 23                 INX     H               ;NEXT BYTE
0155 15                 DCR     D               ;DECREMENT BYTE COUNT
0156 C24E01             JNZ     LOOP1           ;NOT DONE - CONTINUE
0159 B7                 ORA     A               ;WAS CHKSUM ZERO?
015A CA6A01             JZ      GOODRD          ;THINGS ARE OK
                ;
015D 1E42       BADRD:  MVI     E,'B'           ;'B' FOR 'BAD'
015F CDBB01             CALL    CONOUT
0162 1E42               MVI     E,'B'           ;SEND 'B' TO UPLOAD
0164 CD9301             CALL    WRCOM
0167 C34701             JMP     TRYAGN
                ;
016A 1E2E       GOODRD: MVI     E,'.'           ;PRINT A PERIOD
016C CDBB01             CALL    CONOUT
016F 115C00             LXI     D,FCB           ;POINT TO FCB
0172 0E15               MVI     C,21            ;WRITE SEQ.
0174 CD0500             CALL    BDOS
0177 1E47               MVI     E,'G'           ;SEND UPOAD A 'G' FOR 'GOOD'
0179 CD9301             CALL    WRCOM
017C C34701             JMP     TRYAGN
                ;
                ;
017F 3210E0     DONE:   STA     APPKBD+10H      ;CLR KBD STROBE
0182 115C00             LXI     D,FCB
0185 0E10               MVI     C,16            ;CLOSE THE FILE
0187 CD0500             CALL    BDOS
018A 11E101             LXI     D,DONMSG        ;ALL DONE MSG
018D CDB601     EXIT:   CALL    PRMSG           ;PRINT THE MESSAGE
0190 C30000             JMP     BOOT
                ;
0193 3AAEE0     WRCOM:  LDA     COMSTS          ;COM CARD STATUS
0196 E602               ANI     2               ;CHECK BIT 2
0198 CA9301             JZ      WRCOM
019B 7B                 MOV     A,E             ;GET CHAR TO SEND
019C 32AFE0             STA     COMDAT          ;STORE HERE
019F C9                 RET
                ;
01A0 3AAEE0     RDCOM:  LDA     COMSTS          ;COM CARD STATUS
01A3 1F                 RAR                     ;STS BIT TO CARRY
01A4 DAB201             JC      READIT
01A7 3A00E0             LDA     APPKBD          ;SEE IF CTRL-C TYPED
01AA FE83               CPI     083H            ;??
01AC CA7F01             JZ      DONE

01AF C3A001             JMP     RDCOM           ;NO, WAIT FOR CHAR
                ;
01B2 3AAFE0     READIT: LDA     COMDAT          ;GET INCOMING CHARACTER
01B5 C9                 RET
                ;
01B6 0E09       PRMSG:  MVI     C,9             ;PRINT MESSAGE
01B8 C30500             JMP     BDOS
                ;
01BB 0E02       CONOUT: MVI     C,2             ;CONSOLE OUTPUT
01BD C30500             JMP     BDOS
                ;
01C0 436F6D6D61CMDMSG:  DB      'Command Error$'
01CE 4E6F206469NDSMSG:  DB      'No directory space$'
01E1 0D0A446F77DONMSG:  DB      13,10,'Download complete$'
01F5 446F776E6CWRKMSG:  DB      'Downloading',0
                ;
0201                    END
```