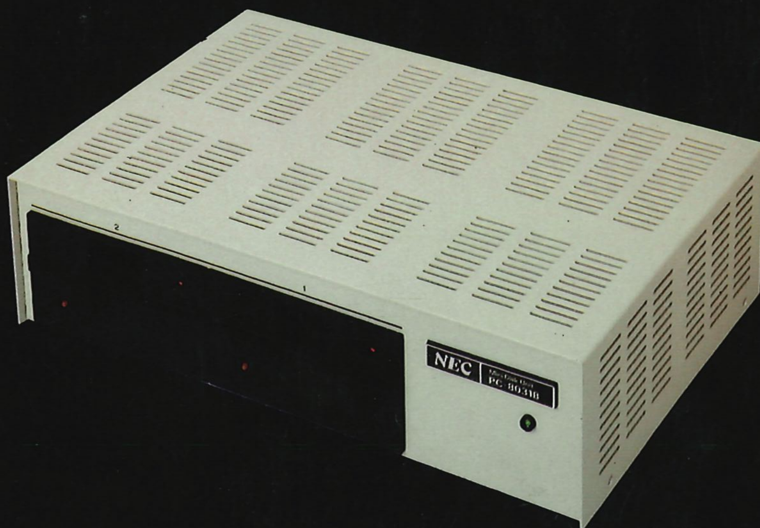
PC-8031B /32B Mini Disk Unit Reference Manual



NEC

PC-8105B
PTS-073

PC-8031B/32B Mini Disk Unit Reference Manual

NEC

IMPORTANT NOTICE

(1) All rights reserved. This manual is protected by copyright. No part of this manual may be reproduced in any form whatsoever without the written permission of the copyright owner.
(2) The policy of NEC being that of continuous product improvement, the contents of this manual are subject to change, from time to time, without notice.
(3) All efforts have been made to ensure that the contents of this manual are correct; however, should any errors be detected, NEC would greatly appreciate being informed.
(4) NEC can assume no responsibility for errors in this manual or their consequences.

© Copyright by Nippon Electric Co., Ltd.

CAUTION

Precautions for Carriage

When carrying this unit, make sure of the following two requirements.

1. Use of the prescribed packing materials
   Be sure to use the same packing materials as used at the time of shipment.

2. Resetting of the drive heads to track 0
   For protection's sake, all the drive heads should have been reset to each track 0 before the packing.
   Procedure for resetting the drive heads to track 0
   ① Take out all the floppy disks from each drive.
   ② Depress the reset switch of PC-8001.
   ③ Confirm that the drive No. 1 LED (red lamp) lights up.
   ④ Disconnect PC-8001 and PC-8031 (including PC-8032) from the power supply.
   Thus, resetting of all the drive heads (including those of PC-8032, if this unit is mounted) to each track 0 can be accomplished.

TABLE OF CONTENTS

I. Foreword 1

II. Hardware Section 3

1. Location and Names of Important Parts . . 6

1.2 PC-8032 6

2. Cable Connections 7

2.1 Using the NEC PC-8033 FDC I/O Port . 7

2.2 Connection Using PC-8011 8

2.3 Connecting the PC-8031 and PC-8032 . 8

3. Putting Disks into Drive 9

4. Motor Control 11

5. Specifications for the PC-8031 and PC-8032 . 12

5.1 Storage Capacity 12

5.2 Data Transmission Speed between the PC-8001 and PC-8031 12

5.3 Environmental Conditions 13

5.4 Power Supply 13

5.5 Dimensions 13

6. Cautions 14

• Concerning Power Supply 14

• Environmental Conditions 14

• Miscellaneous 15

• In Case of Malfunction 15

III. Floppy Disk 17

1. Names of Disk Parts 19

2. Use, Storage and Shipping Conditions . . . 20

2.1 Condition during Normal Use 20

2.2 Storage Conditions 20

2.3 Shipping Conditions 20

3. Labels 21

3.1 Write Protect Label 21

3.2 Precautions Concerning Label Entries . 21

4. General Precautions 23

4.1 During Operation 23

4.2 General Precautions 23

5. Unusable Disks 25

6. Disk Life 26

IV. Software Section 27

Chapter 1 — An Outline of the Software Section . 29

Chapter 2 — Booting DISK BASIC 30

Chapter 3 — Loading and Saving Programs . . . 33

3.1 Declaration of Disk Use — MOUNT Command 33

3.2 Saving Programs — SAVE Command . 34

3.3 Listing Filenames — FILES Command . 35

3.4 Loading Programs — LOAD Command . 36

3.5 Declaring End of Disk Use — REMOVE Command 37

3.6 ASCII SAVE and MERGE 39

3.7 Deleting Filenames — KILL Command . 39

3.8 File Attribute and SET Statement . . 40

3.9 Changing Filenames — NAME Command 41

3.10 RUN Command and R Option . . . 42

3.11 Chaining Programs 42

Chapter 4 — Sequential Files 45

4.1 Meaning of "Sequential File" 45

4.2 OPEN Statement 45

4.3 Reading and Writing Sequential Files . 47

4.4 CLOSE Statement 56

4.5 How the Interpreter Processes OPEN and CLOSE Statements 60

4.6 Appending Records to Sequential Files . 61

Chapter 5 — Random Access Files 63

5.1 Records 63

5.2 Fields 65

5.3 Flow of Random File Processing . . 65

5.4 Opening Files 66

5.5 Defining Fields 66

5.6 LSET and RSET Statements 67

5.7 PUT Statement 68

5.8 GET Statement 69

5.9 Using Random Access Files — Example 1 70

5.10 Handling Numerical Data 73

5.11 Using Random Access Files — Example 2 75

Chapter 6 — Functions 78

6.1 DSKF 78

6.2 INPUT$ 79

6.3 LOC 81

6.4 FPOS 81

6.5 ATTR$ 81

Chapter 7 — DSKO$ and DSKI$ 83

7.1 DSKI$ 83

7.2 DSKO$ 84

7.3 An Example of the DSKI$ Statement . 85

Chapter 8 — Structure of Disk Files 88

8.1 Storing Data on Disks 88

8.2 Extents, FAT, and Directory 89

8.3 Track Allocation 90

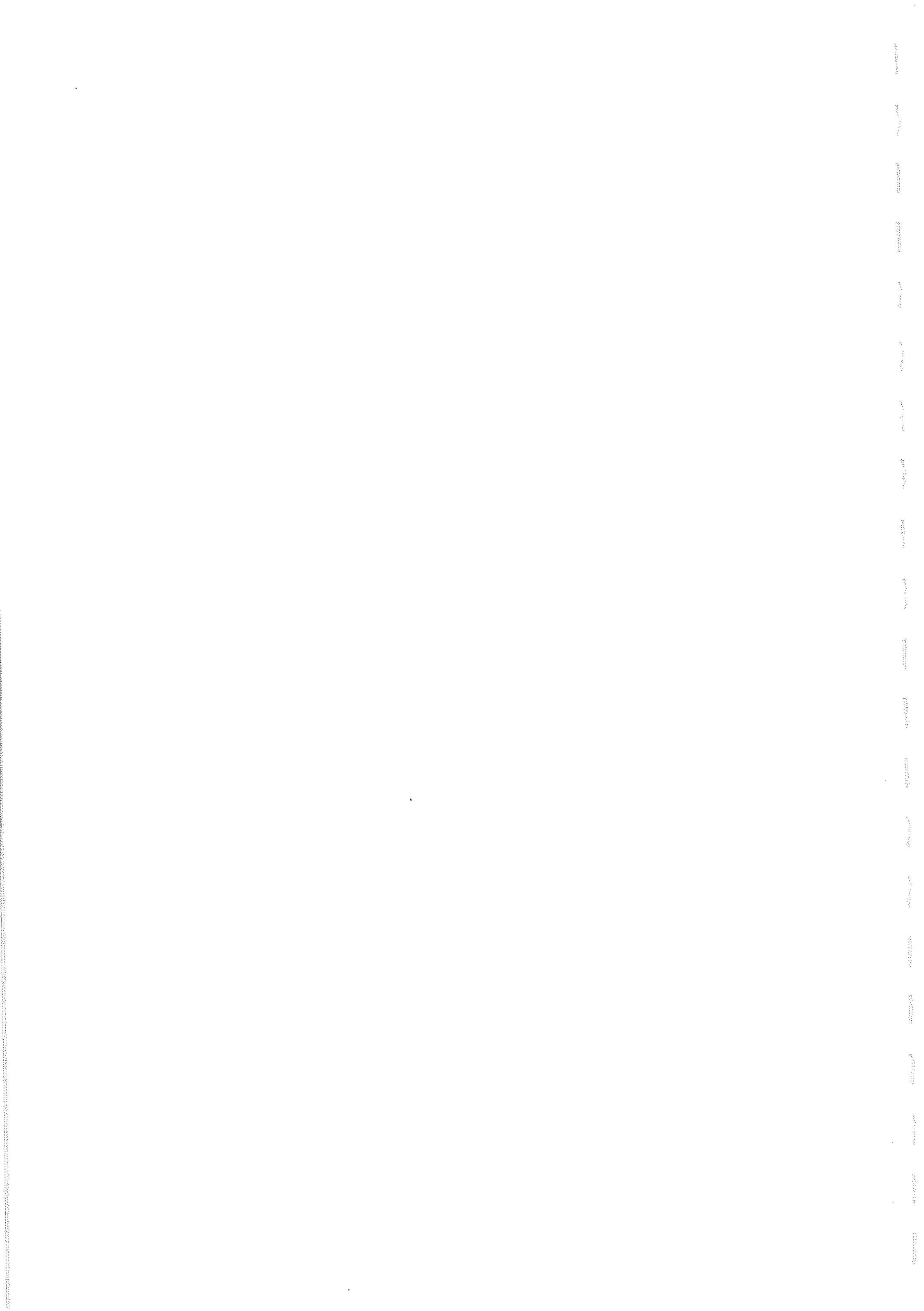
8.4 Directory 91

8.5 FAT 92

8.6 ID 93

Chapter 9 — Formatting Disks 94

Chapter 10 — Preparing Backup Disks 95



I. FOREWORD

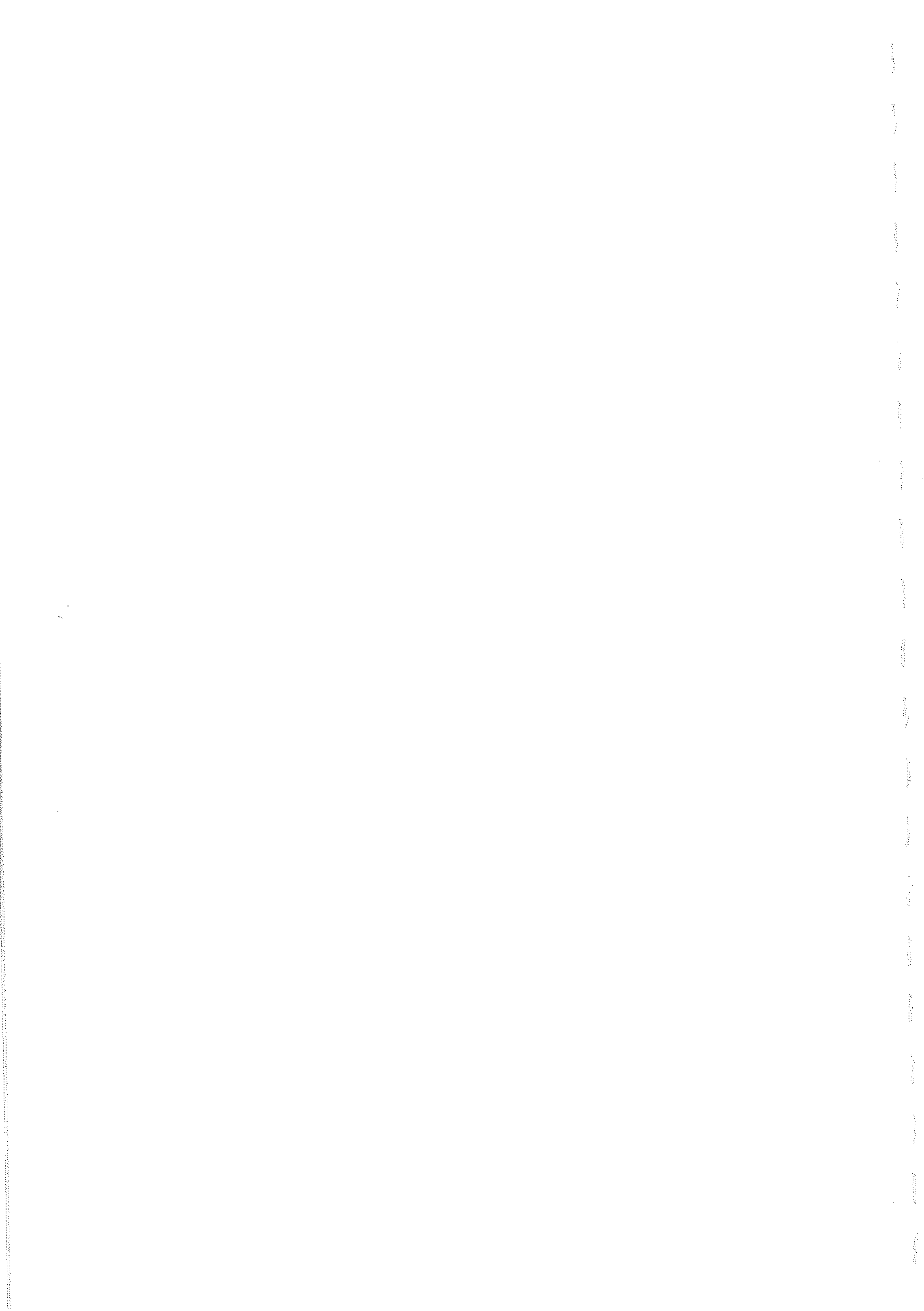
The PC-8031 dual mini-floppy disk unit is an external storage unit developed for use with the PC-8001 personal computer.

The PC-8031 includes: two mini-floppy drive units, the associated control circuitry, and power supply. The PC-8031 is an intelligent drive, lessening the burden of the PC-8001's CPU. Data to be stored in the PC-8031 is merely passed to it through an I/O port, and logic within the PC-8031 does the rest.

The total storage available on one disk is 143.36K bytes, however, since a portion of this must be used for control tables, the user area is 139.264K bytes. Thus, the total storage available with one dual-drive unit is 278.528K bytes.

The PC-8032 expansion drive unit contains two mini-floppy drives and power supply, and is controlled by the controller in the PC-8031. The storage capacity of the PC-8032 is the same as the PC-8031.

Note: We recommend that disks distributed by NEC be used in the PC-8031 and PC-8032 drive units.



II. HARDWARE SECTION

Contents:

1. Locations and Names of Important Parts

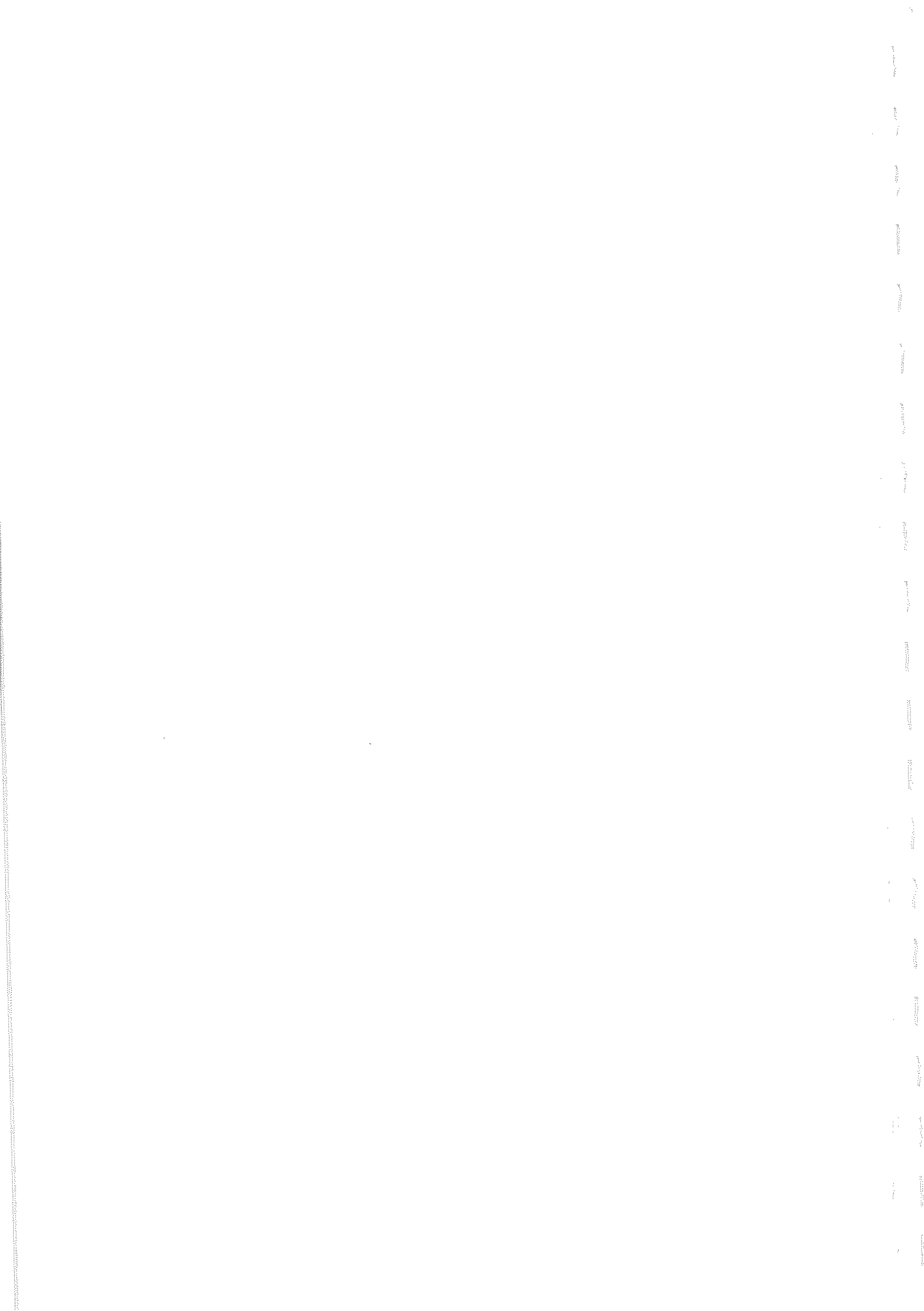
2. Cable Connections

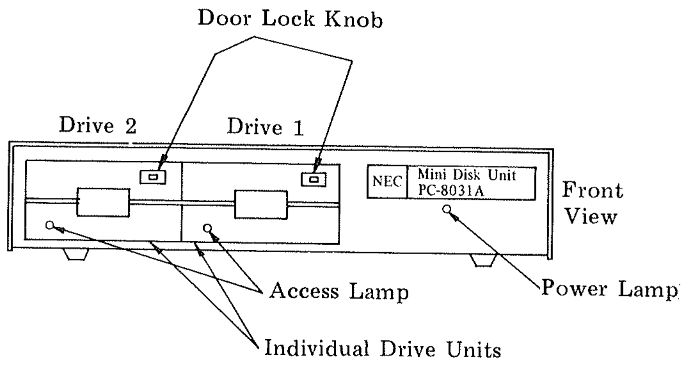
3. Putting Disks into Drive

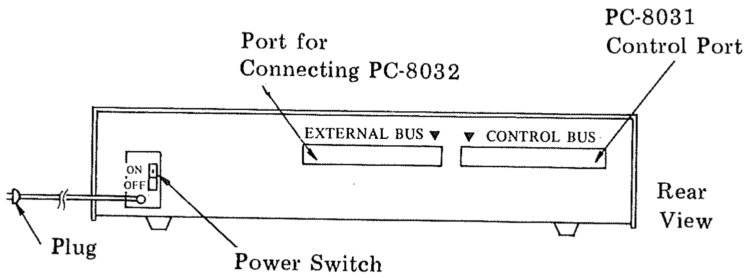
4. Motor Control

5. Specifications for the PC-8031 and PC-8032

6. Cautions




Door Lock Knob
Drive 2
Drive 1
NEC
Mini Disk Unit
PC-8031A
Front View
Access Lamp
Power Lamp
Individual Drive Units


Port for Connecting PC-8032
PC-8031 Control Port
EXTERNAL BUS ▼
▼ CONTROL BUS
ON
OFF
Rear View
Plug
Power Switch

1. Locations and Names of Important Parts

1. Power Lamp

This lamp comes on when the power supply is turned on.

2. Access Lamp

When all drives (except drive 4) are in stand-by, all access lamps are lit to about 20% of full brightness. But the access lamp of drive 4 is lit to about 80% of full brightness. When a drive is accessed, its access lamp goes to full brightness, and all other access lamps go out.

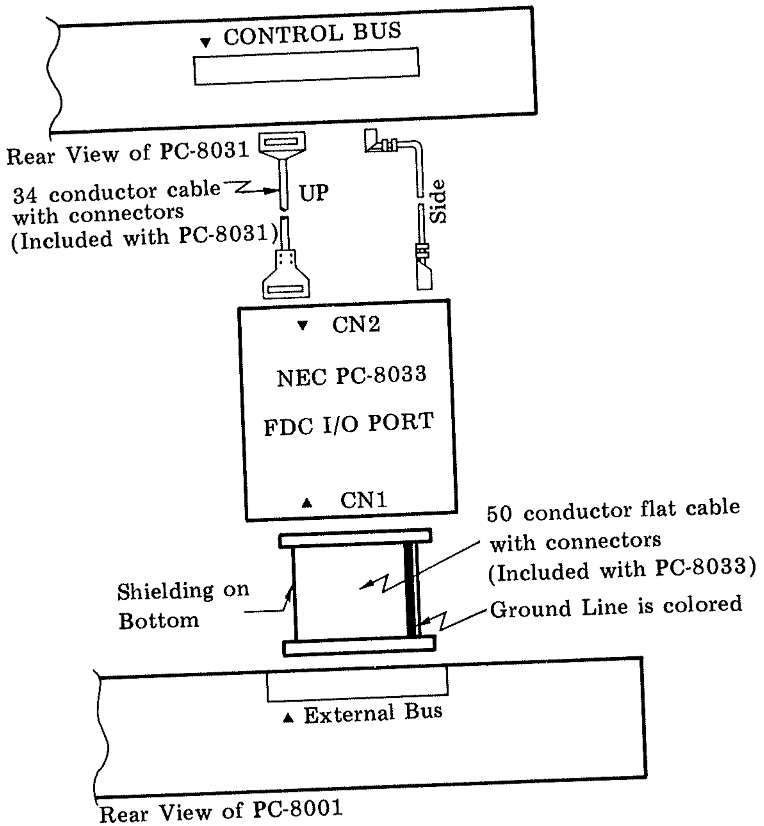
1.2 PC-8032

Specifications for the NEC PC-8032 are the same as those for the PC-8031, except that the NEC PC-8032 has provisions for only one connector.

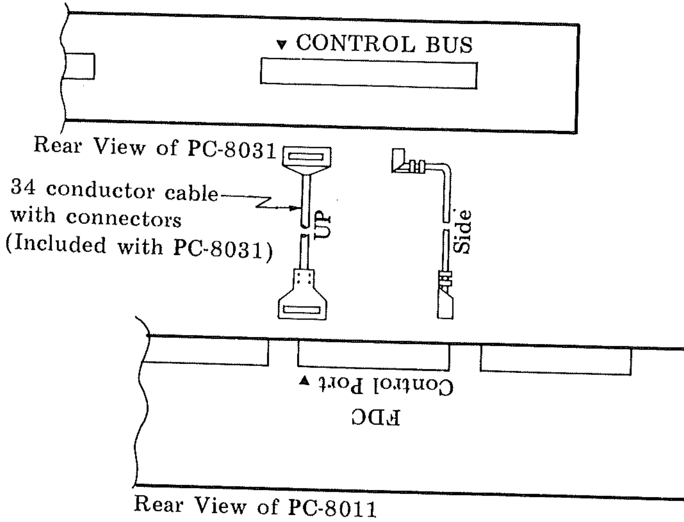
2. Cable Connections

2.1 Using the NEC PC-8033 FDC I/O Port

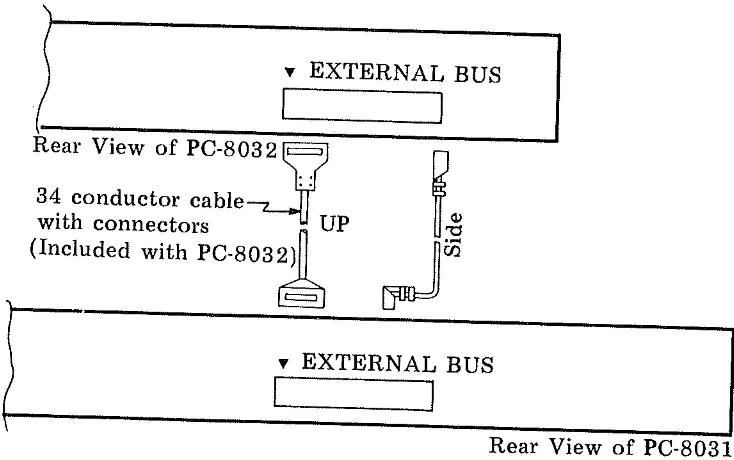
The NEC PC-8033 Floppy Disk Controller I/O Port may be used in lieu of the PC-8011 expansion unit. With power disconnected from all units, first connect the PC-8001 to the PC-8033, then connect the PC-8033 to the PC-8031.



2.2 Connection Using PC-8011



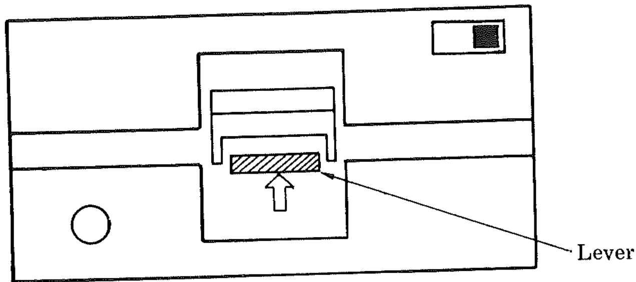
2.3 Connecting the PC-8031 and PC-8032



3. Putting Disks into Drive

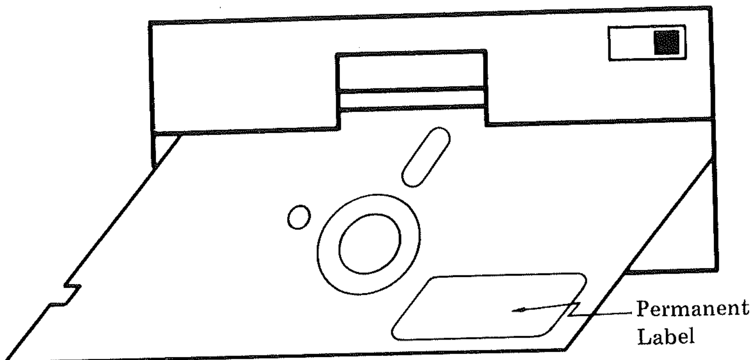
Step 1: Open drive door

Confirm that the door lock knob is set to the lock-release position. Then, gently lift up the door lever in the arrow direction to open the door.



Step 2: Insert disk into drive

Orient the disk properly and gently slide it as far as it will go into the drive. Do not bend disk.



Step 3: Close drive door

The door of the drive has two metal hooks which catch on the disk if it is not completely inside the drive, preventing the door from closing. Forcing the door closed when the disk is not properly inserted, will damage the disk.

4. Motor Control

Basically, a floppy disk drive is similar to a record player. Instead of a grooved disk, there is a disk coated with a ferromagnetic compound, and instead of a needle, there is a recording head similar to those used in cassette recorders. But still, the principle is the same. The disk must be spinning to carry the information to the head, and the motor which does the spinnig must be carefully controlled.

The PC-8031, being an intelligent device, controls its own motor. When a disk is accessed, it automatically turns the motor on, and if no disk access takes place for a period of three minutes, it automatically turns the motor off. This feature minimizes wear and tear on both the drive and the disk.

If the disk is accessed when the motor is off, there will be a wait of approximately two seconds before the disk can be accessed.

5. Specifications for the PC-8031 and PC-8032

Note: All specifications given in this section are the same for the PC-8031 and PC-8032.

5.1 Storage Capacity

Per Unit	286.72K bytes Note: Some of this is used for file tables. User area is 278.528K bytes.
Per Drive	143.36K bytes (User area = 139.264K bytes)
Tracks per Disk	35 tracks (34 user tracks)
Sectors per Track	16 sectors
Data per Sector	256 bytes

5.2 Data Transmission Speed between the PC-8001 and PC-8031

	Minimum*	Maximum*
1 track read time	0.3 seconds	1.62 seconds
1 track write time	0.5 seconds	1.82 seconds

*Minimum times are for the case where the head is directly over the track to be read or written. Maximum times are for case where the head is the maximum distance away, for example, when the head is at track 1 and track 35 is to be read.

5.3 Environmental Conditions

	Ambient Temperature	Humidity	Max. Wet Bulb Temp.
Operation	10°C ~ 35°C 50°F ~ 95°F	20%~ 80%	25°C (77°F) in all cases (no condensation)
Shipping	−40°C ~ 62°C −40°C ~ 144°F	1%~ 95%	
Storage	−22°C ~ 47°C −8°F ~ 117°F	1%~ 95%	

5.4 Power Supply

Supply Voltage	220/240 V ~ ±10%, 50/60 Hz
Power Consumption	40 W

5.5 Dimensions

Width	440.2 mm — $17\frac{5}{16}$ in.
Height	128.0 mm — $5\frac{1}{32}$ in.
Depth	291.5 mm — $11\frac{1}{2}$ in.
Weight	PC-8031 8.3 kg $18\frac{2}{7}$ lbs
	PC-8032 7.8 kg $17\frac{2}{5}$ lbs

6. Cautions

- Concerning Power Supply
  (1) After turning the power supply off, wait at least five seconds before turning it back on. In the same manner, if the power supply is unplugged with the power switch on, wait five seconds before plugging it back in.
  (2) Always operate the power supply on 60 cycle 120 VAC.
  (3) Always grip the plug properly when plugging and unplugging.

- Environmental Conditions
  (1) To prevent over-heating, the case of the drive has been provided with vents for air circulation. Do not block the vents or operate the drive where the air cannot properly circulate. Also, the drive should not be stored or operated under extreme temperatures or temperature changes.
  (2) The drive should not be stored or operated in direct sunlight or near sources of heat.
  (3) The drive should not be stored or operated in dusty or moist places.
  (4) The drive contains delicate electronic and mechanical components and should not be subjected to strong shocks or vibrations.
  (5) Never operate the drive when foreign materials, especially liquids and metallic objects are inside the case.
  (6) Vaports of solvents and other strong chemicals may cause disks to deteriorate.
  (7) Storing or operating the drive with the cover removed may result in damage to the device or electrical shock.
  (8) Do not place heavy objects on top of the drive.

(9) Operating the drive close to a radio or TV may cause interference with the reception of the radio or TV. Also, operation of the drive may be affected if placed in a strong electrical field.

- Miscellaneous
  (1) The cover of the drive may be cleaned by wiping with a soft cloth, dampened with a little water or liquid detergent. Never use solvents, such as benzene, or chemicals, such as pesticides.

- In Case of Malfunction
  (1) If a malfunction is detected, the disk should be disconnected, and the matter discussed with your NEC dealer.

III. FLOPPY DISK

Contents:

1. Names of Disk Parts

2. Use, Storage and Shipping Conditions

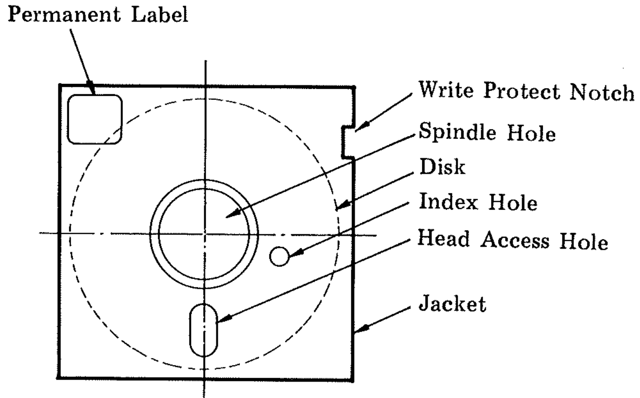
3. Labels

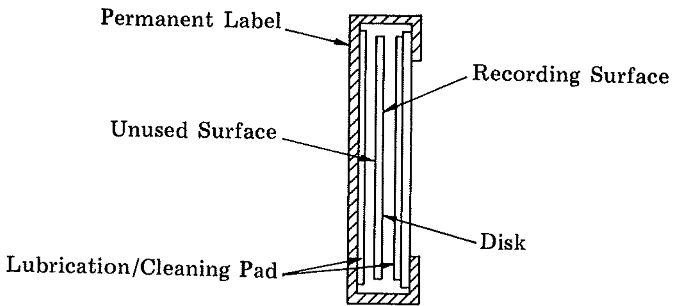
4. General Precautions

5. Unusable Disks

6. Disk Life

1. Names of Disk Parts





2. Use, Storage and Shipping Conditions

2.1 Condition during Normal Use

	Tolerance Range
Temperature	10 °C ~ 50 °C/50 °F ~ 122 °F
Humidity	8% ~ 80% (Max. wet bulb temp. = 29 °C/84 °F)
Electrical Fields	Less than 4,000 A/m (50 oersteds)
Acclimatization	After being subjected to other than normal operating conditions, disks should be acclimatized before use for at least five minutes. (Temp. acclimatization = 20 °C/68 °F/hr)

2.2 Storage Conditions

	Tolerance Range
Temperature	10 °C ~ 50 °C/50 °F ~ 122 °F
Humidity	8% ~ 80% (Max. wet bulb temp. = 29 °C/84 °F)
Electrical Fields	Less than 4,000 A/m (50 oersteds)

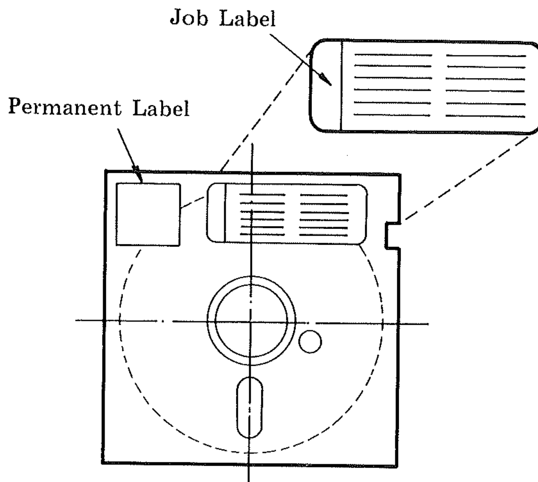
2.3 Shipping Conditions

	Tolerance Range
Temperature	—40° C ~ 52° C/—40° F ~ 126° F
Humidity	8% ~ 80%
Temperature Fluctuation	20° C/68° F/hr.

3. Labels

3.1 Write Protect Label

This label is placed on the disk's jacket at the time of shipment, and may be readily removed and re-affixed to the jacket. The write protect label mechanically protects a disk from being erased by covering up its write protect notch. Do not stick this label anywhere else on the disk's jacket.



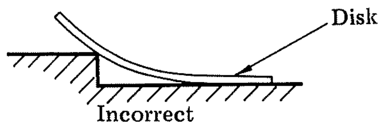
3.2 Precautions Concerning Label Entries

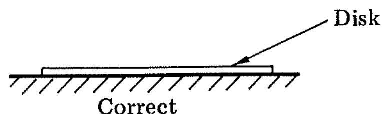
(1) Do not write directly on the disk's jacket. Always use the prescribed labels and fill the label in before sticking it on the jacket.

(2) Fill in the label with a felt tip marker. Do not write with a pencil or ball-point pen, and do not use an eraser on the label. Small particles left by these objects may get inside the jacket and damage the disk. Pressure from a pencil or ball-point pen may damage the disk's surface.

(3) Labels may be filled in while on the disk if a felt tip marker is used.

(4) When writing on a label that is on a disk, make sure that the disk is lying on a flat surface.





(5) When a job label becomes full, peel it off, carefully removing all glue from the jacket, and apply a new label. Clearances within the drive are minimal, and sticking labels on top of other labels may cause damage to the device.

4. General Precautions

4.1 During Operation

(1) Before Using

(a) Do not use disks that are visibly scarred or warped or have any other visible damage.

(b) Before using a disk that has been stored under conditions other than those specified for normal operation, remove the disk from its protective envelope and acclimatize for at least five minutes. Acclimatization time for temperature differences is 20°C/68°F/hr.

(2) Inserting Disk

Slide the disk straight and smoothly into its slot until you feel it meet with the backstop. Careless insertion may damage the edge of the disk, making it unusable.

While the disk is in use, place the protective envelope back into the storage box and leave the storage box in a clean location to prevent dust contamination.

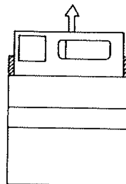
(3) After Using

After using a disk, place it back into its protective envelope and return them to the storage box. The box may be positioned either vertically or horizontally.

4.2 General Precautions

(1) Placing heavy objects on top of disks may damage them.

(2) When removing a disk from its protective cover, grasp it by the upper edge of its jacket.



(3) Do not place in direct sunlight or heat (over 65°C/149°F).
(4) Do not place clips or clamps on disks.
(5) Never touch a disk's recording surface. Oil picked up from the skin will cause dust and other particles to stick to the disk, resulting in damage to the disk and/or the head.
(6) Do not attempt to clean disks.
(7) Placing disks close to magnets or in strong electrical fields may destroy the data recorded on them.
(8) Do not bend disks.

5. Unusable Disks

If any of the following occur, the disk concerned may no longer be used.

(1) Bending, tearing, warping, or any other physical damage to the disk.
(2) Soiling of the disk or its jacket by liquids, especially sticky substances, or abrasive materials such as metal powders.
(3) Direct damage or soiling of recording surface.

Caution: Attempted use of a damaged disk may result in permanent damage to the drive's head. If the head of the drive is damaged or soiled without the user being aware of it, other disks placed into the drive may be damaged also.

6. Disk Life

Normal life expectancy of a floppy disk varies between 3 million and 10 million passes*, however, usable life depends upon how the disk is handled, what kind of drive it is used in, whether one side or both sides are used, and cleanliness of the environment in which the disk is used.

Besides replacing those disks for which errors have occurred, all disks in use should be controlled according to the following procedure.

(1) Before using a new disk, assign it a control number, and enter the number on the disk's permanent label.

(2) Make a control record in which the data that disks are used and their length of use are recorded.

(3) Periodically check the control record, and replace disks which have been used for an excessively long time and disks for which errors occurred repeatedly.

*One pass is defined as a continuous read or write operation of the same track for one full turn of the disk.

IV. SOFTWARE SECTION

Contents:

CHAPTER 1 — An Outline of the Software Section

CHAPTER 2 — Booting Disk Basic

CHAPTER 3 — Loading and Saving Programs

CHAPTER 4 — Sequential Files

CHAPTER 5 — Random Access Files

CHAPTER 6 — Functions

CHAPTER 7 — DSKO$ and DSKI$

CHAPTER 8 — Structure of Disk Files

CHAPTER 9 — Disk Formatting

CHAPTER 10 — Preparing Backup Disks

CHAPTER 1: An Outline of the Software Section

The following is a brief outline of the material presented in this section of the manual.

Chapter 2 ～ 4:	These chapters explain the fundamentals of DISK BASIC, and should be read and understood by all users.
Chapter 5:	This chapter covers random files. Random files are especially attractive for users writing programs which must update large amounts of data.
Chapter 6:	Chapter 6 explains how to use the major functions found in DISK BASIC. The more fundamental functions are explained in Chapter 4, and a detailed study of Chapter 6 is necessary only for those users who must use the full range of N-BASIC's disk capabilities.
Chapter 7:	Explanation of DSKI$ function and DSKO$ statement.
Chapter 8:	This chapter describes the structure of files used by the PC-8000 systems DISK BASIC.
Chapter 9:	Chapter 9 explains how to format disks.
Chapter 10:	The final chapter tells how to prepare back-up disks.

CHAPTER 2: Booting DISK BASIC

The following steps are required to boot **DISK BASIC** on the PC-8001.

1. Interconnect all units in the system according to the explanation in the hardware section of this manual.
2. Turn on the PC-8031's power supply.
3. Insert the system disk into drive one.
4. Turn on the PC-8001 and press the reset button.
5. At this point, the motor will start. Confirmation that the motor is running may be made by looking through the door of drive 2.
6. After approximately 9 seconds, drive one's LED will come on and a clicking sound can be heard. This indicates that the disk is being accessed.

WARNING

Never open a drive's door while its LED is lit. Attempts to remove a disk while the drive is in operation will destroy the disk.

****

7. Approximately 9 seconds later, the drive's LED will go out, and the following message will appear on the screen:

   DISK VERSION
   How many files **(0 — 15)?**

   The meaning of this message is explained in Chapter 4. At this point, enter "3" followed by a carriage return.
8. The screen should appear as below, indicating that **DISK BASIC** is ready to be used.

DISK VERSION
How many files (0 – 15)? 3
NEC PC-8001 BASIC Ver 1.1
Copyright 1979(c) by Microsoft
OK

(Above messages may be different reflecting the version of your system disk.)

9. System disk come with the PC-8031. To be on the safe side, we recommend that you copy the system onto the blank disk for use as a back-up. To do this, leave the system disk in drive 1, and place the blank disk in drive 2. Next, enter the following command:

   mount 1 [Cr]

   After this command is entered, drive one's LED will come on momentarily and a clicking sound will be heard. After the "OK" prompt appears on the screen, enter the next command:

   run "backup" [Cr]

   Again, drive one's LED will come on, a click will be heard, and the following message will appear on the screen:

   Back up a disk
   Mount master disk on drive 1, then hit return

   Press the carriage return key at this point, and the next message will appear:

   Mount new disk on drive 2, then hit return

Again, press the carriage return, and the following message will be printed:

Format disk 2 (y/n)?

In reply to this prompt, enter a lower case "y". Copying will then begin. The LED's of drive one and drive two will go on and off alternately, accompanied by clicking sounds. "Copying track 'n' " messages will then appear, as shown in Figure 1 (the numbers 'n' may be different). When the message "Complete" appears, the copy operation is finished. Store one copy of the system disk in a safe place. For a more detailed description of the system disk, see Chapter 10.

```
   back up a disk

Mount master disk on drive 1,then hit return
Mount new disk on drive 2,then hit return
Format disk 2 (y/n)? y
Formatting disk 2
Copying track 0
Copying track 1
Copying track 2
Copying track 17
Copying track 18
Complete
Ok
```

Figure 1 Sample run of Backup Program

CHAPTER 3: Loading and Saving Programs

This chapter explains how to read and write program files using the floppy disk drive. Programs saved to disk are not only much easier to handle than programs on cassette tape, but also much more reliable.

3.1 Declaration of Disk Use — MOUNT Command

Format: MOUNT [<drive number> [,<drive number>. . .]]

Purpose: To prepare the disk(s) in specified drive(s) available for use.

Remarks: The MOUNT command must always be executed before any other disk related function is used. If <drive number> is omitted, all disks currently in drives will be mounted. When a disk is mounted, N-BASIC reads the disk's file allocation table into memory and checks the table for errors. If no errors are found, the disk is mounted. If there are errors in the table, one or both of the back-up tables are read, and the message, "X copies of allocation bad on drive Y". X may be either 1 or 2, and Y is the drive number. A new copy should be made of any disk for which this message is issued. If all allocation tables are bad, it is an unrecoverable error, and the message, "bad allocation table" is printed. A disk for which such an error has occurred cannot be mounted. During operation, the file allocation table is updated from time to time. However, unless the read-after-write attribute has been specified for that drive, no error check is performed. See explanation for SET statement (Section 3.8).

Example: MOUNT 1

Note: Before a disk is taken out of a drive, it must be REMOVEd (see Section 3.6).

Note: Do not attempt to MOUNT an empty disk drive. This will cause the system to lock up and require manual resetting on the PC-8001 to release the drive.

3.2 Saving Programs — SAVE Command

The SAVE command is used for saving programs to disk. To avoid destroying files already on disk when using this command, you should be familiar with the contents of Chapters 2 and 3. The format of the SAVE command is:

SAVE <filename>

where <filename> is of the form:

"<drive number>: <filename>. <extended identifier>"

NOTE:

1. <drive number> is an integer from 1 to 4, when the PC-8032 is attached, or from 1 to 2 when only the PC-8031 is attached.
2. If an illegal drive number is specified, the "Bad drive number" message will be displayed.
3. When using drive 1, "1:" may be omitted from the filename.
4. <filename> may be up to six characters only.
5. <extended identifier>indicates the file type and may be up to three characters in length. When dealing exclusively with BASIC files, the <extended identifier> is not necessary and may be omitted along with the preceeding period.

Examples:

SAVE "prog 1"	Saves the current program to the disk in drive 1 under the name "prog 1".
SAVE "prog 1.bas"	Saves the current program to the disk in drive 1 under the name "prog 1.bas".
SAVE "2:prog 1"	Saves the current program to the disk in drive 2 under the name "prog 1".

Files previously stored under the same name will be replaced by the new file when the SAVE command is executed.

WARNING

In the PC-8001's DISK BASIC, the drive number is part of the filename, and must always be specified, except when using drive 1. Failure to follow this notation can result in destruction of files. For example, the KILL command deletes a specified file for a specified drive. If drives 1 and 2 both contain disks with files of the same name, say "fred" for example, the command KILL "fred" will delete the file on the disk in drive 1 even though the operator intended it for the other disk.

****

3.3 Listing Filenames — FILES Command

The FILES command will cause the names of all files on a disk to be displayed. The format of the FILES command is:

FILES <drive number>

If <drive number> is omitted, drive 1 is assumed. If there are no files on the specified disk, only a blank line is printed. Examples of the FILE and SAVE commands are given in Figure 2.

```
save "prog 1"
OK
files
prog 1.    1
OK
```

Figure 2

The LFILES command outputs a list of all filenames to the printer. Its format is:

lfiles <drive number>

In Figure 2 there is a period following "prog 1". This period indicates that the file was saved in non-ASCII format. As will be explained in Section 3.6, program files may be saved in either "condensed binary" form (non-ASCII) or their "listed" form (ASCII). In ASCII files, (all data files are ASCII), there is a space between the sixth and seventh character of the filename, instead of the period which appears in binary files. This period or space only appears in the filename when it is printed out by the FILES command. When referencing a file, the filename may be entered in exactly the same form as when it was assigned to the file. For example, the filename "abcdefg" will be output by the FILES command as "abcdef.g" if saved in the binary form, or as "abcdef g" if saved in the ASCII form, but will be referenced as "abcdefg".

3.4 Loading Programs — LOAD Command

The LOAD command reads program files from disk into memory. The format for the LOAD command is:

LOAD <filename>

The format of <filename> is the same as explained in Section 3.2.

Examples:

load "prog1"
load "prog1.bas"
load "2: prog1"

As shown in the last example, when a drive other than drive 1 is being used, the drive number should be specified in the filename.

There are two kinds of files, program files and data files. When writing files to disk, these commands are the SAVE command and PRINT # command, respectively. Do not load a data file, this could cause the system to enter an endless loop, requiring it to be reset. This problem can be avoided by using the extended identifier to indicate whether a file is a program file or data file. For example, extended identifiers could be assigned in the following manner:

.bas	For program files
.asc	For programs in ASCII format
.dat	For data files

However, assigning an extended identifier each time a file is created, is a bit of a chore, and so we'll leave the use of this procedure up to the user.

3.5 Declaring End of Disk Use — REMOVE Command

The REMOVE command informs the system that you have finished using a disk. It's format is:

REMOVE <drive number>

Examples:

remove 1
remove 1,2
remove

If <drive number> is omitted as in the last example, all disks are assumed.

We would like to emphasize that, although drive numbers are specified as arguments for the MOUNT and REMOVE commands, these operations are actually carried out on the disk in the specified drive. For example, after MOUNTing and using a disk in drive 1, if you want to use a different disk in the same drive, you must first execute the REMOVE command for the disk currently in the drive, and then execute the MOUNT command for the next disk you place in the drive. This procedure is necessary to keep the FAT table updated. When the MOUNT command is executed, the FAT table of the disk currently in the drive is copied into RAM, and all changes that are made to data on the disk while it is in the drive are recorded in this copy of the FAT table. The copy of the FAT table on disk is not updated until the REMOVE command is issued. Thus, if the disk is removed from the drive, or the system is reset without executing the REMOVE command, the interpreter will not be able to locate any of the files you have added during that session, and in the worst case, all data on the disk will be lost. Such a disk is not physically damaged, and may be used after reformatting; however, there is no way to recover the lost data.

Notes:

1. Even if you take a disk out of a drive without executing the REMOVE command, there is a possibility that the data on the disk may be saved. Immediately return the disk to the drive, and after making sure that it is properly positioned, execute

the REMOVE command. If the system replies with the "OK" prompt, you have succeeded. If the message "I/O fault" appears, you have failed.

2. When taking a disk which has not been MOUNTed out of a drive, there is no need to execute the REMOVE command.

3.6 ASCII SAVE and MERGE

The SAVE command normally saves files in the condensed form. The condensed form not only saves storage area, but also loads faster. Some files however, must be stored on disk in their listed form. One example of such files are those to be MERGEd. Files are saved in their listed form by means of the ASCII save. The format for the ASCII save is:

SAVE <filename>,a

where "a" specifies ASCII.

The MERGE command creates a new program in memory by merging the lines of the specified program on disk with the lines of the program currently in memory. The format of the MERGE command is:

MERGE <filename>

The file specified by <filename> must be one which was saved by an ASCII save.

3.7 Deleting Filenames — KILL Command

Files may be deleted from a disk by means of the KILL statement. Its format is:

KILL <filename>

Examples:

kill "prog1"
kill "2:fileA"

If a file is SAVEd under the same name as a file already on the disk, the previous file will be deleted, and the new file written in its place. This is convenient in the sense that it is not necessary to first issue a KILL command to delete the old file, but can be dangerous, since it is possible to destroy a file by forgetting which filenames are presently being used. To prevent this, the "write protect" attribute, explained in the next section, can be assigned to important files.

3.8 File Attribute and Set Statement

There are two file attributes:

1. Write-protect
2. Read-after-write

Attributes are specified by the SET statement. There is no default attribute. The write-protect attribute should be assigned to important files to protect them from inadvertent modification or deletion.

The read-after-write statement verifies the data written to the disk is a copy of that which resides in RAM.

One format of the SET statement is:

SET <filename>, <attribute>

<attribute> is normally assigned as R, specifying read-after-write, or P, specifying write-protect. If any other character is used for <attribute>, the SET statement will merely cancel the current attribute.

Examples:

set "prog1","P"	Assigns the write-protect attribute to the file "prog1".
set "prog1"," "	Cancels any attribute of the file "prog1".

Note: The read-after-write and write-protect attributes are assigned by a capital "R" and capital "P". Using "p" or "r" in a SET statement will merely cancel the current attribute.

There are some additional forms of the SET statement shown below.

(a) SET <drive number>, <attribute>

This form of the SET statement assigns an attribute to drive. Thereafter, any new files written to that drive will assume the drive's attribute. The formats of <drive number> and <attribute> are the same as above. A disk's attribute is stored in the first byte of the ID. When the MOUNT command is issued, the ID is read and this value is set into RAM to serve as a default for the attribute.

(b) SET <file number>, <attribute>

This form of the SET statement assigns a temporary attribute to the file OPENed as #<file number>. The attribute is valid only so long as the file is OPENed, and does not affect attributes recorded on the disk.

3.9 Changing Filenames — NAME Command

The NAME statement changes the name of files already stored on disk. Its format is:

NAME <old filename> as <new filename>

Example:

name "oldfil" as "newfil"

3.10 RUN Command and R Option

In DISK BASIC the RUN command has the following optional forms.

(1) RUN <filename>

This form of the RUN command will load the specified file and immediately execute it.

Example:

```
run "prog1"
```

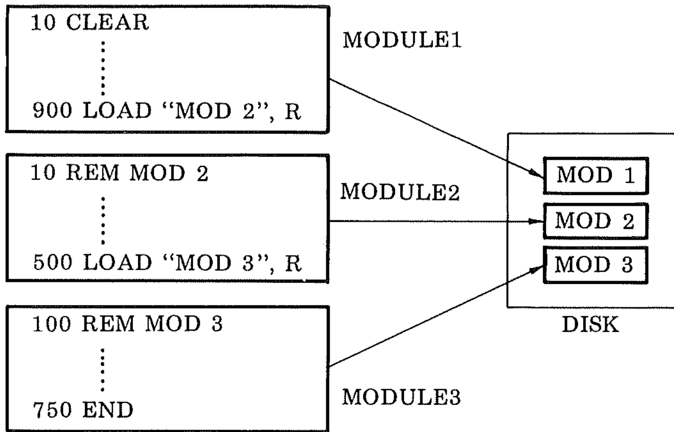
The R option can be used with the LOAD command and RUN command. The formats of these commands when used with the R option are:

```
LOAD <filename>, r
RUN <filename>, r
```

Files which are open when the above commands are executed are not closed, as they are without the R option. The R option is convenient for chaining programs, as explained in the next section. The OPEN and CLOSE statements are explained in the next chapter.

3.11 Chaining Programs

Programs which are too long to fit into RAM may be divided into modules, which are stored on disk and called in sequence. This process is called "chaining". In chaining programs the LOAD command is used as a program statement along with the R option. This is illustrated in the next chart.



The last line of each module, except for the last one, contains the **LOAD** command with the R option. Thus, by calling module 1 using the **LOAD** command with the R option, all three modules will be executed. When the last line of module 1 is executed, all lines of the program currently in memory, namely module 1 itself, will be cleared, and module 2 will be loaded and executed. Note that only one module exists in **RAM** at any one time.

Very large programs can be executed by chaining; however, the following special considerations must be kept in mind when preparing modules for chaining.

1. The same line numbers may be used in different modules since all memory is cleared each time a module is loaded.
2. No statement may refer to a line number outside its own module; that is, it is impossible to jump from module to module by using such statements as **GOTO** or **GOSUB**.
3. Data cannot be directly transferred from one module to the next, because all program memory is cleared each time a module is loaded. Data can, however, be transferred by means of disk files. In the example below the values of variables A, B, and C$ are passed

in this manner.

```
         10  REM MODULE MOD 1

         100 OPEN "VAR" FOR OUTPUT AS #1
         110 PRINT #1,A,B
MOD 1    120 PRINT #1,C$
         130 CLOSE #1
         140 LOAD "MOD2", R

         10  REM MODULE MOD 2
         20  OPEN "VAR" FOR INPUT AS #1
MOD 2    30  INPUT #1,A,B
         40  LINE INPUT#1, C$
         50  CLOSE #1
              .
              .
              .
```

Chaining programs considerably slows down execution speed, especially where large amounts of data must be passed from module to module. This disadvantage of chaining programs should always be taken into consideration. Details of the statements used to create the date files in the above example are given in the next chapter.

CHAPTER 4: Sequential Files

4.1 Meaning of "Sequential File"

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 in the order it is sent, and is read back in the same way; whereas, data may be written or read from a random file starting from any location.

The statements and functions that are used with sequential files are: **OPEN, PRINT #, INPUT #, PRINT# USING, LINE INPUT #, CLOSE, EOF,** and **LOC**.

When the **LOC** function is evaluated for a sequential file, the returned value is the number of sectors used since that file was opened. (One sector consists of 256 bytes).

4.2 OPEN Statement

In **DISK BASIC**, all file references are made by filename, and before any disk I/O is performed, the system must be told the name of the file concerned. This is done using the **OPEN** statement.

By means of the **OPEN** statement, the user declares the following three items to the system.

1. The name of the file for which I/O is requested.
2. Kind of operation being requested (input or output).
3. Reference number.

The following example illustrates the meaning usage of the open statement.

```
OPEN "data1" FOR INPUT AS #1
       ↑       ↑            ↑
      (I)     (II)        (III)
```

The example statement will tell the system that we wish to read from a file by the name of "data1". When this file is referenced by a PRINT # statement or INPUT #

statement, it must be "named" in some manner. Of course, the file could be referred to by its filename: however, to simplify matters, each time a file is **OPENed** a temporary reference number is assigned to it. This is the reference number mentioned in item 3 above. This number serves as a kind of "abbreviated name", and until the file is **CLOSEd**, this number may be used in place of the filename. This reference number is properly called the "file number", and declaration of file use is called "opening" a file.

Each time a file is **OPENed** it may be assigned a different file number.

Before performing I/O for a data file, it must be **OPENed**. If not, an error will result, and the message "File not OPEN" will be displayed.

The file number may be an integer from 1 to the number specified in reply to the message, "HOW MANY FILES?" (Maximum 15). As you probably realize by now, this message is asking what is the maximum number of files that will be open at any one time.

So as to not complicate matters, in Chapter 2 we merely instructed you to input "3" in reply to the above message. In this case, permissible file numbers would be 1, 2, and 3.

The number given in reply to the "HOW MANY FILES" message determines how many file I/O buffers will be reserved. As can easily be checked by the **FRE** function, the larger the number specified, the less **RAM** available to the user, thus a number no larger than necessary should be specified. For most applications three I/O buffers, as specified in the example, should be sufficient. Later, in the section on the **CLOSE** statement, we will explain in detail what we mean by "the number of files open simultaneously".

A file may be opened for:

1. **INPUT**
2. **OUTPUT**
3. **APPEND**

Or, this specification may be omitted. The meaning of "FOR INPUT" and "FOR OUTPUT" is obvious by now. In the beginning of this chapter, we explained that the one type of revision possible with sequential files is the addition of records to the end of the file. This is indicated by opening a file "FOR APPEND". Although OPEN statements in which the "FOR" specification has been omitted are most frequently used in connection with random access files, such statements may also be used for sequential files. We will explain the use of these statements later, but first let's see how the INPUT # and PRINT # statements are actually used.

4.3 Reading and Writing Sequential Files

Sequential files are input and output using the INPUT # and PRINT # statements, and their variations, the LINE INPUT # and PRINT # USING statements.

First, let's discuss the INPUT # and PRINT # statements. The general formats of these statements are:

```
INPUT # <file number>, <list of variables>
PRINT # <file number>, <list of data for output>
```

Examples:

```
INPUT # 1, A, B, C
INPUT # 2, A$, B$
PRINT # 1, 5, SIN(A), "DISK"
```

In the examples above, the file numbers are those specified in the corresponding OPEN statements.

Next, let's write an example program.

```
100 OPEN "TEST" FOR OUTPUT AS #1
200 FOR I = 1 to 10
300 PRINT #1, I, I∧2, SOR (I)
400 NEXT I
500 CLOSE
```

There is a statement in this program which we have not yet explained. That is the **CLOSE** statement in line 500. For the time being we will simply explain the function of the **CLOSE** statement as declaring the end of file use.

Of course, the example program is not a practical program. There would be no reason for storing a table of squares on disk as a data file. At any rate, let's execute the program and see what happens. The disk will be accessed briefly, and the "OK" prompt will be displayed. The file "TEST" should now be stored on disk. This can be checked with the **FILES** command.

Next, let's write a program for reading the file we have just created. Such a program might be written as follows:

```
100 OPEN "TEST" FOR INPUT AS #1
200 INPUT #1, I, J, K
300 PRINT I, J, K
400 GOTO 200
```

Now, let's execute our program.

```
run
 1             1
 1
 2             4
 1.41421
 3             9
 1.73205
 4             16
 2
 5             25
 2.23607
 6             36
 2.44949
 7             49
 2.64575
 8             64
 2.82843
 9             81
 3
 10            100
 3.16228
Input past end in 200
Ok
```

Figure 3

As you can see from the error message at the end of run listing in Figure 3, there is an error in our program. The "Input past end" message tells us that there was an attempt to read more data items than existed in the file.

The program reads all items in the file sequentially. After all items have been read, the next execution of line 200 will cause an "Input past end" error. There are several methods to avoid this problem. One idea is to write the number of items as well as items themselves. Figures 4 and 5 show the programs based on this idea.

```
list
10 REM fig 4
20 OPEN "TEST" FOR OUTPUT AS #1
30 PRINT #1,10
40 FOR I=1 TO 10
50 PRINT #1,I,I^2,SQR(I)
60 NEXT I
70 CLOSE
Ok
```

Figure 4

```
list
100 OPEN "TEST" FOR INPUT AS #1
110 INPUT #1,COUNT
120 FOR L=1 TO COUNT
200 INPUT #1,I,J,K
300 PRINT I,J,K
400 NEXT L
Ok
```

Figure 5

Another method to avoid the "Input past end" error is to store a special value as the last data item in the file. Example programs using this method are shown in Figure 6 and Figure 7.

```
list
10 REM fig 6
20 OPEN "TEST" FOR OUTPUT AS #1
30 FOR I=1 TO 10
40 PRINT #1,I,I^2,SQR(I)
50 NEXT I
60 PRINT #1,-1
70 CLOSE
Ok
```

Figure 6

```
list
10 REM fig 7
20 OPEN "TEST" FOR INPUT AS #1
30 INPUT #1,I
40 IF I=-1 THEN END
50 INPUT #1,J,K
60 PRINT I,J,K
70 GOTO 30
Ok
```

Figure 7

Above method does not work if the special value used for a stopper is contained in data items.

The most convenient solution is to use **EOF** function. **EOF** (End Of File) function returns false (= 0) if more data are left, true (= −1) if the end of the file has been reached. Syntax of **EOF** function is:

EOF (<file number>)

Now, we can write the final version of the example program which is shown below.

```
list
10 REM fig 8
20 OPEN "TEST" FOR INPUT AS #1
30 IF EOF(1) THEN END
40 INPUT #1,I,J,K
50 PRINT I,J,K
60 GOTO 30
Ok
```

Figure 8

In our example programs, we have used files which consisted of numerical tables. Normally, when using such tables, we do not read all data in the table sequentially, from the beginning to end, but rather pull out and use only that portion we are interested in. This is possible with sequential files, but not very practical. For example, if we wanted to read the square and square root of one particular number only from the file "test", we might use the following program.

```
list
10 REM fig 9
20 OPEN "TEST" FOR INPUT AS #1
30 INPUT I
40 FOR L=1 TO I
50 INPUT #1,A,B,C
60 NEXT L
70 PRINT A,B,C
80 CLOSE
90 GOTO 20
Ok
```

Figure 9

In the above example, we have, with a bit of effort, written a program which will read and display the square and square root of the number 1 to be input from the keyboard. In line 30, input of the number for which we wish to find the square and square root is requested. In

lines 40 through 60, the file is being read merely to get up to the required data item. Finally, after the desired data is read in the last execution of the **FOR-NEXT** loop, it is printed by line 70. Before the next data item is read, the file must be **CLOSEd** and re-**OPENed**. As explained in Section 4.5, this operation is analogous to rewinding a tape.

Sequential files are best suited for jobs in which all data must be read from beginning to end each time. When only portions of data are to be accessed, it is best to use random access files.

The explanation so far is sufficient for handling numerical data, however, a few more points should be mentioned concerning alphabetic data. To introduce these points, let's first execute the following two example programs.

```
list
10 REM fig 10
20 OPEN "test" FOR OUTPUT AS #1
30 PRINT #1,"abcdefg";"hijklmn"
40 CLOSE
Ok
█
```

Figure 10

```
list
10 REM fig 11
20 OPEN "test" FOR INPUT AS #1
30 INPUT #1,A$
40 PRINT A$
50 CLOSE
Ok
run
abcdefghijklmn
Ok
```

Figure 11

Looking at the two example programs, and notice that the desired results are not obtained. The output of the program in Figure 10 was two data items. However, the file is read by the program in Figure 11, the data items are concatenated and output as one data item when the program runs. This can be explained as follows:

The **PRINT #** statement writes data to disk in exactly the same format as displayed on the screen. Also, the data input from disk with the **INPUT #** statement is handled in the same manner as data input from the keyboard. Now, let's look back at our examples. The **PRINT #** statement in Figure 10 writes the two data items to disk as one continuous item. Then, when the **INPUT #** statement in Figure 11 is executed, it will read everything from the beginning of file "test" up to the first carriage return/line feed or comma as one data item. Thus, to correct the program in Figure 10, the **PRINT #** line should be written.

PRINT #1, A$;",";B$

or, written as two lines,

PRINT #1, A$
PRINT #1, B$

It is not correct to write the line as,

PRINT #1, A$,B$

writing the statement like this will only result in spaces being inserted between the two strings, A$ and B$. When read from disk, the two strings will be separated by the spaces, but they will still be read as one data item. The only way to separate the two items is to actually write a "," to disk.

As the reader may have realized, a problem arises when a commas is actually part of a data item. To demonstrate what we mean, let's execute the following two programs.

```
list
10 REM fig 12
20 OPEN "test" FOR OUTPUT AS #1
30 A$="abcdef,ghijkl"
40 PRINT #1,A$
50 CLOSE
Ok
```

Figure 12

```
list
10 REM fig 13
20 OPEN "test" FOR INPUT AS #1
30 INPUT #1,A$
40 PRINT A$
Ok
run
abcdef
Ok
```

Figure 13

As you can see from the run of the program in Figure 13, the **INPUT** # statement interpreted the comma in the string A$ as a delimiter, and read only up to that point. There are two ways of avoiding this problem.

In the first method, illustrated in Figure 14, the problem is corrected at the time the file is written to disk. In the program in Figure 14, quotation marks are written to the disk immediately preceding and immediately following the string A$.

```
list
10 REM fig 14
20 OPEN "test" FOR OUTPUT AS #1
30 A$="abcdef,ghijkl"
40 QUOTE$=CHR$(34)
50 PRINT#1,QUOTE$+A$+QUOTE$
60 CLOSE
Ok
```

Figure 14

The file formed by this program may be read by the program in Figure 13. In this case, everything between the quotation marks, including any commas, will be interpreted as data. However, this method will not work for strings containing quotation marks as data.

In the second method, the problem is corrected when the file is read from disk. This is done by using the **LINE INPUT #** statement in place of the **INPUT #** statement. The program in Figure 15 will properly read the file created by the program in Figure 12.

```
list
10 REM fig 15
20 OPEN "test" FOR INPUT AS #1
30 LINEINPUT #1,A$
40 PRINT A$
Ok
```

Figure 15

When reading data from disk, the **LINE INPUT #** statement functions just like the **LINE INPUT** statement does for data input from the keyboard. All input up to a carriage return/line feed is interpreted as data. This includes commas and quotation marks.

Here we would like to introduce the disk version of the **PRINT USING** statement. The beginning of the **PRINT # USING** statement is formatted as:

PRINT # <file number>, USING - - - - - -

That portion of the statement following "USING" is identical to the non-disk version.

Example:

PRINT #1, USING "####";A$, B$, C$

4.4 CLOSE Statement

Up until now, we have used the **CLOSE** statement in many of the example programs without really explaining it. This section gives a detailed explanation of the functions of the **CLOSE** statement.

Before reading or writing a file, use of that file must be requested by the **OPEN** statement. In the same manner, when you are through using a file, you must declare this fact to the system. This declaration is performed by the **CLOSE** statement. Once a file has been opened by an **OPEN** statement, that file stays open until closed by a **CLOSE** statement.

When the system was started, you had to reply to the question, "How many files?". This question asks not the total of files to be used during the course of your program, but rather the maximum number of files to be opened simultaneously. This number may be less than the total number of files used.

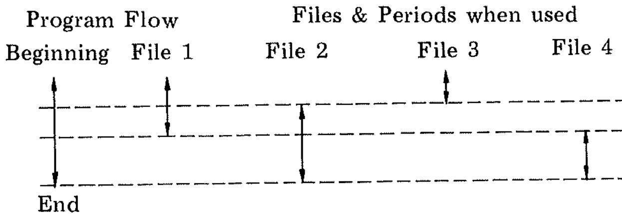


Figure 16

In the example in Figure 16, a total of four files is used, but no more than two files are used simultaneously. In this case, replying "3" to "How many files?" provides more buffer space than really needed; however, only the file numbers 1, 2, and 3 may be used. Since there are more files than file numbers, after a file has been **CLOSEd**, its file number may have to be re-used to **OPEN** another file. To show how this applies to the example in Figure 16, let's

expand the figure and assign file numbers.

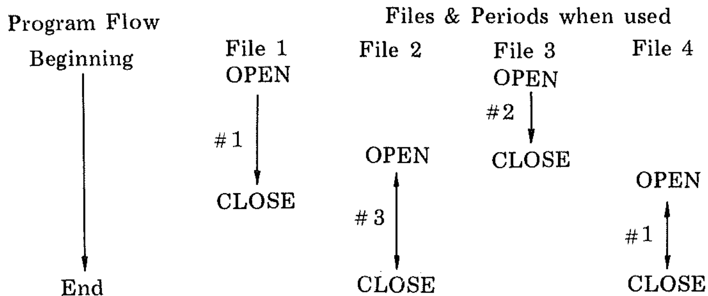


Figure 17

In the example in Figure 17, the file number # 1 has been used for both file A and D. The relation between a file and it's file number exists only as long as the file remains open. The relation between a file and its file number is terminated by the **CLOSE** statement. The format of the **CLOSE** statement is:

CLOSE
CLOSE <file number> **[**,<file number>**]**

Example:

CLOSE
CLOSE 1
CLOSE # 3
CLOSE 1,2,3

When the argument of the **CLOSE** statement is omitted, as in the first example, all files are closed. If file numbers are given in the **CLOSE** statement, only those files are closed. As shown in the second and third examples, the pound sign, "#" is optional.

Additionally, any **OPEN** files are automatically **CLOSEd** whenever:

1. an **END** statement is executed.
2. Program revision occurs.
3. Execution of a **REMOVE** command.
4. Execution of **RUN** or **LOAD** commands.

This concludes the fundamental explanation of sequential files. Before going on, however, let's take a look at another example program.

The following program reads strings from the keyboard until "end" is input, and then writes these strings to disk. Next, it reads the file created, converts all numbers to asterisks (*), and writes this data to a new file. Finally, it reads and displays the converted data stored in the second file. This program serves no purpose other than to demonstrate the various ways of handling sequential files.

```
list
10 '
20 '
30 '
40 OPEN "data" FOR OUTPUT AS #1
50 LINEINPUT "?";A$
60 IF A$="end" THEN 90
70 PRINT #1,A$
80 GOTO 50
90 CLOSE #1
100 '
110 '
120 '
130 OPEN "data" FOR INPUT AS #1
140 OPEN "data1" FOR OUTPUT AS #2
150 IF EOF(1) THEN 230
160 LINEINPUT #1,A$
170 IF LEN(A$)=0 THEN 210
180 FOR I=1 TO LEN(A$)
190 IF MID$(A$,I,1)>="0" AND MID$(A$,I,1
)<="9" THEN MID$(A$,I,1)="*"
200 NEXT I
210 PRINT #2,A$
220 GOTO 150
230 CLOSE 1,2
240 '
250 '
260 '
270 OPEN "data1" FOR INPUT AS #1
280 IF EOF(1) THEN 320
290 LINEINPUT #1,A$
300 PRINT A$
310 GOTO 280
320 CLOSE #1
330 END
Ok
```

Figure 18

```
run
?abc123defgh789ijk lmn
?o0pq678rstu338...
?12345678
?zxcv b44   4ff
?end
abc***defgh***ijk lmn
o*pq***rstu***...
********
zxcv b**   *ff
Ok
```

Figure 19

4.5 How the Interpreter Processes OPEN and CLOSE Statements

This section gives a few details on the data structure which the N-BASIC interpreter uses to handle OPEN and CLOSE statements.

File Pointer:

N-BASIC has one pointer for each OPENed file which indicates the position that is being accessed in the file. The OPEN statement always initializes this pointer.

OPENing a file for input:

When an OPEN for input statement is issued, the file pointer is set to the top of the file. The file specified must already exist on disk.

OPENing a file for output:

When an OPEN for output statement is issued, if the specified file exists on disk, the file pointer is set to the top of the file, and the previous contents of the file will be replaced by newly written data. If the specified file does not exist on disk, the new filename is created in the directory, and the file pointer is set to the top of the disk space reserved for that file.

OPEN statement with mode omitted:

When an **OPEN** statement without the mode clause is given, if the specified file exists on the disk, the file pointer is set to top of the file. If the specified file does not exist, the new file with given file name is created, and the file pointer is set to the top of the new file.

OPENing a file for append:

An **OPEN** statement for append sets the file pointer to the bottom of the specified file. The file should already exist on the disk. Otherwise, an error message will be given.

Disk buffer:

N-BASIC maintains an I/O buffer for each file being accessed. One I/O buffer holds one sector of data, i.e., 256 bytes. Normally, data is not written to disk until the buffer is full, and then the entire buffer is written at once. Similarly, when reading the value of a variable from the disk, an entire sector must be read if the desired sector is not already in the I/O buffer. This is because the minimum disk access unit is one sector.

An **OPEN** statement reserves one of the available I/O buffers for the file being opened. This buffer is not available for use by other files until the file for which it was reserved is closed. The total number of I/O buffers available is specified in answer to the "How many files?" message during system initialization.

4.6 Appending Records to Sequential Files

The only modification possible for sequential files is the addition of new records to the end of the file. Figure 20 contains an example program in which files are opened for append, and Figure 21 contains the run listing for this program. Lines 10 to 60 of the example program create a sequential file, and line 80 prints the file data by calling a subroutine in lines 200 to 240. Next, lines 100 to 130 append new data to the file, and line 150 prints the

modified file.

```
10 REM create data file
20 OPEN "test.tmp" FOR OUTPUT AS #1
30 FOR I=&H30 TO &H35
40 PRINT #1,STRING$(20,I)
50 NEXT I
60 CLOSE
70 PRINT "data before append"
80 GOSUB 200
90 REM append
100 OPEN "test.tmp" FOR APPEND AS #1
110 PRINT #1,STRING$(20,"?")
120 PRINT #1,STRING$(20,"!")
130 CLOSE
140 PRINT "data after append"
150 GOSUB 200
160 CLOSE
170 END
200 REM type out data file
210 OPEN "test.tmp" FOR INPUT AS #1
220 IF EOF(1) THEN CLOSE:RETURN
230 INPUT #1,A$:PRINT A$
240 GOTO 220
Ok
```

Figure 20

```
run
data before append
00000000000000000000
11111111111111111111
22222222222222222222
33333333333333333333
44444444444444444444
55555555555555555555
data after append
00000000000000000000
11111111111111111111
22222222222222222222
33333333333333333333
44444444444444444444
55555555555555555555
????????????????????
!!!!!!!!!!!!!!!!!!!!
Ok
```

Figure 21

CHAPTER 5: Random Access Files

As mentioned several times before, arbitrary locations within a sequential file cannot be directly accessed. This problem has been corrected in random access files by structuring files in units of "records", any one of which may be directly accessed. Before discussing the grammer of random access files, let's take a look at the concept of a "record".

5.1 Records

Until now, the word record has been used rather loosely, almost as if it has the same meaning as the word "data". Actually, "record" is a precise term which we will now define.

> A record is a unit of data transferred in one logical I/O operation.

As explained before, the physical unit for disk I/O transfer is the sector; however, the above definition refers to logical I/O.

Sequential files are delimited by a carriage return and a line feed and may be variable length. The basic structure of sequential files is illustrated in Figure 22.

```
ABCDEFC/RL/F        100, 200 C/RL/F        ZC/RL/F
└─1 Record─┘        └─1 Record─┘           |       |
                                           1 Record
```

Figure 22

Before going any further, let's introduce two new terms: "variable length records", such as illustrated in Figure 22, and "fixed length records", which will be explained below. Until now, all files were categorized as random access files or sequential files, telling us how the files were accessed. The terms "fixed length record" and "variable length record" tell us how a file's records are formatted.

If an appropriate look-up table were created, it would be possible to construct random access files using variable length records; however, this would defeat the main purpose of random access files — speed. To achieve maximum access speed it is necessary to use fixed length records, which we will now discuss.

As the name implies, fixed length records all have the same length, allowing easy computation of the location of any record within the file. Random access files in the PC-8001's **DISK BASIC** use fixed length records with a length of 256 bytes, i.e., one sector.

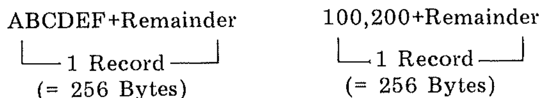


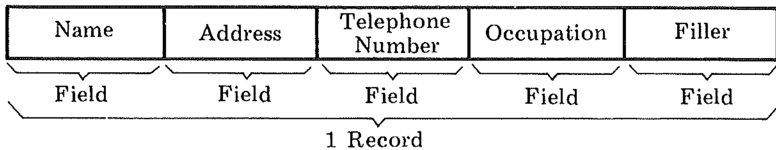
Figure 23

As shown in Figure 23, most fixed length records will end with an unused portion.

As a result of using fixed length records in random access files, access speed goes up, but efficiency in use of disk area goes down. Exactly the opposite can be said of sequential files. You, as the user, must decide which is most important — speed or storage capacity, and format your files accordingly.

5.2 Fields

As an example, let's create a name list using a random access file. In such a case, it would be best to structure the file so that one record contained all information on one person. If this information consists of the person's name, address, telephone number and occupation, the record would be structured as follows:



In the above manner, each record is divided into a number of fields. When using random access files, it is necessary to determine the number of fields required for each record, and the length of each field.

5.3 Flow of Random File Processing

The individual statements used in processing random access files will be explained in later sections. First, however, we would like to present the following steps as an outline to help the reader understand overall program flow.

1. OPEN file. (OPEN statement)
2. Define each field within each record. (FIELD statement).

When writing:

3. Set the data for each field of one record. (LSET or RSET statement).
4. Write the sector to the disk. (PUT statement)

When reading:

3. Read a sector from the disk. (GET statement)
4. Retrieve data from respective fields within buffer.

Random access files are more difficult to use than sequential files. Since it is impossible to understand example programs without understanding each of the statements listed above, we have included an explanation of each in the following pages. Please bear with these explanations and you will find the information presented to be of great use later on.

5.4 Opening Files

Random access files are opened similarly to sequential files, except that no mode is specified for random access files. (See explanation of mode option for **OPEN** statement).

As was previously explained, one I/O buffer is allocated for each file. For random access files, this buffer is called a "random buffer".

5.5 Defining Fields

Each field must be defined using the **FIELD** statement. The format of the **FIELD** statement is:

FIELD # <field number>, <field specification>

The format for <field specification> is:

```
<field width> AS <string variable>, <field width>
    AS <string variable> ------
```

Example:

FIELD #1 10 AS A$, 20 AS B$, 50 AS C$

Execution of this line causes each record of file #1 to be divided into three fields. The first field is 10 bytes long and is reserved for string A$, the second field is 20 bytes long and is reserved for B$, and the third field is 50 bytes long and is reserved for C$. The remaining 176 bytes (256 − 10 − 20 − 50 = 176) of each record are not used.

The FIELD statement is used only to specify how the area within each record of a particular file is to be used. This statement has nothing to do with actually reading or writing data.

CAUTION

String variables used in FIELD statements must not appear on the left side of LET statement, nor be used in INPUT or READ statement. Such use would cause the GET and PUT statements to not execute properly.

****
BAD EXAMPLE
****

```
100 FIELD #1 100 AS A$, 100 AS B$
200 A$ = "BAD"
300 PRINT A$
400 GET #1
```

****

5.6 LSET and RSET Statements

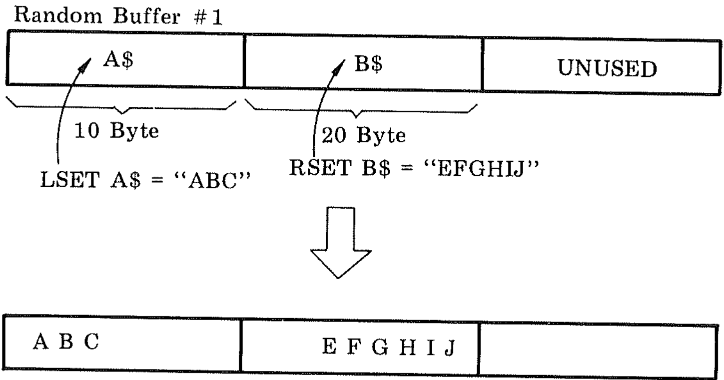
The LSET and RSET statements are used to set the actual data to be written to the disk into the fields of the random buffer. The formats of LSET and RSET statements are:

```
LSET <string variable> = <string>
RSET <string variable> = <string>
```

Examples:

```
FIELD #1, 10 AS A$, 20 AS B$
LSET A$ = "ABC"
RSET B$ = "EFGHIJ"
```

When this program segment is executed, the string "ABC" is set into the first field of the random buffer corresponding to file # 1, and the string "EFGHIJ" is set into the second field of the same buffer. It should be remembered that at this point the data has merely been placed into the random buffer located in RAM, and no writing to the disk has taken place.



The difference between **LSET** and **RSET** is that the LSET statement left justifies the data it stores and the RSET statement right justifies the data it stores. For either statement, unused portions of a field are filled with spaces. If a string is longer than the field it is being **SET** into, the portion that overflows the field is truncated.

5.7 PUT Statement

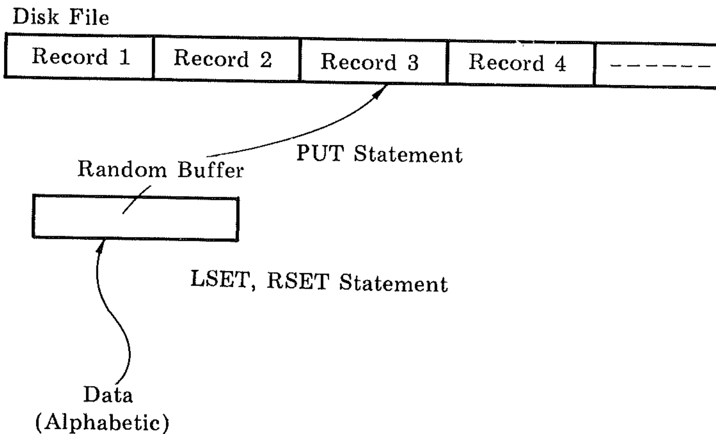
The **PUT** statement writes the contents of a random buffer to the disk. The general format of the **PUT** statement is:

PUT [#] <file number> [, <record number>]

<record number> indicates the record to which the contents of the random buffer are to be written.

Example:

PUT # 1, 3



As shown in the above figure, data is written to the disk in two steps:

1. data → buffer (**LSET** or **RSET**)
2. buffer contents → disk (**PUT**)

If the record number is omitted from a **PUT** statement, its default is the record number immediately following that of the most recent **GET** or **PUT** statement.

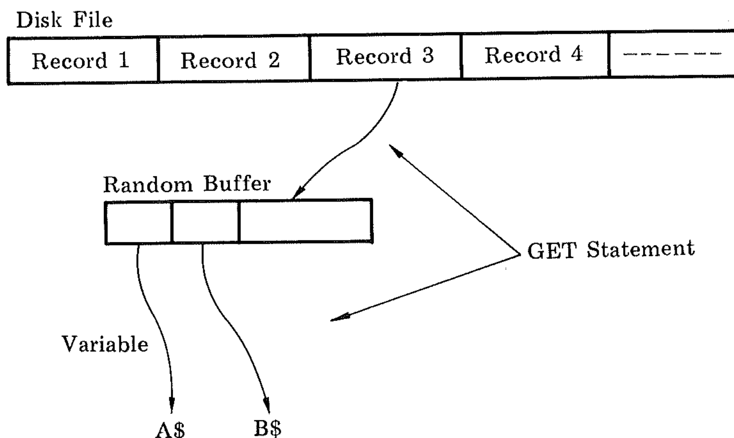
5.8 GET Statement

The **GET** statement reads one record from disk into the random buffer, and sets the variables specified in the **FIELD** statement equal to the values written into their corresponding fields. The formats of the **GET** statement are the same as those of the **PUT** statement.

GET [#] <file number> [, <record number>]

Example:

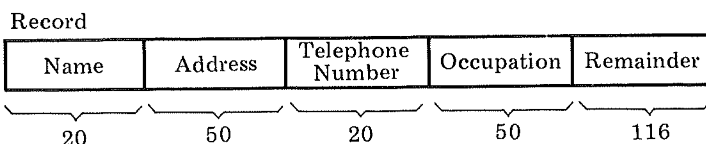
```
FIELD # 1, 10 AS A$, 20 AS B$
GET # 1, 3
```



If the record number is omitted from a **GET** statement, the record number immediately following that of the most recent **GET** or **PUT** statement is assumed.

5.9 Using Random Access Files (Example 1)

In the preceding sections we have briefly covered the basic statements used in handling random access files. Now we would like to show you some actual examples of how to write programs using random access files. First, let's write a program for making the name list used as an example in Section 5.2. Each record contains all the data for one person and that data consists of the items illustrated below.



The total number of bytes allocated to the different fields must not exceed 256. In the example, 140 bytes have been allocated. The remaining 116 bytes are unused. An example program for creating such a file is shown in Figure 24, and a run of the program is shown in Figure 25.

```
10  REM fig. 24
20  CLEAR 1000
30  OPEN “rndtst” AS #1
40  FIELD #1, 20 AS N$, 50 AS A$, 20 AS T$, 50
        AS O$
50  INPUT “read (r) or write (w)”;P$
60  IF P$ = “r” THEN 70 ELSE IF P$< >“w”
        THEN PRINT “error”: GOTO 50
70  INPUT “record number”;R
80  IF P$ = “w” THEN 170
90  REM read
100 GET #1, R
110 PRINT “name:”;N$
120 PRINT “address:”;A$
130 PRINT “telephone number:”;T$
140 PRINT “occupation:”;O$
160 GOTO 50
170 REM write
180 LINE INPUT “name”;I$
190 LSET N$ = I$
200 LINE INPUT “address”;I$
210 LSET A$ = I$
220 LINE INPUT “telephone number”;I$
230 LSET T$ = I$
240 LINE INPUT “occupation”;I$
250 LSET O$ = I$
280 PUT #1, R
290 GOTO 50
OK
```

Figure 24

```
run
read (r) or write (w)? w
record number? 1
name? Albert Jones
address? Southbend, Indiana
telephone number? 429-7712
occupation? engineer
read (r) or write (w)? w
record number? 2
name? Ronald Smith
address? Albany, New York
telephone number? 472-9927
occupation? doctor
read (r) or write (w)? r
record number? 1
name: Albert Jones
address: Southbend, Indiana
telephone number: 429-7712
occupation: engineer
read (r) or write (w)?
```

HT auto goto list run[cr]

Figure 25

5.10 Handling Numerical Data

All data is stored in random access files as character strings. To store numeric data in a random access file the data type must be converted so that the data will be written as a string. Conversely, data from a random access file must be read as a character string, and the type must be changed back to be interpreted as numeric data. The following functions are provided to perform the type conversions.

MKI$: Converts an integer to a two-character string.
MKS$: Converts a single-precision number as a four-character string.
MKD$: Converts a double-precision number as an eight-character string.
CVI: Converts a two-character string as an integer.
CVS: Converts a four-character string as a single-precision real number.
CVD: Converts an eight-character string as a double precision real number.

The following example program shows how the integer variables A% and B%, and the single-precision real number variables A and Z are written and read from a random access file.

```
list
10  REM fig. 26
20  OPEN "rndnum" AS #1
30  FIELD #1, 2 AS A1$, 2 AS A2$, 4 AS A3$, 4
         AS A4$
40  INPUT A%, B%, A, Z
50  REM write
60  LSET A1$ = MKI$ (A%)
70  LSET A2$ = MKI$ (B%)
80  LSET A3$ = MKS$ (A)
90  LSET A4$ = MKS$ (Z)
100 PUT #1, 1
110 REM read
120 GET #1, 1
130 PRINT CVI(A1$), CVI(A2$)
140 PRINT CVS(A3$), CVS(A4$)
150 CLOSE
OK
```

Figure 26

5.11 Using Random Access Files (Example 2)

The name list program used as an example in Section 5.9 was ridiculously clumbsy. For one reason, when using random access files it shouldn't be necessary for the operator to specify a record number each time he forms a new record. With random access files, the sequence in which the records are written is generally not of importance, as it is for sequential files. The only matter of concern is whether there is any available area remaining in the file. Another problem with the program in Section 5.9 was that to read out the data for a particular person, it was necessary to specify the proper record number. To do this, the operator would have to keep a list of all names in the file and the corresponding file numbers, which would defeat the purpose of making the file. In this section let's write a program in which these faults have been corrected.

The first problem mentioned above is easily corrected by using the **LOF** function. The value returned by the **LOF** function is the number of the last record read or written. Thus, when a new record is written, its number can be specified as **LOF+1**. The format of the **LOF** function is:

LOF (<file number>)

If records of a random access file are written discontinuously it is possible that some records within the file will be unused. Since the **LOF** function always returns the number of the 1st record written, it can be used to sequentially assign record numbers, without leaving any gaps. However, any unused records prior to use of this function will remain. The solution is to always write to the record with the smallest available record number, right from the start. In Figure 27 there is an example program in which problem 1 above has been corrected.

```
10 REM fig 27
20 REM inproved version of fig26
30 CLEAR 1000
40 OPEN "rndtst" AS #1
50 FIELD #1,20 AS N$,50 AS A$,20 AS T$,5
0 AS O$
60 WR=LOF(1)+1 ' next record # to be wri
tten
70 INPUT "read(r) or write(w)";P$
80 IF P$="w" THEN 170 ELSE IF P$="r" THE
N 90 ELSE PRINT "error":GOTO 70
90 INPUT "record number";R
100 REM read
110 GET #1,R
120 PRINT "name";N$
130 PRINT "address" ;A$
140 PRINT "telephone number" ;T$
150 PRINT "occupation" ;O$
160 GOTO 70
170 REM write
180 LINEINPUT "name" ; I$
190 LSET N$=I$
200 LINEINPUT "address" ; I$
210 LSET A$=I$
220 LINEINPUT "telephone number" ; I$
230 LSET T$=I$
240 LINEINPUT "occupation" ; I$
250 LSET O$=I$
260 PUT #1,WR
270 WR=WR+1
280 GOTO 70
Ok
```

Figure 27

Next, let's correct the second fault mentioned above. This is a little more difficult than the first problem. To help us tell the computer how to perform this task, let's first try to imagine how a human would go about forming a correspondence between the names in a list and their record numbers. Possible steps for such a procedure might be:

1. Prepare a correspondence chart on paper.
2. Update the chart each time a record is written.
3. When looking for a particular person's record, use the name as a key to look up the record number in the table.
4. Preserve the table for future use.

If we have the computer carry out the above procedure, the program must make and manage a new data file which will serve as the correspondence table. The correspondence table will contain pairs of data, each pair consisting of a person's name and the number of the record containing the data on that person. There will be as many of the pairs of data as there are entries in the name list file.

When the program is first executed the file containing the correspondence table can be read into memory, and an array formed from its data. Then while the program is running all reference to the table and modifications of the table can be made in RAM. Finally, before execution terminates, as one of the last steps, the up-to-date copy of the table in RAM can be written back to disk so that it is ready to be used the next time the program is executed.

By maintaining the correspondence table in RAM during program execution, the number of the desired record can be quickly looked up, referencing it by the person's name.

CHAPTER 6: Functions

In this chapter, more of the user-callable functions of PC-8001's DISK BASIC will be explained: DSKF, INPUT$, LOC, FPOS and ATTR$. We have explained eight of them in the previous chapters namely:

EOF . . . . Section 4.3
MKI$ . . . . Section 5.10
MKS$ . . . . Section 5.10
MKD$ . . . . Section 5.10
CVI . . . . Section 5.10
CVS . . . . Section 5.10
CVD . . . . Section 5.10
LOF . . . . Section 5.11

These will not be covered in this chapter. Also, there is one function which will not be covered until the next chapter. This is the DSKI$ function which will be covered in Chapter 7.

6.1 DSKF

The DSKF function returns the value of the available area remaining on the disk. Its format is:

DSKF (<drive number>)

The value returned is in units of extents. As explained in Chapter 8:

1 extent = 8 sectors = 2 K bytes

Example:

```
files
MAP SF     2          MASTER RND   2
rndtst     1          fig26.       1
tst        2          rndbug.      1
rndnum     1          fig28.       1
fig29.     1          fig30.       1
map        1          lintst       1
linbug     1
OK
print dskf (1)
    46
OK
```

Figure 28

6.2 INPUT$

INPUT$ is a function for reading a specified number of characters from disk. The general format is:

INPUT$ (<number of characters>, # <file number>)

A pound sign, "#", must appear immediately in front of the file number.

Example

```
list
10 '
20 ' *** input$ function ***
30 '
40 OPEN "test.tmp" FOR OUTPUT AS #1
50 FOR I=ASC("0") TO ASC("9")
60 PRINT #1,CHR$(I);
70 NEXT I
80 PRINT #1,
90 CLOSE
100 '
110 OPEN "test.tmp" FOR INPUT AS #1
120 INPUT "How many letters";A
130 IF A=0 THEN CLOSE:END
140 PRINT INPUT$(A,#1)
150 GOTO 120
Ok


run
How many letters? 2
01
How many letters? 4
2345
How many letters? 3
678
How many letters?
Break in 120
Ok
```

Figure 29

In lines 40 to 90 of the example program in Figure 29, the file "test.tmp" was created, and the string, "0 1 2 3 4 5 6 7 8 9" stored in this file. In lines 110 to 150, the file is **OPEN**ed, and the number of characters specified from the keyboard are read from the file and printed.

6.3 LOC

The **LOC** function returns the number of the next record to be read or written. As we know from the explanation of the **GET** and **PUT** statements, when the record number is omitted, the record number immediately following the number of the last record to be operated on by a **GET** or **PUT** statement is assumed. If you lose track of this number, it may be found by using the **LOC** function. The general format of the **LOC** function is:

LOC (<file number>)

The **LOC** function can also be evaluated for sequential files. In this case, the value returned will be the total number of sector read or written so far.

6.4 FPOS

The **FPOS** function is rarely used. The value which this function returns is the actual physical location, i.e., the sector number, of the head of a file. The numbering of sectors by this function is different from the normal sector number. Numbering starts at zero and is continuous throughout the disk. Thus, sector 1 of track 0 would correspond to a value of 0, and sector 1 of track 1 would correspond to a value of 16. The general format of **FPOS** function is:

FPOS (<file number>)

6.5 ATTR$

The **ATTR$** function returns the attribute of a specified drive or file. For an explanation of attributes, see Section 3.8. The various formats of the **ATTR$** function are:

ATTR$ (<drive number>)
ATTR$ (<#file number>)
ATTR$ (<filename>)

The above functions return the present attribute of a drive, the attribute of an **OPEN**ed file, and the fixed attribute (recorded in the directory) of a file. The strings which are returned as values of this function and their meanings are:

"␣␣P"	write protect
"␣␣R"	read after write
"␣␣␣"	no attribute

"␣" indicates one space.

Example:

```
set "prog1", "P"
OK
print attr$ ("prog1")
␣␣P
OK
```

CHAPTER 7: DSKO$ and DSKI$

In this chapter, one statement and one function will be introduced. This statement should be used with extreme caution.

7.1 DSKI$

DSKI$** is a function, whose value is the contents of the sector specified as its argument. By using the **DSKI$ function, disk data may be directly accessed by specifying its physical address. The general format of the **DSKI$** function is:

DSKI$ (<drive number>, <track number>, <sector number>)

Example:

A$ = DSKI$ (1,2,5)

Since the value of the **DSKI$** function is a string with a maximum length of 255 bytes, and since a sector contains 256 bytes, when the above example is executed, the last byte in the sector is not included in the string A$. To prevent this byte from being lost, random buffer #0 is permanently reserved for the **DSKI$** function, and when this function is performed the contents of the specified sector are written into this buffer.

Example:

```
100 FIELD # 0, 128 AS A$, 128 AS B$
200 DUMMY$ = DSKI$ (1,2,5)
```

When this segment is executed, the data in the first half of sector 5 of track 2 of drive 1 is assigned to variable A$, the data in the second half of the same sector is assigned to the variable B$, and the first 255 bytes of the sector will be assigned to the string DUMMY$. As mentioned previously, variables specified in **FIELD** statements must not be used on the left side of **LET** statements, or in **INPUT** or **READ** statements. For example, replacing "**DUMMY$**" in line 200 of the example program above with "A$" or "B$" would result in an error.

7.2 DSKO$

When this statement is executed, the sector specified as its argument is written with the contents of the random buffer specified by a **FIELD #0** statement. The format of the **DSKO$** statement is:

DSKO$ <drive number>, <track number>,
<sector number>

The following is merely an example. Please don't execute it!

```
100 FIELD #0, 128 AS A$, 128 AS B$
200 LSET A$ = STRING$ (128,255)
300 LSET B$ = STRING$ (128,0)
400 DSKO$ 1, 18, 1
```

A disk may be accessed by **DSKI$** and **DSKO$** without executing the **MOUNT** command, because **DSKI$** and **DSKO$** bypass the **FAT** table and disk directory. Therefore **DSKO$** must be used with extreme caution. All other disk functions, commands, and statements are controlled by the information in the **FAT** table and file attributes specified in the directory to prevent unintentional change to any data. Only the **DSKO$** statement is outside this control. Since **DSKO$** bypasses the directory, it is quite possible to damage a file that has a write protected attribute.

7.3 An Example of the DSKI$ Statement

The example of the **DSKI$** statement in the program below is a very useful utility program. This program dumps the contents of a specified sector in either hexadecimal or alphabetical format.

For a good example of the **DSKO$** statement, see the format utility.

```
10 '
20 ' file dump program
30 '
40 ' copyright 1979 by nippon electric c
ompany
50 '
60 CLEAR 1000
70 FIELD #0,128 AS A$,128 AS B$
80 INPUT "drive,track,secter in decimal"
;D,T,S
90 IF D<1 OR D>4 THEN 420
100 IF T<0 OR T>34 THEN 440
110 IF S<1 OR S>16 THEN 460
120 INPUT "output to crt(c) or lpt(l)";L
$
130 IF L$="c" THEN LPT=0:ELSE IF L$="l"
THEN LPT=1:ELSE 480
140 D$=DSKI$(D,T,S)
150 IF LPT THEN LPRINT"DUMP LIST":ELSE P
RINT "DUMP LIST    (c) 1979 by NEC"
160 IF LPT THEN LPRINT " DRIVE";D;",TRAC
K";T;",SECTR";S:ELSE PRINT " DRIVE";D;",
TRACK";T;",SECTOR";S
170 FOR I=1 TO 2
180 FOR J=1 TO 128/8
190 IF LPT THEN LPRINT HEX$((J-1)¥2+(I-1
)*8)+": ";:ELSE PRINT HEX$((J-1)¥2+(I-1)
*8)+": ";
200 X$=""
210 FOR K=1 TO 8
220 IF I=1 THEN D$=A$:ELSE D$=B$
230 H0$=MID$(D$,K+(J-1)*8,1)
240 H$=HEX$(ASC(H0$))
250 IF H0$<" " THEN H0$="." :ELSE IF H0$
>"z" THEN H0$=".":ELSE IF H0$>"Z" AND H0
$<"a" THEN H0$="."
260 X$=X$+H0$
270 IF LEN(H$)=1 THEN H$="0"+H$
280 IF LPT THEN LPRINT H$+" ";:ELSE PRIN
T H$+" ";
290 NEXT K
300 IF LPT THEN LPRINT "  "+X$:ELSE PRIN
T "  "+X$
310 NEXT J

 auto  go to  list  run
```

```
320 IF LPT THEN LPRINT :ELSE PRINT
330 NEXT I
340 '
350 ' goto next sector
360 '
370 S=S+1:IF S>16 THEN S=1:T=T+1:IF T>34
 THEN PRINT "end":END
380 GOTO 140
390 '
400 ' print error message
410 '
420 BEEP:PRINT "drive number should be 1
,2,3 or 4
430 GOTO 80
440 BEEP:PRINT "track number should be 0
 to 34
450 GOTO 80
460 BEEP:PRINT "sector number should be
1 to 16
470 GOTO 80
480 BEEP:PRINT "respond by c or l"
490 GOTO 120
Ok
```

```
run
drive,track,secter in decimal? 1,18,1
output to crt(c) or lpt(l)? c
DUMP LIST    (c) 1979 by NEC
 DRIVE 1 ,TRACK 18 ,SECTOR 1
0:  4D 41 50 20 20 20 53 46   MAP   SF
0:  20 00 23 FF FF FF FF FF    .#.....
1:  4D 41 53 54 45 52 52 4E   MASTERRN
1:  44 00 26 FF FF FF FF FF   D.&.....
2:  72 6E 64 74 73 74 20 20   rndtst
2:  20 00 1C FF FF FF FF FF    .......
3:  66 69 67 32 36 20 20 20   fig26
3:  20 80 20 FF FF FF FF FF    . .....
4:  74 73 74 20 20 20 20 20   tst
4:  20 00 28 FF FF FF FF FF    .(.....
5:  72 6E 64 62 75 67 20 20   rndbug
5:  20 80 1F FF FF FF FF FF    .......
6:  72 6E 64 6E 75 6D 20 20   rndnum
6:  20 00 1E FF FF FF FF FF    .......
7:  66 69 67 32 38 20 20 20   fig28
7:  20 80 2A FF FF FF FF FF    .*.....

8:  66 69 67 32
```

Figure 30

CHAPTER 8: Structure of Disk Files

This chapter explains how PC-8001's **DISK BASIC** formats its files. The information presented in this chapter is not necessary for operation of the PC-8031.

8.1 Storing Data on Disks

Locations within a computer's main storage are assigned addresses and a given location is accessed by specifying its address. In the same manner, information is stored on disks according to an addressing system which uses units of "tracks" and "sectors". Figure 31 shows the general format of a floppy disk. The smallest access unit for floppy disks is the sector. Since the PC-8031 uses double density recording, a sector consists of 256 bytes. As shown in Figure 31, tracks are numbered from 0 to 34, and sectors are numbered from 1 to 16.

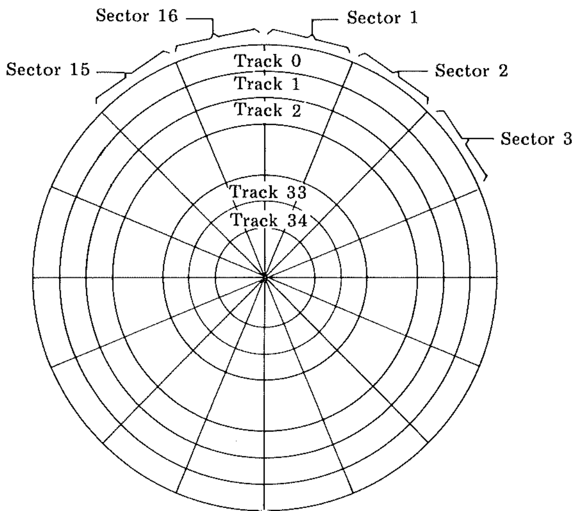


Figure 31

8.2 Extents, FAT, and Directory

When using DISK BASIC, the user needn't be aware of the basic formatting of the disk. As explained in Chapter 3, all disk accesses in DISK BASIC are made in terms of file names. The conversion of the file name into the corresponding track number and sector number is performed by the BASIC interpreter.

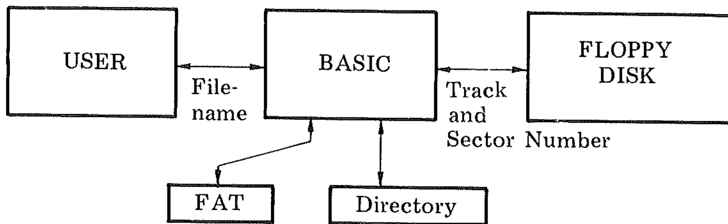


Figure 32

The relation between different parts of the disk system is shown in Figure 32. DISK BASIC maintains two tables which it uses to manage disk files. One table, "FAT", has already been briefly mentioned in Chapter 3. This was a rather loose explanation, and mentioned some of the functions of the second table, the "directory"; although, the term "directory" was not used at that time. Actually, these are two distinct tables.

The directory contains a list of the filenames of all files on the disk, their attributes, and location.

FAT (File Allocation Table) is a list of all locations on the disk, along with an indication to show whether each is being used or not.

The minimum access unit for the floppy disk drive unit is the sector; however, PC-8001's DISK BASIC manages its files in units of "extents".

One extent contains eight sectors. Extent 0 consists of sectors 1 to 8 in track 0, extent 1 consists of sectors 9 to 16 in track 0, and the final extent on the disk, extent 69,

is made up of sectors 9 to 16 of track 34.

8.3 Track Allocation

The allocation of disk areas by the PC-8001, PC-8031 disk system is shown in the next two tables.

(1) System Disks (Disks including DISK BASIC)

Track	Sector	Contents
0	1	IPL
0	2-16	DISK code
1-2	all	
3-17	all	User area
18	1-12	Directory
18	13	ID
18	14-16	FAT
19-34	all	User area

(2) Data Disk (Disks not including DISK code)

Track	Sector	Contents
0-17	all	User area
18-34	all	(same as system disks)

8.4 Directory

Each entry in the directory consists of 16 bytes. The format of directory table entries is given in the next table.

Byte	Contents
0-5	Filename
6-8	Extended identifier
9	Attribute
10	First extent of file
11-15	Unused

The meanings of the codes used to specify attributes are given in the next table.

Code (hex.)	Meaning
0	ASCII format
10	ASCII format write protected
20	ASCII format with read after write
80	Non-ASCII format
90	Non-ASCII format write protected
A0	Non-ASCII format with read after write

When the first byte of a directory entry is FF (hex), it indicates that the entry is not being used. When the first byte is 00 (hex), it indicates that the entry corresponds to a file that has been **KILL**ed.

8.5 FAT

The **FAT** table contains the present status of each extent. The **FAT** table is stored in sectors 14 to 16 of track 18. When the **MOUNT** command is executed, this table is copied into main storage. Each of the sectors 14 to 16 contain exactly the same information, and bytes 0 to 69 of each sector correspond to extents 0 to 69. The meanings of the codes used in the **FAT** table are given in the next table.

Code (hex.)	Meaning
0-4F	Extent is being used, and is one of a chain of extents, the number of the next extent in the chain being indicated by this code.
C1-C8	Extent is being ussd, and is the last in the chain of extents. The lower four bits of this code indicate how many sectors of that extent are actually being used.
FE	The extent is reserved and cannot be used for files. Such extents indicate those used for **DISK** code, IPL, the directory, and the **FAT** table.
FF	The extent is not being used.

In the above table, the term "a chain" was used. Notice that extents are not necessarily in contiguous locations on the disk. Parts of a file may be scattered around the disk in various locations which are logically chained together by information in the **FAT** table.

8.6 ID

One sector is allocated to the **ID** section; however, in the PC-8000 system's **DISK BASIC**, only the first byte of this sector is used.

This one byte contains the disk's attribute. The values used and their meanings are the same as for the directory.

As explained in the section on the **SET** statement, if a file's attribute is omitted, it is assumed to be the same as that of the disk. To protect all files on a disk, the write protect attribute can be assigned to each file individually, but the same thing can be accomplished by setting the disk's attribute to write protect, i.e., by storing the value 10 (hex) in the first byte of sector 13 of track 18. However, new files cannot be added to such a disk.

There is no particular command for writing the **ID**. The **DSKO$** command may be used for this purpose; however, as explained in Chapter 7, this command should be used with extreme caution. The **ID** should be re-written only in exceptional cases.

CHAPTER 9: Formatting Disks

Before a new disk is used on the PC-8031, it must be formatted. The **FORMAT** statement and the format utility have been provided for this purpose. Two steps are involved in formatting new disks:

STEP 1: Level 1 formatting is performed by the **FORMAT** statement. After level 1 formatting has been performed on a disk, the PC-8031 can read and write data to it. Also, the user can read and write to the disk by using **DSKI$** and **DSKO$**; however, none of the other **DISK BASIC** functions, commands, or statements will execute. The reason for this is that the formatting for the **FAT** table, directory, and **ID** has not yet been carried out.

STEP 2: Level 2 formatting is performed by the format utility included in the PC-8031's system disk. This program initializes the disk's **FAT** table, directory, and **ID**. After this step is performed all **DISK BASIC** features may be used on the formatted disk.

Through improper operation it is possible to ruin a disk's formatting. The most common cause is removing a disk from a drive without executing the **REMOVE** command. In such cases, only level 2 formatting is destroyed, and the disk may be re-formatted by carrying out step 2 above.

WARNING

Re-formatting a disk erases all files on the disk. Since a disk's attribute offers it no protection against formatting operations, the user must be extremely careful not to destroy any important files.

****

CHAPTER 10: Preparing Backup Disks

Data stored on disk may be lost either through operator errors, such as commonly occur with the **REMOVE**, **KILL**, and **DSKO$** features, or through improper operation.

To minimize any losses that might occur from destruction of one of your important files, backup copies should be periodically made and safely stored away.

This chapter describes the "backup" program, which can be used to make copies of important files. The program is recorded in PC-8031's system disk, and its listing is given in Figure 33.

Following is a step-by-step explanation of how to use the backup program.

1. Load and execute the backup program.
2. Place the source disk in drive # 1, and give a carriage return.
3. Place the destination disk in drive # 2, and give a carriage return.
4. The destination disk may be any usable disk and does not necessarily need to be previously formatted. When the prompt "Format disk 2(y/n)?" appears, respond Y if the destination disk has not previously been formatted. The backup program then formats the disk and immediately proceeds with copying. If the disk is already formatted, reply with an "n" and copying will begin immediately.
5. As the disk is being copied, the number of the track being copied will be displayed. These numbers will not necessarily be sequential since only tracks containing actual data are copied. An example run of the backup program is given in Figure 34. When copying the system disk, all system programs including the disk code are copied. Thus, the backup program provides a convenient way of making new system disks.

As you may notice, none of the disk I/O commands we have discussed are used in the backup program. The only **DISK BASIC** feature used in the **DSKI$** command, which is used to read the **FAT** table. Other I/O operations are carried out by direct control of the PC-8031.

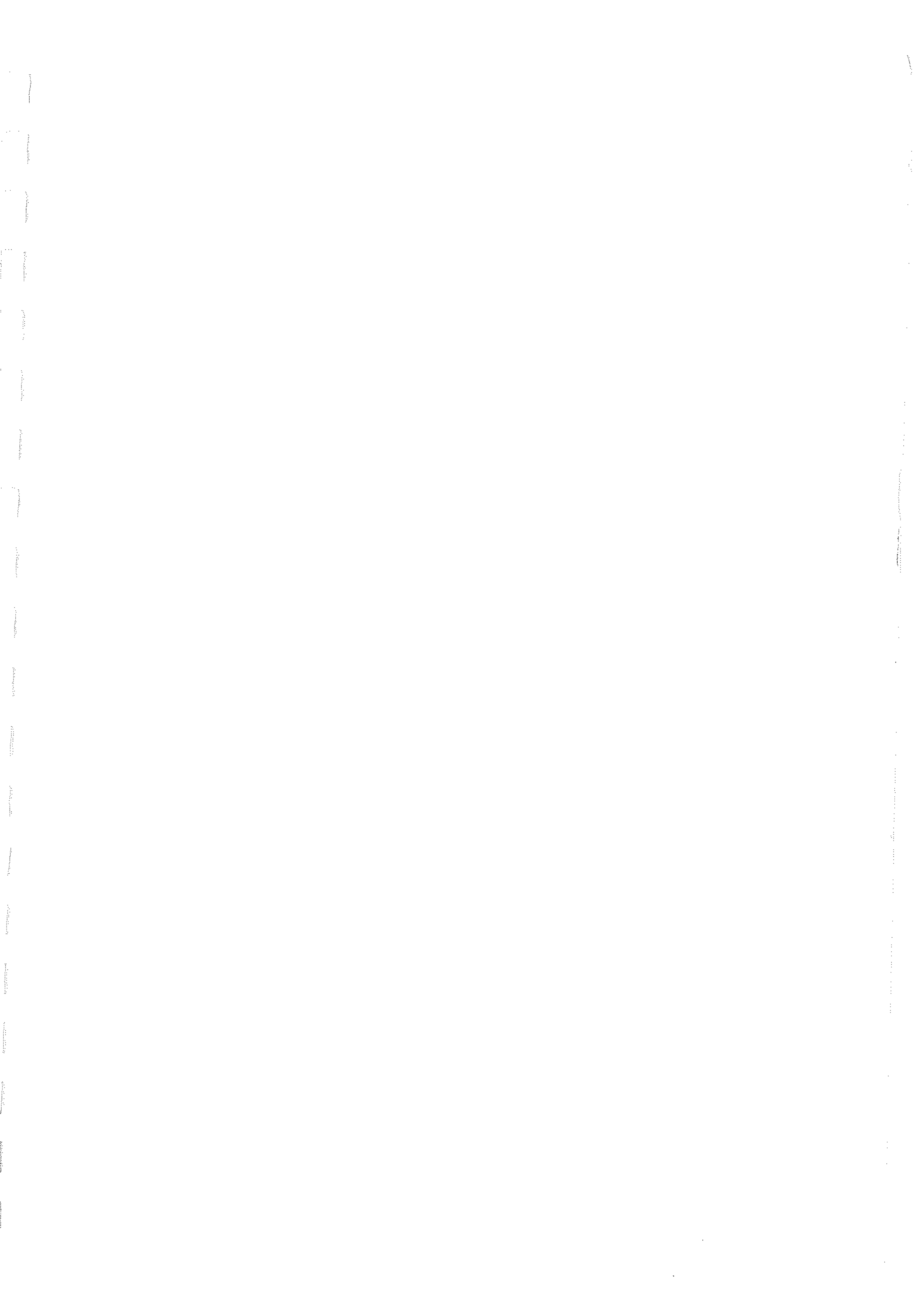
The PC-8031 is an intelligent disk drive, and is controlled by commands sent from the PC-8001. These commands are sent through a parallel interface (μPD8255) located in either the PC-8033 or the expansion unit. The backup program directly accesses the μPD8255 and sends the correct sequence of commands to the drive unit.

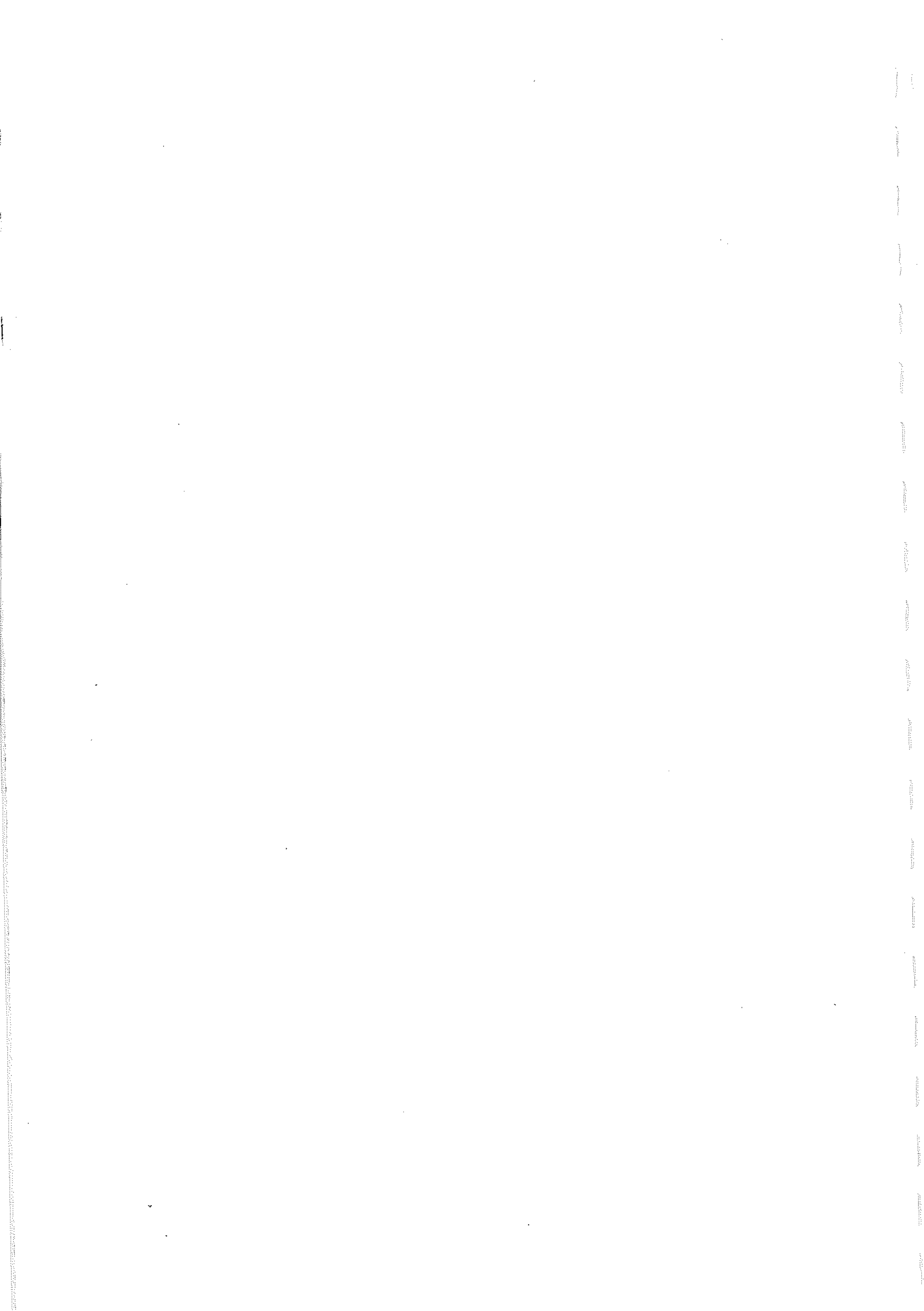
```
10 '
20 ' back up a disk
30 '
35 '
40 ' copyright 1979,1980 by NEC
50 '
55 CONSOLE ,,0
56 COLOR 0
60 DIM PARA (8)
70 DATA 4,16,0,0,1,1,0,1
80 FOR J=1 TO 8:READ PARA(J):NEXT J
90 PA=&HFC:PB=&HFD:PC=&HFE:CW=&HFF
100 PRINT "Back up a disk
110 REMOVE
115 PRINT CHR$(12);"   back up a disk
120 PRINT
130 PRINT "Mount ";:COLOR 4:PRINT"master";:COLOR 0:PRINT " disk on drive ";:COLO
R 4:PRINT"1";:COLOR 0:PRINT",then hit return"
135 LINEINPUT A$
140 PRINT "Mount ";:COLOR 4:PRINT "new";:COLOR 0:PRINT " disk on drive ";:COLOR
4:PRINT"2";:COLOR 0:PRINT",then hit return"
145 LINEINPUT A$
150 INPUT "Format disk 2 (y/n)";A$
160 IF A$="y" THEN 170 ELSE IF A$="n" THEN 190 ELSE BEEP:GOTO 150
170 PRINT "Formatting disk 2"
180 FORMAT 2
190 FAT$=DSKI$(1,18,14)
200 FP=1
210 FOR TRACK=0 TO 34
220 A=ASC(MID$(FAT$,FP,1)) AND ASC(MID$(FAT$,FP+1,1)):FP=FP+2
230 IF A=255 THEN 310
240 PRINT "Copying track";TRACK
250 PARA(4)=TRACK:PARA(7)=TRACK
260 OUT CW,15
270 FOR J=1 TO 8
280 A=PARA(J):GOSUB 340
290 NEXT J
300 GOSUB 410:IF ER THEN PRINT "I/O fault":PRINT "Aborted.":END
310 NEXT TRACK
320 PRINT "Complete"
330 END
340 IF (INP(PC) AND 2)=0 THEN 340
350 OUT CW,14
360 OUT PB,A:OUT CW,9
370 IF (INP(PC) AND 4)=0 THEN 370
380 OUT CW,8
390 IF INP(PC) AND 4 THEN 390
400 RETURN
410 A=6:OUT CW,15:GOSUB 340
420 OUT CW,11
430 IF (INP(PC) AND 1)=0 THEN 430
440 OUT CW,10
450 A=INP(PA)
460 OUT CW,13
470 IF INP(PC) AND 1 THEN 470
480 OUT CW,12
490 ER=A AND 1
500 RETURN
```

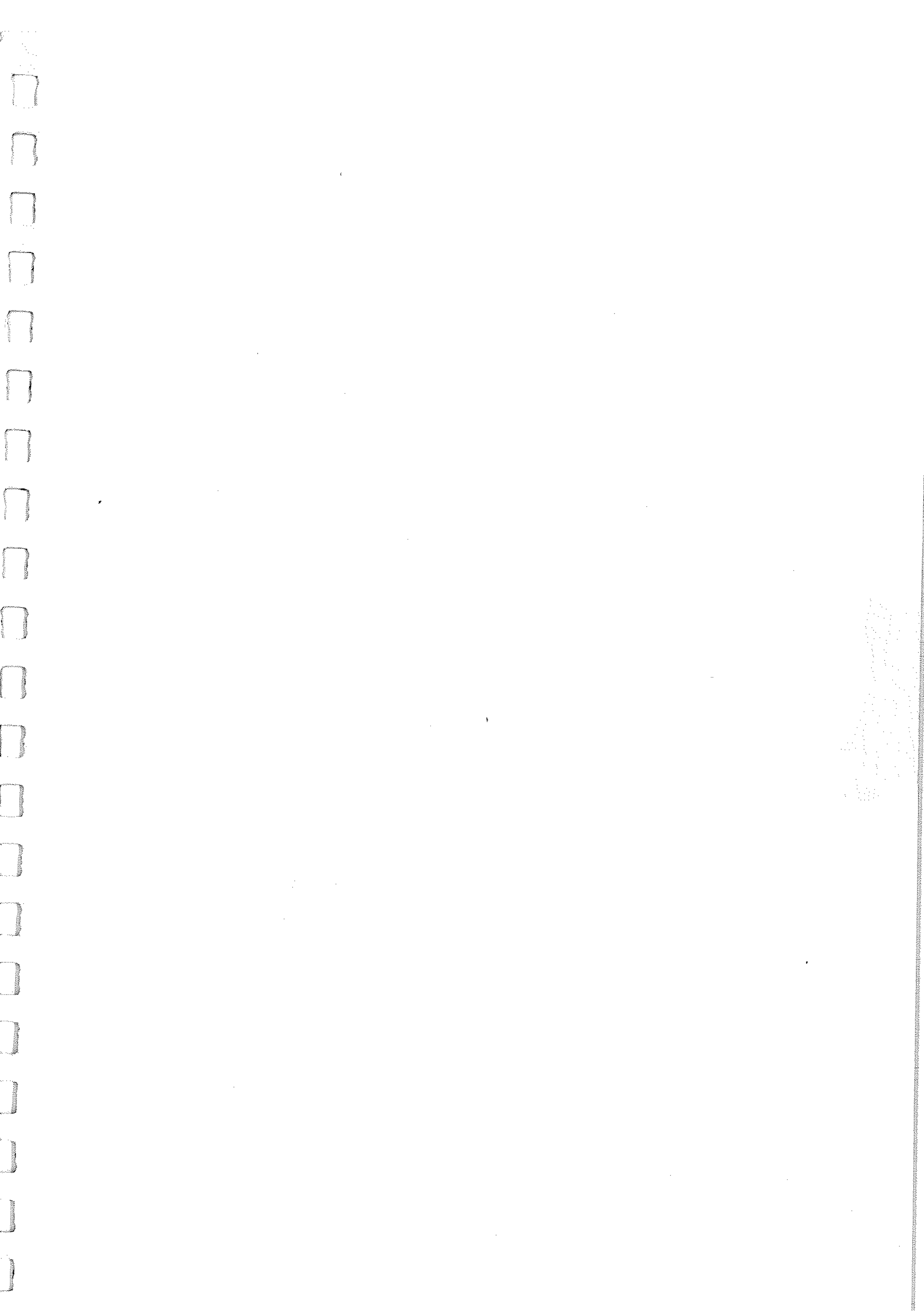
Figure 33

```
   back up a disk

Mount master disk on drive 1,then hit return
Mount new disk on drive 2,then hit return
Format disk 2 (y/n)? y
Formatting disk 2

Copying track 0

Copying track 1
Copying track 2
Copying track 17
Copying track 18
Complete
Ok
```

Figure 34







NEC

Printed in Japan