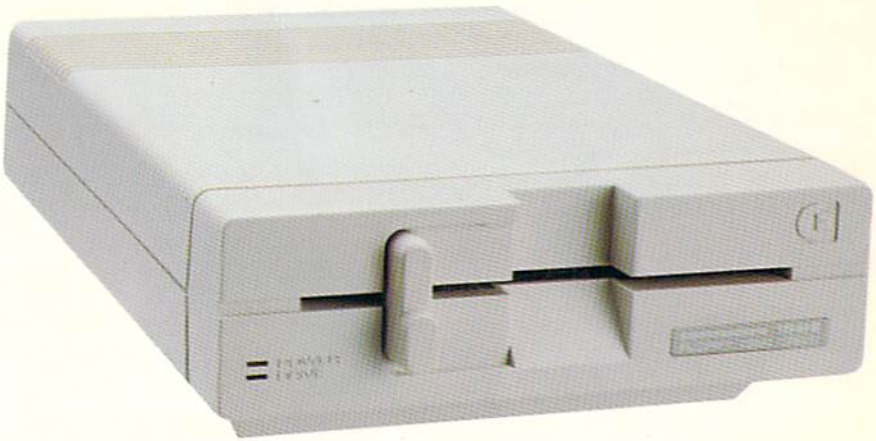
COMMODORE 1541-II™

DISK DRIVE

user's guide



COMMODORE 1541-II DISK DRIVE USER'S GUIDE

This manual contains copyrighted and proprietary information. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Commodore Electronics Limited.

USER'S MANUAL STATEMENT

WARNING: This equipment has been certified to comply with the limits for a Class B computing device, pursuant to subpart J of Part 15 of the Federal Communications Commission's rules, which are designed to provide reasonable protection against radio and television interference in a residential installation. If not installed properly, in strict accordance with the manufacturer's instructions, it may cause such interference. If you suspect interference, you can test this equipment by turning it off and on. If this equipment does cause interference, correct it by doing any of the following.

- Reorient the receiving antenna or AC plug.
- Change the relative positions of the computer and the receiver.
- Plug the computer into a different outlet so the computer and receiver are on different circuits.

CAUTION: Only peripherals with shield-grounded cables (computer input-output devices, terminals, printers, etc.), certified to comply with Class B limits, can be attached to this computer. Operation with non-certified peripherals is likely to result in communications interference.

Your house AC wall receptacle must be a three-pronged type (AC ground). If not, contact an electrician to install the proper receptacle. If a multi-connector box is used to connect the computer and peripherals to AC, the ground must be common to all units.

If necessary, consult your Commodore dealer or an experienced radio-television technician for additional suggestions. You may find the following FCC booklet helpful: "How to Identify and Resolve Radio-TV Interference Problems." The booklet is available form the U.S. Government Printing Office, Washington, D.C. 20402, stock no. 004-000-00345-4.

COMMODORE 1541-II DISK DRIVE USER'S GUIDE

commodore COMPUTERS

The information in this manual has been reviewed and is believed to be entirely reliable. No responsibility, however, is assumed for inaccuracies. The material in this manual is for information purposes only, and may be changed without notice.

© Commodore Electronics Ltd., 1986

"All rights reserved."

TABLE OF CONTENTS

Introduction 1
The advantages of a disk drive 1
Features of the 1541 1
How to use this book 2
Basic Operating Steps 2
Specifications of the 1541 Disk Drive 4

PART ONE: GUIDE TO OPERATION

Chapter 1: Disk Drive 5
Unpacking 5
Empty the drive 6
Connecting the cables 6
Turning on the power 7
Troubleshooting guide 9
Simple maintenance tips 11

Chapter 2: Diskettes 12
What is a diskette? 12
Safety rules for diskette care 13
Inserting a diskette 13
Safety rules for removing diskettes 14
Loading a packaged program 14
How to prepare a new diskette 15
Reusing an old diskette 16
Organizing a diskette library 17
Backups 17

Chapter 3: Directories 18
What is a directory? 18
Viewing a directory 18
What a directory shows 18
Watch out for splat files 20
Printing a directory 21
Reading a directory as a file 21
Another way to be selective 21
Pattern matching and wild cards 22

PART TWO: GUIDE TO ADVANCED OPERATION AND PROGRAMMING

Chapter 4: Commands 25
Command Channel 25
Reading the Error Channel 26
Housekeeping hints 28
Saving programs 29
Save with replace 30
Verifying programs 30
Erasing programs 31
Scratch for advanced users 33
Renaming programs 34
Renaming and scratching troublesome programs 36
Copying programs 37
Validating the diskette 38
Initializing 40

Chapter 5: Sequential Data Files 42
The concept of files 42
Opening a sequential file 42
Adding to a sequential file 45
Writing file data: Print# 46
Closing a file 48
Reading file data using INPUT# 49
More about INPUT (advanced) 50
Numeric Data Storage on Diskette 51
Reading File Data: Using GET# 52
Demonstration of Sequential Files 54

Chapter 6: Relative Data Files 55
The value of relative access 55
Files, Records, and Fields 55
File limits 56
Creating a relative file 56
Using relative files: Record# 57
Completing relative file creation 59
Expanding a relative file 60
Writing relative file data 61
Designing a relative record 61
Writing the record 62
Reading a relative record 63
The value of index files 64

Chapter 7: Direct Access Commands 65
A tool for advanced users 65
Diskette organization 65
Opening a data channel 65
Block-Read 66
Block-Write 67
The original commands 68
The buffer pointer 69
Allocating blocks 70
Freeing blocks 71
Using random files (advanced) 72

Chapter 8: Internal Disk Commands 73
1541 Memory Map 73
Memory Read 74
Memory Write 75
Memory Execute 77
Block Execute 77
User commands 78

Chapter 9: Machine Language Programs 80
Disk-related kernal subroutines 80

Appendices
A. Changing the Device Number 81
B. Error Messages 83
C. Diskette Formats 87
D. Disk Command Quick Reference Chart 92
E. Test/Demo Diskette 93

List of Figures
1. Front Panel 5
2. Back Panel 6
3. Floppy Disk Hookup 8
4. Position for Diskette Insertion 12

INTRODUCTION

The 1541 disk drive greatly increases the speed, storage capacity, flexibility and reliability of your Commodore computer. As you use the 1541 disk drive, you will appreciate its superiority to the cassette recorder you may have used before and to disk drives offered for other brands of computers.

The 1541-II disk drive is fully compatible with the commodore 1541 disk drive, therefore we may often omit the "-II" suffix throughout the remainder of this manual.

THE ADVANTAGES OF A DISK DRIVE

- Speed
  If you have used a cassette recorder for data storage, you probably know it can take up to an hour just to search one long cassette tape looking for a specific program. With the 1541 disk drive, a list of all the programs on a diskette appears on your screen in seconds. The speed of program loading is also greatly improved. It takes the 1541 only a minute to load a large program that would take a half-hour to load from tape.

- Reliability
  Reliability is another reason for choosing a disk drive. It is all too common for a cassette user to accidentally erase a valuable program by saving a new program on top of the old one, without realizing it. The 1541 disk drive automatically verifies everything it records.

- Direct File Access
  A third advantage of a disk drive is the ability to use relative files (discussed in Chapter 6). On a diskette, any part of a relative file can be accessed and altered separately, without affecting the rest of the file.

Overall, using a disk drive makes for easier and more powerful computing.

FEATURES OF THE 1541

The 1541 is one of the most affordable disk drives on the market. Compared to competitors, the 1541 has high capacity, and even higher intelligence. It is one of the most cost-effective disk drives available. Most home and personal computers that use a disk take at least 10K of RAM memory from the computer to hold a disk operating system (known as a DOS.) This large program must be in memory the whole time the disk is being used, and much of it must also be kept on every diskette.

The Commodore 1541 works differently and more effectively. It contains its own built-in microcomputer to control its various operations, along with enough ROM and RAM memory to operate without any help from the computer. Commodore's DOS "lives" entirely inside the disk drive, and does not require any internal memory in the computer to do its work, nor does it have to be loaded before use like DOS on other computers. It is so independent that once it begins working on a command, it will complete it while the computer goes on to some other task, effectively allowing you to do two things at once.

Another key advantage of the Commodore 1541 over disk drives for other computers is its dynamic allocation of disk space. Many other disk drives make you think about every program you save. Where can I store it on this diskette, and should I pack the disk first? (Packing is the process of moving all the leftover work areas to the end of the diskette's storage space.) All this is handled automatically on Commodore disk drives. The 1541 disk drive always knows where the next program will go, and automatically fits it into the best available spot.

Diskettes created on the 1541 may be read by several other Commodore disk drives, including the former 1540, 2040, and 4040, and the 2031. It is usually possible, though not recommended, to write data to any one of these drives from any of the others.

The 1541 communicates with the computer and other devices over a cable and interface known as the Commodore serial bus. It is patterned after the IEEE-488 bus used on Commodore's PET and CBM models, except that the serial version only uses one wire for data. The two serial ports on the 1541 allow several devices to be connected together at once, each plugged into the next in "daisy chain" fashion. Up to 4 disk drives and 2 printers can be connected this way.

HOW TO USE THIS BOOK

This book is divided into two main parts. The first part gives you the information you need to use the 1541 effectively, even if you know little or nothing about programming. This part of the book tells you how to set up the system, how to prepare diskettes for use, how to read a directory, and how to load programs. Part two of the book is for advanced users and those who would like to become advanced users. This part provides more advanced commands, tells about the different files the 1541 uses, and how to manage them, as well as giving a few hints for machine language programmers.

Both beginning and advanced users will find valuable information in the appendices—a quick reference list of disk commands, a list of disk error messages and what they mean, a glossary of words used in this manual, how to use two or more disk drives at once, and explanations of some programs on the Test/Demo diskette packed with your 1541.

Since owners of four different Commodore computers use the 1541, we have separated several explanations into two versions, depending on which Basic your computer uses. If you have a VIC 20 or Commodore 64, please read the pages marked Basic 2. Those with the Commodore 16 or the Plus/4 should read pages marked Basic 3.5. For many commands, there will be an added page or two of further comments and advanced techniques. Feel free to skip anything you don't understand on those pages now, and come back later.

BASIC OPERATING STEPS

If you're like most people, you're anxious to start using your new disk drive right away. In view of that, we have outlined the basic steps you need to know in order to get started.

Once you've mastered the basic steps however, you will need to refer to the rest of this manual in order to make full use of the 1541's features. In fact, before you begin, you should take a look at the following short sections, which offer precautions on handling the equipment: "Simple maintenance tips," "Safety rules for diskette care," and "Safety rules for removing diskettes."

1. *Unpack, hook-up, and turn on the disk drive.*

There's no shortcut through this part. You'll have to read this section to find out what connects to what, when to turn everything on, and how to empty the drive.

If you run into any problems at this point, refer to the Troubleshooting Guide.

Gently insert a pre-programmed diskette.

For the purpose of demonstration, use the Test/Demo diskette that was included with the disk drive.

If you run into any problems at this point, refer to Chapter 2, "Inserting a Diskette."

3. *Type: LOAD "HOW TO USE",8 (for Basic 2)*
   DLOAD "HOW TO USE" (for Basic 3.5)
   Press: RETURN

HOW TO USE is the name of a particular program on the Text/Demo diskette. To load a different BASIC program, substitute the name of that program inside the quotation marks.

If you want to load a program that isn't written in BASIC language, you must use the Basic 2 command and add the following after the 8 in that command: ,1

If you run into any problems at this point, refer to Chapter 2, the section entitled "Loading a Packaged Program."

4. *After you perform step 3, the following will appear on the screen:*

SEARCHING FOR HOW TO USE
LOADING
READY

At this point, type RUN and press the RETURN key and follow the directions for the program.

SPECIFICATIONS OF THE 1541 DISK DRIVE

STORAGE

Total formatted capacity	174848 bytes per diskette
Maximum Sequential file size	168656 bytes per diskette
Maximum Relative file size	167132 bytes per diskette
Records per file	65535
Files per diskette	144
Tracks per diskette	35
Sectors per track	17–21
Sectors per diskette	683 total 664 free for data
Bytes per sector	256

INTEGRATED CIRCUIT CHIPS USED

1 6502 Used for overall control	microprocessor
2 6522 VIA Used for input and output, and as internal timers	Versatile Interface Adapters
1 6116 RAM Used as 2K of buffers	Random Access Memory
1 16K ROM Contains a 16K Disk Operating System (DOS)	Read-Only Memory

INTERFACE USED

Commodore serial bus with two 6-pin DIN connectors

Device number	selectable from 8-11
Secondary addresses	0-15

PHYSICAL DIMENSIONS

Height	77 mm
Width	184 mm
Depth	256 mm

ELECTRICAL REQUIREMENTS

Three wire-grounded detachable power cable.

Voltage	U.S.	100–120 VAC
	Export	220–240 VAC
Frequency	U.S.	60 HZ
	Export	50 HZ
Power used		25 Watts

MEDIA

Any good quality 5¼ inch diskette may be used (Commodore diskettes are recommended).

PART 1: GUIDE TO OPERATION

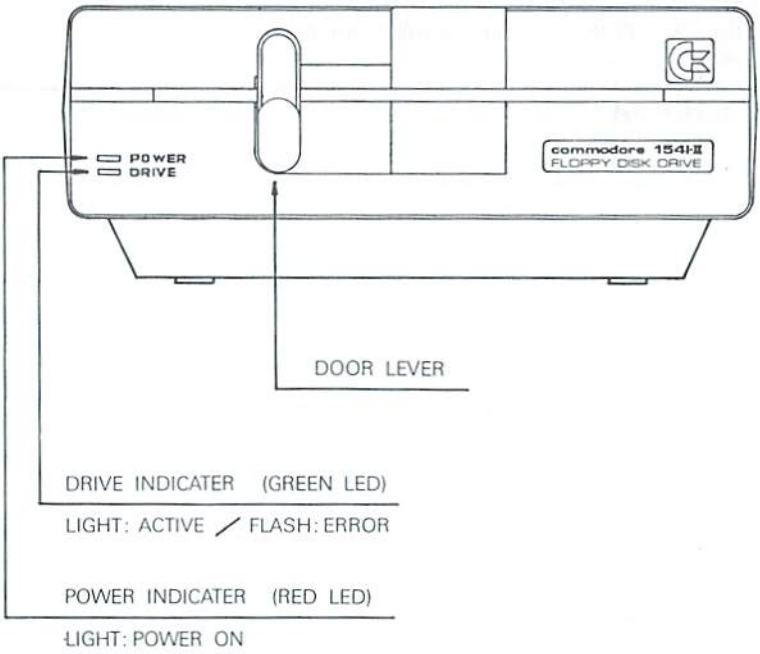
CHAPTER 1
DISK DRIVE

UNPACKING

The first thing you will need to do with your disk drive is unpack it. Inside the carton in which you found this manual, there should also be: a 1541 disk drive, supply power a black cable to connect the disk drive to the computer, a demonstration diskette. and a warranty card to be filled out and returned to Commodore.

Please don't connect anything until you've read the next three pages! It could save you a lot of trouble.

Fig 1. Front Panel



THE PRODUCT DOES NOT NECESSARILY RESEMBLE THE PICTURE INSIDE THE USER'S MANUAL.

EMPTY THE DRIVE

First, it is very important to be sure nothing is inside the disk drive. If you turn the power off or on with a diskette in the drive, you could lose its contents and have to re-record it. Since you wouldn't like having to do that, always check to be sure nothing is inside the drive before turning it off or on.

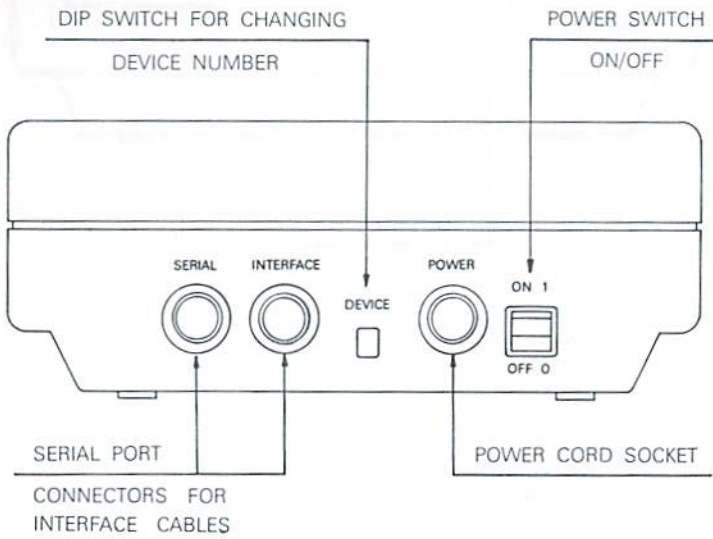
When you first unpack the disk drive, you will find a cardboard shipping spacer inside. Following the instructions below, pull it out as though it were a diskette, but don't throw it away. You will want to put it back inside the slot any time you move or ship the disk drive later.

To check whether the drive is empty (Fig. 1), simply rotate the lever on the front of the disk drive counter-clockwise until it stops, one-quarter turn at most. Then reach inside the long slot the lever covers when it points down, and pull out any diskette you find there.

CONNECTING THE CABLES

With the power supply, the power cable plugs into the back of the disk drive at one end, and into a grounded (3-prong) outlet at the other end. It will only go in one way. Before you plug it in though, make sure that your entire computer system is turned off. The disk drive's on/off switch is in the back, on the right side (when viewed from the back). It is off when the botton half is pushed inward. Leave your whole system off until everything is connected. We will tell you when it is safe to turn it on.

Fig 2. Back Panel



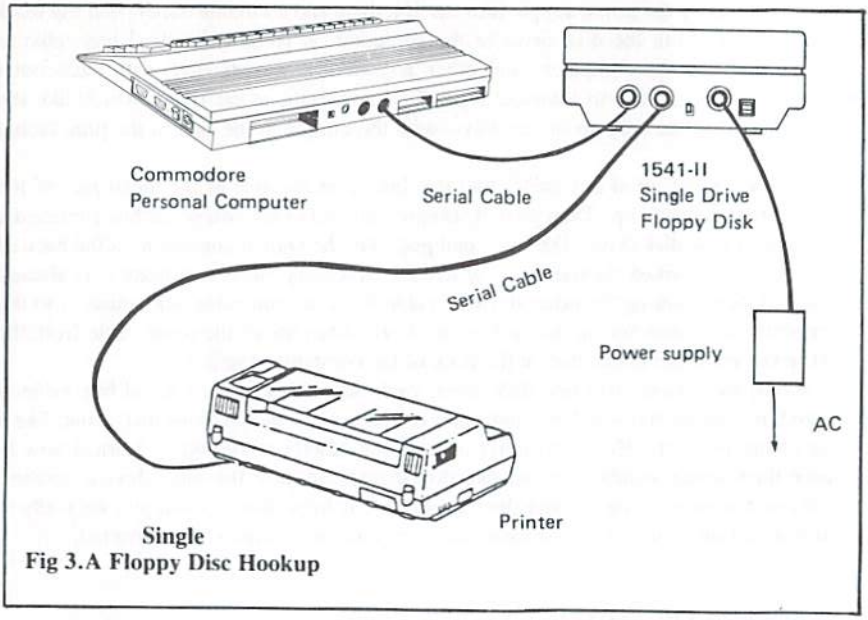
After plugging the power supply into the disk drive and a suitable outlet, find the black cable that goes from the disk drive to the computer. It is called a serial bus cable to describe the way the computer, and other accessories communicate with each other through its wires. It has an identical 6-pin DIN connector at each end, which like the power connector can only go in one way—with the dimple in the side of the plug facing up.

To plug in the serial bus cable, find the dimple on the side of the metal part of the plug and turn that side up. Then push it straight into one of the two serial bus connectors on the back of the disk drive. The other end goes into the similar connector on the back of your computer, marked "serial bus." If another accessory, such as a printer, is already connected there, unplug the other device's cable from the computer, and attach it to the spare serial bus connector on the back of the 1541. Then install the serial cable from the 1541 in the serial bus connector on the back of the computer (Fig 3.A.)

If you have more than one disk drive, each added disk drive's serial bus cable is plugged into the second serial bus connector on the back of the previous disk drive, like a daisy chain (Fig 3.B). However, don't connect the other(s) until you've learned how to change their device numbers, as no two disk drives can have the same device number. We'll cover ways of changing disk device numbers in Appendix A. Until you are ready to read that section, you may find it easier to leave your extra drive(s) unconnected.

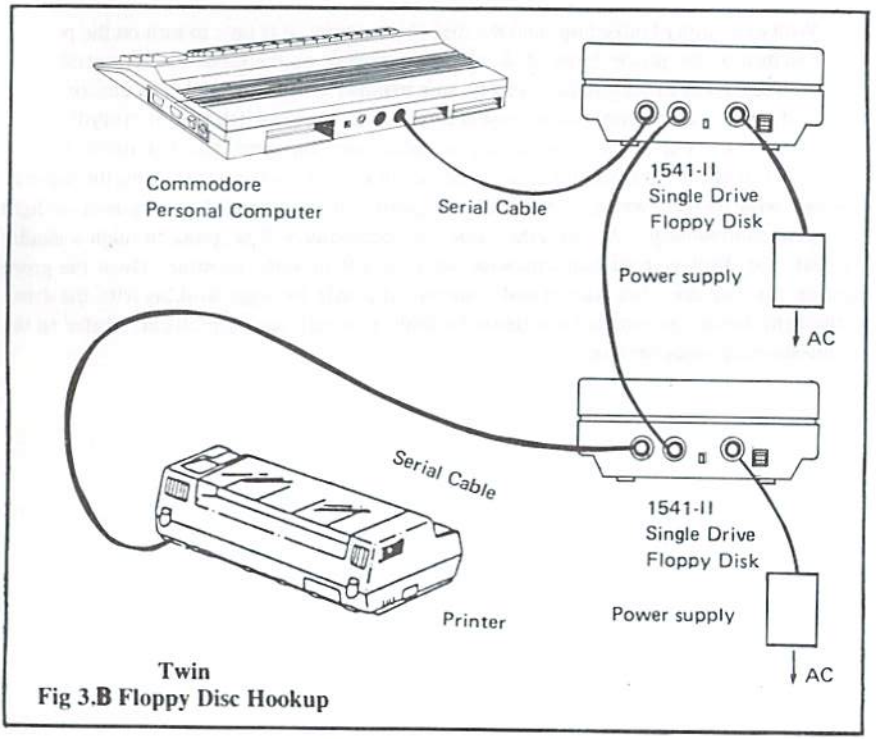
TURNING ON THE POWER

With everything hooked up, and the disk drive empty, it is time to turn on the power. You can turn on the power to the disk and other devices on the serial bus (connected via serial cables) in any order you like. Just be sure to either turn on the power to the computer itself last, or to use a multiple outlet power box with a master switch to turn everything off and on at once. When everything is on, including the computer, the disk drive will go through a self check for a second or so, to be sure it is working correctly. After the drive is satisfied with its own health, it will flash the green light once, and the red power-on light will glow continuously. At the same time, the computer will be going through a similar self-test, and displaying its hello message on your TV or video monitor. Once the green light on the disk drive has flashed and gone out, it is safe to begin working with the drive. If the light doesn't go out, but continues to flash, you may have a problem. Refer to the troubleshooting guide for help.



Single

Fig 3.A Floppy Disc Hookup



Twin

Fig 3.B Floppy Disc Hookup

TROUBLESHOOTING GUIDE

Symptom	Cause	Remedy
Red indicator light on the 1541 not on	Disk drive not turned on	Make sure power switch is in the "on" position
	Power supply not plugged in	Check both cords of power cable to be sure they are fully inserted
	Power off to wall outlet	Replace fuse or reset circuit breaker in house
Green error light on drive flashes continously on power-up, before any disk commands have been given	The disk drive is failing its power-on self-test	Turn the system off for a minute and try again. If it repeats, try again with the serial bus disconnected. If it still repeats, call your dealer. If unplugging the serial cable made a difference, check the cable for proper connection at both ends.
		This can also be caused by some cartridges on the C-64 and always by a 16K cartridge on the VIC 20. Remove the cartridge and power-up the disk drive again to determine where the problem is.

(The principle behind unplugging the serial cable is "divide and conquer." The drive can do its power-on test even when not connected to a computer. If it succeeds that way, then the problem is probably in the cable or the rest of the system, not the 1541.)

TROUBLESHOOTING GUIDE

Symptom	Cause	Remedy
Programs won't load, and computer says "DEVICE NOT PRESENT ERROR."	Serial bus cable not well connected, or disk not turned on.	Be sure serial bus cable is correctly inserted and disk drive is turned on
Programs won't load, but computer and disk drive give no error message.	Another device on the serial bus may be interfering.	Unplug all other devices on the serial bus. If that cures it, plug them in one at a time. The one just added when the trouble repeats is most likely the problem. Also, trying to load a machine language program into BASIC space will cause this problem.
(Such devices may not be turned on properly, or may have conflicting device numbers. Only one device on the bus can have any one device number.)		
Programs won't load and disk error light flashes.	A disk error has occurred.	Check the disk error channel to see why the error occurred. Follow the advice in Appendix B to correct it.
(Be sure to spell program names exactly right, as the disk drive is very particular, even about spaces and punctuation marks, and will not load a program unless you call it exactly the same thing it was called when it was saved on the diskette.)		
Your own programs Load fine, but commercial programs and those from other 1541 owners fail to load.	Either the diskette you are loading is faulty, (some mass-produced diskettes are) or your disk drive is misaligned.	Try another copy of the troublesome programs. If several programs from several sources always fail to load, have your dealer align your disk drive.
Your own programs that used to Load won't any more, but programs saved on newly-formatted diskettes still work.	Older diskettes have been damaged. The disk drive has gone out of alignment.	See the section on safety rules for diskette care. Recopy from backups. Have your dealer align your disk drive.
The disk drive powers up with the activity light blinking.	Hardware failure (RAM, ROM, PCB).	Have your dealer send it out for repair.

SIMPLE MAINTENANCE TIPS

Your 1541 should serve you well for years to come, but there are a few things you can do to avoid costly maintenance.

1. Keep the drive well-ventilated. Like a refrigerator, it needs a few inches of air circulation on all sides to work properly.
2. Use only good quality diskettes. Badly-made diskettes could cause increased wear on the drive's read/write head. If a particular diskette is unusually noisy in use, it is probably causing added wear, and should be replaced.
3. Avoid using programs that "thump" the drive as they load. Many commercial programs, and diskettes that are failing, cause the disk drive to make a bumping or chattering noise as it attempts to read a bad sector. If the diskette can be copied to a fresh diskette, do so immediately. If it is protected by its maker against copying, the thumping is intentional and will have to be endured. Be aware, however, that excessive thumping, especially when the drive is hot, caused some older 1541's to go out of alignment and led to costly repairs. Current 1541's have been redesigned to prevent the problem.
Note: the "Memory-Write" example in Chapter 8 temporarily turns off the bumps.
4. It would be a good idea to have your 1541 checked over about once a year in normal use. Several items are likely to need attention: the felt load pad on the read/write head may be dirty enough to need replacement, the head itself may need a bit of cleaning (with 91% isopropyl alcohol on a cotton swab), the rails along which the head moves may need lubrication (with a special Molybdenum lubricant, NOT oil), and the write protect sensor may need to be dusted to be sure its optical sensor has a clear view. Since most of these chores require special materials or parts, it is best to leave the work to an authorized Commodore service center. If you wish to do the work yourself, ask your dealer to order the 1541 maintenance guide for you (part number 990445), but be aware that home repair of the 1541 will void your warranty.

CHAPTER 2
DISKETTES

WHAT IS A DISKETTE?

Before we actually begin using the drive, let's take a moment to look at the Test/Demo diskette packed with the disk drive. To do this, grasp it by the label, which should be sticking out of the paper jacket. Then pull it out of the jacket which keeps it free of dust and other contaminants. (Save the jacket; the diskette should always be kept in its jacket except when actually in use in the disk drive.) It is often called a floppy diskette, because it is flexible, even though it is not safe to bend diskettes.

A diskette is much like a cassette tape, but in the form of a circle and enclosed within a protective square plastic cover. As on a cassette tape, only a small exposed portion of the magnetic recording surface is sensitive. You may touch the rest of the diskette any time you like, but avoid touching the few small portions that are not covered by the protective cover. Also, never try to remove this cover. Unlike the paper jacket, the plastic diskette cover is intended to remain on permanently.

Next, notice the notch on one side of the diskette (it may be covered by a piece of tape). This notch is called the write protect notch. When it is covered with the opaque tape packed with blank diskettes, the disk drive cannot change the contents of that diskette. Never remove the tape on the Test/Demo diskette.

The label on the top of the diskette says "1541 Test/Demo" on it, and tells you which diskette you are using. Blank diskettes come with extra labels in addition to one applied by the maker of the diskette. Use them to describe your own diskettes.

At least two other parts of the diskette are worth mentioning: The hub and the access slot. The hole in the center is called the hub. A cone-shaped spindle fills it when the drive door is closed, and its edges are clamped. This keeps them from slipping, when the diskette spins at 300 RPM in use.

The oval opening in the diskette opposite the label is called the access slot. It exposes just enough of the diskette's surface for the read/write head and load pad inside the drive to touch a one inch long line from the center to the edge of the diskette's working surface. The bottom side of that slot is where all the information is written as the diskette spins. It is one place your fingers should never touch.

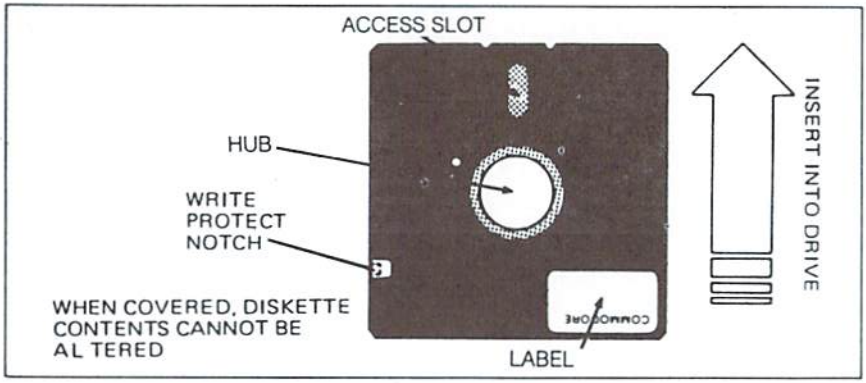


Fig.4. Position for Diskette Insertion

SAFETY RULES FOR DISKETTE CARE

1. Keep the disk drive and its diskettes away from all moisture, dust, smoke, food, and magnets (including the electromagnets in telephones and TV's). Also keep them away from temperatures either too hot or too cold for you to work in for extended periods.
2. When not in the drive, diskettes should be stored upright inside their paper jackets. Do not allow them to become bent or folded. Since the working part of the diskette is on the bottom, never set it down on a table top or other place where dust or moisture might affect it, and be especially careful to keep your fingers away from the openings in the diskette cover.
3. Although some people sell kits intended to "double your diskette's capacity" by cutting an extra write-protect notch into a diskette, it is best not to use the other side of the diskette on the 1541 drive, even if your diskette is labeled "double-sided." Doing so will cause added wear to your diskettes and drive, and may cost you an important program some day.
4. When buying diskettes, you may use any good quality 5¼ inch diskette.
5. Make sure the diskette center hole is more or less centered in its opening before inserting the diskette into the drive. Although the hub assembly will correctly center most any diskette, it would be very difficult to rescue data from a diskette recorded with its hub off-center. One way to help center diskettes is to "tickle" the drive door shut instead of slamming it down. By gently closing it part-way, and then opening a bit and then closing the rest of the way, you give the spindle more chances to center the diskette properly. Another way to ease diskette centering is to buy diskettes that come with reinforced hubs. These hard plastic rings around the hub opening make the diskette hub more rigid, and easier to center properly.

INSERTING A DISKETTE

To insert a diskette in a 1541 (Fig. 1), first open the drive door by rotating the door lever counter-clockwise one quarter turn until it stops, with the lever level with the slot in the front of the drive.

Grasp the diskette by the side opposite the large oval access slot, and hold it with the label up and the write-protect notch to the left. Now insert the diskette by pushing it straight into the slot, the access slot going in first and the label last. Be sure the diskette goes in until it stops naturally, with no part showing outside the drive, but you shouldn't have to force or bend it to get it there.

With the diskette in position, seat it properly for use by twisting the door lever clockwise one-quarter turn, vertically over the slot until it stops. If it does not move easily, stop! You may have put the diskette in the wrong way, or incompletely. If that happens, reposition the diskette until the door lever closes easily.

SAFETY RULES FOR REMOVING DISKETTES

Always remove diskettes before turning a drive off or on. If a diskette were in place and the door closed at power on or off, you could lose part or all of the data on that diskette.

Similarly, do not remove a diskette from its drive when the green drive activity light is on! That light only glows when the drive is actually in use. Removing the diskette with it on may result in your losing information currently being written to the diskette.

LOADING A PACKAGED PROGRAM

To use prepackaged BASIC programs available on diskette, here is the procedure:

After turning on your computer system, carefully insert the preprogrammed diskette as described on the previous page. For purpose of demonstration, use the Test/Demo diskette included with the disk drive. The following commands will load a program from the diskette into the computer:

BASIC 2:

LOAD "program name",device number,relocate flag

Example: LOAD "HOW TO USE",8

BASIC 3.5:

DLOAD "program name",Ddrive #,Udevice number,relocate flag

Example: DLOAD "HOW TO USE"

After each command press the RETURN key

In both cases the exact name of the program wanted is placed between quotation marks. Drive # is optional.

Next is the device number of your disk drive which, unless you change it, is always 8 on the 1541. If you have more than one drive however, you will need to change the device number on any additional drive (see Appendix A for instructions on setting a drive's device number).

Last is the relocate flag. It can have one of two values, 0 and 1. If the relocate flag is 0 or omitted, all Commodore computers that use the 1541 automatically relocate the programs they load to live in the part of computer memory reserved for BASIC programs. If the relocate flag value is 1, auto-relocation is turned off, and the program is loaded into the exact part of memory from which it was originally saved. This feature allows machine-language and other special purpose programs to come into the computer at the correct location for them to operate properly. At this point in your learning, the only thing you need to know about the relocate flag is how to use it. Simply include the 1 at the end of the LOAD command if a particular program doesn't run properly when loaded the usual way.

After you type in the command and press the RETURN key, the following will appear on the screen:

```
SEARCHING FOR "HOW TO USE"
LOADING

READY.
■
```

When the word READY and the flashing cursor reappear on the screen and the green light goes off on the drive, the program named "HOW TO USE" on the Test/Demo diskette has been loaded into the computer. To use it, just type the word RUN and press the RETURN key.

The same Load command will also allow you to load other prepackaged programs from the Test/Demo or other diskettes. Merely substitute the exact program name that you want to use between the quotation marks in the above example, and that will be the program the computer will load (a list of Test/Demo programs is shown in Chapter 3, in the section entitled "What a Directory Shows").

Note: here and in the remainder of the book, we will assume you are in graphic mode, seeing upper case letters and graphic characters when you type. This is the normal condition of all Commodore computers covered by this manual when they are first turned on. If you now see lower-case letters when you type without using the SHIFT key, you are in text mode instead. Press the COMMODORE key ( ), at the lower left corner of your keyboard, together with a SHIFT key to switch to graphic mode.

HOW TO PREPARE A NEW DISKETTE: BASIC 2

A diskette needs a pattern of magnetic grooves in order for the drive's read/write head to find things on it. This pattern is not on your diskettes when you buy them, but adding it to a diskette is simple once you know to do it. Here is the procedure:

```
FORMAT FOR THE DISK NEW COMMAND

OPEN 15,device #,15,"Ndrive #:diskette name,id"
CLOSE 15
```

This Open command will be described more fully in Chapters 4 and 5. For now, just copy it as is, replacing only the parts given in lower case. These include: the device

number of the 1541, normally 8; the drive number, always 0 on the 1541; any desired name for the diskette, up to 16 characters in length, followed by a 2 character diskette ID number. The Close command is often optional; just don't Open that same file again without Closing it the first time.

EXAMPLE:

OPEN 15,8,15,"N0:MY FIRST DISK,01":CLOSE 15

Note: the chattering or thumping noise you hear just after the disk New command begins is entirely normal. The disk must be sure it is at track 1, which it assures by stepping outward 45 times (on a 35 track diskette.) The noise you hear is the head assembly hitting the track 1 bumper after its inevitable arrival.

HOW TO PREPARE A NEW DISKETTE: BASIC 3.5

A diskette needs a pattern of magnetic grooves in order for the drive's read/write head to find things on it. This pattern is not on your diskettes when you buy them, but adding it to a diskette is simple once you know to do it. Here is the procedure:

FORMAT FOR THE HEADER COMMAND

HEADER "diskette name",Iid,Ddrive #,Udevice #

Where "diskette name" is any desired name for the diskette, up to 16 characters in length; "id" is a 2 character diskette ID number; "drive #" is the drive number, 0 if omitted (as it must be on the 1541); and "device #" is the disk's device number, assumed to be 8 if omitted. As described in the next page, "id" is optional if (and only if) the diskette has been previously formatted on a 1541. Also, the ID must be a string literal, not a variable or expression, and may not include Basic reserved words. Thus, ",IFI" cannot be used because If is a Basic keyword, and ",I(A$)" is not allowed because A$ is a variable. ",IA$" is allowed, but the ID number will be the letter "A" plus a dollar sign ($), not the contents of the variable A$. If you need a variable ID number, use the Basic 2 form of the format command.

EXAMPLE:

HEADER "MY FIRST DISK,I01,DO"

Note: the chattering or thumping noise you hear just after the Header command begins is entirely normal. The disk must be sure it is at track 1, which it assures by stepping outward 45 times (on a 35 track diskette). The noise you hear is the head assembly hitting the track 1 bumper after its inevitable arrival.

REUSING AN OLD DISKETTE

After you have once formatted a particular diskette, you can re-format it as though it were brand new at any time, using the above procedures. However, you can also change its name and erase its programs more quickly and easily by omitting the ID number in

your format command. By leaving off the ID number, the format command will finish in a few seconds instead of the usual 90 seconds.

ORGANIZING A DISKETTE LIBRARY

Though you may not believe it now, you will eventually have dozens, if not hundreds of diskettes. You can ease life then by planning now. Assign each diskette a unique ID number when you format it. There are diskette cataloging programs you can buy, that store and alphabetize a list of all your file names, but are of limited value unless your diskette ID numbers are unique.

At least two valid approaches are used in assigning ID numbers. One starts at 00 with the first diskette, and continues upward with each new diskette, through 99, and then onward from AA through ZZ. Another organizes diskettes within small categories, and starts the ID number for each diskette in that category with the same first character, going from 0 to 9 and A to Z with the second character as before. Thus, all "Tax" diskettes could have ID numbers that begin with "T." Either approach works well when followed diligently.

While on this subject, may we suggest you choose names for diskettes on the same basis, so they too will be unique, and descriptive of the files on them.

BACKUPS

When to do a Backup

Although the 1541 is far more reliable than a cassette drive under most circumstances, its diskettes are still relatively fragile, and have a useful life of only a few years in steady use. Therefore, it is important to make regular backups of important programs and files. Make a backup whenever you wouldn't want to redo your current work. Just as you should save your work every half hour or so when writing a new program, so you should also back up the diskette you're using at least daily while you are changing it frequently. In a business, you would make an archival backup every time important information was due to be erased, such as when a new accounting period begins.

How to do a Backup

We have included programs on the Test/Demo diskette that can be used for similar purposes. These programs are described further in Appendix E.

How to Rotate Backups

Once you begin to accumulate backups, you'll want to recycle older ones. One good method is to date each backup. Then retain all backups until the current project is finished. When you are sure the last backup is correct, make another backup of it to file, and move all older backups to a box of diskettes that may be reused.

One other popular approach, suited to projects that never end, is to rotate backups in a chain, wherein there are son backups, father backups, and grandfather backups. Then, when another backup is needed, the grandfather diskette is reused, the father becomes the grandfather, and the son becomes the father.

Whichever approach is used, it is recommended that the newly-made backup become the diskette that is immediately used, and the diskette that is known to be good should be filed away as the backup. That way, if the backup fails, you'll know it immediately, rather than after all the other backups have failed some dark day.

CHAPTER 3
DIRECTORIES

WHAT IS A DIRECTORY?

One of the primary advantages of a disk drive is that it can, with nearly equal ease and speed, access any part of a diskette's surface, and jump quickly from one spot to another. A DATASSETTE™, on the other hand, usually reads a cassette file from the beginning to the end, without skipping around. To see what's on a cassette, it is necessary to look at its entire length, which could take as long as an hour. On a disk drive, by way of contrast, it is a quick and simple matter to view a list of the programs and data files on a diskette. This list is called the directory.

VIEWING THE DIRECTORY: BASIC 2

To view the directory in Basic 2, it is usually necessary to load it, like a program. As when you load other programs, this erases anything already in Basic memory, so be sure to save any work you don't want to lose before loading the directory in Basic 2. (Chapter 4 describes how to Save a program.)

For example, to load the entire directory from disk device 8, type:

```
LOAD"$",8
```

Then, to display the directory on your screen after it loads into computer memory, type LIST. You may slow the listing by pressing the CONTROL key on the VIC 20 and Commodore 64, and halt it entirely by pressing the STOP key.

You can also use this command to limit the directory to desired files by using pattern-matching characters described later in this chapter.

VIEWING THE DIRECTORY: BASIC 3.5

To view the directory in Basic 3.5, simply type the word DIRECTORY on a blank line, and press the RETURN key. Unlike Basic 2's method of loading a directory, this does not erase anything already in Basic memory, so you can safely ask for a directory at almost any time, even from within another program.

Again, you may slow a directory listing on the Commodore 16 and Plus/4 by holding down the COMMODORE key (C=), or halt it entirely by pressing the STOP key. You may also pause it with CONTROL-S (by holding down the CONTROL key while pressing the "S" key), and resume by pressing any other key.

WHAT A DIRECTORY SHOWS

Now let's look at a typical directory on your 1541 Test/Demo Diskette.

```
READY.
 0 "                    " 2A
 14    "HOW TO USE"        PRG
 12    "HOW PART 2"        PRG
 12    "HOW PART 3"        PRG
 4     "VIC-20 WEDGE"      PRG
 1     "C-64 WEDGE"        PRG
 4     "DOS 5.1"           PRG
 9     "PRINTER TEST"      PRG
 4     "DISK ADDR CHANGE"  PRG
 6     "VIEW BAM"          PRG
 4     "CHECK DISK"        PRG
 14    "DISPLAY T&S"       PRG
 9     "PERFORMANCE TEST"  PRG
 5     "SEQ.FILE.DEMO"     PRG
 7     "SD.BACKUP.C16"     PRG
 7     "SD.BACKUP.PLUS4"   PRG
 10    "SD.BACKUP.C64"     PRG
 7     "PRINT.64.UTIL"     PRG
 7     "PRINT.C16.UTIL"    PRG
 7     "PRINT.+4.UTIL"     PRG
 30    "C64 BASIC DEMO"    PRG
 35    "+4 BASIC DEMO"     PRG
 8     "LOAD ADDRESS"      PRG
 7     "UNSCRATCH"         PRG
 5     "HEADER CHANGE"     PRG
 10    "REL.FILE.DEMO"     PRG
 426 BLOCKS FREE.
```

IMPORTANT NOTE: *Your Test/Demo diskette may contain additional programs. Commodore may update the diskette from time to time.*

Starting with the top line, here is what it tells us:

The 0 at the left end tells us that the 1541's single drive is drive 0. If we had gotten this directory from a dual disk drive, it might have said "1" instead.
The next thing on the top line of the directory after the format type is the name of the diskette, enclosed in quotation marks, and printed in reverse field. Just as each program has a name, so does the diskette itself, assigned when the diskette was formatted. The diskette name may be up to 16 characters long, and serves mainly to help you organize your diskette library. By keeping related files together on the same diskette, you'll ease the task of finding the program you want later, when you have dozens or hundreds of diskettes. The two character code to the right of the name is the diskette ID, also created when the diskette was formatted, and equally useful for individualizing diskettes.

The 2A at the right end of the top line tells us that the 1541 uses version 2 of Commodore's DOS (disk operating system), and that it, like most Commodore drives, uses format "A."

The rest of the directory contains one line per program or file, each line supplying three pieces of information about its subject.

At the left end of each line is the size of that line's file in blocks (or sectors) of 256 characters. Four blocks are equivalent to 1K (1024 characters) of RAM (read/write) memory inside the computer.

The middle of each directory line contains the name of the file, enclosed in quotation marks. All characters between the quote marks are part of the name, and must be included when loading or opening that file.

The right portion of each directory line is a three character abbreviation for the file type of that entry. As we will see in later chapters, there are many ways to store information on a diskette, most of which are associated with a distinctive file type.

TYPES OF FILES AVAILABLE
Currently used file types include:

PRG = Program files
SEQ = Sequential data files
REL = Relative data files
USR = User (nearly identical to sequential)
DEL = Deleted (you may never see one of these.)
(**Note:** Direct Access files, also called Random files, do not automatically appear in the directory. They are discussed in Chapter 7.)

After all the directory entries have listed, the directory finishes with a message showing how many blocks of the diskette are still available for use. This number can vary from 664 on a new diskette to 0 on one that is already completely full.

WATCH OUT FOR SPLAT FILES!

One indicator you may occasionally notice on a directory line, after you begin saving programs and files, is an asterisk appearing just before the file type of a file that is 0 blocks long. This indicates the file was not properly closed after it was created, and that it should not be relied upon. These "splat" files (as they are called in England) will normally need to be erased from the diskette and rewritten. However, **do not** use the Scratch command to get rid of them. They can only be safely erased by the Validate and Collect commands. One of these should normally be used whenever a splat file is noticed on a diskette. (All these commands are described in the next chapter.)

There are two exceptions to the above warning: one is that Validate and Collect cannot be used on some diskettes that include direct access (random) files, and the other is that if the information in the splat file was crucial and can't be replaced, there is a way to rescue whatever part of the file was properly written. (This option is also described in the next chapter).

PRINTING A DIRECTORY

To make a permanent copy of a directory, perhaps to fasten to the diskette's outer (paper) envelope, you will need to send the directory to a printer, such as Commodore's MPS 801, 1520 and 1526 serial bus models. To do this, you may need to refer to your printer manual, but briefly the procedure for listing a directory to device 4 is as follows:

```
LOAD"$0",8

OPEN 4,4:CMD 4:LIST

PRINT#4:CLOSE 4
```

Also note that all of the statements that can be combined on one line already have been. Type them in immediate mode to avoid disturbing the directory.

All other options, such as differing device numbers, and selective directories (see next section) can also be specified as usual in the Load command.

> **WARNING:** Be sure to include the PRINT# command after every printer listing. Otherwise, the printer will remain as an unwanted listener on the serial bus, and may disrupt other work. Also, do not abbreviate PRINT# as ?#. Although it will look proper when listed out, it will cause a SYNTAX ERROR in use. The proper abbreviation for PRINT# is pR.

READING A DIRECTORY AS A FILE

If you would like to read a directory from within a program, you may do so. In Basic 3.5, simply include the DIRECTORY command in your Basic program. In Basic 2, however, and optionally in the others, you will have to Open the directory as though it were a data file and read it character by character. See the discussion of Get# in Chapter 5 for more information.

ANOTHER WAY TO BE SELECTIVE

Before discussing the pattern-matching options available for use in several disk commands, let's cover one more that is only usable in a directory. Several different types of files can coexist peacefully on the same diskette. By altering our directory load command, we can create a directory from the files of a single selected type. Thus, we might request a list of all sequential data files (see Chapter 5), one of all the relative data files (see Chapter 6), or one of only program files. To do this, simply add to the end of your selective directory request the equals sign (=) followed by the first letter of the file type you want in your directory. For example, the Basic 2 command:

LOAD"$0:*=S",8

will load a directory of all sequential files, while the Basic 3.5 command:

DIRECTORY,"A*=R"

will display a directory consisting only of relative files beginning with the letter "A".

The possible file types, and their abbreviations for this use are:

P = Program
S = Sequential
R = Relative
U = User
D = Deleted
A = Append
M = Modify

PATTERN MATCHING AND WILD CARDS

Just as cassette users can load programs without giving a full name, disk users can use special pattern matching characters to load a program from a partial name. The same characters can also be used to provide selective directories. The two special characters used in pattern matching are the asterisk (*) and the question mark (?). They act something like a wild card in a game of cards. The difference between the two is that the asterisk makes all characters in and beyond its position wild, while the question mark only makes its own character position wild. Here are some examples, and their results:

LOAD "A*",8 loads the first file on disk that begins with an "A", regardless of what follows. "ARTIST", "ARTERY", and "AZURE" would all qualify, but "BARRY" wouldn't, even though it has an "A" elsewhere in its name.

DLOAD "SM?TH" (Basic 3.5) loads the first program that starts with "SM", ends with "TH", and has one other character between. This would load "SMITH" or "SMYTH", but not "SMYTHE"

OPEN 8,8,2,"R?C*,S,R" We'll study Open in Chapter 5, but the pattern used here means that the selected file will begin with an "R" and have a "C" in the third character of its name.

DIRECTORY,"Q*" (Basic 3.5) will load a directory of files whose names begin with "Q".

LOAD "*",8 and DLOAD "*" are special cases. When an asterisk is used alone as a name, it matches the last file used. If none have been used yet on the current diskette since turning on the drive, using the asterisk alone loads the first program on the diskette.

10 INPUT A$:LOAD A$+"*",8 loads any file whose name starts with the characters entered in A$.

FORMAT FOR PATTERN MATCHING:	EXAMPLES:
"expression*"	"C-64*"
or	
"expression?expression"	"C?64 WEDGE"
or	
"expression?expression*"	"C-64?WED*"

Use any of the above patterns in any of the disk commands whose format includes a pattern. This applies to Load, Dload, Directory, Open, Scratch, and to the source file in the Copy and Rename commands. More than one "?" can appear in the same pattern.

As you might expect, their use in pattern matching means you can't use the asterisk or question mark in a file name when saving or writing a file (see next chapter.)

PART TWO:
GUIDE TO ADVANCED OPERATION AND PROGRAMMING

CHAPTER 4
COMMANDS

COMMAND CHANNEL

Commodore disk drives expect to receive many of their instructions over what is known as a command channel. Although we will not explain the concepts behind it until Chapter 6, we will learn it to use it now, so you can give your 1541 disk the commands it needs to do some essential chores.

To instruct the command channel, we use a Basic Open statement to the disk, with a secondary address of 15. The usual form of this statement is:

```
OPEN 15,8,15
```

The first 15 is a file number, and could be any number from 1 to 255. It is used to match the secondary address (the last number on the line), which is also 15. The middle number is the primary address, better known as the device number, and is normally 8 when talking with the 1541. A second disk drive would usually be 9, a third, 10 and so on.

Once the command channel has been opened, use the Basic Print# command to send information to the disk drive, and Basic's Input# command to receive information back from the disk drive. These two commands are like Basic's Print and Input statements, except that they use the device number specified in the preceding Open statement instead of defaulting to the screen and keyboard respectively.

In Basic 2, you'll use both Print#15 and Input#15 extensively, to send housekeeping commands to the disk and to check its error status. Basic 3.5 has built-in commands for most of these chores. Even so, it will be good for those of you with Basic 3.5 to see how such commands are sent.

Sending a Command via the Command Channel

Here is the way we send the Initialize command to the disk via the command channel.

```
PRINT#15,"I0"
```

This command assumes we have already opened the file 15 to the command channel. "IO" can be replaced with any string expression that is a valid disk command. If file 15 isn't already open, we can combine the Open and the Print# in a single statement:

```
OPEN 15,8,15,"I0"
```

However, this only works for the first disk command given. After that, file 15 is already open, and Opening it again would cause a "FILE OPEN" error. Added commands are sent via Print# instead.

FORMAT FOR SENDING DISK COMMANDS

```
OPEN 15,device #,15,command$
```

or

```
Print#15,command$
```

Examples:

```
OPEN 15,8,15,"VO"
```

or

```
PRINT#15,"VO"
```

where "device #" is the disk's device number, normally 8, and "command$" is any valid string expression. If it is not also a valid disk command, it will result in an error on the disk drive. This is indicated by a flashing error light on the disk drive, and an error message such as "31,SYNTAX ERROR" when the error channel is read as described on the next two pages.

READING THE ERROR CHANNEL: BASIC 2

In Basic 2, there is no simple way to learn what is causing the error light to flash on the disk drive without writing a small program. This, in turn, causes you to lose any program variables already in memory. The reason for this is that the INPUT# command cannot easily be used in immediate mode (that is, without a line number).

You will often need to be able to read the disk error channel, to see why the disk error light is flashing, and thereby turn the error light off again.

Here is a brief program to check for disk errors:

```
10 OPEN 15,8,15
20 INPUT#15,EN,EM$,ET,ES
30 PRINT EN,EM$,ET,ES
40 CLOSE 15
```

This little program reads the error channel into 4 Basic variables, and prints the results on the screen. A message will be displayed whether there is an error or not, but if there was an error, this program will also clear it from disk memory and turn off the error light on the disk drive.

Once the message is on the screen, you can look it up in Appendix B to see what it means, and what to do about it.

READING THE ERROR CHANNEL: BASIC 3.5

In Basic 3.5, it is very easy to learn what is causing the error light to flash on the disk drive, and no need to write a program. Simply type:

```
PRINT DS$
```

or abbreviate it to

```
? DS$
```

either in immediate mode or within a program, and the current error status message of the disk will be displayed on the screen. A message will be displayed whether there is an error or not, but if there was an error, printing its message will also clear it from the disk memory and turn off the error light on the disk drive.

Once the message is on the screen, you can look it up in Appendix B to see what it means, and what to do about it.

ERROR CHECK SUBROUTINE

Since those of you who are writing programs should be checking the error status after each disk command, you may want to include a small subroutine in each program to take care of the error channel. Here is one we use:

Basic 2 version:

```
59980 REM READ ERROR CHANNEL
59990 INPUT#15, EN,EM$,ET,ES
60000 IF EN>1 THEN PRINT
      EN,EM$,ET,ES:STOP
60010 RETURN
```

Basic 3.5 version:

```
59990 REM READ ERROR CHANNEL
60000 IF DS>1 THEN PRINT DS$:STOP

60010 RETURN
```

The Basic 2 version assumes file 15 has already been opened earlier in the program, and that it will be closed somewhere else at the end of the program.

This subroutine reads the error channel and puts the results into the named variables. In the Basic 2 version, they are EN, EM$, ET, and ES, which stand for Error Number, Error Message, Error Track and Error Sector respectively. Of the four, only EM$ has to be a string, and you could choose other variable names, although these have become traditional for this use.

The Basic 3.5 version subroutine uses the reserved variables DS and DS$ already set aside for this purpose. They are updated automatically by Basic whenever they are used. Otherwise, the two versions of the subroutine are equivalent.

Two error numbers are harmless: 0 means everything is OK, and 1 tells how many files were erased by a Scratch command. If the error status is anything else, line 60000 prints the error message and halts the program. After you have repaired the damage, you may then continue the program with Basic's Cont command.

Because this is a subroutine, you access it with the Basic Gosub command, either in immediate mode or from a program. (For example, "200 GOSUB 59990".) The Return statement in line 60010 will jump back to immediate mode or the next statement in your program, whichever is appropriate.

HOUSEKEEPING HINTS

Hint #1: It is best to open file 15 once at the very start of a program, and only close it at the end of the program, after all other files have already been closed. Do this because closing the command channel automatically closes all other disk files. By opening once at the start, the file is open whenever needed for disk commands elsewhere in the program. Closing it at the end makes sure all disk files are properly closed without interrupting any other file commands.

Hint #2: If Basic halts with an error when you have files open, Basic aborts them without closing them properly on the disk. To close them properly on the disk, you must type:

```
OPEN 15,8,15:CLOSE 15
```

This opens the command channel and immediately closes it, along with all other disk files. Failure to close a disk file properly both in Basic and on the disk may result in losing the entire file!

HINT #3: One disk error message is not always an error. Error 73, "CBM DOS 2.6 1541" will appear if you read the disk error channel before sending any disk commands when you turn on your computer. This is a handy way to check which version of DOS you are using. However, if this message appears later, after other disk commands, it means there is a mismatch between the DOS used to format your diskette and the DOS in your drive.

HINT #4: To reset drive, type: OPEN 15,8,15,"UJ" Then wait until the drive activity LED is off and motor goes off, then type: CLOSE 15. This also applies to sending a UI+ or a UI−

SAVING PROGRAMS: BASIC 2

Before you can save a program to diskette, the diskette must be formatted, as described earlier. Saving to diskette is just like saving to cassette, except that the device number of the disk drive is not optional.

FORMAT FOR THE SAVE COMMAND

```
SAVE "drive #:file name",device #
```

where "file name" is any string expression of up to 16 characters, preceded by the drive number (always 0 on the 1541) and a colon, and followed by the device number of the disk, normally 8.

However, it will not work in copying programs that are not in the Basic text area, such as "DOS 5.1" for the Commodore 64. To copy it and similar machine language programs, you will need a machine language monitor program. Its use for this purpose is identical to the monitor save described on the next page under Basic 3.5.

Note: the "0:" at the start of file names is a holdover from the days when all Commodore disks had two drives in the same cabinet. Although the 1541 will normally default to drive 0 (not having a drive 1,) it is best to specify the drive number whenever saving or writing a file. This avoids potential confusion in DOS (the Disk Operating System.)

SAVING PROGRAMS: BASIC 3.5

Before you can save a program to diskette, the diskette must be formatted, as described earlier. Saving to diskette is just like saving to cassette, except that the device number of the disk drive is not optional.

FORMAT FOR THE DSAVE COMMAND

DSAVE "file name",Ddrive #,Udevice #

where "file name" is any string expression of up to 16 characters, optionally followed by the drive number (the "D" parameter, always 0 on the 1541), and the device number of the disk drive (the "U" parameter). If omitted, the drive number defaults to 0, and the device number to 8.

However, it will not work in copying programs that are not in the Basic text area, such as "DOS 5.1" for the Commodore 64. To copy it and similar machine language programs, you will need the .S command of the machine language monitor built into the Commodore 16 and Plus/4. To access a built-in monitor, type MONITOR. To exit a monitor, type X alone on a line.

FORMAT FOR A MONITOR SAVE

S"drive #:file name",device #,starting address,ending address + 1

where "drive #:" is the drive number, 0 on the 1541; "file name" is any valid file name up to 14 characters long (leaving 2 for the drive number and colon); "device #" is a two digit device number, normally 08 on the 1541 (the leading 0 is required); and the addresses to be saved are given in Hexadecimal (base 16,) but without a leading dollar sign ($). Note that the ending address listed must be 1 location beyond the last location to be saved.

EXAMPLE:

Here is the required syntax to save a copy of "DOS 5.1"

S"0:DOS 5.1",08,CC00,CF5A

SAVE WITH REPLACE OPTION

If a file already exists, it can't be saved again because the disk only allows one copy of any given file name per diskette. It is possible to get around this problem using the Rename and Scratch commands described later. However, if all you wish to do is replace a program or data file with a revised version, another command is more convenient. Known as Save-with-replace , or @Save, this option tells the disk to replace any file it finds in the directory with the same name, substituting the new file for it.

FORMAT FOR SAVE WITH REPLACE: BASIC 2	FORMAT FOR SAVE WITH REPLACE: BASIC 3.5
SAVE "@Drive #:file name", device #	DSAVE "@file name",Ddrive #, Udevice #

where all the parameters are as usual except for adding a leading "at" sign (@.) The "0:" in the Basic 2 version, though a holdover from earlier dual drives is required here.

EXAMPLES:

SAVE"@0:REVISED PROGRAM",8	DSAVE "@REVISED PROGRAM"

The actual procedure is that the new version is saved completely, then the old version is scratched, and its directory entry altered to point to the new version. Because it works this way, there is little, if any, danger that a disaster such as having the power going off midway through the process would destroy both the old and new copies of the file. Nothing happens to the old copy until after the new copy is saved properly.

However, we do offer one caution—do not use @Save on an almost-full diskette. Only use it when you have enough room on the diskette to hold a second complete copy of the program being replaced. Due to the way @Save works, both the old and new versions of the file are on disk simultaneously at one point, as a way of safeguarding against loss of the program. If there is not enough room left on diskette to hold that second copy, only as much of the new version will be saved on the 1541 as there is still room for. After the command completes, a look at a directory will show the new version is present, but doesn't occupy enough blocks to match the copy in memory. Unfortunately, the Verify command (see next section) will not detect this problem, because however much did get saved will have been saved properly.

VERIFYING PROGRAMS

Although not as necessary with a disk drive as with a cassette, Basic's Verify command can be used to make doubly certain that a program file was properly saved to disk. It works much like the Load command, except that it only compares each character in the program against the equivalent character in the computer's memory, instead of actually being copied into memory.

If the disk copy of the program differs even a tiny bit from the copy in memory, "VERIFY ERROR" will be displayed, to tell you that the copies differ. This in itself doesn't mean either copy is bad, but if they were supposed to be identical, one or the other has a problem.

Naturally, there's no point in trying to verify a disk copy of a program after the original is no longer in memory. With nothing to compare to, an apparent error will always be announced, even though the disk copy is always and automatically verified as it is written to the disk.

FORMAT FOR THE VERIFY COMMAND:

VERIFY "drive #:pattern",device #, relocate flag

where "drive #:" is an optional drive number (0 on the 1541,) "pattern" is any string expression that evaluates to a file name, with or without pattern-matching characters, and "device #" is the disk device number, normally 8. If the relocate flag is present and equals 1, the file will be verified where originally saved, rather than relocated into the Basic text area.

A useful alternate form of the command is:

VERIFY "*",device #

It verifies the last file used without having to type its name or drive number. However, it won't work properly after save-with-replace, because the last file used was the one deleted, and the drive will try to compare the deleted file to the program in memory. No harm will result, but "VERIFY ERROR" will always be announced. To use verify after @SAVE, include at least part of the file name that is to be verified in the pattern.

One other note about Verify—when you Verify a relocated file, an error will nearly always be announced, due to changes in the link pointers of Basic programs made during relocation. It is best to only verify files saved from the same type of machine, and identical memory size. For example, a Basic program saved from a VIC 20 can't easily be verified using a Commodore 64, even when the program would work fine on both machines (unless the program is re-saved). This shouldn't matter, as the only time you'll be verifying files on machines other than the one which wrote them is when you are comparing two disk files to see if they are the same. This is done by loading one and verifying against the other, and as suggested, can only be done on the same machine and memory size as the one on which the files were first created.

ERASING OLD PROGRAMS: BASIC 2

The Scratch command allows you to erase unwanted files, and free the space they occupied for use by other files. It can be used to erase either a single file, or several files at once via pattern-matching.

FORMAT FOR THE SCRATCH COMMAND:

PRINT#15,"SCRATCH0:pattern"

or abbreviate it as:

PRINT#15,"S0:pattern"

"pattern" can be any file name or combination of characters and wild card characters. As usual, it is assumed that the command channel has already been opened as file 15. Although not absolutely necessary, it is best to include the drive number in Scratch commands.

If you check the error channel after a Scratch command, as described in the prior section, the value for ET (error track) will tell you how many files were scratched. For example, if your diskette contains program files named "TEST", "TRAIN", "TRUCK", and "TAIL", you may scratch all four, along with any other files beginning with the letter "T", by using the command:

PRINT#15,"S0:T*"

Then, to prove they are gone, you can type:

GOSUB 59990

to call the error checking subroutine given earlier in this chapter, and if the four listed were the only files beginning with "T", you will see:

01,FILES SCRATCHED,04,00

READY. ■

The "04" tells you 4 files were scratched.

ERASING OLD PROGRAMS: BASIC 3.5

The Scratch command allows you to erase unwanted programs and files from your diskettes, and free up the space they occupied for use by other files and programs. It can be used to erase either a single file, or several files at once via pattern-matching (described at the end of Chapter 3).

FORMAT FOR THE SCRATCH COMMAND:

SCRATCH "pattern",Ddrive #,Uunit #

"pattern" can be any file name or combination of characters and wild card characters. As usual, "D" stands for drive number, which may only be 0 on the 1541. If the drive number is omitted, 0 is assumed. Likewise, "U" stands for unit (device) number, normally 8. If "U" is omitted, 8 is assumed.

Thanks to the defaults, the usual form of the Scratch command becomes:

```
SCRATCH "pattern"
```

As a precaution, you will be asked:

```
ARE YOU SURE? ■
```

before Basic obeys a Scratch command. If you **are** sure, simply press Y and RETURN. If not, press RETURN alone or type any other answer, and the command will be cancelled.

The number of files that were scratched will be automatically displayed. For example, if your diskette contains program files named "TEST", "TRAIN", "TRUCK", and "TAIL", you may scratch all four, along with any other files beginning with the letter "T", by using the command:

```
SCRATCH "T*"
```

and if the four listed were the only files beginning with "T", you will see:

```
01,FILES SCRATCHED,04,00

READY.■
```

The "04" tells you 4 files were scratched.

SCRATCH (FOR ADVANCED USERS)

Scratch is a powerful command, and should be used with caution, to be sure you only delete the files you really want erased. When using it with a pattern, we suggest you first use the same pattern in a Directory command, to be sure exactly which files will be deleted. That way you'll have no unpleasant surprises when you use the same pattern in the Scratch command.

Recovering from a Scratch

If you accidentally Scratch a file you shouldn't have, there is still a chance of saving it. Like Basic's New command, Scratch doesn't really wipe out a file itself; it merely clears the pointers to it in the diskette directory. If you immediately set the diskette aside and protect it with a write-protect notch, to be sure no one adds any files to the diskette, a skilled user in a nearby Commodore user group may be able to recover your file for you. It will help if you can remember what kind of file it was you scratched (program, sequential, etc.), since that information cannot be directly recovered from what is left of the file.

More about Splats

One other warning—**never** scratch a splat file. These are files that show up in a directory listing with an asterisk (*) just before the file type for an entry. The asterisk (or splat) means that file was never properly closed, and thus there is no valid chain of sector links for the Scratch command to follow in erasing the file.

If you Scratch such a file, odds are you will improperly free up sectors that are still needed by other programs, and cause permanent damage to those other programs later when you add more files to the diskette. If you find a splat file, or if you discover too late that you have scratched such a file, immediately validate the diskette using the Validate command described later in this chapter. If you have added any files to the diskette since scratching the splat file, it is best to immediately copy the entire diskette onto another fresh diskette, but do this with a copy program rather than with a backup program. Otherwise, the same problem will be recreated on the new diskette. When the new copy is done, compare the number of blocks free in its directory to the number free on the original diskette. If the numbers match, no damage has been done. If not, very likely at least one file on the diskette has been corrupted, and all should be immediately checked.

Locked Files

Very occasionally, a diskette will contain a locked file; that is one which cannot be erased with the Scratch command. Such files may be recognized by the "<" character which immediately follows the file type in their directory entry. If you wish to erase a locked file, you will have to use a disk monitor to clear bit 6 of the file-type byte in the directory entry on the diskette. Conversely, to lock a file, you would set bit 6 of the same byte. For more information on how such tricks are done, see Chapter 9 and Appendices D and E.

RENAMING PROGRAMS: BASIC 2

The Rename command allows you to alter the name of a program or other file in the diskette directory. Since only the directory is affected, Rename works very quickly.

FORMAT FOR RENAME COMMAND:

```
PRINT#15,"RENAME0:new name=old name"
```

or it may be abbreviated as:

```
PRINT#15,"R0:new name=old name"
```

where "new name" is the name you want the file to have, and "old name" is the name it has now. "new name" may be any valid file name, up to 16 characters in length. It is assumed you have already opened file 15 to the command channel.

One caution—be sure the file you are renaming has been properly closed before you rename it.

EXAMPLES:

Just before saving a new copy of a "calendar" program, you might type:

PRINT#15,"R0:CALENDAR/BACKUP=CALENDAR"

Or to move a program called "BOOT", currently the first program on a diskette to someplace else in the directory, you might type:

PRINT#15,"R0:TEMP=BOOT"

followed by a Copy command (described later), which turns "TEMP" into a new copy of "BOOT", and finishing with a Scratch command to get rid of the original copy of "BOOT", since renamed to "TEMP" by the command above.

RENAMING PROGRAMS: BASIC 3.5

The Rename command allows you to alter the name of a program or other file in the diskette directory. Since only the directory is affected, Rename works very quickly.

FORMAT FOR RENAME COMMAND:

RENAME "old name" TO "new name",Ddrive #,Udevice #

where "new name" is the name you want the file to have, and "old name" is the name it has now. "new name" may be any string expression that evaluates to a valid file name, up to 16 characters in length. If "D" is left out, drive 0 is assumed. If "U" is absent, device 8 is assumed. One caution—be sure the file you are renaming has been properly closed before you rename it.

EXAMPLES:

Just before saving a new copy of a "calendar" program, you might type:

RENAME "CALENDAR" TO "CALENDAR/BACKUP"

Or to move a program called "BOOT", currently the first program on a diskette to someplace else in the directory, you might type:

RENAME "BOOT" TO "TEMP"

followed by a Copy command (described later), which turns "TEMP" into a new copy of "BOOT", and finishing with a Scratch command to get rid of the original copy of "BOOT", since renamed to "TEMP" by the command above.

RENAMING AND SCRATCHING TROUBLESOME FILES (ADVANCED USERS)

Eventually, you may run across a file which has a crazy filename, such as a comma by itself (","") or one that includes a SHIFTed SPACE. Or perhaps you will find one that includes nonprinting characters. Any of these can be troublesome. Comma files, for instance, are an exception to the rule that no two files can have the same name. Since it shouldn't be possible to make a file whose name is only a comma, the disk never expects you to do it again.

Files with a SHIFT-SPACE in their name can also be troublesome, because the disk interprets the shifted SPACE as signalling the end of the file name, and prints whatever follows after the quotation mark that marks the end of a name in the directory. This technique can also be useful, allowing you to have a long file name, but also make the disk recognize a small part of it as being the same as the whole thing without using pattern-matching characters.

In any case, if you have a troublesome filename, you can use the Chr$() function to specify troublesome characters without typing them directly. This may allow you to build them into a Rename command. If this fails, you may also use the pattern-matching characters in a Scratch command. This gives you a way to specify the name without using the troublesome characters at all, but also means loss of your file.

For example, if you have managed to create a file named ""MOVIES", with an extra quotation mark at the front of the file name, you can rename it to "MOVIES" by using the Chr$() equivalent of a quotation mark in the Rename command:

BASIC 2 FORMAT:

```
PRINT#15,"R0:MOVIES="+CHR$(34)+"MOVIES"
```

BASIC 3.5 FORMAT:

```
RENAME (CHR$(34)+"MOVIES") TO "MOVIES"
```

The CHR$(34) forces a quotation mark into the command string without upsetting Basic. The procedure for a file name that includes a SHIFT-SPACE is similar, but uses CHR$(160).

In cases where even this doesn't work, for example if your diskette contains a comma file, (one named ",") you can get rid of it this way:

BASIC 2 FORMAT:	BASIC 3.5 FORMAT:
`PRINT#15,"S0:?"`	`SCRATCH "?"`

Depending on the exact problem, you may have to be very creative in choosing pattern-matching characters that will affect only the desired file, and may have to rename other files first to keep them from being scratched too.

COPYING PROGRAMS: BASIC 2

The Copy command allows you to make a spare copy of any program or file on a diskette. However, on a single drive like the 1541, the copy must be on the same diskette, which means it must be given a different name from the file copied. It's also used to concatenate up to four sequential data files (combining them by linking one to another, end to end in a chain). Files are linked in the order in which they appear in the command. The source files and other files on the diskette are not changed. Files must be closed before they are copied or concatenated.

FORMAT FOR THE COPY COMMAND

PRINT#15,"COPYdrive #:new file=drive #:old file"

EXAMPLES:

PRINT#15,"COPY0:BACKUP=ORIGINAL"

or abbreviated as

PRINT#15,"Cdrive #:new file=drive #:old file"

PRINT#15,"C0:BACKUP=ORIGINAL"

where "drive #" is the drive number (0 on the 1541,) "new file" is the copy, and "old file" is the original.

FORMAT FOR THE CONCATENATE OPTION

PRINT#15,"Cdrive #:new file=drive #:file 1,drive#:file 2,drive #:file 3,drive #:file 4"

where "drive #" is the drive number for each file. Since it is always 0 on the 1541, the drive number is often omitted.

EXAMPLES:

After renaming a file named "BOOT" to "TEMP" in the last section's example, we can use the Copy command to make a spare copy of the program elsewhere on the diskette, under the original name:

PRINT#15,"C0:BOOT=TEMP"

After creating several small sequential files that each fit easily in memory along with a program we are using, we can use the concatenate option to combine them in a master file, even if the result is too big to fit in memory. (Do be sure it will fit in remaining space on the diskette—it will be as big as the sum of the sizes of the files in it.)

PRINT#15,"C0:A-Z=A-G,H-M,N-Z"

NOTE: Dual drives make fuller use of this command, copying programs from one diskette to another in a single disk unit. To do that on the 1541, refer to Appendix E to find the programs that you need.

COPYING PROGRAMS: BASIC 3.5

The Copy command allows you to make a spare copy of any program or file on a diskette. However, on a single drive like the 1541, the copy must be on the same diskette, which means it must be given a different name from the file copied. The source file and other files on the diskette are not changed. Files must be closed before they are copied. Although the 1541 supports a Concatenate option, Basic 3.5 doesn't have a special command for it. The Basic 2 syntax from the previous page may be used instead.

FORMAT FOR THE COPY COMMAND

COPY Ddrive #,"old file" to Ddrive #,"new file",Udevice #

where "D" is the drive number (always 0 on the 1541,) "new file" is the copy, "old file" is the original, and "U" is the device number. If omitted, the drive number defaults to 0 and the device number (unit) to 8.

EXAMPLES:

After renaming a file named "BOOT" to "TEMP" in the last section's example, we can use the Copy command to make a spare copy of the program elsewhere on the diskette, under the original name:

COPY "TEMP" TO "BOOT"

To copy a file on a second disk drive, we would use:

COPY "ORIGINAL" TO "BACKUP",U9

NOTE: Dual drives make fuller use of this command, copying programs from one diskette to another in a single disk unit. To do that on the 1541, refer to Appendix E to find the programs that you need.

VALIDATING THE DISKETTE: BASIC 2

The Validate command recalculates the Block Availability Map (BAM) of the current diskette, allocating only those sectors still being used by valid, properly-closed files and programs. All other sectors (blocks) are left unallocated and free for re-use, and all improperly-closed files are automatically Scratched. However, this bare description of its workings doesn't indicate either the power or the danger of the Validate command. Its power is in restoring to good health many diskettes whose directories or block availability maps have become muddled. Any time the blocks used by the files on a diskette plus the blocks shown as free don't add up to the 664 available on a fresh diskette, Validate is needed, with one exception below. Similarly, any time a diskette contains an improperly-

closed file (splat file), indicated by an asterisk (*) next to its file type in the directory, that diskette needs to be validated. In fact, but for the one exception below, it is a good idea to validate diskettes whenever you are the least bit concerned about their integrity.

The exception is diskettes containing Direct Access files, as described in Chapter 7. Most direct access (random) files do not allocate their sectors in a way the Validate command can recognize. Thus, using Validate on such a diskette may result in un-allocating all direct access files, with loss of all their contents when other files are added. Unless specifically instructed otherwise, never use Validate on a diskette containing direct access files. (Note: these are **not** the same as the relative files described in Chapter 6. Validate may be used on relative files without difficulty.)

FORMAT FOR THE VALIDATE COMMAND

```
PRINT#15,"VALIDATE0"
```

or abbreviated as

```
PRINT#15,"V0"
```

where "0" is the drive number (always 0 on the 1541.) As usual, it is assumed file 15 has been opened to the command channel on the 1541.

EXAMPLE:

```
PRINT#15,"V0'
```

VALIDATING THE DISKETTE: BASIC 3.5

The Collect command in Basic 3.5 is the same as the Validate command in Basic 2. It recalculates the Block Availability Map (BAM) of the current diskette, allocating only those sectors still being used by valid, properly-closed files and programs. All other sectors (blocks) are left unallocated and free for re-use, and all improperly-closed files are automatically Scratched. However, this bare description of its workings doesn't indicate either the power or the danger of the Collect command. Its power is in restoring to good health many diskettes whose directories or block availability maps have become muddled. Any time the blocks used by the files on a diskette plus the blocks shown as free don't add up to the 664 available on a fresh diskette, Collect is needed (with one exception below.) Similarly, any time a diskette contains an improperly-closed file (splat file), indicated by an asterisk (*) next to its file type in the directory, that diskette needs to be collected. In fact, but for the one exception below, it is a good idea to collect diskettes whenever you are the least bit concerned about their integrity. Just note the number of blocks free in the diskette's directory before and after using Collect, and if the totals differ, there was indeed a problem, and the diskette should probably be copied onto a fresh diskette file by file, using the Copy command described in the previous section, rather than using a backup command or program..

The exception is diskettes containing Direct Access files, as described in Chapter 7. Most direct access (random) files do not allocate their sectors in a way Collect can recognize. Thus, collecting such a diskette may result in un-allocating all direct access

files, with loss of all their contents when other files are added. Unless specifically instructed otherwise, never collect a diskette containing direct access files. (Note: these are **not** the same as the relative files described in Chapter 6. Collect may be used on relative files without difficulty.)

FORMAT FOR THE COLLECT COMMAND

COLLECT Ddrive #,Udevice #

where "D" is the drive number (always 0 on the 1541,) and "U" the device number. As usual, if omitted they default to drive 0 and device 8 respectively.

EXAMPLE:

COLLECT D0

INITIALIZING

One command that should not often be needed on the 1541, but is still of occasional value is Initialize. On the 1541, and nearly all other Commodore drives, this function is performed automatically, whenever a new diskette is inserted. (The optical write-protect switch is used to sense when a diskette is changed.)

The result of an Initialize, whether forced by a command, or done automatically by the disk, is a re-reading of the current diskette's BAM (Block Availability Map) into a disk buffer. This information must, of course, always be correct in order for the disk to store new files properly. However, since the chore is handled automatically, the only time you'd need to use the command is if something happened to make the information in the drive buffers unreliable. Even so, you may use the command for reassurance, as often as you like, so long as you close all your files except for the command channel first.

FORMAT FOR THE INITIALIZE COMMAND	EXAMPLE:
PRINT#15,"INITIALIZEdrive #"	PRINT#15,"INITIALIZE 0"

or it may be abbreviated to

PRINT#15,"Idrive #"	PRINT#15,"I0"

where the command channel is assumed to be opened by file 15, and "drive #" is 0 on the 1541.

One use for Initialize is to keep a cleaning diskette spinning, if you choose to use one. (There is no need to use such kits on any regular basis under normal conditions of cleanliness and care.) Nonetheless, if you are using such a kit, the following short program will keep the diskette spinning long enough for your need:

```
10 OPEN 15,8,15
20 FOR I=1 TO 99
30 : PRINT#15,"I0"
40 NEXT
50 CLOSE 15
```

It uses an Initialize loop to keep the drive motor on for about 20 seconds.

CHAPTER 5
SEQUENTIAL DATA FILES

THE CONCEPT OF FILES

A file on a diskette is just like a file cabinet in your office—an organized place to put things. Nearly everything you put on a diskette goes in one kind of file or another. So far all we've used are program files, but there are others, as we have mentioned. In this chapter we will learn about sequential data files.

As we just suggested, the primary purpose of a data file is to store the contents of program variables, so they won't be lost when the program ends. A sequential data file is one in which the contents of the variables are stored "in sequence," one right after another, just as each link in a chain follows the previous link. You may already be familiar with sequential files from using a DATASSETTE™, because sequential files on diskette are just like the data files used on cassettes. Whether on cassette or diskette, sequential files must be read from beginning to end, without skipping around in the middle.

When sequential files are created, information (data) is transferred byte by byte, through a buffer, onto the magnetic media. Once in the disk drive, program files, sequential data files, and user files all work sequentially. Even the directory acts like a sequential file.

To use sequential files properly, we will learn some more Basic words in the next few pages. Then we'll put them together in a simple but useful program.

Note: Besides sequential data files, two other file types are recorded sequentially on a diskette, and may be considered varying forms of sequential files. They are program files, and user files. When you save a program on a diskette, it is saved in order from beginning to end, just like the information in a sequential data file. The main difference is in the commands you use to access it. User files are even more similar to sequential data files—differing, for most purposes, in name only. User files are almost never used, but like program files, they could be treated as though they were sequential data files and are accessed with the same commands.

For the advanced user, the similarity of the various file types offers the possibility of such advanced tricks as reading a program file into the computer a byte (character) at a time and rewriting it to the diskette in a modified form. The idea of using one program to write another is powerful, and available on the Commodore disk drives.

OPENING A FILE

One of the most powerful tools in Commodore Basic is the Open statement. With it, you may send almost any data almost anywhere, much like a telephone switchboard that can connect any caller to any destination. As you might expect, a command that can do this much is fairly complex. You have already used Open statements regularly in some of your diskette housekeeping commands.

Before we study the format of the Open statement, let's review some of the possible devices in a Commodore computer system:

Device #:	Name:	Used for:
0	Keyboard	Receiving input from the computer operator
1	DATASSETTE™	Sending and receiving information from cassette
2	RS232	Sending and receiving information from a Modem
3	Screen	Sending output to a video display
4	Printer	Sending output to a hard copy printer
8	Disk drive	Sending and receiving information from diskette

Because of the flexibility of the Open statement, it is possible for a single program statement to contact any one of these devices, or even others, depending on the value of a single character in the command. Often an Open statement is the only difference between a program that uses a DATASSETTE™ and one using the 1541. If the character is kept in a variable, the device used can even change each time that part of the program is used, sending data alternately and with equal ease to diskette, cassette, printer and screen.

REMEMBER TO CHECK FOR DISK ERRORS!

In the last chapter we learned how to check for disk errors after disk commands in a program. It is equally important to check for disk errors after using file-handling statements. Failure to detect a disk error before using another file-handling statement could cause loss of data, and failure of the Basic program.

The easiest way to check the disk is to follow all file-handling statements with a Gosub statement to an error check subroutine.

EXAMPLE:

```
840 OPEN 4,8,4,"0:DEGREE DAY DATA,S,W"
850 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

FORMAT FOR THE DISK OPEN STATEMENT:

OPEN file #, device #, channel #,"drive #:file name,file type,direction"

where:

"file #" is an integer (whole number) between 1 and 255. If the file number is greater than 127, a line-feed character is inserted after each carriage return in the file opened. Though this may be helpful in printer files, it will cause severe problems in disk files, and is to be avoided at all costs. Do not open a disk file with a file number greater than 127. After the file is open, all other file commands will refer to it by the number given here. Only one file can use any given file number at a time.

"device #" is the number, or primary address, of the device to be used. This number is an integer in the range 0-31, and is normally 8 on the 1541.

"channel #" is a secondary address, giving further instructions to the selected device about how further commands are to be obeyed. In disk files, the channel number selects a particular channel along which communications for this file can take place. The possible range of disk channel numbers is 0-15, but 0 is reserved for program Loads, 1 for program Saves, and 15 for the disk command channel. Also be sure that no two disk files have the same channel number unless they will never be open at the same time. (One way to do this is to make the channel number for each file the same as its file number.)

"drive #" is the drive number, always 0 on the 1541. Do not omit it, or you will only be able to use two channels at the same time instead of the normal maximum of three. If any pre-existing file of the same name is to be replaced, precede the drive number with the "at" sign (@) to request Open-with-replace.

"file name" is the file name, maximum length 16 characters. Pattern matching characters are allowed in the name when accessing existing files, but not when creating new ones.

"file type" is the file type desired: S = sequential, P = program, U = user, and L = length of a relative file.

"direction" is the type of access desired. There are three possibilities: R = read, W = write, and M = modify. When creating a file, use "W" to write the data to diskette. When viewing a completed file, use "R" to read the data from diskette. Only use the "M" (modify) option as a last ditch way of reading back data from an improperly-closed (Splat) file. (If you try this, check every byte as it is read to be sure the data is still valid, as such files always include some erroneous data, and have no proper end.)

"file type" and **"direction"** don't have to be abbreviated. They can be spelled out in full for clarity in printed listings.

"file #", "device #" and **"channel #"** must be valid numeric constants, variables or expressions. The rest of the command must be a valid string literal, variable or expression.

The maximum number of files that may be open simultaneously is 10, including all files to all devices. The maximum number of sequential disk files that can be open at once is 3 (or 2 if you neglect to include the drive number in your Open statement), plus the command channel.

EXAMPLES OF OPENING SEQUENTIAL FILES:

To create a sequential file of phone numbers, you could use:

```
OPEN 2,8,2,"0:PHONES,SEQUENTIAL,WRITE"
```

or save yourself some typing with:

```
OPEN 2,8,2,"0:PHONES,S,W"
```

On the off-chance we've already got a "PHONES" file on our diskette, we can avoid a "FILE EXISTS" error message by doing an @OPEN

```
OPEN 2,8,2,"@0:PHONES,S,W"
```

Of course, this erases all our old phone numbers, so make sure that any information that may be deleted is of no importance. After writing our phone file, we remove our diskette and turn off the system. Later, to recall the data in the file, we would reopen it with something like

```
OPEN 8,8,8,"0:PHONES,S,R"
```

It doesn't matter whether the file and channel numbers match the ones we used before, but the file name does have to match. However, it is possible to use an abbreviation form of the file name, if there are no other files that would have the same abbreviation:

```
OPEN 10,8,6,"0:PH*,S,R"
```

If we have too many phone numbers, they might not fit in one file. In that case, we might use several similar file names, and let a program choose the correct file.

```
100 INPUT "WHICH PHONE FILE (1-3)";PH
110 IF PH<>1 AND PH<>2 AND PH<>3 THEN 100
120 OPEN 4,8,2,"PHONE" + STR$(PH) + ",S,R"
```

You can omit the drive number on an Open command to read a file. Doing so allows those with dual drives to search both diskettes for the file.

> **Note:** Basic 2 and Basic 3.5 use the same file handling commands and the same direct access commands (chapters 7-8). Unless otherwise noted, you may use the same commands for both throughout the remainder of this book.

ADDING TO A SEQUENTIAL FILE

On Commodore's PET and CBM models, an Append command allows you to reopen an existing sequential file and add more information to the end of it. The same thing can be done another way on the 1541. In place of the "type" and "direction" parameters in your Open statement, substitute ",A" for Append. This will reopen your file, and position the disk head at the end of the existing data in your file, ready to add to it.

FORMAT FOR THE APPEND OPTION

```
OPEN file #,device #,channel #,"drive #:file name,A"
```

where everything is as on the previous page except for the ending "A" replacing the "type" and "direction" parameters.

EXAMPLE:

If you are writing a grading program, it would be convenient to simply tack on each student's new grades to the end of their existing grade files. To add data to the "JOHN PAUL JONES" file, we could type

```
OPEN 1,8,3,"0:JOHN PAUL JONES,A"
```

In this case, DOS will allocate at least one more sector (block) to the file the first time you append to it, even if you only add one character of information. You may also notice that using the Collect or Validate command didn't correct the file size. On the other hand, your data is quite safe, and if the wasted space becomes a problem, you can easily correct it by copying the file to the same diskette or a different one, and scratching the original file. Here's a sequence of commands that will copy such files to the original diskette under the original name, for ease of continued use:

BASIC 2:

```
PRINT#15,"R0:TEMP=JOHN PAUL JONES"
PRINT#15,"C0:JOHN PAUL JONES=TEMP"
PRINT#15,"S0:TEMP"
```

BASIC 3.5:

```
RENAME "JOHN PAUL JONES" TO "TEMP"
COPY "TEMP" TO "JOHN PAUL JONES"
SCRATCH "TEMP"
```

If you are using Basic 2, be sure to open file 15 to the command channel beforehand (i.e., with OPEN 15,8,15) and close it afterwards (i.e., with CLOSE 15).

WRITING FILE DATA: USING PRINT#

After a sequential file has been opened to write (with a type and direction of ",S,W"), we use the Print# command to send data to it for storage on diskette. If you are familiar with Basic's Print statement, you will find Print# works **exactly** the same way, except that the list of items following the command word is sent to a particular file, instead of automatically appearing on the screen. Even the formatting options (punctuation and such) work in much the same way as in Print statements. This means you have to be sure the items sent make sense to the particular file and device used.

For instance, a comma between variables in a Print statement acts as a separator in screen displays, making each successive item appear in the next preset display field (typically at the next column whose number is evenly divisible by 10). If the same comma is included between variables going to a disk file, it will again act as a separator, again inserting extra spaces into the data. This time, however, it is inappropriate, as the extra spaces are simply wasted on the diskette, and may create more problems when reading the file back into the computer. Therefore, you are urged to follow the following format precisely when sending data to a disk file.

FORMAT FOR THE PRINT # STATEMENT

```
PRINT#file #,data list
```

where "file #" is the same file number given in the desired file's current Open statement. During any given access of a particular file, the file number must remain constant because it serves as a shorthand way of relating all other file-handling commands back to the correct Open statement. Given a file number, the computer can look up everything else about a file that matters.

The "data list" is the same as for a Print statement - a list of constants, variables and/or expressions, including numbers, strings or both. However, it is strongly recommended that each Print# statement to disk include only one data item. If you wish to include more items, they must be separated by a carriage return character, not a comma. Semicolons are permitted, but not recorded in the file, and do not result in any added spaces in the file. Use them to separate items in the list that might otherwise be confused, such as a string variable immediately following a numeric variable.

> **Note:** Do not leave a space between PRINT and #, and do not abbreviate the command as ?#. The correct abbreviation for Print# is pR.

EXAMPLES:

To record a few grades for John Paul Jones, using a sequential disk file #1 previously opened for writing, we could use:

```
200 FOR CLASS = 1 TO COURSES
210 : PRINT#1,GRADE$(CLASS)
220 NEXT CLASS
320 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

(assuming your program includes an error check subroutine like the one in the last chapter).

In using Print# there is an exception to the requirement to check for disk errors after every file-handling statement. When using Print#, a single check after an entire set of data has been written will still detect the error, so long as the check is made before any **other** file-handling statement or disk command is used. You may be familiar with Print statements in which several items follow each other:

```
400 PRINT NAME$,STREET$,CITY$
```

To get those same variables onto sequential disk file number 5 instead of the screen, the best approach would be to use three separate Print# statements, as follows:

```
400 PRINT#5,NAME$
410 PRINT#5,STREET$
420 PRINT#5,CITY$
```

However, if you need to combine them, here is a safe way to do it:

```
400 PRINT#5,NAME$;CHR$(13);STREET$;CHR$(13);CITY$
```

CHR$(13) is the carriage return character, and has the same effect as putting the print items in separate lines. If you do this often, some space and time may be saved by previously defining a variable as equal to CHR$(13):

```
10 CR$=CHR$(13) ... 400 PRINT#5,NAME$;CR$;STREET$;CR$;CITY$
```

The basic idea is that a proper sequential disk file write, if redirected to the screen, will display only one data item per line, with each succeeding item on the next line.

CLOSING A FILE WHEN YOU ARE DONE USING IT

After you finish using a data file, it is extremely important that you Close it. During the process of writing a file, data is accumulated in a memory buffer, and only written out to the physical cassette or diskette when the buffer fills.

Working this way, there is almost always a small amount of data in the buffer that has not been written to diskette or cassette yet, and which would simply be lost if the computer system were turned off. Similarly, there are diskette housekeeping matters, such as updating the BAM (Block Availability Map) of sectors used by the current file, which are not performed during the ordinary course of writing a file. This is the reason for having a Close statement. When we know we are done with a file, the Close statement will write the rest of the data buffer out to cassette or diskette, update the BAM, and complete the file's entry in the directory. Always Close a data file when you are done using it! Failure to do so may cause loss of the entire file!

However, do not close the disk command channel until all other files have been Closed. The command channel (described in the last chapter), when used, should be the first file Opened, and the last file Closed in any program. Otherwise, remaining files may be closed automatically. As also described there, this may be used to advantage if a program halts on an error while disk files are open.

FORMAT FOR THE CLOSE STATEMENT

```
CLOSE file #
```

where "file #" is the same file number given in the desired file's current Open statement.

EXAMPLES:

To close the data file #5 used as an example on the previous page, we would use

```
CLOSE 5
```

In Commodore's CBM and PET computers, there is a Dclose statement, that, when used alone, closes all disk files at once. With a bit of planning, the same can be done in Basic 2 and 3.5 via a program loop. Since there is no harm in closing a file that wasn't

open, close every file you even think might be open before ending a program. If for example, we always gave our files numbers between 1 and 10, we could close them all with

```
9950 FOR I=1 TO 10
9960 CLOSE I
9970 GOSUB 59990:REM CHECK FOR DISK ERRORS
9980 NEXT I
```

(assuming your program includes an error check subroutine like the one in Chapter 4)

READING FILE DATA: USING INPUT#

Once information has been written properly to a diskette file, it may be read back into the computer with an Input# statement. Just as the Print# statement is much like the Print statement, Input# is nearly identical to Input, except that the list of items following the command word comes from a particular file instead of the keyboard. Both statements are subject to the same limitations—halting input after a comma or colon, not accepting data items too large to fit in Basic's Input buffer, and not accepting non-numeric data into a numeric variable.

FORMAT FOR THE INPUT# STATEMENT

```
PRINT#file #,variable list
```

where "file #" is the same file number given in the desired file's current Open statement, and "variable list" is one or more valid Basic variable names. If more than one data element is to be input by a particular Input# statement, each variable name must be separated from others by a comma.

EXAMPLES:

To read back in the grades written with the Print# example, use:

```
300 FOR CLASS=1 TO COURSES
310 INPUT#1,GRADE$(CLASS)
320 GOSUB 59990:REM CHECK FOR DISK ERRORS
330 NEXT CLASS
```

(assuming your program includes an error check subroutine like the one on page 27).

To read back in the address data written by another Print# example, it is safest to use:

```
800 INPUT#5,NAME$
810 GOSUB 59990:REM CHECK FOR DISK ERRORS
820 INPUT#5,STREET$
830 GOSUB 59990:REM CHECK FOR DISK ERRORS
840 INPUT#5,CITY$
850 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

but many programs cheat on safety a bit and use

```
800 INPUT#5,NAME$,STREET$,CITY$
810 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

This is done primarily when top speed in the program is essential, and there is little or no risk of reading improper data from the file.

MORE ABOUT INPUT# (FOR ADVANCED USERS)

Troublesome Characters

After you begin using data files regularly, you may encounter two Basic error messages more or less frequently. They are "STRING TOO LONG ERROR" and "FILE DATA ERROR". Both are likely to halt your program at an Input# statement, but may also have been caused by errors in a Print# statement when the file was written.

"STRING TOO LONG" ERRORS

A Basic string may be up to 255 characters long, although the longest string you can enter via a single Input statement is just under 2 lines of text (4 on the VIC 20). This lower limitation is due to the 88 character size of the Input buffer in Commodore's serial bus computers. The same limit applies to Input# statements. If a single data element (string or number) being read from a disk file into an Input# statement contains more than 87 characters, Basic will halt with a "STRING TOO LONG ERROR". To prevent this error, be sure to limit each string to under 88 characters, and separate all file data items with carriage returns (See the next section for a cure once the error has occurred.)

"FILE DATA" ERRORS

The other error message "FILE DATA ERROR" is caused by attempting to read a non-numeric character into a numeric variable. To a computer, a number is the characters 0 through 9, the "+" and "−" signs, the decimal point (.), the SPACE character, and the letter "E" used in scientific notation. If any other character appears in an Input# to a numeric variable, "FILE DATA ERROR" will be displayed and the program will halt. The usual causes of this error are a mismatch between the order in which variables are written to and read from a file, a missing carriage return within a Print# statement that writes more than one data item, or a data item that includes either a comma or a colon without a preceding quotation mark. Once a file data error has occurred, you should correct it by reading the data item into a string variable, and then converting it back to a number with the Basic Val() statement after removing non-numeric characters with the string functions described in your computer users manual.

COMMAS (,) AND COLONS (:)

As suggested before, commas and colons can cause trouble in a file, because they delimit (end) the data element in which they appear and cause any remaining characters in the data element to be read into the next Input# variable. (They have the same effect in an Input statement, causing the common "EXTRA IGNORED" error message.) However, sometimes we really need a comma or colon within a data element, such as a name written as "Last, First". The cure is to precede such data elements with a quotation mark. After a

quotation mark, in either an Input or Input# statement, all other characters except a carriage return or another quotation mark are accepted as part of the current data element.

EXAMPLES:

To force a quotation mark into a data element going to a file, append a CHR$(34) to the start of the data element. For example:

```
PRINT#2,CHR$(34)+"STRASMA, JIM"
```

or

```
PRINT#2,CHR$(34);"STRASMA, JIM"
```

If you do this often, some space and time may be saved by previously defining a variable as equal to CHR$(34) as we did earlier with CHR$(13):

```
20 QT$=CHR$(34)
:..
400 PRINT#5,QT$+NAME$
```

In each case, the added quotation mark will be stripped out of the data by the Input or Input# statement, but the comma or colon will remain safely part of the data.

NUMERIC DATA STORAGE ON DISKETTE

Inside the computer, the space occupied by a numeric variable depends only on its type. Simple numeric variables use 7 bytes (character locations) of memory. Real array variables use 5 bytes per array element, and integer array elements use 2 bytes each. In contrast, when a numeric variable or any type is written to a file, the space it occupies depends entirely on its length, not its type.

Numeric data is written to a file in the form of a string, as if the Str$() function had been performed on it. The first character will be a blank space if the number is positive, and a minus sign (−) if the number is negative. Then comes the number, digit by digit. The last character is a cursor right character.

This format allows the disk data to be read back into a string or numeric variable later. It is, however, somewhat wasteful of disk space, and it can be difficult to anticipate the space required by numbers of unknown length. For this reason, some programs convert all numeric variables into strings before writing them to diskette, and use string functions to remove any unneeded characters in advance. Doing so still allows those data elements to be read back into a numeric variable by Input# later, although file data errors may be avoided by reading all data in as strings, and converting to numbers after the information is inside the computer.

For example, "N$=MID$(STR$(N),2)" will convert a positive number N into a string N$ without the usual leading space for its numeric sign. Then instead of writing PRINT#5,N, you would use PRINT#5,N$.

READING FILE DATA: USING GET#

The Get# statement retrieves data from the disk drive, one character at a time. Like the similar keyboard Get statement in Basic, it only accepts a single character into a specified variable. However, unlike the Get statement, it doesn't just fall through to the next statement if there is no data to be gotten. The primary use of Get# is to retrieve from diskette any data that cannot be read into an Input# statement, either because it is too long to fit in the input buffer or because it includes troublesome characters.

FORMAT FOR THE GET# STATEMENT:

GET#file#,variable list

where "file #" is the same file number given in the desired file's current Open statement, and "variable list" is one or more valid Basic variable names. If more than one data element is to be input by a particular Get# statement, each variable name must be separated from others by a comma.

In practice, you will almost never see a Get or Get# statement containing more than one variable name. If more than one character is needed, a loop is used rather than additional variables. Also as in the Input# statement, it is safer to use string variables when the file to be read might contain a non-numeric character.

Data in a Get# statement comes in byte by byte, including such normally invisible characters as the Carriage Return, and the various cursor controls. All but one will be read properly. The exception is CHR$(0), the ASCII Null character. It is different from an empty string (one of the form A$ = ""), even though empty strings are often referred to as null strings. Unfortunately, in a Get# statement, CHR$(0) is converted into an empty string. The cure is to test for an empty string after a Get#, and replace any that are found with CHR$(0) instead. The first example below illustrates the method.

EXAMPLES:

To read a file that may contain a CHR$(0), such as a machine language program file, we could correct any CHR$(0) bytes with

```
1100 GET#3,G$:IF G$ = "" THEN G$ = CHR$(0)
```

If an overlong string has managed to be recorded in a file, it may be safely read back into the computer with Get#, using a loop such as this

```
3300 B$ = ""
3310 GET#1,A$
3320 IF A$<>CHR$(13) THEN B$ = B$ + A$:GOTO 3310
```

The limit for such a technique is 255 characters. It will ignore CHR$(0), but that may be an advantage in building a text string.

Get# may be especially useful in recovering damaged files, or files with unknown contents. The Basic reserved variable ST (the file STatus variable) can be used to indicate when all of a properly-closed file has been read.

```
500 GET#2,S$
510 SU=ST:REM REMEMBER FILE STATUS
520 PRINT S$;
530 IF SU=0 THEN 500:REM IF THERE'S MORE TO BE REAL
540 IF SU<>64 THEN PRINT "STATUS ERROR: ST =";SU
```

Copying ST into SU is often an unneccessary precaution, but must be done if any other file-handling statement appears between the one which read from the file and the one that loops back to read again. For example, it would be required if line 520 was changed to

```
520 PRINT#1,S$;
```

Otherwise, the file status checked in line 530 would be that of the write file, not the read file.

POSSIBLE VALUES OF THE FILE STATUS VARIABLE "ST", AND THEIR MEANINGS

IF ST =	THEN
0	All is OK
1	Receiving device was not available (time out on talker)
2	Transmitting device was not available (time out on listener)
4	Cassette data file block was too short
8	Cassette data file block was too long
16	Unrecoverable read error from cassette, verify error
32	Cassette checksum error—one or more faulty characters were read
64	End of file reached (EOI detected)
128	Device not present, or end of tape mark found on cassette

DEMONSTRATION OF SEQUENTIAL FILES

Use the following program for your first experiments with sequential files. Comments have been added to help you better understand it.

```
150 CR$=CHR$(13)                    Make a carriage return variable
160 OPEN 15,8,15
170 PRINT CHR$(147):REM CLEAR
    SCREEN
190 PRINT "** WRITE A FILE **"
210 PRINT
220 OPEN 2,8,2,"@0:SEQ FILE,S,W"    Open demo file with replace
230 GOSUB 500                       Check for disk errors
240 PRINT"ENTER A WORD,
    THEN A NUMBER"
250 PRINT"OR 'END,0' TO STOP"
260 PRINT
270 INPUT A$,B                      Accept a string & number
                                    from keyboard
280 PRINT#2,A$;CR$;B                Write them to the disk file
290 GOSUB 500
300 IF A$<>"END" THEN 270           Until finished
310 PRINT
320 CLOSE 2                         Tidy up
340 PRINT "** READ SAME FILE
    BACK **"
360 PRINT
370 OPEN 2,8,2,"0:SEQ FILE,S,R"     Reopen same file for reading
380 GOSUB 500
390 INPUT#2,A$,B                    Read next string & number from file
400 RS=ST                           Remember file status
410 GOSUB 500
420 PRINT A$,B                      Display file contents
430 IF RS=0 THEN 390                until done,
440 IF RS<>64 THEN
    PRINT"STATUS=";RS               unless there's an error
450 CLOSE 2                         Then quit
460 END
480 REM ** ERROR CHECK S/R **       A Basic 3.5-only version could
500 INPUT#15,EN,EM$,ET,ES           replace line 500 with
510 IF EN>0 THEN PRINT              500 IF DS>0 THEN PRINT
    EN,EM$,ET,ES:STOP
                                    DS$:STOP
520 RETURN                          and delete line 510
```

CHAPTER 6
RELATIVE DATA FILES

THE VALUE OF RELATIVE ACCESS

Sequential files are very useful when you're just working with a continuous stream of data — i.e., information that can be read or written all at once. However, sequential files are not useful or desirable in some situations. For example, after writing a large list of mail labels, you wouldn't want to have to re-read the entire list each time you need a person's record. Instead, you need some kind of random access, a way to get to a particular label in your file without having to read through all those preceding it first.

As an example, compare a record turntable with a cassette recorder. You have to listen to a cassette from beginning to end, but a turntable needle can be picked up at any time, and instantly moved to any spot on the record. Your disk drive works like a turntable in that respect. In this chapter we will learn about a type of file that reflects this flexibility.

Actually, two different types of random access files may be used on Commodore disk drives: relative files and random files. Relative files are much more convenient for most data handling operations, but true random access file commands are also available to advanced users, and will be discussed in the next chapter.

FILES, RECORDS, AND FIELDS

When learning about sequential files, we did not worry about the organization of data within a file, so long as the variables used to write the file matched up properly with those which read it back into the computer. But in order for relative access to work, we need a more structured and predictable environment for our data.

The structure we will use is similar to that used in the traditional filing cabinet. In a traditional office, all customer records might be kept in a single file cabinet. Within this file, each customer has a personal record in a file folder with their name on it, that contains everything the office knows about that person. Likewise, within each file folder, there may be many small slips of paper, each containing one bit of information about that customer, such as a home phone number, or the date of the most recent purchase.

In a computerized office, the file cabinet is gone, but the concept of a file containing all the information about a group or topic remains. The file folders are gone too, but the notion of subdividing the file into individual records remains. The slips of paper within the personal records are gone too, replaced by subdivisions within the records, called fields. Each field is large enough to hold one piece of information about one record in the file. Thus, within each file there are many records, and within each record there are typically many fields.

A relative file takes care of organizing the records for you, numbering them from 1 to whatever, by ones, but the fields are up to you to organize. Each record will be of the same size, but the 1541 won't insist that they all be divided the same way. On the other hand, they normally will all be subdivided the same way, and if it can be known in advance exactly where each field starts within each record, there are even fast ways to access a desired field within a record without reading through the other fields. As all of this implies, access speed is a primary reason for putting information into a relative disk file. Some well-written relative file programs are able to find and read the record of one

desired person out of a thousand in under 15 seconds, a feat no sequential file program could match.

FILE LIMITS

One of the nicest aspects of relative files is that all this is done for you without your having to worry at all about exactly where on the diskette's surface a given record will be stored, or whether it will fit properly within the current disk sector, or need to be extended onto the next available sector. DOS takes care of all that for you. All you need to do is specify how long each record is, in bytes, and how many records you will need. DOS will do the rest, and organize things in such a way that it can quickly find any record in the file, as soon as it is given its record number (ordinal position within the file).

The only limit that will concern you, is that each record must be the same size, and the record length you choose must be between 2 and 254 characters. Naturally the entire file also has to fit on your diskette too, which means that the more records you need, the shorter each must be.

CREATING A RELATIVE FILE

When a relative file is to be used for the first time, its Open statement will create the file; after that, the same Open statement will be used to re-open the file for both reading and writing.

FORMAT STATEMENT TO OPEN A RELATIVE FILE:

OPEN file #, device #, channel #, "drive #: file name, L," + CHR$ (record length)

where "file #" is the file number, normally an integer between 1 and 127; "device #" is the device number to be used, normally 8 on the 1541; "channel #" selects a particular channel along which communications for this file can take place, normally between 2 and 14; "drive #" is the drive number, always 0 on the 1541; and "file name" is the file name, with a maximum length of 16 characters. Pattern matching characters are allowed in the name when accessing an existing file, but not when creating a new one. The "record length" is the size of each record within the file in bytes used, including carriage returns, quotation marks and other special characters.

Notes:

1. Do not precede the drive number with the "at" sign (@); there is no reason to replace a relative file.

2. ,L ,"+CHR$(record length) is only required when a relative file is first created, though it may used later, so long as the "record length" is the same as when the file was first created. Since relative files may be read from or written to alternately and with equal ease, there is no need to specify Read or Write mode when opening a relative file.

3. "file #", "device #" and "channel #" must be valid numeric constants, variables or expressions. The rest of the command must be a valid string literal, variable or expression.

4. Only 1 relative file can be open at a time on the 1541, although a sequential file and the command channel may also be open at the same time.

EXAMPLES:

To create or re-open a relative file named "GRADES", of record length 100, use

```
OPEN 2,8,2,"GRADES,L,"+CHR$(100)
```

To re-open an unknown relative file of the user's choice that has already been created, we could use

```
200 INPUT"WHICH FILE";FI$
210 OPEN 5,8,5,FI$
```

USING RELATIVE FILES: RECORD#

When a relative file is opened for the first time, it is not quite ready for use. Both to save time when using the file later, and to assure that the file will work reliably, it is necessary to create several records before closing the file for the first time. At a minimum, enough records to fill more than 2 disk sectors (512 bytes) should be written. In practice, most programs go ahead and create as many records as the program is eventually expected to use. That approach has the additional benefit of avoiding such problems as running out of room on the diskette before the entire file is completed.

If you simply begin writing data to a just-opened relative file, it will act much like a sequential file, putting the data elements written by the first Print# statement in Record #1, those written by the second Print# statement in record #2 and so on. (As this implies, each relative record must be written by a single Print# statement, using embedded carriage returns within the data to separate fields that will be read in via one or more Input# statements later.) However, it is far better to explicitly specify which record number is desired via a Record# command to the disk. This allows you to access records in any desired order, hopping anywhere in a file with equal ease. Properly used, it also avoids a subtle error (bug) common to all Commodore disk drives.

FORMAT FOR THE RECORD# COMMAND:

PRINT#15, "P" +CHR$ (channel # +96) + CHR$ (<record #) + CHR$ (>record #) + CHR$ (offset)

where "channel #" is the channel number specified in the current Open statement for the specified file, "<record #" is the low byte of the desired record number, expressed as a two byte integer, ">record #" is the high byte of the desired record number, and an optional "offset" value, if present, is the byte within the record at which a following Read or Write should begin.

To fully understand this command, we must understand how most integers are stored in computers based on the 6502 and related microprocessors. In the binary arithmetic used by the microprocessor, it is possible to express any unsigned integer from 0-255 in a single byte. It is also possible to store any unsigned integer from 0-65535 in 2 bytes, with 1 byte holding the part of the number that is evenly divisible by 256, and any remainder in the other byte. In machine language, such numbers are written backwards, with the low-order byte (the remainder) first, followed by the high order byte. In assembly language programs written with the Commodore Assembler, the low part of a two byte number is indicated by preceding its label with the less-than character (<). Similarly, the high part of the number is indicated by greater-than (>).

SAFETY NOTE: GIVE EACH RECORD# COMMAND TWICE!

To avoid the remote possibility of corrupting relative file data, it is necessary to give Record# commands twice—once before a record is read or written, and again immediately afterwards.

EXAMPLES:

To position the record pointer for file number 2 to record number 3, we could type:

PRINT #15, "P" +CHR$ (98) +CHR$ (3) +CHR$ (0)

The CHR$(98) comes from adding the constant (96) to the desired channel number (2). (96+2=98) Although the command appears to work even when 96 is not added to the channel number, the constant is normally added to maintain compatibility with the way Record# works on Commodore's CBM and PET computers.

Since 3 is less than 256, the high byte of its binary representation is 0, and the entire value fits into the low byte. Since we want to read or write from the beginning of the record, no offset value is needed.

Since these calculations quickly become tedious, most programs are written to do them for you. Here is an example of a program which inputs a record number and converts it into the required low byte/high byte form:

```
450 INPUT"RECORD # DESIRED";RE
460 IF RE<1 OR RE>65535 THEN 450
470 RH=INT(RE/256)
480 RL=RE-256*RH
490 PRINT#15, "P"+CHR$ (98) +CHR$ (RL) +CHR$ (RH)
```

Assuming RH and RL are calculated as in the previous example, programs may also use variables for the channel, record, and offset required:

```
570 INPUT "CHANNEL, RECORD, & OFFSET DESIRED";CH,RE,OF
630 PRINT#15,"P"+CHR$(CH+96)+CHR$(RL)+CHR$(RH)+CHR$ (OF)
```

ANOTHER RECORD# COMMAND

Basic 4.0 on Commodore's PET and CBM models includes a Basic Record# command not found in any of the serial bus computers. However, some available utility programs for these models include it. It serves the same function as the Record# command explained above, but has a simplified syntax:

RECORD#file #,record #,offset

where "file #" is the relative file number being used, not the command channel's file, "record #" is the desired record number, and "offset" is as above.

If you see a Record# command written in Basic 4 form in a program you want to use, simply convert it into the usual form for both Basic 2 and 3.5 described in this section.

COMPLETING RELATIVE FILE CREATION

Now that we have learned how to use both the Open and Record# commands, we are almost ready to properly create a relative file. The only additional fact we need to know is that CHR$(255) is a special character in a relative file. It is the character used by the DOS to fill relative records as they are created, before a program fills them with other information. Thus, if we want to write the last record we expect to need in our file with dummy data that will not interfere with our later work, CHR$(255) is the obvious choice. Here is how it works in an actual program which you may copy for use in your own relative file programs.

```
1020 OPEN 15,8,15                          Open command channel
1380 INPUT"ENTER RELATIVE FILE NAME";FI$   Select file parameters
1390 INPUT"ENTER MAX. # OF RECORDS";NR
1400 INPUT"ENTER RECORD LENGTH";RL
```

```
1410 OPEN 1,8,2,"0:"+FI$+",L,"+CHR$(RL)      Begin to create desired file
1420 GOSUB 59990                              Check for disk errors
1430 RH=INT(NR/256)                           Calculate length values
1440 RL=NR-256*RH
1450 PRINT#15,"P"+CHR$(96+2)+
     CHR$(RL)+CHR$(RH)                        Position to last record
                                              number
1460 GOSUB 59990
1470 PRINT#1,CHR$(255);                       Send default character to it
1480 GOSUB 59990
1490 PRINT#15,"P"+CHR$(96+2)+                 Re-position for safety
     CHR$(RL)+CHR$(RH)
1500 GOSUB 59990
1510 CLOSE 1                                  Now the file can be safely
                                              closed
1520 GOSUB 59990
9980 CLOSE 15                                 And the command channel
                                              closed
9990 END                                      Before we end the pro-
                                              gram
59980 REM CHECK DISK SUBROUTINE
59990 INPUT#15,EN,EM$,ET,ES
60000 IF EN>1 AND EN<>50 THEN PRINT
      EN,EM$,ET,ES:STOP                       Ignore "RECORD NOT
                                              PRESENT"
60010 RETURN
```

Two lines require additional explanation. When line 1470 executes, the disk drive will operate for up to ten or more minutes, creating all the records in the file, up to the maximum record number you selected in line 1390. This is normal, and only needs to be done once. During the process you may hear the drive motor turning and an occasional slight click as the head steps from track to track, everything is probably just fine. Second, line 60000 above is different from the equivalent line in the error check subroutine given earlier. Here disk error number 50 is specifically ignored, because it will be generated when the error channel is checked in line 1460. We ignore it because not having a requested record would only be an error if that record had previously been created.

EXPANDING A RELATIVE FILE

What if you underestimate your needs and need to expand a relative file later? No problem. Simply request the record number you need, even if it doesn't currently exist in the file. If there is no such record yet, DOS will create it as soon as you try to write information in it, and also automatically create any other missing records below it in number. The only penalty will be a slight time delay while the records are created.

WRITING RELATIVE FILE DATA

The commands used to read and write relative file data are the same Print#, Input#, and Get# commands used in the preceding chapter on Sequential files. Each command is used as described there. However, some aspects of relative file access do differ from sequential file programming, and we will cover those differences here.

DESIGNING A RELATIVE RECORD

As stated earlier in this chapter, each relative record has a fixed length, including all special characters. Within that fixed length, there are two popular ways to organize various individual fields of information. One is free-format, with individual fields varying in length from record to record, and each field separated from the next by a carriage return character (each of which does take up 1 character space in the record). The other approach is to use fixed-length fields, that may or may not be separated by carriage returns. If fixed length fields are not all separated by carriage returns, you will either need to be sure a carriage return is included within each 88 character portion of the record. If this is not done, you will have to use the Get# command to read the record, at a significant cost in speed.

Relative records of 88 or fewer characters, or final portions of records that are 88 or fewer characters in length, need not end in a carriage return. The 1541 is smart enough to recognize the end of a relative record even without a final carriage return. Though the saving of a single character isn't much, when multiplied by the number of records on a diskette, the savings could be significant.

Since each relative record must be written by a single Print# statement, the recommended approach is to build a copy of the current record in memory before writing it to disk. It can be collected into a single string variable with the help of Basic's many string-handling functions, and then all written out at once from that variable.

Here is an example. If we are writing a 4-line mail label, consisting of 4 fields named "NAME", "STREET", "CITY & STATE", and "ZIP CODE", and have a total record size of 87 characters, we can organize it in either of two ways:

WITH FIXED LENGTH FIELDS

Field Name	Length
NAME	27 characters
STREET	27 characters
CITY & STATE	23 characters
ZIP CODE	10 characters
Total length	87 characters

WITH VARIABLE LENGTH FIELDS

Field Name	Length
NAME	31 characters
STREET	31 characters
CITY & STATE	26 characters
ZIP CODE	11 characters
Potential length	99 characters
Edited length	87 characters

With fixed length records, the field lengths add up to exactly the record length. Since the total length is just within the Input buffer size limitation, no carriage return characters are needed. With variable length records, we can take advantage of the variability of actual address lengths. While one name contains 27 letters, another may have only 15, and the same variability exists in Street and City lengths. Although variable length records lose 1 character per field for carriage returns, they can take advantage of the difference

between maximum field length and average field length. A program that uses variable record lengths must calculate the total length of each record as it is entered, to be sure the total of all fields doesn't exceed the space available.

WRITING THE RECORD

Here is an example of program lines to enter variable length fields for the above file design, build them into a single string, and send them to record number RE in file number 3 (assumed to be a relative file that uses channel number 3).

```
150 CR$=CHR$(13)                          Carriage Return
...
2000 INPUT"NAME";NA$                      Enter fields
2010 IF LEN(NA$)>30 THEN 2000             And check length of each
2020 INPUT"STREET";SA$
2030 IF LEN(SA$)>30 THEN 2020
2040 INPUT"CITY & STATE";CS$
2050 IF LEN(CS$)>25 THEN 2040
2060 INPUT"ZIP CODE";ZP$
2070 IF LEN(ZP$)>10 THEN 2060
2080 DA$=NA$+CR$+SA$+CR$+
     CS$+CR$+ZP$                          Build output data string
2090 IF LEN(DA$)<87 THEN 2120             Check its length
2100 PRINT"RECORD TOO LONG"               If too long overall
2110 GOTO 2000
2120 RH=INT(RE/256)                       Calculate record number
2130 RL=RE-256*RH
2140 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)                    Position to record number RE
2150 GOSUB 59990                          Check for disk errors
2160 PRINT#3,DA$                          Send data to it
2170 GOSUB 59990
2180 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)                    Re-position for safety
2190 GOSUB 59990
```

To use the above program lines for the version with fixed length fields, we would alter a few lines as follows:

```
150                                       Not needed this time
160 BL$="                          "      27 shifted space characters
...
2000 INPUT"NAME";NA$
2005 LN=LEN(NA$)
2010 IF LEN(NA$)>27 THEN 2000             Checking for different lengths
2015 NA$=NA$+LEFT$(BL$,27-LN)             And padding to preset sizes
2020 INPUT"STREET";SA$
2025 LN=LEN(SA$)
```

```
2030 IF LEN(SA$)>27 THEN 2020
2035 SA$=SA$+LEFT$(BL$,27-LN)
2040 INPUT"CITY & STATE";CS$
2045 LN=LEN(CS$)
2050 IF LEN(CS$)>23 THEN 2040
2055 CS$=CS$+LEFT$(BL$,23-LN)
2060 INPUT"ZIP CODE";ZP$
2065 LN=LEN(ZP$)
2070 IF LEN(ZP$)>10 THEN 2060
2075 ZP$=ZP$+LEFT$(BL$,10-LN)
2080 DA$=NA$+SA$+CS$+ZP$                    Note lack of separators
2120 RH=INT(RE/256)
2130 RL=RE-256*RH
2140 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)+CHR$(1)
2150 GOSUB 59990
2160 PRINT#3,DA$;                           Note added semicolon
2170 GOSUB 59990
2180 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)+CHR$(1)
2190 GOSUB 59990
```

If field contents vary in length, variable field lengths are often preferable. On the other hand, if the field lengths are stable, fixed field lengths are preferable. Fixed length fields are also required if you want to use the optional offset parameter of the Record# command to point at a particular byte within a record. However, one warning must be made about using the offset this way. When any part of a record is written, DOS overwrites any remaining spaces in the record. Thus, if you must use the offset option, never update any field in a record other than the last one unless all succeeding fields will also be updated from memory later.

The above programs are careful to match record lengths exactly to the space available. Programs that don't do so will discover that DOS pads short records out to full size with fill characters, and truncates overlong records to fill only their allotted space. When a record is truncated, DOS will indicate error 51, "RECORD OVERFLOW", but short records will be accepted without a DOS error message.

READING A RELATIVE RECORD

Once a relative record has been written properly to diskette, reading it back into computer memory is fairly simple, but the procedure again varies, depending on whether it uses fixed or variable length fields. Here are the program lines needed to read back the variable fields created above from record number RE in file and channel 3:

```
3000 RH=INT(RE/256)                         Calculate record number
3010 RL=RE-256*RH
3020 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)+CHR$(1)              Position to record number RE
```

```
3030 GOSUB 59990                          Check for disk errors
3040 INPUT#1,NA$,SA$,CS$,ZP$              Read in fields
3050 GOSUB 59990
3060 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)                    Re-position for safety
```

Here are the lines needed to read back the version with fixed length fields:

```
3000 RH=INT(RE/256)
3010 RL=RE-256*RH
3020 PRINT#15,"P"+CHR$(96+3)+CHR$
     (RL)+CHR$(RH)
3030 GOSUB 59990
3040 INPUT#1,DA$                          Read in entire record
3050 GOSUB 59990
3060 PRINT#15,"P"+CHR$(96+3)+
     CHR$(RL)+CHR$(RH)
3070 NA$=LEFT$(DA$,27)                    Split data into fields
3080 SA$=MID$(DA$,28,27)
3090 CS$=MID$(DA$,55,23)
3100 ZP$=RIGHT$(DA$,10)
```

This ends our discussion of relative files. A complete "RELATIVE FILE" program, similar to the examples in this chapter, is included on the Test/Demo diskette.

THE VALUE OF INDEX FILES (ADVANCED USERS)

In the last two chapters we have learned how to use sequential and relative files separately. But they are often used together, with the sequential file used to keep brief records of which name in the relative file is stored in each record number. That way the contents of the sequential file can be read into a string array and sorted alphabetically. After sorting, a technique known as a binary search can be used to very quickly find an entered name in the array, and read in or write the associated record in the relative file. Advanced programs can maintain two or more such index files, sorted in differing ways simultaneously.

CHAPTER 7
DIRECT ACCESS COMMANDS

A TOOL FOR ADVANCED USERS

Direct access commands specify individual sectors on the diskette, reading and writing information entirely under your direction. This gives them almost complete flexibility in data-handling programs, but also imposes tremendous responsibilities on the programmer, to be sure nothing goes awry. As a result, they are normally used only in complex commercial programs able to properly organize data without help from the disk drive itself.

A far more common use of direct access commands is in utility programs used to view and alter parts of the diskette that are not normally seen directly. For instance, such commands can be used to change the name of a diskette without erasing all of its programs, to lock a program so it can't be erased, or hide your name in a location where it won't be expected.

DISKETTE ORGANIZATION

There are a total of 683 blocks on a 1541 diskette, of which 664 are available for use, with the rest reserved for the BAM (Block Availability Map) and the Directory.

The diskette's surface is divided into tracks, which are laid out as concentric circles on the surface of the diskette. There are 35 different tracks, starting with track 1 at the outside of the diskette to track 35 at the center. Track 18 is used for the directory, and the DOS fills up the diskette from the center outward, alternately in both directions.

Each track is subdivided into sectors (also called blocks). Because there is more room on the outer tracks, there are more sectors per track there. The outermost tracks contain 21 sectors each, while the innermost ones only have 17 sectors each. The table below shows the number of sectors per track.

Table 6.1: Track and Sector Format

TRACK NUMBER	SECTOR NUMBERS	TOTAL SECTORS
1 to 17	0 through 20	21
18 to 24	0 through 18	19
25 to 30	0 through 17	18
31 to 35	0 through 16	17

In this chapter we will describe the DOS commands for directly reading and writing any track and block on the diskette, as well as the commands used to mark blocks as used or unused. Unless otherwise notes, all direct access commands are the same in both Basic 2 and Basic 3.5.

OPENING A DATA CHANNEL FOR DIRECT ACCESS

When working with direct access data, you need two channels open to the disk: the command channel we've used throughout the book, and another for data. The command channel is opened with the usual OPEN 15,8,15 or equivalent. A direct access data

channel is opened much like other files, except that the pound sign (#), optionally followed by a memory buffer number, is used as a file name.

FORMAT FOR DIRECT ACCESS FILE OPEN STATEMENTS:

OPEN file #,device #, channel #, "#buffer #"

where "file #" is the file number, "device #" is the disk's device number, normally 8; "channel #" is the channel number, a number between 2 and 14 that is not used by other files open at the same time; and "buffer #", if present, is a 0, 1, 2, or 3, specifying the memory buffer within the 1541 to use for this file's data.

EXAMPLES:

If we don't specify which disk buffer to use, the 1541 will select one:

OPEN 5,8,5,"#"

Or we can make the choice ourselves:

OPEN 4,8,4,"#2"

BLOCK-READ

The purpose of a Block Read is to load the contents of a specified sector into a file buffer. Although the Block Read command (B-R) is still part of the DOS command set, it is nearly always replaced by the U1 command.

FORMAT FOR THE BLOCK-READ COMMAND:

PRINT#15, "U1"; channel #; drive #; track #; sector #

where "channel #" is the channel number specified when the file into which the block will be read was opened, "drive #" is the drive number, always 0 on the 1541, and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be read into the file buffer.

ALTERNATE FORMATS:

PRINT#15,"U1:"channel #;drive #;track #;sector #
PRINT#15,"UA:"channel #;drive #;track #;sector #
PRINT#15,"U1:channel #,drive #,track #,sector #"

EXAMPLE:

Here is a complete program to read a sector into disk memory using U1, and from there into computer memory via Get#. (If a carriage return will appear at least once in every 88 characters of data, Input# may be used in place of Get#).

```
110 MB=7936:REM $1F00          Define a memory buffer
120 INPUT"TRACK TO READ";T     Select a track
130 INPUT"SECTOR TO READ";S    and sector
140 OPEN 15,8,15               Open command channel
150 OPEN 5,8,5,"#"             Open direct access channel
160 PRINT#15,"U1";5;0;T;S      Read sector into disk buffer
170 FOR I=MB TO MB+255         Use a loop to
180 GET#5,A$:IF A$="" "        copy disk buffer
    THEN A$=CHR$(0)            into computer memory
190 POKE I,ASC(A$)             Tidy up after
200 NEXT
210 CLOSE 5:CLOSE 15
220 END
```

As the loop progresses, the contents of the specified track and sector are copied into computer memory, beginning at the address set by variable MB in line 160, and may be examined and altered there. This is the basis for programs like "DISPLAY T & S" on the Test/Demo diskette.

BLOCK-WRITE

The purpose of a Block Write is to save the contents of a file buffer into a specified sector. It is thus the reverse of the Block Read command. Although the Block Write command (B-W) is still part of the DOS command set, it is nearly always replaced by the U2 command.

FORMAT FOR THE BLOCK-WRITE COMMAND:

```
PRINT#15,"U2";channel #;drive #;track #;sector #
```

where "channel #" is the channel number specified when the file into which the block will be read was opened; "drive #" is the drive number (always 0 on the 1541); and "track #" and "sector #" are respectively the track and sector numbers that should receive the block of data being saved from the file buffer.

ALTERNATE FORMATS:

```
PRINT#15,"U2:"channel #;drive #;track #;sector #
PRINT#15,"UB:"channel #;drive #;track #;sector #
PRINT#15,"U2:channel #,drive #,track #,sector #"
```

EXAMPLES:

To restore track 18, sector 1 of the directory from the disk buffer filled by the Block Read example on page 82, we can use

```
PRINT#15,"U2";5;0;18;1
```

We'll return to this example on the next page, after we learn to alter the directory in a useful way.

We can also use a Block Write to write a name in Track 1, Sector 1, a rarely-used sector. This can be used as a way of marking a diskette as belonging to you. Here is a program to do it, using the alternate form of the Block Write command:

```
110 INPUT" YOUR NAME";NA$          Enter a name
120 OPEN 15,8,15                   Open command channel
130 OPEN 4,8,4,"#"                 Open direct access channel
140 PRINT#4,NA$                    Write name to buffer
150 PRINT#15,"U2";4;0;1;1          Write buffer to Track 1,
160 CLOSE 4                        Sector 1 of diskette
170 CLOSE 15                       Tidy up after
180 END
```

THE ORIGINAL BLOCK-READ AND BLOCK-WRITE COMMANDS (EXPERT USERS ONLY)

Although the Block Read and Block Write commands are nearly always replaced by the U1 and U2 commands respectively, the original commands can still be used, as long as you fully understand their effects. Unlike U1 and U2, B-R and B-W allow you to read or write less than a full sector. In the case of B-R, the first byte of the selected sector is used to set the buffer pointer (see next section), and determines how many bytes of that sector are read into a disk memory buffer. A program may check to be sure it doesn't attempt to read past the end of data actually loaded into the buffer, by watching for the value of the file status variable ST to change from 0 to 64. When the buffer is written back to diskette by B-W, the first byte written is the current value of the buffer pointer, and only that many bytes are written into the specified sector. B-R and B-W may thus be useful in working with custom-designed file structures.

FORMAT FOR THE ORIGINAL BLOCK-READ AND BLOCK-WRITE COMMANDS:

```
PRINT#15,"BLOCK-READ";channel #;drive #;track #;sector #
```

abbreviated as: PRINT#15,"B-R";channel #;drive #;track #;sector #

and

```
PRINT#15,"BLOCK-WRITE";channel #;drive #;track #;sector #
```

abbreviated as: PRINT#15,"B-W";channel #;drive #;track #;sector #

where "channel #" is the channel number specified when the file into which the block will be read was opened, "drive #" is the drive number (always 0 on the 1541), and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be partially read into or written from the file buffer.

IMPORTANT NOTES:

1. In a true Block-Read, the first byte of the selected sector is used to determine how many bytes of that sector to read into the disk memory buffer. It thus cannot be used to read an entire sector into the buffer, as the first data byte is always interpreted as being the number of characters to read, rather than part of the data.

2. Similarly, in a true Block-Write, when the buffer is written back to diskette, the first byte written is the current value of the buffer pointer, and only that many bytes are written into the specified sector. It cannot be used to rewrite an entire sector onto diskette unchanged, because the first data byte is overwritten by the buffer pointer.

THE BUFFER POINTER

The buffer pointer points to where the next Read or Write will begin within a disk memory buffer. By moving the buffer pointer, you can access individual bytes within a block in any order. This allows you to edit any portion of a sector, or organize it into fields, like a relative record.

FORMAT FOR THE BUFFER-POINTER COMMAND:

PRINT#15,"BUFFER-POINTER";channel #;byte

usually abbreviated as: PRINT#15,"B-P";channel #;byte

where "channel #" is the channel number specified when the file reserving the buffer was opened, and "byte" is the character number within the buffer at which to point.

ALTERNATE FORMATS:

PRINT#15,"B-P:"channel #;byte
PRINT#15,"B-P:channel #;byte"

EXAMPLE:

Here is a program that locks the first program or file on a 1541 diskette. It works by reading the start of the directory (Track 18, Sector 1) into disk memory, setting the buffer

pointer to the first file type byte (see Appendix C for details of directory organization), locking it by setting bit 6 and rewriting it.

```
110 OPEN 15,8,15                                  Open command channel
120 OPEN 5,8,5,"#"                                Open direct access channel
130 PRINT#15,"U1";5;0;18;1                        Read Track 18, Sector 1
140 PRINT#15,"B-P";5;2                            Point to Byte 2 of the buffer
150 GET#5,A$:IF A$="" " THEN A$=CHR$(0)           Read it into memory
160 A=ASC(A$) OR 64                               Turn on bit 6 to lock
170 PRINT#15,"B-P";5;2                            Point to Byte 2 again
180 PRINT#5,CHR$(A);                              Overwrite it in buffer
190 PRINT#15,"U2";5;0;18;1                        Rewrite buffer to diskette
200 CLOSE 5                                       Tidy up after
210 CLOSE 15
220 END
```

After the above program is run, the first file on that diskette can no longer be erased. If you later need to erase that file, re-run the same program, but substitute the revised line 160 below to unlock the file again:

```
160 A=ASC(A$) AND 191                             Turn off bit 6 to unlock
```

ALLOCATING BLOCKS

Once you have written something in a particular sector on a diskette with the help of direct access commands, you may wish to mark that sector as "already used," to keep other files from being written there. Blocks thus "allocated" will be safe until the diskette is validated.

FORMAT FOR BLOCK-ALLOCATE COMMAND:

PRINT#15,"BLOCK-ALLOCATE";drive #; track #;sector #

usually abbreviated as: PRINT#15,"B-A";drive #; track #;sector #

where "drive #" is the drive number, always 0 on the 1541, and "track #" and "sector #" are the track and sector containing the block of data to be read into the file buffer.

ALTERNATE FORMAT:

PRINT#15,"B-A:";drive #; track #;sector #

EXAMPLE:

If you try to allocate a block that isn't available, the DOS will set the error message to number 65, NO BLOCK, and set the track and block numbers in the error message to

the next available track and block number. Therefore, before selecting a block to write, try to allocate that block. If the block isn't available; read the next available block from the error channel and allocate it instead. However, do not allocate data blocks in the directory track. If the track number returned is 0, the diskette is full.

Here is a program that allocates a place to store a message on a diskette.

```
100 OPEN15,8,15                    Open command channel
110 OPEN5,8,5,"#"                  " direct access "
120 PRINT#5,"I THINK
    THEREFORE I AM"                Write a message to buffer
130 T=1:S=1                        Start at first track & sector
140 PRINT#15,"B-A";0;T;S           Try allocating it
150 INPUT#15,EN,EM$,ET,ES          See if it worked
160 IF EN=0 THEN 210               If so, we're almost done
170 IF EN<>65 THEN PRINT
    EN,EM$,ET,ES:STOP              "NO BLOCK" means already allocated
180 IF ET=0 THEN PRINT
    "DISK FULL":STOP               If next track is 0, we're out of room
190 IF ET=18 THEN ET=19:ES=0       Don't allocate the directory!
200 T=ET:S=ES:GOTO 140             Try suggested track & sector next
210 PRINT#15,"U2";5;0;T;S          Write buffer to allocated sector
220 PRINT "STORED AT:",T,S         Say where message went
230 CLOSE 5:CLOSE 15               and tidy up
240 END
```

FREEING BLOCKS

The Block-Free command is the opposite of Block-Allocate. It frees a block that you don't need any more, for re-use by the DOS. Block-Free updates the BAM to show a particular sector is not in use, rather than actually erasing any data.

FORMAT FOR BLOCK-FREE COMMAND:

PRINT#15,"BLOCK-FREE";drive #;track #;sector #

abbreviated as: PRINT#15,"B-F";drive #;track #;sector #

where "drive #" is the drive number (always 0 on the 1541), and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be read into the file buffer.

ALTERNATE FORMAT:

PRINT#15,"B-F:";drive #;track #;sector #

EXAMPLE:

To free the sector in which we wrote our name in the Block Write example, and allocated in the first Block-Allocate example, we could use the following command:

```
PRINT#15,"B-F";0;1;1
```

USING RANDOM FILES (ADVANCED USERS ONLY)

By combining the commands in this chapter, it is possible to develop a file-handing program that uses random files. What you need to know now is how to keep track of which blocks on the disk such a file has used. (Even though you know a sector has not been allocated by your random file, you must also be sure it wasn't allocated by another unrelated file on the diskette.)

The most common way of recording which sectors have been used by a random file is in a sequential file. The sequential file stores a list of record numbers, with the track, sector, and byte location of each record. This means three channels are needed by a random file: one for the command channel, one for the random data, and the last for the sequential data.

CHAPTER 8
INTERNAL DISK COMMANDS

Expert programmers can give commands that directly alter the workings of the 1541, much as skilled programmers can alter the workings of Basic inside the computer with Peeks, Pokes and Sys calls. It is also possible to write machine language programs that load and run entirely within the 1541, either by writing them into disk memory from the computer, or by loading them directly from diskette into the desired disk memory buffer. In use, this is similar to loading and running machine language programs in your computer.

As when learning to use Peek(), Poke and Sys in your computer, extreme caution is advised in using the commands in this chapter. They are essentially machine language commands, and lack all of Basic's safeguards. If anything goes wrong, you may have to turn the disk drive off and on again (after removing the diskette) to regain control. Do not practice these commands on any important diskette. Rather, make a spare copy and work with that. Knowing how to program a 6502 in machine language will help greatly, and you will also need a good memory map of the 1541. A brief 1541 map appears below.

1541 MEMORY MAP

Location	Purpose
	2K of RAM memory
0000-00FF	Zero page work area: job queue, important variables & pointers
0100-01FF	Stack work area
0200-02FF	Command buffers & tables: channels, parser, output, variables
0300-07FF	Data buffers 0-4, 1 per page of memory.
	Input/Output chips
1800-180F	6522 VIA: I/O to computer
1C00-1C0F	6522 VIA: I/O to disk controller
	Disk Operating System ROM
C100-F258	Interface Processor: receive & interpret commands from computer
F259-FE66	Floppy Disk Controller: executes IP's commands, controls mechanism
FE67-FE84	IRQ handler: switches from IP to FDC & back every 10 ms.
FE85-FEE6	ROM tables & constants
FEE7-FF0F	Patch area
FFE6-FFFF	JMP table: User command vectors

Other Resources:

More detailed information about Commodore disk drives can be found in these books:

Inside Commodore DOS, by Immers & Neufeld (Datamost, c1984)
The Anatomy of the 1541 Disk Drive, by Englisch & Szczepanowski (Abacus, c1984)
Programming the PET/CBM, by West (Level Limited, c1982)
The PET Personal Computer Guide, by Osborne & Strasmas (Osborne/McGraw-Hill, c1982)

MEMORY-READ

The disk contains 16K of ROM (Read-Only Memory), as well as 2K of RAM (Read-Write Memory). You can get direct access to any location within these, or to the buffers that the DOS has set up in RAM, by using memory commands. Memory-Read allows you to select which byte or bytes to read from disk memory into the computer. The Memory-Read command is the equivalent of the Basic Peek() function, but reads the disk's memory instead of the computer's memory.

Note: Unlike other disk commands, those in this chapter cannot be spelled out in full. Thus, M-R is correct, but MEMORY-READ is not a permitted alternate wording.

FORMAT FOR THE MEMORY-READ COMMAND:

PRINT#15,"M-R"CHR$(<address)CHR$(>address)CHR$(# of bytes)

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory to be read. If the optional "# of bytes" is specified, it selects how many memory locations will be read in, from 1-255. Otherwise, 1 character will be read. If desired, a colon (:) may follow M-R inside the quotation marks.

ALTERNATE FORMAT:

PRINT#15,"M-R:"CHR$(<address)CHR$(>address)CHR$(# of bytes)

The next byte read using the Get# statement through channel #15 (the error channel), will be from that address in the disk controller's memory, and successive bytes will be from successive memory locations.

Any Input# from the error channel will give peculiar results when you're using this command. This can be cleared up by sending any other command to the disk, except another memory command.

EXAMPLES:

To see how many tries the disk will make to read a particular sector, and whether "seeks" one-half track to each side will be attempted if a read fails, and whether "bumps" to track one and back will be attempted before declaring the sector unreadable, we can use the following lines. They will read a special variable in the zero page of disk memory, called REVCNT. It is located at $6A hexadecimal ($6A hexadecimal = 6 x 16 + 10 = 106).

```
110 OPEN 15,8,15                                    Open command channel
120 PRINT#15,"M-R"CHR$(106)CHR$(0)                  Same as G=PEEK(106)
130 GET#15,G$:IF G$="" THEN G$=CHR$(0)
140 G=ASC(G$)
150 B=G AND 128:B$="ON":IF B THEN B$="OFF"          Check bit 7
160 S=G AND 64:S$="ON":IF S THEN S$="OFF"           Check bit 6
170 T=G AND 31:PRINT "# OF TRIES IS";T              Check bits 0-5
180 PRINT "BUMPS ARE";B$                            and give results
190 PRINT "SEEKS ARE";S$
200 CLOSE 15                                        Tidy up after
210 END
```

Here's a more general purpose program that reads one or more locations anywhere in disk memory:

```
110 OPEN15,8,15                                     Open command channel
120 INPUT"# OF BYTES TO READ (0=END)";NL            Enter number of bytes wanted
130 IF NL<1 THEN CLOSE 15:END                       unless done
140 IF NL>255 THEN 120                              or way out of line
150 INPUT"STARTING AT ADDRESS";AD                   Enter starting address
160 AH=INT(AD/256):AL=AD-AH*256                     Convert it into disk form
170 PRINT#15,"M-R"CHR$(AL)CHR$(AH)                  Actual Memory-Read
    CHR$(NL)
180 FOR I=1 TO NL                                   Loop til have all the data
190 : GET#15,A$:IF A$="" THEN A$=CHR$(0)
200 : PRINT ASC(A$);                                printing it as we go
210 NEXT I
220 PRINT
230 GOTO 120                                        Forever
```

MEMORY-WRITE

The Memory-Write command is the equivalent of the Basic Poke command, but has its effect in disk memory instead of within the computer. M-W allows you to write up to 34 bytes at a time into disk memory. The Memory-Execute and some User commands can be used to run any programs written this way.

FORMAT FOR THE MEMORY-WRITE COMMAND:

PRINT#15,"M-W"CHR$(<address)CHR$(>address)CHR$
(# of bytes)CHR$(data byte(s))

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory to begin writing, "# of bytes" is the number of memory locations that will be written (from 1-34), and "data byte" is 1 or more byte values to be written into disk memory, each as a CHR$() value. If desired, a colon (:) may follow M-W within the quotation marks.

ALTERNATE FORMAT:

PRINT#15,"M-W:"CHR$(<address)CHR$(>address)CHR$
(# of bytes)CHR$(data byte(s))

EXAMPLES:

We can use this line to turn off the "bumps" when loading DOS-protected programs (i.e., programs that have been protected against being copied by creating and checking for specific disk errors).

PRINT#15,"M-W"CHR$(106)CHR$(0)CHR$(1)CHR$(133)

The following line can be used to recover bad sectors, such as when an important file has been damaged and cannot be read normally.

PRINT#15,"M-W"CHR$(106)CHR$(0)CHR$(1)CHR$(31)

The above two examples may be very useful under some circumstances. They are the equivalent of POKE 106,133 and POKE 106,31 respectively, but in disk memory, not inside the computer. As mentioned in the previous section's first example, location 106 in the 1541 disk drive signifies three separate activities to the drive, all related to error recovery. Bit 7 (the high bit), if set means no bumps (don't thump the drive back to track 1). Bit 6, if set, means no seeks. In that case, the drive won't attempt to read the half-track above and below the assigned track to see if it can read the data that way. The bottom 6 bits are the count of how many times the disk will try to read each sector before and after trying seeks and bumps before giving up. Since 31 is the largest number that can be expressed in 6 bits, that is the maximum number of tries allowed.

From this example, you can see the value of knowing something about Peeks, Pokes, and machine-language before using direct-access disk commands, as well as their potential power.

MEMORY-EXECUTE

Any routine in disk memory, either in RAM or ROM, can be executed with the Memory-Execute command. It is the equivalent of the Basic Sys call to a machine language program or subroutine, but works in disk memory instead of within the computer.

FORMAT FOR THE MEMORY-EXECUTE COMMAND:

PRINT#15,"M-E"CHR$(<address)CHR$(>address)

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory at which execution is to begin.

ALTERNATE FORMAT:

PRINT#15,"M-E:"CHR$(<address)CHR$(>address)

EXAMPLE:

Here is a Memory-Execute command that does absolutely nothing. The first instruction it executes is an RTS, which ends the command:

PRINT#15,"M-E"CHR$(88)CHR$(242)

A more plausible use for this command would be to artificially trigger an error message. Don't forget to check the error channel, or you'll miss the message:

PRINT#15,"M-E"CHR$(201)CHR$(239)

However, most uses require intimate knowledge of the inner workings of the DOS, and preliminary setup with other commands, such as Memory-Write.

BLOCK-EXECUTE

This rarely-used command will load a sector containing a machine language routine into a memory buffer from diskette, and execute it from the first location within the buffer, until a ReTurn from Subroutine (RTS) instruction ends the command.

FORMAT FOR THE BLOCK-EXECUTE COMMAND:

PRINT#15,"B-E";channel #;drive #;track #;sector #

where "channel #" is the channel number specified when the file into which the block will be loaded was opened, "drive #" is the drive number (always 0 on the 1541), and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be loaded into the file buffer and executed there.

ALTERNATE FORMATS:

```
PRINT#15,"B-E:";channel #;drive #;track #;sector #
PRINT#15,"B-E:channel #,drive #,track #,sector #"
```

EXAMPLES:

Assuming you've written a machine language program onto Track 1, Sector 8 of a diskette, and would like to run it in buffer number 1 in disk memory (starting at $0400 hexadecimal, you could do so as follows:

```
110 OPEN 15,8,15          Open command channel
120 OPEN 2,8,2,"#1"       Open direct access channel to buffer 1
130 PRINT#15,"B-E";2;0;1;8  Load Track 1, Sector 8 in it & execute
140 CLOSE 2               Tidy up after
150 CLOSE 15
160 END
```

USER COMMANDS

Most User commands are intended to be used as machine language JMP or Basic SYS commands to machine language programs that reside inside the disk memory. However, some of them have other uses as well. The User1 and User2 commands are used to replace the Block-Read and Block-Write commands, UI re-starts the 1541 without changing its variables, UJ cold-starts the 1541 almost as if it had been turned off and on again, and UI- speeds up the 1541 when used with the VIC 20 only. (Note: VIC 20 owners don't have to use UI-; the 1541 works with the VIC 20, with or without this command.)

User Command	Function
U1 or UA	replaces Block-Read
U2 or UB	replaces Block-Write
U3 or UC	JMP $0500 (same as SYS 5*256, but within the 1541 itself.)
U4 or UD	JMP $0503 (SYS 5*256+3)
U5 or UE	JMP $0506 (" 5*256+6)
U6 or UF	JMP $0509 (" 5*256+9)
U7 or UG	JMP $050C (" 5*256+12)
U8 or UH	JMP $050F (" 5*256+15)
U9 or UI	1541 NMI (non-maskable interrupt—warm start)
U: or UJ	1541 reset (cold start, allow 2 seconds before next command.)
UI+	restore 1541 to usual speed
UI-	speed 1541 up by 25% when used with VIC 20 only.

By loading these memory locations with another machine language JMP command, such as JMP $0520, you can create longer routines that operate in the disk's memory along with an easy-to-use jump table.

FORMAT FOR USER COMMANDS:

PRINT#15,"Ucharacter"

where "character" defines one of the preset user commands listed above.

EXAMPLES:

PRINT#15,"U:"	Preferred form of DOS RESET command
PRINT#15,"U3"	Execute program at start of buffer 2

CHAPTER 9
MACHINE LANGUAGE PROGRAMS

Here is a list of disk-related Kernal ROM subroutines and a practical example of their use in a program which reads a sequential file into memory from disk. Note that most require advance setup of one or more processor registers or memory locations, and all are called with the assembly language JSR command.

For a more complete description as to what each routine does and how parameters are set for each routine, see the Programmer's Reference Guide for your specific computer.

DISK-RELATED KERNAL SUBROUTINES

Label		Address	Function
SETLFS	=	$FFBA	;SET LOGICAL, FIRST & SECOND ADDRESSES
SETNAM	=	$FFBD	;SET LENGTH & ADDRESS OF FILENAME
OPEN	=	$FFC0	;OPEN LOGICAL FILE
CLOSE	=	$FFC3	;CLOSE LOGICAL FILE
CHKIN	=	$FFC6	;SELECT CHANNEL FOR INPUT
CHKOUT	=	$FFC9	;SELECT CHANNEL FOR OUTPUT
CLRCHN	=	$FFCC	;CLEAR ALL CHANNELS & RESTORE DEFAULT I/O
CHRIN	=	$FFCF	;GET BYTE FROM CURRENT INPUT DEVICE
CHROUT	=	$FFD2	;OUTPUT BYTE TO CURRENT OUTPUT DEVICE

```
START   LDA  #4          ;SET LENGTH & ADDRESS
        LDX  #<FNADR     ;OF FILE NAME, LOW
        LDY  #>FNADR     ;& HIGH BYTES
        JSR  SETNAM      ;FOR NAME SETTER
        LDA  #3          ;SET FILE NUMBER,
        LDX  #8          ;DISK DEVICE NUMBER,
        LDY  #0          ;AND SECONDARY ADDRESS
        JSR  SETLFS      ;AND SET THEM
        JSR  OPEN        ;OPEN 3,8,0,"TEST"
        LDX  #3
        JSR  CHKIN       ;SELECT FILE 3 FOR INPUT
NEXT    JSR  CHRIN       ;GET NEXT BYTE FROM FILE
        BEQ  END         ;UNTIL FINISH OR FAIL
        JSR  CHROUT      ;OUTPUT BYTE TO SCREEN
        JMP  NEXT        ;AND LOOP BACK FOR MORE
;
END     LDA  #3          ;WHEN DONE
        JSR  CLOSE       ;CLOSE FILE
        JSR  CLRCHN      ;RESTORE DEFAULT I/O
        RTS              ;BACK TO BASIC
;
FNADR   .BYT "TEST"      ;STORE FILE NAME HERE
```

APPENDIX A: CHANGING THE DEVICE NUMBER

SOFTWARE METHOD

One way to temporarily change the device number of a disk drive is via a program. When power is first turned on, the drive reads an I/O location whose value is controlled by a jumper on its circuit board, and writes the device number it reads there into memory locations 119 and 120. Any time thereafter, you may write over that device number with a new one, which will be effective until it is changed again, or the 1541 is reset.

FORMAT FOR TEMPORARILY CHANGING THE DISK DEVICE NUMBER:

```
PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$
  (device # + 32)CHR$(device # + 64)
```

EXAMPLE:

Here is a program that sets any device number from 8-11:

```
10 INPUT'NEW DEVICE NUMBER';DV
20 IF DV<8 OR DV>11 THEN 10
30 OPEN 15,8,15
40 PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(DV + 32)CHR$(DV + 64)
50 CLOSE 15
```

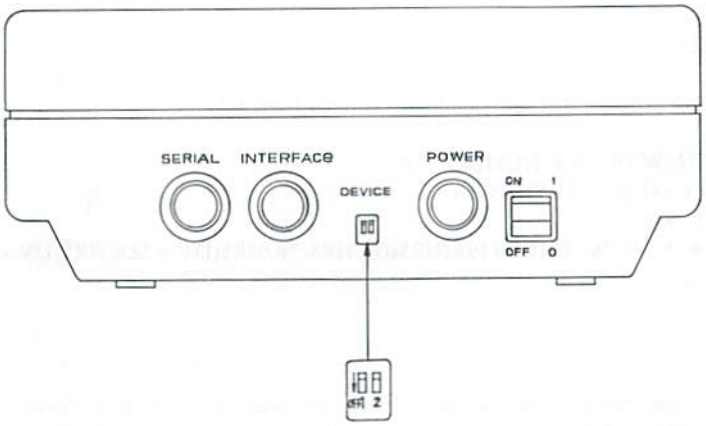
Note: If you will be using two disk drives, and want to temporarily change the device number of one, you will need to run the above program with the disk drive whose device number is not to be changed turned off. After the program has been run, you may turn that drive back on. If you need to connect more than two drives at once, you will need to use the hardware method of changing device numbers, although you may be able to get by in an emergency by unplugging the serial bus cable from drives whose device number has already been set while changing others. This is not recommended, however, as there is always danger of damaging electronic devices when plugging in cables with the power on.

STEPS TO CHANGE THE DEVICE NUMBER USING THE SWITCH

1. Turn off the disk drive.
2. Refer to the following chart to set the DIP SW at the back of the disk drive for device number setting.

Device #	sw1	sw2
8	ON	ON
9	OFF	ON
10	ON	OFF
11	OFF	OFF

3. Turn on the disk drive.



APPENDIX B: DOS ERROR MESSAGES AND LIKELY CAUSES

Note: Many commercial program diskettes are intentionally created with one or more of the following errors, to keep programs from being improperly duplicated. If a disk error occurs while you are making a security copy of a commercial program diskette, check the program's manual. If its copyright statement does not permit purchasers to copy the program for their own use, you may not be able to duplicate the diskette. In some such cases, a safety spare copy of the program diskette is available from your dealer or directly from the company for a reasonable fee.

00: OK (not an error)
This is the message that usually appears when the error channel is checked. It means there is no current error in the disk unit.

01: FILES SCRATCHED (not an error)
This is the message that appears when the error channel is checked after using the Scratch command. The track number tells how many files were erased.

NOTE: If any other error message numbers less than 20 ever appear, they may be ignored. All true errors have numbers of 20 or more.

20: READ ERROR (block header not found)
The disk controller is unable to locate the header of the requested data block. Caused by an illegal block, or a header that has been destroyed. Usually unrecoverable.

21: READ ERROR (no sync character)
The disk controller is unable to detect a sync mark on the desired track. Caused by misalignment, or a diskette that is absent, unformatted or improperly seated. Can also indicate hardware failure. Unless caused by one of the above simple causes, this error is usually unrecoverable.

22: READ ERROR (data block not present)
The disk controller has been requested to read or verify a data block that was not properly written. Occurs in conjunction with Block commands and indicates an illegal track and/or sector request.

23: READ ERROR (checksum error in data block)
There is an error in the data. The sector has been read into disk memory, but its checksum is wrong. May indicate grounding problems. This fairly minor error is often repairable by simply reading and rewriting the sector with direct access commands.

24: READ ERROR (byte decoding error)
The data or header has been read into disk memory, but a hardware error has been created by an invalid bit pattern in the data byte. May indicate grounding problems.

25: WRITE ERROR (write-verify error)
The controller has detected a mismatch between the data written to diskette and the same data in disk memory. May mean the diskette is faulty. If so, try another. Use only high quality diskettes from reputable makers.

26: WRITE PROTECT ON
The controller has been requested to write a data block while the write protect sensor is covered. Usually caused by writing to a diskette whose write protect notch is covered over with tape to prevent changing the diskette's contents.

27: READ ERROR (checksum error in header)
The controller detected an error in the header bytes of the requested data block. The block was not read into disk memory. May indicate grounding problems. Usually unrecoverable.

28: WRITE ERROR (long data block)
The controller attempts to detect the sync mark of the next header after writing a data block. If the sync mark does not appear on time, this error message is generated. It is caused by a bad diskette format (the data extends into the next block) or by a hardware failure.

29: DISK ID MISMATCH
The disk controller has been requested to access a diskette which has not been initialized. Can also occur if a diskette has a bad header.

30: SYNTAX ERROR (general syntax)
The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names, or an illegal pattern. Check your typing and try again.

31: SYNTAX ERROR (invalid command)
The DOS does not recognize the command. It must begin with the first character sent. Check your typing and try again.

32: SYNTAX ERROR (invalid command)
The command sent is longer than 58 characters. Use abbreviated disk commands.

33: SYNTAX ERROR (invalid file name)
Pattern matching characters cannot be used in the Save command or when Opening files for the purpose of Writing new data. Spell out the file name.

34: SYNTAX ERROR (no file given)
The file name was left out of a command or the DOS does not recognize it as such. Typically, a colon (:) has been omitted. Try again.

39: SYNTAX ERROR (invalid command)
The DOS does not recognize a command sent to the command channel (secondary address 15). Check your typing and try again.

50: RECORD NOT PRESENT
The requested record number has not been created yet. This is not an error in a new relative file or one that is being intentionally expanded. It results from reading past the last existing record, or positioning to a non-existent record number with the Record# command.

51: OVERFLOW IN RECORD
The data to be written in the current record exceeds the record size. The excess has been truncated (cut off). Be sure to include all special characters (such as carriage returns) in calculating record sizes.

52: FILE TOO LARGE
There isn't room left on the diskette to create the requested relative record. To avoid this error, create the last record number that will be needed as you first create the file. If the file is unavoidably too large for the diskette, either split it into two files on two diskettes, or use abbreviations in the data to allow shorter records.

60: WRITE FILE OPEN
A write file that has not been closed is being re-opened for reading. This file must be immediately rescued, as described in Housekeeping Hint #2 in Chapter 4, or it will become a splat (improperly closed) file and probably be lost.

61: FILE NOT OPEN
A file is being accessed that has not been opened by the DOS. In some such cases no error message is generated. Rather the request is simply ignored.

62: FILE NOT FOUND
The requested file does not exist on the indicated drive. Check your spelling and try again.

63: FILE EXISTS
A file with the same name as has been requested for a new file already exists on the diskette. Duplicate file names are not allowed. Select another name.

64: FILE TYPE MISMATCH
The requested file access is not possible using files of the type named. Reread the chapter covering that file type.

65: NO BLOCK
Occurs in conjunction with B-A. The sector you tried to allocate is already allocated. The track and sector numbers returned are the next higher track and sector available. If the track number returned is zero (0), all remaining sectors are full. If the diskette is not full yet, try a lower track and sector.

66: ILLEGAL TRACK AND SECTOR

The DOS has attempted to access a track or sector which does not exist. May indicate a faulty link pointer in a data block.

67: ILLEGAL SYSTEM T OR S

This special error message indicates an illegal system track or block.

70: NO CHANNEL (available)

The requested channel is not available, or all channels are in use. A maximum of three sequential files or one relative file plus one sequential file may be opened at one time, plus the command channel. Do not omit the drive number in a sequential Open command, or only two sequential files can be used. Close all files as soon as you no longer need them.

71: DIRECTORY ERROR

The BAM (Block Availability Map) on the diskette does not match the copy in disk memory. To correct, Initialize the diskette.

72: DISK FULL

Either the diskette or its directory is full. DISK FULL is sent when 2 blocks are still available, allowing the current file to be closed. If you get this message and the directory shows any blocks left, you have too many separate files in your directory, and will need to combine some, delete any that are no longer needed, or copy some to another diskette.

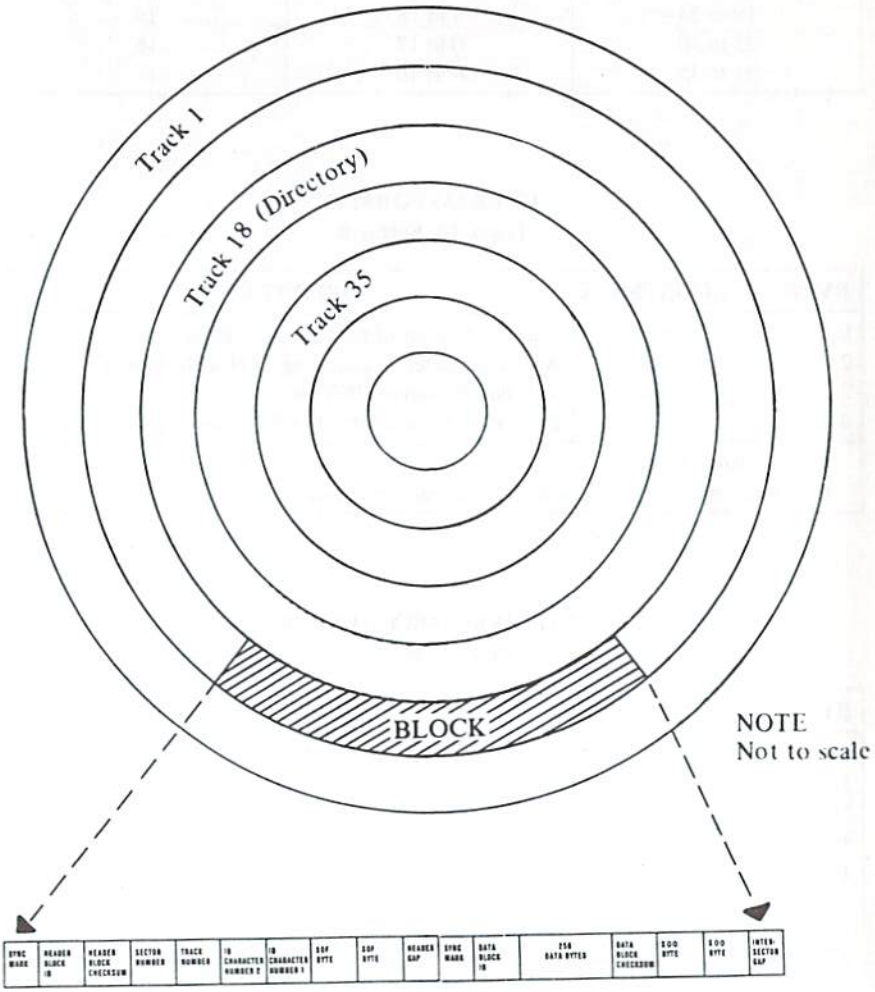
73: DOS MISMATCH (CBM DOS V2.6 1541)

If the disk error status is checked when the drive is first turned on, before a directory or other command has been given, this message will appear. In that use, it is not an error, but rather an easy way to see which version of DOS is in use. If the message appears at other times, an attempt has been made to write to a diskette with an incompatible format, such as the former DOS 1 on the Commodore 2040 disk drive. Use one of the copy programs on the Test/Demo diskette to copy the desired file(s) to a 1541 diskette.

74: DRIVE NOT READY

An attempt has been made to access the 1541 single disk without a formatted diskette in place. Blank diskettes cannot be used until they have been formatted.

PENDIX C: DISKETTE FORMATS



1541 Format: Expanded View of a Single Sector

1541 BLOCK DISTRIBUTION BY TRACK

Track number	Range of Sectors	Total # of Sectors
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17

1541 BAM FORMAT
Track 18, Sector 0.

BYTE	CONTENTS	DEFINITION
0,1	18,01	Track and sector of first directory block.
2	65	ASCII character A indicating 1541/4040 format.
3	0	Null flag for future DOS use.
4-143		Bit map of the available blocks in tracks 1-35.
1 = available block 0 = block not available (each bit represents one block)		

1541 DIRECTORY HEADER
Track 18, Sector 0.

BYTE	CONTENTS	DEFINITION
144-159		Diskette name padded with shifted spaces.
160-161	160	Shifted spaces.
162-163		Diskette ID.
164	160	Shifted space
165-166	50,65	ASCII representation of 2A, which are, respectively, the DOS version (2) and format type (1541/4040).
167-170	160	Shifted spaces.
170-255	0	Nulls ($00), not used.

PROGRAM FILE FORMAT

BYTE	DEFINITION
FIRST SECTOR	
0,1	Track and sector of next block in program file.
2,3	Load address of the program
4-255	Next 252 bytes of program information stored as in computer memory (with key words tokenized).
REMAINING FULL SECTORS	
0,1	Track and sector of next block in program file.
2-255	Next 254 bytes of program info stored as in computer memory (with key words tokenized).
FINAL SECTOR	
0,1	Null ($00), followed by number of valid data bytes in sector.
2-???	Last bytes of the program information, stored as in computer memory (with key words tokenized). The end of a Basic file is marked by 3 zero bytes in a row. Any remaining bytes in the sector are garbage, and may be ignored.

SEQUENTIAL FILE FORMAT

BYTE	DEFINITION
ALL BUT FINAL SECTOR	
0-1	Track and sector of next sequential data block.
2-255	254 bytes of data.
FINAL SECTOR	
0,1	Null ($00), followed by number of valid data bytes in sector.
2-???	Last bytes of data. Any remaining bytes are garbage and may be ignored.

1541 RELATIVE FILE FORMAT

BYTE	DEFINITION
DATA BLOCK	
0,1	Track and sector of next data block.
2-255	254 bytes of data. Empty records contain $FF (all binary ones) in the first byte followed by $00 (binary all zeros) to the end of the record. Partially filled records are padded with nulls ($00).
SIDE SECTOR BLOCK	
0-1	Track and sector of next side sector block.
2	Side sector number (0-5)
3	Record length
4-5	Track and sector of first side sector (number 0)
6-7	Track and sector of second side sector (number 1)
8-9	Track and sector of third side sector (number 2)
10-11	Track and sector of fourth side sector (number 3)
12-13	Track and sector of fifth side sector (number 4)
14-15	Track and sector of sixth side sector (number 5)
16-255	Track and sector pointers to 120 data blocks.

1541 DIRECTORY FILE FORMAT
Track 18, Sector 1.

BYTE	DEFINITION
0,1	Track and sector of next directory block.
2-31	File entry 1*
34-63	File entry 2*
66-95	File entry 3*
98-127	File entry 4*
130-159	File entry 5*
162-191	File entry 6*
194-223	File entry 7*
226-255	File entry 8*

*STRUCTURE OF EACH INDIVIDUAL DIRECTORY ENTRY

BYTE	CONTENTS	DEFINITION
0	128 + type	File type OR'ed with $80 to indicate properly closed file. (if OR'ed with $C0 instead, file is locked.) TYPES: 0 = DELeted 1 = SEQuential 2 = PROGram 3 = USER 4 = RELative
1-2		Track and sector of first data block.
3-18		File name padded with shifted spaces.
19-20		Relative file only: track and sector of first side sector block.
21		Relative file only: Record length.
22-25		Unused.
26-27		Track and sector of replacement file during an @SAVE or @OPEN.
28-29		Number of blocks in file: stored as a two-byte integer, in low byte, high byte order.

APPENDIX D: DISK COMMAND QUICK REFERENCE CHART

General Format: OPEN 15,8,15:PRINT#15,command:CLOSE 15 (Basic 2)

HOUSEKEEPING COMMANDS

BASIC 2	NEW	"N0:diskette name,id"
	COPY	"C0:new file=0:old file"
	RENAME	"R0:new name=old name"
	SCRATCH	"S0:file name"
	INITIALIZE	"I0"
	VALIDATE	"V0"
BASIC 3.5	NEW	HEADER "diskette name," Iid,D0
	COPY	COPY "old file" TO "new file"
	RENAME	RENAME "old name" TO "new name"
	SCRATCH	SCRATCH "file name"
	VALIDATE	COLLECT
BOTH	INITIALIZE	"I0"

FILE COMMANDS

BASIC 2	LOAD	LOAD "file name",8
	SAVE	SAVE "0:file name",8
BASIC 3.5	LOAD	DLOAD "file name"
	SAVE	DSAVE "file name"
BOTH	CLOSE	CLOSE file #
	GET#	GET#file #,variable list
	INPUT#	INPUT#file #,variable list
	OPEN	OPEN file #,8,channel #,"0:file name,file type,direction"
	PRINT#	PRINT#file #,data list
	RECORD#	"P"+CHR$(channel #)+CHR$(<record #)+CHR$(>record #)+CHR$(offset)

DIRECT ACCESS COMMANDS

BLOCK-ALLOCATE	"B-A";0;track #;sector #
BLOCK-EXECUTE	"B-E";channel #;0;track #;sector #
BLOCK-FREE	"B-F";0;track #;sector #
BUFFER-POINTER	"B-P";channel #;byte
BLOCK-READ	"U1";channel #;0;track #;sector #
BLOCK-WRITE	"U2";channel #;0;track #;sector #
MEMORY-EXECUTE	"M-E"CHR$(<address)CHR$(>address)
MEMORY-READ	"M-R"CHR$(<address)CHR$(>address)CHR$(# of bytes)
MEMORY-WRITE	"M-W"CHR$(<address)CHR$(>address)CHR$(# of bytes) CHR$ (data byte)
USER	"Ucharacter"

APPENDIX E: TEST/DEMO DISKETTE

HOW TO USE

The "HOW TO USE" programs provide brief descriptions of the other programs on the Test/Demo diskette.

THE VIC-20 & C-64 WEDGES

Additional commands are available which allow you to type short instructions to the disk drive. Load and run the VIC-20 WEDGE if you have a VIC; use the C-64 WEDGE if you have a Commodore 64. Using either, you will be able to press backslash (/) followed by the program name and the RETURN key to load a program; the "/" means load from disk drive. For example, type "/how to use" to load that program. Type ">" or "@" and then press RETURN to display the current disk error status. Type ">$" or "@$" and RETURN to display the directory without erasing the current program.

DOS 5.1

The DOS 5.1 program is not intended to be loaded directly, but is loaded instead from the program C-64 WEDGE. Its load address is $CCØØ hexadecimal.

PRINTER TEST

The PRINTER TEST prints a listing of characters in a form that makes it easy to check the mechanical and electronic capabilities of the printer.

DISK ADDR CHANGE

Use this program to change the device number of the disk drive. It is a soft change—when the system is turned off, the disk drive is reset to its original device number.

VIEW BAM

The VIEW BAM program allows a programmer to examine the contents of the sectors that make up the block availability map (BAM)—the table that DOS uses to identify blocks that have been allocated to the files on that diskette.

CHECK DISK

The CHECK DISK program can be used to make sure a new diskette that has been headered is in fact a good diskette. The program writes to every block to verify its ability to store information and identifies any diskette that contains a bad block. Don't use such diskettes.

DISPLAY T&S

The DISPLAY T&S program allows a programmer to examine the contents of a block by specifying the particular track number and sector number that identifies that block.

PERFORMANCE TEST

The PERFORMANCE TEST program tests the electronic and mechanical capabilities of the disk drive whenever necessary. Use this program whenever you suspect there may be damage to the drive.

SEQ.FILE.DEMO AND REL.FILE.DEMO

These two files are included as programming examples or guidelines when writing your own programs. They also illustrate the important technique of checking the error channel after each access to the disk drive.

SD.BACKUP.xx

These three programs are entitled SD.BACKUP.C64, SD.BACKUP.C16, and SD.BACKUP.PLUS4. Each, when loaded into its respective computer, allows you to create an exact duplicate of a diskette by switching a blank diskette and the diskette to be copied in and out of the drive at the appropriate times. Loading and running them incorrectly may damage a diskette.

PRINT.xx.UTIL

These three programs are actually entitled PRINT.64.UTIL, PRINT.+4, and PRINT.C16.UTIL. They provide two functions: a printout of any Text-Mode screen display, and a listing of the contents of all scaler (non-array) variables in a Basic program to screen or printer. Any CBM printer may be used for either function. Printing of reverse-video and graphics characters depends on the specific printer model used. These programs can run from tape.

C64 BASIC DEMO +4 BASIC DEMO

These programs offer a set of demo routines which can perform minimal system testing on the computers. Routines are provided for testing the video and sound output, keyboard and joy stick input, and disk unit I/O.

LOAD ADDRESS

LOAD ADDRESS is a simple program that tells you where a program was originally located in memory. Some programs can only run in the same locations from which they were saved. Load such programs with LOAD"filename",8,1.

UNSCRATCH

Allows you to restore a file that's been deleted (scratched) from a diskette as long as the diskette hasn't been written to since the scratch was performed.

HEADER CHANGE

Allows you to rename a diskette without losing the data currently stored in the diskette.

IMPORTANT NOTE:
Your Test/Demo diskette may contain additional programs. Commodore may update the diskette from time to time.

ALL ADDITIONS, DELETIONS, OR MODIFICATIONS TO THE PROGRAMS LISTED IN THIS APPENDIX WILL BE NOTED IN THE "HOW TO USE" PROGRAMS ON YOUR TEST/DEMO DISKETTE.

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Ltd.
3470 Pharmacy Avenue • Agincourt, Ontario, M1W 3G3

Commodore is a registered trademark of
Commodore Electronics Limited.

Commodore 1541C is a trademark of
Commodore Electronics Limited.

PRINTED IN TAIWAN

P/N: 354139-01

41A081-001