COMMODORE

1571

DISK DRIVE

user's guide

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 from the U.S. Government Printing Office, Washington, D.C. 20402, stock no. 004-000-00345-4.

FOR USERS IN UK

WARNING: THIS APPARATUS MUST BE EARTHED !

IMPORTANT. The wires in this mains lead are coloured in accordance with the following code:

Green and yellow	:	Earth
Blue	:	Neutral
Brown	:	Live

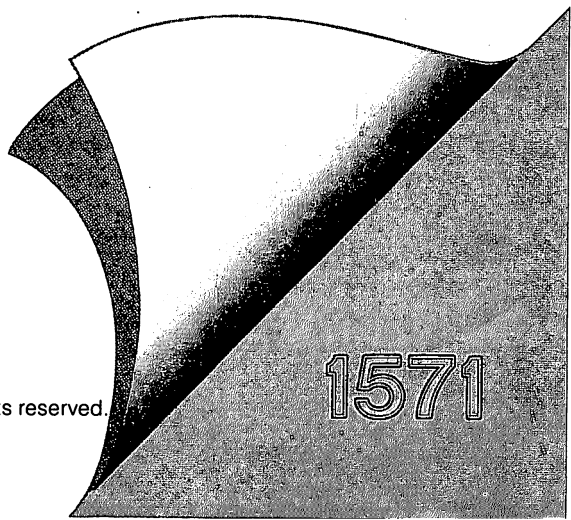
As the colours of the wires in the mains lead of this apparatus may not correspond with the coloured marking identifying the terminals in your plug, proceed as follows:

The wire which is coloured green and yellow must be connected to the terminal in the plug which is marked by the letter E or by the safety earth symbol—or coloured green or green and yellow.

The wire which is coloured blue must be connected to the terminal which is marked with the letter N or coloured black.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L or coloured red.

Disk Drive User's Guide 1571



CONTENTS

INTRODUCTION

MAIN OPERATING FEATURES

The 1571 is a versatile disk drive that handles multiple disk formats and data transfer rates. Disk formats range from single-sided, single-density to double-sided, double-density. The 1571 can be used with a variety of computers, including the Commodore 128, the Commodore 64, the Plus 4, C16, and VIC 20.

When used with the Commodore 128 Personal Computer, the 1571 offers the following features:

- *Standard and fast serial data transfer rates*—The 1571 automatically selects the proper data transfer rate (fast or slow) to match the three operating modes available on the Commodore 128 computer (C128 mode, C64 mode, and CP/M mode).
- *Ability to read and write in double-density MFM format*—This allows access to the CP/M software libraries of other personal computers.
- *Double-sided, double-density data recording*—Provides up to 339K storage capacity per disk (169K per side).
- *Special high-speed burst commands*—These commands, used for machine language programs, transfer data several times faster than the standard or fast serial rates.

When used with the Commodore 64 computer, the 1571 disk drive supports the standard single-density GCR format disks used with the Commodore 1541, 1551, 4040, and 2031 disk drives.

NOTE
CP/M disks are included in Commodore 128 carton; CP/M operating information is presented in the Commodore 128 user manuals.

HOW THIS GUIDE IS ORGANIZED

This guide is divided into two main parts and seven appendices, as described below:

PART ONE: BASIC OPERATING INFORMATION—includes all the information needed by novices and advanced users to set up and begin using the Commodore 1571 disk drive. PART ONE is subdivided into four chapters:

- Chapter 1 tells you how to use disk software programs that you buy, like *Perfect Writer*® and *Jane*®. These pre-written programs help you perform a variety of activities in fields such as business, education, finance, science, and recreation. If you're interested only in loading and running pre-packaged disk programs, you need read no further than this chapter. If you are also interested in saving, loading, and running your own programs, you will want to read the remainder of the guide.
- Chapter 2 describes the use of the BASIC 2.0 disk commands with the Commodore 64 and Commodore 128 computers.

- Chapter 3 describes the use of the BASIC 7.0 disk commands with the Commodore 128.
- Chapter 4 describes the use of the DOS Shell program, which provides you with a convenient alternative way to execute disk drive commands. The DOS Shell is included on the 1571 test/demo diskette supplied with your disk drive.

PART TWO: ADVANCED OPERATION AND PROGRAMMING—is primarily intended for users familiar with computer programming. PART TWO is subdivided into six chapters:

- Chapter 5 discusses the concept of data files, defines *sequential* data files, and describes how sequential data files are created and used on disk.
- Chapter 6 defines the differences between *sequential* and *relative* data files, and describes how relative data files are created and used on disk.
- Chapter 7 describes direct access disk commands as a tool for advanced users and illustrates their use.
- Chapter 8 centers on internal disk commands. Before using these advanced commands, you should know how to program a 6502 chip in machine language and have access to a good memory map of the 1571.
- Chapter 9 provides a list of disk-related kernal ROM subroutines and gives a practical example of their use in a program.
- Chapter 10 gives information on high-speed burst commands.

APPENDICES A THROUGH G—provide various reference information; for example, Appendix A tells you how to set the device number through use of two switches on the back of the drive.

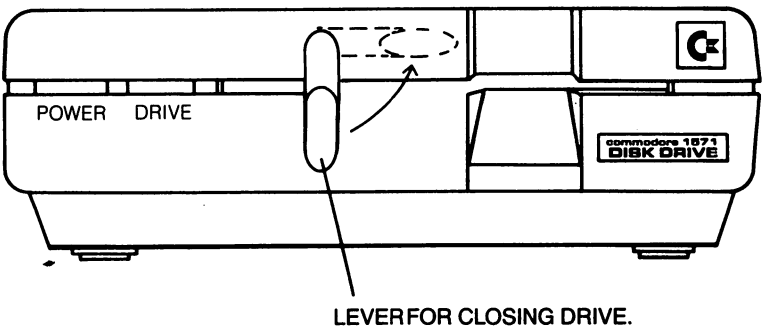
PART ONE:
BASIC OPERATING INFORMATION

CHAPTER 1
HOW TO UNPACK, SET UP AND BEGIN USING THE 1571

STEP-BY-STEP INSTRUCTIONS

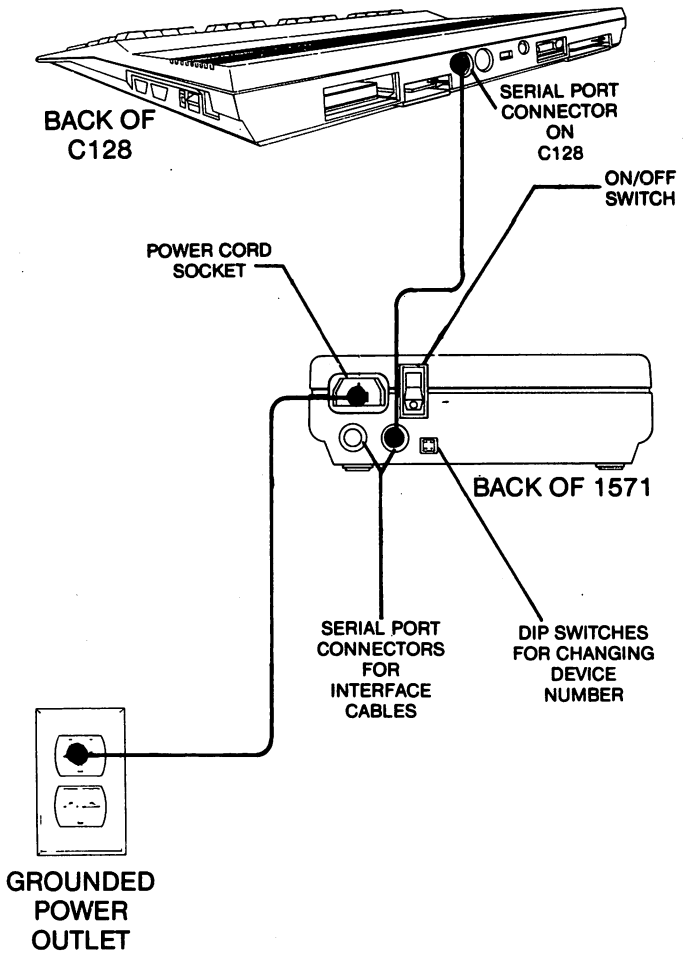
1. Inspect the shipping carton for damage.
   If you find any damage to the shipping carton and suspect that the disk drive may have been affected, contact your dealer.
2. Check the contents of the shipping carton.
   Packed with the 1571 and this book, you should find the following: 3-prong electrical power cord, interface cable, Test/Demo diskette, and a warranty card to be filled out and returned to Commodore.
3. Remove the cardboard shipping spacer from the disk drive.
   The spacer is there to protect the inside of the drive during shipping. To remove it, rotate the lever on the front of the drive counter-clockwise (see Figure 1) and pull out the spacer.

Figure 1. Front of 1571 Disk Drive



4. Connect the power cord.
   Check the ON/OFF switch on the back of the drive (see Figure 2) and make sure it's OFF. Connect the cord where indicated in Figure 2. Plug the other end into a grounded (3-prong) outlet. Don't turn the power on yet.

Figure 2. Connection of Power Cord and Interface Cables to 1571



5. Connect the interface cable.

   Make sure your computer and any other peripherals are OFF. Plug either end of the interface cable into either serial port on the back of the drive (see Figure 2). Plug the other end of the cable into the back of the computer. If you have another peripheral (printer or extra drive), plug its interface cable into the remaining serial port on the drive.

6. Turn ON the power.

   With everything hooked up and the drive empty, you can turn on the power to the peripherals in any order, but turn on the power to the computer last. When everything is on, the drive goes through a self test. If all is well, the green light will flash once and the red power-on light will glow continuously. If the green light continues to flash, there may be a problem. Refer to the Troubleshooting Guide.

OPERATING MODES

To achieve maximum compatability between the 1571 and other disk drives, the 1571 has two modes of operation that allow it to be used in a variety of situations.

1541 Mode

When you first turn on the 1571, it powers up in 1541 mode. That is, it works almost exactly as a 1541 disk drive, which helps maintain compatability with certain special programs and copy-protected diskettes. Also, many loading routines are dependent upon the 1541-speed internal timing. However, when the drive is in this mode, it can't take advantage of the special 1571 features.

1571 Mode

In 1571 mode, the drive can take advantage of such features as double-sided diskettes, MFM-format diskettes, fast data transfer rates, and burst transfer protocol. The 1571 can determine whether or not the host computer can receive fast data transfer and transmits accordingly.

Mode Selection

As mentioned above, the 1571 powers up in 1541 mode. However, it automatically shifts to 1571 mode upon the first of a fast serial transfer initiated by the host (the Commodore 128). The Commodore 64 and Plus 4 can't perform fast serial transfers on the serial bus, but you can shift from 1541 to 1571 mode by sending the following BASIC command: OPEN 1,8,15,"U0>M1". Although these computers can't directly take advantage of the drive's faster transfer speeds, they can have twice the normal storage.

If you are using a Commodore 128 and want the 1571 to remain in 1541 mode, you can send the following BASIC command: OPEN 1,8,15,"U0>M0". The drive will then remain in 1541 mode until the next power up or system reset.

The possible combinations of modes with the Commodore 128 and 1571 are shown below:

C128/1571 modes—Turn on the drive, then turn on or reset the computer.

C128/1541 modes—Turn on or reset the computer, and when the cursor appears, turn on the drive. In this case, the first disk access will switch the drive to 1571 mode, so send the BASIC command given on the previous page to lock the drive into 1541 mode.

C64/1571 modes —Turn on the drive, then turn on or reset the computer and type: GO 64

C64/1541 modes —Turn on the drive, then hold down the COMMODORE key while you turn on or reset the computer.

TROUBLESHOOTING GUIDE

Problem	Possible Cause	Solution
Red power-on indicator not lit	Power not ON	Make sure ON/OFF switch is ON
	Power cable not plugged in	Check both ends of power cable to be sure they are fully inserted
	Power off to wall outlet	Replace fuse or reset circuit breaker in house
Green drive light flashing	Drive failing its self test	Turn the system off for a moment then try again. If the light still flashes, turn the drive off and on again with the interface cable disconnected. If the problem persists, contact your dealer. If unplugging the interface cable made a difference, make sure the cable is properly connected. If that doesn't work, the problem is probably in the cable itself or somewhere else in the system
Programs won't load and the computer says "DEVICE NOT PRESENT ERROR"	Interface cable not well connected or drive not ON	Be sure the cable is properly connected and the drive is ON
	Switches on back of drive may not be set for correct device number	Check Appendix A for correct setting to match LOAD command
Programs won't load, but the computer and disk drive give no error message	Another part of the system may be interfering	Unplug all other machines on the computer. If that cures it, plug them in one at a time. The one just added when the trouble repeats is most likely the problem
		Trying to load a machine language program into BASIC space will cause this problem

TROUBLESHOOTING GUIDE (continued)

Problem	Possible Cause	Solution
Programs won't load and green drive light flashes	Disk error	Check the error channel to determine the error, then follow the advice in Appendix B to correct it. The error channel is explained in Chapters 2 and 3
(Be sure to spell program names correctly and include the exact punctuation when loading the programs)		
Your programs load OK, but commercial programs and those from other 1571s don't	Either the diskette is faulty, or your disk drive is misaligned	Try another copy of the program. If several programs from several sources fail to load, have your dealer align your disk drive
Your programs that used to load, won't anymore, but programs saved on newly-formatted diskettes will	Older diskettes have been damaged	See the safety tips for diskettes in the next section. Recopy from backups
	The disk drive has gone out of alignment	Have your dealer align your disk drive

SIMPLE MAINTENANCE TIPS

1. Keep the drive well ventilated.
   A couple of inches of space to allow air circulation on all sides will prevent heat from building up inside the drive.
2. Use Commodore diskettes.
   Badly-made diskettes can cause increased wear on the drive's read/write head. If you're using a diskette that is unusually noisy, it could be causing added wear and should be replaced.
3. The 1571 should be cleaned once a year in normal use.
   Several items are likely to need attention: the two read/write heads may need 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. Since these chores require special materials or parts, it is best to leave the work to an authorized Commodore service center. If you want to do the work yourself, ask your dealer for the appropriate materials. IMPORTANT: Home repair of the 1571 will void your warranty.

INSERTING A DISKETTE

To insert a diskette, first open the drive door by rotating the door lever counter-clockwise one quarter turn until it stops, with the lever parallel to the horizontal 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 (See Figure 3). 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. You shouldn't have to force or bend it to get it in.*

NOTE: When the write/protect notch is covered by tape, the contents of the diskette cannot be altered or added to. That prevents accidental erasing of information you want to preserve. If a diskette comes without a write/protect notch, the contents of that diskette were not meant to be altered.

Blank diskettes may not have a label on them when you purchase them.

With the diskette in position, seat it properly for use by twisting the door lever clockwise one-quarter turn until it stops vertically over the slot. *Warning:* If it doesn't 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.

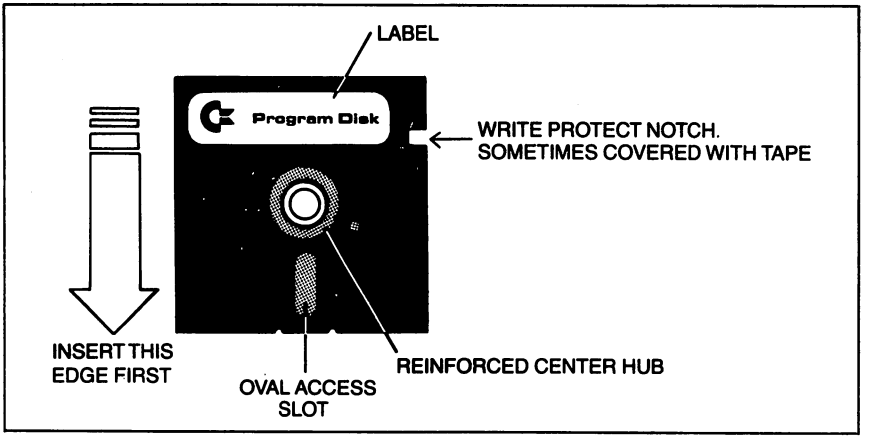


Figure 3. Inserting a Diskette

DISKETTE CARE

1. Don't touch the exposed parts of a diskette.
   That includes the access slot and the center hub (the hole in the middle).
2. Don't bend a diskette.
   They're called floppy diskettes, but they're not supposed to be flopped.
3. Keep the diskettes (and disk drive) away from magnets.
   That includes the electromagnets in telephones, televisions, desk lamps, and calculator cords. Keep smoke, moisture, dust, and food off the diskettes. Store diskettes upright in their paper jackets.
4. Buy diskettes with reinforced hubs.
   Although the drive usually centers a diskette correctly, it would be difficult to rescue data from a diskette recorded with its hub off-center. Reinforced hubs make it easier for the drive to center a diskette.
5. Remove a diskette before turning the drive off.
   If you don't, you might lose part or all the data on the diskette.
6. Don't remove a diskette from the drive while the green light is glowing.
   That light glows when the drive is in use. If you remove the diskette then, you might lose the information currently being written to the diskette.

USING PRE-PROGRAMMED (SOFTWARE) DISKETTES

Your software user's guide should list the procedure for loading the program into your computer. Nevertheless, we've included the following procedure as a general guide. You'll also use this procedure to load programs or files from your own diskettes. For purposes of demonstration, use the Test/Demo diskette included with the disk drive.

1. Turn on system.
2. Insert diskette.
3. If you are using a VIC 20, Commodore 64, or a Commodore 128 computer in C64 mode, type: LOAD "HOW TO USE",8
   If you are using a Plus/4 or Commodore 128 in C128 mode, type: DLOAD "HOW TO USE"
4. Press the RETURN key.
5. The following will then appear on the screen:

   SEARCHING FOR 0:HOW TO USE
   LOADING

   READY

6. Type: RUN
7. Press the RETURN key.

To load a different program or file, simply substitute its name in place of HOW TO USE inside the quotation marks. NOTE: The HOW TO USE program is the key to the Test/Demo diskette. When you LOAD and RUN it, it provides instructions for using the rest of the programs on the diskette. To find out what programs are on your Test/Demo diskette, refer to the section entitled "DIRECTORIES" later in this chapter.

If a program doesn't load or run properly using the above method, it may be that it is a machine language program. But unless you'll be doing advanced programming, you need not know anything about machine language. A program's user's guide should tell you if it is written in machine language. If it is, or if you are having trouble loading a particular program, simply add a ,1 (comma and number 1) at the end of the command.

IMPORTANT NOTE

Throughout this manual, when the format for a command is given, it will follow a particular style. Anything that is capitalized must be typed in exactly as it is shown (these commands are listed in capital letters for style purposes, DO NOT use the SHIFT key when entering these commands). Anything in lower case is more or less a definition of what belongs there. Anything in brackets is optional.

For instance, in the format for the HEADER command given on the following page, the word HEADER, the capital I in Iid, the capital D in Ddrive#, and the capital U in Udevice# must all be typed in as is (Ddrive# and Udevice# are optional).

On the other hand, diskette name tells you that you must enter a name for the diskette, but it is up to you to decide what that name will be. Also, the id in Iid is left to your discretion, as is the device# in Udevice#. The drive# in Ddrive# is always 0 on the 1571, but could be 0 or 1 on a dual disk drive. Be aware, however, that there are certain limits placed on what you can use. In each case, those limits are explained immediately following the format (for instance, the diskette name cannot be more than sixteen characters and the device# is usually 8).

Also be sure to type in all punctuation exactly where and how it is shown in the format.

Finally, press the RETURN key at the end of each command.

HOW TO PREPARE A NEW DISKETTE

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 you can use the HEADER command or the NEW command to add it to a diskette. That is known as formatting the disk. This is the command to use with the C128 in C128 mode or Plus/4:

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

Where:
"diskette name" is any desired name for the diskette, up to 16 characters long (including spaces). "id" can be any two characters as long as they don't form a BASIC keyword (such as IF or ON) either on their own or with the capital I before them. "drive#" is 0 . "device#" is 8, unless you have changed it as per instructions in Appendix A (the 1571 assumes 8 even if you don't type it in).

The command for the C64, VIC 20, or C128 in C64 mode is this:

OPEN 15,device#,15,"NEWdrive#:diskette name,id"
CLOSE 15

The device#, drive#, diskette name, and id are the same as described above. The OPEN command is explained in the next chapter. For now, just copy it as is.

NOTE TO ADVANCED USERS

If you want to use variables for the diskette name or id, the format is as follows:

C128, Plus/4: HEADER (A$),I(B$),D0

C64: OPEN 15,8,15:PRINT#15,"N0:"+A$+B$:CLOSE15

Where:

A$ contains the diskette name (16 character limit)
B$ contains the id (2 characters long)

After you format a particular diskette, you can reformat it at any time. You can change its name and erase its files faster by omitting the id number in the HEADER command.

DISKETTE DIRECTORY

A directory is a list of the files on a diskette. To load the directory on the C128 or Plus/4, type the word DIRECTORY on a blank line and press the RETURN key or simply press the F3 key on the C128. That doesn't erase anything in memory, so you can call up a directory anytime—even from within a program. The C64 directory command, LOAD "$",8 (press RETURN) LIST (press RETURN), does erase what's in memory.

If a directory doesn't all fit on the screen, it will scroll up until it reaches the last line. If you want to pause, stop, or slow down the scrolling, refer to your particular computer's user's manual for instructions as to which keys to use.

To get an idea of what a directory looks like, load the directory from the Test/Demo diskette.

The 0 on the left-hand side of the top line is the drive number of the 1571 (on a dual disk drive it could be 0 or 1). The diskette name is next, followed by the diskette id—both of which are determined when the diskette is formatted.

The 2A at the end of the top line means the 1571 uses Version 2A of Commodore's disk operating system (DOS).

Each of the remaining lines provides three pieces of information about the files on the diskette. At the left end of each line is the size of the file in blocks of 254 characters. Four blocks are equivalent to almost 1K of memory inside the computer. The middle of the line contains the name of the file enclosed in quotation marks. All characters within the quotation marks are part of the filename. The right side of each line contains a three-letter abbreviation of the file type. The types of files are described in later chapters.

TYPES OF FILES

PRG—Program
SEQ—Sequential
REL—Relative
USR—User
DEL—Deleted (you won't see this type)

Note: Direct Access files, also called Random files, do not automatically appear in the directory. They are covered in Chapter 7.

The bottom line of a directory shows how many blocks are available for use. This number ranges from 664 (in 1541 mode) and 1328 (in 1571 mode) on a newly formatted diskette to 0 on one that is completely full.

SELECTIVE DIRECTORIES

By altering the directory LOAD command, you can create a kind of "sub-directory" that lists a single selected type of file. For example, you could request a list of all sequential data files (Chapter 5), or one of all the relative data files (Chapter 6). The format for this command is:

LOAD"$0:pattern=filetype",8 (for the C64)

where pattern specifies a particular group of files, and filetype is the one-letter abbreviation for the types of files listed below:

P = Program
S = Sequential
R = Relative
U = User

The command for the C128 and Plus/4 is this: DIRECTORY"pattern=filetype"

Some examples:

LOAD"$0:*=R",8 and DIRECTORY"*=R" display all relative files.

LOAD"$0:Z*=R",8 and DIRECTORY"Z*=R" display a sub-directory consisting of all relative files that start with the letter Z (the asterisk (*) is explained in the section entitled "Pattern Matching."

PRINTING A DIRECTORY

To get a printout of a directory, use the following:

```
LOAD"$",8
OPEN4,4:CMD4:LIST
PRINT#4:CLOSE4
```

PATTERN MATCHING

You can use special pattern-matching characters to load a program from a partial name or to provide the selective directories described earlier.

The two 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 makes only its own 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

DLOAD"SM?TH" loads the first file that starts with SM, ends with TH, and one other character between

DIRECTORY"Q*" loads a directory of files whose names begin with Q

LOAD"*",8 is a special case. When an asterisk is used alone as a name, it matches the last file used (on the C64 and C128 in C64 mode).

LOAD "0:*",8 loads the first file on the diskette (C64 and C128 in C64 mode).

DLOAD "*" loads the first file on the diskette (Plus/4 and C128 in C128 mode).

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 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 or COLLECT commands. One of these should normally be used whenever a splat file is noticed on a diskette. All of these commands are described in the following chapters.

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 (Chapter 7). 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 described in the next chapter.

CHAPTER 2
BASIC 2.0 COMMANDS

This chapter describes the disk commands used with the VIC 20, Commodore 64 or the Commodore 128 computer in C64 mode. These are Basic 2.0 commands.

You send command data to the drive through something called the command channel. The first step is to open the channel with the following command:

```
OPEN15,8,15
```

The first 15 is a file number or channel number. Although it could be any number from 1 to 255, we'll use 15 because it is used to match the secondary address of 15, which is the address of the command channel. The middle number is the primary address, better known as the device number. It is usually 8, unless you change it (see Appendix A).

Once the channel has been opened, use the PRINT# command to send information to the disk drive and the INPUT# command to receive information from the drive. You must close the channel with the CLOSE15 command.

The following examples show the use of the command channel to NEW an unformatted disk:

```
OPEN15,8,15
PRINT#15,"NEWdrive#:diskname,id"
CLOSE15
```

You can combine the first two statements and abbreviate the NEW command like this:

```
OPEN15,8,15,"Ndrive# :diskname,id"
```

If the command channel is already open, you must use the following format (trying to open a channel that is already open results in a "FILE OPEN" error):

```
PRINT#15,"Ndrive#:diskname,id"
```

ERROR CHECKING

In Basic 2.0, when the green drive light flashes, you must write a small program to find out what the error is. This causes you to lose any program variables already in memory. The following is the error check program:

```
10 OPEN15,8,15
20 INPUT#15,EN,EM$,ET,ES
30 PRINT EN, EM$,ET,ES
40 CLOSE15
```

This little program reads the error channel into four BASIC variables (described below), and prints the results on the screen. A message is displayed whether there is an

error or not, but if there was an error, the program clears it from disk memory and turns 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.

For those of you who are writing programs, the following is a small error-checking subroutine you can include in your programs:

```
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
```

This assumes file 15 was opened earlier in the program, and that it will be closed at the end of the program.

The subroutine reads the error channel and puts the results into the named variables—EN (Error Number), EM$ (Error Message), ET (Error Track), and ES (Error Sector). Of the four, only EM$ has to be a string. You could choose other variable names, although these have become traditional for this use.

Two error numbers are harmless—0 means everything is OK, and 1 tells how many files were erased by a SCRATCH command (described later in this chapter). If the error status is anything else, line 60000 prints the error message and halts the program.

Because this is a subroutine, you access it with the BASIC GOSUB command, either in immediate mode or from a program. The RETURN statement in line 60010 will jump back to immediate mode or the next statement in your program, whichever is appropriate.

BASIC 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. By opening it once at the start, the file is open whenever needed for disk commands elsewhere in the program.

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:

```
CLOSE 15:OPEN 15,8,15,"I":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 3.0 1571" 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. DOS is Disk Operating System.

Hint #4: To reset drive, type: OPEN 15,8,15,"UJ":CLOSE 15. This also applies to sending a UI+ or a UI.

SAVE

Use the SAVE command to preserve a program or file on a diskette for later use.

Before you can SAVE to diskette, the diskette must be formatted, as described earlier.

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 and a colon, and followed by the device number of the disk, normally 8.

However, the SAVE command will not work in copying programs that are not in the BASIC text area, such as "DOS 5.1" for the C64. To copy it and similar machine-language programs, you will need a machine-language monitor program.

FORMAT FOR A MONITOR SAVE

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

where "drive #:" is the drive number, 0 on the 1571; "file name" is any valid file name up to 14 characters long (leaving two for the drive number and colon); "device #" is a two digit device number, normally 08 (the leading 0 is required); and the addresses to be saved are given in Hexadecimal but without a leading dollar sign ($). Note the ending address listed must be one 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,D000

SAVE WITH REPLACE OPTION

If a file already exists, it can't be saved again with the same name because the disk drive 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 drive to replace any file it finds in the diskette directory with the same name, substituting the new file for the old version.

FORMAT FOR SAVE WITH REPLACE:

SAVE"@Drive #:file name", device #

where all the parameters are as usual except for adding a leading "at" sign (@.) The "drive #:" is required here.

EXAMPLE:

SAVE"@0:REVISED PROGRAM",8

The actual procedure is that the new version is saved completely, then the old version is erased. Because it works this way, there is little danger 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.

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 as there is still room for. After the command executes, a look at the 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 below) will not detect this problem, because whatever was saved will have been saved properly.

VERIFY

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 doesn't mean either copy is bad, but if they were supposed to be identical, there is 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 diskette.

FORMAT FOR THE VERIFY COMMAND:

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

where "drive#:" is an optional drive number, "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 files 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 BASIC 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 VERIFY files saved from the same type of machine, and identical memory size. For example, a BASIC program saved from a Plus/4 can't be verified easily with a C64, even when the program would work fine on both machines. 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 can only be done on the same machine and memory size as the one on which the files were first created.

SCRATCH

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 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, 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. 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.

MORE ABOUT SCRATCH

SCRATCH is a powerful command and should be used with caution to be sure you delete only 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. There may be an "Unscratch" program on your Test/Demo diskette.

> **NOTE:** If you accidentally SCRATCH a file within the DOS Shell (see Chapter 4), you can unscratch it with the Shell's RESTORE FILES function.

More about Splats

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 or files and cause permanent damage to those 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 checked immediately.

Locked Files

Occasionally, a diskette will contain a locked file; 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.

RENAME

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."

RENAMING AND SCRATCHING TROUBLESOME FILES (ADVANCED USERS)

Eventually, you may run across a file which has an odd filename, such as a comma by itself (","") or one that includes a Shifted Space (a Shifted Space looks the same as a regular space, but if a file with a space in its name won't load properly and all else is correct, it's probably 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 Shifted Space in their name can also be troublesome, because the disk interprets the Shifted Space as signaling 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 be useful by allowing you to have a long file name, and making 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" using the CHR$() equivalent of a quotation mark in the RENAME command:

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

The CHR$(34) forces a quotation mark into the command string without upsetting BASIC. The procedure for a file name that includes a Shifted Space is similar, but uses CHR$(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:

```
PRINT#15,"S0:?"
```

This example deletes all files with one character names.

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.

In some cases, it may be easier to copy desired files to a different diskette and leave the troublesome files behind.

COPY

The COPY command allows you to make a spare copy of any program or file on a diskette. On a single drive like the 1571, 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 combine up

to four sequential data files (linking the files 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 linked.

FORMAT FOR THE COPY COMMAND

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

EXAMPLES:

PRINT#15,"COPY0:BACKUP=ORIGINAL"

or abbreviated as

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

PRINT#15,"C0:BACKUP=ORIGINAL"

where "drive #" is the drive number "new file" is the copy and "old file" is the original.

FORMAT FOR THE COMBINE OPTION

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

where "drive #" is always 0,

NOTE: The length of a command string (command and filenames) is limited to 41 characters.

EXAMPLES:

After renaming a file named "BOOT" to "TEMP" in the last section's example, you 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 fit easily in memory along with a program we are using, you can use the concatenate option to combine them in a master file, even if the result is too big to fit in memory. (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 1571, check your Test/ Demo diskette to find the programs that you need or use the DOS Shell described in Chapter 4.

VALIDATE

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. This brief 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 (in 1541 mode) or 1328 (in 1571 mode) 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, 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. As usual, it is assumed file 15 has been opened to the command channel and will be closed after the command has been executed.

EXAMPLE:

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

INITIALIZE

One command that should not often be needed on the 1571, but is still of occasional value is INITIALIZE. On the 1571, 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 into a disk buffer. This information must 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.

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.

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 15
30 PRINT#15,"I0"
40 NEXT I
50 CLOSE 15
```

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

CHAPTER 3
BASIC 7.0 COMMANDS

This chapter describes the disk commands used with the Commodore 128 computer (in C128 mode). This is BASIC 7.0, which includes BASIC 2.0, BASIC 3.5, and BASIC 4.0 commands, all of which can be used.

ERROR CHECKING

When the drive light (green light) flashes, you must use the following command to find out what the error is:

```
PRINT  DS$
```

A message is displayed whether there is an error or not. If there was an error, this command clears it from disk memory and turns 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.

For those of you who are writing programs, the following is a small error-checking subroutine you can include in your programs:

```
59990  REM READ ERROR CHANNEL
60000  IF DS>1 THEN PRINT DS$:STOP
60010  RETURN
```

The subroutine reads the error channel and puts the results into the reserved variables DS and DS$. They are updated automatically by BASIC.

Two error numbers are harmless—0 means everything is OK, and 1 tells how many files were erased by a SCRATCH command (described later in this chapter). If the error status is anything else, line 60000 prints the error message and halts the program.

Because this is a subroutine, you access it with the BASIC GOSUB command, either in immediate mode or from a program. The RETURN statement in line 60010 will jump back to immediate mode or the next statement in your program, whichever is appropriate.

SAVE

This command will save a program or file so you can reuse it. The diskette must be formatted before you can save it to that diskette.

FORMAT FOR THE SAVE COMMAND

```
DSAVE "file name" [,Ddrive#] [,Udevice#]
```

This command will not work in copying programs that are not written in BASIC. To copy these machine language programs, you can use the BSAVE command or the built-in Monitor S command.

FORMAT FOR THE BSAVE COMMAND

BSAVE "file name" [,Ddrive#] [,Udevice#] [Bbank#]
[,Pstarting address] [TO Pending address + 1]

where the usual options are the same and bank# is one of the 16 banks of the C128. The addresses to be saved are given in decimal. Note that the ending address must be 1 location beyond the last location to be saved.

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 1571; "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 1571 (the leading 0 is required); and the addresses to be saved are given in Hexadecimal (base 16,) but without a leading dollar sign (for the Plus/4). On the C128, the addresses need not be in Hexidecimal. Note that the ending address listed must be 1 location beyond the last location to be saved.

SAVE WITH REPLACE

If a file already exists, it can't be saved again with the same name because the disk drive allows only 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 in this chapter. 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 drive to replace any file it finds in the diskette directory with the same name, substituting the new file for the old version.

FORMAT FOR SAVE WITH REPLACE

DSAVE "@file name" [,Ddrive#] [,Udevice#]

The actual procedure is this—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 danger 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.

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 the second copy, only as much of the new

version will be saved as there is still room for. After the command is completed, a look at a directory will show the new version is present, but doesn't occupy enough blocks to match the copy in memory.

DVERIFY

This command makes a byte-by-byte comparison of the program currently in memory against a program on diskette. This comparison includes the BASIC line links, which may be different for different types of memory configurations. What this means is that a program saved to disk on a C64 and reloaded into a C128 wouldn't verify properly because the line links point to different memory locations. If the disk copy of the program differs at all from the copy in memory, a "VERIFY ERROR" will be displayed. This doesn't mean either copy is bad, but if they were supposed to be identical, there is a problem.

FORMAT FOR THE DVERIFY COMMAND

DVERIFY "file name" [,Ddrive#] [,Udevice#]

The following version of the command verifies a file that was just saved:

DVERIFY "*"

This command 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 DVERIFY after @SAVE, include at least part of the file name that is to be verified in the pattern.

COPY

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 1571, 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 can be copied or concatenated.

FORMAT FOR THE COPY COMMAND

COPY [Ddrive#,] "old file name" TO [Ddrive#,] "new file name" [,Udevice#]

Where both drive#s would be 0 if included.

NOTE: If you want to copy a file from one diskette to another, you cannot use the COPY command. Instead, use the copy program on the Test/Demo diskette or the DOS Shell (see Chapter 4).

CONCAT

The CONCAT command allows you to concatenate (combine) two sequential files.

FORMAT FOR THE CONCAT COMMAND

CONCAT [Ddrive#,] "add file" TO [Ddrive#,] "master file" [,Udevice#]

Where the optional drive# would be 0 in both cases. The old "master file" is deleted and replaced with a new "master file" which is the concatenation of the old "master file" and "add file".

NOTE: The length of a command string (command and filenames) is limited to 41 characters.

SCRATCH

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.

FORMAT FOR THE SCRATCH COMMAND

SCRATCH "pattern" [,Ddrive#] [,Udevice#]

Where, "pattern" is any valid file name or pattern-matching character.

You will be asked as a precaution:

ARE YOU SURE? ■

If you ARE sure, simply press Y and RETURN. If not, press RETURN alone or type any other answer, and the command will be canceled.

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.

You can perform a SCRATCH within a program, but there will be no prompt message displayed.

MORE ABOUT SCRATCH

SCRATCH is a powerful command and should be used with caution to be sure you delete only 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. There may be an "Unscratch" program on your Test/Demo diskette.

> **NOTE:** If you accidentally SCRATCH a file within the DOS Shell (see Chapter 4), you can unscratch it with the Shell's RESTORE FILES function.

More about Splat Files

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 or files, and cause permanent damage to those other programs or files 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 COLLECT 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 checked immediately.

Locked Files

Occasionally, a diskette will contain a locked file; 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.

RENAME

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. If you try to RENAME a file by using a file name already in the directory, the computer will respond with a "FILE EXISTS" error. A file must be properly closed before it can be renamed.

FORMAT FOR RENAME COMMAND:

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

where both drive#s, if included, would be 0

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 Shifted Space in their name can also be troublesome, because the disk interprets the Shifted Space as signaling 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 be useful by allowing you to have a long file name, and making 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 discussed for 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" using the CHR$() equivalent of a quotation mark in the RENAME command:

Example:

```
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:

Example:

```
SCRATCH"?"
```

This example deletes all files with one-character names.

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.

In some cases, it may be easier to copy desired files to a different diskette and leave the troublesome files behind.

COLLECT

The COLLECT 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 reuse, and all improperly closed files are automatically scratched. However, this brief description of COLLECT doesn't indicate either the power or the danger of the 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 (in 1541 mode) or 1328 (in 1571 mode) 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 concerned about their integrity. Just note the number of blocks free in the diskette's directory before and after using COLLECT. 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 unallocating 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#]

INITIALIZE

There is no BASIC 7.0 command for initializing, so refer to Chapter 2 for the BASIC 2.0 INITIALIZE command.

CHAPTER 4
DOS SHELL

The DOS Shell is a program that provides you with an alternate way to execute disk drive commands. Prompting messages take you step-by-step through each operation, making the use of the 1571 easier and more understandable.

The Shell is automatically loaded when you insert the diskette, then turn on or reset the computer. You can enter or exit the Shell by pressing the F1 key.

LANGUAGE SELECTION

The Shell displays its messages in any one of four languages: English, French, German, and Italian. When you enter the Shell, each language appears, in turn, on the screen for about six seconds. To choose one, press the space bar when it appears on the screen. You can skip to the next language without waiting the six seconds by pressing the CRSR-down key.

> **NOTE:** The DOS Shell can be used in 40-column or 80-column mode. The only difference is a change in the layout of the screen. You can switch the screen size by first pressing the ESC key and then pressing the X key. This must be done before you select a function from the primary menu screen (see below).

PRIMARY MENU SCREEN

After a language is chosen, the primary menu screen appears. To choose a function, use the CRSR keys to position the cursor on that function then press the SPACE BAR.

DISK/PRINTER SETUP

Within the Shell there are two drives: A and B. They are called logical drives because they reside only within the Shell. You can set them to represent a single disk drive unit, two single units, or a dual drive unit. Assuming you have a single unit, the usual settings for A and B would show a device# of 8 and a drive# of 0. Those are the default settings (in other words, when you first enter the Shell, the Disk Setup is such that it represents a single unit). With two single units, you would change the device# of one of the drives. The drive#s can stay the same. With a dual drive unit, you would change the drive# of one of the drives.

To change a setting, use the CRSR keys to position the cursor over the desired value. After you have changed any settings, press the F7 key to lock them in. If you change your mind, press the STOP key instead of F7 and the previous settings are left unchanged.

The printer device# is selected in the same way.

To change the device number of the actual drive (the physical drive), position the cursor on the 'CHANGE DISK DEVICE #' line and press the SPACE BAR. A message then appears on the screen guiding you through the steps to change the device#. If you change your mind before completing the steps, press the STOP or F5 keys to cancel the device# change. F5 returns you to the logical drive selections and STOP returns you to the primary menu screen.

Changing the device# locks in all current parameters in this function.

NOTE: Unless the Disk Setup is set for a single drive unit, when you choose any of the following functions, a question appears on the screen asking you which drive you want to use. In describing these functions, we'll assume that you have a single disk. If you have more than one, then you'll have to provide the appropriate answer to that question through the use of the CRSR keys and SPACE BAR.

RUN A PROGRAM

This function automatically loads and runs a program file that you choose from a file list shown on the screen (for this function the file list contains only program files). Use the CRSR keys to position the cursor over the desired file, then press the SPACE BAR to choose that file. If you change your mind, press the F5 key to un-select the file. Press the F7 key to load and run it. If you press the stop key before the loading is complete, the function is cancelled.

You can print out the file list, whenever it's displayed for a function, by pressing the F3 key. Make sure your printer is turned on and the paper is at the top of a page.

Note that some machine language programs won't run in the Shell.

FORMAT A DISK

When you select this function, a message appears on the screen telling you to insert a blank disk into the drive. After you do that, press the SPACE BAR and the Shell checks to see if the disk has already been formatted. If it has, the disk header information is displayed on the screen as a precaution that there may be files on the disk. To proceed with formatting, either enter a new disk name or accept the old one and press RETURN. If the disk was previously formatted, it retains its old ID code. If not, the Shell randomly generates one. You can enter your own ID code by entering a comma and two-digit ID code as the last three characters of the disk name.

CLEANUP A DISK

This function enables you to validate the Block Availability Map of a disk (see the VALIDATE command in chapter 2 or the COLLECT command in chapter 3).

COPY A DISK

Use this function to make backups of your diskettes. The disk drive setup (drives A and B) determines what type of copy is performed: single unit, dual drive, or two single units.

Dual Drive or Two Unit Copy

The Shell asks you from which drive you want to copy (use CRSR keys and space bar to select A or B). Insert the original disk into that drive and the copy disk into the other drive and press the SPACE BAR. Press the STOP key to cancel this function. The F5 key or SPACE BAR can be used to retry a disk operation if an error occurs.

Single Drive Copy

The Shell displays the following warning message: 'PROGRAMS IN MEMORY WILL BE DESTROYED'. Press the STOP or F5 keys to cancel the function at this point. Press the SPACE BAR to continue.

Throughout this function you will alternately insert the original and copy disks according to prompts from the Shell. It takes from one to four swaps to complete the copy. The first time you insert the copy disk, it is formatted with the same disk name as the original disk, but a different ID code is generated. Then, each time you insert a disk, its ID code is checked to make sure that it's the correct disk. If it's not, an error message is displayed on the screen. If that happens, press the SPACE BAR or the F5 key to return to the last-used disk swapping prompt.

If the STOP key is pressed, the function is cancelled and the copy will be incomplete.

COPY FILES

This function makes copies of selected files, the type of copy is determined by the Disk Drive Setup. Once you select the drive from which the files will be copied, the file list is displayed so you can select the files using the CRSR keys and SPACE BAR. You can "scroll" the list up or down using the CRSR keys.

Once you position the cursor next to a file you want to copy, press the SPACE BAR. You can 'unselect' any file(s) by positioning the cursor on the filename and pressing the F5 key.

Press the F7 key when you've completed file selection. The Shell then asks you if it is "OK TO COPY FILE-LIST: N Y". If you answer Yes, the function continues. If you answer No (the default answer), the Shell returns to file selection with all previous selections intact.

Press the F5 key to restart file selection with all previous selections cancelled.

Press the STOP key to cancel the function and return to the primary menu.

Dual Drive or Two Unit Copy

After you answer the 'OK TO COPY' question, the messages 'INSERT COPY DISK INTO DRIVE: X' and 'THEN PRESS SPACE' are displayed. After you do that, the message 'WANT TO FORMAT THE COPY DISK: N Y' is displayed (unless, however, the copy disk has never been formatted, in which case the Shell automatically does it without asking you). The default answer is No. If you answer Yes, the 'FORMAT A

DISK' function begins at the point where you are asked to enter a new disk name for formatting.

Whether you answer Yes or No to the formatting question, the Shell checks the available space on the copy disk against the total size of the selected file(s). If the copy disk can't hold all the files, a warning is displayed and you can press the STOP key to cancel the function or press the SPACE BAR to continue.

NOTE: For any type of file copy, if the next file to be copied won't fit onto the copy disk, the following message is displayed: 'COPY DISK FULL—ANOTHER DISK: Y N'. If the answer is No (the default answer), the copy is terminated. If Yes, the function returns to the 'INSERT COPY DISK' message.

Single Drive Copy

After you answer the 'OK TO COPY' question, the following warning is displayed on the screen: 'PROGRAMS IN MEMORY WILL BE DESTROYED,' along with the instruction 'PRESS SPACE TO CONTINUE'. Press the F5 or STOP keys to cancel the function and retain any program in memory.

You will alternately insert the original disk and the copy disk according to prompts on the screen. It takes from one to four disk swaps to complete the copy.

The first time the copy disk is inserted, the 'WANT TO FORMAT THE COPY DISK: N Y' message is displayed (unless, however, the copy disk has never been formatted, in which case the Shell automatically does it without asking you). Again, whether the answer is Yes or No, the Shell checks the available space on the copy disk against the total size of the file(s) to be copied and alerts you if there isn't enough room on the copy disk.

DELETE FILES

This function erases one or more files. Use the CRSR keys and SPACE BAR to select the file(s) from the file list displayed on the screen. Press the F7 key when you've completed file selection. The Shell then asks you if it is "OK TO DELETE FILE-LIST: N Y". If you answer Yes, the function continues. If you answer No (the default answer), the Shell returns to file selection with all previous selections intact.

Press the F5 key to restart file selection with all previous selections cancelled.

Press the STOP key to cancel the function and return to the primary menu.

RESTORE FILES

This function restores one or more files that have been deleted. A file list of only deleted files is displayed on the screen and you again use the the CRSR keys and SPACE BAR to select the file(s) to be restored.

After you select a file, the Shell checks to see if it can be restored without corrupting other files. If not, the message 'CANNOT RESTORE FILE: file name id' is displayed. If everything's OK, the file selection continues.

As each file is selected, you are asked to 'CHOOSE FILE-TYPE: SEQ PRG USR'. Use the CRSR keys and SPACE BAR to choose. Relative files are automatically identified, so you needn't specify that type.

You can 'unselect' any file(s) by positioning the cursor on the filename and pressing the F5 key.

Press the F7 key when you've completed file selection. The Shell then asks you if it is "OK TO RESTORE FILE-LIST: N Y". If you answer Yes, the function continues. If you answer No (the default answer), the Shell returns to file selection with all previous selections intact.

Press the F5 key to restart file selection with all previous selections cancelled.

Press the STOP key to cancel the function and return to the primary menu.

RENAME FILES

The function changes the name of one or more files. Use the CRSR keys and SPACE BAR to select a file to rename. The message 'ENTER NEW NAME:' is displayed along with the original file name. You must now type in the new name or press F5 to un-select that file and continue.

Each new file name is checked to assure that it is unique. If it isn't, the message 'ERROR: FILE NAME NOT UNIQUE—RETRY' is displayed and the function returns to the 'ENTER NEW NAME:' message.

Press the F7 key when you've completed file selection. The Shell then asks you if it is "OK TO CHANGE FILE-LIST: N Y". If you answer Yes, the function continues. If you answer No (the default answer), the Shell returns to file selection with all previous selections intact.

Press the F5 key to restart file selection with all previous selections cancelled.

Press the STOP key to cancel the function and return to the primary menu.

REORDER DIRECTORY

This function changes the order in which the file names appear in a diskette directory. After you select this function, you are asked if you 'WANT TO ALPHABETIZE DIRECTORY: N Y'. The defaut answer is No. If you answer Yes, the file names are automatically reordered alphabetically.

Whether you answered Yes or No to the ALPHABETIZE DIRECTORY question, you can manually reorder the directory. Select a file with the CRSR keys and SPACE BAR. Then use the CRSR keys to 'drag' the file name through the file list to its new location. Press the SPACE BAR to deposit the file name.

Press the F7 key when you've completed file selection. The Shell then asks you if it is "OK TO RE-WRITE DIRECTORY: N Y". If you answer Yes, the function continues. If you answer No (the default answer), the Shell returns to file selection at the "WANT TO ALPHABETIZE: N Y" message with all previous changes intact.

Press the F5 key to restart file selection at the "WANT TO ALPHABETIZE: N Y" message with all previous changes cancelled.

Press the STOP key to cancel the function and return to the primary menu.

PART TWO:
ADVANCED OPERATION AND PROGRAMMING

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 you've used are program files, but there are others. In this chapter you'll learn about sequential data files.

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. 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.

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. 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 can be even more similar to sequential data files. User files are almost never used, but like program files, they could be treated as though they were sequential data files and some can be accessed with the same commands.
>
> For the advanced user, the similarity of the various file types offers the possibility of reading a program file into the computer a byte (character) at a time and rewriting it to the diskette in a modified form.

OPENING A FILE

One of the most powerful tools in Commodore BASIC is the OPEN statement. With it, you may send data almost anywhere, much like a telephone switchboard. 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 commands.

Before you 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
4,5	Printer	Sending output to a hard copy printer
8,9,10,11	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. If the character is kept in a variable, the device 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 you 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:

```
BASIC 7.0
840 DOPEN#4,"DEGREE DAY DATA",D0,U8,W
850 GOSUB 59990: REM CHECK FOR DISK ERRORS

BASIC 2.0
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 FOR SEQUENTIAL FILES:

```
BASIC 7.0
DOPEN#file#, "file name" [,Ddrive#] [,Udevice#] [,W]

BASIC 2.0
OPEN file #, device #, channel #,"drive #:file name,file type,direction"
```

where:

"file #" is an integer (whole number) between 1 and 255. Do not open a disk file with a file number greater than 127 it will cause severe problems. 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 8-11, and is normally 8 on the 1571.

"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 1571. 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, A = append 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.

"w" is an option that must be specified to write the sequential file, or the file will be opened to read.

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 three (or two 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:

```
BASIC 7.0: DOPEN#2,"PHONES",D0,U8,W
BASIC 2.0: OPEN 2,8,2,"0:PHONES,SEQUENTIAL,WRITE"
or
OPEN 2,8,2,"0:PHONES,S,W"
```

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

```
BASIC 7.0: DOPEN#2, "@PHONES", D0,U8,W
BASIC 2.0: OPEN 2,8,2,"@0:PHONES,S,W"
```

This erases all your old phone numbers, so make sure that any information that may be deleted is of no importance. After writing our phone file, remove the diskette and turn off the system. To recall the data in the file, reopen it with something like:

```
BASIC 7.0: DOPEN#8,"PHONES",D0,U8
BASIC 2.0: 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. It's possible to use an abbreviation form of the file name, if there are no other files that would have the same abbreviation:

```
BASIC 7.0: DOPEN#10,"PH*",D0,U8
BASIC 2.0: OPEN 10,8,6,"0:PH*,S,R"
```

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

```
BASIC 7.0:
100 INPUT "WHICH PHONE FILE (1-3)";PH
110 IF PH<>1 AND PH<>2 AND PH<>3 THEN 100
120 DOPEN#4,"PHONE" + STR$(PH),D0,U8

BASIC 2.0:
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.

ADDING TO A SEQUENTIAL FILE

The APPEND command allows you to reopen an existing sequential file and add more information to the end of it. 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

BASIC 7.0: APPEND#file#,"file name"[,Ddrive#] [,Udevice#]
BASIC 2.0: 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, type:

BASIC 7.0: APPEND#1,"0:JOHN PAUL JONES",D0,U8
BASIC 2.0: OPEN 1,8,3,"0:JOHN PAUL JONES,A"

In this case, the Disk Operating System (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. 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:

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

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 such as punctuation 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 wasted on the diskette, and may create more problems when reading the file back into the computer. Therefore, follow the following format precisely when sending data to a disk file.

FORMAT FOR THE PRINT# COMMAND:

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's better if each PRINT# statement to disk include only one data item. If you wish to include more items, they should 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, use:

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

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$
```

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

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 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 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 you are done with a file, the CLOSE statement will write the rest of the data buffer out to 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 should be the first file opened, and the last file closed in any program.

FORMAT FOR THE CLOSE STATEMENT

```
BASIC 7.0: DCLOSE#file# [,Udevice#]
BASIC 2.0: 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, use:

```
BASIC 7.0: DCLOSE#5
BASIC 2.0: CLOSE 5
```

In BASIC 7.0, when the DCLOSE statement is used alone (no# or file# parameters), it closes all disk files at once. With a bit of planning, the same can be done 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 you always gave your files numbers between 1 and 5, you could close them all with

```
9950 FOR I=1 TO 5
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 the last chapter.

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

```
INPUT#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 in the last chapter.

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 risk of reading improper data from the file.

MORE ABOUT INPUT# (ADVANCED USERS)

Troublesome Characters

After you begin using data files regularly, you may encounter two BASIC error messages. 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 two lines of text. This lower limitation is due to the 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 88 (BASIC 2) and 160 (BASIC 7) characters, BASIC will halt with a "STRING TOO LONG ERROR."

"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 converting it back to a number with the BASIC VAL() statement after removing non-numeric characters with the string functions described in your computer user's 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 you 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)+"DOE, JOHN"
```

or

```
PRINT#2,CHR$(34);"DOE, JOHN"
```

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 from the data by the Input or INPUT# statement, but the comma or colon will remain part of the data.

NUMERIC DATA STORAGE ON DISKETTE

Up to this point we have discussed string data storage, now let's look at numeric storage.

Inside the computer, the space occupied by a numeric variable depends only on its type. Simple numeric variables use seven bytes (character locations) of memory. Real array variables use five bytes per array element, and integer array elements use two 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. This is because 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, 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 using the VAL () function after the information is inside the computer.

For example, "N$ = RIGHT$(STR$(N),LEN(STR$(N)) − 1)" will convert a positive number N into a string N$ 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$(0), such as a machine language program file, you could correct any CHR$(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 read back safely 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$(0), but that may be an advantage in building a text string. If CHR$ (0) is required in the file, then use the following alternate line:

```
3320 If A$<>CHR$(13) THEN B$=B$+(A$ + CHR$(0): GOTO 3310
```

GET# may be 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 READ
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.

The following table applies to single errors or a combination of two or more errors.

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 (BASIC 2.0)

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$,B                          Display file contents
430 IF RS=0 THEN 390                    until done,
440 IF RS<>64 THEN
    PRINT"STATUS=";RS                   unless there's an error
450 CLOSE 2                             Then quit
455 CLOSE 15
460 END
480 REM ** ERROR CHECK S/R **           A Basic 3.5-only version could
500 INPUT#15,EN,EM$,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
```

DEMONSTRATION OF SEQUENTIAL FILES (BASIC 7.0)

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
170 PRINT CHR$(147):REM CLEAR
    SCREEN
190 PRINT "** WRITE A FILE **"
210 PRINT
220 DOPEN #2,"@SEQ FILE",W                Open demo file with replace
230 GOSUB 500                             Check for disk errors
240 PRINT"ENTER A WORD,
    THEN A NUMBER"
250 PRINT"OR 'END,0' TO STOP"
260 PRINT
270 INPUT A$,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$<>"END" THEN 270                 Until finished
310 PRINT
320 DCLOSE #2                             Tidy up
340 PRINT "** READ SAME FILE
    BACK **"
360 PRINT
370 DOPEN #2,"SEQ FILE"                   Reopen same file for reading
380 GOSUB 500
390 INPUT#2,A$,B                          Read next string & number from file
400 RS=ST                                 Remember file status
410 GOSUB 500
420 PRINT A$,B                            Display file contents
430 IF RS=0 THEN 390                      until done,
440 IF RS<>64 THEN
    PRINT"STATUS=";RS                     unless there's an error
450 DCLOSE #2                             Then quit
460 END
480 REM ** ERROR CHECK S/R **
500 IF DS>0 THEN PRINT DS$:STOP
510 RETURN
```

CHAPTER 6
RELATIVE 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 in some situations. For example, after writing a large list of mail labels, you wouldn't want to have to reread 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.

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 you 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, you 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, you need a more structured and predictable environment for our data.

The structure you 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 the highest record number, by ones, but the fields are up to you to organize. Each record will be of the same size, but the 1571 won't insist that they all be divided the same way. On the other hand, they normally will 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

With relative files, you don't have to worry 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 the 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 record 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 Open statement is used to reopen the file for both reading and writing.

FORMAT STATEMENT TO OPEN A RELATIVE FILE:

BASIC 7.0: DOPEN # file #, "file name", L record length [,Ddrive #] [,Udevice #]

BASIC 2.0: 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 1571; "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 1571; and "file name" is the name of the file, 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 file name (in BASIC 7.0) or the drive number (in BASIC 2.0) with the "at" sign (@); there is no reason to replace a relative file.

2. L record length (in BASIC 7.0) or, L ," + CHR$(record length) (in BASIC 2.0) 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. In BASIC 7.0 DOPEN, whenever a variable or expression is used as a file name it must be surrounded by parentheses.

4. Only 1 relative file can be open at a time on the 1571, although a sequential file and the command channel may also be open at the same time. However, if you have a sequential and relative file open at the same time, you can't request a directory.

EXAMPLES:

To create or reopen a relative file named "GRADES", of record length 100, use

```
BASIC 7.0: DOPEN#2,"GRADES",L100,D0,U8
BASIC 2.0: OPEN 2,8,2,"GRADES,L," + CHR$(100)
```

To reopen an unknown relative file of the user's choice that has already been created, use:

```
BASIC 7.0: 200 INPUT"WHICH FILE";FI$
           210 DOPEN#5,(FI$),D0,U8

BASIC 2.0: 200 INPUT"WHICH FILE";FI$
           210 OPEN 5,8,5,FI$
```

USING RELATIVE FILES: RECORD# COMMAND

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 two 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. This means each 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.

FORMAT FOR THE RECORD# COMMAND:

BASIC 7.0: RECORD # file #, record number [,offset]
BASIC 2.0: PRINT#15, "P" + CHR$ (channel # + 96) + CHR$
(<record #) + CHR$(>record #) + CHR$(offset)

where "file #" is the file # specified in the current DOPEN statement for the specified file, "record number" is the desired record number, "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, you 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 two bytes, with one 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# command once before the Read or Write access and once after the access.

EXAMPLES:

In BASIC 7.0, to position the record pointer for file #2 to record number 3, type:

```
RECORD#2,3
```

In BASIC 2.0, to position the record pointer for channel #2 to record number 3, 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 in BASIC 7.0.

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 you 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 NUMBER 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)
```

COMPLETING RELATIVE FILE CREATION

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

BASIC 2.0:

```
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
```

(continued)

```
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$(RH)                      Position to last record
                                            number
1455 PRINT#15,"P"+CHR$(96+2)+               Re-position for safety
     CHR$(RL)+CHR$(RH)
1460 GOSUB 59990
1470 PRINT#1,CHR$(255);                     Send default character to it
1480 GOSUB 59990
1500 GOSUB 59990
1510 CLOSE 1                                Now the file can be safely
                                            closed
1520 GOSUB 59990
9980 CLOSE 15                               And the command channel
                                            closed
9990 END                                    Before we end the pro-
                                            gram
59980 REM CHECK DISK SUBROUTINE
59990 INPUT#15,EN,EM$,ET,ES
60000 IF EN>1 AND EN<>50 THEN PRINT         Error check subroutine
      EN,EM$,ET,ES:STOP                     Ignore "RECORD NOT
                                            PRESENT"
60010 RETURN
```

BASIC 7.0:

```
1380 INPUT"ENTER RELATIVE FILE NAME";FI$    Select file parameters
1390 INPUT"ENTER MAX. # OF RECORDS";NR
1400 INPUT"ENTER RECORD LENGTH";RL
1410 DOPEN#1,(FI$),L(RL)                    Begin to create desired file
1420 GOSUB 60000                            Check for disk errors
                                            Calculate length values
1450 RECORD#1,(NR)                          Position to last record
                                            number
1455 RECORD#1,(NR)
1460 GOSUB 60000
1470 PRINT#1,CHR$(255);                     Send default character to it
1480 GOSUB 60000
1500 GOSUB 60000
1510 CLOSE 1                                Now the file can be safely
                                            closed
```

(continued)

```
1520 GOSUB 60000
9980 CLOSE 15                          And the command channel
                                       closed
9990 END                               Before we end the pro-
                                       gram

59980 REM CHECK DISK SUBROUTINE
60000 IF DS>1 AND DS<>50 THEN PRINT     Error check subroutine
      DS,DS$:STOP                      Ignore "RECORD NOT
                                       PRESENT"
60010 RETURN
```

Two lines require additional explanation. When line 1470 executes, the disk drive will operate for up to several 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. 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. Ignore it because not having a requested record would only be an error if that record had been created previously.

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. When the first record beyond the current end record is written, the DOS returns "50, Record Not Present" error. This is expected and correct.

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 one 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 (88 is for BASIC 2, 160 is for BASIC 7). If this is not done, you will have to use the GET# command to read the record, at a significant cost in speed.

Since each relative record is most easily 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		WITH VARIABLE LENGTH FIELDS	
Field Name	Length	Field Name	Length
NAME	27 characters	NAME	31 characters
STREET	27 characters	STREET	31 characters
CITY & STATE	23 characters	CITY & STATE	26 characters
ZIP CODE	10 characters	ZIP CODE	11 characters
Total length	87 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, you 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 one 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).

BASIC 7.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 :
120 DOPEN#3,"MYRELFILE"L88
130 CR$=CHR$(13)
140 INPUT"NAME"; NA$
150 IF LEN(A$)>30 THEN 140
160 INPUT"STREET";SA$
170 IF LEN(SA$)>30 THEN 160
```

(continued)

```
180 INPUT"CITY & STATE"; CS$
190 IF LEN(CS$)>25 THEN 180
200 INPUT"ZIP CODE";ZP$
210 IF LEN(ZP$)>10 THEN 200
220 DA$=NA$+CR$+SA$+CR$+CS$+CR$;ZP$
230 IF LEN(DA$)<88 THEN 260
240 PRINT"RECORD TOO LONG"
250 GOTO 140
260 :
270 :
280 RECORD#3,(RE),1
290 IFDS=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 RECORD#3,(RE),1
340 GOSUB1000
350 DCLOSE3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINTDS$:DCLOSE3:END
```

BASIC 2.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 OPEN 15,8,15
120 OPEN3,8,3,"MYRELFILE,L,"+CHR$(88)
130 CR$=CHR$(13)
140 INPUT"NAME"; NA$
150 IF LEN(A$)>30 THEN 140
160 INPUT"STREET";SA$
170 IF LEN(SA$)>30 THEN 160
180 INPUT"CITY & STATE"; CS$
190 IF LEN(CS$)>25 THEN 180
200 INPUT"ZIP CODE";ZP$
210 IF LEN(ZP$)>10 THEN 200
220 DA$=NA$+CR$+SA$+CR$+CS$+CR$;ZP$
230 IF LEN(DA$)<88 THEN 260
240 PRINT"RECORD TOO LONG"
250 GOTO 140
260 RH=INT(RE/256)
270 RL=RE-256*RH
280 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
290 GOSUB1000:IF EN=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
```

(continued)

```
340 GOSUB1000
350 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1010 PRINTEM$:CLOSE3:CLOSE15:END
```

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

BASIC 7.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 :
120 DOPEN#3,"MYRELFILE",L88
130 BL$="(27 shited space chars)"
140 INPUT"NAME"; NA$
145 LN=LEN(NA$)
150 IF LEN>27 THEN 140
155 NA$=NA$+LEFT$(BL$,27-LN)
160 INPUT"STREET";SA$
165 LN=LEN(SA$)
170 IF LEN>27 THEN 160
175 SA$=SA$+LEFT$(BL$,27-LN)
180 INPUT"CITY & STATE"; CS$
185 LN=LEN(CS$)
190 IF LEN>23 THEN 180
195 CS$=CS$+LEFT$(BL$,23-LN)
200 INPUT"ZIP CODE";ZP$
205 LN=LEN(ZP$)
210 IF LN>10 THEN 200
215 ZP$=ZP$+LEFT$(BL$,10-LN)
220 DA$=NA$+SA$+CS$+ZP$
260 :
270 :
280 RECORD#3,(RE),1
290 IFDS=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 RECORD#3,(RE),1
340 GOSUB1000
350 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#):END
```

BASIC 2.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 OPEN 15,8,15
120 OPEN#3,8,3,"MYRELFILE,L,"+CHR$(88)
130 BL$="(27 shited space chars)"
140 INPUT"NAME"; NA$
145 LN=LEN(NA$)
150 IF LEN>27 THEN 140
155 NA$=NA$+LEFT$(BL$,27-LN)
160 INPUT"STREET";SA$
165 LN=LEN(SA$)
170 IF LEN>27 THEN 160
175 SA$=SA$+LEFT$(BL$,27-LN)
180 INPUT"CITY & STATE"; CS$
185 LN=LEN(CS$)
190 IF LN>23 THEN 180
195 CS$=CS$+LEFT$(BL$,23-LN)
200 INPUT"ZIP CODE";ZP$
205 LN=LEN(ZP$)
210 IF LN>10 THEN 200
215 ZP$=ZP$+LEFT$(BL$,10-LN)
220 DA$=NA$+SA$+CS$+ZP$
260 RH=INT(RE/256)
270 RL=RE-256*RH
280 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
290 GOSUB1000:IF EN=50THENPRINT#3)CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
340 GOSUB1000
350 GOSUB1000:CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,E
1002 IF EN<20 THEN RETURN
1010 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END
```

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:

BASIC 7.0:

```
10 :
20 DOPEN#3,"MYRELFILE",L88
30 INPUT"ENTER RECORD NUMBER";RE
40 :
50 :
60 RECORD#3,(RE),1
70 GOSUB1000
80 INPUT#3,NA$,SA$,CS$,ZP$
90 GOSUB1000
100 RECORD#3,(RE),1
110 GOSUB1000
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#3:END
```

BASIC 2.0:

```
10 OPEN 15,8,15
20 OPEN3,8,3,"MYRELFILE,L,"+CHR$(88)
30 INPUT"ENTER RECORD NUMBER";RE
40 RH=INT(RE/256)
50 RL=RE-256*RH
60 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
70 GOSUB1000
80 INPUT#3,NA$,SA$,CS$,ZP$
90 GOSUB1000
100 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
110 GOSUB1000
120 PRINTNA$:PRINTSA$
```

(continued)

```
130 PRINTCS$:PRINTZP$
140 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1002 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END

READY.
```

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

BASIC 7.0:

```
10 :
20 DOPEN#3,"MYRELFILE",L88
30 INPUT"ENTER RECORD NUMBER";RE
40 :
50 :
60 RECORD#3,(RE),1
70 GOSUB1000
80 INPUT#3,DA$
90 GOSUB1000
100 RECORD#3,(RE),1
110 GOSUB1000
112 NA$=LEFT$(DA$,27)
114 SA$=MID$(DA$,28,27)
116 CS$=MID$(DA$,55,23)
118 ZP$=RIGHT$(DA$,10)
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#3:END
```

BASIC 2.0:

```
10 OPEN 15,8,15
20 OPEN3,8,3,"MYRELFILE,L"+CHR$(88)
30 INPUT"ENTER RECORD NUMBER";RE
40 RH=INT(RE/256)
50 RL=RE-256*RH
60 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
70 GOSUB1000
80 INPUT#3,DA$
90 GOSUB1000
100 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
110 GOSUB1000
```

(continued)

```
112 NA$ = LEFT$(DA$,27)
114 SA$ = MID$(DA$,28,27)
116 CS$ = MID$(DA$,55,23)
118 ZP$ = RIGHT$(DA$,10)
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1002 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END

READY.
```

THE VALUE OF INDEX FILES (ADVANCED USERS)

In the last two chapters you 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 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 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.

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.

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 1571 to use for this file's data.

EXAMPLES:

If we don't specify which disk buffer to use, the 1571 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 (See Chapter 8).

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, 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.

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; 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 a BLOCK-READ, use:

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

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

You 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)

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. 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, and "track #" and "sector #" are 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. 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 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, rerun 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, 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, 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)

By combining the commands in this chapter, it is possible to develop a file-handling 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 1571, 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 1571, either by writing them into disk memory from the computer, or by loading them directly from diskette into the desired disk memory buffer. 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 1571. A brief 1571 map appears below.

1571 MEMORY MAP

Location	Purpose
0000-00FF	Zero page work area, job queue, variables
0100-01FF	GCR overflow area and stack (1571 mode BAM side one)
0200-02FF	Command buffer, parser, tables, variables
0300-07FF	5 data buffers, 0-4 — one of which is used for BAM
1800-65C22A	Serial, controller ports
1C00-65C22A	Controller ports
8000-FFE5	32K byte ROM, DOS and controller routines
FFE6-FFFF	JMP table, user command vectors

NOTE: The 1571, as well as other Commodore peripherals, is designed to support interfacing via software command structures. The software commands provided in the 1571 allow for a smooth and controllable interface between the peripheral and CPU. Commodore reserves the right to change the ROM and I/O structure at any time.

MEMORY-READ

The disk contains 32K of ROM (Read-Only Memory), as well as 4K of RAM (Read-Write Memory) of which only 2K is used. 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.

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, you 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.

```
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$(NL)                                        Loop until have all the data,
180 FOR I=1 TO NL
190 GET#15,A$: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.

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)

These 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 1571 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 six 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 six 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$(179)CHR$(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, 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 1571 without changing its variables, UJ cold-starts the 1571 almost as if it had been turned off and on again.

User Command	Function
u0	restores default user jump table
u1 or ua	block read replacement
u2 or ub	block write replacement
u3 or uc	jump to $0500
u4 or ud	jump to $0503
u5 or ue	jump to $0506
u6 or uf	jump to $0509
u7 or ug	jump to $050c
u8 or uh	jump to $050f
u9 or ui	jump to ($fffa) reset tables
u: or uj	power up vector

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:";	Form of DOS RESET command
PRINT#15,"U3";	Execute program at start of buffer 2

UTILITY LOADER

This command loads a user-type file into the drive RAM. The first two bytes of the file must contain the low and high addresses respectively. The third byte is the amount of characters to follow. In addition, a trailing checksum byte must be included. The load address is the starting address.

FORMAT FOR THE UTILITY LOADER COMMAND

PRINT#"&0:filename"

NOTE: The filename parameter is valid only in 1571 mode.

CHAPTER 9
MACHINE LANGUAGE PROGRAMS

Here is a list of host computer 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
```

CHAPTER 10
BURST COMMANDS

The Burst Command Instruction Set (BCIS) is a series of powerful, versatile, and complex commands that enables the user to format, read, and write in numerous formats. Burst commands are sent via kernal calls, but the handshaking of data is done by the user for maximum performance. There is no parameter checking, so exercise care when using the BCIS. For instance, if a burst read with an illegal track address is sent to a 1571, the drive will noisily keep trying to find the invalid track. Reading and writing in other formats is automatic if the commands are given in proper sequence. Please become thoroughly familiar with all the commands and follow the examples given in this chapter. It's important to follow the handshake conventions exactly for maximum performance.

READ

BYTE	BIT7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	T	E	B	S	0	0	0	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE: All values are determined by the particular disk format.

SWITCHES: T—transfer data (1 = no transfer)
E—ignore error (1 = ignore)
B—buffer transfer only (1 = buffer transfer only)
S—side select (MFM only)
N—drive number

PROTOCOL: Burst handshake

CONVENTIONS: Before you can READ or WRITE to a diskette, it must be logged-in using either the INQUIRE DISK or QUERY DISK FORMAT command (both are described later). This must be done once each time you change diskettes.

OUTPUT: One burst status byte, followed by burst data, is sent for each sector transfered. An error prevents data from being sent unless the E bit is set.

WRITE

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	T	E	B	S	0	0	1	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE: All values are determined by the particular disk format:

SWITCHES: T—transfer data (1 = no transfer)
E—ignore error (1 = ignore)
B—buffer transfer only (1 = buffer transfer only)
S—side select (MFM only)
N—drive number

PROTOCOL: Go output (spout), send data, go input (spin), pull clock low, wait for status, release clock (for multi-sector, start over i.e. Go output, etc.).

CONVENTIONS: Before you can READ or WRITE to a diskette, it must be logged-in using either the INQUIRE DISK or QUERY DISK FORMAT command (both are described later). This must be done once each time you change diskettes.

INPUT: Host must transfer burst data.

OUTPUT: One burst status byte following each WRITE operation.

INQUIRE DISK

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	S	0	1	0	N

SWITCHES: S—side select (MFM only)
N—drive number

PROTOCOL: Burst handshake

OUTPUT: One burst status byte following each INQUIRE DISK operation.

FORMAT MFM

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	P	I	D	S	0	1	1	N
03	M=1	T	LOGICAL STARTING SECTOR					
04	INTERLEAVE		(OPTIONAL DEF-0)					
05	SECTOR SIZE		*(OPTIONAL DEF-01,256 BYTE SECTORS)					
06	LAST TRACK NUMBER		(OPTIONAL DEF-39)					
07	NUMBER OF SECTORS		**(OPTIONAL DEPENDS ON BYTE 05)					
08	LOGICAL STARTING TRACK		(OPTIONAL DEF-0)					
09	STARTING TRACK OFFSET		(OPTIONAL DEF-0)					
0A	FILL BYTE		(OPTIONAL DEF-$E5)					
0B-??	SECTOR TABLE		(OPTIONAL T-BIT SET)					

*00—128 BYTE SECTORS	**DEF 26—128 BYTE SECTORS
01—256 BYTE SECTORS	16—256 BYTE SECTORS
02—512 BYTE SECTORS	9—512 BYTE SECTORS
03—1024 BYTE SECTORS	5—1024 BYTE SECTORS

SWITCHES: P—partial format (1 = partial)
I—index address mark written (1 = written)
D—double sided flag (1 = format double sided)
S—side select
T—sector table included (1 = included, all other parameters must be included)
N—drive number

PROTOCOL: Conventional

CONVENTIONS: This command must be followed by the INQUIRE DISK or QUERY DISK FORMAT command to log in the diskette.

OUTPUT: None. The status is updated within the drive.

FORMAT GCR (NO DIRECTORY)

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	X	0	1	1	N
03	M=0							
04			ID LOW					
05			ID HIGH					

SWITCHES N—drive number
X—don't care

PROTOCOL: Conventional

CONVENTIONS: This command must be preceded by the INQUIRE DISK or QUERY DISK FORMAT command to log in the diskette.

OUTPUT: None. The status is updated within the drive.

SECTOR INTERLEAVE

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	W	X	X	0	1	0	0	N
04			INTERLEAVE					

SWITCHES W—write switch (0=write)
N—drive number
X—don't care

PROTOCOL: Burst handshake (W=1)

CONVENTIONS: This is a soft interleave used for multi-sector burst READ and WRITE.

OUTPUT: None (W = 0), Interleave burst byte (W = 1)

QUERY DISK FORMAT

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	F	X	X	S	1	0	1	N
03			OFFSET		(OPTIONAL F-BIT SET)			

SWITCHES: F—force flag (F = 1 steps the head with the offset specified in byte 03)
N—drive number
X—don't care
S—side select

PROTOCOL: Burst handshake

CONVENTIONS: Determines the diskette format on any particular track. Also logs non-standard diskettes (i.e. minimum sector addresses other than one).

OUTPUT: *burst status byte (no bytes will follow if there is an error or if the format is GCR)
**burst status byte (no bytes will follow if there was an error in compiling MFM format information)
number of sectors (the number of sectors on a particular track)
logical track (the logical track number found in the disk header)
minimum sector (the logical sector with the lowest value address)
maximum sector (the logical sector with the highest value address)
CP/M interleave (the hard interleave found on a particular track)

*status from track offset zero
**if F bit is set, status is from offset track

INQUIRE STATUS

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	W	C	X	0	1	1	0	N
03			NEW STATUS (W-BIT CLEAR)					

SWITCHES: W—write switch (0 = write)
C—change (C = 1 and W = 0—log in disk)
(C = 1 and W = 1—return whether disk was logged, i.e. $B error or old status)
N—drive number
X—don't care

PROTOCOL: Burst handshake (W = 1)

CONVENTIONS: This is a method of reading or writing current status.

OUTPUT: None (W = 0), Burst status byte (W = 1)

CHGUTL UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	1	1	1	1	0
03	UTILITY COMMANDS: 'S', 'R', 'T', 'M', 'H', #DEV							
04	COMMAND PARAMETER							

SWITCHES: X—don't care

UTILITY COMMANDS: 'S'—DOS sector interleave
'R'—DOS retries
'T'—ROM signature analysis
'M'—mode select
'H'—head select
#DEV—device #

Note: Byte 02 is equivalent to '>'.

EXAMPLES: "U0>S" + CHR$ (SECTOR INTERLEAVE)
"U0>R" + CHR$ (RETRIES)
"U0>T" (If the ROM signature failed, the activity LED blinks 4 times)
"U0>M1" = 1571 MODEL"U0>M0" = 1541 MODE
"U0>H0" = SIDE ZERO"U0>H1" = SIDE ONE (1571 mode only)
"U0>" + CHR$ (#DEV), where #DEV = 4 − 30

FASTLOAD UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	P	X	X	1	1	1	1	1
03-??	FILE NAME							

SWITCHES: P—sequential file bit (P = 1, does not have to be a program file)
X—don't care

PROTOCOL: Burst handshake

OUTPUT: Burst status byte preceding each sector transfered. Track and sector for link are automatically treated. In a program file, the loading address should be handled correctly.

STATUS IS AS FOLLOWS: 0000000X—OK
00000010—file not found
00011111—EOI

Any other status byte should be considered a file read error.

Note: The byte after the EOI status byte denotes how many data bytes follow.

STATUS BYTE BREAKDOWN

BIT 7	6	5	4	3	2	1	0
MODE	DN	SECTOR SIZE		[	CONTROLLER STATUS		]

MODE—1 = MFM, 0 = GCR
DN—DRIVE NUMBER

SECTOR SIZE—(MFM ONLY)
00 ..... 128 BYTE SECTORS
01 ..... 256 BYTE SECTORS
10 ..... 512 BYTE SECTORS
11 ..... 1024 BYTE SECTORS

CONTROLLER STATUS (GCR)
000X ..... OK
0010 ..... SECTOR NOT FOUND
0011 ..... NO SYNC
0100 ..... DATA BLOCK NOT FOUND
0101 ..... DATA BLOCK CHECKSUM ERROR
0110 ..... FORMAT ERROR
0111 ..... VERIFY ERROR
1000 ..... WRITE PROTECT ERROR
1001 ..... HEADER BLOCK CHECKSUM ERROR
1010 ..... DATA EXTENDS INTO NEXT BLOCK
1011 ..... DISK ID MISMATCH/DISK CHANGE
1100 ..... RESERVED
1101 ..... RESERVED
1110 ..... SYNTAX ERROR
1111 ..... NO DRIVE PRESENT

CONTROLLER STATUS (MFM)

000X .....OK
0010 .....SECTOR NOT FOUND
0011 .....NO ADDRESS MARK
0100 .....RESERVED
0101 .....DATA CRC ERROR
0110 .....FORMAT ERROR
0111 .....VERIFY ERROR
1000 .....WRITE PROTECT ERROR
1001 .....HEADER BLOCK CHECKSUM ERROR
1010 .....RESERVED
1011 .....DISK CHANGE
1100 .....RESERVED
1101 .....RESERVED
1110 .....SYNTAX ERROR
1111 .....NO DRIVE PRESENT

BURST TRANSFER PROTOCOL

Before using the following burst transfer routines, you must determine whether or not the peripheral is a fast device. The Fast Serial (byte mode) protocol makes that determination internally when you include a query routine (send-cmd-string;). This routine addresses the peripheral as a listener and thereby determines its speed.

BURST READ

```
send-cmd-string;                    (*determine speed*)
if device-fast then
    serial-in;                      (*turn 6526 to input*)
    toggle-clock;                   (*toggle state of clock line*)
    repeat                          (*repeat for all sectors*)
        read-error;                 (*retrieve error byte*)
        toggle-clock;               (*no error*)
            repeat                  (*repeat for all sectors*)
                wait-byte;          (*poll 6526 for byte*)
                toggle-clock;       (*toggle clock*)
                store-data;         (*save data*)
        until last-byte;            (*last byte ?*)
    until last-sector;              (*any more sectors ?*)
    set-clock-high;                 (*release clock line*)
else
    read-1541;                      (*send unit read*)
```

BURST WRITE

```
send-cmd-string;                    (*determine speed*)
if device-fast then
    repeat                          (*repeat for multi-sector*)
        serial-out;                 (*serial port out*)
            repeat                  (*repeat for sector-size*)
                send-byte;          (*send byte*)
            until last-byte;        (*last byte ?*)
        serial-in;                  (*serial port in*)
        clock-low;                  (*ready for status*)
        read-err;                   (*controller error ?*)
        clock-high;                 (*restore clock*)
    until last-sector;              (*until last sector*)
else
    write-1541;                     (*unit write*)
```

EXPLANATION OF PROCEDURES

send-cmd-string	sends command string to determine whether the drive is fast or slow.
toggle-clock	changes the state of the clock line.
clock-hi	changes the state of the clock to logic 1.
clock-lo	changes the state of the clock to logic 0.
wait-byte	polls the 6526 for a byte ready.
read-error	calls wait-byte, then returns to the main if there are no errors.
store-data	stores the data in a particular memory location.
last-byte	depending on sector size, will increment and compare value to sector size.
last-sector	decrements the number of sector transfers requested and stops when done.
serial-in	sets the 6526 serial port and driver circuit to input mode.
read-err	calls wait-byte and evaluates the status of the previous controller job.
serial-out	sets the 6526 serial port and driver circuit to output mode.

send-byte sends a byte of data to the 1571.

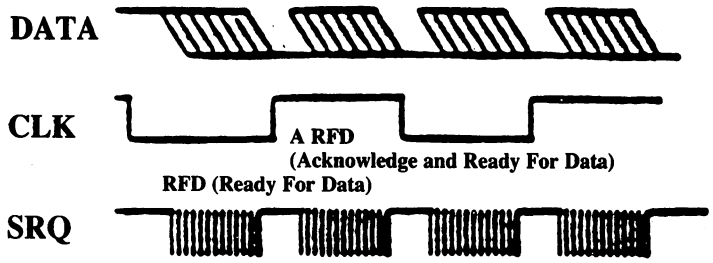
read-1541 sends a typical unit read to a 1541.

write-1541 sends a typical unit write to a 1541.

HANDSHAKE

The figure below shows the burst transfer protocol. It is a stat-dependent protocol (simple and fast). As the clock line is toggled, a byte of data is sent. Burst protocol is divided into three parts:

1. Send Command: send string using existing kernal routines.
2. Query: determine whether the peripheral is fast.
3. Handshake Code: follow handshake conventions.



EXAMPLE BURST ROUTINES

```
        *=$1800

; ROUTINE TO READ N-BLOCKS OF DATA
; COMMAND CHANNEL MUST BE OPEN ON DRIVE
; OPEN 15,8,15
; BUFFER AND CMD_BUF, AND CMD_LENGTH MUST BE SETUP
; PRIOR TO CALLING THIS COMMAND.

serial   = $0alc          ; fast serial flag
d2pra    = $dd00
clkout   = $10
dlicr    = $dc0d
dlsdr    = $dc0c
```

```
stat      = $fa
buffer    = $fb              ; $fb & $fc
          lda   #15          ; logical file number
          ldx   #8           ; device number
          ldy   #15          ; secondary address
          jsr   setlfs       ; setup logical file

          lda   #0           ; noname
          jsr   setnam       ; setup file name

          jsr   open         ; open logical channel

; after the command channel is open subsequent calls should be from 'read'

read      lda   #$00
          sta   stat         ; clear status
          lda   serial
          and   #%10111111   ; clear bit 6 fast serial flag
          sta   serial
          ldx   #15
          jsr   chkout       ; open channel for output
          ldx   #0
          ldy   cmd_length   ; length of the command
sendit    lda   cmd_buf,x    ; get command
          jsr   bsout        ; send the command
          inx
          dey
          bne   sendit

          jsr   clrchn       ; send eoi
          bit   serial       ; check speed of drive
          bvc   error        ; slow serial drive

          sei
          bit   dlicr        ; clear interrupt control reg
          ldx   cmd_buf+5    ; get # of sectors
          lda   d2pra        ; read serial port
          eor   #clkout      ; change state of clock
          sta   d2pra        ; store back

read_it   lda   #8
wait 1    bit   dlicr        ; wait for byte
          beq   wait 1

          lda   d2pra        ; read serial port
          eor   #clkout      ; change state of clock
          sta   d2pra        ; store back
```

```
;          Code to check status byte.
;
;     1) This code will check for mode
;          whether GCR or MFM.
;     2) Verify sector size.
;     3) Check for error, if ok then continue.
;          On error, check error switch if set continue
;          otherwise abort.
;     4) Verify switches

          lda    dlsdr          ; get data from serial data reg
          sta    stat           ; save status
          and    #15
          cmp    #2             ; just check for (3)
          bcs    error

          ldy    #0             ; even page
top_rd    lda    #8
wait 2    bit    dlicr          ; wait for byte
          beq    wait 2

          lda    d2pra          ; toggle clock
          eor    #clkout
          sta    d2pra

          lda    dlsdr          ; get data
          sta    (buffer),y     ; save data
          iny
          bne    top_rd         ; continue for buffer size

          dex
          beq    done_read      ; done ?

          inc    buffer+1       ; next buffer
          jmp    read_it

done_read
          clc
          .byte  $24
error     sec
          rts                   ; return to sender

cmd_buf   .byte  'U0',0,0,0,0,0
```

```
*=$1800

; ROUTINE TO WRITE N-BLOCKS OF DATA
; COMMAND CHANNEL MUST BE OPEN ON DRIVE
; OPEN 15,8,15
; BUFFER AND CMD_BUF, AND CMD_LENGTH MUST BE SETUP
; PRIOR TO CALLING THIS COMMAND.

serial   = $0alc          ; fast serial flag
d2pra    = $dd00
clkout   = $10
old_clk  = $fd
clkin    = $40
dlicr    = $dc0d
dlsdr    = $dc0c
stat     = $fa
buffer   = $fb            ; $fb & $fc
mmureg   = $d505
d1timh   = $dc05          ; timer a high
dltiml   = $dc04          ; timer a low
dlcra    = $dc0e          ; control reg a

         lda   #15        ; logical file number
         ldx   #8         ; device number
         ldy   #15        ; secondary address
         jsr   setlfs     ; setup logical file
         jsr   setnam     ; setup file name

         jsr   open       ; open logical channel

; after the command channel is open subsequent calls should be from 'write'

write    lda   #$00
         sta   stat
         lda   serial
         and   #%10111111 ; clear bit 6 fast serial flag
         sta   serial
         ldx   #15
         jsr   chkout     ; open channel for output

         ldx   #0
         ldy   cmd_length ; length of the command
sendit   lda   cmd_buf,x  ; get command
         jsr   bsout      ; send the command
         inx
         dey
         bne   sendit
```

```
        jsr     clrchn          ; send eoi

        bit     serial          ; check speed of drive
        bvc     error
        sei
        lda     #clkin
        sta     old_clk         ; clock starts high

        ldy     #0              ; even page
        ldx     cmd_buf+5       ; get # of sectors
wrt_it  jsr     spout           ; serial port out
topwr   lda     d2pra           ; check clock
        cmp     d2pra           ; debounce
        bne     topwr

        eor     old_clk
        and     #clkin
        beq     topwr

        lda     old_clk         ; change status of old clock
        eor     #clkin
        sta     old_clk

        lda     (buffer),y      ; send data
        sta     dlsdr

OPTIONAL                        ; put code here or before spinp

        lda     #8
wait 1  bit     dlicr           ; wait for transmission time
        beq     wait 1

END OPTIONAL

        iny
        bne     topwr           ; continue for buffer size

; talker turn around

        jsr     spinp           ; serial port input
        bit     dlicr           ; clear pending
        jsr     clklo           ; set clk low, tell him we are ready

        lda     #8
wait 2  bit     dlicr           ; wait for status byte
        beq     wait 2
```

```
            lda     dlsdr           ; get data from serial data reg
            sta     stat            ; save status
            jsr     clkhi           ; set clock high

;           Code to check status byte.
;
;       1) This code will check for mode
;          whether GCR or MFM.
;       2) Verify sector size.
;       3) Check for error, if ok then continue.
;          On error, check error switch if set continue
;          otherwise abort.
;       4) Verify switches

            lda     stat            ; retrieve status
            and     #15
            cmp     #2              ; just chk for error only (3)
            bcs     error           ; finish

            dex
            beq     done_wr         ; done?

            inc     buffer+1        ; next buffer
            jmp     wrt_it

done_wr
            cli
            .byte   $24
error       sec
            rts

cmd buf     .byte   'U0',2,0,0,0,0
```

```
; subroutines (refer to C128 User's Guide for SPINP and SPOUT vectors)

spout   lda     mmureg          ; change serial direction to output
        ora     #$08
        sta     mmureg
        lda     #$7f
        sta     dlicr           ; no irq's
        lda     #$00
        sta     dltimh
        lda     #$03
        sta     dltiml          ; low 6 us bit (fastest)
        lda     dlcra
        and     #$80            ; keep TOD
        ora     #$55
        sta     dlcra           ; setup CRA for output
        bit     dlicr           ; clr pending
        rts

spinp   lda     d1cra           ; input, 6526
        and     #$80
        ora     #$08
        sta     d1cra
        lda     mmureg
        and     #$f7
        sta     mmureg          ; mmu serial direction in
        rts

clklo   lda     d2pra           ; set clock low
        ora     #clkout
        sta     d2pra
        rts

clkhi   lda     d2pra           ; set clock high
        and     #$ff-clkout
        sta     d2pra
        rts
```

APPENDIX A
CHANGING THE DEVICE NUMBER

HARDWARE METHOD

Two switches on the back of the 1571 enable you to change the device # of the drive. You can use a screwdriver, pen, or any other small tool to set the switches. The following table shows the settings required for each device number:

Left	Right	Device #
UP	UP	8
DOWN	UP	9
UP	DOWN	10
DOWN	DOWN	11

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 the two switches 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 1571 is reset.

FORMAT FOR TEMPORARILY CHANGING THE DISK DEVICE NUMBER:

PRINT#15,"U0>" + CHR$(n)

Where n = 8 to 30

EXAMPLE:

Here is a program that sets any device number:

```
5  INPUT "OLD DEVICE NUMBER"; ODV
10 INPUT "NEW DEVICE NUMBER"; DV
20 IF DV<8 or DV>30 then 10
30 OPEN 15,ODV,15, "U0>" + CHR$(DV): 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.

APPENDIX B
DOS ERROR MESSAGES

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 reading simply 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 (long line)
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 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 reopened for reading. This file must be immediately rescued, as described in BASIC Hint #2 in Chapter 2, 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 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 two 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 V3.0 1571)
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 1571 diskette.

74: DRIVE NOT READY
An attempt has been made to access the 1571 single disk without a formatted diskette in place. Blank diskettes cannot be used until they have been formatted.

APPENDIX C
DISKETTE FORMATS: GCR/MFM

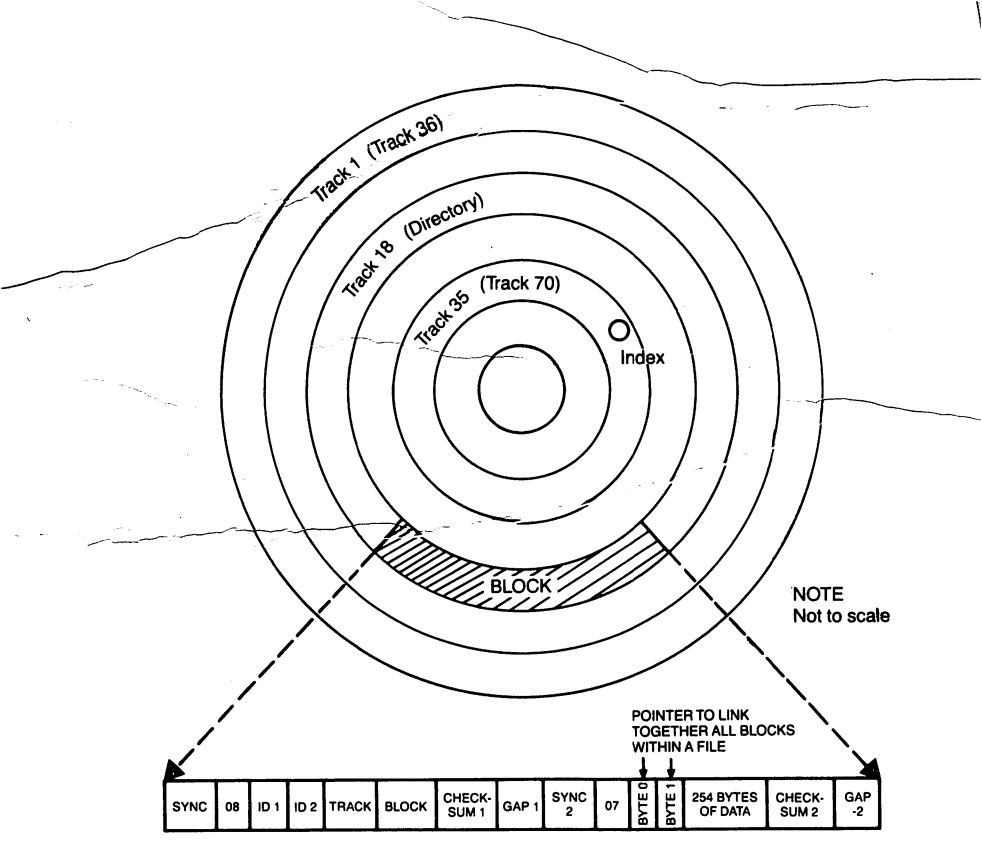
Basically, the surface of a formatted diskette is divided into a number of concentric circles called tracks. Each track is subdivided into sectors (also called blocks) and each sector consists of a number of bytes. The number of tracks per diskette, sectors per track, and bytes per sector depends on the type of recording format used.

The 1571 has the ability to read and write two different formats—GCR and MFM.

GCR (Group Code Recording) is the only format used by the Disk Operating System in all other Commodore disk drives. On a GCR-formatted disk (see Figure C-1), there are 35 tracks (70 on a double-sided diskette) numbered 1 to 35 (36 to 70) starting at the outermost track. The number of sectors per track is fixed, although not uniform, throughout the entire surface (see the table entitled "Block Distribution by Track"), and each sector consists of 256 bytes.

MFM (Modified Frequency Modulation) is the 'standard' format used by many other companies. The ability of the 1571 to handle MFM diskettes provides the Commodore 128 with access to applications software in other companies' formats. On an MFM-formatted diskette (see Figure C-2), the tracks per diskette, sectors per track, and bytes per sector are not fixed. With the 1571, those parameters are programmable, allowing greater flexibility in reading and writing different forms of MFM.

To make Commodore's version of CP/M usable with disk drives other than the 1571, CP/M files can be used in either MFM or GCR format. In this manual, information on CP/M appears in Appendix G. More detailed CP/M information can be found in the *Commodore 128 System Guide*.



Note: Tracks 36 to 70 refer to double-sided disks readable in 1571 mode.

Figure C-1. GCR-Formatted Diskette

SYNC 1	— 40 bits of ones (all 1's is the highest write frequency).
08	— Disk I.D. field identification mark.
CHECKSUM	— Checksum of ID1, ID2, SECTOR, TRACK.
SECTOR	— The number of this sector
TRACK	— Track which this sector resides on.
ID1, ID2	— 2 bytes (20 bits) indicating the diskette I.D. (the I.D. is specified by the user when the diskette is formatted).
GAP1	— 72 bits of GCR0 or 01010101 (8 bit byte ×9)
SYNC 2	— 40 bits of ones.
07	— Disk data field identification mark.
BYTE0, BYTE1	— Track and sector of next related block.
DATA	— 254 bytes (2540 bits) of data.
CHECKSUM 2	— Checksum of BYTE0, BYTE1, and DATA.
GAP 2	— Variable gap of GCR 0. This gap is variable depending on the speed of the disk during format. GAP 2 will be constant for all sectors except for the tail gap (the gap between first and the last sectors).

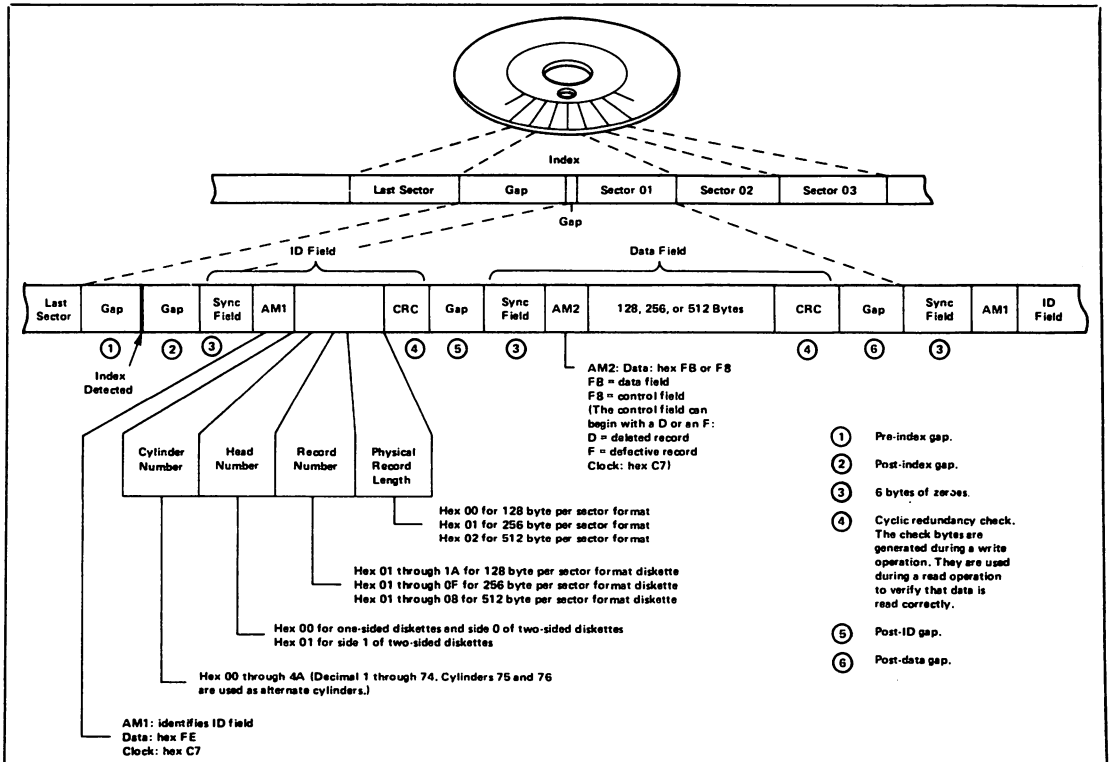


Figure C-2. MFM-Formatted Diskette

NOTE: The information in the following tables applies only to GCR diskettes that are used for the Commodore Disk Operating System, not CP/M.

BLOCK DISTRIBUTION BY TRACK

Track number	Range of Sectors	Total # of Sectors	
1 to 17	0 to 20	21	single sided, double sided
18 to 24	0 to 18	19	single sided, double sided
25 to 30	0 to 17	18	single sided, double sided
31 to 35	0 to 16	17	single sided, double sided
36 to 52	0 to 20	21	double sided
53 to 59	0 to 18	19	double sided
60 to 65	0 to 17	18	double sided
66 to 70	0 to 16	17	double sided

BAM FORMAT/DIRECTORY HEADER (1541 MODE)
Track 18, Sector 0

BYTE	CONTENTS	DEFINITION
0	18	Track of next directory block (always 18)
1	1	Sector of next directory block (always 1)
2	65	ASCII character A indicating 1541/1551/1571/4040 format
3		Double-sided Flag (ignored 1541 mode)
4		Number of sector available on track 1
5		Track 1, sector 0-7 availability map*
6		Track 1, sector 8-16 availability map*
7		Track 1, sector 17-23 availability map*
8		Number of sector available on track 2
9		Track 2, sector 0-7 availability map*
10		Track 2, sector 8-16 availability map*
11		Track 2, sector 17-23 availability map*
. . .		
140		Number of sector available on track 35
141		Track 35, sector 0-7 availability map*
142		Track 35, sector 8-16 availability map*
143		Track 35, sector 17-23 availability map*
144-159		Diskette name padded with shifted spaces [CHR$(160)]
160-161	160	Shifted space [CHR$(160)]
162-163		Diskette ID
164	160	Shifted space [CHR$(160)]
165-166		ASCII representation of 2A, which are respectively, the DOS version (2) and format type (1540/1541/1551/ 1571/4040/2030).
167-170		Shifted spaces [CHR$(160)]
171-255		Nulls [CHR$(0)], not used

BAM FORMAT/DIRECTORY HEADER (1571 MODE)
Track 18, Sector 0

BYTE	CONTENTS	DEFINITION
0	18	Track of next directory block (always 18)
1	1	Sector of next directory block (always 1)
2	65	ASCII character A indicating 1541/1551/1571/4040 format
3		Double-sided Flag: $80 = double-sided; $00 = single-sided
4		Number of sector available on track 1
5		Track 1, sector 0-7 availability map*
6		Track 1, sector 8-16 availability map*
7		Track 1, sector 17-23 availability map*
8		Number of sector available on track 2
9		Track 2, sector 0-7 availability map*
10		Track 2, sector 8-16 availability map*
11		Track 2, sector 17-23 availability map*
. . .		
140		Number of sector available on track 35
141		Track 35, sector 0-7 availability map*
142		Track 35, sector 8-16 availability map*
143		Track 35, sector 17-23 availability map*
144-159		Diskette name padded with shifted spaces [CHR$(160)]
160-161	160	Shifted space [CHR$(160)]
162-163		Diskette ID
164	160	Shifted space [CHR$(160)]
165-166		ASCII representation of 2A, which are respectively, the DOS version (2) and format type (1540/1541/1551/1571/4040/2030).
167-170		Shifted spaces [CHR$(160)]
171-220		Nulls [CHR$(0)], not used
221-237		Number of sector available on track 36 – 52 (each sector by each byte)
238	0	Number of sector available on track 53 (always 0, all sectors are allocated)
239-244		Number of sector available on track 54 – 59 (each track by each byte)
245-250		Number of sector available on track 60 – 65 (each track by each byte)
251-255		Number of sector available on track 66 – 70 (each track by each byte)

*%1 = available sector (% means binary)
%0 = sector not available (each bit represent one block)

BAM Side One (1571 mode)
Track 53, Sector 0

BYTE	CONTENTS	DEFINITION
0		Track 36, sector 0-7 availability map
1		Track 36, sector 8-16 availability map
2		Track 36, sector 17-23 availability map
. . .		
102		Track 70, sector 0-7 availability map
103		Track 70, sector 8-16 availability map
104		Track 70, sector 17-23 availability map
105-255	0	Nulls ($00) not used

%1 = available block (% means binary)
%0 = block not available (each bit represent one block)

PROGRAM FILE FORMAT

BYTE	DEFINITION
FIRST SECTOR	
0,1	Track and sector of next block in program file 1.
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 1.
2-255	Next 254 bytes of program information 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 three 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.

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.

DIRECTORY FILE FORMAT
Track 18, Sectors 1-19

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 2.0	NEW	"N0:diskette name,id"
	COPY	"C0:new file=0:old file"
	RENAME	"R0:new name=old name"
	SCRATCH	"S0:file name"
	VALIDATE	"V0"
BASIC 7.0/ 3.5	NEW	HEADER"diskette name",Iid,D0
	COPY	COPY "old file" TO "new file"
	RENAME	RENAME "old name" TO "new file"
	SCRATCH	SCRATCH "file name"
	VALIDATE	COLLECT
BOTH	INITIALIZE	"I0"

FILE COMMANDS

BASIC 2.0	LOAD	LOAD"file name",8
	SAVE	SAVE"file name",8
	VERIFY	VERIFY"file name",8
BASIC 7.0/ 3.5	LOAD	DLOAD"file name"
	SAVE	SAVE"file name"
	VERIFY	DVERIFY"file name" (BASIC 7.0 only)
BASIC 7.0 only	BLOAD	BLOAD"filename",Bbank#,Pstart address
	BSAVE	BSAVE"filename",Bbank#,Pst.add To Pen.add
	BOOT	BOOT"filename"
	OPEN	DOPEN#file#,"filename" [.Lrecord length] [,W]
	CLOSE	DCLOSE#file#
	RECORD#	RECORD#file#,record number [,offset]
BOTH	OPEN	OPENfile#,8,channel#,"0:file name,file type, direction"
	CLOSE	CLOSEfile#
	RECORD#	"P"+CHR$(channel#)+CHR$(<record#) +CHR$(>record#)+CHR$(offset)
	PRINT#	PRINT#file#,data list
	GET#	GET#file#,variable list
	INPUT#	INPUT#file#,variable list

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"
UTILITY LOADER	"&0:file name"
BURST	"U character" + character(s)

APPENDIX E
SPECIFICATIONS OF THE 1571 DISK DRIVE

STORAGE

GCR format

	single sided	double sided
Total unformatted capacity	252019 bytes	252019*2 bytes
Total formatted capacity	174848 bytes	349696 bytes
Maximum Sequential file size	168656 bytes	337312 bytes
Maximum Relative file size	167132 bytes	167132 bytes
Record per file	65535	65535
Files per diskette	144	144
Tracks per diskette	35	70
Sectors per track	17-21	17-21
Sectors per diskette	683 total	1366 total
	664 free	1328 free
Bytes per sector	256	256

MFM format

Total unformatted capacity	250000 bytes per side
Total formatted capacity	
Sector Size 128	133120 bytes per side
Sector Size 256	163840 bytes per side
Sector Size 512	184320 bytes per side
Sector Size 1024	204800 bytes per side
Maximum tracks per disk	40 per side
Sectors per track	
Sector Size 128	26
Sector Size 256	16
Sector Size 512	9
Sector Size 1024	5

INTEGRATED CIRCUIT CHIPS USED

6502A	microprocessor
(2) 65C22A	I/O
23256	32K bytes ROM
4016	2K bytes RAM
64H156/64H157	Gate Array
R/W Hybrid IC	Analog Circuit IC (MFM, GCR)

PHYSICAL DIMENSIONS

Height	76 mm
Width	216 mm
Depth	346 mm
Weight	3.5 kg

ELECTRICAL REQUIREMENTS

Voltage	North America	100–120 VAC
	Europe/Australia	220–240 VAC
Frequency	North America	60 Hz
	Europe/Australia	50 Hz
Power used		25 Watts

MEDIA

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

APPENDIX F
SERIAL INTERFACE INFORMATION

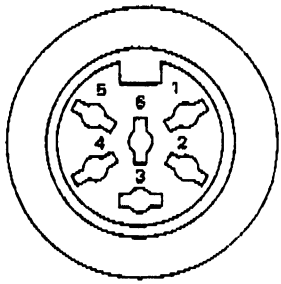
The Serial Interface consists of two 6-pin DIN Female Connectors on each drive. The second connector is for daisy chaining to other drives and/or peripherals. The voltage interface is a serial interface at TTL levels.

There are three types of operation over a serial bus—Control, Talk, and Listen. The host is the controller and initiates all protocol on the serial bus. The host requests the peripheral to listen or talk (if the peripheral is capable of talking as disk drive). All devices connected to the serial bus receive data transmitted over the bus. To allow the host to route its data to an intended destination, each device has a bus address (known as device number). Disk drive's device addresses are 8-11 (8 is normal).

Data and control signals as follows:

Pin No.	Signal	Direction	Description
Pin 1	SRQ (Service Request)	in/out	Used by fast serial bus as a bi-direction fast clock line. Unused by the slow serial bus.
Pin 2	GND (Ground)		Logic ground
Pin 3	ATN (Attention)	in	The host brings this signal low which then generates an interrupt on the controller board. The attention sequence is followed by a device address. If the device does not respond within a preset time the host will assume the device addressed is not on the bus.
Pin 4	CLK (Clock)	in/out	This signal is used for timing the data sent on slow serial bus (software clocked).
Pin 5	DATA	in/out	Data on the serial bus is transmitted one bit at a time (software toggled).
Pin 6	RESET		This line will reset the peripheral upon host reset.

The 6-pin DIN connector looks like (from outside):



In detail, the 1571 serial bus supports the new FAST serial communication as well as standard (SLOW) serial communication.

The important difference between the FAST serial bus and the SLOW serial bus is the incorporation of the hardware controlled lines for the CLOCK and DATA lines. Fast serial communication is transparent to any peripheral connected to the serial bus that does not contain the necessary hardware or software to talk at fast speed.

To remain compatible with the SLOW serial bus all bytes sent under attention are sent slow.

APPENDIX G
DISK OPERATING SYSTEMS: Commodore and CP/M

The Disk Operating System (DOS) is the software that provides an interface between the host computer and the drive's internal mass storage devices and diskettes. The DOS keeps track of the file management details necessary to create, modify, and delete files. It also monitors the amount of free space on a diskette, keeps track of the file names in a directory, and remembers where each is so that the files won't overlap.

Commodore DOS

The 1571 is an 'intelligent' device. That means it contains its own CPU and ROM, which allow it to function, in part, without the aid of the host CPU. Despite that, the 1571 is considered a 'slave' device, existing primarily to perform tasks requested by a host computer.

An example of the host/drive relationship is when the host requests a directory of all files on a diskette. The host sends the directory command to the drive, then simply waits for the drive to send back the information. The drive, on the other hand, must first check to see if the diskette is inserted, start it spinning, wait until it is spinning at the proper speed, move the read/write head to the track where the appropriate information is stored, read it, sort it out, format the information properly, and send it down the serial bus to the host.

The ROM software in the 1571 can be broken down into two major components—the DOS and the Controller. As mentioned above, the DOS takes care of the file management details. The information for those files is arranged on the diskette in uniformly-sized chunks called sectors. The DOS only understands information in the form of sectors. It is the job of the Controller to actually go out and read sectors from the diskette and hand them to the DOS.

The C64 computer or the C128 computer in C64 mode contain no DOS of their own. All of the DOS software is in the drive itself. That's also the case with the C128 in C128 mode, the difference being that the relationship between the 1571 and the C128 is friendlier when the computer is in C128 mode. As a result, some of the more common commands such as SAVE/DSAVE, LOAD/DLOAD, and NEW/HEADER are easier to execute.

For instance, to format a disk with a C64 or with a C128 in C64 mode, you must send the following command (after opening channel 1 to the drive's command channel 15): PRINT# 1,"NEW0:name,id". With the C128 in C128 mode, the command is this: HEADER "name,id",D0 or HEADER "name", Iid, D0.

The latter method is easier because the C128 automatically opens the channel and sends the correctly formatted command string to the drive. It then requests status information and reports any errors. To check the error in C64 mode, you would have to type several more lines.

CP/M DOS

In CP/M mode, the DOS resides inside the host computer, and only the Controller portion of the 1571 is used. The DOS code is loaded into the host's RAM when the CP/M system is booted, and is executed by the host's CPU. All file handling functions are performed within the host. Commands sent to the drive in CP/M mode request individual sectors of data from the diskette. These sectors are located and read by the Controller in the 1571, and handed to the CP/M DOS over the serial bus to the C128.

BESCHEINIGUNG DES HERSTELLERS

Hiermit wird bestätigt, dass die Floppy Disk

COMMODORE 1571

in Übereinstimmung mit den Bestimmungen der

Amtsblattverfügung Nr. 1046/1984

funk-entstört ist.

Der Deutschen Bundespost wurde das Inverkehrbringen dieses Gerätes angezeigt und die Bereichtigung zur Überprüfung der Serie auf Einhaltung der Bestimmungen eingeräumt.

COMMODORE BÜROMASCHINEN GMBH

CERTIFICATE OF THE MANUFACTURER

Herewith we certify that our device Floppy Disk

COMMODORE 1571

complies to the regulations

Amtsblattverfügung Nr. 1046/1984

concerning radio interference.

The German Bundespost has been informed that this unit is on the market and it has the right to check on the mass production limits are kept.

COMMODORE BUSINESS MACHINES LIMITED

COMMODORE SALES CENTERS

Commodore Business Machines, Inc.
1200 Wilson Drive
West Chester, PA 19380, U.S.A.

Commodore Business Machines Limited
3370 Pharmacy Avenue, Agincourt
Ontario, M1W 2K4, Canada

Commodore Business Machines (UK) Ltd.
1, Hunters Road, Weldon
Corby, Northants, NN17 1QX, England

Commodore Bueromaschinen GmbH
PO BOX 710126, Lyonerstrasse 38
6000 Frankfult 71, West Germany

Commodore Italiana S.P.A.
Via Fratelli Gracchi 48
20092 Cinisello Balsamo, Milano, Italy

Commodore Business Machines Pty Ltd.
5 Orion Road
Lane Cove, NSW 2066, Australia

Commodore Computer B.V.
Marksingel 2e, 4811 NV BREDA
Postbus 720, 4803 AS BREDA, Netherlands

Commodore AG(Scheweiz)
Aeschenvorstadt 57
Ch-4010 Basel, Switzerland

Commodore Computer NV-SA
Europalaan 74
1940 ST-STEVENS-WOLUWE, Belgium

Commodore Data AS
Bjerrevej 67
Horsens, Denmark

COMMODORE

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380
Commodore Business Machines, Limited
3370 Pharmacy Avenue • Agincourt, Ontario, M1W 2K4

252095-04

Printed in Taiwan