commodore 64 disk companion

essential routines for commodore disk users

david lawrence and mark england



commodore 64 disk companion

essential routines for commodore disk users

david lawrence and mark england

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

Copyright © David Lawrence and Mark England, 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

British Library Cataloguing in Publication Data
Lawrence, David, 19---
Commodore 64 disk companion.
1. Commodore 64 (Computer)
2. Data disk drives
I. Title II. England, Mark
001.64'42 QA76.8.C64

ISBN 0–946408–49–1

Cover design by Grad Graphic Design Ltd.
Illustration by Steiner Lund.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

		Page
Notes on Program Listings		vii
Introduction		ix
1	Disks and Disk Drives	1
2	Setting up Your System	9
3	Saving and Loading Programs	15
4	Disk Housekeeping Commands	21
5	Pattern Matching	47
6	The Error Channel	51
7	Sequential and User Files	57
8	Program Files	75
9	Relative Files	95
10	Random Files	107
11	The Disk Directory	121
12	Machine Code Programming Commands	131
13	Changing Device Numbers	137
APPENDICES		
A	Disk Error Messages	139
B	Additional Machine Code Commands	141
C	DOS Support Commands	143
Index		145

Contents in detail

CHAPTER 1

Disks and Disk Drives

Introduction — the layout of a floppy disk — the disk drive.

CHAPTER 2

Setting up Your System

Connecting up the system — switching on the system — working with disks — inserting and removing disks — problems while the system is running — turning off the system.

CHAPTER 3

Saving and Loading Programs

How often should programs be saved — the SAVE and LOAD commands with a disk drive — SAVEing and LOADing with more than one disk drive — an easy technique to simplify saving programs — the use of VERIFY — overwriting files with '@0:' — saving what you have SAVEd.

CHAPTER 4

Disk Housekeeping Commands

Introduction — OPEN — CLOSE — PRINT # — NEW — SCRATCH — RENAME — COPY — INITIALIZE — VALIDATE.

CHAPTER 5

Pattern Matching

What is pattern matching — patterns with '*' — patterns with '?' — combining '*' and '?' — using pattern matching with commands.

CHAPTER 6

The Error Channel

What is the error channel — getting messages from the error channel — using the error channel — summary of error channel.

CHAPTER 7

Sequential and User Files

What is a sequential file — OPENing a sequential file — printing and retrieving data: INPUT # and GET # — GET # — detecting the end of file — ending output or input with CLOSE — examples of the use of sequential files — summary of user files — summary of sequential files.

CHAPTER 8

Program Files

What is a program file — the structure of a BASIC program file — using program files for other purposes — output of program files to printer — merging programs using program files on disk — renumbering a program file on disk.

CHAPTER 9

Relative Files

Introduction — creating a relative file — OPENing a relative file — specifying a position in a relative file — writing to a relative file — reading from a file — CLOSEing a relative file — using relative files.

CHAPTER 10

Random Files

OPENing a random file — position of data in the buffer — writing data to the buffer — writing data to disk — loading data from the disk to the buffer — getting the data back into the 64 — marking and freeing sectors on the disk — executing machine code from the disk — two utility programs using random files.

CHAPTER 11

The Disk Directory

The format of the directory — reading the directory — repeating a process on multiple files.

CHAPTER 12

Machine Code Programming Commands

Reading the memory of the disk drive — writing to memory — executing machine code in the disk drive memory.

Notes on Program Listings

For the sake of clarity, control characters in the program lines which follow have been set out as follows:

CURSOR UP	[CU]
CURSOR DOWN	[CD]
CURSOR LEFT	[CL]
CURSOR RIGHT	[CR]
CLEAR SCREEN	[CLR]
HOME CURSOR	[HOME]
REVERSE ON	[RVS]
REVERSE OFF	[RVO]

Colours are represented by the colour name in square brackets. Thus the control character for yellow is:

[YEL]

Introduction

The advent of inexpensive, reliable disk drives for home use represents a revolution in what can be achieved with a microcomputer. Any computer, no matter how large or small, comes into its own only when it is able to access data far more quickly than can be achieved with the trusty but tortoise-like cassette drive. Without fast mass storage, substantial programs become tiresome burdens which can take up to 15 minutes to load. The handling of data becomes a nightmare, with its constant necessity to rewind or change tapes and the wait while data is slowly added to memory. In short, the best computer is only as good as the means it uses to store data. The disk drive is far from a luxury, it is an essential component of an effective microcomputer system.

Nowhere is this more true than in relation to the Commodore 64. The success of the 64 as far more than a games machine or a computing toy is based to a large extent on Commodore's ability to produce a disk drive which works to the highest standards and yet is within the price range of anyone who takes their computing at all seriously.

But simply to buy a disk drive is not the solution to the problem of enlarging the scope of the 64. Like any disk drive, the 1541 has its own way of working, and its own habits which are neglected at its owner's peril. Poorly thought out working methods with a disk often mean that much of its potential can remain unrealised. Carelessness, or lack of understanding, can mean that valuable data or programs are irretrievably lost. And yet, for all that, the 1541 is a well designed piece of equipment, full of features that many disk drive owners can only dream about.

This book is an attempt on our part to share some of the many discoveries that we have made about the 1541 disk drive, the enjoyment that comes with using it and the power that is released when it is used *well*.

Note on the applications of this book

This book is designed specifically for the use of Commodore 64 owners who are working with one or more 1541 disk drives. Most of the commands and techniques contained in the book will be applicable to the older 1540 drive or to users of the VIC 20 computer. Having said that, the material presented here has *not* been tested against this equipment and occasions will arise when facilities described here are not available. In relation to the

VIC 20, an adjustment must be made to the speed at which the 1541 disk drive runs using the 'UI–' user command, as described in the 1541 disk manual. Users of the 1540 drive will not, according to the 1540 manual, be able to employ the relative file techniques described in Chapter 9.

CHAPTER 1
Disks and Disk Drives

1) Introduction
2) The layout of a floppy disk
3) The disk drive

Section 1. Principles of magnetic disk storage

Disk storage, like tape storage, depends on the fact that a thin layer of a ferrous metal compound is capable of being magnetised and demagnetised. When magnetised by proximity to a magnetic field, such compounds have the capacity to maintain their magnetised state.

In practice, the fact that magnetism is employed is irrelevant (unless you are in the habit of putting magnets on top of your disks). What *is* significant is that, by some means or other, such ferrous compounds can record the fact that something has happened to them — they can record information. When spread thinly, tiny amounts of such compounds are capable of recording the fact that an electromagnet has passed near to them, the degree of magnetism it possessed and the direction of the current in it. Having been magnetised, the film can be read by an electromagnet which has no current being fed through it, since it is a property of electromagnets that they produce an electric current when passed through a magnetic field — even the minute field stored by a thin film of ferrous compound.

Provided, then, that an electromagnet can be made to pass over the film sufficiently closely; and the state of the electromagnet can be changed in a controlled manner; and provided that this can be done with sufficient accuracy that the same position can be returned to time and time again; then the magnetic qualities of a thin layer of a ferrous compound can be used to store information.

In the context of a computer disk drive, the film of ferrous compound is held on the surface of a 5⅛ inch piece of thin, flexible plastic. The electromagnet is provided by the disk drive in the form of a tiny magnetic recording head capable of being moved with great accuracy in a straight line between the centre of the disk and its circumference. The movement of the head over the film is provided by the revolving of the disk. In essence

then, a disk system consists of a revolving disk and an electromagnet which can scan in and out across it as it moves.

The advantage of this system compared to tape is not simply the speed with which a single block of information can be stored — some tape systems are very fast indeed. The real power of the disk system lies in the speed at which it can *find* the information or the place where it is to be stored. A good analogy is the difference between an ordinary audio cassette recorder and a long playing record. Provided that in both cases you know where the information you want is stored, ie which track of an album you wish to play, the disk will provide you with much faster access since you are able to move the needle directly in towards the centre of the turntable until it is positioned correctly. Once there, another kind of movement, ie the revolutions of the disk itself, allows you to recall what you want. With the tape system, you have only one kind of movement available to you and you will have no choice but to fast-wind until the correct place is found.

2. The layout of a floppy disk

Unlike a long playing record, the disks used by the 1541 disk drive do not come with individual tracks laid out in a permanent form. The film of recording medium is, or should be, of a uniform consistency over the whole of the disk's surface. Dividing up the disk into easily identifiable 'tracks' for the storage and retrieval of data is a task undertaken by the disk drive itself in a process called 'formatting'.

The purpose of the formatting process is to mark the disk magnetically with a series of areas called 'sectors', roughly three quarters of an inch long. Sectors fall into rings which, as with long playing records, are known as tracks, thirty-five of them in all, with the number of sectors varying according to the distance of the track from the centre of the disk — the further from the centre, the longer the track and the more sectors it will contain.

This simple process is accompanied by some more subtle ones which will enable the finely tuned disk-drive mechanism to identify its place on the disk and move the recording head. Each sector is created with an area of 256 bytes for the storage of data but also has written into it other information; such as the identification number of the disk, the number of the track on which the sector falls, and the number of the sector within the track, plus some standard data which the disk drive will later use to check that it is properly synchronised with the disk as it turns.

Apart from the blank sectors prepared for the reception of data, an area of the disk (track 18) is reserved for the use of the 'directory' or list of files which the disk will eventually contain. When the disk is first formatted,

only the first two sectors of track 18 will be used for this purpose — other sectors will be brought in as programs are added. Included in the directory is an area of housekeeping information known as the Block Allocation Map. The purpose of the BAM is to record, for every sector on the disk, whether that sector is available for the storage of information or if it is occupied by part of an existing file.

The BAM is positioned in the first sector (sector zero) of track 18 and consists of 140 bytes of disk space. This space is itself divided up into 35 sets of four bytes each. The first byte of the group indicates the number of sectors available on one of the disk's 35 tracks. The next three bytes record the individual state of sectors 0–7, 8–16, and 17–23 of the corresponding track. If sector zero of the corresponding track is available for storage, for instance, bit zero of the value stored in the second of the four bytes will be 'set' (ie equal to one rather than zero). If the value of a whole byte (ie eight bits) is zero, so that none of its bits is set, this would indicate that the eight sectors it is recording are all in use by a current file. You may note that the BAM makes provision for recording 24 sectors (0–23), even though there is a maximum of 21 on the long outer tracks and less as the tracks near the centre. The BAM overcomes this potential difficulty by registering these non-existent tracks as unavailable when the disk is formatted.

From **Table 1.1** it can be seen that if there are more than eight files on the disk then another sector of track 18 will need to be added to the directory. The new sector will have the same format as sector 1 shown in the table. The last sector of the directory is indicated by the fact that the first two bytes, which normally indicate the address of the following sector, point to track zero, sector 255, which does not exist.

The structure of a file on the disk

Having set up the disk structure and the initial directory, the disk is now ready for the storage of information in units which are known as 'files'. The two types of file which are used most often are the program file, which is what is created when a program is SAVEd, and the sequential file, which is created when a file is OPENed for the storage of items of data. Both these

Table 1.1: Structure of the Block Allocation Map

BYTE	REMARKS
1	Holds a number indicating the number of blocks available on this track.
2	Sectors 0–7, bit 0 represents sector 0
3	Sectors 8–15, bit 0 represents sector 8
4	Sectors 16–23, bit 0 represents sector 16

Table 1. 2: Structure of the Directory Track (Track 18)

Track 18, Sector 0

BYTE	REMARKS
0	Track of next directory block (always track 18)
1	Sector of next directory block (sector 1)
2	A valueof 65 here indicates that the disk is formatted for 1541
3	Not used, normally has value 0
4	First byte of Block Availability Map (BAM), containing the 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 sectors 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
	. . . . .
143	Track 35, sector 17–23 availability map (this is the last byte of the BAM)
144–161	Disk name padded with shifted spaces (CHR$(160))
162	First byte of the disk ID
163	Second byte of the disk ID
164	Not used, normally has value 160
165	Character '2' indicates DOS version
166	Character 'A' indicates DOS version
167	Not used, normally has value 160
168–170	Not used, value indeterminate

Track 18, sector 1

BYTE	REMARKS
0	Track of next directory sector (normally 18 but 0 if end of directory)
1	Sector of next directory sector (255 means the end of the directory)
2–31	File entry 1 — for details see Chapter 11
32	Not used
33	Not used
34–63	File entry 2
64	Not used
65	Not used
66–95	File entry 3

96	Not used
97	Not used
98–127	File entry 4
128	Not used
129	Not used
130–159	File entry 5
160	Not used
161	Not used
162–191	File entry 6
192	Not used
193	Not used
194–223	File entry 7
224	Not used
225	Not used
226–255	File entry 8

types are stored on the disk in exactly the same way, so we shall take for an example the SAVEing of an ordinary program file.

Sequence of events in SAVEing a program file

1) The SAVE command is entered by the user and the 64 instructs the disk drive to open a program file of that name.

2) The disk drive checks its directory to see that a file of that name does not already exist.

3) Provided that there is no file of the same name, the disk drive records the filename in the directory with a starting track and sector of 0,255 — ie a non-existent track.

4) Using the BAM, which is always kept in the disk drive memory, the drive begins to search for the nearest track to the directory track, either out towards the edge of the disk or in towards the centre, which has a free sector (SECTOR1) and marks that sector as allocated in the BAM.

5) Having found SECTOR1, the drive records its position and then accepts 254 bytes of the program from the 64 and places them into a buffer in the disk memory.

6) Another search is now made for the nearest free sector other than the one discovered in step 4 (SECTOR2).

7) The address of SECTOR2 (discovered by step 6) is now written into the first two bytes of the disk buffer created in step 5.

8) The whole of the contents of the buffer are now written into SECTOR1 (discovered in step 4).

9) SECTOR2 is now regarded as SECTOR1 and the process is repeated from step 5 until the 64 informs the disk drive to close the file, by which time the whole of the program has been received.

10) For the final bufferful of data, the address of the next sector (the first two bytes) is set at track zero, sector 255, to mark the end of the file.

11) The directory entry for the new file is altered to record the sector used for the beginning of the file and the number of bytes contained in the file.

Table 1. 3 : Allocation of Sectors on a typical track

TRACK = 15

0	SEQ. F	,PRG BLOCK 2
1	EOF. F	,PRG BLOCK 2
2	LIST T&S. F	,PRG BLOCK 14
3	APRICOT. F. WOW	,SEQ BLOCK 1
4	LIST T&S. F	,PRG BLOCK 12
5	SEQ ARRAYS. F	,PRG BLOCK 4
6	TEST1	,SEQ BLOCK 1
7	SCREEN SAVE	,PRG BLOCK 1
8	PROG READ	,PRG BLOCK 1
9	SCREEN	,PRG BLOCK 1
10	SEQ. F	,PRG BLOCK 3
11	EOF. F	,PRG BLOCK 3
12	SEQ ARRAYS. F	,PRG BLOCK 3
13	SEQ ARRAYS. F	,PRG BLOCK 5
14	LIST T&S. F	,PRG BLOCK 13
15	SCREEN	,PRG BLOCK 3
16	LIST T&S. F	,PRG BLOCK 11
17	SCREEN	,PRG BLOCK 4
18	PROG READ	,PRG BLOCK 2
19	SCREEN	,PRG BLOCK 2
20	SEQ. F	,PRG BLOCK 4

By the time programs have been written to the disk, removed and overwritten during the course of time, the structure of the disk will appear most confused to the human eye, with a mishmash of sectors on each track allocated to a variety of programs. Provided that nothing happens to corrupt the directory, however, or the two bytes at the beginning of each sector which record the position of the next sector of the file, the disk drive will always be able to find the start of a file that it holds and read that file sector by sector without difficulty. An indication of the kind of disk structure that will be found on a well-used disk is given by **Table 1.3**, which is the output of a program contained in Chapter 10 of this book. The table displays the contents of a single track of one of the disks used in the development of the book.

3. The disk drive

So far, we have looked at disks and their layout but taken the activities of the 1541 drive itself for granted. It would be wrong to conclude this chapter, however, without a reminder that the 1541 is an extremely sophisticated and powerful piece of equipment, driven by a 6502 microprocessor and its own internal Disk Operating System program which is as large as the ROM of the 64 itself. The advantage of this is that, unlike disk drives for the majority of other personal microcomputers, the use of the 1541 drive requires no memory to be set aside by the host computer to run it. Rather than relying on the 64 for detailed instructions as to the handling of its affairs, the 1541 normally requires only to be informed of the name of the task to be carried out. It will then proceed without further help on one of the complex procedures it is capable of performing. For this reason, the 1541 is known as an 'intelligent drive'.

CHAPTER 2
Setting up Your System

1) Connecting up the system
2) Switching on the system
3) Working with disks
4) Inserting and removing disks
5) Problems while the system is running
6) Turning off the system

1. A beginner's guide to connecting up the system

To operate a disk system you need at least one 1541 disk drive, together with its connecting lead, to add to your existing system. You will also require a set of 5¼ inch floppy disks — these normally come in boxes of ten but can be bought singly. It should not be forgotten that you will also require an extra power socket from which to run the disk drive.

Assuming that your 64, its power unit and the television/monitor which you normally use are all properly connected, and that the power is OFF to all the equipment, follow this procedure:

1) Turn the 64 around so that the back of the machine faces you.

2) Reading from right to left, you will find two rectangular slots in both of which may be seen the edge of the 64's printed circuit board.

3) To the left of these two slots are two circular plug sockets. The one on the right is a six-pin socket and is called the serial bus. It is the means by which the 64 communicates with outside devices such as printers and disk drives.

4) If you have a printer connected to the serial bus socket, disconnect it for the time being.

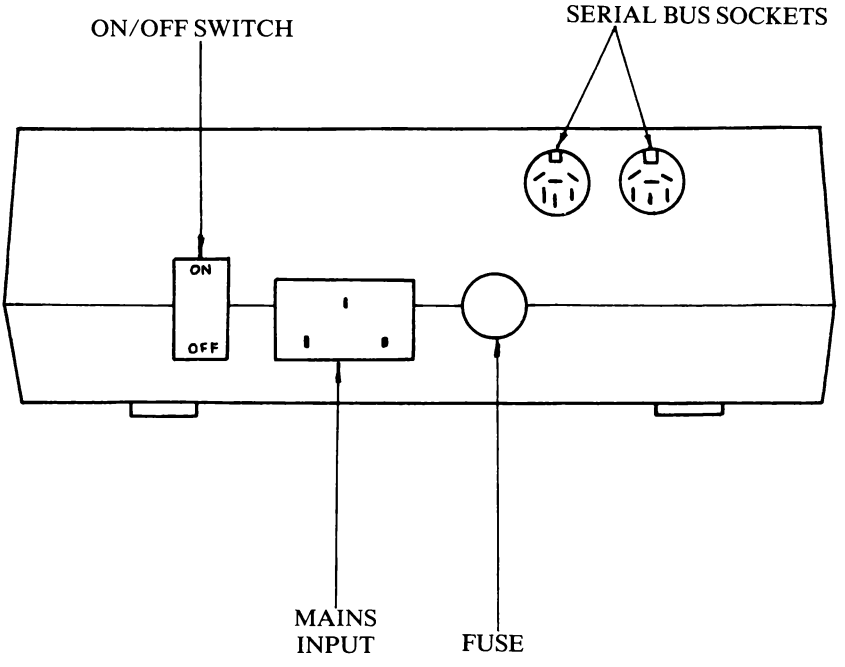
5) Plug into the serial bus socket one of the small round plugs (DIN plugs) on the connecting lead which came with the disk drive.

6) You can now turn the 64 round so that the keyboard is again facing you.

7) Take the disk drive and place it next to the 64 so that its back is facing you.

8) On the back of the 1541 you will find the connections shown in **Figure 2.1**

Figure 2.1: Back Panel of 1541 Disk Drive



9) Take the other end of the connecting cable you have just plugged into the back of the 64 and plug it into either of the sockets marked 'SERIAL BUS' in the illustration.

10) If you have a second disk drive, plug its connecting cable into the other socket. (If you wish to operate more than one drive you will also have to read Chapter 13 for advice on altering the device number of one of the drives if the drives have not been permanently modified.) A whole series of drives can be chained in this way if desired.

11) If you have a Commodore compatible printer, you may now connect it to the spare serial bus socket on the last disk drive to be connected to the system.

12) Ensure that the 1541, the Commodore 64, the TV/monitor (and the printer if connected) are all switched off. Plug the mains connection lead

into the back of the 1541 and then connect the other end of the lead to the mains. Switch on the mains power to the 64, TV/monitor (and printer if connected). Do not at this stage switch on the equipment itself.

13) Turn the 1541 around so that its front is facing you, being careful not to snag any of the leads under the drive.

14) On the front of the 1541 you will see a small bar protruding. The bar will be in one of two positions:

a) slightly below the slot which goes across the front of the drive or

b) slightly above it.

15) If the bar is in position a), press it in gently with your finger and allow it to move gently upwards on its spring.

16) The disk drive door is now open. To make absolutely sure that the drive does not contain a disk or the square of cardboard used to protect the internal mechanism during carriage, close the door by pressing gently down until it locks and then opening it again.

17) If a disk or the protective card is now visible, remove by sliding gently towards you.

18) Switch the disk drive on (and the printer if connected). Both the green and red lights on the front of the drive will come on, the drive will whirr for a second or so, then the whirring will stop and the red light will go out. If the red light does not go out, switch the drive off, check the connections and repeat this step. If the red light still remains on, consult your dealer.

19) Switch on your 64. (You should see the same sequence as described under step 18.)

20) Switch on your TV/monitor and tune it to the output of the 64.

21)You are now ready to use your Commodore 64 disk system.

2. Switching on the system once it is set up

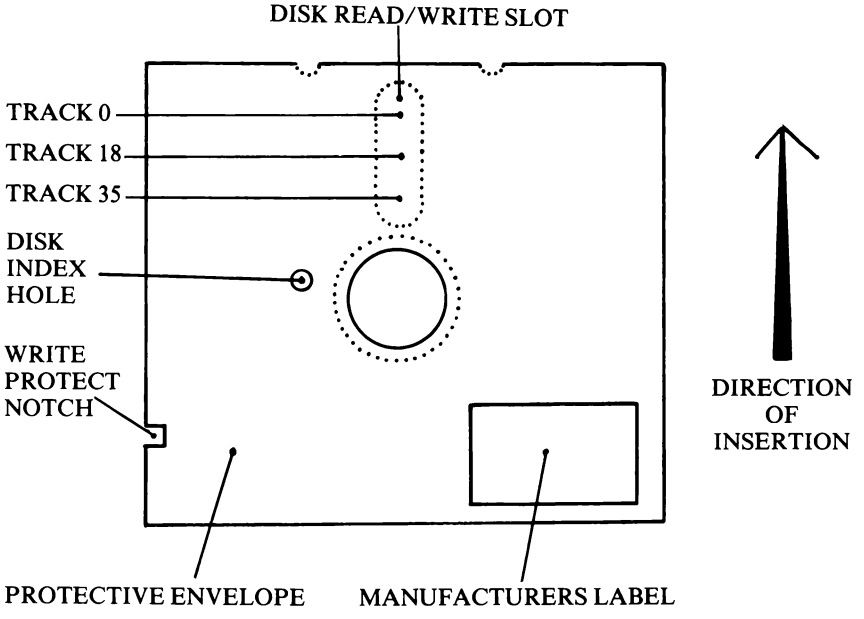
The recommended sequence for switching on a disk system once it has been properly set up is:

PRINTER→DISK DRIVE→COMPUTER

3. Working with disks

Given in **Figure 2.2** is the layout of a typical 5¼ inch floppy disk.

Figure 2.2: A 5¼" disk



Such disks will serve you well if you observe a few simple rules:

1) The disk is *never* removed from its protective envelope. It can revolve quite satisfactorily within the envelope and will be read through the READ/WRITE SLOT.

2) Floppy disks, or flexi-disks, are not designed to be deliberately flopped or flexed. If, by accident, a disk is momentarily bent slightly, it should survive but there is no guarantee of this. There is always a slight amount of give in the disk when inserting it into the drive or removing it — do not worry about this, simply do not go to extremes.

3) Your disks will have been supplied with a protective sleeve which covers most of the disk envelope and protects the READ/WRITE SLOT. When not in use, disks should always be replaced in the sleeve. Disks should preferably be stored upright in a plastic (ie non-magnetic) box specially designed for the purpose. Never leave disks lying on any surface, out of their sleeves. It is common to see disks carelessly left lying about unpro-

tected in the belief that, provided that the READ/WRITE SLOT on the 'front' of the disk (the side with the labels) is not touched, all will be well. Since the 1541 actually reads the disk from the back, this is unlikely to be true.

4) Disks should never be exposed to a magnetic field, which includes leaving them on the top of the disk drive or the TV/monitor.

5) Disks should never be exposed to dampness or extremes of temperature, which includes leaving them in direct sunlight.

6) When writing disk labels either write the label *before* attaching to the disk or use a felt-tip pen which does not place pressure on the disk through the protective envelope.

7) It should go without saying that you should never touch the READ WRITE SLOT of the disk.

8) Cheap disks, like cheap tapes, can lead to disaster. Only you can decide how much your programs and data are worth to you.

4. Inserting and removing disks

To place the disk in the drive, first ensure that the disk drive door is open, then orient the disk so that the manufacturer's label is upwards and the READ/WRITE SLOT end of the disk is towards the 1541. Push the disk gently into the horizontal slot on the front of the drive. If the disk catches slightly as it is pushed home, do *not* attempt to force it — remove it and try again. If you continue to have problems, check to ensure that another disk is not caught in the drive.

Provided that no problems are encountered, the disk should be pushed fully home until no part of it protrudes and it remains in the drive without pressure from the fingers. Finally, close the disk drive door — the disk drive cannot be accessed by the 64 until this has been done.

Disks are removed from the drive simply by opening the disk drive door, when the disk will slide out approximately an inch. If the disk does not appear, close and open the door again. Disks which are reluctant to come out of the drive can just be reached with the fingers, though no force should be used.

If disks continually stick in the drive, either there is a problem with the disks (such as a label sticking out over the edge) or the drive is faulty. Never poke inside the drive with any kind of tool to try to free a disk — especially if the drive is connected to the mains.

Never open the door of the disk drive while the red light is on and the drive motor is running, or damage to the disk may result. Note that some editions of the 1541 manual incorrectly state that the disk may not be removed while the *green* light is on. The green light is the disk power indicator and the only way to extinguish it is to switch off the power to the drive. On no account do this before removing the disk.

5. Problems while the system is running

Like any other complex piece of equipment, it is possible for the 1541 to lapse momentarily from its usual high standard of reliability. It is possible, for instance, for errors to be encountered in the reading or writing of a program, or for some other problem to arise which prevents a disk command being properly carried out. In this case the red drive light will flash on and off and you should repeat the procedure which led to the error if you are sure that the mistake is not your own —see Chapter 6.

In some circumstances, it is possible for a situation to develop where the 64 and the disk drive simply refuse to communicate with one another. Following the recommended switching-on procedure, the solution would be to remove the disk, to switch off the 64 and the disk drive and then to switch them on again in the correct order.

If the 64 contains a program which you are trying to save then our experience is that removing the disk and switching off the 1541 (and any other device connected to the serial bus, such as a printer), and then switching it back on again will almost invariably solve the problem. It is as well to remember that, if all else fails, the Datassette recorder may well be capable of redeeming the situation by saving the program until the disk system can be sorted out.

It is unlikely that the 1541 will give you many problems if you remember that it is a precision engineered machine which, unlike the 64, has moving parts which should not be subject to vibration, sudden shocks or excessive heat (including direct sunlight).

6. Turning off the system

Though it seems a small point to be given a separate section, *do* always check that the disk drive door is open and the drive empty before switching off the drive. In actual fact, disks are seldom damaged by being left in the drive when it is turned off, but it *can* happen.

CHAPTER 3
Saving and Loading Programs

1) How often should programs be saved
2) The SAVE and LOAD commands with a disk drive
3) SAVEing and LOADing with more than one disk drive
4) An easy technique to simplify saving programs
5) The use of VERIFY
6) Overwriting files with '@0:'
7) Saving what you have SAVEd

The first use that anyone makes of a disk is to store programs. There is no doubt that, if you enjoy computing and use your 64 more than occasionally, the difference in speed with which you can access programs makes the cost of a disk drive worthwhile compared to a cassette recorder. At the same time, it is always surprising how little care most people take in the keeping of programs that they have spent long periods developing, failing to save regular updates when a program is being developed, failing to check that a program has been properly saved, keeping only one copy of important programs and abusing disks by leaving them around exposed to the elements. Given below are one or two common-sense rules when it comes to saving programs.

Section 1. How often should programs be saved

As you develop new programs, SAVE them regularly. Like any other microcomputer, the 64 can lose programs if there is a momentary surge in the electricity supply, or if someone kicks the plug, or even because in your programming you manage to upset the 64's equilibrium. How much work you will have lost will depend on how long it has been since you last saved your program. If a program is being entered rapidly, you should not normally expect to enter lines for more than 15 minutes without resaving the program. When a program is being debugged, so that relatively fewer changes are being made, perhaps you might increase that period to half-an-hour. It really depends on how much you are prepared to lose, but you can

depend on the fact that if you do not save programs regularly you will, sooner or later, lose an important program that has taken a long time to enter.

2. The SAVE and LOAD commands with a disk drive

In order to embark on a policy of SAVEing programs regularly, you need first to know the command which will store a program on the disk drive. If you have previously been working with a Datassette cassette recorder, then you will have become used to the format:

SAVE "< PROGRAM NAME> "

to SAVE a program or:

LOAD "< PROGRAM NAME> "

when LOADing a program back into memory.

With your disk drive installed, the situation changes slightly. While the 64 can work perfectly well with the 1541, it is designed on the assumption that it will be used with a cassette recorder. This fact allows users of a tape recorder to get away without specifying one very important piece of information, namely the number of the device on which the program is to be stored. The device number of the cassette recorder is one, and when the instruction:

SAVE "PROGRAM"

is entered, the 64 assumes that what is meant is:

SAVE "PROGRAM",1

When working with disk, the programmer cannot rely on the 64 to put this important piece of information in, so the format of the SAVE command will normally be:

SAVE "< PROGRAM NAME> ",8

and of LOAD:

LOAD "< PROGRAM NAME> ",8

3. SAVEing and LOADing with more than one disk drive

When a single disk drive is bought from the dealer, it is set up internally to think of itself as device eight, and it will respond to any instructions addressed to a device with that number, like the SAVE and LOAD commands in the last section. An increasing number of people, however, are discovering the advantages of running more than one disk drive. With more than one drive, however, a problem arises in that the drives *must* know which of them is being addressed at any one time.

To overcome this, 1541 drives are built with the ability to change their device numbers so that a command can, for example, be issued in the form:

```
SAVE "PROGRAM",9
```

to access one of the drives, leaving device eight completely untouched. There are two ways in which disk drive device numbers can be changed, in hardware and in software, that is to say you can either make a modification to the equipment itself or you can use a program to make a temporary change. Of the two, if you are going to be permanently using more than one drive, the hardware solution is by far the best. It normally involves making a small cut in a single track of the printed circuit board inside the drive. Details of this are given in the 1541 manual but they are not very clear and, frankly, we would recommend that when you purchase your second 1541 you do so on the understanding that the dealer will do the two minutes' work involved. If your dealer baulks at the idea then perhaps you might consider finding one who knows what he is doing.

To change the device number of a disk drive with a program is not difficult (see Chapter 13), but it can become tiresome, since it has to be done every time the drive is switched on. Even so, if you are merely borrowing a friend's drive for the day, the software solution is a better one than ripping his drive to pieces.

4. An easy technique to simplify saving programs

To make saving a program easier and to encourage yourself to *do* it, it is a good idea to build a program saving facility into each program you develop, along the following lines:

```
1 GOTO 3
2 SAVE "@0:PROGRAM NAME",8 : VERIFY
"PROGRAM NAME",8 : STOP
3 REM
```

Including such a routine in a program has the virtue that you are unlikely to save the program under the wrong name due a typing error, it can be saved simply by entering GOTO 2: as an added bonus, it means that all your programs can be started with a uniform GOTO 1 if you do not wish to use RUN and wipe out any stored variables.

Two features of this routine need some explanation, the command VERIFY and the modifier '@0:' at the beginning of the program name.

5. The use of VERIFY

One of the main reasons for building the SAVE routine into the program as shown is that it can then be combined with VERIFY. The purpose of VERIFY is to check that a program stored on a specified device is the same in every respect as the program currently in memory, ie that a program has been correctly SAVEd. The format of verify is:

VERIFY "< PROGRAM NAME> ",< DEVICE>

where PROGRAM NAME is the name of a program stored on the device. Note that it is not important that the name of the program on the disk is the same as the name that you have allocated to the program in memory. The name of the program is stored on the disk's directory but not with the program itself, and no name is stored in the memory of the 64 for the current program. All you are doing is giving the disk drive the information to find a particular file.

Unlike the cassette recorder, the disk drive requires no work from the user when VERIFY is employed. In the little SAVE routine in the previous section, the drive will automatically search out the program which has been SAVEd without the user intervening.

6. Overwriting files with '@0:'

In one respect the disk drive is a little less easy to use than the Datassette recorder. When you wish to store a program for a second time on tape, all that you have to do is rewind the tape and issue the SAVE command — the previous program will be overwritten. Not so with the disk drive, for it is specifically designed to prevent you from making the mistake of accidentally overwriting a file by unwittingly SAVEing another of the same name. This is fine in most circumstances but when successive versions of a program are SAVEd it can become a little tiresome. The Disk Operating System (DOS) provides a facility to overcome this problem in the form of

the modifier '@0:' attached to the front of the name of a file (whether a program file or any of the other kinds described later — with the exception of relative files).

When the DOS comes across a filename which begins with '@0': it immediately scans the current disk to see whether there is a program with the same name as the specified filename *less* the '@0:'. If there is not, then the program is stored normally. If there *is* a program of the same name, the program being SAVEd replaces it on the disk — the previous version will not be recoverable since it is overwritten.

A note of caution has to be sounded over the use of '@0:', due to the fact that the routine which runs the facility has a 'bug'. On disks which are becoming full, you will sometimes find that the use of '@0:' will successfully store the file named, but will corrupt other files on the disk. The reason for this is that '@0:' seems, under some circumstances, to fail to register in the Block Allocation Map (BAM) the correct picture of the sectors on the disk which it has used or which it has freed, so that subsequent files are SAVEd in places they should not be.

There several solutions to this problem:

1) Include a VALIDATE command (see Chapter 4) in line 2 of the little SAVE routine earlier in this chapter. This reconstructs the BAM and ensures that there will be no corruption, the only drawback being that it can take longer to VALIDATE than it does to format a disk.

2) Start off by calling the program something like TEST01 and, each time you SAVE it, LIST line 2 and change the number on the end of the program name. This is perfectly simple but it does take up a lot of disk space while a program is being developed.

3) Ignore the bug — it will very seldom, if ever, affect you.

4) Best of all, use RENAME and SCRATCH, two housekeeping commands described in Chapter 4 to create a much more stylish and secure method of SAVEing programs and other types of file.

7. Saving what you have SAVEd

The process of keeping a valuable program safe does not end once you have stored it on a single disk. Disks can be damaged or accidentally corrupted in some way. If a program is worth keeping, then you should *always* have a second copy of it stored somewhere safely away from where you normally keep your disks.

In addition, don't neglect the relative safety and reliability of tape for backup copies of important material. A serious disk drive fault can be extremely frustrating if your only copies of the required program are on disk. If experience is anything to go by, most people starting out with a disk drive will ignore this advice, at least until the first occasion on which they totally lose a program on which they have been working for weeks.

CHAPTER 4
Disk Housekeeping Commands

1) Introduction
2) OPEN
3) CLOSE
4) PRINT#
5) NEW
6) SCRATCH
7) RENAME
8) COPY
9) INITIALIZE
10) VALIDATE

1: Introduction

We have already seen that, on the simplest level, the disk drive can be used for the fast storage and retrieval of programs. When using it in this way, the user has no control over the disk or its contents other than to save or load a program. Effective use of the disk drive, even if only for the storage of programs, involves being able to communicate with the disk drive and using that communication to exercise some control over *how* items are stored on the disk. In this chapter, we consider a series of commands which relate not so much to the way the disk drive accepts information from the 64 for storage, but to the manner in which disk files of all types are handled once they are stored on a disk.

Communicating with the disk drive — a simple analogy

Probably the best way to understand some of the problems involved in communication between the 64 and the disk drive is to imagine a situation in which you set out to communicate with someone else using a bank of 15 black telephones and an unlimited supply of note-pads. Your job is to store information for the person at the other end of the phones and to send it back when they want it. The information that is going to be stored is quite complex, for example the prices of shares and commodities on the stock market. The information comes in many different varieties and, of course,

some of it has to go from you to the other person and some of has to come *to* you. On top of all that, the different types of information that will be zipping backwards and forwards need to be stored in different places according to their type. How are you going to handle all of this?

Well, one way would be to assign each different type of information to a different phone and to keep a separate pad by each phone. If phone number 4 rings you will then know that, when the voice at the other end of the line says '567', that this is the current price of gold and you note it down on the pad next to the phone, which is labelled 'GOLD'. If the message had come through on phone 5, it would have meant the price of a particular security and it would have been noted on a different pad. Equally, if phone 8 were to ring you would know that what was expected was for you to simply say the last price of silver — the person at the other end would know exactly what was meant because phone number 8 was being used.

In later chapters we shall spend some considerable time examining exactly how the disk drive can be persuaded to store information and then to surrender it again on request in a way that corresponds to our imaginary phone system. For the moment, we need to note two requirements of the phone system which we have not yet discussed but which are vital to its proper use.

Firstly, at some time before any information is exchanged, there needs to be an understanding as to which line will be used for what. In other words, how do you know that the pad labelled 'GOLD' is to be kept next to phone number 4. Clearly, the person at the other end of the line has to *tell* you that that is what number 4 will be used for. He can't tell you by using any of the 15 normal phones because you are being deliberately simple-minded and refuse to answer any of the phones until you know what kind of information they are to be used for. To overcome this, you are provided with one extra phone, this time a red one. Whenever the red phone rings you answer it immediately and you expect to hear something like:

'Take the pad labelled GOLD away from phone number 4 and store it somewhere safe. Replace it with the pad labelled XYZ Corporation shares.'

or

'Phone number 4 will not be used until further notice, store the GOLD pad away.'

or

'Place the SILVER pad next to phone number 5, which is not currently in use.'

Whenever you get such an instruction down the red phone, you obey it instantly, and the result is that you always know exactly which phone is being used for what.

The second extra requirement we have not yet discussed relates to the storage of the note-pads themselves. We said before, in setting up the example, that you had an unlimited supply of pads. While that is true, you have a problem in that you only have room safely to store a limited number, say 144. The person at the other end of the line places great value on the information contained within the pads and knows that you have limited storage space. From time to time, therefore, he gives you instructions as to how to handle your stock of pads. To accomplish this, he first calls you on the red phone and says: 'The next message on phone number 15 will be an instruction to do something with one or more pads.' If he knows what he is doing, then shortly afterwards phone number 15 will ring and you will hear a message such as:

'We don't need the silver prices any more, you can erase everything contained in the SILVER pad and remove the label from it.'
or
'I want to keep yesterday's gold prices separate from today's, so relabel the GOLD pad as GOLD1 and start a fresh pad called GOLD2.'
or
'I'm getting a little worried about the amount of valuable information stored in these pads, please make copies of all of them.'

Now all of this may seem trivial in the extreme, but if you take the trouble to understand it you will be well on the way to understanding the way in which the 64 and the 1541 work together. In the sections which follow, we shall take a first look at the use of the OPEN command, which corresponds to the use of the red phone in the example. We also examine a series of other commands which carry out 'housekeeping' on disk files, the kind of instructions which are equivalent, in the phone example, to the instructions received on phone number 15.

Section 2: OPEN

FUNCTION: OPEN allocates a unique file number to a unique channel, allowing information to be sent to that file and received by the disk drive (or vice versa). In addition, OPEN is used to specify filenames and the use to which a file is to be put.

When a program is SAVEd or LOADed, the Commodore 64's BASIC interpreter performs a series of operations which are hidden from the user. The most important of these actions, apart from actually putting the correct data on to the disk or getting it back, is to open a channel of communication with the disk drive, the equivalent of the use of the red phone in the example above. In the case of SAVEing, for example, the disk drive must be made aware that data is coming and told what to do with that data. In the case of LOAD and SAVE, this process of opening a channel is performed automatically, without the user having to do anything other than enter the correct keyword.

If the 1541 were *only* capable of storing and retrieving programs, then all that would need to be done would be to switch it between SAVE and LOAD mode. In fact, and we shall go into this in much more detail later on, the 1541 is capable of accepting a wide variety of different types of information for storage as well as programs. In addition, it is capable of listening to instructions or commands, and of recognising that these are *not* items to be stored in some way. It is capable of accepting whole series of items of data to be stored, and of storing each of those items in a different place according the instructions of the programmer. It is capable of accepting a piece of data for storage, then answering a request to retrieve another piece of data from somewhere else, and then going back to the place where the first item was stored to add some more. In other words, the 1541 can accomplish all the tasks achieved by the 15 phones described above.

Like the phone system, the 64 and the 1541 are capable of communicating along several different 'lines' (the precise number will vary according the types of file being used). The areas on the disk drive in which information can be stored are known as files, and they correspond to the note-pads in our example. When sending or retrieving information, there may be different lines used for communication, known as 'file numbers', each of which must correspond with a different file. They are not physically separate wires between the two devices, simply numbers which allow the 64 to specify which file on the disk information is to be taken from or placed into.

To send information to the 1541, or to retrieve it, all that needs first to be done is to specify, using the OPEN command, the following items of information:

1) The number to be assigned to the file (the number of the phone).

2) The fact that this is meant to be used for communication between the 64 and the 1541 rather than between the 64 and some other device such as the cassette recorder or printer.

3) Whether the file is to be used to store information or as a source of information, or whether what is going to be communicated is housekeeping information.

In some cases, though not for the purposes of this chapter, an extra piece of information will be necessary, namely:

4) The name of the file (note-pad) on the disk to be assigned to the specified file number.

For the moment, we are interested only in the uses of OPEN in allowing us to use the housekeeping commands illustrated in the phone example. There are many other uses for the OPEN command and they will be examined in later chapters, but for the purposes of this chapter the format of the OPEN command will be:

OPEN< **FILE NUMBER**> ,< **DEV**> ,**15** eg OPEN 15,8,15

1) **FILE NUMBER** — used to indicate a number in the range 1–127. Normally, for these housekeeping tasks, programmers stick to one file number that they will always recognise as being for that purpose. The reason that this can be a good idea is that the housekeeping instructions are very powerful and can result in loss of information on the disk if mistakes are made in the housekeeping mode. Since, as you will see below, the final figure of the OPEN command when going into housekeeping mode is always 15, many programmers prefer always to use a file number of 15 for housekeeping and not to use that file number for anything else.

2) **DEV** — a number which indicates to the 64 which device it is going to be communicating with. It will normally be eight when talking to the disk drive but can have other values, as indicated in Chapter 13.

3) **15**, the final number — called a 'secondary address', or 'channel'. In the case of all the different devices, a number in this position is used to indicate to the specified device that any data which follows is to be treated in a special way. Thinking back to the phone example, you will remember that before any housekeeping information was communicated, you were told that a particular phone was going to be used for that purpose. In the format of the OPEN command, a secondary address of 15 contains the message 'whatever is sent to this particular file, until further notice, should be treated as housekeeping information'.

You will see in later chapters that what we are calling 'housekeeping mode' in this chapter is in fact much more than that. Channel 15 is known

technically as the 'error channel' and it puts us directly in touch with the complex program which is built into and runs the 1541 and allows many otherwise impossible tasks to be accomplished. You will also see that there is much more to the OPEN command but, for the moment:

OPEN 15,8,15

will suffice to get us into housekeeping mode and explain the commands that go with it.

Summary of OPEN

1) Information cannot be sent to, or retrieved from the 1541 unless a file is first OPENed.

2) In OPENing a file, the file number, the device number, the channel number must always be specified. For some purposes, the filename, file type and whether the file is for reading or writing must be specified. The last three specifiers are covered in other chapters.

3) The file number will be in the range 1–127.

4) The device number will normally be 8 if you have one drive only but may vary if other drives are being used.

5) The channel number will be in the range 2–14 for most uses and 15 when the file is being opened to the error channel.

6) The format for OPEN is:

   OPEN< FILE NUMBER> ,< DEVICE> ,< CHANNEL NUMBER>

3. CLOSE

FUNCTION: CLOSE acts to put a terminator on the end of a disk file and to write its start address and length to the disk directory.

In the phone example, you may remember that one type of instruction which might be given over the red phone was to stop using a particular note-pad, or indeed a particular phone, until further notice. This instruction has its parallel in the CLOSE command, which tells the disk drive that there is no further use for a file at the present time, so it can be safely and tidily stored away somewhere on the disk. In subsequent chapters, as with

OPEN, we shall see that CLOSE is used in a wide variety of circumstances. For the moment, however, we are interested in the 'housekeeping mode' and will limit ourselves to what we need to know to deal with that.

Example procedure to illustrate the need for CLOSE

1) Switch on the 64 and the disk drive and place a disk in the drive.

2) Type:

OPEN 15,8,15[RETURN]

3) Type:

OPEN 15,8,15[RETURN]

4) You should see the error message:

?FILE OPEN ERROR

5) Type:

CLOSE15[RETURN]

6) Type:

OPEN 15,8,15[RETURN]

7) This time, no error message is generated.

8) If you're tidy-minded, you'll now type CLOSE15[RETURN] for a second time.

The meaning of all this is quite simple. Once you have told the system that a particular file number is assigned to a particular purpose, it remains assigned to that purpose until you free it. The first OPEN command assigned file number 15 to the error channel on the disk drive and it can only be available for another assignment when the first has been ended. This is the function of the CLOSE command. It is good practice always to close a file that you are not using for a time but especially important in the case of the error channel because of the power of the commands associated with it. Once the file is closed, there is no danger that you will inadvertently

send an instruction to it. Remember then, when you have used one of the housekeeping commands described in this chapter, always close the error channel before going on.

There is much more to be said about the way to OPEN and CLOSE the error channel, but most of it would add little to your understanding of the housekeeping commands in this chapter. One note of caution needs to be sounded, however, if you intend to use the housekeeping commands in your programs before going on to read the separate chapter on the error channel. Closing the error channel results in all the files to the disk drive being closed at the disk drive end of the system without the 64 being aware of the fact. If you are using other files to store information, either close them first before opening the error channel or open the error channel before all the other files and close it after them.

Summary of close

1) File numbers which have been allocated to files by the use OPEN remain unavailable for other use until those files are CLOSEd.

2) Files which are not properly CLOSEd when communication between the 64 and 1541 is ended will not be properly recorded on the disk and their contents will be lost.

3) CLOSEing the error channel results in all current files being closed at the disk end of the system though the 64 will still regard the files as open.

4) The format for CLOSE is:

   CLOSE< FILE NUMBER>

4. PRINT

FUNCTION: PRINT# sends information in the form of string or numeric variables to a specified file.

So far in this chapter, all that we have learned to do is to OPEN and CLOSE a particular file, not how to actually *do* anything with it. In this section we take a first look at how information can be sent to the disk drive for storage in a file. Once again, for the purposes of the current chapter, we are interested only in how, once the error channel is open, we can send housekeeping instructions to the 1541. Later chapters will deal more extensively with the use of PRINT #.

Example procedure to illustrate the use of PRINT#

1) Enter the following short program and RUN it:

```
10 PRINT "[CLS]"
20 OPEN 1,3
30 PRINT#1,"THIS IS OUTPUT TO A FILE"
40 CLOSE1
```

The result should be that the words specified in line 30 are printed on the screen — so what has happened? PRINT # is a way of outputting information to a file, and in this case a file with a number of 1 was opened to device 3, which is the screen itself. The file having been opened, PRINT # accomplished the task of sending the specified characters to the specified device.

In the sections which follow, PRINT # will be employed to send down the open error channel a message telling the DOS which housekeeping operation is to be carried out.

One note of caution on the use of PRINT # (and the parallel commands INPUT # and GET # which you will find in later chapters). Many 64 owners have become skilled at entering BASIC keywords in shortened form (eg L followed by SHIFT/I for list) and seldom use any other method. It needs to be remembered that none of these commands can be entered by using the abbreviation for the normal keyword and then adding '#' on the end. Entering '? #' into a program line will result in a listing which will show 'PRINT #' at the correct position in the line but will produce a syntax error every time the line is executed. The correct abbreviations for the three commands which contain '#' is:

PRINT # — P SHIFT/R

INPUT # — I SHIFT/N

GET # — *no abbreviation*

Summary of PRINT#

1) PRINT # is used to send information to a file.

2) The correct abbreviation for PRINT # IS *not* '? #' but 'P SHIFT/R'.

3) In the context of sending housekeeping commands to the 1541, the format of the PRINT # statement is:

   PRINT #< FILE NUMBER> ,< COMMAND STRING>

4) The maximum length of the command string which may be sent along the error channel appears to be 41 characters for the 1541 and not the 58 characters specified in the manual.

5) PRINT # has a wide variety of other uses which are discussed in other chapters.

Avoiding the use of PRINT # in housekeeping commands

Those who have some experience of the use of the disk drive will already know that there is a shorter method of sending commands than the use of PRINT #, and that is to include the 'command string' (eg the words in quotes in line 30 of the example program earlier in this chapter), as part of the OPEN command eg:

OPEN 15,8,15,

followed by

PRINT # 15,"COMMAND"

would become

OPEN 15,8,15,"COMMAND".

This can only be done in the case of messages addressed to the error channel and examples of its use are given in the summary for each command. In general we have chosen to illustrate the use of the commands by the use of PRINT #, however, since this uncomplicated format distracts less from the main focus of attention, which should be the command itself.

5. NEW

FUNCTION: To set up a floppy disk ready for the reception of data, a process which is known as formatting. The disk is divided into tracks and sectors, and a directory of contents and Block Allocation Map created. A less drastic form of NEW changes the title of the disk without erasing its current information.

In the description of the nature of a floppy disk in Chapter 1, we have already seen that a diskette needs to be prepared before it can be used for the storage of programs or data. The 1541 is equipped to carry out this process in one of two ways:

1) Erasing whatever is on the the disk and reformatting it from scratch, *or*

2) If the disk has been previously formatted, it is possible to clear the directory of the disk without actually reformatting. Files subsequently stored on the disk will simply overwrite the programs on it, since these have become invisible to the 1541 with the clearing the directory. The advantage of this second method is that it takes far less time than a full reformatting of a disk as if there were nothing already on it.

It should be stressed that in either of the above cases, any data contained on the disk will be lost. In the case of method one, the situation is irretrievable, since the data has been physically erased by the disk drive's magnetic heads. If method two has been employed, it may be possible to read the lost data directly from the disk by bypassing the directory, though this is unlikely to be easy. The moral of this is that you should always be extremely careful when formatting a disk. If in any doubt, first examine the directory of the disk to make absolutely sure that it does not contain material you wish to keep.

Having said something about the dangers, the best way to understand the procedure for formatting a disk is to do it.

Explanatory procedure for the use of NEW

1) Turn on the 64 and the disk drive.

2) Choose a new, unformatted disk or one which contains nothing you want to keep and place it in the disk drive.

3) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"NEW0:TEST,01" [RETURN]
```

4) Wait until the red light on the 1541 goes out, during which time the disk drive will click and whirr.

5) Type:

```
CLOSE15 [RETURN]
LOAD "$",8 [RETURN]
LIST [RETURN]
```

6) You should now see the following displayed on the screen (though the heading will be in inverse characters):

```
0 "TEST                " 01 2A
664 BLOCKS FREE.
```

7) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"NEW0:TEST2" [RETURN]
```

8) Wait until the red light goes out on the disk.

9) Type:

```
LOAD "$",8 [RETURN]
LIST
```

10) You should now see the following displayed on the screen:

```
0 "TEST2               " 01 2A
664 BLOCKS FREE.
```

What you have done is to format the disk twice, once completely from scratch and a second time by simply altering the name and clearing the directory (not that there was anything in the directory anyway). You will have noticed that the second form was considerably faster than the first.

Format of the 'NEW' command

Examining the steps given above, it can quickly be seen that the two commands which have accomplished the formatting and reformatting of the disk are:

1) PRINT # 15,"NEW0:TEST1,01"

and

2) PRINT # 15,"NEW0:TEST2"

and the explanation of them is as follows:

a) **NEW:** the command itself. All the housekeeping commands begin with the command keyword itself, or an abbreviation of it. In the case of NEW, the abbreviation is 'N', so that the command could have been expressed as:

PRINT#15,"N0:TEST1,01"

b) **'0:'** : as in the chapter on SAVEing and LOADing, this specifies the disk drive on which the operation is to be carried out and will normally be zero.

c) **TEST1** (or TEST2): the name to be given to the disk. The sole use of the name is to identify the disk when the directory is printed out.

d) **'01'**: the identification number or ID of the disk. This can be any two alphanumeric characters, ie anything from 00 to ZZ. Two purposes are served by the ID. Firstly, it allows a series of disks to be given the same title but distinguished by means of their ID. Secondly, the ID is written into every block on the disk when it is formatted and is used by the disk drive when it picks up a block as a check to see whether the disk has been changed since the last time the drive was accessed. This not a very effective check, since most people tend to number all their disks '01' anyway. If a disk is replaced with another of the same ID, it is possible that the disk drive will continue to work on the basis of the directory of the previous disk and not realise that a change has been made. This may not be so important when LOADing files but can be disastrous when saving programs or data since existing information can be overwritten. The way around the problem is to initialise the drive every time you change disks (see INITIALIZE command later in this chapter).

You will note that in the second example of the NEW command, no ID was specified. In this case, the disk is left as it was originally formatted and only the title and directory changed. The avoidance of the need to place the timing markers and block markers all over the diskette, together with the ID on every block, accounts for the difference in time between the two methods.

Summary of NEW

1) NEW is essential in all cases where a freshly bought disk is to be used for the first time.

2) In the case of disks which have been previously formatted, the form of NEW which does not specify a new ID saves time.

3) NEW effectively destroys all the existing data on a diskette and so needs to be used with caution.

4) The format for NEW can be any of the following:

   PRINT # 15,"NEW0:< FILENAME> ,< ID> " — total reformat of disk

   PRINT # 15,"N0:< FILENAME> ,< ID> " — as above but abbreviated keyword

   PRINT # 15,"NEW0:< FILENAME> " — disk renamed and directory cleared

   PRINT # 15,"N0:< FILENAME> " — as above but abbreviated keyword

5) Using the condensed form of OPEN, a typical format might be:

   OPEN 15,8,15,"N0:TEST1,01"

6. SCRATCH

FUNCTION: SCRATCH removes a disk from the file by marking it in the directory as a scratched file. SCRATCHed files remain a part of the contents of the disk but the space they take up is made available to subsequent files and will be overwritten if further files are opened and written to.

Effective use of the disk drive involves regular decisions as to what files you will keep on your disks and what you will discard. Disks are not expensive but their cost can quickly mount up if they are crammed with early versions of programs and with data which is never going to be used again. In addition, disks which contain large quantities of material which is never likely to be used again make the task of finding the right program much more difficult. What is usually needed is a development disk for each of the projects you are working on at the moment, each containing a lot of half-finished material which you will probably not want to keep in the long term. Once the development stage is over, you should be aiming to keep disks which contain only useful material and, preferably, disks devoted to a single application so that you know exactly which disk to reach for to accomplish a particular task.

To achieve the goal of a library of disks containing what you *want* rather than what happens to be there, you need not only to be able to store files on a disk, you need to be able to delete them as well, and this is what the command SCRATCH allows you to do.

Example procedure to illustrate the use of SCRATCH

1) Ensure that the 64 and disk drive are both on and that the disk TEST2, which you formatted in the previous section, is properly inserted in the drive.

2) Enter the following one line program:

```
10 REM THIS IS A TEST PROGRAM
```

3) Type:

```
SAVE "TEST1",8 [RETURN]
SAVE "TEST2",8 [RETURN]
```

4) When SAVEing is finished, type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

5) You should now see displayed:

```
0 "TEST2            " 01 2A
1    "TEST1"            PRG
1    "TEST2"            PRG
662 BLOCKS FREE.
```

6) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"SCRATCH0:TEST1" [RETURN]
CLOSE15 [RETURN]
```

7) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

8) You should now see displayed:

```
0 "TEST2            " 01 2A
1    "TEST2"            PRG
663 BLOCKS FREE.
```

What has happened is that, of the two files you placed on the disk, TEST1 and TEST2, one of them, TEST1, has been deleted and the space it took up on the disk freed for other use. After the warnings about the use of NEW, it goes without saying that SCRATCH too needs to be treated with caution, since it is all too easy to mistake the names of files and to end up deleting the wrong one. If in doubt, examine the contents of a file before deleting it.

In the following chapter, on 'pattern matching', you will find that SCRATCH can be made to work on more than one file at a time on the basis of similarities in the filenames. There is no difference in principle in this use and it will be left until a fuller explanation of pattern matching can be given.

Summary of SCRATCH

1) The housekeeping command SCRATCH removes a named file from the disk in the drive.

2) SCRATCH is an essential command for the proper management of your disk library, to avoid valuable space being taken up by unwanted material.

3) SCRATCH needs to be used with caution, since SCRATCHed files are not normally recoverable.

4) The format for SCRATCH is as follows:

   PRINT # 15,"SCRATCH0:< FILENAME> "

   or

   PRINT # 15,"S0:< FILENAME> "

5) Using the condensed form of OPEN, the format is:

   OPEN 15,8,15,"S0:< FILENAME> "

6) Provided that they have not been overwritten, SCRATCHed files may be recovered by the use of the program UNSCRATCH given in Chapter 10.

7. RENAME

FUNCTION: RENAME changes the name of a file on disk, provided that the new name to be allocated to the file is not already present in the directory. The contents of the file are not altered in any way.

The purpose of the RENAME command is to allow you to change the name of a file which is on disk without in any way altering the rest of the contents. Few people make much use of RENAME since they see very little reason to make regular changes in the names of files. This is a pity, for RENAME in combination with SCRATCH is a powerful tool in the effective management of disk files.

The format for RENAME is:

RENAME0:< NEWFILENAME> = 0:< OLDFILENAME>

and its use will be illustrated below.

Example procedure to illustrate the use of RENAME

1) Ensure that the 64 and disk drive are both on and that the disk drive contains the disk TEST2 which you formatted earlier. The contents of TEST2 should be the single program TEST2 which was SAVEd during the testing of the SCRATCH command.

2) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"RENAME0:TEST1=0:TEST2" [RETURN]
CLOSE15 [RETURN]
```

3) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

4) You should see:

```
0 "TEST2            " 01 2A
1    "TEST1"            PRG
663 BLOCKS FREE.
```

5) Type:

```
LOAD "TEST1",8 [RETURN]
LIST [RETURN]
```

6) You should see:

```
10 REM THIS IS A TEST PROGRAM
```

What has happened is that a disk containing a program called TEST2 now contains one called TEST1, though the contents of the file have not been altered. This is interesting though it hardly seems very significant, but RENAME has much more important uses.

Example procedure illustrating the creation of backup files with SCRATCH and RENAME

1) Enter this short program:

```
1 GOTO 10
2 SAVE "TEMPORARY",8 : VERIFY "TEMPORARY",8
3 OPEN 15,8,15
4 PRINT#15,"SCRATCH0:TEST1.BAK"
5 PRINT#15,"RENAME0:TEST1.BAK=0:TEST1"
6 PRINT#15,"RENAME0:TEST1=0:TEMPORARY"
7 CLOSE 15
10 REM THIS IS VERSION 1
```

2) Start the program with RUN 2 and wait until the word READY is printed and the flashing cursor returns.

3) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

4) You should see from the directory that there are two files, TEST1 and TEST1. BAK

5) Type:

```
LOAD "TEST1",8 [RETURN]
```

and change line 10 to read:

```
10 REM THIS IS VERSION 2
```

6) Start the program with RUN 2 and wait until the flashing cursor returns.

7) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

8) You should see from the directory that there are still two files called TEST1 and TEST1. BAK, but that TEST1. BAK is now first.

9) Type:

```
LOAD "TEST1.BAK",8 [RETURN]
LIST [RETURN]
```

and you will find that what is loaded is version 1 of the program.

10) Type:

```
LOAD "TEST1",8 [RETURN]
LIST [RETURN]
```

and you will find that what is loaded is version 2 of the program.

What has happened here, if you follow the little program through, is that the program to be saved is first recorded on the disk under the name TEMPORARY. If you are going to use this particular technique then it is important that the filename TEMPORARY should only ever be used as the initial name under which a current program is stored. It can, of course, be used for the same purpose with all the programs you want to save, since it will only exist on the disk for a few moments anyway.

Having saved TEMPORARY, the program then scans the disk for a program called FILENAME. BAK (in our case TEST1. BAK) and deletes it. If the program is not present, as it will not be the first and second time something is saved by this method, no problems are caused, the instruction is simply ignored.

The third step is to rename the program FILENAME (in our case TEST1) as FILENAME. BAK, where '. BAK' is short for 'backup'. Thus, as opposed to the method of saving updated versions of a program described in Chapter 3, the existing version of the program being worked on is not erased, it is simply renamed.

Finally, the file TEMPORARY is renamed as FILENAME (in our case TEST1) and the process is complete. The latest version of the program is

now called FILENAME and the latest of all the previous versions is called FILENAME. BAK. The only thing to watch out for here is that adding the '. BAK' marker to the end of the filename means that the maximum length of the filename to begin with is reduced from 16 characters to 12.

The advantage of this is that if you do, by some mistake, corrupt a program and then save it, you have one previous copy. A second advantage is that the bug associated with the '@0:' facility is avoided by the method — nothing is overprinted in this method. In addition, the disk drive, like any other piece of equipment, can go wrong. It can sometimes corrupt a file as it is being saved, or perhaps a surge in the electricity supply may lead to the SAVEing process being interrupted and, what is worse, the 64 itself losing the program. In that case, you are usually left with a corrupted program on the disk and no program in the 64 to reSAVE — in other words a disaster. Using the procedure given above, the situation could be easily remedied since at no time is there only one version of the program on the disk to be corrupted — there should always be one version left intact.

Summary of RENAME

1) The function of RENAME is to give a new filename to a file on disk without otherwise altering the contents of the disk.

2) RENAME, in conjunction with SCRATCH, provides powerful facilities for creating backup files of material which would otherwise be overwritten.

3) RENAME has some of the limitations of SAVE, ie it will not create a file if one of the same name already exists.

4) The format for RENAME is:

   PRINT # 15,"RENAME0:< NEWFILENAME> = 0:< OLDFILE NAME> "

   or

   PRINT # 15,"R0:< NEWFILENAME> = 0:< OLDFILENAME> "

5) The format for RENAME using the condensed form of OPEN is:

   OPEN 15,8,15,"R0:< NEWFILENAME> = 0:< OLDFILE NAME> "

8. COPY

FUNCTION: COPY duplicates an existing file under a different name — it differs from RENAME in that the original file is still present. COPY can also be used to add one file on to the end of another.

The purpose of COPY is to duplicate any file on the disk currently in use with another file with a different name but the same contents. Note that this differs from RENAME in that the original version of the file is still present on the disk. The main use of COPY on the 1541 is to create backup copies of a file on the same disk. COPY is, frankly, far less useful on a single drive like the 1541 than it would be on a dual drive such as the 4040 or 8050. With a dual drive, files can be copied *between* the two drives so that backups can be made on a separate disk.

Concatenation of files

One extra facility which COPY provides is the ability to 'concatenate' files on disk. The effect of this is to take two to four files and to combine them 'nose to tail' into one new file — once again the originals are left unchanged. You may notice that in the disk manual supplied with the 1541 this facility is mentioned but no use is suggested for it. At least part of the reason for this is that that there *are* hardly any uses for concatenation. In the later chapter on sequential files we show how data files can be run together usefully. Program files which are concatenated will load but only the first program will be available to the user, the rest will simply clutter up the memory. The reason for this is that each program file begins with two bytes which tell the 64 where the program is to be loaded into memory and ends with two zeros which signify the end of the file. In their proper places these markers are essential, but locked up in the middle of the new concatenated file they make it impossible for the 64 to see the whole of the concatenated program which has been loaded into memory.

Example procedure to illustrate the uses of COPY

1) Ensure that the disk drive and 64 are on and the TEST2 disk is properly inserted.

2) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"C0:TEST2=0:TEST1" [RETURN]
CLOSE15 [RETURN]
```

3) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

4) You should see from the directory that the programs TEST1 and TEST1. BAK have been joined by TEST2.

5) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"C0:TEST3=0:TEST1,0:TEST2" [RETURN]
CLOSE15 [RETURN]
```

6) Type:

```
LOAD "$",8 [RETURN]
LIST [RETURN]
```

7) You should see from the directory that another program has been added, namely TEST3, and that this new program is two blocks long.

What has happened here is that file TEST1 has been duplicated under the name TEST2, giving two files on the disk. These two files have then been concatenated into another called TEST3. You can, if you wish, LOAD the program TEST3 into memory. You will find that, even though the program appears to be twice as large on disk, it appears to be the same as the previous TEST1 program we examined under the RENAME section. In fact, a duplicate of the program is lost somewhere in the memory of the 64, accounting for the increased size on the disk.

Summary of COPY

1) The function of COPY is to duplicate a file under a different name but with the same contents.

2) COPY can also be used to concatenate two to four files together though this is of limited use (see Chapter 7).

3) The format of COPY when use to duplicate a file is:

   PRINT # 15,"COPY0:< SECONDFILENAME> = 0:< FIRSTFILE NAME> "

or

PRINT # 15,"C0:< SECONDFILENAME> = 0:< FIRSTFILE NAME> "

4) The format for COPY when used to concatenate files is:

PRINT # 15,"COPY0:< NAME4> = 0:< NAME1> ,0:< NAME2> ,0:< NAME3> "

or

PRINT # 15,"C0:< NAME4> = 0:< NAME1> ,0:< NAME2> ,0:< NAME3> "

and it is important to remember that the command string sent along the error channel may not exceed 41 characters.

5) Using the condensed form of OPEN, the two formats are:

OPEN15,8,15,"C0:< SECONDFILENAME> = 0:< FIRSTFILE NAME> "

or

OPEN15,8,15,"C0:< NAME4> = 0:< NAME1> ,0:< NAME2> ,0: < NAME3> "

9. Initialize

FUNCTION: INITIALIZE forces the disk drive to re-read the Block Allocation Map from the disk into its memory and thus ensures that the drive is working on the most up-to-date version of the BAM available. This overcomes problems which may occur during the use of VALIDATE or when a disk is changed for another which may have the same identification number (disk ID).

In the general introduction to the working of the disk drive, we noted that the DOS uses a map of the disk called the Block Allocation Map, BAM, when reading or writing data. At times, as we saw in the chapter on SAVEing, it is possible for the disk drive to become confused as to the correct state of the BAM and to write data on to disk sectors which are already occupied, corrupting existing files.

This confusion arises because the physical copy of the BAM which is kept on the disk is only updated when a file is CLOSEd. When the disk

drive is switched on and a disk first accessed, the disk drive takes the BAM into its own memory. Changes which occur, such as the writing of data to a file, are registered in the version of the BAM which exists in the disk drive memory but *not* in the physical copy of the BAM (until a file is closed, as mentioned). In a small number of cases it is possible that a disk error of some kind may lead to the BAM in memory becoming corrupted so that the drive will be working with an incorrect picture of the disk. This can happen particularly with the VALIDATE command but will also happen if the disk is changed for another with the same ID (see NEW). In this case the disk drive will continue to work on the basis of the BAM from the first disk until a file is CLOSEd, by which time it will probably be too late.

The function of INITIALIZE is to force the disk drive to re-read the BAM from the disk, thus overcoming the problem of the corrupted BAM in memory. INITIALIZE should be used whenever a problem is encountered during the use of VALIDATE (see next section) or whenever disks are changed and you are not absolutely sure that the disk IDs are different. Use of INITIALIZE will produce no visible changes in the disk, the only demonstration that the command is functioning correctly is that you do not end up with corrupted disks so often.

Summary of INITIALIZE

1) The function of INITIALIZE is to ensure that the Block Allocation Map on which the disk drive is working is an accurate reflection of the current state of the disk.

2) INITIALIZE should be used whenever problems are encountered in the execution of a VALIDATE command or when a disk is replaced with another that may have the same ID number.

3) The format for INITIALIZE is:

   PRINT # 15,"INITIALIZE"

   or

   PRINT # 15,"I"

5) Using the condensed form of OPEN the format is:

   OPEN 15,8,15,"I"

10. VALIDATE

FUNCTION: VALIDATE reconstructs the Block Allocation Map on the basis of what is actually stored on the disk. Sectors incorrectly allocated or

tied up in files which have been deleted or which have not been properly CLOSEd, are freed for use.

The function of VALIDATE is to tidy up a disk and free space for storage of data. The reason that this is necessary is that various housekeeping commands, such as SCRATCH and the '@0:' facility for overwriting existing files, do not always delete all the blocks on the disk which are associated with the file that is being SCRATCHed or overwritten. Since files are constantly being SCRATCHed or overwritten on a working disk, the situation may eventually arise where a large proportion of the blocks on the disk are allocated according to the BAM but are performing no useful function.

What VALIDATE does is to trace through each file on the disk, block by block from start to finish. When it comes across blocks which do not fall into the correct structure of the file or which are allocated to files which no longer exist on the disk, it de-allocates those blocks and records in the BAM that they are free. The result of the process is a BAM which has been totally reconstructed on the basis of what is actually on the disk. It is not possible, with a fresh disk such as our TEST01 disk, to demonstrate the use of VALIDATE, but if you would like to apply it to a disk which you have used extensively, especially for large programs, the number of blocks freed for use can sometimes be quite dramatic.

One powerful use of VALIDATE is to overcome the limitations of the '@0:' facility which were mentioned in Chapter 3. The problem with '@0:' is that it does not always appear to register accurately in the BAM the sectors which it has freed in scratching a previous version, or the blocks used in storing the new version of a file. This does not prevent the proper storage of the current file but means that future files may be stored according to an incorrect BAM, with all the drastic consequences that that implies. This problem can be overcome if VALIDATE is used immediately following the SAVEing of a file using '@0:' but, frankly, the time taken by VALIDATE means that a far more practical method of overwriting a file is to use the 'SCRATCH and RENAME' technique described under the section on RENAME.

Summary of VALIDATE

1) The function of VALIDATE is to free blocks on the disk which are improperly allocated.

2) The format of VALIDATE is:

```
PRINT#15,"VALIDATE"
```

or

PRINT # 15,"V"

3) Using the condensed form of OPEN, the format is:

OPEN 15,8,15,"V"

4) If you are making use of random files (see Chapter 10), VALIDATE must be used with caution since it will probably delete them.

CHAPTER 5
Pattern Matching

1) What is pattern matching
2) Patterns with ''*
3) Patterns with '?'
4) Combining '' and '?'*
5) Using pattern matching with commands

1. What is pattern matching

The SCRATCH command described in the last chapter, together with LOAD and VERIFY, can be increased in flexibility by the use of a simple technique known as pattern matching.

In essence, pattern matching is like setting up a kind of stencil which will be applied to the names of files on the disk. The stencil will have some letters cut out of it so that what is on the paper below can be seen, while in other places it will hide the paper. If the stencil reads 'PROG– – – – – – –', (where the dashes represent character positions that cannot be seen through) and the stencil is then moved over a block of text with letters of the same size as those cut out, if there are any words with the letters 'PROG' in them, those four letters will be plainly seen through the stencil, others will be meaningless jumbles. Pattern matching applies the same technique to the names of files in the directory of the disk drive, allowing the user to specify a pattern or stencil and to carry out an operation on all the files that fit it.

2: Patterns with '*'

The simplest form of pattern to understand is a pattern involving the use of the asterisk, '*'. Whenever an asterisk is found at the end of a file name, the DOS understands that the user does not care what character or characters follow the position occupied by the asterisk, provided that there *is* at least one more character. Given below is an example of a pattern using '*' and the filenames that would agree with it:

PATTERN	ACCEPTABLE NAMES	UNACCEPTABLE NAMES
SMITH*	SMITHSON, SMITHERS,	SMIT,SMITH, SMYTHE

3. Patterns with '?'

The other character that may be employed in creating a pattern is the question mark, '?'. When a question mark is encountered, the DOS interprets it as meaning, 'there must be a single character in this position but it does not matter what it is'. Multiple question marks can be used to map an area of specific size within a name where the actual characters do not matter. As opposed to '*', the question mark can be embedded in a pattern or can start one for that matter.

PATTERN	ACCEPTABLE	UNACCEPTABLE
SM??H	SMITH,SMYTH, SMASH	SMITHSON,SMIT, SMART

4. Combining '*' and '?'

The two pattern specifiers can be combined to produce a wide variety of effects, but they do need to be used with caution since it is sometimes difficult to see immediately what the result of a particular combination will be — with unfortunate results if the pattern applies a SCRATCH command to the wrong file:

PATTERN	ACCEPTABLE	UNACCEPTABLE
SM??*	SMITH,SMART, SMITHERS	SMI,SMIT
???T*	SMITH,SMITHERS	SMART,SMIT
??I?*	SMITH,SMITHERS, PRICE	SIMS,SMART,SMIT

5. Using pattern matching with commands

Pattern matching techniques can be used with the SCRATCH, LOAD and VERIFY commands by including the pattern matching symbols within the quotes as if they were a normal part of the file name.

For instance:

"S0:T*"

as a message down the error channel would SCRATCH every file beginning with 'T', while

"S0:*"

would SCRATCH every file.

In the case of LOAD and VERIFY, the pattern will be used to find the

first matching pattern in the disk directory (not the last matching file used as the manual states). This can considerably shorten the process of loading programs if this has to be done frequently. LOAD "*" will load the first file on the directory, a technique used on many commercial disks.

CHAPTER 6
The Error Channel

1) What is the error channel
2) Getting messages from the error channel
3) Using the error channel
4) Summary of error channel

Several times in previous chapters we have made use of what we have referred to as the error channel, the term applied in the 1541 manual to a file opened with a secondary address, or channel, of 15. In fact, what we have been using this particular channel for so far has been the conveying of instructions from the 64 to the DOS. In this chapter, we look more closely at the way in which the 1541 is capable of informing the 64 of problems which may arise during the handling of files.

1. What is the error channel

Problems can be encountered with the disk drive at all levels of use. On the straightforward level of SAVEing and LOADing, a direct command to perform an operation can result in a flashing red light indicating that for some reason the command cannot be carried out. This is frustrating enough but other errors can be far worse. We shall see in Chapters 7 to 10 that for the storage of data, rather than simply program files, control over the disk by the program running in the 64 is essential. While, with proper preparation, this control can be expected to run smoothly for the vast majority of the time, there will inevitably be occasions when faulty disks, a momentary problem with the 1541 or deficient programming will lead to problems in the storage or retrieval of data. Such a situation is seldom disastrous provided that the program controlling the disk drive is aware that something is wrong and capable of either taking some action or at least informing the operator that something needs to be done. If, however, the program continues unaware of the problem with the disk drive, valuable data can be lost or corrupted beyond repair.

Fortunately, the 1541 provides a more-than-adequate means of discovering the nature of any problems with disk handling, in the form of the error

channel. Not only can this direct line to the 1541 DOS be used for conveying instructions, it is capable of generating more than 50 separate error messages covering the kind of problems that occur, and sending them in such a way that either the 64 or the operator can remedy the situation. These error messages are provided in the form of four separate items of information which the 1541 DOS has ready to send down the error channel every time the disk is accessed. In most cases, these items of information are not sent, since the error channel is not open — they are, however, always available and can be used to significantly increase the security and effectiveness of disk-handling procedures.

2. Getting messages from the error channel

For a simple illustration of the use of the error channel, follow the steps given below.

Explanatory procedure for reading the error channel

1) Enter the following short program:

```
10 OPEN 15,8,15
20 INPUT#15,A,B$,C,D
30 PRINT A,B$,C,D
40 CLOSE 15
50 END
```

2) Ensure that the disk drive is empty and then switch it off.

3) Switch the disk drive on again.

4) RUN the program. You should see:

```
73          CBM DOS V2. 6 1541      0          0
```

5) Still without a disk in the drive, enter:

SAVE "T",8[RETURN]

6) RUN the program. You should see:

```
74          DRIVE NOT READY         0          0
```

What has happened here is that you have communicated directly with the internal processor of the 1541 and allowed it to tell you exactly what the problem is that prevents you from carrying out the command specified. Four items of information are supplied — in the order: number, string, number, number — and their meaning is as follows:

1) **The error number:** despite the fact that a verbal message is supplied, the error number should not be ignored. The error messages themselves fall into groups, all having the same verbal message. In these cases the only way to distinguish between the possible causes of the error message is to look up the specific error number for further information. Details of the meaning of the error numbers are provided in an appendix to the 1541 Disk Drive Manual.

2) **The error message:** this two or three word message is a general indication of the problem encountered (see 'the error number', above).

3) **First number:** the track of the disk on which the problem has been encountered. If the command last entered was SCRATCH, and no problem has been encountered in carrying it out, the error number will be one and the number in the 'track' position will indicate the number of files scratched.

4) **Second number:** the sector within the track within which the problem has been encountered. In the two example cases given above, the zeros in positions 3) and 4) indicate that the problem is not one on the disk itself but with the drive. In most cases, the track and sector information gives very little extra help in identifying what has gone wrong. This is not always the case, however. If a READ or WRITE error is indicated on track 18, sector 0, it is likely that you are trying to use an unformatted disk — track 18 is where the disk directory normally resides. When using random access files (see Chapter 10), the track and sector information can be vital in identifying an error, since the very nature of such files is that they access individual sectors under program control.

Note: Though it is possible to open the error channel in direct mode (ie an instruction without a line number), it is not possible to obtain the error message from the error channel in direct mode since INPUT# (like INPUT) will generate an ILLEGAL DIRECT ERROR message. Error channel information can *only* be obtained by instructions contained in program lines.

3. Using the error channel

The five-line test program given above is an example of the way in which the messages generated by the error channel can be displayed by the main system. Many programmers, when developing a program which makes use of the disk drive, tag these lines on to their program as a separate routine at the end. In the event that a disk error develops whose nature is not obvious, the program can be stopped and GOTO < error routine line> entered. This is a useful technique but clearly is not going to be satisfactory when the program is finished and working. What is needed is a method of accessing the error channel while a program is running, so that action can be taken by the program itself or, at the very least, the operator informed of the nature of the problem.

Given below is a short subroutine which can be written into a program to open the error channel and read the messages it provides.

Subroutine to access error channel under program control

```
5000 REM#********************************
5010 REM DISC ERROR STATUS
5020 REM********************************
5030 INPUT#15,EN,EM$,ET,ES
5040 IF EN=73 THEN 5030
5050 IF EN<20 THEN RETURN
5060 PRINT "[CLR][CD][CD][CD][CD] ******
**********************************"
5070 PRINT "[CD][CD]                DISC ER
ROR"
5080 PRINT "[CD][CD]      ERROR -" EN " "
 EM$
5090 PRINT "[CD]      AT TRACK -" ET "
AND SECTOR -"ES
5100 PRINT "[CD][CD]      PROGRAM
EXECUTION TERMINATED"
5110 PRINT "[CD][CD][CD] ***************
************************"
5120 CLOSE 15
5130 SYS 65511
5140 END
```

Explanation of error channel subroutine

The main points of the subroutine should be clear on first examination. The subroutine works on the assumption that the error channel has been opened when the program was first initialised with a command such as:

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

If no errors are encountered during the course of program execution, the program should make provision for CLOSEing the error channel. Once again, it needs to be remembered that the error channel should be closed *after* all other disk files have been finished with, since closing the error channel results in the closure of all files at the disk end of the system.

Provided that the error channel has been opened, the subroutine can be called from a data file module whenever a file is opened or, if extreme security is required, every time an item is written to or read from a file, though this latter technique will slow down the execution of the file operation. In most cases, calling the subroutine will have no visible effect whatsoever, since the message received along the error channel will be '0,OK,0,0', indicating that all is well. When the error channel does notify that a problem has arisen, the subroutine will immediately print out the error message from the 1541. This is followed by CLOSE15, which closes all files at the disk end of the system, and SYS 65511, a call to the routine in the kernal which closes all files from within the 64. Finally, the subroutine terminates program execution.

It should be stressed that the approach adopted in the subroutine is only one of many. It is perfectly possible to set up the subroutine so that for certain error messages it merely displays the error for a time, then returns execution to the file module of the program or to the main menu from which the file module was called. In the case of a program which allows the user to specify a file name for data to be saved, you might wish to avoid the overwriting of existing files. In this case, the data-saving module of the program would allow the filename to be specified, attempt to open the file then call up the subroutine which would receive the message '63,FILE EXISTS,0,0' along the error channel. This would be displayed for the operator as an indication that the data had *not* been saved and execution would return to the main program where the operator would have the option to save the file under another name. In other cases, for example when complex data is being written from one file to another, failure to open one of the files might very well be a reason to stop the program immediately before corrupt data was generated.

4. Summary of the error channel

1) The error channel, ie a file opened to the disk drive with a secondary address of 15, is a convenient way of discovering the nature of any problems which are occurring during disk handling.

2) Error messages are generated by the 1541 DOS in the form of a group of four items in the order: NUMBER1, STRING, NUMBER2, NUMBER3.

NUMBER1 refers to the error number, allowing the precise nature of the error to be ascertained from the disk drive manual. STRING provides a shorthand description of the type of error in words. NUMBER2 and NUMBER3 refer to the track and sector in which the error was encountered or to the number of files scratched if the last command to the disk was SCRATCH and the error number is one eg '1,FILES SCRATCHED,2,0', showing that two files have been scratched.

3) Error messages can *only* be obtained from the error channel by a program line, they cannot be accessed in direct mode.

4) CLOSEing the file opened to the error channel closes all disk files at the disk end of the system. In general, if the error channel is to be accessed during the course of a program, it is better to open the file at the beginning of the program and to close it at the end.

CHAPTER 7
Sequential and User Files

1) What is a sequential file
2) OPENing a sequential file
3) Printing and retrieving data — INPUT# and GET#
4) GET#
5) Detecting the end of file
6) Ending output or input with CLOSE
7) Examples of the use of sequential files

In this chapter we examine some of the techniques for storing and retrieving data under program control. Such techniques are an essential part of programs which need to store quantities of information for future use or which require more memory than is available within the 64 at one time. In effect, the proper use of the 1541 for the storage of data provides the 64 with an enormous extension of its memory, and memory that will not be cleared when the system is switched off.

This chapter is intended to refer to two types of file ideally suited for this purpose, sequential and user files. The *only* differences between these two types are that one is opened with the specifier 'S' while the other employs the specifier 'U', and that sequential files appear in the disk directory with the type 'SEQ' attached to their name while user files are labelled 'USR'. Throughout this chapter, all references to sequential files may also be read as applying to user files provided that the 'S' specifier, where present, is replaced with a 'U'. A parallel summary for user files is included at the end of the chapter, for ease of reference.

1. What is a sequential file

In essence, a sequential file is no more than a straightforward list of items, strings or numbers or a mixture of both, kept on the disk in the order in which they were stored. Items are stored in the form of individual bytes, character for character. No distinction is made within the file itself between string characters and characters stored as numeric variables, and

the layout of the characters within the file will be determined solely by the order in which they were stored and the punctuation used when they are printed to the file. Information kept within the file can normally only be accessed in the order in which it was stored. The disk drive works on an internal pointer which is set to the beginning of the file when it is OPENed. When items are reloaded from the file, the internal pointer is updated to indicate the track, sector and position within the sector of the next character of the file.

To use a sequential file, all that is necessary is to:

1) Open the file for storage or retrieval.

2) Print items to the file, or retrieve them from it, in the correct order.

3) Close the file.

2. OPENing a sequential file

We have already examined, in the chapter on housekeeping commands, some of the uses of the OPEN command, the command which corresponds in our phone analogy to the use of the red phone to issue instructions about the use of the different lines of communication.

To OPEN a sequential file, five pieces of information are needed:

1) The file number that is to be allocated to the particular usage. This number will be unique to the file specified in the OPEN command for as long as the file is in use. The permissible range for a file number in normal circumstances is 1 to 127.

2) The device number of the disk drive, which is normally 8 but can be set from 4 to 31 if multiple disks are being used.

3) The channel number or secondary address, which for a sequential file can be in the range 2–14. Unlike the cassette system there are no special rules as to the secondary address for reading or writing data. No two files can use the same secondary address at the same time, however.

4) The name of the file to be accessed, which must be a sequential file.

5) Whether the file is to be used for the storage or retrieval of information. The same file cannot be OPENed for storage and retrieval at the same time.

Having determined all these items, the format of the OPEN command for a sequential file is as follows:

OPEN< FILE NUMBER> ,< DEVICE> ,< CHANNEL> , "< FILE NAME> ,S [,R or ,W]"

Of these, FILE NUMBER, DEVICE, CHANNEL and FILENAME should be clear from the explanation given above. The ',S' attached to the filename indicates that the file type is sequential. This is then followed by an option specifying whether the file is for storage of data, ',W' which stands for 'write file', or for retrieval of data, ',R' which stands for 'read file'. If this specifier is omitted entirely, the disk drive will assume that the file being opened is for reading only. Examples of the OPEN command would include:

OPEN 1,8,2,"DATASTORE,S,W" data can be written to the file 'DATASTORE'

OPEN 1,8,2,"DATASTORE,S,R" data can be read from the file 'DATASTORE'

OPEN 1,8,2,"DATASTORE,S" exactly the same as the previous format

3. Printing and retrieving data — PRINT # and INPUT

Storage and retrieval of items of data on a disk are accomplished by two special commands, PRINT # and INPUT #. The PRINT # command places one or more items on the disk, the INPUT # command is the reverse and loads one or more items from the disk. In this section we shall provide a variety of examples of the two commands and the way in which they interact.

Format of PRINT # and INPUT #

The format of the two commands is:

PRINT #< FILE NUMBER> ,LIST OF VARIABLES

and

INPUT #< FILE NUMBER> , LIST OF VARIABLES

Punctuation with PRINT # and INPUT #

Of the two commands, INPUT # is the easier to deal with, since its use will simply depend on the way in which items have been previously stored

on the disk using PRINT#. The punctuation for a list of variables attached to an INPUT# statement is invariably a comma between each of the items to be input, as with the normal INPUT command in BASIC.

Apart from the fact that there is a special command to accomplish it, printing to a file is little different from the normal printing to the screen itself. The way the data is stored on the disk will depend upon the order in which it is printed, the punctuation (or lack of it) used when printing items and the presence or absence of carriage return characters.

As an example of the kind of problems that have to be dealt with, enter and RUN the following short program:

```
10 PRINT "ABC" "DEF";"GHI","JKL";
20 PRINT "MNO";CHR$(13);"PQR"
```

What you will see on the screen is the following:

```
ABCDEFGHI       JKLMNO

PQR
```

Examining the two line program and what it prints out, we can see that, where two items are printed consecutively without punctuation, the second is run on from the first without a gap. The same thing happens, not surprisingly, when a semicolon is inserted between two items, regardless of whether the second item is part of a second PRINT statement. Printing a comma spaces out the two items with cursor right characters (not spaces) though this is not visible on the screen. Finally, printing the carriage return character (code 13) results in items being printed on a new line, ie as completely separate from what went before. These are all useful features when outputting to the screen, but unless they are watched they can create problems when printing data to a sequential file.

To see the effects of different punctuations, enter and RUN the following program (with the disk drive on and the TEST disk inserted).

PRINT# punctuation test program

```
10 A$ = "AAA" : B$ = "BBB" : C$ = "CCC"
 : A = 100 : B = 200 : C = 300
20 R$ = CHR$(13)
1000 REM#################################
1010 OPEN 1,8,10,"SEQTEST,S,W"
1020 PRINT#1,A$B$C$
```

```
1030 PRINT#1,A$;B$;C$
1040 PRINT#1,A$,B$,C$
1050 PRINT#1,A$R$B$R$C$
1060 PRINT#1,A$ R$ B$ R$ C$
1070 PRINT#1,A$;R$;B$;R$;C$
1080 PRINT#1,A$,R$,B$
1090 PRINT#1,A;B;C
1100 PRINT#1,A,B,C
1110 PRINT#1,AR$,BR$,C
1120 PRINT#1,A R$ B R$ C
1130 PRINT#1,A;R$ B;R$ C
1140 CLOSE 1
2000 REM*********************************
2010 OPEN 1,8,6,"SEQTEST"
2020 INPUT#1,A$ : PRINT ">" A$ "<"
2030 INPUT#1,A$ : PRINT ">" A$ "<"
2040 INPUT#1,A$ : PRINT ">" A$ "<"
2050 INPUT#1,A$,B$,C$ : PRINT ">" A$
 "<>" B$ "<>" C$ "<"
2060 INPUT#1,A$,B$,C$ : PRINT ">" A$
 "<>" B$ "<>" C$ "<"
2070 INPUT#1,A$,B$,C$ : PRINT ">" A$
 "<>" B$ "<>" C$ "<"
2080 INPUT#1,A$,B$ : PRINT ">" A$ "<>"
 B$ "<"
2090 INPUT#1,A : PRINT ">" A "<"
2100 INPUT#1,A : PRINT ">" A "<"
2110 INPUT#1,A : PRINT ">" A "<"
2120 INPUT#1,A : PRINT ">" A "<"
2130 INPUT#1,A,B,C : PRINT ">" A "<>"
 B "<>" C "<"
2140 CLOSE 1
```

Output from the PRINT # punctuation test

```
>AAABBBCCC<
>AAABBBCCC<
>AAA           BBB           CCC<
>AAA<>BBB<>CCC<
>AAA<>BBB<>CCC<
>AAA<>BBB<>CCC<
>AAA           <>BBB<
> 100200300 <
```

```
> 100200300 <
> 300 <
> 300 <
> 100 <> 200 <> 300 <
```

Explanation of output of PRINT# program

Examining the output of the program will give you a very good idea of some of the pitfalls involved in storing and retrieving data in sequential files. Each line of the table corresponds to what is printed on to the file by one of the PRINT # lines in the first half of the program, and each actual string that is stored and retrieved is marked by a '>' at the beginning and a '<'. In the second half of the program, each INPUT # line is designed to pick up exactly what was printed by one PRINT # line, though why they correspond sometimes takes some thinking about:

Lines 1020 and 2020: Since there is no punctuation, the three strings are stored as one.

Lines 1030 and 2030: The semicolons achieve the same result as before.

Lines 1040 and 2040: The commas again result in a single string, but this time the elements are separated by spaces.

Lines 1050 and 2050: Separating the items to be printed with the carriage return character (CHR$(13)) ensures that they are stored as separate items.

Lines 1060 and 2060; 1070 and 2070: Provided that the carriage return character is included, separating the items with spaces or semicolons makes no difference to the results in 1050/2050.

Lines 1080 and 2080: Using the carriage return character but separating the items with commas ensures that the items are stored separately, but A$ is stored as A$ plus the padding added by the comma before the carriage return which terminates the string. Note that the final item, B$, does *not* have padding added to the front by the comma following the carriage return character (R$). The spaces created by this comma *were* stored on the disk but the 64's operating system invariably strips leading spaces from a string on INPUT.

Lines 1090 and 2090: Numeric values are treated no differently from strings when it comes to the effects of semicolons — the same effect would be observed for spaces or no punctuation at all.

Lines 1100 and 2100: Interestingly enough, using commas with numeric values produces the same effect on retrieval as the previous line. Spaces were, however, taken up on the disk itself.

Lines 1110 and 2110: The result displayed in this line merits careful consideration because it illustrates an important feature of disk storage, ie that the disk does not store empty strings. The two non-existent strings AR$ and BR$ are specified in the print statement *before* the numeric variable C. However, when we use INPUT # to pick up a numeric variable which we call A, instead of a type mismatch error indicating that we have tried to input a number when the next item was in fact a string, we find that there is no record of AR$ and BR$ on the disk.

Lines 1120 and 2120: Separating the numeric values properly from the carriage returns with spaces makes no difference since spaces are not accepted by the ROM routine which analyses the line for the BASIC interpreter. The result is the same as the previous line.

Lines 1130 and 2130: These lines show that properly separated numeric items are printed separately on the disk. Note that the separation can *either* be by punctuation of some kind *or*, in the case of strings, by the '$' sign which terminates the variable name.

Other rules for PRINT# and INPUT#

1) There is no connection between the names under which items are stored and the names under which they are retrieved. An item stored as, for example, A$ may be input again under any valid string variable name.

2) Items printed as numeric variables may always be re-input as strings. Items stored as strings *may* be capable of recall as numeric values provided that the string is a valid numeric format, eg '1234'

3) The longest string which can be PRINTed to the disk is the maximum string length of 255 characters.

4) The longest string which can be INPUT from the disk is 88 characters. Strings longer than this need to be accessed by the use of GET # (see next section).

5) Empty strings are *not* stored on disk. The disk does not keep any record of the way in which items are broken down or of the names associated with

items — these are all supplied by the BASIC system of the 64. Sending an empty string to the disk results in no characters being sent and no characters being stored.

6) String variables may be printed to the disk with leading spaces but on re-input the leading spaces are stripped by the 64. This characteristic can be used to good advantage to overcome the limitation of point 5 above. If it is important to save an empty string variable in a sequence of variables, it can be made equal to a single space. It will be properly stored and may be reloaded, when it will become once again an empty string.

7) Several characters in the character set which may be saved to disk cannot be successfully reloaded with INPUT # . These include the characters with the following ASCII codes:

0 / 13 / 32 (when in the form of a leading space) / 34 / 44 / 58

This limitation makes it next to impossible to store material such as machine code or numeric data in the form of strings to be recalled with INPUT # . In general, any character which can be visibly printed, plus the colour and cursor control characters (ie all the characters normally usable in one way or another to print to the screen), can be printed and then retrieved using INPUT # .

4. GET

Just as PRINT and INPUT have a parallel command for use with the disk drive, the normal BASIC command GET, which accepts a single character from the keyboard without the use of RETURN, has a disk equivalent in GET # . The function of GET # is to pick up the next single character from the disk at the point indicated by the drive's internal pointer. Having picked up the character, the pointer is incremented by one. As with GET, single digit numbers can be retrieved by the use of GET # , though there is no advantage to this compared to INPUT # and a significant disadvantage in that an instruction to GET # a numeric variable will produce a syntax error if a non-numeric character is encountered. In general, GET # , like GET, is used for the input of characters which will be treated as string variables.

The advantages of GET # are that it is capable of accepting characters from the disk which would be rejected by the normal INPUT # instruction *except for CHR$(0)* and that strings may be input from the disk which exceed the limit of 88 characters. The following program demonstrates some of the differences between INPUT # and GET # .

GET # demonstration program

```
10 OPEN 8,8,15,"S0:GET TEST" : CLOSE 8
1000 REM**********************************
1010 OPEN 1,8,2,"GET TEST,S,W"
1020 PRINT#1,"    AAA"
1030 PRINT#1,"AAA[3*SHIFT X]BBB" :
REM THREE CLUB SYMBOLS
1040 PRINT#1,"AAA:BBB"
1050 PRINT#1,"123"
1060 PRINT#1,123
1070 A$ = "AAAA" : A$ = A$+A$ : A$ =
A$+A$ : A$ = A$+A$ : A$ = A$+A$ : A$ =
 A$+A$
1080 PRINT#1,A$
1090 CLOSE 1
2000 REM**********************************
2005 PRINT "[CD]RETRIEVAL WITH GET#[CD]"
2010 OPEN 1,8,2,"GET TEST,S"
2020 A$ = ""
2030 GET#1,T$ : IF T$<>CHR$(13) THEN
A$ = A$+T$ : GOTO 2030
2040 PRINT ">" A$ "<"
2050 A$ = ""
2060 GET#1,T$ : IF T$<>CHR$(13) THEN
A$ = A$+T$ : GOTO 2060
2070 PRINT ">" A$ "<"
2080 A$ = ""
2090 GET#1,T$ : IF T$<>CHR$(13) THEN
A$ = A$+T$ : GOTO 2090
2100 PRINT ">" A$ "<"
2110 GET#1,A : PRINT ">" A "<"
2120 GET#1,A : PRINT ">" A "<"
2130 GET#1,A : PRINT ">" A "<" :
GET#1,A$
2140 GET#1,A$ : GET#1,A : PRINT ">" A
"<"  : INPUT#1,A$
2150 A$ = ""
2160 GET#1,T$ : IF T$<>CHR$(13) THEN
A$ = A$+T$ : GOTO 2160
2170 PRINT ">" A$ "<"
2180 CLOSE 1
3000 REM**********************************
3005 PRINT "[CD]RETRIEVAL WITH INPUT#[CD]"
```

```
3010 OPEN 1,8,2,"GET TEST,S"
3020 INPUT#1,A$ : PRINT ">" A$ "<"
3030 INPUT#1,A$ : PRINT ">" A$ "<"
3040 INPUT#1,A$ : PRINT ">" A$ "<"
3050 INPUT#1,A : PRINT ">" A "<"
3060 INPUT#1,A : PRINT ">" A "<"
3070 INPUT#1,A$ : PRINT ">" A$ "<"
3080 CLOSE 1
3090 END
```

Table of output from GET # test program

```
>    AAA<
>AAA[3*SHIFT X]BBB<
>AAA:BBB<
> 1 <
> 2 <
> 3 <
> 1 <
>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA<
>AAA<
>AAA[3*SHIFT X]BBB<
>AAA<
> 123 <
> 123 <
```

After this point, the program will stop with a STRING TOO LONG ERROR IN 2070 error message. You should enter CLOSE1[RETURN] to close the file.

Explanation of output of GET # test program

The purpose of the program is to illustrate some of the features of GET # and the differences between GET # and INPUT #. This is best done by commenting, as with the previous program Punctuation Test, on the parallel sets of lines for each individual PRINT #, INPUT # and GET #.

Lines 1020; 2020–2040; 3020: The first thing to note is the relative complexity of recovering an item by the use of GET #, requiring two program lines

compared to the one simple instruction for INPUT #. The function of the GET # lines is to continue picking up characters from the disk and adding them to an originally empty A$ until such time as a carriage return character (CHR$(13)) is encountered. The difference in the result which is produced is that GET # accepts the leading spaces which were printed as part of A$.

Lines 1030; 2050–2070; 3030: No problems are encountered by either method in retrieving normal graphics characters.

Lines 1040; 2080–2100; 3040: No problems are encountered by GET #, but INPUT # will not accept that part of the string which falls after the colon (the same would be true for a comma).

Lines 1050; 2110–2130; 3050 : The numeric value A is successfully read by INPUT #, despite the fact that it was stored as a string variable — another illustration of the fact that no distinction is made on the disk itself. When it comes to retrieval with GET #, however, each character of the string must be picked up individually, even though it is being treated as a number. Note the GET # 1, A$ on the end of the third line (2130). This must be included to take account of the carriage return character at the end of the string on the disk.

Lines 1060; 2140; 3060 : Where an item is stored to the disk as a numeric value, there is no difference from the previous method where it was stored as a string. INPUT # will retrieve the number in one operation, while GET # picks up only one character of the number at a time.

Lines 1070–1080; 2150–2170; 3070 : The PRINT # section of the program creates a string 128 characters long and prints it to the disk. GET # successfully retrieves the string but INPUT # generates a STRING TOO LONG error since the string is longer than 88 characters.

GET # and the storage of numbers in strings

One further usage of GET # should be briefly mentioned since it can be of importance in reducing the amount of space devoted on the disk to the storage of numeric data and is often used to reduce the amount of memory taken up by numeric variables. Single byte values, that is numbers in the range 0–255, can easily be stored in the form of string characters, with each character of the string representing a single value in the range 0–255, according to the ASCII code of the character. Values which require more than one byte can be represented by multiple characters.

As mentioned already, GET # has the advantage over INPUT # that it can accept a wider range of characters from the disk, the only exception being CHR$(0). When numbers are stored in the form of strings, the inability to cope with CHR$(0) is clearly a major disadvantage. A line such as the following will overcome the problem:

```
100 GET#1, T$ : T$=LEFT$(T$+CHR$(0),1)
```

As each character is obtained from the disk, CHR$(0) is added to it to form, in most cases, a two-character string. The leftmost character of this string is then taken to be the character obtained from the disk. In the majority of cases, this leads to no alteration whatsoever in what is obtained from the disk. In the case of CHR$(0), however, GET # fails to pick up anything but a null, or empty, string. When CHR$(0) is added to this and the leftmost character extracted, the result is clearly the CHR$(0) which was added to the empty string. The result is that when GET # picks up an empty string from the disk, indicating that CHR#(0) was on the disk, the result obtained is CHR$(0). The use of this technique is illustrated more fully in the example program at the end of this chapter.

5. Detecting the end of file

When retrieving data which has been stored on disk, it is clearly necessary to know when the end of the data has been reached. One method, and usually the most desirable in programming terms, is to know how much data there is before you begin to use it: the second is to let the disk drive tell you when you are in danger of reading off the end of the file.

Program to test for end of file

```
10 OPEN 8,8,15,"S0:END OF FILE" : CLOSE 8
1000 REM**********************************
1010 OPEN 1,8,2,"END OF FILE,S,W"
1020 ITEMS = 20 : PRINT#1,ITEMS
1030 FOR I = 0 TO ITEMS-1
1040 PRINT#1,I
1050 NEXT I
1060 CLOSE 1
2000 REM**********************************
2010 PRINT "[CD][CD]USING NUMBER OF DATA
ITEMS[CD][CD]"
2020 OPEN 1,8,2,"END OF FILE,S"
2030 INPUT#1,ITEMS
```

```
2040 FOR I = 0 TO ITEMS-1
2050 INPUT#1,X : PRINT X ; "/" ;
2060 NEXT
2070 CLOSE 1
2080 PRINT
3000 REM**********************************
3010 PRINT "[CD][CD]USING STATUS[CD][CD]"
3020 OPEN 1,8,2,"END OF FILE,S"
3030 INPUT#1,ITEMS
3040 INPUT#1,X : SS = ST : PRINT X ; "/" ;
3050 IF SS<>64 THEN 3040
3060 CLOSE 1
3070 PRINT
```

The first module of the program prints the number 20 to the file and then prints a series of twenty numbers. The loop used to print the values begins at zero and continues to the value of ITEMS−1, since in most cases the values to be printed would be being taken from an array (eg PRINT # 1,ARRAY(I)) and 20 items would take up positions 0−19 in the array.

The second module works by first reading the value of ITEMS back into that variable and then reading the specified number of items from the disk. In a working program, the values would normally be read back into an array, eg INPUT # 1, ARRAY(I).

The third module works by using INPUT # to pick up the items and, after each item is retrieved, sampling the value of the system variable ST. If the value of ST, which is supplied by the disk drive, equals 64, the item just input is the last in the file. In general it is better to test ST immediately after INPUT # (or GET #) to ensure that its value is not changed by some other operation before it is sampled. Note that, in this third module, the value of ITEMS has to be bypassed by the use of INPUT #. Normally, if the method employing ST were being used, ITEMS would not have been printed to the disk in the first place.

Employing ST to detect the end of a file, rather than a program-created variable, becomes essential if use is made of the COPY housekeeping command to concatenate sequential files. Sequential files can be successfully tagged on to the end of each other using COPY, and the file produced can be read normally. Clearly, it is not desirable to have a variable such as ITEMS stuck in the middle of such a file, so in such cases there is little choice but to use the ST method of detecting the end of file.

6. Ending output or input with CLOSE

When output or input has ended it is important to CLOSE the file.

Whether the file is for output or input, leaving it open ties up the file number associated with it and any attempt to open another file with that number will result in a FILE OPEN error message. More importantly, if an output file is not closed immediately, there is a chance that the program will terminate, perhaps through an error in the BASIC program, with the file still open. In this case, the file is marked in the disk directory as being unclosed and when the disk is next VALIDATEd the file will be scratched.

To check whether you have any unclosed files on a disk, simply load the directory and examine the file type for each file. Any files which have not been correctly closed will have their file type preceded by an asterisk (*). It is unlikely that a file in this condition will be capable of recovery.

7. Examples of the use of sequential files

In this section, we give some examples of the way in which a sequential file may be employed to store various types of data. Using the methods illustrated in the program below, you should find little difficulty in using sequential files to store the data for a normal applications program.

Sequential file test program

```
10 OPEN 8,8,15,"S0:TEST FILE" : CLOSE 8
1000 REM**********************************
1010 REM INITIALISE THE ARRAYS
1020 CLR
1030 R$ = CHR$(13)
1040 DIM ARRAY$(100),ARRAY(100)
1050 A$ = ""
1060 FOR I = 0 TO 100
1070 IF I/5=INT(I/5) THEN A$ = A$+"A"
1080 ARRAY$(I) = A$
1090 NEXT I
1100 FOR I = 0 TO 100
1110 ARRAY(I) = I
1120 NEXT I
1130 NUMBER$ = "9876543210123456789"
1140 CODE$ = ""
1150 FOR I = 0 TO 100
1160 CODE$ = CODE$+CHR$(I)
1170 NEXT I
1180 V1 = 1111
1190 V2 = 222
```

```
1200 V3 = 33
1210 VB$ = "BBBB"
1220 VC$ = "CCCCCCC"
1230 VD$ = "DDD"
1240 ITEMS = 101
2000 REM**********************************
2010 OPEN 1,8,2,"TEST FILE,S,W"
2020 PRINT#1,ITEMS;R$;V1;R$;VB$
2030 FOR I = 0 TO ITEMS-1
2040 PRINT#1,ARRAY$(I)
2050 NEXT I
2060 PRINT#1,V2;R$;VC$
2070 FOR I = 0 TO ITEMS-1
2080 PRINT#1,ARRAY(I)
2090 NEXT I
2100 PRINT#1,NUMBER$
2110 PRINT#1,CODE$
2120 PRINT#1,V3;R$;VD$
2130 CLOSE 1
3000 REM**********************************
3010 OPEN 1,8,2,"TEST FILE,S"
3020 INPUT#1,ITEMS,V1,VB$
3030 FOR I = 0 TO ITEMS-1
3040 INPUT#1,ARRAY$(I)
3050 NEXT I
3060 INPUT#1,V2,VC$
3070 FOR I = 0 TO ITEMS-1
3080 INPUT#1,ARRAY(I)
3090 NEXT I
3100 INPUT#1,NUMBER$
3110 CODE$ = ""
3120 FOR I = 0 TO ITEMS-1
3130 GET#1,T$ : T$ = LEFT$(T$+CHR$(0),1)
3140 CODE$ = CODE$+T$
3150 NEXT
3160 INPUT#1,V3,VD$
3170 CLOSE 1
4000 REM**********************************
4010 FOR I = 0 TO ITEMS-1
4020 PRINT ">" ARRAY$(I) "< "
4030 NEXT
4040 INPUT"[CD]PRESS [RVS ON]RETURN
[RVS OFF] TO CONTINUE ";T$
4050 PRINT
```

```
4060 FOR I = ITEMS-1 TO 0 STEP -1
4070 PRINT ARRAY(I) "/" ;
4080 NEXT
4090 PRINT
4100 INPUT"[CD]PRESS [RVS ON]RETURN
[RVS OFF] TO CONTINUE ";T$
4110 PRINT
4120 FOR I = 1 TO LEN(CODE$)
4130 PRINT ASC(MID$(CODE$,I,1)) "/" ;
4140 NEXT I
4150 PRINT
4160 INPUT"[CD]PRESS [RVS ON]RETURN
[RVS OFF] TO CONTINUE ";T$
4170 PRINT
4180 PRINT "NUMBER$ = " NUMBER$
4190 PRINT "V1 =" V1
4200 PRINT "V2 =" V2
4210 PRINT "V3 =" V3
4220 PRINT "VB$ = " VB$
4230 PRINT "VC$ = " VC$
4240 PRINT "VD$ = " VD$
4250 END
```

Explanation of sequential file test program

The purpose of the program is to print a variety of data to the disk from arrays such as those which might be used in a program, then to retrieve that data from disk and replace it into arrays.

The first module, lines 1000–1240, set up the arrays. ARRAY$ contains 101 strings which consist of increasing numbers of As — the number of As increasing by one for every five strings stored. ARRAY contains the numbers 0–101 in ascending order. The string NUMBER$ contains the characters '9876543210123456789'. The string CODE$ contains 101 characters whose ASCII values are 0–100. The purpose of CODE$ is to demonstrate the technique of storing single-byte numbers in strings, a method which is extremely economical in terms of memory. Each character of CODE$ will be important, not so much as a character, but because its ASCII value represents a value which must be stored. In addition there are seven other variables whose values can be observed in lines 1180–1240, including ITEMS, whose value of 101 reflects the number of data items which are to be stored in each of the arrays. The value of ITEMS would normally be determined by the program itself as new items of data were added or deleted.

The second module, lines 2000–2130, prints the contents of the arrays and variables to the disk.

The most complex of the modules is clearly module three, lines 3000–3170. The purpose of this module is, using techniques outlined in this chapter, to recover the data from the disk and store it back into the various arrays.

The fourth module, lines 4000 onwards, has the simple purpose of printing the retrieved data to the screen to demonstrate the success of the procedure.

Summary of user files

1) User files are an essential tool for serious programming which will almost invariably involve the need to store more or less complex data from one run of the program to another.

2) User files must be OPENed with a command such as:

   OPEN 1,8,2,"USERFILE,U,W"

   which includes the specification of:

   a) a unique file number

   b) the device number (normally 8)

   c) the unique channel number (2–14)

   d) the name of the file, including the termination ',U' to indicate that it is user.

   e) whether the file is for output or input (',W' or ',R').

3) Data is placed into User files by the use of the PRINT # statement.

4) Data is retrieved from user files using either the INPUT # or the GET # statement.

5) User files must be CLOSEd when no further use is to be made of them, otherwise they risk being invalidated on the disk and the data they contain lost.

Summary of sequential files

1) Sequential files are an essential tool for serious programming which will almost invariably involve the need to store more or less complex data from one run of the program to another.

2) Sequential files must be OPENed with a command such as:

   OPEN 1,8,2,"SEQFILE,S,W"

   which includes the specification of:

   a) a unique file number

   b) the device number (normally 8)

   c) the unique channel number (2–14)

   d) the name of the file, including the termination ',S' to indicate that it is sequential

   e) whether the file is for output or input (',W' or ',R').

3) Data is placed into sequential files by the use of the PRINT # statement.

4) Data is retrieved from sequential files using either the INPUT # or the GET # statement.

5) Sequential files must be CLOSEd when no further use is to be made of them, otherwise they risk being invalidated on the disk and the data they contain lost.

CHAPTER 8
Program Files

1) What is a program file
2) The structure of a BASIC program file
3) Using program files for other purposes
4) Output of program files to printer
5) Merging programs using program files on disk
6) Renumbering a program file on disk

When the directory of a disk is displayed it is likely that it will contain files of at least two types, sequential and program. So far, we have talked about the simple process of LOADing and SAVEing and have analysed sequential files in some detail. There are, however, many extremely useful operations that can be carried out on program files apart from LOAD and SAVE, but, before they can be understood, there needs to be some explanation of the nature of a program file as it is stored on disk.

What are the main differences between these two important file types, the program and sequential? The answer is quite simple: the difference between 'PRG' files and 'SEQ' files is simply that one is called 'PRG' and the other is called 'SEQ'. Try the following experiment:

1) Enter this single line program:

```
10 REM THIS IS A "SEQUENTIAL PROGRAM"!
```

2) Type:

```
SAVE "SEQPROG,S",8 [RETURN]
```

3) Examine the directory of the disk and you will find that there is a file on the disk called SEQPROG, and that it is a sequential file.

4) Type:

```
NEW [RETURN]
```

to clear the test program out of memory.

5) Type:

```
LOAD "SEQPROG,S",8 [RETURN]
```

6) LIST the program and you will find that you have just successfully LOADed a program which is stored on disk as a sequential file.

1. What is a program file

We have already clearly demonstrated that a program can be saved in the form of a sequential file? The question then arises, what is a program file? The simple answer is that 'PRG', the indicator of a program file on the disk, is a flag used *by the 64* not by the disk drive itself, and indicates to the 64 that this is a file in which it would normally expect to find a program stored. When told to SAVE a program, unless another type of file is specified, the 64 sets up a 'PRG' file: it then writes the current start address of BASIC and the contents of the program memory, byte by byte, into that file, though the file itself is no different from a sequential file, except that it has been given the special designation 'PRG'. Similarly, when the 64 is instructed to LOAD a program, unless it is told otherwise, it searches for a program file with the specified name and loads its contents back into program memory.

To sum up then, a program file is a sequential file containing the contents of BASIC memory and having a special designation 'PRG' which indicates the type of file the BASIC interpreter defaults to on LOADing and SAVEing.

2. The structure of a BASIC program file

Not surprisingly, program files reflect the structure of programs, and so to understand their structure we have first to say something about the way programs are stored in memory.

BASIC programs normally occupy an area of memory beginning at address 2049. Each line of the program is represented, in order, and contains four distinct sections:

1) Two bytes known as 'link bytes'. These record the address of the start of the *next* line of program, ie the byte following the end of the current line.

2) Two bytes which record the line number.

3) An indeterminate number of bytes representing the text of the line.

4) One byte containing the value zero, which indicates the end of the line.

At the end of the program, following the zero terminating the final line, are two more bytes containing the value zero which act as a marker for the BASIC interpreter.

When it comes to storing a program on disk, the format is almost exactly the same as the format in memory. The single exception is that, since it is possible for a BASIC program to start almost anywhere in memory, the disk file commences with two bytes which record where the start of the particular program lay when it was SAVEd. Given below is a program which reads a program file on disk as if it were an ordinary sequential file. At the present time it is set up to read its own program file, so must be entered, SAVEd, and then allowed to read the disk on which it is SAVEd.

Program reader

```
10 OPEN 8,8,8,"PROG READ"
15 GET#8,T$ : GET#8,T$
20 GET#8,T$ : GET#8,T$
30 IF T$="" THEN 110
40 GET#8,T$ : T = ASC(T$+CHR$(0))
50 GET#8,T$ : T = ASC(T$+CHR$(0))*256+T
60 PRINT T " ";
80 GET#8,T$ : IF T$>"Z" THEN T$ = "["+
MID$(STR$(ASC(T$)),2)+"]"
90 IF T$<>"" THEN PRINT T$ ; : GOTO 80
95 PRINT
100 GOTO 20
110 CLOSE 8
120 END
```

Explanation of program reader

Line 15: Obtain the first two bytes of the file — nothing is done with them.

Lines 20–30: Pick up a pair of link bytes. If the second of them, which should be the high byte, is zero then the end of the file must have been reached.

Lines 40–60: Pick up and print the line number.

Lines 80–90: The bytes which make up the line are picked up and printed

one by one. Some of them will be single-byte tokens for keywords and will not be printable. These are represented by their ASCII value in square brackets.

Output of program reader

```
 10  [159] 8,8,8,"PROG READ"
 15  [161]#8,T$ : [161]#8,T$
 20  [161]#8,T$ : [161]#8,T$
 30  [139] T$[178]"" [167] 110
 40  [161]#8,T$ : T [178] [198](T$[170][
199](0))
 50  [161]#8,T$ : T [178] [198](T$[170][
199](0))[172]256[170]T
 60  [153] T " ";
 80  [161]#8,T$ : [139] T$[177]"Z" [167]
 T$ [178] "[91]"[170][202]([196]([198](T
$)),2)[170]"[93]"
 90  [139] T$[179][177]"" [167] [153] T$
 ; : [137] 80
 95  [153]
 100  [137] 20
 110  [160] 8
 120  [128]
```

The reason for giving the output of the program is not in order to analyse it in great detail, but merely to illustrate roughly the structure of a program file. The values contained in square brackets are the ASCII codes of 'tokens', the single-character shortened forms of the BASIC keywords which the program contained. Apart from these tokens, you can see that the rest of the content of the program is stored in a very straightforward format, much as you would expect to see it on the screen.

3. Using program files for other purposes

Just as sequential files can be used for the purposes of storing programs, so program files can be used for other purposes. It is perfectly possible to use a program file for the storage of data by using a command like:

OPEN 1,8,2,"DATASTORE,P,W"

and the data retrieved with:

OPEN 1,8,2,"DATASTORE,P,R"

Having opened the file correctly, it may be manipulated in all the ways described in the chapter on sequential files.

A more common use of program files, however, is to save areas of memory holding material such as machine code programs or the screen display. This is a fast method of storing and retrieving data compared with the laborious process of printing and inputting item by item from disk. Given below is a short routine taken from a working program which saves the contents of screen memory to disk as a program file.

Routine to save memory as a program file

```
14040 IF A=1 THEN END
14050 FOR I=0 TO 3 : A%(I)=PEEK(43+I)
 : NEXT
14060 POKE 43,0 : POKE 44,4 : POKE 45,
0 : POKE 46,8
14070 SAVE "SCREEN",8
14080 POKE 43,A%(0) : POKE 44,A%(1) :
 POKE 45,A%(2) : POKE 46,A%(3)
14090 PRINT "[CLR]"
15000 FOR I = 55296 TO 56319 : POKE I,1
 : NEXT
15005 A = 1
15010 LOAD "SCREEN",8,1
```

Leave the program listing on the screen and RUN the program (with a disk in the drive). The disk drive should now start up, and the screen memory is saved to the disk. The screen should then clear and the program listing reappear in white. Finally the program will end with the cursor flashing near the top of the screen.

Explanation of routine:

Lines 14050–14060: The registers which record the start and end of the BASIC program area are first recorded, then altered so that they indicate the start and finish of the area of memory to be SAVEd — in this case the screen memory.

Line 14070: A simple SAVE instruction now suffices to store the specified area of memory to disk as if it were a program.

Line 14080: The original values of the registers recorded in 14050 are replaced so that the system knows where the BASIC program really is. Note that this cannot be done with a loop since the system will be confused as to the location in memory of the value of the loop variable by the temporary alteration to the end of BASIC — which is where variables begin. Using an array to store the values creates no problems since we have not altered the register which indicates the position of the start of arrays in memory.

Lines 14090–15000: The screen is now cleared. Since we have not chosen to save the contents of the colour memory along with the screen, the value 1 is POKEd into all 1024 bytes of colour memory, specifying that any characters in screen memory will appear white. If this were not done, the reloaded screen would not be visible.

Line 15010: The program file containing the screen is now LOADed. The only difference between the instruction and a normal LOAD is that a secondary address of one is tagged on to the end. The effect of this is to tell the system to reload the program file into exactly the same area of memory as that from which it was taken. Note that, since the program file is reloaded under program control and not from an instruction in direct mode, the start and end of BASIC pointers are not changed to reflect the position and size of what is being reloaded. In other words, though a program called SCREEN is being LOADed, the system continues to regard the program which called for the LOADing of SCREEN as the operative program.

Lines 14040 and 15005: These two lines appear not to fit easily into the development of the program as outlined above. When program files are LOADed under program control, the 64 automatically re-RUNs the BASIC program, although without clearing the variables. Rather than be caught in an endless loop, the variable A is set to one when the screen has been reloaded and the value of this variable is used as a flag to inform the system that there is no need to RUN the program again. If the routine were embedded in a larger program, line 14040 would normally return execution to the point after SCREEN had been reloaded rather than simple ENDing.

Although SAVEing the screen can be a neat and useful procedure, the ability to SAVE areas of memory is probably more often employed to store machine code programs on to disk for fast access.

4. Output of program files to printer

Once the nature of program files is understood, it becomes possible to use

and manipulate them in a variety of useful ways — often with far greater ease than could be achieved with a program in memory. One example of this is the formatting of a program for a printed listing. There is no doubt that the kind of 'square brackets format' adopted for the programs in this and many other Commodore programming books, where control characters are represented by short mnemonics in square brackets rather than by inverse graphics characters, is far clearer to read. It is possible to attach a routine to a program in memory to list it in this format, but much simpler to adopt the method used in the program given below of reading the program from the disk and listing from that source.

Square brackets listing program

```
10 GOTO 16000
5000 REM#********************************
5010 REM DISC ERROR STATUS
5020 REM********************************
5030 INPUT#15,EN,EM$,ET,ES
5040 IF EN=73 THEN 5030
5050 IF EN<20 THEN RETURN
5060 PRINT "[CLR][CD][CD][CD][CD] ******
**********************************"
5070 PRINT "[CD][CD]                DISC
 ERROR"
5080 PRINT "[CD][CD]      ERROR -" EN
 " " EM$
5090 PRINT "[CD]      AT TRACK -" ET "
 AND SECTOR -"ES
5100 PRINT "[CD][CD]    PROGRAMME
 EXECUTION TERMINATED"
5110 PRINT "[CD][CD][CD] ***************
************************"
5120 CLOSE 15
5130 CLR
5140 END
13000 REM#********************************
13010 REM LIST DISC FILE INITIALISATION
13020 REM********************************
13030 DIM C$(255,1)
13040 FOR I = 0 TO 255 : C$(I,0) = CHR$
(I) : C$(I,1) = CHR$(I) : NEXT
13050 AD = 41118 : T1 = 128
13060 T$ = ""
```

```
13070 T = PEEK(AD) : AD = AD+1
13080 T$ = T$+CHR$(T AND 127)
13090 IF T<128 THEN 13070
13100 C$(T1,0) = T$ : T1 = T1+1
13110 IF PEEK(AD) THEN 13060
13120 RESTORE
13130 READ T$ : IF T$<>"FOR LIST" THEN
 13130
13140 READ T
13150 IF T<0 THEN RETURN
13160 READ T$ : C$(T,1) = "[" +T$+"]"
13170 GOTO 13140
14000 REM*********************************
14010 DATA "FOR LIST"
14020 REM*********************************
14030 DATA 5,WHT,17,CD,18,RVS,19,HOME
14040 DATA 20,DEL,28,RED,29,CR,30,GRN
14050 DATA 31,BLU,129,ORANGE,133,F1
14060 DATA 134,F3,135,F5,136,F7,137,F2
14070 DATA 138,F4,139,F6,140,F8,144,BLK
14080 DATA 145,CU,146,RVO,147,CLR
14090 DATA 148,INST,149,BROWN,150,LT. RED
14100 DATA 151,GREY 1,152,GREY 2
14110 DATA 153,LT. GRN,154,LT. BLU
14120 DATA 155,GREY 3,156,PUR,157,CL
14130 DATA 158,YEL,159,CYN
14140 DATA -1
15000 REM*********************************
15010 REM DO LIST
15020 REM*********************************
15030 OPEN 3,DEV,3,NR$ : GOSUB 5000
15040 OPEN 4,4
15050 GET#3,T$ : GET#3,T$
15060 GET#3,T$ : GET#3,T$
15070 QU = 0
15080 IF T$="" THEN CLOSE 3 : CLOSE 4
 : RETURN
15090 GET#3,T$ : T = ASC(T$+CHR$(0))
15100 GET#3,T$ : T = ASC(T$+CHR$(0))
*256+T
15110 PRINT#4,MID$(STR$(T),2) " " ;
15120 GET#3,T$
15130 IF T$=CHR$(34) THEN QU = 1-QU
15140 IF T$<>"" THEN PRINT#4,C$(ASC
```

```
(T$),QU) ; : GOTO 15120
15150 PRINT#4
15160 GOTO 15060
16000 REM#*******************************
16010 REM INPUT DATA FOR LIST
16020 REM*******************************
16030 GOSUB 13000
16040 INPUT "DEVICE NUMBER ? 8[CL][CL]
[CL]";DEV
16050 OPEN 15,DEV,15
16060 INPUT "FILE TO BE PRINTED ";NR$
16070 GOSUB 15000
16080 INPUT "ANY MORE FILES (Y/N) ? N
[CL][CL][CL]";T$
16090 IF T$="N" THEN CLOSE 15 : END
16100 IF T$<>"Y" THEN PRINT "[CU][CU]
[CU]" : GOTO 16080
16110 GOTO 16060
```

Explanation of square brackets listing program

Lines 16000–16110: The purpose of this module is to allow the user to specify the device number of the disk on which the program is stored, the name of the file, and to open the error channel to that device. When output of the program to the printer has ended, the user has the option of specifying another file to be listed.

Lines 13000–14140: The first of these two modules initialises the program, largely using data drawn from the second module.

Lines 13030–13040: The first step is to set up a 256-line array, each line having two elements. To begin with, both of the elements are set equal to the character which has an ASCII value equal to the line number in the array (0–255).

Line 13050: AD is the address of the start in memory of the table of BASIC keywords. T1 is set to 128, the value of the first character in the character set used as a token for a BASIC keyword in a program file.

Lines 13060–13090: Characters are PEEKed from the BASIC keyword table and added to a temporary string until a character with a value greater than 127 is encountered. In the keyword table itself, the end of each keyword is marked by the final character having its value ANDed with 128.

Line 13100: Starting at line 128 of the array, just as the keyword tokens start at CHR$(128), the keyword is stored in element zero of the relevant line of C$.

Line 13110: The process continues until a byte containing the value zero is found, marking the end of the keyword table.

Lines 13120–13130: The beginning of the DATA module is marked with a data statement reading DATA FOR LIST. These lines read and discard any data up to that point. In this particular program there *are* no DATA statements before line 13130, but the two modules are meant to be relocatable in a larger program if you so desire.

Lines 13140–13170: The mnemonics for the various control characters which the program is designed to reformat are read from the DATA statements and placed into element 1 of the relevant line of the array C$. By the time the module has finished its work, the zero elements of C$ will contain keywords which correspond to the ASCII value of any character used as a token and the '1' elements will contain the square bracket representations of any characters used to represent control characters. The reason that the two types have to be kept in two different halves of the array is that the same characters can serve different purposes according to whether or not they are enclosed in quotes. Thus CHR$(137) will represent the F2 key if it falls within a string and GOTO if does not.

Lines 15000–15160: This module performs the task of reading the program line by line from the disk and printing it out. It is similar in structure to the PROG READ program given in the second section of this chapter.

Lines 15030–15040: A file is opened to the specified device under the specified program name. A second file is opened to the printer to receive the output.

Line 15050: The first two bytes of the file, which record the start address of the program in BASIC, are read and discarded.

Lines 15060 – 15080: These lines pick up what should be the link bytes at the beginning of the line and check that they are not in fact the end of file markers. The variable QU is used to record whether the characters being picked up are within quotes — it is set to zero at the beginning of the line.

Lines 15090–15100: The two bytes containing the line number are picked up from the disk and the line number printed.

Lines 15120–15140: The characters of the line are picked up from the disk one by one. If the character picked up is a quotation mark, the value of QU is toggled between zero and one. For other characters, the contents of one or other side of a line in the array C$ is printed, depending on the value of QU. Most of the characters in C$ will be normal printing or graphics characters but others, as we have seen, have been replaced with keywords or square brackets representations to correspond with characters which are used as keyword tokens or control characters.

Lines 5000–5140: The disk status routine we examined earlier.

Using the program

To make use of the program, first LOAD it into memory. Ensure that the disk drive is on and that it holds the disk containing the program you wish to list. The printer must be on. Run the program and respond to the prompts for device number and program name. The list of the square brackets mnemonics with which the program replaces control characters is given in the Notes on Program Listings, at the beginning of this book.

5. Merging programs using program files on disk

The ability to merge programs, loading them both into memory at the same time in such a way that they become one, is extremely useful, but not something that is built into the Commodore 64. The program given below uses the ability to read program files sequentially to merge two BASIC program files to form a third which consists of the lines from both programs. In the event that there are lines with the same number in both files, the lines contained in the first file specified will be overwritten by those contained in the second.

Program to merge two BASIC program files

```
10 GOTO 7000
5000 REM#********************************
5010 REM DISC ERROR STATUS
5020 REM********************************
5030 INPUT#15,EN,EM$,ET,ES
5040 IF EN=73 THEN 5030
5050 IF EN<20 THEN RETURN
5060 PRINT "[CLR][CD][CD][CD][CD] ******
```

```
**********************************"
5070 PRINT "[CD][CD]                DISC
 ERROR"
5080 PRINT "[CD][CD]      ERROR -" EN
 " " EM$
5090 PRINT "[CD]      AT TRACK -" ET
 " AND SECTOR -"ES
5100 PRINT "[CD][CD]    PROGRAMME
EXECUTION TERMINATED"
5110 PRINT "[CD][CD][CD] ***************
************************"
5120 CLOSE 15
5130 CLR
5140 END
7000 REM#*********************************
7010 REM MERGE 2 PROGRAMME FILES
7020 REM*********************************
7030 INPUT "INPUT FILE 1 NAME";N1$
7040 INPUT "INPUT FILE 2 NAME";N2$
7050 INPUT "OUTPUT FILE NAME";N3$
7060 INPUT "DEVICE NUMBER ? 8[CL][CL]
[CL]";DEV : OPEN 15,DEV,15
7070 PRINT "MERGING " N1$ " WITH " N2$ "
7080 PRINT "TO MAKE " N3$
7090 GOSUB 8000
7100 CLOSE 15
7110 END
8000 REM#*********************************
8010 REM MERGE 2 PROGRAMME FILES
8020 REM*********************************
8030 L3$ = "" : L4$ = ""
8040 OPEN 3,DEV,3,N1$ : GOSUB 5000
8050 OPEN 4,DEV,4,N2$ : GOSUB 5000
8060 OPEN 5,DEV,5,N3$+",P,W" : GOSUB 5000
8070 FI = 3
8080 T$ = ""
8090 GOSUB 10000: GOSUB 10000: PRINT#5,T$ ;
8100 FI = 4 : GOSUB 10000: GOSUB 10000
8110 GOSUB 11000
8120 GOSUB 12000
8130 IF L3<=L4 THEN 8160
8140 IF LEN(L4$)>2 THEN PRINT#5,L4$;
8150 GOTO 8120
8160 IF LEN(L3$)>2 THEN PRINT#5,L3$;
```

```
8170 IF L3=L4 AND L3<65536 THEN 8110
8180 GOSUB 11000
8190 IF L3<>65536 OR L4<>65536 THEN 8130
8200 PRINT#5,CHR$(0) CHR$(0) ;
8210 CLOSE 3
8220 CLOSE 4
8230 CLOSE 5
8240 RETURN
9000 REM*********************************
9010 REM READ LINE INTO T$,LINE NO IN T
9020 REM*********************************
9030 T$ = ""
9040 GOSUB 10000: GOSUB 10000
9050 IF T1$=CHR$(0) THEN 9100
9060 GOSUB 10000: T = ASC(T1$)
9070 GOSUB 10000: T = ASC(T1$)*256+T
9080 GOSUB 10000
9090 IF T1$<>CHR$(0) THEN 9080
9100 RETURN
10000 REM*********************************
10010 REM READ A SINGLE CHR FORM DISC
10020 REM*********************************
10030 GET#FI,T1$
10040 T1$ = LEFT$(T1$+CHR$(0),1)
10050 T$ = T$+T1$
10060 RETURN
11000 REM*********************************
11010 REM READ LINE FROM FILE 3
11020 REM*********************************
11030 IF LEN(L3$)=2 THEN L3 = 65536 : GOTO
 11080
11040 FI = 3
11050 GOSUB 9000
11060 L3$ = T$
11070 L3  = T
11080 RETURN
12000 REM*********************************
12010 REM READ LINE FROM FILE 4
12020 REM*********************************
12030 IF LEN(L4$)=2 THEN L4 = 65536 : GOTO
 12080
12040 FI = 4
12050 GOSUB 9000
12060 L4$ = T$
```

```
12070 L4   = T
12080 RETURN
```

Explanation of merge program

Lines 7000–7110: This module allows the user to specify the names of the programs to be employed in the merge and the name of the program file to be produced.

Lines 10000–10080: This module, which services others within the program, uses GET # to read a single byte from a file whose number is specified in the variable FI. The character contained in the byte is added to a temporary string, T$, for use by other modules.

Lines 9000–9100: This module is exactly equivalent to those we have examined previously in this chapter which print out the contents of a program line. For an explanation of the method see the second section of this chapter.

Lines 11000–11080 and 12000–12080: These two subroutines store the lines obtained by the previous modules in separate variables for the two files, together with the line numbers reached. If the line length left in the variable L4$ or L3$ is two, then on a previous examination of the relevant file the end of file marker has been detected. In that case, no further reading is done from the file.

Lines 8000–8240: The control module, which allocates work amongst the remainder of the program.

Lines 8040–8060: The two files to be merged are opened. The output file is opened with the suffix ',P,W' in order that material can be written to it.

Lines 8070–8090: The two bytes which record the start position in memory of program 1 are read from file 3 and written to the new program file. The merged program will therefore have the same start address as program 1.

Line 8100: The start address of program 2 is read and discarded.

Lines 8110–8180: This section picks up lines from the two program files and examines them to see which line number comes first, then prints that line to the output file and picks up another from the file which supplied the line which has just been printed. If the two lines have the same number, it is

the line from program 2 which is printed and the corresponding line from program 1 is discarded. In every case, lines are only printed if the variables L3$, L3, L4$ and L4 do not indicate that the end of file has been reached in the relevant program.

Lines 8190–8230: When the end of file has been reached in both programs, two zeros are added to the end of the output file and all the files closed.

Using the merge program

Once the program is in memory, the disk drive must contain a disk holding *both* the programs to be merged and with sufficient free space to allow the creation of the merged program. On RUNning the program, the user is required to specify the name of the first and second program to be merged, together with the name of the merged file which is to be created. It should be stressed that, in the case of lines with the same number, the lines taken from the second file named will overwrite the lines from the first file.

6. Renumbering a program file on disk

The ability to renumber a program is a useful addition to the armoury of a programmer who wishes his or her programs to be attractively laid out and easy to read. The program presented in this section accomplishes renumbering of a program file on disk, once again using the ability to read from one file into another to accomplish the task.

Renumber program

```
1000 REM#********************************
1010 REM RENUMBER ANY DISC FILE
1020 REM********************************
1030 DEFFNHI(X) = INT(X/256) : DEFFNLO
(X) = X-FNHI(X)*256
1040 INPUT "FILE TO BE RENUMBERED ";NR$
1050 NW$ = "TEMPORARY"
1060 INPUT "DEVICE NUMBER ? 8[CL][CL][CL]
";DEV
1070 N$ = CHR$(0)
1080 INPUT "START AT LINE ? 10[CL][CL]
[CL][CL]";SL
1090 INPUT "STEP ? 10[CL][CL][CL][CL]
```

```
";SP
1100 LMAX = 2500 : DIM LN(LMAX,1)
1110 OPEN 15,DEV,15 : PRINT "[CLR]"
 : GOSUB 5000
1120 PRINT#15,"S0:TEMPORARY" : GOSUB 5000
1130 GOSUB 2000
1140 GOSUB 3000
1150 PRINT#15,"S0:"+NR$ : GOSUB 5000
1160 PRINT#15,"R0:"+NR$+"=0:TEMPORARY"
 : GOSUB 5000
1170 CLOSE 15
1180 GOSUB 4000
1190 END
2000 REM*********************************
2010 REM PASS 1
2020 REM*********************************
2030 PTR = 0 : NL = SL
2040 OPEN 3,DEV,3,NR$ : GOSUB 5000
2050 GET#3,T$ : GET#3,T$
2060 GET#3,T$ : GET#3,T$
2070 IF T$="" THEN CLOSE 3 : RETURN
2080 GET#3,T$ : T = ASC(T$+CHR$(0))
2090 GET#3,T$ : T = ASC(T$+CHR$(0))*256+T
2100 PRINT "[HOME][CD][CD][CD][CD][CD]
[CD][CD][CD][CD]CALCULATING NEW LINE
 NUMBER FOR" T "          "
2110 GET#3,T$
2120 IF T$=CHR$(143) THEN GET#3,T$ :
 IF T$= "#" THEN NL = INT(1+NL/1000)*1000
2130 IF T$<>"" THEN 2110
2140 LN(PTR,1) = NL
2150 LN(PTR,0) = T
2160 NL = NL+SP
2170 IF NL>63999 THEN T$ = "STEP TOO
 BIG" : GOTO 6000
2180 PTR = PTR+1
2190 IF PTR<=LMAX THEN 2060
2200 T$ = "TOO MANY LINES TO RENUMBER"
2210 GOTO 6000
2220 RETURN
3000 REM*********************************
3010 REM PASS 2
3020 REM*********************************
```

```
3030 OPEN 3,DEV,3,NR$ : GOSUB 5000
3040 OPEN 4,DEV,4,NW$+",P,W" : GOSUB 5000
3050 GET#3,T$:PRINT#4,LEFT$(T$+N$,1);
:GET#3,T$:PRINT#4,LEFT$(T$+N$,1);
3060 FOR J = 0 TO PTR-1
3070 QU = 0
3080 GET#3,T$:PRINT#4,LEFT$(T$+N$,1);
:GET#3,T$:PRINT#4,LEFT$(T$+N$,1);
3090 GET#3,T$
3100 GET#3,T$
3110 PRINT "[HOME][CD][CD][CD][CD][CD]
[CD][CD][CD][CD][CD][CD][CD][CD][CD]
 RENUMBERING LINE "MID$(STR$(LN(J,0))
+"       ",2,6);
3120 PRINT " AS" LEFT$(STR$(ABS(LN
(J,1)))+"       ",6)
3130 T = ABS(LN(J,1))
3140 PRINT#4,CHR$(FNLO(T)) CHR$(FNHI
(T)) ;
3150 GET#3,T$
3160 IF T$=CHR$(34) THEN QU = NOT QU
3170 PRINT#4,LEFT$(T$+N$,1) ;
3180 IF(T$=CHR$(137)ORT$=CHR$(141)ORT$
=CHR$(167)ORT$=CHR$(138))ANDNOTQUTHEN3240
3190 IF T$<>"" THEN 3150
3200 NEXT J
3210 PRINT#4,N$ N$ ;
3220 CLOSE 3 : CLOSE 4
3230 RETURN
3240 GET#3,T$ : IF T$=" " THEN PRINT#4,
T$; : GOTO 3240
3250 IF T$<"0" OR T$>"9" THEN 3170
3260 T1$ = T$
3270 GET#3,T$
3280 IF T$=" " THEN 3270
3290 IF T$>="0" AND T$<="9" THEN T1$ =
 T1$+T$ : GOTO 3270
3300 T = VAL(T1$)
3310 T1 = -1
3320 FOR I = 0 TO PTR-1
3330 IF LN(I,0)=T THEN T1 = I
3340 NEXT
3350 T1$ = "????"
3360 IF T1>0 THEN T1$ = MID$(STR$(LN
```

```
(T1,1)),2) : GOTO 3380
3370 LN(J,1) = -ABS(LN(J,1))
3380 PRINT#4,T1$ ;
3390 IF T$="," THEN PRINT#4,T$; : GOTO
 3240
3400 PRINT#4,LEFT$(T$+N$,1) ;
3410 GOTO 3190
4000 REM#################################
4010 REM UNDEFINED LINES
4020 REM#################################
4030 PRINT "[CLR][CD][CD]UNDEFINED LINE
 NUMBERS IN THE FOLLOWING"
4040 PRINT "[CD]            RENUMBERED
 LINES[CD]"
4050 UL = 0
4060 FOR I = 0 TO PTR-1
4070 IF LN(I,1)<0 THEN PRINT -LN(I,1) ,
 : UL = UL+1
4080 NEXT
4090 IF UL=0 THEN PRINT SPC(17) "NONE"
4100 PRINT "[CD][CD][CD]"
4110 RETURN
5000 REM#################################
5010 REM DISC ERROR STATUS
5020 REM#################################
5030 INPUT#15,EN,EM$,ET,ES
5040 IF EN=73 THEN 5030
5050 IF EN<20 THEN RETURN
5060 PRINT "[CLR][CD][CD][CD][CD] ******
**********************************"
5070 PRINT "[CD][CD]                DISC
 ERROR"
5080 PRINT "[CD][CD]      ERROR -" EN
 " " EM$
5090 PRINT "[CD]      AT TRACK -" ET "
 AND SECTOR -"ES
5100 PRINT "[CD][CD]     PROGRAMME
 EXECUTION TERMINATED"
5110 PRINT "[CD][CD][CD] ***************
************************"
5120 CLOSE 15
5130 CLR
5140 END
```

```
6000 REM#################################
6010 REM PROGRAMME ERROR
6020 REM#################################
6030 PRINT "[CLR][CD][CD][CD][CD] ******
**********************************"
6040 PRINT "[CD][CD][CD]          FATAL
 PROGRAMME ERROR"
6050 PRINT "[CD][CD][CD]        ERROR - " T$
6060 GOTO 5100
```

Explanation of renumber program

Lines 1000–1190: The control module which allows the user to specify the file to be renumbered and allocates work amongst the rest of the program.

Line 1030: These two functions will be used to translate a decimal value representing a line number into a two-byte value for storage on disk.

Lines 1110–1120: Unlike the previous program, Merge, this one will delete the source file once it has successfully renumbered the file. During renumbering, the output file will be called 'TEMPORARY'. These lines open the error channel and ensure that any file with the same name is scratched.

Lines 1150–1160: These lines scratch the original file once renumbering is complete and rename the file 'TEMPORARY' with the same name as the original program file.

Lines 2000–2220: This module performs what is known as the 'first pass' through the program. During this process the whole of the file is examined but no changes are made. The purpose of pass 1 is to calculate the new line numbers which will eventually be inserted, and to check for the possibility of certain errors which would lead to the program being corrupted when new line numbers are actually written to the file.

Lines 2050–2100: The familiar lines which pick up the first two bytes of the file and discard them, then pick up, for each line in turn, the link bytes and line number bytes. The only difference is that the lines also inform the user which line is under consideration at the present time.

Lines 2110–2130: The line is scanned until the zero byte signifying the end is detected. In the process, a watch is kept for CHR$(143), the single character token for REM. If this is found and is followed by '#', it is taken as an indication that the line number of the next line should be increased to the

next highest whole thousand. You may have noticed, in the other programs contained within this manual, that new sections of programs are neatly renumbered to the next highest thousand. If you look closer you will also notice that the first REM of each section is followed by '#' — in other words the other programs in this manual were renumbered using this program.

CHAPTER 9
Relative Files

1) Introduction
2) Creating a relative file
3) OPENing a relative file
4) Specifying a position in a relative file
5) Writing to a relative file
6) Reading from a file
7) CLOSEing a relative file
8) Using relative files

1. Introduction

So far we have dealt only with data files which have to be read in a sequential manner, ie one item after another from the beginning of the file. While the speed of the disk drive does help to cut down the waste of time that this involves, the fact remains that time *is* wasted. Using a data file in this way is equivalent to using an array in BASIC but having to read every element from the beginning in order to access any specified element.

What we need, ideally, is a form of file on disk which acts in the same way that a BASIC array does; namely allows the user to specify a position in the file and retrieve information from that position directly. In fact such a type of file does exist and it is called a 'relative file'.

Having said that there is a type of file called a 'relative file', the statement immediately needs to be modified. Relative files are really a method of making two separate files work together. The first of the two files is the one in which the actual data is stored. The structure of this file on the disk is similar to that of a normal sequential file, though it does not appear as such in the disk directory. In addition to this main file, however, is another file which holds, for each item of data in the main file, the track and sector in which the item is stored.

The method of accessing a relative file, therefore, is to specify the number of the data item in the main file but to use the secondary 'pointer' file to discover where that item is on the disk and to read it directly from that position.

2. Creating a relative file

As opposed to sequential files, which can only be OPENed for writing once, a relative file may be OPENed time and time again, and each time it is OPENed it is ready for both reading from and writing to the disk. This is because, in the case of a relative file, the normal OPEN command does not *create* the file. To create a relative file, a special form of the OPEN command must be used, the format of which is as follows:

OPEN< FILENUM> ,< DEV> ,< CHANNEL> ,"< FILENAME> ,
L," + CHR$(< **RECLEN**>)

The only surprises here, compared to what we have done before, are :

1) **L** — the file type specifier for a relative file, which is the letter 'L'.

2) **RECLEN** — though a relative file is in many ways like a string array stored on disk rather than in memory, it differs from a BASIC string array in that every element of the file has a fixed length which must be specified when the file is first set up. RECLEN will be a value in the range 1–254 — the maximum length of one entry being one full sector on the disk, ie 256 bytes minus the two link bytes for the sector (see Chapter 1).

Note, in addition, that the '@0:' facility for overwriting an existing file does *not* function with a relative file.

3. OPENing a relative file

Provided that a relative file has previously been created using the form of the OPEN command shown in the previous section, it *must* subsequently be OPENed with an OPEN command in the following format:

OPEN< FILE NUMBER> ,< DEVICE> ,< CHANNEL> ,
"< FILENAME> "

Note that there is no need to include a specifier for the type of file, since the 1541 is capable of recognising a relative file when it encounters one.

4. Specifying a position in a relative file

The essence of a relative file is that the user is able to specify the position in the file to which an item is to be written or from which one is to be read. Before examining techniques of reading and writing to the file, it is therefore necessary first to explain the use of the position command, which

allows the number of the record within the file to be specified. The format of the position command is as follows:

PRINT#<**FILE NUMBER**>, "**P**" CHR$<**CHAN**> CHR$<**LO**> CHR$<**HI**> [CHR$<**POS**>]

1) **FILE NUMBER** — the first thing to realise is that the position command is not something which is printed to the relative file itself, but rather a command sent to the *error channel*. The file number specified is therefore the number of a file which has been previously OPENed to the error channel in the form OPEN FILE NUMBER, DEVICE, 15.

2) "**P**" — this specifier indicates to the error channel that what is being received is a position command.

3) **CHAN** — the number of the channel allocated to the relative file when it was last OPENed.

4) **LO and HI** — the two values LO and HI specify the record number in what is known as two-byte format. For any number (X), the value of these two specifiers can be calculated as:

HI = INT(X/256)

LO = X – 256*HI

5) **POS** — this optional specifier may be added to the command to set the pointer in the file to any character within the record indicated by LO and HI. In normal circumstances the records will be read from the beginning and the value of POS will be 1 — the position command in any case defaults to 1 if CHR$(POS) is omitted.

Note that, if a position command specifies a record number which is greater than the position of the current last record, the 1541 will immediately write sufficient blank records to the disk to bring the number of records available up to the number specified for the new position. This process can take some time and, rather than have it happen during the input of data it is often advisable to set up the file to the maximum size required when it is first created. This can be be done by specifying a position of, for example, 1000 immediately after the file is created. The result will be that 1000 empty records will be written to the disk and minimum delay will be experienced when subsequently accessing records between 1 and 1000.

5. Writing to a relative file

Having informed the 1541 of the position of the record you wish to write to in the file, it is now possible to write an item of data to that record. The format for the PRINT # command is:

PRINT #< **FILE NUMBER**> ,**VARIABLE LIST**

1) **FILE NUMBER** — The file number under which the relative file was last OPENed.

2) **VARIABLE LIST** — the data to be printed, either numeric or string, or both (see punctuation below).

Punctuation when writing to a relative file

When writing items to a file, some care must taken with the punctuation employed. As with a sequential file, the nature of the punctuation used in printing a series of variables to a record will affect the way in which the data is stored (see Chapter 7). Thus, printing items separated by commas to a file will result in spaces being placed in the file between the items. The reason that care must be taken is that the length of the record to which the items are being written is fixed. If careless punctuation leads to the length of the record being exceeded, the data will be truncated and a disk error message OVERFLOW IN RECORD generated, though the program will not stop.

Even when using only CHR$(13) to separate items, account has to be taken of the number of carriage return characters being used, since these too contribute to the length of the data being printed.

Maximum economy of space is achieved by running items together with the use of the semicolon separator or no separators at all, but this does imply that when reading the data back from the file the program must know exactly how long each item in a record is — the record itself will contain no indication of where each item begins and ends.

6. Reading from a file

Provided that the position command has been used to specify which record is to be read from, either INPUT # or GET # can be used to retrieve data from the file, the format for the commands being the same as for a sequential file. The form in which data is read will depend upon the punctuation used when storing it. If carriage returns have been placed between every item in a record, then INPUT # may be used to pick up each item in turn. The limitations on the use of INPUT # and GET # are the same as

when retrieving data from a sequential file (see Chapter 7).

In addition, it should be stressed once again that each record has a fixed length, and, once items have been read back to a total length equal to that of the record, a RECORD OVERFLOW disk error message will be generated in the error channel. Reading will *not* move on to the next record until a new position command is issued.

One extra facility which INPUT # and GET # may make use of when reading from a relative file is the ability of the position command to specify the *character* in the record from which reading or writing is to take place. Thus if a 100-character record were to contain 10 items (or fields), each of 9 characters in length and terminated by a carriage return, it would be possible to pick up any of the items individually using INPUT # provided that the character position was correctly specified in a position command.

7. CLOSEing a relative file

The format for the CLOSE command in relation to relative files is no different from that for sequential files, ie:

CLOSE< FILE NUMBER>

8. Using relative files

So far we have examined the commands necessary to handle relative files. We now turn to a practical example of their use in the form of a simple database program called Diskbase.

Diskbase program listing

```
10 REM DISC BASE PROGRAMME
20 DEV = 8
30 DIM F$(10),F%(10),DA$(10)
40 DEF FNHI(X) = INT(X/256)
50 DEF FNLO(X) = X-INT(X/256)*256
60 PAD$ = "                    " : PAD$ =
 PAD$+PAD$
70 PAD$ = PAD$+PAD$ : PAD$ = PAD$+PAD$
 : PAD$ = LEFT$(PAD$,127)+PAD$
80 GOSUB 9000
90 GOSUB 10000
100 GOTO 80
```

```
1000 REM#********************************
1010 REM READ NEW DEVICE NUMBER
1020 REM********************************
1030 PRINT "[CD]       NEW DEVICE NUMBER
 ?" DEV "[CL][CL][CL][CL]";
1040 IF DEV>9 THEN PRINT "[CL]" ;
1050 INPUT T
1060 IF T<4 OR T>31 THEN PRINT "[CU][
CU]" : GOTO 1000
1070 DEV = T
1080 RETURN
2000 REM#********************************
2010 REM INITIALISE FILE
2020 REM********************************
2030 INPUT "[CLR][CD]FILE NAME ";NA$
2040 IF LEN(NA$)>12 OR LEN(NA$)<1 THEN
 2030
2050 INPUT "[CD]NUMBER OF FIELDS (1-10)
 ";NF
2060 IF NF<1 OR NF>10 THEN PRINT "[CU]
[CU][CU]" : GOTO 2050
2070 FS = 0
2080 FOR I = 1 TO NF
2090 PRINT "[CD]NAME FOR FIELD" I ;
2100 INPUT F$(I-1)
2110 F$(I-1) = LEFT$(F$(I-1),30)
2120 PRINT "SIZE FOR FIELD" I;
2130 INPUT F%(I-1)
2140 FS = FS+F%(I-1)
2150 NEXT
2160 IF FS>254 THEN PRINT "TOO MANY
 FIELDS OR FIELDS TO BIG" : GOTO 2050
2170 PRINT "[CLR]FILE NAME - " NA$
2180 PRINT "[CD]FIELD NAME
       SIZE"
2190 FOR I = 0 TO NF-1
2200 PRINT "[CD]" F$(I) SPC(30-LEN(F$
(I))) F%(I)
2210 NEXT
2220 INPUT "[CD]IS THIS CORRECT (Y/N)
 ";T$
2230 IF T$="N" THEN 2000
2240 IF T$<>"Y" THEN PRINT "[CU][CU][CU]
[CU][CU]" : GOTO 2220
```

```
2250 IT = 0
2260 PRINT#15,"S0:"+NA$+".D
2270 GOSUB 4000
2280 OPEN 8,DEV,8,NA$+".D,L,"+CHR$(FS)
2290 PRINT#15,"P" CHR$(8) CHR$(100) CHR$
(0) CHR$(1)
2300 INPUT#15,A,B$,C,D
2310 CLOSE 8
2320 RETURN
3000 REM*********************************
3010 REM READ THE CONTROL FILE
3020 REM*********************************
3030 INPUT "[CD]          FILE NAME ";NA$
3040 IF LEN(NA$)>12 OR LEN(NA$)<1 THEN
 3030
3050 OPEN 8,DEV,8,NA$+".C,U"
3060 FS = 0
3070 INPUT#15,EN,EM$,ET,ES
3080 IF EN<19 THEN 3120
3090 PRINT "[CD]       DISC ERROR " EM$
3100 FOR I = 0 TO 2000 : NEXT : CLOSE 8
3110 GOTO 3170
3120 INPUT#8,IT,NF
3130 FOR I = 0 TO NF-1
3140 INPUT#8,F$(I),F%(I)
3150 FS = FS+F%(I)
3160 NEXT
3170 CLOSE 8
3180 RETURN
4000 REM*********************************
4010 REM REWRITE THE CONTROL FILE
4020 REM*********************************
4030 CLOSE 8
4040 OPEN 8,DEV,8,"@0:"+NA$+".C,U,W"
4050 PRINT#8,IT
4060 PRINT#8,NF
4070 FOR I = 0 TO NF-1
4080 PRINT#8,F$(I)
4090 PRINT#8,F%(I)
4100 NEXT
4110 CLOSE 8
4120 RETURN
5000 REM*********************************
5010 REM READ DATA FROM DATA BASE
```

```
5020 REM*********************************
5030 PRINT#15,"P" CHR$(8) CHR$(FNLO(T+1)
) CHR$(FNHI(T+1)) CHR$(1)
5040 FOR I = 0 TO NF-1
5050 DA$(I) = ""
5060 FOR J = 1 TO F%(I)
5070 GET#8,T$
5080 T1 = ASC(T$) AND 127
5090 IF T1<31 THEN T$ = ""
5100 DA$(I) = DA$(I)+LEFT$(T$+CHR$(0),1)
5110 NEXT J,I
5120 RETURN
6000 REM*********************************
6010 REM WRITE DATA TO DATA BASE
6020 REM*********************************
6030 PRINT#15,"P" CHR$(8) CHR$(FNLO(T+1)
) CHR$(FNHI(T+1)) CHR$(1)
6040 T$ = ""
6050 FOR I = 0 TO NF-1
6060 T$ = T$+LEFT$(DA$(I)+LEFT$(PAD$,F%
(I)-LEN(DA$(I))),F%(I))
6070 NEXT I
6080 PRINT#8,T$ ;
6090 RETURN
7000 REM*********************************
7010 REM ALTER CONTENTS OF DATA BASE
7020 REM*********************************
7030 PRINT "[CLR]" IT "ITEMS IN DATA
 FILE [RVS]" NA$ "[CD][CD][CD]"
7040 INPUT "RECORD NUMBER TO ALTER ? #
[CL][CL][CL]";T$
7050 T = VAL(T$) : IF T$="#" THEN T = IT
7060 IF T<0 OR T>IT THEN PRINT "[CU][CU]
" : GOTO 7040
7070 PRINT
7080 FOR I = 0 TO NF : DA$(I) = "" : NEXT
7090 IF IT<>T THEN GOSUB 5000
7100 FOR I = 0 TO NF-1
7110 PRINT F$(I) " ? " DA$(I)
7120 PRINT "[CU]" F$(I) " " ;
7130 INPUT T$
7140 IF LEN(T$)>F%(I) THEN PRINT "[CU]
[CU]" : GOTO 7110
7150 DA$(I) = T$
```

```
7160 NEXT I
7170 PRINT "[CD][CD]ADDING TO DATA BASE"
7180 GOSUB 6000
7190 IF T=IT THEN IT = IT+1
7200 INPUT "[CD][CD]MORE ITEMS (Y/N) ? Y
[CL][CL][CL]";T$
7210 IF T$="Y" THEN 7000
7220 IF T$<>"N" THEN PRINT "[CU][CU][CU]
[CU]" : GOTO 7200
7230 RETURN
8000 REM#################################
8010 REM LIST DATA ITEMS
8020 REM#################################
8030 LN = 0
8040 PRINT "[CLR]" IT "ITEMS IN DATA
 FILE [RVS]" NA$ "[CD][CD][CD]"
8050 PRINT "RECORD NUMBER TO LIST ?" LN
8060 IFIT=0THEN PRINT "[CD][CD][CD][CD]
[CD] NO ITEMS IN FILE" :FOR I = 0TO2000
 : NEXT : RETURN
8070 INPUT "[CU]RECORD NUMBER TO LIST ";T
8080 IF T<0 OR T>=IT THEN 8040
8090 PRINT
8100 FOR I = 0 TO NF : DA$(I) = "" : NEXT
8110 GOSUB 5000
8120 FOR I = 0 TO NF-1
8130 PRINT F$(I) " = " DA$(I)
8140 NEXT I
8150 LN = LN+1 : IF LN>=IT THEN LN=0
8160 INPUT "[CD][CD]MORE ITEMS (Y/N) ? Y[CL]
[CL][CL]";T$
8170 IF T$="Y" THEN 8040
8180 IF T$<>"N" THEN PRINT "[CU][CU][CU]
[CU]" : GOTO 8160
8190 RETURN
9000 REM#################################
9010 REM OPENING MENU
9020 REM#################################
9030 OPEN 15,DEV,15
9040 PRINT "[CLR]"
9050 PRINT SPC(12) "1541 DISC BASE"
9060 PRINT SPC(13) "[CD]OPENING MENU"
9070 PRINT SPC(8) "[CD][CD][CD][CD]1)
 CREATE NEW DATA FILE"
```

```
9080 PRINT SPC(8) "[CD]2) OPEN EXISTING
 DATA FILE"
9090 PRINT SPC(8) "[CD]3) CHANGE DEVICE
 NUMBER"
9100 PRINT SPC(8) "[CD]4) EXIT TO BASIC
[CD][CD][CD][CD]"
9110 INPUT "       COMMAND (1-4) ? 2[CL
][CL][CL]";T$
9120 CO = VAL(T$)
9130 IF CO<1 OR CO>4 THEN PRINT "[CU]
[CU]" : GOTO 9110
9140 ON CO GOSUB 2000,3000,1000,9180
9150 CLOSE 15
9160 IF CO=3 OR EN>19 THEN 9000
9170 RETURN
9180 PRINT "[CLR]"
9190 CLOSE 15
9200 END
10000 REM*********************************
10010 REM MAIN MENU
10020 REM*********************************
10030 OPEN 15,DEV,15
10040 OPEN 8,DEV,8,NA$+".D"
10050 PRINT "[CLR]"
10060 PRINT SPC(12) "1541 DISC BASE"
10070 PRINT SPC(15) "[CD]MAIN MENU"
10080 PRINT SPC(8) "[CD][CD][CD][CD]1)
 ADD/EDIT DATA ITEMS"
10090 PRINT SPC(8) "[CD]2) LIST EXISTING
 DATA ITEMS"
10100 PRINT SPC(8) "[CD]3) RETURN TO
 OPENING MENU[CD][CD][CD][CD]"
10110 INPUT "       COMMAND (1-3) ? 2[CL]
[CL][CL]";T$
10120 T =  VAL(T$)
10130 IF T<1 OR T>3 THEN PRINT "[CU][CU]
" : GOTO 10110
10140 IF T=3 THEN 10180
10150 ON T GOSUB 7000,8000
10160 CLOSE 8
10170 GOTO 10040
10180 CLOSE 8
10190 CLOSE 15
10200 RETURN
```

Explanation of Diskbase

Lines 9000–9200: The opening menu of the program.

Lines 10000–10200:The main menu encountered once the user has opened a file.

Lines 2000–2320: This section allows the user to specify the shape of the file to be created, ie the number of fields which each record will contain, their names and their length. The only interesting part from the point of view of relative files themselves is lines 2260–2320, which scratch any existing file of the same name, open the relative file, in line 2280, and issue a position command for record 100, ie one hundred blank records are written to the disk.

Lines 4000–4120: A second file, this time a user file, is employed to store the details of the field names, sizes and so forth. When the main data file which has been created is accessed in future, the program will first of all read the details of fields from the user file so that it can allocate the correct names and lengths to the fields within each record.

Lines 3000–3180: As mentioned under the previous section, when the program accesses an existing data file, it reads back the field names and sizes from a user file before beginning to recall data. This is the first program section called if the user specifies that an existing data file is to be read. The file name is specified by the user and the information read from the user file in lines 3120–3170. The user file itself will have the same name as the data file except that its name will terminate in '. c', standing for 'control', rather than '. d', standing for 'data'. Thus if the user specifies a file name of 'DATASTORE', the control file will be called DATASTORE. C and the data file DATASTORE. D.

Lines 5000–5120: This program section reads a single record from the data file. The number of the record to be read is specified elsewhere in the program and is contained in the variable T. The two functions, FNLO and FNHI, which were set up at the very beginning of the program, calculate the low and high byte of the two-byte number representing the number of the record. The lines from 5060 to 5110 use GET # to read the items back one by one from the record, including checks that control or null characters on the disk do not disrupt the working of the program. As each item is retrieved from the record it is stored in the array DA$.

Lines 8000–8190: These lines allow the user to specify the number of a record to be recalled and then call up the module at line 5000 to retrieve the data.

Lines 6000–6090: These lines issue a position command for a record specified by the user and then write to that record the items which have been entered by the module at line 7010.

Lines 7000–7230: This section allows the user to specify a record number. The data currently held at that point is first retreived and the user then has the option to re-enter the existing items or to change them. The items finally entered by the user are then written to the file using the module at line 6000.

The program is not immensely sophisticated but is a sound basic example of what can be achieved with a relative file in terms of speed of access for both reading and writing data. It would make a good basis for further developments in the use of such files.

CHAPTER 10
Random Files

1) OPENing a random file
2) Position of data in the buffer
3) Writing data to the buffer
4) Writing data to disk
5) Loading data from the disk to the buffer
6) Getting the data back into the 64
7) Marking and freeing sectors on the disk
8) Executing machine code from the disk
9) Two utility programs using random files

So far, we have dealt only with files where the main work in reading and writing is done by the sophisticated and massive Disk Operating System program. We have specified the data to be stored, and perhaps even the position which it is to occupy in a file, but it is the DOS which has managed the disk space, searching out the best storage space and recording the complex pattern of sectors which go to make up the file in the Block Allocation Map. There is, however, one type of file available which allows the user to bypass at least some of the DOS functions and to decide where and how data is to be stored on the disk. Such files are known as 'random files'.

Random files are not included amongst the facilities available on the 1541 for any particular reason, it is simply that they are the basic file type which is used by the DOS in creating the files which we have examined so far. As a type, random files are probably the least useful form of file but, even so, simply because they are so crude in their structure and management they can often find applications where the user needs to access the disk in unexpected ways.

It is possible, using random files, to simulate any of the file types that we have considered. The end result of such efforts, however, is hardly likely to be as polished as what Commodore have already provided in the DOS, so the considerable programming involved would hardly be worth the effort. In this chapter we examine only the basics of working with a random file but we do show how the commands used to control such files can be put to one or two unusual uses.

1: OPENing a random file

Using random access has two aspects, the communication of data and the issuing of instructions. Data sent to a random file is not automatically written on to the disk, for instance, until a separate command is issued through the error channel specifying where and how the data is to be stored. When a random file is OPENed, therefore, what is being opened is not a direct path from the 64 to the disk itself but a path from the 64 to an area of memory within the disk drive known as a buffer, capable of holding up to 256 bytes of data. Since the disk drive has more than one buffer available for this purpose, the OPEN command for a random file needs to specify not only the file number, device and channel to be used, but which buffer is to be used.

The format of the OPEN command for random files is:

OPEN< FILE NUMBER> ,< DEVICE> ,< CHANNEL> ,"#
[< **BUFFER**>]"

1) # — the '#' symbol replacing the more usual filename at this point is the indication to the disk drive that the file to be OPENed is a random file.

2) **BUFFER** — the value BUFFER can be used to specify which of the disk drive's six buffers is to be associated with the file number. The numbers of the buffers are 0 to 5, but of these at least two are likely to be in use by the disk drive at any one time, even before you open files which will use up further buffers. Because of the uncertainty as to which buffers are available at any one time, the optional BUFFER value is hardly ever used. When it is omitted, the disk drive allocates a free buffer (if one is available) to the file. It is of no relevance to the programmer which buffer is used, unless a particular buffer is going to be used for the execution of machine code programs in the DOS RAM.

2. Position of data in the buffer

Having allocated a buffer to the file, the next task (whether the intention is to write data from the 64 to the buffer or to read data from the buffer to the 64) is to inform the drive at what point in the buffer to set a special pointer known, unsurprisingly, as the buffer pointer. Subsequent calls to read from or write to the buffer will be executed from the position indicated by the buffer pointer and not necessarily from the start of the buffer.

The format for the buffer pointer command is:

PRINT #< **FILE NUMBER**> ,"**B-P:**"< **CHANNEL**> ,< **CHAR**>

1) **FILE NUMBER** — like all the random access commands, this is an instruction sent along the error channel, so the file number is that of a previously opened file to channel 15.

2) **B-P:** — this is the abbreviated form of the command 'BUFFER-POINTER:' — the full form is accepted but is a little unwieldy.

3) **CHANNEL** — the CHANNEL specified will be the one previously allocated in an OPEN command to the random file to be acted upon.

4) **CHAR** — the number of the byte within the buffer from which a subsequent read or write process will take place; will be in the range 0–255.

3. Writing data to the buffer

To write data to the buffer, the ordinary PRINT # command is used in the format:

PRINT #< FILE NUMBER> ,VARIABLE LIST

where FILE NUMBER is the number allocated to the random file when it was OPENed.

Since the buffer is strictly limited to 256 bytes (0–255), random files are like relative files in that, if more data is written than can be held in the buffer, the extra data will be lost.

4. Writing data to disk

Data written into the buffer will be lost when the disk drive is switched off unless something further is done with it. Accordingly there are two commands which can be used to write the contents of a buffer to the disk. The first command, 'BLOCK-WRITE:', is used to write the part of the buffer's contents between byte 0 and the position of the buffer pointer to the disk. (*Note:* Although only part of the *buffer* may be written, the whole of the sector into which the data is placed is changed). The second command is called 'U2:' and is used to write the *whole* of the contents of the buffer to a specified track and sector. Since there is seldom much need to write only a part of the buffer to the disk, the 'U2:' form of the command is the one more commonly used for random files.

The format of the two commands is:

PRINT #< **FILE NUMBER**> ,"**B-W:**"< **CHANNEL**> ,< **DRIVE**> ,
< **TRACK**> ,< **SECTOR**>

or

PRINT #< FILE NUMBER> ,"**U2:**"< CHANNEL> ,< DRIVE> ,
< TRACK> ,< SECTOR>

1) **FILE NUMBER** — the number of the file previously opened to the error channel.

2) **"B-W:"** — the full form 'BLOCK-WRITE:' is also accepted.

3) **CHANNEL** — the number of the channel allocated to the random file in the OPEN command.

4) **DRIVE** — always zero when using the 1541.

5) **ILLEGAL TRACK AND SECTOR** error message — generated in the error channel if a non-existent part of the disk is specified.

6) **"U2:"** — an alternative form, 'UB:', is also acceptable.

Note: Since the safeguards in the DOS are being bypassed by the use of random files, there is no protection against overwriting important data, including the directory, with these commands.

5. Loading data from the disk to the buffer

Parallel to the two write commands in the last section are the two commands 'B-R:' and 'U1:', which allow a track and sector to be specified and read from the disk *into* the buffer. The format of the two commands is a mirror image of the format of the two write commands, ie:

PRINT #< FILE NUMBER> ,"**B-R:**"< CHANNEL> ,< DRIVE> ,
< TRACK> ,< SECTOR>

or

PRINT #< FILE NUMBER> ,"**U1:**"< CHANNEL> ,< DRIVE> ,
< TRACK> ,< SECTOR>

1) **"B-R:"** — the full form 'BLOCK-READ:' is acceptable.

2) "**U1:**" — the alternative form 'UA:' is acceptable.

6. Getting the data back into the 64

Once the required data has been retrieved from the disk and placed in the buffer, it is a simple matter to load it back into the 64 by means of the command INPUT # and GET #. The buffer pointer command can be used to determine where in the buffer the GET # or INPUT # will begin, but other than that the process is no different from that for any other type of file.

The format for INPUT # and GET # will be:

INPUT #< FILE NUMBER> ,VARIABLE LIST

or

GET #< FILE NUMBER> , VARIABLE LIST

where FILE NUMBER is the number allocated to the random file.

7. Marking and freeing sectors on the disk

One problem remains to be solved. When data is written to the disk in the form of a random file, the use of the sectors for the purposes of the file is not recorded in the Block Allocation Map. There is, therefore, a danger that the DOS, working on the basis of the BAM, will simply overwrite the sectors allocated to the random file. The solution to this lies in the command 'BLOCK-ALLOCATE:' (and its opposite 'BLOCK-FREE:').

The function of 'B-A:' is to register in the BAM the fact that a sector is in use. Once registered, the DOS will not overwrite it with any incoming material. 'B-F:' performs the opposite function, of registering in the BAM that a sector which was previously marked as being in use is now free. The format for the two commands is:

PRINT #< FILE NUMBER> ,"B-A:"< DRIVE> ,< TRACK> ,
< SECTOR>

and

PRINT #< FILE NUMBER> ,"B-F:"< DRIVE> ,< TRACK> ,
< SECTOR>

In fact, 'BLOCK-ALLOCATE:' is normally used *before* a sector is written to. Printing the command to the error channel results in the error message

NO BLOCK coming back down the error channel if the sector is already allocated to a file. Along with the error message text, the track and sector figures represent the next highest sector which *is* free — no scan is done for free space in tracks and sectors with numbers lower than the one you specified. It is for this reason that random files often start on track 1, sector 0, so that they can use the facilities afforded by 'BLOCK-ALLOCATE:' to the full.

8. Executing machine code from the disk

One further block command remains, though seldom used. This is 'BLOCK-EXECUTE:', and it has the format:

PRINT #< FILE NUMBER> ,"B-E:"< DRIVE> ,< TRACK> ,
< SECTOR>

Its effect is to load the specified sector into the buffer and then to run it as a machine code program within the disk drive, ie the program affects the 6502 chip which controls the 1541, not the 6510 chip which is the CPU of the 64. Running the disk drive from a machine code program in this way is a task which should be approached with caution since loss of data or damage to the mechanism itself can result from improper control of the drive.

9. Two utility programs using random files

In this section, we present two interesting disk utilities which make use of the power of random files. The first of the two programs is called UN-SCRATCH and its purpose is to reverse the effects of the SCRATCH command. The sector program LIST TRACK AND SECTOR, lists out the contents of a disk on the screen, giving the allocation of every sector, track by track.

Listing of program UNSCRATCH

```
10 GOTO 27000
27000 REM#********************************
27010 REM UNSCRATCH DISC FILE
27020 REM********************************
27030 INPUT "DEVICE NUMBER ? 8[CL][CL]
[CL]";DEV
27040 OPEN 8,DEV,8,"#"
27050 OPEN 15,DEV,15
```

```
27060 N$ = CHR$(0)
27070 S$ = CHR$(160)+CHR$(160)
27080 S$ = S$+S$ : S$ = S$+S$
27090 S$ = S$+S$
27100 INPUT "SCRATCHED FILE NAME ";NA$
27110 INPUT "SCRATCHED FILE TYPE ";TY$
27120 GOSUB 30000
27130 IF CO THEN 27150
27140 PRINT "FILE " NA$ " IS NOT RECOVE
RABLE" : GOTO 27180
27150 IF T>128 THEN 27180
27160 PRINT "FILE CAN BE RECOVERED BUT
 FILE TYPE "
27170 PRINT TY$ " IS UNAVAILABLE"
27180 PRINT#15,"V0"
27190 CLOSE 8 : CLOSE 15
27200 END
28000 REM*********************************
28010 REM SEARCH FOR SCRATCHED FILE
28020 REM*********************************
28030 CO = 0
28040 NT = 18 : NS = 0
28050 GOSUB 29000
28060 FI = 0
28070 PRINT#15,"B-P:"8,FI*32+2
28080 GET#8,T$
28090 T = ASC(T$+N$) AND 127
28100 IF T<>0 THEN 28170
28110 GET#8,T$ : GET#8,T$
28120 T1$ = ""
28130 FOR I = 0 TO 15
28140 GET#8,T$ : T1$ = T1$+LEFT$(T$+N$,1)
28150 NEXT
28160 IF LEFT$(NA$+S$,16)=T1$ THEN CO=-1
 : GOTO 28190
28170 IF FI<7 THEN FI = FI+1 : GOTO 28070
28180 IF NT<>0 THEN 28050
28190 RETURN
29000 REM*********************************
29010 REM READ NEXT TRACK AND SECTOR
29020 REM*********************************
29030 TR = NT : S = NS
29040 PRINT#15,"U1:"8;0;TR;S
29050 PRINT#15,"B-P:"8;0
```

```
29060 GET#8,T$ : NT = ASC(T$+N$)
29070 GET#8,T$ : NS = ASC(T$+N$)
29080 RETURN
30000 REM*********************************
30010 REM DO UNSCRATCH
30020 REM*********************************
30030 GOSUB 28000
30040 IF NOT CO THEN 30140
30050 PRINT#15,"B-P:"8;FI*32+2
30060 T = 0
30070 IF TY$="SEQ" THEN T=1
30080 IF TY$="PRG" THEN T=2
30090 IF TY$="USR" THEN T=3
30100 IF TY$="REL" THEN T=4
30110 T = T+128
30120 PRINT#8,CHR$(T) ;
30130 PRINT#15,"U2:"8;0;TR;S
30140 RETURN
```

Explanation of UNSCRATCH program

Lines 27000–27200: This section allows the user to specify the name of the file which has been scratched and the file type to be allocated to it when it is reinstated. At the end of the module the VALIDATE command is used, after the rest of the program has reinstated the file in the directory, to update the BAM, which has registered the sectors once used by the file as free. *Note:* This program cannot reinstate a file once other material has been written to the disk on the sectors which were originally used by the file which was SCRATCHed, or if the directory entry has been overwritten.

Lines 29000–29080: These lines pick up the current sector from the directory track and store the pointers to the next track and sector in the variables NT and NS. Each time this module is called, the next sector will be picked up into the buffer using 'U1:' and the track and sector pointers updated to the following sector.

Lines 28000–28190: The function of this module is to search through the directory track for deleted files and to compare the names of any found with the name of the file to be UNSCRATCHed. This is done by reading a sector of the directory into the buffer using the module at line 29000, then moving the buffer pointer through the buffer in 32-byte steps — a single

entry in the directory is 30 bytes long with two spare bytes between each entry. At each step, the first byte of the entry is tested to see whether it is zero or 128, the sign of a SCRATCHed file. If a scratched file *is* found, its name is extracted from the buffer and compared with the name of the specified program. If they match, the variable CO is set to minus one; if not, the search through the directory continues until the end of directory marker is found.

Lines 30000–30140: If the flag CO indicates that the correct file has been found, then the number corresponding to the file type under which it is to be UNSCRATCHed is printed to the first character of the directory entry in the buffer and the buffer is then rewritten to the disk, effectively changing the status of the file in the directory. If the file type specified is invalid, the program notifies the user of this fact and gives the file the type 'DEL', or 'deleted'. It will appear in the directory and can be unscratched under a valid file type.

Listing of program LIST TRACK AND SECTORS

```
21000 REM#*******************************
21010 REM LIST TRACK AND SECTOR ON DISC
21020 REM#*******************************
21030 DIM D$(35,20),DI$(144),TY$(3)
21040 TY$(0) = "SEQ"
21050 TY$(1) = "PRG"
21060 TY$(2) = "USR"
21070 TY$(3) = "REL"
21080 INPUT "DEVICE NUMBER ? 8[CL][CL]
[CL]";DEV
21090 OPEN 15,DEV,15
21100 OPEN 8,DEV,8,"#"
21110 PRINT "[CLR][CD][CD][CD][CD][CD]
     READING DISC"
21120 GOSUB 24000
21130 CLOSE 8 : CLOSE 15
21140 GOSUB 26000
21150 END
22000 REM#*******************************
22010 DATA "FOR TRACK AND SECTOR SIZE"
22020 REM#*******************************
22030 DATA 1,17,20,18,24,18,25,30,17,31
```

```
,35,16
23000 REM********************************
23010 REM READ FILE STARTING AT TR & S
23020 REM********************************
23030 BL = 1
23040 D$(TR,S) = N$+" BLOCK"+STR$(BL)
23050 BL = BL+1
23060 PRINT#15,"U1:"8;0;TR;S
23070 PRINT#15,"B-P:"8;0
23080 GET#8,T$ : TR = ASC(T$+CHR$(0))
23090 GET#8,T$ : S = ASC(T$+CHR$(0))
23100 IF TR>0 THEN 23040
23110 RETURN
24000 REM********************************
24010 REM READ ALL OF DISC
24020 REM********************************
24030 RESTORE
24040 READ T$ : IF T$<>"FOR TRACK AND
 SECTOR SIZE" THEN 21070
24050 FOR I = 1 TO 4
24060 READ T1,T2,S1
24070 FOR TR = T1 TO T2
24080 FOR S = 0 TO S1
24090 D$(TR,S) = "UNUSED"
24100 NEXT S,TR,I
24110 REM --------- READ BAM ----------
24120 PRINT#15,"U1:"8;0;18;0
24130 PRINT#15,"B-P:"8;4
24140 T1$ = ""
24150 FOR I = 0 TO 143 : GET#8,T$ :
 T1$ = T1$+LEFT$(T$+CHR$(0),1) : NEXT
24160 T = 0
24170 RESTORE
24180 READ T$ : IF T$<>"FOR TRACK AND
 SECTOR SIZE" THEN 24180
24190 FOR I= 1 TO 4
24200 READ T1,T2,S1
24210 FOR TR = T1 TO T2
24220 FOR S = 0 TO S1
24230 T = TR*32+S-24
24240 T3 = INT(T/8)+1
24250 T4 = 2^(T-(T3-1)*8)
24260 T = ASC(MID$(T1$,T3,1))
24270 IF (T4 AND T) = 0 THEN D$(TR,S)
```

```
 = "RANDOM FILE"
24280 NEXT S,TR,I
24290 REM ----- READ DIRECTORY ----------
24300 N$ = "DIRECTORY"
24310 TR = 18 : S = 0
24320 GOSUB 23000
24330 REM -- READ FILES ON DIRECTORY ---
24340 GOSUB 25000
24350 IF DP<1 THEN RETURN
24360 FOR I = 1 TO DP
24370 T = ASC(DI$(I))
24380 IF T<129 OR T>132 THEN 24430
24390 TR = ASC(MID$(DI$(I),2,1))
24400 S  = ASC(MID$(DI$(I),3,1))
24410 N$ = MID$(DI$(I),4,16)+","+TY$(T
-129)
24420 GOSUB 23000
24430 NEXT I
24440 RETURN
25000 REM#################################
25010 REM READ DIR. INTO DI$
25020 REM#################################
25030 DP = -1 : NT = 18 : NS = 0
25040 TR = NT : S = NS
25050 PRINT#15,"U1:" 8;0;TR;S
25060 PRINT#15,"B-P:" 8;0
25070 GET#8,T$ : NT = ASC(T$+CHR$(0))
25080 GET#8,T$ : NS = ASC(T$+CHR$(0))
25090 IF TR=18 AND S=0 THEN 25040
25100 PRINT#15,"B-P:" 8;0
25110 FOR I = 0 TO 7
25120 GET#8,T$ : GET#8,T$
25130 DP = DP+1
25140 DI$(DP) = ""
25150 FOR J = 0 TO 29
25160 GET#8,T$
25170 DI$(DP) = DI$(DP)+LEFT$(T$+CHR$
(0),1)
25180 NEXT J,I
25190 IF TR>0 THEN 25040
25200 RETURN
26000 REM#################################
26010 REM DISPLAY TRACK AND SECTOR
26020 REM#################################
```

```
26030 TR = 1 : S = 0
26040 PRINT "[CLR]"
26050 PRINT "[HOME]TRACK =    [CL][CL]
[CL][CL]"TR"  "
26060 RESTORE
26070 READ T$ : IF T$<>"FOR TRACK AND
 SECTOR SIZE" THEN 26070
26080 READ T1,T2,S1
26090 IF TR>T2 THEN 26080
26100 FOR S = 0 TO S1
26110 PRINT LEFT$(STR$(S)+"
           ",8) ;
26120 PRINT LEFT$(D$(TR,S)+"
                                        ",31)
26125 REM 40 SPACES IN PRECEDING LINE
26130 NEXT
26140 FOR T = S1 TO 20
26150 PRINT "
           " : REM 38 SPACES
26160 NEXT T
26170 PRINT "F1 = NEXT,F3 = LAST,F5 =
 NEW,F7 = EXIT";
26180 GET T$
26190 T = ASC(T$+CHR$(0))-132
26200 IF T<1 OR T>4 THEN 26180
26210 ON T GOTO 26220,26260,26290,26410
26220 TR = TR+1
26230 IF TR>35 THEN TR = 1
26240 PRINT "[HOME][CD][CD][CD][CD][CD]
[CD][CD][CD][CD][CD][CD][CD][CD][CD]
[CD][CD][CD][CD][CD][CD][CD][CD][CD
                                    "
26250 GOTO 26050
26260 TR = TR-1
26270 IF TR<1 THEN TR = 35
26280 GOTO 26240
26290 PRINT "[HOME]TRACK =
                    [HOME]TRACK =  " ;
26300 T1$ = ""
26310 GET T$ : IF T$<>CHR$(20) AND T$<>
CHR$(13) AND (T$<"0" OR T$>"9") THEN 26310
26320 IF T$<>CHR$(13) THEN 26360
26330 T=VAL(T1$)
26340 IF T>=1 AND T<=35 THEN TR = T :
```

```
 GOTO 26240
26350 GOTO 26290
26360 IF T$=CHR$(20) AND T1$<>"" THEN T1
$ = LEFT$(T1$,LEN(T1$)-1) : GOTO 26390
26370 IF LEN(T1$)>=2 THEN 26310
26380 T1$ = T1$+T$
26390 PRINT "[CL]" T$ " ";
26400 GOTO 26310
26410 PRINT "[CLR]"
26420 RETURN
```

Explanation of LIST TRACK AND SECTOR

Lines 21000–21150: This module sets up the arrays the program will use to store the contents of each sector on the disk and the entries in the directory, then sets up the file to the error channel and a random file to the specified device.

Lines 23000–23110: This module is sent two items of information each time it is called — the name of a file and the address on the disk of the first sector of that file. The purpose of the module is to use the track and sector pointers at the beginning of each sector to trace through the file on the disk. As it does so, it marks the corresponding element in the array D$ with the name of the file to which the sector is allocated and the number of the sector within the file.

Lines 24000–24100: Based on the data statement at line 22030, the array D$ is filled with the word 'UNUSED' in positions corresponding to the track and sector numbers which exist on the disk. Thus, D$(1,0) to D$(1,20) are initially marked unused since there are 21 sectors in track 1. Lines of D$ which correspond to tracks further in will have fewer elements used.

Lines 24110–24280: Having marked all possible sectors as unused in the array, the second section of the module reads track 18, sector 0, into the buffer and then copies the BAM into a string variable, T1$. Based on the track and sector figures from line 22030, lines 24210–24280 examine every bit of the bytes within T1$ which record the allocation of sectors. For every bit which is reset (zero), the mark of an allocated sector, the appropriate element of the array D$ is changed to read RANDOM FILE. Later parts of the program will rewrite most of these entries on the basis of information contained in the directory but any entry which the BAM showed to be allocated and which does not belong to a directory file will remain flagged as part of a random file.

Lines 24290–24320: These lines call up the module at line 23000 to trace through the sectors which make up the directory and mark the position of each in the array D$.

Lines 24330–24440: After calling up the module at line 25000, which reads every directory entry into the array DI$, these lines send the information about the start of each file to the module at line 23000, which traces the file through, sector by sector, again marking the name of the file in the relevant element of the array D$.

Lines 25000–25200: This module reads the contents of the directory into the array DI$. For each entry, the two spare bytes are read and discarded (line 25120) and then 30 bytes, the length of a directory entry, are read into DI$: then the counter DP is incremented. The process continues until the track pointer at the beginning of a directory sector is found to be zero, indicating the end of the directory. A fuller explanation is contained in Chapter 11.

Lines 26000–26420: This is the display module which lists the contents of a single track to the screen and allows the user to specify a track for display.

Note: Due to the extensive use of string arrays, the program pauses noticeably from time to time while the string tidying function known as 'garbage collection' is carried out by the 64.

CHAPTER 11
The Disk Directory

1) The format of the directory
2) Reading the directory
3) Repeating a process on multiple files

In the first chapter of this book, we included a brief description of the disk directory. Since then we have taken the function of the directory, in allowing the user to examine the contents of a disk and in allowing the Disk Operating System to find specified files on the disk, rather for granted. In this chapter we shall take a brief look at the directory, its layout and the way in which it may be directly accessed by the user.

1. The format of the directory

In Table 1.2 (page 4) the overall layout of the tracks which make up the directory is given. Examining the table shows that the directory is held on track 18 of the disk, beginning at sector 0. The first sector of the directory is given over to the Block Allocation Map, but the remainder of track 18 is reserved for the details of individual files on the disk. The second section of Table 1. 2 shows how each of these sectors is capable of holding the details of eight files. Given that there are 17 sectors on track 18 of the disk, simple arithmetic shows that the maximum number of files which the disk can hold, regardless of how much space is free, is 16*8, or 144.

Within the overall structure of the directory, the format of the entry for a single file is given in **Table 11.1**.

In fact, most of this table will be familiar from previous chapters. The file types, stored in byte 0 of the entry, we have made use of in the UNSCRATCH program in Chapter 10, where file types were altered to reinstate files which were registered in the directory as having been deleted.

The first track and sector bytes, and the filename itself, were used by the LIST TRACK AND SECTOR program to trace through the sectors allocated to each particular file and then to display the name of the relevant file against each sector on the disk. In normal use, the purpose of these bytes is

to allow the DOS to search through the directory for a specified filename and then to find the beginning of a file which it has been instructed to access.

Table 11. 1: Format of a Single Directory Entry

BYTE	REMARK
0	Type of file in use 0 = Unused or DELeted file 1 = Unclosed SEQuential file 2 = Unclosed PRoGram file 3 = Unclosed USeR file 4 = Unclosed RELative file 128 = Closed DELeted file 129 = Closed SEQuential file 130 = Closed PRoGram file 131 = Closed USeR file 132 = Closed RELative file
1	Track of first block in file
2	Sector of first block in file
3–18	File name padded with shifted spaces (CHR$(160))
19	Relative files — Track of first side sector of file Other file types — Not used
20	Relative files — Sector of first side sector of file Other file types — Not used
21	Relative files — Length of record Other file types — Not used
22–25	Not used
26–27	Only used when disk is SAVEing or OPENing a file with '@0:'
28–29	Number of blocks in this file

In Chapter 9 we saw how relative files are in fact made up of two quite separate sections, one containing the data and the other recording where the sectors holding the data are on the disk. The table shows that the start address of this second part of a relative file is held in bytes 19 and 20, while the fixed length of each record in a relative file is held in byte 21.

Bytes 26 and 27 are new to us, but their use is quite simple. When a file is SAVEd or OPENed using the '@0:' modifier to specify that any previous

file of the same name and type is to be overwritten, these bytes serve the purpose of holding the starting track and sector until the new file has been created.

Finally, when the directory is displayed for the user, the size of each file in terms of the sectors used is given with it, and this figure is stored in bytes 28 and 29 of the file entry.

In all, each individual file entry in the directory takes up 30 bytes (0–29). In order to space the eight possible entries regularly within the 256 bytes of the sector, two extra bytes are added to the end of the first seven entries. These bytes contain no useful information, their purpose is solely to allow the DOS to scan along the directory in steps of 32 bytes.

2. Reading the directory

There are two main ways in which the directory may be read:

1) By loading it into memory with the command LOAD "$",< DEV>, where DEV is the device number of the particular drive. When loaded in this way, the directory is treated in much the same manner as a program file, and any program presently in memory is lost. Loading is possible because the '$' indicates to the DOS that it has to translate the directory as it is on the disk into program file format, treating each entry as if it were a program line, supplying the zero bytes to finish lines and space for link bytes. In other words, the format supplied to the 64 when the LOAD "$" command is entered is entirely different to the format of the directory on the disk itself.

2) By reading the directory from the disk under program control. The 'DOS support' software provided free with later 1541s provides a neat means to accomplish this and print the contents of the directory to the screen without interference to the current program (see Appendix C). It is, however, quite possible to read the directory from BASIC and an example of this is shown in the LIST TRACK AND SECTOR program in Chapter 10. Given below are two short programs which will load the contents of the directory into an array, the first by reading the directory file much as a program file would be read (see Chapter 8), and the second reading the disk more directly.

Program to read the directory into an array from file '$'

```
10 DIM DI$(150)   : DEV = 8
20 GOSUB 1000
30 FOR I = 0 TO DP-1
40 PRINT CHR$(34) DI$(I)
50 NEXT
```

```
60 END
1000 REM**********************************
1010 REM READ 1541 DIRECTORY
1020 REM**********************************
1030 DP = 0
1040 OPEN 8,DEV,0,"$"
1050 GET#8,T$ : GET#8,T$
1060 GET#8,T$ : GET#8,T$ : IF T$="" 
THEN CLOSE 8 : RETURN
1070 GET#8,T$ : GET#8,T$
1080 GET#8,T$
1090 IF T$<>CHR$(34) AND T$<>"" THEN
 1080
1100 IF T$="" THEN 1060
1110 T1$ = ""
1120 GET#8,T$
1130 IF T$<>"" THEN T1$ = T1$+T$ :
GOTO 1120
1140 DI$(DP) = T1$ : DP = DP+1
1150 GOTO 1060
```

Explanation of reading into array from file '$' program

If you have read Chapter 8, on the use of program files, you should have little difficulty in recognising the techniques being used here. The DOS supplies the directory in the form of a program file, with every filename built into a separate line and the whole thing properly structured with link bytes and so forth. There is no point in trying to compare what is being read by the GET # statements with the contents of the table at the beginning of the chapter, since there is almost no relation between the two. What is being read here is not the directory itself but the translated version of the directory supplied by the DOS.

Program to read directory directly from disk into an array

```
10 OPEN 15,8,15 : OPEN 8,8,8,"#"
20 DIM DI$(150)
30 GOSUB 1000
40 FOR I = 0 TO DP
50 T = ASC(DI$(I)) : IF T>=129 AND T
<=130 THEN PRINT MID$(DI$(I),4,16)
```

```
60 NEXT
70 CLOSE 8 : CLOSE 15
80 END
1000 REM*********************************
1010 REM READ DIR. INTO DI$
1020 REM*********************************
1030 DP = -1 : NT = 18 : NS = 0
1040 TR = NT : S = NS
1050 PRINT#15,"U1:" 8;0;TR;S
1060 PRINT#15,"B-P:" 8;0
1070 GET#8,T$ : NT = ASC(T$+CHR$(0))
1080 GET#8,T$ : NS = ASC(T$+CHR$(0))
1090 IF TR=18 AND S=0 THEN 1040
1100 PRINT#15,"B-P:" 8;0
1110 FOR I = 0 TO 7
1120 GET#8,T$ : GET#8,T$
1130 DP = DP+1
1140 DI$(DP) = ""
1150 FOR J = 0 TO 29
1160 GET#8,T$
1170 DI$(DP) = DI$(DP)+LEFT$(T$+CHR$
(0),1)
1180 NEXT J,I
1190 IF NT>0 THEN 1040
1200 RETURN
```

Explanation of reading into array from disk program

Lines 10–80: This section controls the execution of the program. Its three main functions are to open the error channel and call for the allocation of a disk memory buffer, to call up the next module, and then to print out selected files from the array DI$.

Lines 1000–1200: We have come across these lines before in the LIST TRACK AND SECTOR program in Chapter 10. Their overall function is to read the contents of the directory into the array DI$.

Lines 1050–1060: The contents of a single sector are read into the buffer and the buffer pointer set to the beginning of the buffer. The first sector to be read will be track 18, sector 0.

Lines 1070–1090: The first two bytes of the sector, which are pointers, are obtained and stored in the two variables NT and NS, standing for Next Track and Next Sector. On the first pass through the module, the sector

picked up will be the BAM, so the program immediately moves on to the next sector.

Lines 1100–1180: The buffer pointer is set back to the beginning of the block, then the eight file entries contained in the sector are successively read. This involves discarding the two unused leading bytes and then obtaining the next 30 characters. The 30-character entry is then placed in a line of the array DI$.

Lines 1190–1200: If the next track pointer indicates track 0 at this stage it is a sign that the sector which has just been dealt with is the last in the directory.

3. Repeating a process on multiple files

Given the flexibility of the means provided by the LOAD "$" method and the DOS support facility to print the directory, there are few occasions on which it is worth reading the directory directly. One use, however, might be whenever an operation is to be performed on multiple files. In Chapter 5 we noted that very few commands could be used with the pattern matching facilities that the 1541 supports. With a little bit of programming, however, it is relatively easy to construct routines to carry out an operation on a whole series of files which match a certain pattern, and this depends on the ability to read and make use of the information contained in the directory.

Program to repeat an operation on a series of files

```
30900 GOTO 33000
31000 REM*********************************
31010 REM READ 1541 DIRECTORY
31020 REM*********************************
31030 DP = 0
31040 OPEN 8,DEV,0,"$"
31050 GET#8,T$ : GET#8,T$
31060 GET#8,T$ : GET#8,T$ : IF T$="" 
 THEN CLOSE 8 : RETURN
31070 GET#8,T$ : GET#8,T$
31080 GET#8,T$
31090 IF T$<>CHR$(34) AND T$<>""
 THEN 31080
```

```
31100 IF T$="" THEN 31060
31110 T1$ = ""
31120 GET#8,T$
31130 IF T$<>"" THEN T1$ = T1$+T$
: GOTO 31120
31140 DI$(DP) = T1$ : DP = DP+1
31150 GOTO 31060
32000 REM*********************************
32010 REM PATTERN MATCH N$ WITH PA$
32020 REM*********************************
32030 SAME = -1
32040 T = LEN(N$)
32050 IF LEN(PA$)>T THEN T = LEN(PA$)
32060 FOR I = 1 TO T
32070 T1 = (MID$(PA$,I,1)=MID$(N$,I,1))
 OR (MID$(PA$,I,1)="?")
32080 SAME = SAME AND ((MID$(PA$,I,1)=
"*") OR T1)
32090 IF MID$(PA$,I,1)<>"*" THEN NEXT I
32100 RETURN
33000 REM*********************************
33010 REM REPEAT FOR ALL PROGRAMMES
33020 REM*********************************
33030 DIM DI$(150)
33040 INPUT "DEVICE NUMBER ? 8[CL][CL]
[CL]";DEV
33050 INPUT "PATTERN ? *[CL][CL][CL]
";PA$
33060 GOSUB 31000
33070 IF DP<2 THEN 33200
33080 FOR I0 = 1 TO DP-1
33090 T$ = DI$(I0) : T = -1
33100 FOR I1 = 1 TO LEN(T$)
33110 IF MID$(T$,I1,1)=CHR$(34) THEN
 T = I1-1
33120 NEXT I1
33130 IF T<1 THEN 33190
33140 N$ = LEFT$(T$,T)
33150 TY$ = MID$(T$,19,3)
33160 GOSUB 32000
33170 IF NOT SAME THEN 33190
33180 GOSUB XXXXX : REM THIS IS THE
 ROUTINE TO BE EXECUTED
33190 NEXT I0
```

```
33200 END
```

Explanation of repeating an operation on a series of files program

Lines 31000–31150: The module to read the filenames from the directory using the first of the two methods illustrated in the second section of this chapter.

Lines 32000–32100: These lines compare two strings, one of which is the name of a file taken from the directory, the second being a string input in the next module which is the pattern against which all the disk files are to be matched. The pattern may be set up in the same way as described in Chapter 5, using the '*' and '?' indicators. The only important product of the module is the value of the variable SAME. If the filename being considered by the module matches the pattern, then the value of SAME will be left at minus one, otherwise it will be zero when execution of the module ends.

Lines 33000–33200: This section is the main control module, which first calls up the module at line 31000 to read the directory into the array DI$, then sends successive file names to the preceding module for comparison with the pattern input by the user. An extra facility is provided in the form of the creation of TY$, which records the type of the file. No use is made of this in the current program but you might like to employ it to exclude certain file types from an operation, regardless of their name.

In actual use, there would need to be another module specifying exactly what action was to be performed on a file which matched the pattern. This extra section would be written as another subroutine and would be called by the GOSUB at line 33180. *Note:* Since there is no valid line number at 33180, the routine cannot be run successfully in its present form — you *must* first add the new section specifying the action to be performed. Given below is an example procedure illustrating the use of the REPEAT facility.

Example procedure illustrating the use of REPEAT

1) Enter and SAVE the REPEAT program given above.

2) Take a disk which contains no important files (something may go wrong!) or format a new disk and SAVE on it three files with different names — the content of the files is irrelevant but the filenames should be *less* than 16 characters long.

3) LOAD the repeat facility and amend it by entering the following new or changed lines:

```
38180 GOSUB 34000

34000 REM*****************************
34010 REM RENAME ALL FILES
34020 REM*****************************
34030 OPEN 15,DEV,15
34040 COM$="RENAME0:Z" + N$ + "=
0:" + N$
34050 PRINT#15,COM$
34060 CLOSE 15
34070 RETURN
```

4) SAVE the amended program under the name REPEAT2.

5) RUN the program and, when asked to enter the pattern, simply press RETURN which enters a single asterisk indicating that any filename will be acceptable as a match.

6) When the program terminates, load the directory and you should find that every one of the files on the disk has a 'Z' at the beginning of the filename. If so, you have successfully carried out a procedure which would be impossible by means of normal pattern matching.

CHAPTER 12
Machine Code Programming Commands

1) Reading the memory of the disk drive
2) Writing to memory
3) Executing machine code in the disk drive memory

Despite the immense power and flexibility of the Disk Operating System program which takes up 16K of ROM within the 1541, there may be occasions when competent programmers wish to write their own machine code routines to control the disk drive. It is worth stressing again that this is not a decision to be taken lightly. Though it is unlikely, it is *possible* to damage the disk drive by running inadequately thought-out programs since, unlike the Commodore 64, the 1541 has moving parts which are under the control of its 6502 microprocessor.

1. Reading the memory of the disk drive

Reading the contents of the disk drive RAM and ROM, which extend from address $0–7FF and $C000–FFFF respectively, is made possible by the use of the MEMORY-READ command. The format for this is:

PRINT#<**FILE NUMBER**>,"**M-R**" CHR$(**LO**) CHR$(**HI**) [CHR$(**CHARS**)]

1) **FILE NUMBER** — the number of a file previously opened to the error channel.

2) "**M-R**" — the abbreviation for MEMORY-READ — the full version of this command is not accepted. *Note:* The absence of a colon, ':', from the end of the command is *not* an error. Editions of the 1541 disk manual up to the date of publication of this book are wrong in including the colon in the format of the command. None of the 'M-' commands will be accepted by the 1541 if a colon is appended.

3) **LO and HI** — calculated on the basis that, if X is the address to be read,

then HI = INT(X/256) and LO = X−256*HI.

4) **CHARS** — the number of bytes to be read — see below on the use of GET # to obtain the data.

The M-R command does not, by itself, read the memory. What it does, unless the optional CHARS specifier is added, is load a single byte from the specified location into the error channel buffer, ready for reading by the 64. If a GET # instruction is then issued, this will obtain the single byte from the address specified. The technique given in editions of the 1541 manual up to the publication of this book, where the M-R command is employed without the use of the CHARS specifier and then a succession of bytes obtained by the use of GET #, does not appear to work. In our experience, all that is obtained after the first byte is the characters of the current error message. Thus, it would appear that, contrary to information in the manual, either a fresh M-R command must be issued for each successive byte to be picked up from the drive memory, or the number of bytes to be picked up must be specified, with CHR$(0) representing the maximum of 256 bytes without the use of another M-R command. Note that, whenever less characters are read than have been specified, the sequence must be terminated by the use of INITIALIZE since, until this is done, the *only* thing that will be obtained from the error channel is bytes of memory.

The program given below shows how the M-R command can be used to dump out an area of the disk drive memory. Those who are familiar with our other books will recognise that the program consists of the memory dump routines taken from the Mastercode Assembler, adapted to read and format the disk memory instead of the internal memory of the 64.

Program using memory-read to dump disk memory contents

```
17000 REM#*******************************
17010 REM DO MEMORY DUMP
17020 REM*******************************
17030 DEV = 8
17040 OPEN 15,DEV,15
17050 DEFFNHI(X) = INT((X-INT(X/65536)
*65536)/256)
17060 DEFFNLO(X) = X-INT(X/256)*256
17070 INPUT " START ADDRESS ";T$
17080 IF LEFT$(T$,1)="$" THEN T$ = MID$
(T$,2) : GOTO 17140
17090 AD = VAL(T$)
17100 IF AD<>0 THEN 17170
```

```
17110 FOR I = 1 TO LEN(T$) : T1$ = MID$
(T$,I,1)
17120 IF T1$>"9" OR T1$<"0" THEN PRINT
 "[CU][CU]" : GOTO 17070
17130 NEXT : GOTO 17240
17140 GOSUB 20000
17150 IF ERR THEN PRINT "[CU][CU]" :
 GOTO 17070
17160 AD = T
17170 IF AD>65535 OR AD<0 THEN PRINT
 "[CU][CU]" : GOTO 17070
17180 INPUT " OUTPUT TO PRINTER (Y/N)
 ? N[CL][CL][CL]";T$
17190 OP = T$="Y"
17200 IF NOT OP AND T$<>"N" THEN PRINT
 "[CU][CU]" : GOTO 17180
17210 T = 3
17220 IF OP THEN T = 4
17230 OPEN 1,T
17240 PRINT "[CLR]"
17250 FOR I = 0 TO 20
17260 GOSUB 18000
17270 PRINT#1,HE$
17280 NEXT
17290 INPUT "[CD]  CONTINUE (Y/N)
 ? Y[CL][CL][CL]";T$
17300 IF T$="Y" THEN 17240
17310 IF T$<>"N" THEN PRINT "[CU][CU]
[CU]" : GOTO 17290
17320 PRINT#15,"I0"
17330 CLOSE 1
17340 CLOSE 15
17350 END
18000 REM*********************************
18010 REM READ 8 BYTES FROM DISC
18020 REM*********************************
18030 O1$ = "" : O2$ = ""
18040 T1$ = "" : T2$ = ""
18045 PRINT#15,"M-R" CHR$(FNLO(AD))
 CHR$(FNHI(AD)) CHR$(0)
18050 FOR J = 0 TO 7
18070 GET#15,T$ : T$ = LEFT$(T$+CHR$
(0),1)
18080 T = ASC(T$)
```

```
18090 GOSUB 19000
18100 T1$ = T1$+RIGHT$("00"+HE$,2)+" "
18110 T3$ = "."
18120 IF T$>CHR$(31) AND T$<CHR$(128)
 THEN T3$ = T$
18130 T2$ = T2$+T3$
18140 NEXT
18150 T = AD
18160 GOSUB 19000
18170 HE$ = RIGHT$("0000"+HE$,4)+" "+T1$
+" "+T2$
18180 AD = AD+8
18190 RETURN
19000 REM#********************************
19010 REM CONVERT T TO HEX
19020 REM********************************
19030 HE$ = ""
19040 T1 = T - INT(T/16)*16
19050 T = INT(T/16)
19060 HE$ = CHR$(T1+48-(T1>9)*7)+HE$
19070 IF T>0 THEN 19040
19080 RETURN
20000 REM#********************************
20010 REM CONVERT HEX TO DECIMAL
20020 REM********************************
20030 T = 0 : ERR = 0
20040 FOR I = 1 TO LEN(T$)
20050 T1 = ASC(MID$(T$,I,1))-48
20060 IF T1>9 THEN T1 = T1-7+(T1>22)*15
20070 T = T*16+T1
20080 ERR = T1>15 OR T1<0 OR ERR
20090 NEXT
20100 RETURN
```

Explanation of memory dump program

Lines 17000–17350: This section of the program allows the user to specify the start address from which memory is to be dumped out. The address may be entered in decimal or, if preceded by '$', in hexadecimal. Display of the results of the dump is normally to the screen but output to printer can be specified. After 20 lines of eight bytes have been displayed, the user has the option to continue or terminate the program. Note that, on termination,

the disk drive is initialised to ensure that it is prepared for any subsequent commands — a disk error message will be generated if no disk is present in the drive.

Lines 19000–19080: These lines convert a number expressed in decimal to its equivalent in hexadecimal.

Lines 20000–20100: The converse of the above, these lines convert a number expressed in hexadecimal to its equivalent in decimal. In addition, the error variable ERR will be set to minus one as a flag if the hexadecimal number input is incorrect in its format.

Lines 18000–18190: This section reads one line of eight bytes from the disk drive memory, translates each byte into hexadecimal by calling up the module at line 19000, then adds the string representing the hexadecimal value to T1$ and, if appropriate, the ASCII character with the same code to the string T2$. These are then combined into the string HE$ to be printed, line by line, by the module at line 17000. The display which results gives the hexadecimal form of eight bytes on the lefthand side of the screen and shows clearly the presence of any meaningful groups of ASCII characters on the righthand side.

2. Writing to memory

Data can be written to the disk drive memory in blocks of up to 34 bytes, using the MEMORY-WRITE command, the format of which is:

PRINT #< **FILE NUMBER**> ,"**M-W**" CHR$(**LO**) CHR$(**HI**) CHR$(**CHARS**)< **BYTES**>

1) **FILE NUMBER** — the file number of a file previously opened to the error channel.

2) "**M-W**" — the abbreviation for MEMORY-WRITE — the full form is not accepted.

3) **LO and HI** — see the same values in the previous section.

4) **CHARS** — the number of bytes to be sent, up to 35 in all with a single command.

5) **BYTES** — the bytes to be sent, either in the form CHR$(BYTE1),

CHR$(BYTE2), etc, or in the form of a string of ASCII characters where appropriate. Should the number of characters specified exceed those contained in the string sent to the error channel, the carriage return character, ASCII code 13, will be stored at the end of the characters being written to memory.

Explanatory procedure to illustrate the use of M-W

1) Enter the MEMORY DUMP program given above and SAVE it.

2) With the MEMORY DUMP program in memory and a (not too important) disk in the drive, dump out the contents of the memory beginning at $400. Unless you have been playing around with the disk drive memory, the contents should consist of zeros.

3) Add the following lines to the program:

```
21000 OPEN 15,8,15
21010 PRINT#15,"M-W" CHR$(0) CHR$(4) CHR$
(26) "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
21020 CLOSE 15
```

4) Type:

RUN 21000[RETURN]

5) After a momentary pause, the cursor will return, so RUN the main program. When asked to specify an address, enter $400 again. You should see that the letters of the alphabet have been successfully stored from $400 onwards.

3. Executing machine code in the disk drive memory

As mentioned before, machine code programs to control the 6502 chip which runs the disk drive may be executed in the drive memory. Such programs may consist of routines from the 1541 ROM or programs written by the user. Execution of such programs is accomplished by the MEMORY-EXECUTE command, which has the format:

PRINT #< **FILE NUMBER**> , CHR$(**LO**) CHR$(**HI**)

1) **FILE NUMBER** — the number of a file previously opened to the error channel.

2) **LO and HI** — the start address of the machine code routine, with the two bytes calculated as shown in the first section of this chapter.

CHAPTER 13
Changing Device Numbers

As mentioned in Chapter 1, the use of multiple disk drives depends upon the ability to alter the device number of one or more drives, since drives with the same device number will create confusion within the system. Data sent *to* a device number which is shared by two drives may well be stored properly on both, but chaos will result from a request to two devices to supply information to the 64. There are two methods of changing the device number, one by making a slight modification to the circuit board of the 1541 and the other by a single memory command. The hardware method, we would recommend you have carried out by a competent dealer, but the software method can be accomplished by a command under the following format:

PRINT #< **FILE**> ,"**M-W**" CHR$(**119**)CHR$(**0**) CHR$(**2**)
CHR$(**DEV + 32**)CHR$(**DEV + 64**)

1) **FILE** — the number of a file you have previously opened to the error channel.

2) "**M–W**" — the memory write command.

3) **119 and 0** — the address of a register in the disk memory, ie 119 + 256*0 = 119.

4) **2** — the number of bytes being written to memory.

5) **DEV**— is the device number you wish to assign to the disk drive.

Thus, to change the device number of a disk with device number 8 so that it responds as device number 9, follow this procedure:

1) Ensure that any other disks with a device number of 8 are switched off.

2) Type:

```
OPEN 15,8,15 [RETURN]
PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$
(41)CHR$(73) [RETURN]
CLOSE 15
```

3) Now switch on any other drive(s).

Note that if for any reason the disk drive has to be switched off, this process will have to be repeated as the drive will revert to its original device number.

APPENDIX A
Disk Error Messages

NUMBER	MESSAGE
0	No error condition current.
1	SCRATCH command has been carried out. The number in the 'track' position of the error message represents the number of files scratched.
2–19	These numbers are not implemented on the 1541.
20	The header of the last specified sector cannot be found.
21	The disk drive is unable to pick up the special timing mark placed on to each track. If this happens with more than one disk you should have the drive checked.
22	The drive has been asked to read a sector which cannot be found.
23	The checksum, or figure which is stored along with each sector as a check against corruption, indicates that data has been incorrectly retrieved.
24	More general message indicating that data is being picked up from the disk in garbled form.
25	The data which has been written to the disk does not agree with what *should* have been written.
26	Attempt to write to a disk which is write protected, eg has a write protect tab stuck on.
27	The header of a sector does not agree with its checksum.
28	The disk drive cannot detect the beginning of the block following the one being written to. The disk must normally be reformatted.
29	The ID of the disk does not match the ID with which the disk drive has been working. Disk must be INITIALIZEd if it has been changed. Disk may be faulty.
30	The last command sent to the drive was not understood, probably due to an invalid format.
31	The last command sent to the drive was not understood, probably due to an invalid keyword.
32	The last command sent to the drive was too long.
33	An invalid file name has been employed.

34	The drive cannot detect a file name within the instruction, often caused by omitting the colon which follows the zero drive number specifier.
39	The last command sent to the drive was not understood.
50	The drive has been instructed to read data beyond the end of the file.
51	Data to be printed to a relative file is too long for the specified size of record within the file.
52	A position has been specified within a relative file which would extend the file beyond the limits of disk capacity.
60	The file which being opened for reading was previously opened for reading and has not been properly CLOSEd.
61	The disk drive is told to access a file which has not been OPENed.
62	The disk drive has been asked to find a file which is not present on the disk.
63	The disk drive has been instructed to create a file with the same name as one which already exists on the disk.
64	The disk drive is being asked to treat a file of one type as if it were a file of another.
65	The sector specified in a BLOCK-ALLOCATE command is already allocated. The track and sector figures indicate the next highest track and sector available. Zero in both positions indicates that no higher track and sector is available.
66	The track and sector specified in the last operation do not exist. This may be due to programming error or may arise out of corruption on the pointers indicating the next sector of a file.
67	Indicates an illegal system track or sector.
70	The specific channel requested to the disk drive is not available *or* there are no channels available at all.
71	The Block Allocation Map does not match what is actually on the disk. The disk should be INITIALIZEd, though some damage may have been done to existing files already.
72	Either there are too many files on the disk to add another to the directory, or there is no more space on the disk.
73	An attempt has been made to write on a disk formatted with a different Disk Operating System.
74	The drive has been accessed without a disk being present.

APPENDIX B
Additional Machine Code Commands

In addition to the main commands specified in Chapter 12, there are a series of additional 'U' commands, as follows:

U3 (UC)	jump to $0500
U4 (UD)	jump to $0503
U5 (UE)	jump to $0506
U6 (UF)	jump to $0509
U7 (UG)	jump to $050C
U8 (UH)	jump to $050F
U9 (UI)	jump to vector at $FFFA
U; (UJ)	power up vector
UI+	set drive to Commodore 64 speed
UI–	set drive to VIC 20 speed

Note that the main body of jump commands are spaced by three bytes each in their destinations. This is to enable the user to create a jump table at $0500 in the disk memory.

APPENDIX C
DOS Support Commands

Later 1541 drives have been supplied with a useful program under the name name 'DOS Support', which allows the housekeeping commands to be entered more easily from direct mode (without a line number). To load the DOS Support facility into memory, use the C-64 WEDGE program supplied on the disk of utility programs which comes with your drive. This loads and runs a machine code program to extend the 64's BASIC.
The commands which DOS Support makes available are as follows:

1) @< command string> or > < command string>
2) @ or >
3)/ < filename>

1) The command string referred to is any valid command along the error channel, which is automatically opened by the DOS Support program. Entering '$' or '$0' as a command results in the directory being printed to the screen *without* the current program in memory being lost. While the directory is being listed it can be paused by pressing the spacebar, and listing recommenced by pressing any other key. Pressing RUN/STOP terminates the listing.

2) This form of the commands returns the current disk error message, removing the need to open the error channel, read it, and then close it again.

3) Loads a file without the need for LOAD or the use of quotation marks around the file name. The directory may be loaded by entering '/$'.

If you are making use of multiple disk drives and wish to use DOS Support on a drive other than device 8, you must first load it from device 8, then enter:

OPEN 15,< NEW DEVICE NUMBER> ,15 : CLOSE 15 : SYS 52224

The start up message of the DOS Support program will be displayed and the program will now be configured for the new device number.

It is important to remember that the DOS Support commands *cannot* be used in a program; they can *only* be made use of in direct mode.

Index

1540 drive, ix

B
Backup files, 38
Backup disks, 19
Block Allocation Map, 3, 19, 43
BLOCK-ALLOCATE:, 111
BLOCK-EXECUTE:, 112
BLOCK-FREE:, 111
BLOCK-READ:, 110
BLOCK-WRITE:, 109
Buffer, 108
Buffer pointer, 108

C
Carriage return character, 62
Channel, 25, 58
CLOSE, 26, 69
Concatenation of files, 41
Connecting the drive, 9
COPY, 41

D
Detecting the end of file, 68
Device number, 17, 137
Direct mode, 53
Directory, 2, 121
Disk Operating System, 7
Disk drive memory dump program, 132
Disk drive status, 54
Disk ID, 33
Diskbase program, 99

E
Error channel, 25, 51
Error messages, 52

F
File number, 25, 58
Formatting, 30

G
GET #, 64

I
INITIALIZE, 43

K
Kernal, 55
Keyword abbreviations, 29

L
LIST TRACK AND SECTOR program, 115
Listing a program, 80
LOAD, 16

M
Machine code, 131
MEMORY-READ, 131
MEMORY-WRITE, 135
Merging programs, 85

N
NEW, 30
Numeric variables, 63

O
OPEN, 23, 96
Overwriting files with '@0:', 18

P
Pattern matching, 47
PRINT #, 28
Program files, 75

Punctuation in relative files, 98
Punctuation with PRINT # and INPUT #, 59

R
Random files, 107
Reading the error channel, 52
Relative files, 95
RENAME, 36
Renumbering a program, 89
Repeating a process on multiple files, 126

S
SAVE, 5, 15, 16
Saving areas of memory, 79
SCRATCH, 34
Sector, 2
Sequential files, 57
Storage of numbers in strings, 67
Strings, 63
Structure of a BASIC program file, 76

T
Track, 2

U
U1:, 110
U2:, 110
Unsavable characters, 64
UNSCRATCH program, 112
User files, 57

V
VALIDATE, 19, 44
VERIFY, 18
VIC 20, ix

Other titles from Sunshine

SPECTRUM BOOKS

Artificial Intelligence on the Spectrum Computer
Keith & Steven Brain — ISBN 0 946408 37 8 — **£6.95**

Spectrum Adventures
Tony Bridge & Roy Carnell — ISBN 0 946408 07 6 — **£5.95**

Machine Code Sprites and Graphics for the ZX Spectrum
John Durst — ISBN 0 946408 51 3 — **£6.95**

ZX Spectrum Astronomy
Maurice Gavin — ISBN 0 946408 24 6 — **£6.95**

Spectrum Machine Code Applications
David Laine — ISBN 0 946408 17 3 — **£6.95**

The Working Spectrum
David Lawrence — ISBN 0 946408 00 9 — **£5.95**

Inside Your Spectrum
Jeff Naylor & Diane Rogers — ISBN 0 946408 35 1 — **£6.95**

Master your ZX Microdrive
Andrew Pennell — ISBN 0 946408 19 X — **£6.95**

COMMODORE 64 BOOKS

Graphic Art for the Commodore 64
Boris Allan — ISBN 0 946408 15 7 — **£5.95**

DIY Robotics and Sensors on the Commodore Computer
John Billingsley — ISBN 0 946408 30 0 — **£6.95**

Artificial Intelligence on the Commodore 64
Keith & Steven Brain — ISBN 0 946408 29 7 — **£6.95**

Machine Code Graphics and Sound for the Commodore 64
Mark England & David Lawrence — ISBN 0 946408 28 9 — **£6.95**

Commodore 64 Adventures
Mike Grace — ISBN 0 946408 11 4 — **£5.95**

Business Applications for the Commodore 64
James Hall — ISBN 0 946408 12 2 — **£5.95**

Mathematics on the Commodore 64
Czes Kosniowski — ISBN 0 946408 14 9 — **£5.95**

Advanced Programming Techniques on the Commodore 64
David Lawrence — ISBN 0 946408 23 8 — **£5.95**

The Working Commodore 64
David Lawrence ISBN 0 946408 02 5 **£5.95**

Commodore 64 Machine Code Master
David Lawrence & Mark England ISBN 0 946408 05 X **£6.95**

Programming for Education on the Commodore 64
John Scriven & Patrick Hall ISBN 0 946408 27 0 **£5.95**

ELECTRON BOOKS

Graphic Art for the Electron Computer
Boris Allan ISBN 0 946408 20 3 **£5.95**

Programming for Education on the Electron Computer
John Scriven & Patrick Hall ISBN 0 946408 21 1 **£5.95**

BBC COMPUTER BOOKS

Functional Forth for the BBC Computer
Boris Allan ISBN 0 946408 04 1 **£5.95**

Graphic Art for the BBC Computer
Boris Allan ISBN 0 946408 08 4 **£5.95**

DIY Robotics and Sensors for the BBC Computer
John Billingsley ISBN 0 946408 13 0 **£6.95**

Essential Maths on the BBC and Electron Computer
Czes Kosniowski ISBN 0 946408 34 3 **£5.95**

Programming for Education on the BBC Computer
John Scriven & Patrick Hall ISBN 0 946408 10 6 **£5.95**

Making Music on the BBC Computer
Ian Waugh ISBN 0 946408 26 2 **£5.95**

DRAGON BOOKS

Advanced Sound & Graphics for the Dragon
Keith & Steven Brain ISBN 0 946408 06 8 **£5.95**

Artificial Intelligence on the Dragon Computer
Keith & Steven Brain ISBN 0 946408 33 5 **£6.95**

Dragon 32 Games Master
Keith & Steven Brain ISBN 0 946408 03 3 **£5.95**

The Working Dragon
David Lawrence ISBN 0 946408 01 7 **£5.95**

The Dragon Trainer
Brian Lloyd ISBN 0 946408 09 2 **£5.95**

ATARI BOOKS

Atari Adventures
Tony Bridge — ISBN 0 946408 18 1 — **£5.95**

Writing Strategy Games on your Atari Computer
John White — ISBN 0 946408 22 X — **£5.95**

GENERAL BOOKS

Home Applications on your Micro
Mike Grace — ISBN 0 946408 50 5 — **£6.95**

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, VIC 20 and 64, ZX 81 and other popular micros. Only 40p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role-playing games. Includes reviews of all the latest software, lists of all the software available and programming advice. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01-437 4343

Telex: 296275

Printed in England by Commercial Colour Press, London E7.

>Here's our way of saying Thanks for buying this Sunshine book...

>Sunshine Special Offer 1

Subscribe to Commodore Horizons, Britain's brightest Commodore magazine, and get an extra three months' issues absolutely free!

For just £10 you get a year's subscription plus the next 3 months without charge – a saving of £2.50. Complete and send the order card now, to reserve your special subscription, along with your cheque or postal order, or charge to your Access card.

See over for two other free Sunshine Offers.

>See over for two other free Sunshine Offers ➡

>Sunshine Special Offers Claim Card

Yes, I'd like to take advantage of your Sunshine Special Offers.
Please make sure I get the following:

☐ A year's subscription to Commodore Horizons, plus 3 months free issues, for just £10.

☐ A year's subscription to Popular Computing Weekly, plus 12 issues free, for just £19.95.

☐ A copy of your free colour Sunshine Books brochure.

Payment methods:

☐ I enclose a cheque/postal order payable to Sunshine Books for £_____

☐ Charge my Access card No._____

Signed_____

Name_____

Address_____

_____

All offers end on 31st December 1984.

>Sunshine Special Offer

2

Popular Computing Weekly is the best-selling weekly newsmagazine for every computer whiz. And here's your opportunity to get 12 issues, worth £4.20, free of charge.

Use the special offer order card to claim your special Popular Computing Weekly subscription – 64 issues for the price of a year's copies – at a price of just £19.95. Send today, and look forward to 15 months of great weekly reading.

>Sunshine Special Offer

3

If you liked this Sunshine Book, take a look at the rest of our amazing range. Send today for our free colour brochure detailing all the Sunshine Books you can buy today, and all the new books on the way.

And once you're on our special mailing list, we'll keep you updated on what's new for you and your computer from Sunshine. Just tick the box on the special order card.

Send to:

Sunshine Books,
12/13 Little Newport Street,
London WC2R 3LD.

Commodore 64 Disk Companion is the authoritative guide to making the most of your 1541 disk drive.

The book gives a simple introduction to the ins and outs of disk storage and the way a disk drive works. It then leads the reader into an understanding of the full range of advanced techniques available on the 1541.

There is a full exploration of all the different types of files the 1541 can handle, including the full use of program files over and above simply saving and loading, sequential files, relative files and random access files. Other sections explain the use of the error channel, manipulating the disk directory and accessing the internal operating system of the 1541 for a wide range of applications.

Commodore 64 Disk Companion includes programs, among others, for fast data storage, recovering scratched files, hiding files, listing out every program on a disk, renumbering program files on disk, merging disk files and displaying the internal memory of the disk operating system.

David Lawrence is the author of several best-selling computer books, including The Working Commodore 64 and Advanced Programming Techniques on the Commodore 64. Mark England is co-author, with David Lawrence, of the acclaimed Commodore 64 Machine Code Master and Machine Code Graphics and Sound for the Commodore 64. He is also the author of the successful Mastercode Assembler.

GB £ NET +007.95

ISBN 0-946408-49-1

00795

9 780946 408498

0684
NR17299
178,25

SUNSHINE

ISBN 0 946408 49 1



This was brought to you

from the archives of

http://retro-commodore.eu