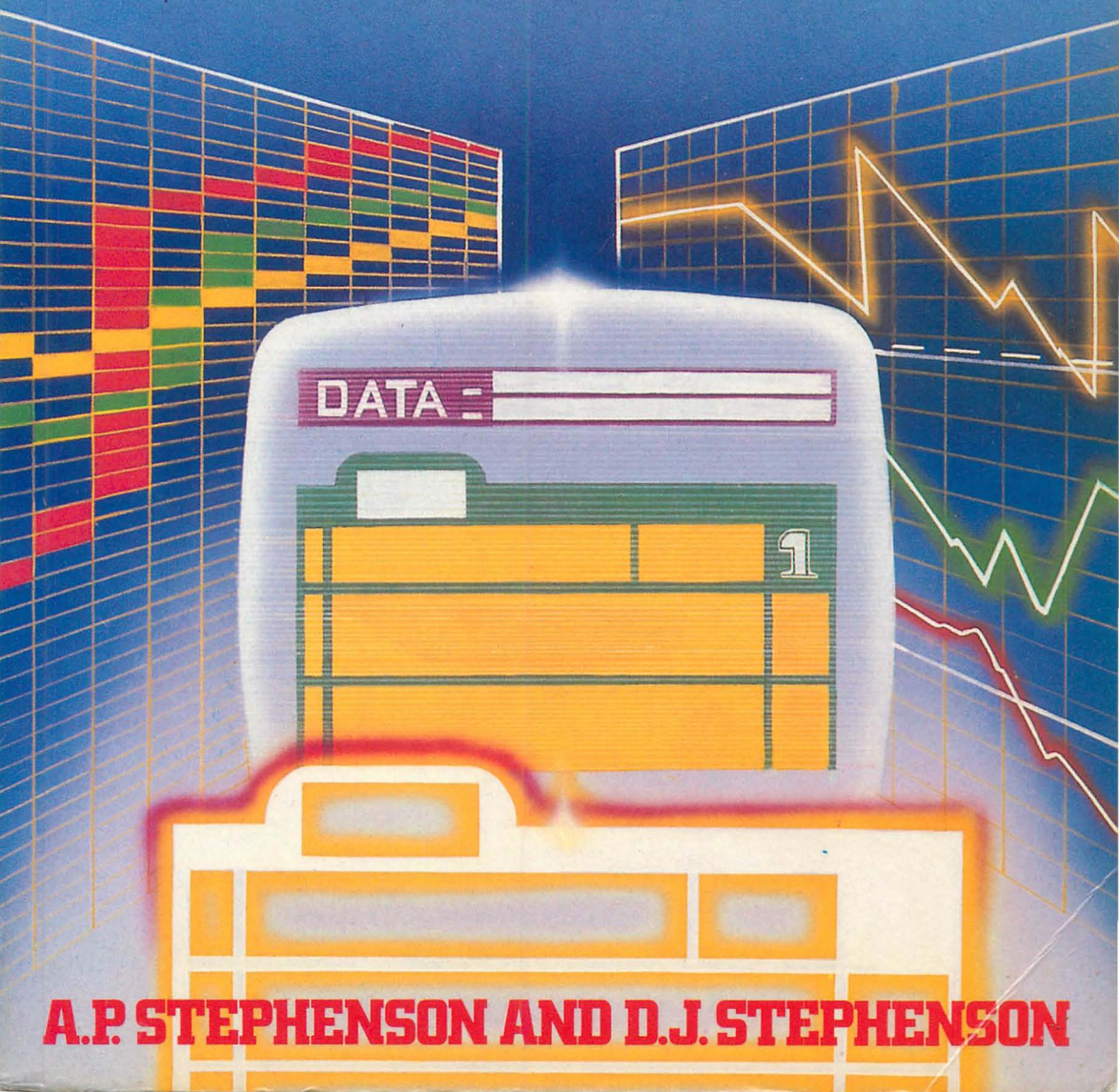
FILING SYSTEMS AND DATABASES FOR THE COMMODORE 64



A.P. STEPHENSON AND D.J. STEPHENSON

Filing Systems and Databases for the Commodore 64

More books for Commodore 64 Users

Commodore 64 Disk Systems and Printers
Ian Sinclair
0 246 12409 1

Introducing Commodore 64 Machine Code
Ian Sinclair
0 246 12338 9

Advanced Machine Code Programming for the Commodore 64
A. P. Stephenson and D. J. Stephenson
0 246 12442 3

Business Systems on the Commodore 64
Susan Curran and Margaret Norman
0 246 12422 9

Adventure Games for the Commodore 64
A. J. Bradbury
0 246 12412 1

Commodore 64 Computing
Ian Sinclair
0 00 383070 5

The Commodore 64 Games Books
Owen Bishop
0 246 12258 7

Software 64: Practical Programs for the Commodore 64
Owen Bishop
0 246 12266 8

40 Educational Games for the Commodore 64
Vince Apps
0 246 12318 4

Commodore 64 Graphics and Sound
Steve Money
0 246 12342 7

Useful Subroutines and Utilities for the Commodore 64
Ian Sinclair
0 00 383012 8

Filing Systems and Databases for the Commodore 64

A. P. Stephenson and

D. J. Stephenson

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.

British Library Cataloguing in Publication Data
Stephenson, A.P.
Filing systems and databases for the
Commodore 64.
1. Data base management 2. Microcomputers
I. Title II. Stephenson, D.J.
001.64'42 QA76.9.D3

ISBN 0-00-383011-X

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

This book is devoted entirely to the storage, manipulation and retrieval of data, a task for which a computer is ideally suited. Although the Commodore 64 features in its title, much of the contents should also appeal to owners of rival machines. The programs, of course, are written in standard Commodore BASIC but it should be an easy task to translate them into other BASIC dialects.

It is hoped that the book will be of interest to both the absolute beginner and the experienced user alike. No previous knowledge of filing systems is assumed since progression is made from the most elementary versions, right through to more complex indexed relative files. Although many of the programs are designed for disk storage, users restricted to cassette tape have not been forgotten.

Apart from several introductory listings, there are four major filing programs, all of which are described in sufficient detail either to use as they stand or to stimulate further thought. BASIC sorting and searching techniques are described in detail. However, alternative ultra-fast machine code sorting routines are given together with full directions for splicing them into the main programs. Since many users may not have assembler facilities, the object code is supplied in the form of a hex loading program written in BASIC. Detailed discussion of 6510/6502 machine code, although beyond the scope of this book, is treated in our companion volume, *Advanced Machine Code Programming for the Commodore 64*, also published by Granada/Collins.

Computer science has grown at an alarming rate, so it is understandable that many technical terms in use are far from standardised. Some terms, when used in a mainframe context, take on a different shade of meaning when applied to home microcomputers. For example, databases and filing systems are carefully distinguished in literature aimed at the mainframe and minicomputer user but, as far as home microcomputers are concerned, most writers

refer to them as if they were one and the same. Again, terms used to distinguish various filing systems are not always used consistently but we have tried to avoid entering into hairsplitting arguments regarding terms.

Thanks are due to Richard Miles and his colleagues of Collins Professional and Technical Books (previously Granada Publishing) who once again have transformed our manuscript into a form suitable for publication.

A. P. Stephenson and D. J. Stephenson

Chapter One
Introduction

Home filing methods

Most people in modern society find a need to store information. If, for the moment, we define 'information' as that which conveys meaning, and a 'file' as a 'collection of related information', then it follows that we all keep files in some form or another, even if they are only gas and electricity bills. It is worth examining some of the filing methods used in the home to see if any of them will help us to design computer filing systems. The following classification is by no means exhaustive. In fact it does little more than represent some of the techniques employed by the authors before they became hooked on computers.

The single cardboard box

In many homes, the domestic filing system consists of a single cardboard box into which is stuffed any piece of paper considered to be of some importance in the future. It has the supreme advantage of freeing the mind from the agonies of decision making. There is only one box so there is only one file. Nothing could be simpler.

Labelled jam jars

Those who pride themselves on being methodical tend to despise the single box system on the grounds that it lacks precision. They prefer to use a battery of jam jars, each neatly labelled in accordance with the general nature of the contents. Paid and unpaid bills may be popped into one jar, cooking recipes in another and perhaps birth and marriage certificates, or other documents of similar status, in a slightly better class of jar. However, classification involves decisions. Once you embark on a methodical approach it is inevitable that the number of jars will begin to increase because either (a) some jars get full up, or (b) some bits of paper turn up, containing information

which will not fit into any one of your labelled categories, or (c) the original simple desire to introduce method could turn into a sinister obsession. If this happens, the number of jars may start to increase without limit.

Fortunately, the catastrophy forecast in (c) above is rare because most of us soon discover the value of the *miscellaneous* label. This is convenient for depositing awkward documents which fall into some unspecified category. The miscellaneous jar is also useful as a temporary (?) home, pending a final decision.

The office-type folder

There are those who despise both the cardboard box and the jam jar. They prefer to use traditional office filing equipment - that is to say, stiff folders neatly labelled with file heading and reference number, which enclose the relevant papers. *Sequential integrity* is preserved by means of a short tag string passing through the papers and the outer folder. However, there is little difference in principle between the office-type folder and the jam jar although the folder is convenient and projects a more up-market image.

The ideal filing system

It is not easy to define an ideal filing system, although we might all agree with the following tentative proposals:

(a) The ideal filing system should have infinite storage capacity.
(b) It should be possible to retrieve any specific item, or items, of information from the file instantaneously and with minimum effort.

Ideals are useful because they stimulate efforts to approach them. Let us commence with the desire for 'infinite storage capacity'. Leaving aside for the moment the impossibility of it ever being achieved, it is worth asking ourselves what we would do with infinite storage capacity even if we had it? According to current media teachings, 'information is power'. The amount of information available is alarming and appears to be growing almost exponentially with time. Up to the end of the Middle Ages, it was possible for a well-educated person with an excellent memory to know virtually all there was to be known. Now, a twenty-four volume encyclopaedia can be little more than a crude index to the world's knowledge. However, having all this knowledge available in printed form is one thing; finding a particular item of information is quite another. This brings us to the second desirable quality mentioned above - the ability to retrieve a specific item instantaneously and with the minimum of

effort. It is almost self-evident that the difficulty of finding a specific item increases proportionally with the total information stored. The larger the library, the more difficult it is to find the right book. The fatter the file, the more difficult it is to find the right document. The longer the cassette tape, the longer it takes to get at the bit you want.

The computer as a filing system

The first stored-program digital computer was constructed at Cambridge University in 1949 and was called EDSAC. It contained 132000 thermionic valves! The first machine delivered commercially was UNIVAC in 1951. At first, computers arrived slowly and were employed mainly by government engineers and scientists for solving mathematical problems or undertaking laborious arithmetical calculations relating to ballistics. A decade passed before computers began to be used for more general work. It was realised, rather belatedly, that the full potential of the computer rested in its ability to store and retrieve information. In other words, the ability to process information was equally, if not more, important than the ability to calculate. As a result, a new buzzword, *data processing*, appeared in order to distinguish it from 'ordinary' computing. Because of this shift in emphasis, the *storage capacity* of a machine, rather than its calculating ability, assumed greater importance.

The technology of the internal read/write memory (that which we now call *RAM*) was centred around a matrix of magnetic cores. Each bit was stored in a small ferrite ring through which three tiny wires passed carrying the signal currents for polarising the magnets either North or South (representing binary 1s and 0s). Because the magnetic core memory was labour-intensive, the cost was relatively enormous in spite of employing Third World labour. However, it had one advantage over the present-day semiconductor RAM. It was a non-volatile read/write memory. That is to say, it retained information even when the computer power was switched off because the bits were stored as blobs of permanent magnetism. Non-volatility was something of a holy cow before the arrival, in the mid-Sixties, of the semiconductor RAM. However, the cheapness and capacity of RAMs was enough to overcome initial opposition and the computer world gradually accepted the idea of a volatile internal memory.

The advent of the RAM resulted in a subtle change in terminology. Before the RAM era, the word 'memory' was seldom used by computer boffins because they disliked the anthropomorphic

association. They preferred the general word *store* but distinguished between internal or 'core' store, and external storage on peripheral equipment. This was referred to as *backing store*.

Memory and storage

Nowadays, when we refer to 'memory' it is tacitly assumed that we mean *semiconductor RAM*. Thus, when we speak of a 64K machine we imply that 64 kilobytes of RAM is installed. The Commodore 64 RAM complement consists of a bank of eight chips, each chip capable of storing 64K bits.

Peripheral devices, such as cassette tape, floppy disks or Winchesters are 'stores'. Certain classes of program make no use of store during a RUN. In such programs, storage is used only to SAVE the program before switching off. On the other hand, software concerned with processing large amounts of data (such as required in file management) will make heavy demands upon the store during program RUNs.

With regard to storage, the majority of Commodore 64 owners start off with cassette units. However, as funds allow or as interest grows, a substantial proportion of owners eventually invest in floppy disks and, providing a suitable interface is designed, the exclusive and well-heeled minority will end up with Winchesters. Because storage assumes great importance in filing systems, it is worth studying the properties of these three storage devices.

Cassette tape storage

Cassette tape has been a boon to home computer users. A computer, even the Commodore 64, is virtually useless without a storage system to back up the RAM. Way back in the early days of home computing, some enterprising technicians connected with an American magazine called *BYTE*, braved the sneers of the computer establishment and managed to interface an ordinary audio cassette unit to a small computer for storing programs. The system became known as the 'Kansas City Standard' (because *BYTE* was published in that city) and, subject to modifications and improvements, has been widely adopted ever since. The revised system became known as CUTS (Computer User's Tape Standard). Although the microprocessor must take the major share of the credit for the spread of home

computing, it is doubtful if it would have been as popular without the humble cassette unit. Not only is the unit itself cheap enough to be within reach of most people, the storage media it uses (blank cassettes) is also cheap.

Storage capacity

Audio cassettes are available in various lengths from C30 to C120 but it is unwise to use tapes longer than C30. Cassettes which are specially designed for computer storage are normally in C12 lengths. The longer the tape, the greater the chance of a jam or break.

The amount of data stored is best measured in terms of ASCII characters for a given *baud* rate and, to some extent, on the data transfer protocol in use. To gain some idea of cassette storage capacity, assume that one ASCII character requires 10 information bits (8 for the code and 2 for start/stop signals). The baud rate is essentially the number of information bits per second. (Strictly, we shouldn't speak of a 'baud rate' because, like the maritime knot, a baud is already a rate in terms of bits per second.)

From the above, it follows that:

Characters stored per second=baud rate/10

The total number of characters (N) which can be stored on a tape of length (C) minutes is therefore:

$$N = C \times 60 \times \text{baud rate}/10$$

Example:

A C12 tape running at, say, 600 baud can store $12 \times 60 \times 600/10 = 43200$ characters. This is equivalent to just over 42K of storage. In practice, some allowance must be made for end of block codes laid down by the cassette operating system and bits of tape which may be wasted at the beginning and end, so it is best to adopt a pessimistic approach and cut the arithmetical figure by half. If we do this, we can use the rough rule that a C12 tape, running at 600 baud, can store about 21K of real data.

Speed problems

It would be foolish to deny that cassette storage is slow. In fact it has often been stated (particularly by floppy disk salesman) that a cassette-based filing system is quite useless. We do not share this view. Cassette filing systems can be designed to a high standard and are certainly not useless. Does it really matter to the home computer user or the small businessman if some files take a few minutes to load?

It is a simple matter to program in a loud beep (or screech) when the loading is complete so you don't have to sit idle, watching the tape pass the reading heads.

Cassette storage is slow because of the tape transport speed and its serial access nature. Unless you are an expert with the fast forward/rewind buttons, you can't go straight to the bit you want. We shall see later that random access filing systems have considerable speed advantages but, unfortunately, they cannot be applied to cassette-based files.

Floppy disk storage

Sooner or later, users of the Commodore 64 will survive the initial love affair with the machine, and settle down to a permanent and perhaps more critical relationship. This inevitably results in the desire for add-ons. If the computer is to be used primarily for filing and data storage then the purchase of a floppy disk should be given the highest priority. In spite of our defence of the cassette unit, it is not in the same class as the floppy disk. No other peripheral (apart from a Winchester) could achieve such a transformation. If you have never seen a floppy disk in action or handled one, you are in for a pleasant surprise because it is like entering a new world. The speed of access to any part of the disk (a few seconds instead of several minutes) is the most significant, but by no means the only, advantage. A floppy disk, unlike the cassette unit is, almost, a *random access* device. This means that it is not necessary to read through unwanted data before reaching the wanted information. The read heads are mechanically positioned, by means of a stepping motor, in the region of the desired disk area before the read process is activated.

Yet another advantage is the impressive list of extra commands for manipulating files which become available when a disk unit is installed.

Disk drives

The credit for the development of the floppy disk is said to lie with an American engineer by the name of Shugart who was on the staff of IBM. The most obvious difference between the floppy and the more traditional hard disk is the square envelope which encloses the disk, even while it is rotating. This idea allowed

(a) disks to be handled with reasonable safety without too much danger to the delicate oxide coating;
(b) the soft lining of the envelope to maintain continuous self-cleaning and lubricating action on the disk surface;
(c) the envelope itself to fix the inter-gap distance between the read/write head and the oxide surface.

Another break from hard disk tradition was the change from continuous drive to the rotate-only-when-called drive. Instead of a continuous fast rotation (typically 3000 rpm) the floppy disk drive remains inactive until commanded to read or write. Thus the disk rotates occasionally, perhaps only a few minutes in total, during a computer session of several hours. This, together with the reduced speed of about 300 rpm, preserves the life of the disk. If it rotates continuously within the felt envelope, it would have a greatly reduced life, even at reduced speed.

Apart from the disk rotation mechanism, the read/write heads are at the end of an arm which is stepped radially across the disk to the selected *track*. It is important to note that new disks have no tracks at all. Tracks (and sectors within tracks) must be recorded on all new disks by a process known as *formatting* before they can be used for storing files. Because of this, the disks are said to be *soft sectored* to distinguish them from *hard sectored* which arrive from the manufacturers with pre-recorded tracks and sectors.

Only soft sectored disks can be used on the 64. Commodore, ever since the days of the PET 2001, have always made it difficult for the ordinary user to employ peripherals not manufactured by them. For example, the cassette tape interface is designed for Commodore tape units only. Other units can be persuaded to work but only after some hardware modifications. It is the same with the floppy disk drive interface which is designed to cater almost solely for Commodore disk drives. The essential feature, which acts against other manufacturer's peripherals, is the rather strange choice of the IEE-448 bus system which was originally designed by Hewlett-Packard for computer-controlled instrumentation work. The IEE-448 bus used on the Commodore 64 is a modified version using *serial*, instead of *parallel*, data transfer between computer and disk drive.

The Commodore 1541 single drive floppy disk unit

Most disk drive units designed for the personal computer market are

literally, disk 'drives'. That is to say, they do little more than drive the disk round and round and move the radial tracking arm across the disk under computer control. They are 90% mechanical and 10% electronic black boxes. Generally, therefore, the complex software program, necessary for controlling the disk and providing the extra commands, is expected to be in the computer itself.

Commodore disk drives, however, are different. For example, the 1541 contains not only the drive but all the extras, including its own mains-driven electrical power unit, an independent 6502 microprocessor, a 16K ROM containing the Disk Operating System (DOS), 2K of RAM and two input/output/timer chips. In fact, it is almost a complete and separate computer system. The self-contained microprocessor allows 'pipeline' communication between disk and computer so that the computer can be engaged on other duties whilst the disk is still dealing with the last batch of data. When you come to think of it, it is rather strange that 16K of ROM was found to be necessary to handle the DOS and yet only 8K of ROM (the Kernal) is used to handle the computer *operating system.*

Since the 1541 has such a massive amount of electronic and software support under the bonnet, we should expect great things from it. However, it must be said that whatever else is good about the 1541, we found the transfer speed between computer and disk to be rather disappointing. If you have only been used to cassette tape, the 1541 will appear to be a miraculous device. If, on the other hand, you have had some experience with disk systems, it is possible that you may share our disappointment. Page 5 of the User Manual which came with our 1541, contains the detailed specification of the unit. It is, overall, quite a detailed and impressive specification but, strangely, there is no mention of block transfer rate. However, the speed will be found adequate for general purpose work in the home and small business.

Ian Sinclair's *Disk Systems and Printers for the Commodore 64* is an excellent introduction to using the 1541 disk drive.

Disk tracks and sectors

At the present time, the most common diameter disk drive continues to be 5¼ inches but new 3 inch models are beginning to compete. The original diameter for Shugart drives was 8 inches and these were designed for the minicomputer rather than the microcomputer market. The 1541 drive uses the common 5¼" soft sectored diskettes.

The capacity of a floppy disk store depends on the drive, the DOS and the disk controller chips. Figure 1.1 shows how the disk surface is arranged into tracks and sectors after a disk has been formatted.

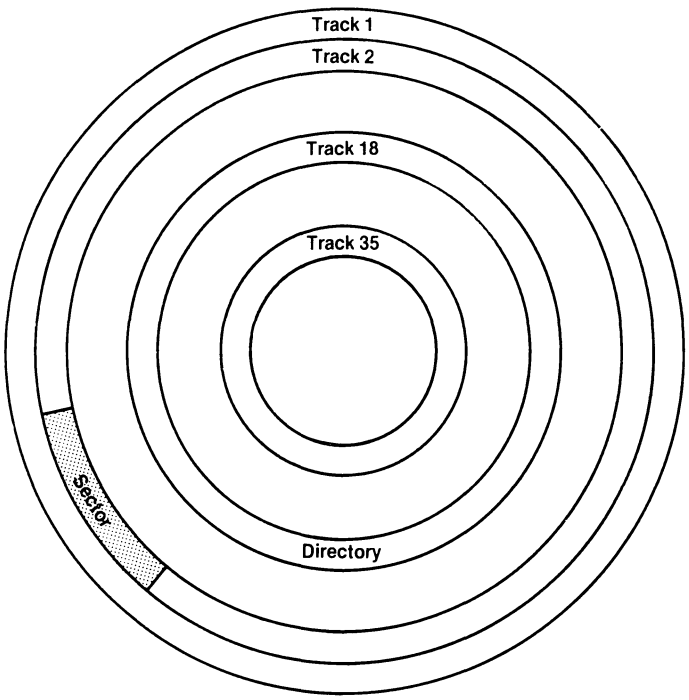


Fig. 1.1. Disk tracks and sectors.

As we have already mentioned, the floppy disk is said to be a *random access* device as distinct from the cassette unit which uses *serial access*. Risking the charge of hair-splitting, we should point out that a floppy disk is certainly random access as far as locating a chosen track is concerned but reverts to serial access within the track. Manufacturers' terms and specifications such as track number, tracks per inch, sector number and density can be very confusing. The following definitions should be read carefully.

Number of tracks

Formatting a blank disk causes 35 tracks, in the form of concentric circles, to be laid down. These tracks are, of course, invisible because they are merely areas of the disk which have been magnetically

marked out and synchronised by *timer markers*. Track 1 is on the outside and the innermost track is numbered 35.

Sectors and blocks

Each track is further subdivided into *sectors*. Each sector has room for 256 bytes of information. The term *block* is also used to indicate 256 bytes of data, so the terms sector and block are synonymous as far as the number of bytes is concerned. It is customary, however, to use the term 'sectors' when we are speaking of the 'disk geometry' but 'block' when we refer to '256 bytes of data'. Machine code users will also be aware that 256 bytes is often referred to as one *page* of memory. The number of sectors per track depends, to some extent, on the particular track. If all tracks were formatted with the same number of sectors, the recording *density* would be far higher on those nearest the centre. The higher recording density could increase the probability of read errors on the inner tracks. In order to equalise the recording density over the entire disk area, the outer tracks are arranged, during the formatting process, to have more sectors than the inner tracks. The sector allocation and addresses are as shown in Table 1.1.

Table 1.1. Track and sector format.

Track number	Sector range	Number of sectors
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17

At this point, it is worth mentioning that some confusion exists between different manufacturers over the interpretation they place on the term *recording density*. According to some authorities, the recording density is determined by the machine controller and, theoretically, has nothing to do with the mechanical drives, the diskettes or the number of tracks. These same authorities consider that 'density' should refer to the number of sectors per track which, in turn, is entirely dependent on the speed at which the computer sends/receives information to the read/write heads. It will be appreciated that passing information to and from the computer at double the rate demands a high quality read/write head on the drive arm. It is also reasonable to expect that more stringent quality

controls are necessary during the manufacture of the diskette surfaces when the recording density is high. Commodore state that 'single density' diskettes are suitable for the 1541. This is an indication of the confusion, mentioned above, with regard to 'density'. Some authorities consider 10 sectors per track to be 'single density' and 20 sectors per track, 'double density'. Single density grade diskettes certainly appear to work well on the 1541 in spite of the fact that there is, on average, 20 sectors per track. However, if a large quantity of important data is to be handled, it may be worth wasting a little extra money on the higher quality diskettes marked 'double density'.

Storage capacity

A single-sided diskette used on the 1541 can store 174848 bytes or 683 blocks, which is nearly 170K. The figure quoted represents total storage, not all of which is available for your files because the Directory consumes 19 blocks, leaving a total of 169984 bytes or 664 blocks free. However, this figure is reduced slightly, depending on the kind of filing system in use; that is to say, the number of bytes of free storage depends on whether the files are sequential, relative or random in nature.

The Directory and the BAM

The Directory acts as a catalogue of all files stored on the disk and occupies part of track 18. The BAM (Block Availability Map) is a kind of overseer, keeping tabs on all block allocations and is physically positioned close to the Directory on the disk. Track 18 was chosen for the directory and BAM because it is the centre track and therefore occupies the best average vantage point for access by the radial track arm. The Directory allows up to 144 file name entries and the BAM controls all the 683 blocks on the disk. It would be pointless to devote space here to the directory commands, or indeed many of the other disk commands because they are laid out clearly in the 1541 User's Manual. However, it is worth repeating, with emphasis, the warning given on page 11 of the manual regarding the independence of the BAM and Directory. Although the Directory is updated every time a program is SAVEd or a file is OPENed for writing, the BAM isn't correspondingly updated until the file is properly CLOSEd. If you forget to CLOSE a file, it is possible to *lose all data* in that file from the disk. Under such circumstances, it can be difficult to maintain a sense of humour.

Compatibility with other drives

There is some degree of compatibility with other Commodore disk drives. For example, any programs or files prepared on the 1541 drive can be read by either the 4040 dual drive or the 2031 and vice versa. The older 2040 drive is partially compatible, in that software prepared on the 2040 can be read by the 1541. The 1541 is, of course, a single drive unit so it is assumed that drive '0' is always in use.

Choice of diskettes

The 1541 has only a single read/write head so is capable of reading only one side of the disk at a time. Diskettes can be single- or double-sided. With regard to single-sided diskettes, it is worth mentioning that although both sides are coated with recording film, only one has passed quality control tests. Some enterprising types with an eye on the purse, have been known to cut a corresponding write-protect notch in the 'other side' so that the drive will accept it as valid. This is not a sensible thing to do because the 'other' side is always suspect. It is absurd economics to jeopardise data for the sake of such a small sum. Double-sided diskettes are less than double the price of single-sided so they are a better buy in the long run and will be useful if you eventually upgrade to a double-sided drive. There appears to be nothing against using double-sided diskettes on a single-side drive.

Formatting new diskettes

As mentioned earlier, the 1541 drive uses soft-sectoring so a new disk, just unwrapped, is useless unless it is first formatted. The instructions for formatting will not be given here because the procedures are adequately covered in the 1541 User Manual. Other useful routines connected with general disk management are also present on the special utility diskette.

Care of diskettes

Diskettes are normally purchased in boxes of ten which include a sheet of instructions on how to look after them in the form of dos and don'ts - mostly don'ts. The list of precautions can strike terror into the hearts of the uninitiated but, for the most part, they are bland and quite obvious. Apparently, the arch-enemy of the disk drive and diskette is dust. We are told that we mustn't bend them, wash them, stick our jammy fingers on the mylar surfaces, write with sharp pointed pens, leave them lying about outside their cardboard wallets or (the most heinous crime of all) allow magnetic fields near them.

Winchester hard disk drives

At the time of writing, no enterprising manufacturer, to our knowledge, had come up with a method of interfacing a Winchester to a Commodore 64. However, bearing in mind the rapid advances now being made in the computer add-on market, it is quite possible that by the time this book reaches the shelves, some firm will announce such a device. Computer filing systems rely heavily on data storage so, at least, a superficial treatment of the Winchester is justified in any book on the subject. It is, after all, the state of the art storage device.

Historically, the hard disk came before the floppy disk. The pioneers, and the trend-setters of disk technology were, and still are, the giant multinational corporation, IBM. The hard disk was intended to complement, and perhaps later replace, the large digitally controlled tape transports which were the storage work-horses. (Tape transports should not be confused with the humble cassette tape unit.) Hard disks rotate continuously at about 3000 rpm. Unlike the floppy disk, they rotate constantly, whether or not they are actually reading or writing data. The increased rpm of the hard disk and the fact that there is no start-up inertia to overcome means that data can be accessed at high speed.

The disk material is a metal alloy of aluminium and is therefore literally 'hard'. Large mainframe computer complexes use disk units containing eight double-sided disks stacked on the same drive shaft. The amount of information which can be stored on a hard disk unit is breathtaking - 1000 megabytes is quite common. Few home computer users would ever get past the wistful dreaming stage of owning one of these marvels. Fortunately, there are scaled down versions of the traditional hard disk known as the Winchester Disk Drive or simply the 'Winchester'. These are sealed units containing one, normally non-interchangeable, hard disk. They can store several megabytes. They are still very much more expensive than floppy disks although prices have recently begun to fall a little. Those who like snippets of useless, but intriguing, information may be fascinated to learn that the name 'Winchester' resulted from the similarity between the manufacturer's original type number and the bore of the famous Winchester rifle. At least, so goes the tale.

Memory

As already mentioned, the semiconductor RAM chips constitute the computer's memory. Although obvious to many readers, it is well to emphasise that although programs can be stored on cassette or disk, the instructions, which constitute the program, can only be executed if they are resting in RAM. The Commodore 64 computer has 64K of RAM but new owners soon discover that not all of this is available for use. Unless certain steps are taken (described later) only 32K is available for programs and data. However, this is quite enough for running quite sophisticated filing systems.

Transferring tape to disk
It is inevitable that the first job, after unpacking and setting up the 1541 will be to transfer important tape programs to disk. The procedure is simple:

(1) Load in the program from cassette
(2) SAVE"name",8

The default storage device on the Commodore 64 is the cassette tape unit but the disk unit is normally wired up to appear on the IEE-448 bus as device number 8 which is why we have to place comma 8 after the program name. Similarly, when loading from disk, we write:

```
LOAD"name",8
```

Summary

1. A filing system should have high storage capacity and fast access to any particular item.
2. The standard internal memory, before semiconductor RAMs were introduced, was the magnetic core memory.
3. A non-volatile internal memory is one which retains stored information after the power is interrupted.
4. Semiconductor RAMs are volatile.
5. New disks, which have no prerecorded sector and track markers, are called 'soft-sectored'.
6. Floppy disks were originally 8 inches in diameter but the 5¼ inch diameter 'diskette' is the most popular for microcomputer use.
7. The floppy disk is random access by track but sequential access by sectors within track.

8. The 1541 disk DOS formats 35 tracks on a disk. The outermost track is Track 1; the innermost, Track 35.
9. The outer tracks have more sectors than the inner tracks in order to average out the recording density.
10. All sectors can store one block of data, which is 256 bytes.

Self test

1.1 What advantage did the old magnetic core memory have over the modern semiconductor RAM?
1.2 What is the difference between memory and store?
1.3 What is the connection between Kansas City and the cassette recorder?
1.4 How many characters, ignoring interblock wastage etc., can be stored on a C12 tape if the baud rate is 600?
1.5 What do the initials DOS stand for?
1.6 What is the difference between storage and memory?
1.7 The Commodore 64 uses a modified version of the IEE-488 bus. What is the major difference between the modified and the official version?
1.8 How many characters are there in a block?
1.9 How many blocks can be stored on one sector in the 1541?
1.10 Does the term 'single' density refer to the number of tracks or the number of sectors per track?
1.11 On which track is the Directory held in the 1541?
1.12 Which came first, UNIVAC or EDSAC?

Chapter Two
Components of a Filing System

File layout

Science has always emphasised the importance of classification. In fact, science could be defined as the orderly arrangement of ascertained knowledge. We are not, at the moment, concerned with the difference between 'ascertained' knowledge and information except to point out that information can, at times, be false. As far as file organisation and arrangement is concerned, however, it is unimportant whether any particular item in the file contains true or false information. The accuracy of the information held on file depends on those responsible for its administration and daily updating. Nevertheless, the design of a filing system should always take into account the possibility of false entries and ensure that simple errors are easily detected at input level and, equally easily, corrected.

It should be realised that programs which organise and process information are little more than tools. The practical value of a computerised filing system depends largely on the manner in which the file information is classified. The computer will not classify it. A good computer program will normally allow the user considerable freedom in the way the file information is structured but, ultimately, it is the user's responsibility to decide the manner in which that freedom is to be exercised. For example, the name given to each heading in the file, the material to be included, the number of different headings, the space allowed for each item of text under a heading, the abbreviations to be used, the particular heading which is to be considered more important than other headings – all these points and many others must be considered carefully before deciding on the initial file layout. It is essential to adopt a long-term attitude during the planning stage. It would be a nuisance, perhaps a disaster, if a file has been in operation for some time and it is suddenly

discovered that an extra heading should have been introduced during the initial planning stage. It takes time and energy to keep a file updated with new or corrected information so to scrap it and re-enter it all again in a different format would be an unpleasant and time-wasting exercise.

It is possible, of course, to prevent such a mishap from occurring by simply creating a spare heading to be left blank until, or if, needed. This is all right if there is plenty of room in the store but it is little more than a cover-up for a potential problem which should not arise if the original file were more carefully planned. It is also possible to arrange a filing program allowing extra headings to be added, or deleted, at any time but this would entail adding yet another dimension of flexibility. Indeed, a file handling program could be designed to cater for any defects in the user's judgment but the program could eventually assume unwieldy proportions and eat deeply into available RAM space.

User-friendliness

'User-friendliness' is a controversial subject so it is not surprising that so much has been written about it. To some, it is almost a cult. Others, whilst recognising that lip service must be paid, feel that its value is grossly overrated. The following is intended as a starting point for discussion rather than a formal definition:

> User-friendliness is a measure of a program's ability to recognise human weakness.

The extra program lines needed for adding a degree of user-friendliness is often called *idiot-proofing*. We are all capable of idiotic behaviour at times so we should not view the term 'idiot' as offensive. The essential issue here is to decide how much idiot-proofing is justified in a filing program.

There are several ways of introducing it but the following arrangement is typical:

(1) The computer asks for some input.
(2) The operator enters it.
(3) The program examines the input and either

   (a) accepts it, or
   (b) rejects it and asks for input again.

The process is repeated indefinitely until the input satisfies the computer. A typical input validation procedure is shown in the following example.

Example:

```
100 INPUT"ENTER NUMBER OF RECORDS";N%
120 IF N%<1 OR N%>500 THEN 100
```

The upper limit of 500 is, of course, arbitrary. This is only partial idiot-proofing because it does not cater for the type who doesn't know why the input has been rejected. A message is needed from the computer to explain why. The program segment now grows a little:

```
100 INPUT"ENTER NUMBER OF RECORDS";N%
120 IF N%<1 OR N%>500 THEN PRINT"NUMBER
MUST BE IN RANGE (1 TO 500)":GOTO100
130 PRINT"NUMBER ACCEPTED"
```

Another aspect of user-friendliness is the incorporation of 'Are you sure? Answer Y/N' messages. These are issued in circumstances where the wrong decision on the part of the operator could cause serious trouble. For example, the original question might have been 'Do you want this record to be erased?'. If the operator, in an unguarded moment, answers 'Y' (but intended 'N') a valuable record might be lost for ever. This could be catastrophic so the further message, 'Are you sure?', would help to reduce such a risk. To provide an added safeguard, it is wise to get the program to ask for one particular key to be pressed for the more dangerous alternative while otherwise accepting any of the others.

The concept of user-friendliness covers a wider field than simple idiot-proofing. Simple and unequivocal screen messages and wise choice of colour for emphasis purposes are also important. Colour for its own sake is not justified even though the Commodore 64 offers a wide range of colour. Bizarre colour effects in games programs are useful as cosmetic aids but, in more serious programs, colour should be used only in cases where it can improve or add force to the screen appearance.

Certain aspects of user-friendliness can be irritating. To start with, we could ask for which particular user the program is intended to be 'friendly'. A program which, by virtue of copious warning messages, might be considered 'friendly' to a novice operator could be absolutely infuriating to a skilled operator. It takes some time for any operator to learn how to make use of all the facilities offered by a

sophisticated program so, initially, the user-friendliness may be appreciated. But the same novice may eventually become skilled and will begin to experience irritation if too many warnings have to be answered with 'Y / N' entries. Another reason for not programming in too much user-friendliness is the cost in RAM space. It is easy to go over the top in trying to cater for all possible forms of human frailty. The result could eventually lead to a situation in which the desire for friendliness is satisfied only by sacrificing useful options. The program examples in this book are 'moderately' friendly but, because full use has been made of program *modules* (subroutines), it would be easy for anyone to add extra input traps without endangering the overall program structure.

Crashing the program

When a program ceases to be under the control of the operator it is said to have *crashed*. This is a pretty wide definition because there are various ways in which a program can be crashed. A crashed program is always annoying because it means starting again with a RUN or even a re-load from store. In the case of programs which handle large amounts of data, a crash can be annoying, particularly with RAM-based files, if it occurs near the end of a long data-entering session. The steps taken in the program to prevent an operator crash is yet another aspect of user-friendliness.

Documentation

All programs, except the most simple, deserve some form of support documentation explaining how they can be used effectively. Good documentation is particularly necessary in filing programs because they normally offer a wide range of processing options which can, for the first-time user, appear bewildering. Good documentation also contributes to user-friendliness by reducing the dependence on screen messages. Too much screen information, although initially providing valuable prompts, rapidly degenerates into useless clutter as an operator becomes more experienced. To be constantly reminded on every display page that we must 'Press Key X to return to Options' can soon become irritating. Besides, it is wasting the screen area. Information of this kind is far better given in supporting documentation.

Some preliminary terms and definitions

Up to this point, we have been using terms like 'information', 'data', 'file', etc, in their everyday sense. We have now reached the stage where, for distinguishing purposes, more formal definitions are needed.

Data

Of all terms in the repertoire of computer jargon, 'data' is probably the most overworked. In the general sense, data could be defined as anything which has meaning to the computer. This definition is too all-embracing for our purposes. What we need is some rule whereby we can distinguish 'data' from other closely related terms. We shall use the following simple definition:

> Data is information which can be held in store for access by a suitable program.

When the data is loaded from store into RAM it is, of course, still 'data' but in the general sense only.

Files

Like data, the term 'file' is also an overworked term and often used with a variety of meanings. For example, it is customary in user handbooks to call everything that is stored on tape or disk as a 'file'. They refer to 'Program files' and 'Data files'. In this book, we shall define a file in the following way:

> A file is a collection of related data held in store and accessed by a program in RAM.

For example, the data in a file could consist of various snippets of information on birds. Another file could be a simple list of customers' names, addresses and telephone numbers.

As explained earlier, the choice of material to go into one file is a human decision. Whether or not the contents of the file are indeed 'related' must depend on the judgment of the person setting up the file.

In accordance with our previous definition of data, it follows that the term 'file' assumes that it is a data file. As most readers will be aware, BASIC programs are stored in a special format, using *tokens*

instead of BASIC keywords, whereas data files are stored in simple ASCII characters.

Record

Although a file contains information which is in some way related, it is still hierarchal in form. That is to say, it will consist of a set of individual 'records'. For example, a file on customers' names and addresses will contain separate records for each customer. We can therefore define a record as follows:

> A record contains all data relevant to one particular entry.

For example, a file on Birds of Britain could contain a number of records, one for each different bird.

Field

A 'field' is to a record as a record is to a file, so the following definition is appropriate:

> A field is a subdivision of a record and represents one aspect of the total information in a record.

Taking, as an example, a file on Volcanoes of the World, one field might give the name of the volcano, another the height and another its location. The number of characters set aside for each field is known as the field-width.

Field headings

All fields must have a *heading* in order for the data to have meaning. For example, if the number 5.67 appeared in one field of a record, it would be meaningless without a heading. A field heading of, say, 'Height in metres' makes the number meaningful. In the interests of programming efficiency, it may be important to distinguish between fields which carry *numerical* data exclusively and those which carry alphanumeric or *string* characters. During the setting up of a new file, the operator can be asked the kind of data to be entered under each field heading – string or numeric? It is easy to assume that all fields are in string form and make the computer sort it out, but it is more efficient for numeric fields to be distinct from string during keyboard entry. This is particularly so if integers, instead of floating point numbers, can be used.

The importance of the key field

Although all fields of a record will contain important information, one of them will enjoy higher status than the others. It is called the *key field* because it is used to identify the record. Which particular field is to be chosen as the 'key' is the responsibility of the person who initially sets up the file. However, because the key field is the record *identifier*, it is essential that it be absolutely unique. In our example of the volcano file, the choice of key field would clearly be the volcano's name, on the grounds that no two volcanoes would share the same name. On the other hand, the volcano's height would not always be unique. A problem could arise in some files, particularly where the key field is the customer's name. It is quite possible for more than one BROWN A.G. to be present in the same file. If this happens, an extra identifier could be added – perhaps BROWN2 A.G. The safe way, particularly in larger files, is to use a code identifier as the key field (perhaps ten or more characters) rather than a person's name. Identity codes are dehumanising and offensive to many people but, because they are necessary for the administration of an over-populated society, should not be regarded as Orwellian.

Files concerned with inventories of equipment will almost certainly have a part number as the key field of each record. A typical electronic components firm may have a separate record for each component stocked. The number of different components could run into tens, or even hundreds, of thousands. It would be a long and unwieldy business trying to find a unique physical description for each key field so a part number is the obvious solution. Although we have used an industrial example, it is not uncommon to find large inventories in homes, particularly if one of the residents is an ardent collector. For example, a stamp collection of 10000 specimens would be considered quite moderate in size by any amateur philatelist and a computer filing system would provide an extra dimension to the hobby. Providing the collection is organised properly on files, the computer can be used to find unexpected relations between groups of specimens. Again, it may be more convenient for the key field to be a stamp catalogue number rather than a description.

In spite of the advantages of numerical or coded key fields, they are inclined to be user-unfriendly. 'Penny black' is easier to remember than, say, '23578642' when searching for the record. To combine the advantages of a coded search key with the friendliness of a literal key,

an ingenious dodge known as *indexing* can be used, the techniques of which will be given in a later chapter.

Figure 2.1 illustrates the relationship between the field, the record and the file.

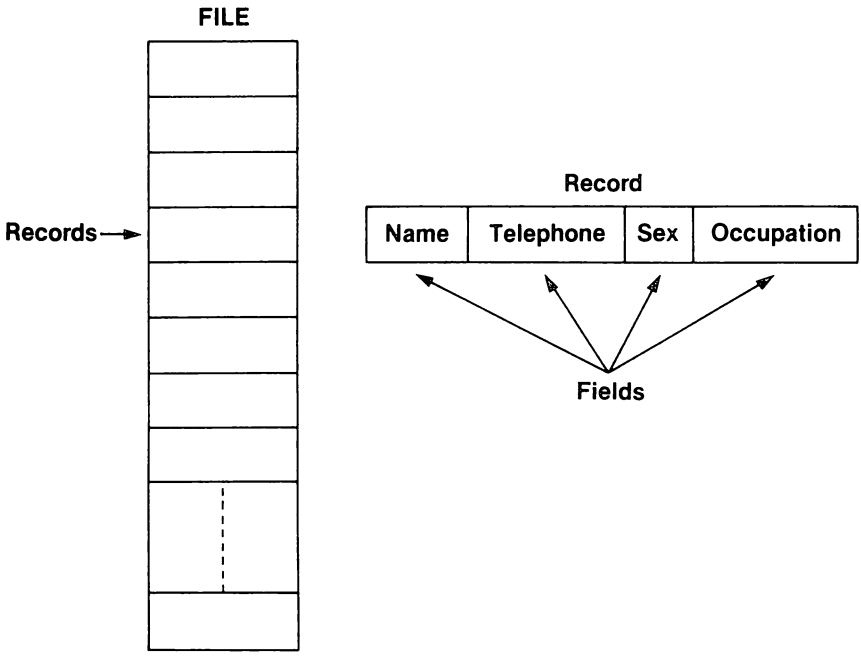


Fig. 2.1. Components of a file.

File size

Various factors can influence the information grouped under one file heading, including personal preferences. Over and above this, there will be physical limitations imposed by the memory and storage system on the maximum number of bytes allowable in a single computer file. These limitations will depend to a large extent on the methods employed in the program. There are several recognised methods of organising filing programs and we shall be dealing with some of them in due course. From the point of view of file length, it is sufficient to distinguish two broad classes of file:

(a) *RAM-based serial files:* the entire file must be loaded from store into RAM before records can be accessed.

(b) *Store-based files:* the file remains in store and only the chosen record, or in some cases a few records, are accessed as required.

From this, it is clear that the length of a RAM-based file is limited solely by the amount of space left in RAM over and above that used by the program. A program with extensive file processing abilities could bite deeply into the RAM space, thereby forcing a compromise between processing facilities and file space. If we assume that 100 characters (including control characters) are required for a typical record, the maximum file length could not exceed 100 records. There will be a natural desire to cram as much information as possible into each record by having lots of fields and space for characters in each field. The following equation, for estimating the size of the file in bytes should act as a sobering influence during the initial planning stage:

Total bytes = Field width × number of fields × number of records.

It is not always appreciated that seventy characters, including spaces, are required for the average name, address, post code and telephone number.

With store-based files, the amount of available RAM is less important because the limiting factor on file length is the amount of on-line storage available. It was mentioned earlier that the 1541 disk drive has storage space for 664 blocks (169984 bytes) after the Directory and BAM have taken their slices. Assuming again that 100 characters are allowed for each record, the number of records we could squeeze onto one diskette is approximately 1700. The largest practical number of characters possible in a single record is about 250, so this would limit the number of records to about 700. Apart from the increase in maximum file length, a store-based file allows more RAM space for the program so the number of options for processing the file need not be so drastically curtailed as they would have to be in RAM-based files.

In spite of some disadvantages, it should not be thought that RAM-based files are entirely useless. In cases where the length of the files are well within the capacity of RAM, there is a distinct advantage in using them. Once a complete file is loaded from store, the essential processes are completed at RAM speed. Sorting records into some kind of order is the one process at which RAM-based files excel. It is possible, but relatively slow and inconvenient, to sort records unless the entire file is resident in RAM. We shall see later

that there are filing systems whereby the main file is kept in store but a relatively short *index file* is operated from within RAM. This combines the file size advantage of store-based with the processing advantage of RAM-based files.

File splitting and merging

The number of records under one file heading can eventually become too large to fit into RAM at one go. This can be overcome with an option for splitting a single file into one or more separate files. For example, if the key field is the name of something, it may be advisable to split the file into names beginning with A to J, another beginning with K to R and another with S to Z. Conversely, it might be useful to have a facility for *merging* two or more smaller files into one.

Processing options

The outstanding advantage of a computer resides in its ability to process data in a variety of ways. The manner in which the various processes are chosen and activated determines whether the program is defined as *menu-driven* or *command-driven*.

A menu-driven program revolves around a display referred to as the 'menu page' which contains the list of options available. The option required is chosen by entering the corresponding option number. If there are less than ten options, the GET function can be employed to avoid the operator having to press RETURN afterwards. Whatever option is active, the menu can be regained by pressing a particular key. A command-driven program has no menu page. Instead, there is a line, or perhaps two lines, reserved at the bottom of all screen displays known as the *command* line. This will normally display a prompt such as 'What next?' followed by a flashing cursor. The particular option required is entered in the form of a single letter, or letter group, chosen by the programmer for mnemonic association with that option. It is up to the operator to memorise the option letters although one of the options may be 'H' (meaning Help) which displays all the options in full.

An experienced operator would probably find little to choose between a menu- and command-driven program. One advantage claimed of a command-driven program is the comfort to be gained from seeing the prompt, the current option and the flashing cursor present at the bottom of all screen displays. However, the menu-driven program has distinct advantages. First, there is no need to

memorise option mnemonics as the menu page is displayed in non-abbreviated form. Second, there is no wasted line, or lines, at the bottom of each screen display. The programs in this book will all be menu-driven. However, it should not be too difficult for anyone who prefers a command-driven system to rewrite the display procedures accordingly.

It is often convenient to split the menu into two levels. The first level is for choosing a *primary* option such as 'creating a new file' or 'accessing an existing file'. The lower level is for selecting one of the many options available for processing an existing file. Some typical processing options will now be described. They should be interpreted in the general sense because variations can be expected in practice.

Create new file

The procedure for creating a new file will consist of a series of prompts, inviting the operator to enter pertinent information on such things as:

- File name
- File size (maximum number of records expected)
- Number of fields
- Field headings
- Field width
- Field data (numeric or string)

Access existing file

If a file exists already it is only necessary to choose one of the following *secondary* options.

Enter record
This is used for:

(a) entering an extra record to a file which already exists, or
(b) entering the first record in a newly opened file.

Display record
There are several ways of displaying information, depending to a large extent on the method used to organise the file. They can be classified as follows:

(a) *The broad-sheet display*
All the fields of each record are presumed to lie along one horizontal line which stretches beyond the visible screen area. The key field remains *permanently displayed* at the left and the other fields are scrolled into view as needed by means of pressing arbitrarily chosen keys. The advantage of this display is in being able to view many records simultaneously. Although only a part of each record appears at a time, as many records as there are lines available can be viewed at the same time. Other arbitrarily chosen keys are used to scroll in another block of records.
(b) *Single record display*
A single record occupies the screen with each field commencing on a new line. If the record is too long for complete display, it is usually arranged to display a screen full of data at a time.

Delete record
The ability to delete a record is one of the facilities which might justify the use of an 'Are you sure?' message. After deletion, a 'hole' will be left in the file unless the program includes a routine to close up the following records. Some programs will insert a special marker, known as a *tombstone*, in the slot formally occupied by a deleted record. The tombstone is used to indicate to the program that the next record to be entered can be placed in the tombstone's slot.

Modify record
Records are often incorrect. They may have been entered incorrectly or later became incorrect because of changing circumstances. Consequently, all filing programs should have an option for correcting part, or whole, of a record.

Search for record
Where it is practical, a filing program will ensure that any record or group of records can be located and recovered, which satisfies one or more search criteria. In fact, the search function alone is more than sufficient to justify the use of a computer for filing information. The normal procedure, explained earlier, for finding a particular record is to quote the specific key field indentifier. For example, the key field in a file on World Politicians would be the name of the particular politician. A search by key field will always be the fastest method. However, there may be many reasons why a record cannot be found in this way.

For example, you may not be quite sure of the politician's name

but have a feeling that the letters 'eag' appear in his name. This is where the option, 'Search for substring' can be used. To find the elusive politician, the substring 'eag' can be entered under the field heading 'Name'. This will bring out records of all politicians whose name contains this substring, so the vital statistics of Reagan R. might suddenly appear on the screen.

Another possible search requirement would be for records in which information in one of the fields lies within some limit or limits. For example, in a file on tropical diseases, we may want to find those which have an incubation period in humans of less than 14 days. In another file on bipolar transistors, we may want to examine the records of those with a forward current gain (Hfe) greater than 100 but less than 130.

A list of search options might appear as follows on the menu page:

1. Search any field for <> n
2. Search any field for = n
3. Search any field for >= n
4. Search any field for < n
5. Search any field for >= n and < m
6. Search any field for substring

Example 1
If we ask for option 3 above, the program might then ask for the following information:

1. Field heading? (Assume we answer volts)
2. Enter number? (Assume we answer 3)

The search will then bring out all records with a value equal to or greater than 3 in the volts column.

Example 2
If we ask for option 5 above and we answer as follows:

1. Field heading? (Assume we answer volts)
2. Enter number? (Assume we answer 5)
3. Enter number? (Assume we answer 15)

The search will then bring out all records with a value in the volts column greater than or equal to 5 and less than 15.

The two examples have been concerned with searching for numerical limits. Some programs work equally well if the limits are given in character form.

Example 3
If we ask for option 4 and answer as follows:

1. Field heading? (Assume we answer name)
2. Enter string? (Assume we answer 'G')

The search will then bring out all records where the name begins with any letter 'earlier' than G.

Sort records

The ability to sort records into order is often considered an essential element of any good filing system. An ordered system always has more *information value* than a disordered system. A special technique, known as a *binary search*, enables a search to be conducted much faster than a simple linear or *sequential* search. However, in order to use a binary search, all records must be in key field order.

It is easy to sort records if the entire file can first be loaded into RAM but not so easy if the records are store-based. Even in RAM, sorting is always a relatively slow process in BASIC, so, if a sort option is necessary on a large file, it is better that it be programmed in machine code.

Subfiles

Some programs offer the facilities of a 'subfile', containing all records which satisfy a certain search criterion. The subfile can then be stored separately under a new file name as illustrated in Fig. 2.2.

This can often be used to split a file which is beginning to show signs of excessive bulk. The question of where to split can be decided sensibly instead of by a crude alphabetical split. A file on Mountains of the World could be split into separate files according to continent or perhaps height above sea level. This would have greater information value than a file split by mountain names into 'A to G', 'H to R', etc. In fact, a filing system which starts life as a hotch-potch of disconnected data can, by careful splitting into subfiles, develop into a highly organised information source. It is worth mentioning that all information is data but not all data is information!

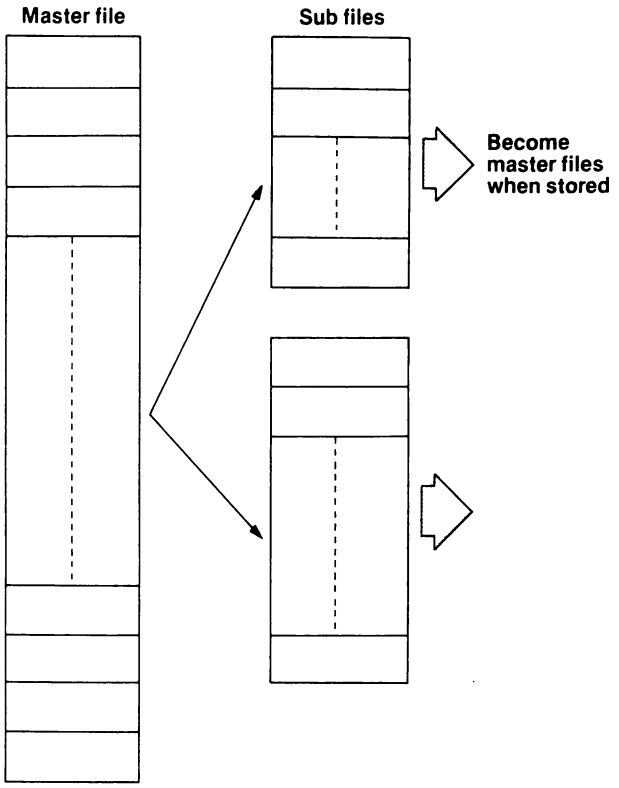


Fig. 2.2. Splitting files.

Printing options

Printers are not standardised. This makes it difficult to program an option for sending selected file data to a printer. The early printers were simple affairs, capable only of printing upper- and lower-case characters in one typeface but this era is now past. Most modern printers are able to deal with italics, underlining, enhanced and double-width characters with variable line spacing and some even produce a choice of colour and right-justified text. Unfortunately, the control characters for activating such extras vary with different printers. Of course, if a standard Commodore printer is used, there is no problem but printers are not yet cheap enough to be discarded for a new one every time the computer is changed. We should mention that there are interface leads available for connecting one of the more common Centronics standard parallel printers so it is not necessary

to rely on Commodore exclusively. Because it is not possible to produce general purpose routines, compatible with all printers, our program listings will exclude them altogether. They can easily be inserted as a subroutine to suit your particular printer if required.

File organisation

The list of common file processing options we have described above is all-important to those who just want to use the program. They may not, or indeed need not, know how the file has been programmed or which particular *file organisation* has been adopted. Although complete program listings are given in this book, the aim is to encourage readers to consider these only as a guide towards writing their own versions. The emphasis, from now on, will be shifted away from the user and more towards the programmer.

There are several, well recognised, ways of organising a filing system. Which method is chosen depends on:

(a) the type of storage equipment;
(b) which options are considered to be of overriding importance;
(c) the amount of on-line storage capacity.

With regard to (a) above, cassette tape is not really practical in any way other than for RAM-based files. This calls for a definition:

> A RAM-based filing system is one in which the entire file is loaded into RAM before records are accessed or processed.

If a file requires updating, it must always be loaded into RAM, the modifications made and rewritten back to store.

On the other hand, *store-based* is defined as follows:

> A store-based filing system is one in which the complete file remains in store and only the required record (or sometimes a few records) is transferred to RAM as required.

Figure 2.3 shows the differences between them.

Store-based files are usually classified as either *sequential access* or *direct access*. These are explained later in this chapter.

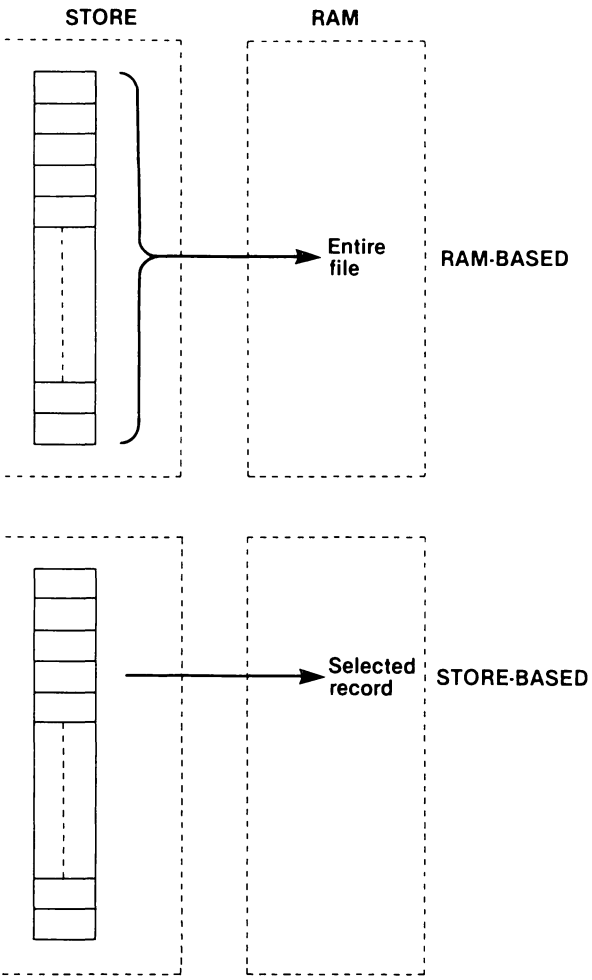


Fig. 2.3. RAM- and store-based files.

RAM-based files

These files can be very useful, even on disk. Instead of having one large file, a 'multifile' system can be used containing many small files of related data. The idea of multifiles will work, regardless of whether cassette tape or disk is on line.

There are a number of ways of organising a file on the Commodore 64, each with advantages and disadvantages. These should be taken

into consideration before deciding which organisation is more suitable for the task in hand.

Sequential access files

All those who use a cassette tape unit for storage purposes will be quite familiar with sequential access files (programs are files in this context). It is implicit in the following definition why it is the only practical filing system possible with tape:

> A sequential access file is one in which each record is serially read into memory until the required record is located.

The hardware design of a tape unit enforces sequential access. In theory, of course, it is possible to use the fast-forward or reverse buttons in conjunction with the tape revolution counter (if there is one) to attempt to locate directly a particular record. It would, however, require superhuman powers, even with the aid of appropriate software. It is not too difficult to locate a particular 'program file' by this method but to locate a particular record within a 'data file' would be a daunting task. The unpleasant feature of sequential access is the time-wasting process of reading unwanted records which are in front of the wanted record. The access time is not constant because it depends on the position of the record in the file. The average access time will be half the time taken to reach the last record. This can be quite long, even with 1200 baud tape operation.

A disk-based sequential access file is formed as follows:

(1) A file is opened for output.
(2) The first record is entered at the keyboard and immediately stored on disk.
(3) The next record is entered and sequentially stored (appended) after the first. This procedure continues until the last record is stored.
(4) The file is closed.

To recover a record, the records are sequentially read into memory until the required one is located.

The main advantages of sequential access files are as follows:

(a) Storage of differing length records is possible.
(b) It is economical in the use of storage space.

(c) It is the simplest file organisation to manage.

The disadvantages are:

(a) It is practically impossible to select a particular record within the file and modify it directly.
(b) It is not possible to *insert* records physically within the body of the file.
(c) Access time depends on the record's position in the file.
(d) Records cannot normally be added to the end of a file at a later date.

However, it is possible to overcome disadvantages (a), (b) and (d) by periodically using the following method:

(1) Open the existing file for input to the computer.
(2) Open a new update file for output from the computer.
(3) Serially read single records into memory.
(4) Modify or delete selected records as required.
(5) Write records sequentially to the update file.
(6) Add any new records to the end of the file (append file).
(7) Close both files.
(8) The old file is deleted but its name is retained for the update file ready for the next time the sequence is repeated.

Overall, it is fair to say that disk-based sequential files are not popular with microcomputer users. Sequential files are more suited to large mainframes and batch processing where facilities exist for overcoming some of the disadvantages described above. For example, it is possible to update or insert records by using multiple tape drives in which transaction files are used in conjunction with master and update files.

Direct access files

Direct or random access files can be defined as follows:

> A direct access file is one in which any record, whatever its position within the file, can be immediately located without the need for any sequential searching.

This type of file has become increasingly popular with microcomputer users. Although normally more difficult to program, direct

access filing systems are particularly easy to implement on the Commodore 64. This is because they have developed, within their disk operating system, a special version known as 'relative files'. Commodore's relative files have the following advantages:

(a) Access time is almost independent of the record's position in the file.
(b) Any record can be updated. An updated record does not necessarily have to be the same length as the original, providing the maximum length, set at the file creation stage, is not exceeded.

There are some disadvantages:

(a) Under-utilisation of allotted record space leads to waste of storage capacity.
(b) All records must be of the same length or padded to make them so. With relative files, the DOS, fortunately, does this padding out automatically.

Summary

1. A 'user-friendly' program can never be user-friendly to all users.
2. A computer-rejected input does not always justify an explanatory message.
3. The 'are you sure' response can be useful or irritating. It depends on the calibre of the operator.
4. It should be impossible for an inexperienced operator to crash a program.
5. It should always be possible to escape at any time back to the menu.
6. Good supporting documentation is an important contribution to user-friendliness.
7. A data file cannot be directly 'SAVEd' or 'LOADed'. It can only be accessed from within a program.
8. A record is a component of a file.
9. A field is a component of a record.
10. The field width is the number of characters in a field.
11. The field heading gives the meaning to be attached to the information within the field.
12. String fields are those which can contain any character (except control characters).
13. Numeric fields contain only numbers, either integers or floating point.

14. The key field is the one which uniquely identifies the record.
15. Numerical key fields are favoured in large files.
16. The file size refers to the number of records it holds.
17. The space a file occupies in store depends on both the file size and the field sizes.
18. If the complete file is processed from within memory, it is said to be RAM-based. The file size is then limited by the RAM capacity.
19. If an individual record can be picked out of store and processed, the file is said to be store-based. The file size is then limited by the store capacity.
20. A diskette on the 1541 drive unit can hold about 1700 records if the number of characters per record is limited to 100, or about 700 records of 250 characters each.
21. Menu-driven filing programs revolve around an option page.
22. Options are chosen in command-driven filing programs from a 'command line' at the bottom of all displays.
23. A tombstone marker signifies a deleted record.
24. Sort routines are easy if the file is RAM-based but there are problems if store-based.
25. Subfiles contain selected records within master files. When stored, they become master files in their own right.
26. Print routines can only exploit the full potential of a printer if they are purpose-designed for that model.
27. Store-based files are classified into sequential access or direct access on the Commodore 64.
28. The required record in a sequential access file is searched by starting at the first record and continuing until the required record is located.
29. Cassette tape files are always sequentially accessed.
30. Sequential access is efficient in storage capacity but slow.
31. Any record in a direct access file can be accessed without searching through unwanted records.
32. Direct access files normally have all records the same length.

Self test

2.1 Can a data file be stored by using SAVE"name"?
2.2 What is the distinguishing feature of the key field?
2.3 Under what circumstances are numerical or coded key fields preferable?

2.4 State an important advantage of a RAM-based file.

2.5 A file containing 100 records, each of 5 equal length fields, occupies about 50K bytes of storage. How many characters in each field, ignoring overheads?

2.6 In a broadsheet display, in which direction are the fields of a record displayed?

2.7 Under what circumstances would the option 'Search any field for substring' be used?

2.8 The facilities for splitting off a subfile are particularly useful in RAM-based files. Why?

2.9 The time taken to access a record from a sequential file is dependent on the position within the file. True or false?

2.10 The records should be in key field order in a direct access file in order to shorten the access time. True or false?

Chapter Three
Simple Filing Programs

The advantage of subroutines

The programs in this book assume that the reader has already gained some experience of the Commodore 64 version of BASIC, particularly in the use of *subroutines*. It is sometimes thought that subroutines should only be used if the functions they perform are needed more than once in the same program. If we take a narrow view of efficiency by taking it to mean achieving the maximum effect with the minimum number of programming lines, such a belief is justified. However, efficiency can also mean reduction in programming time, not only in the initial stages of writing and debugging but when modifications or additions are required in the light of user experience. If this interpretation of efficiency is taken, then subroutines should be used for practically every logically distinguishable function, even if they are only used once in the program. A subroutine is, or should be, a tight, self-sufficient, black box with *one input* and *one output*.

The user manual supplied with the machine should be consulted if there is any difficulty with syntax since, in order to save space and to avoid repetition, only aspects of the language specifically concerned with data files will be treated here. An attempt has been made to follow the teachings of structured programming, at least as far as the BASIC version will allow. Unfortunately, the much-maligned GOTO cannot be avoided altogether. Even a subroutine is, intrinsically, a GOTO. However, it is not a serious crime to use GOTOs within a subroutine, providing that the jump destination remains within its bounds. To jump out of a subroutine before its normal exit, even if it is arranged to arrive back before the normal RETURN, is an offence against the whole idea of structure. If a subroutine has only one input and only one output it is easy to follow the program and arrange modifications without causing trouble.

Pretty displays
The temptation to glamorise displays by adding borders and various colours has been resisted. A program with an attractive display is all very well but it adds to the complexity of a program. The main objective of this book is to help you to understand how filing programs work and how to tailor them to suit your own special purposes. This objective is endangered if the main features of a listing are obscured by packing with graphic symbols and colour information. In any case, it is easy to add these touches afterwards.

Variable names
Bearing in mind that we are restricted to two characters for naming variables, the variables in the program listings throughout this book will be given meaningful names in order to cut down on the number of REMs. As far as possible, the same variable names will have identical functions in all programs. The explanations, which will accompany each major program, will include the meanings of the more important variable names and their functions. This will involve some repetition, but it makes it a little easier to follow the program listings.

Program-based files

In Chapter 2, we listed the file organisations in common use. However, one of them was deliberately left out because it is questionable whether or not it should be considered a true filing system. For want of a better name, we shall refer to it as a *program-based file* because the file is held in DATA statements, and under line numbers, within the program itself rather than stored separately. (We could have called it a 'DATA-based' system but this could be confused with a 'database'.)

Because the file information is written into the program, only a 'programmer' can alter it so the range of options is severely limited. Nevertheless, such an elementary arrangement is excellent for getting a preliminary feel of the subject and well worth examining.

Program to display DATA

It is always best to start right at the bottom, so study the listing of Program 3.1 which, as you will see, is most certainly rock-bottom.

```
10 REM DISPLAY DATA STATEMENTS
20 READ FS%,NF%
30 DIM A$(NF%,FS%)
40 READ H1$,H2$
50 REM READ DATA INTO ARRAY
60 FOR R=1 TO FS%
70 FOR F=1 TO NF%
80 READ A$(F,R)
90 NEXT:NEXT
100 GOSUB150
110 END
120 REM *
130 REM **
140 REM SUBROUTINE VIEW FILE
150 PRINT CHR$(147)
160 PRINT TAB(7)"FILE CONTENTS"
170 PRINT"------------------------------"
180 PRINT TAB(3)H1$ TAB(19) H2$
190 PRINT"------------------------------"
200 FOR R=1 TO FS%
210 FOR F=1 TO NF%
220 PRINT"   "A$(F,R) TAB(18);
230 NEXT
240 PRINT
250 NEXT
260 RETURN
270 REM *
280 REM **
290 REM NUMBER OF RECORDS AND FIELDS
300 DATA 8,2
310 REM *
320 REM **
330 REM FIELD HEADINGS
340 DATA NAME,TELEPHONE
350 REM *
360 REM **
370 REM FILE INFORMATION
380 DATA DEREK,6668
390 DATA GILL,3467
400 DATA KATH,3333
410 DATA STEVE,666
420 DATA JOHN,2357
430 DATA PAUL,1104
440 DATA GRAHAM,1456
450 DATA PAT,7864
```

Program 3.1. Display DATA statements.

The program has a simple objective. It just grabs the records written as DATA statements and presents them to the screen. In spite of this, you are urged to key the thing in because it shows how to present the information in a *two-dimensional array* onto the screen in *rows* and columns. Use it as a guinea pig by altering some of the lines and noting the effect.

List of variables used in Program 3.1

FS% = File Size = number of records in each file.
NF% = Number of Fields.
H1$ = Heading of field 1.
H2$ = Heading of field 2.
F = a particular field.
R = a particular record.
A$(F,R) = an element of the two-dimensional array holding the main file data.

Some explanation is called for in respect of A$(F,R). It is common, in such a file array, to use the variables the other way round, R (record number) first and F (field number) last. Normally, when programming entirely in BASIC, it does not really matter whether an array element is specified by A$(R,F) or A$(F,R) as long as consistency is observed. However, a two-dimensional or rectangular array is stored in the Commodore 64 in *column major* order. That is to say, the sequential storage sequence of string pointers in memory is A$(1,1) A$(2,1) A$(1,2) A$(2,2) A$(1,3) A$(2,3) ... and so on. Therefore, if we wish to ensure sequential storage of field string pointers, relevant to each record, then the form A$(F,R) is preferred. This storage organisation favours fast and efficient methods of sorting a two-dimensional array file using machine code.

The DATA statements of Program 3.1 constitute the file *information* and are set at the bottom of the listing. Because they are down at the bottom, they stand out well if, at some later date, they are to be altered. The example file consists of 8 records, each of two fields, giving NAME and TELEPHONE numbers of friends.

The parameters FS% and NF%, required by the loops and also for the DIMension statement, are READ in first. They happen to be 8,2 (8 records and 2 fields). The DIMension statement is not necessary in Commodore 64 BASIC for arrays not exceeding ten. There is always, however, a chance that you may wish to increase the number of records so it is always good practice to include DIMensions.

The headings of each field are also included in DATA statements

and read into the variables H1$ and H2$. Of course, there was no real need to waste these two variables because the headings NAME and TELEPHONE could have been entered as constants in literal text form in the body of the program, but variables are easier to amend. In any case, in programs which follow, the user is allowed to enter whatever heading is required so they must be variables.

Once the initial parameters have been received, the records can be READ into the appropriate two-dimensional array elements A$(F,R) by the two nested FOR/NEXT loops. Note carefully that the outer loop is controlling R and the inner loop is controlling F.

Information held in DATA lines is easy to alter without too much risk to the rest of the program – so easy, in fact, that it almost justifies a label of 'semivariable'. It is wise to allow a separate line for each record, even if there are only two fields in each record. Cramming a string of records under one line number may save a few bytes of memory but only at the expense of clarity.

As before mentioned, there is no option page and there is only one major subroutine which displays the entire file in *broadsheet* fashion. This is handled by the subroutine VIEW FILE, which simply unravels the data in A$(NF%,FS%) and presents it in humanised form for the screen display. The headings, H1$ and H2$, are enclosed within dotted lines – our only contribution to cosmetics. For simplicity, and because this is the first program, the lines are drawn by simple PRINT statements but, in subsequent programs, the line drawing will be relegated to a separate subroutine. Note that the line which PRINTs out the records ends with a semi-colon in order to suppress a carriage return so that field 2 (telephone number) appears on the same line as field 1 (name).

If you wish to try out fresh data, different headings and more records, simply replace the record DATA lines but remember also to update the loop parameters and field heading DATA accordingly. If you decide to use extra fields, the formatting will require altering in subroutine VIEW. There are several little touches which could have been added but the program is only intended to break the ice – finesse is left until later.

Program-based file with options

The next listing, Program 3.2, although still a program-based filing system, is a shade less primitive because it is *menu-driven*. Admittedly there are only three options, and one of these is EXIT

PROGRAM, but it should help in providing an insight into fresh material.

Variable names used in Program 3.2

FS% = File Size = number of records in each file.
NF% = Number of Fields.
H1$ = Heading of field 1.
H2$ = Heading of field 2.
F = a particular field.
R = a particular record.
A$(F,R) = an element of the two-dimensional array holding the main file data.
S% = option number selected.
K$ = general purpose variable.
FL% = flag.

```
100 REM PROGRAM BASED FILING SYSTEM
110 READ FS%,NF%
120 DIM A$(NF%,FS%)
130 REM ***
140 READ H1$,H2$
150 REM ***
160 REM READ DATA INTO ARRAY
170 FOR R=1 TO FS%
180 FOR F=1 TO NF%
190 READ A$(F,R)
200 NEXT:NEXT
210 REM ***
220 REM ***
230 GOSUB300:REM MENU
240 IF S%=1 THEN GOSUB460:GOTO230
250 IF S%=2 THEN GOSUB610:GOTO230
260 PRINT CHR$(147):PRINT"EXIT":END
270 REM *
280 REM **
290 REM SUBROUTINE MENU
300 PRINT CHR$(147)
310 PRINT TAB(3)"PROGRAM BASED FILING SY
STEM"
320 PRINT:PRINT:PRINT
330 GOSUB790:REM DRAW LINE
340 PRINT"(1) VIEW FILE"
350 PRINT"(2) DISPLAY RECORD"
```

```
360 PRINT"(3) EXIT PROGRAM"
370 GOSUB790:REM DRAW LINE
380 PRINT:PRINT:PRINT:PRINT
390 PRINT"SELECT OPTION ";
400 INPUT S%
410 IF S%<1 OR S%>3 THEN 300
420 PRINT CHR$(147):RETURN
430 REM *
440 REM **
450 REM VIEW FILE SUBROUTINE
460 PRINT TAB(7)"FILE CONTENTS"
470 GOSUB790:REM DRAW LINE
480 PRINT TAB(3)H1$ TAB(19)H2$
490 GOSUB790:REM DRAW LINE
500 FOR R=1 TO FS%
510 FOR F=1 TO NF%
520 PRINT"   "A$(F,R) TAB(18);
530 NEXT
540 PRINT
550 NEXT
560 GOSUB840:REM HOLD DISPLAY
570 RETURN
580 REM *
590 REM **
600 REM GET RECORD SUBROUTINE
610 FL%=0
620 PRINT"ENTER "H1$" REQUIRED ";
630 INPUT K$
640 IF K$="" THEN 620
650 PRINTCHR$(147):PRINT:PRINT:PRINT
660 PRINT TAB(3) H1$ TAB(19) H2$
670 GOSUB790:REM DRAW LINE
680 FOR R=1 TO FS%
690 FOR F=1 TO NF%
700 IF A$(1,R)=K$ THEN PRINT"   "A$(F,R)
 TAB(19);:FL%=1
710 NEXT:NEXT
720 IF FL%=0 THEN PRINT"THIS "H1$" NOT O
N FILE";
730 PRINT:GOSUB790:REM DRAW LINE
740 GOSUB840:REM HOLD DISPLAY
750 RETURN
760 REM *
770 REM **
780 REM SUBROUTINE DRAW LINE
790 PRINT"--------------------------------
--------"
```

```
800 RETURN
810 REM *
820 REM **
830 REM HOLD DISPLAY SUBROUTINE
840 PRINT:PRINT"PRESS ANY KEY TO CONTINU
E"
850 GET K$:IF K$="" THEN 850
860 RETURN
870 REM *
880 REM **
890 REM NUMBER OF RECORDS AND FIELDS
900 DATA 8,2
910 REM *
920 REM **
930 REM FIELD HEADINGS
940 DATA NAME,TELEPHONE
950 REM *
960 REM **
970 REM FILE INFORMATION
980 DATA DEREK,6668
990 DATA GILL,3467
1000 DATA KATH,3333
1010 DATA STEVE,666
1020 DATA JOHN,2357
1030 DATA PAUL,1104
1040 DATA GRAHAM,1456
1050 DATA PAT,7864
```

Program 3.2. Program-based file with options.

The program allows a choice between viewing the entire file in broadsheet form and displaying any one selected record. It can be seen that much of the structure is similar to the previous program. For example, the file information loop parameters are still read into an array from DATA statements but there are some additional subroutines which require study.

Subroutine MENU: displays the menu, issues the prompt, 'SELECT OPTION', and returns with the selected option number in S%.
Subroutine VIEW FILE: responsible for Option 1. It prints a heading and displays the file data.
Subroutine GET RECORD: responsible for Option 2. It allows the user to search for a particular record by quoting the key field. The user is requested to enter the chosen key field by the prompt which will appear on the screen as 'ENTER NAME REQUIRED'. Note

that H1$ is, in this case, NAME. The response is entered into the general purpose variable K$ and a trap for the *null string* is laid if the user inadvertently presses RETURN before entering the name. The searching is carried out by the outer FOR/NEXT loop after the headings have been displayed. As the records are scanned through, A$(1,R), which is the key field of all records, is compared with name in K$. Remember that R is the record variable and '1' defines the key field of the record. On finding the matching field, the complete record is printed out as the inner FOR/NEXT loop cycles through both fields. The sample data has only two fields but the subroutine will cater for any number of fields. If a match is found, the flag FL% is set to 1. If, after scanning through all records, no match is found, FL% will remain at its initialised value of 0 and the message 'THIS NAME NOT ON FILE' will appear.
Subroutine DRAW LINE: a minor subroutine used to draw a dotted line across the screen.
Subroutine HOLD DISPLAY: a minor subroutine which halts the program until, in response to a prompt, any key is pressed.

The flowchart of Fig. 3.1 will help in following the overall structure.

It would be easy to include a 'search' option for finding a record by TELEPHONE number instead of by key field but, as mentioned earlier, this type of file organisation is really not worth taking too seriously. Its value lies only in its simplicity – a stepping stone to better things.

Storing and retrieving data files

All information, stored on cassette tape or disk, is loosely called a 'file'. When a program is SAVEd it is referred to as a file although it should be called a *program file* to distinguish it from *data files*. The procedure for saving and loading programs has been made easy and the user is shielded from the complexity which is peculiar to the IEE-448 input/output bus system. To save and load program files on tape, the commands SAVE"name" and LOAD"name" respectively are sufficient because the default device is tape. To save and load program files on disk, it is necessary to inform the IEE-448 bus that device 8 is to be used so we write SAVE"name",8 and LOAD"name",8 respectively because 8 is the default device number for the floppy disk unit. Many programs, filing programs in particular, require the data

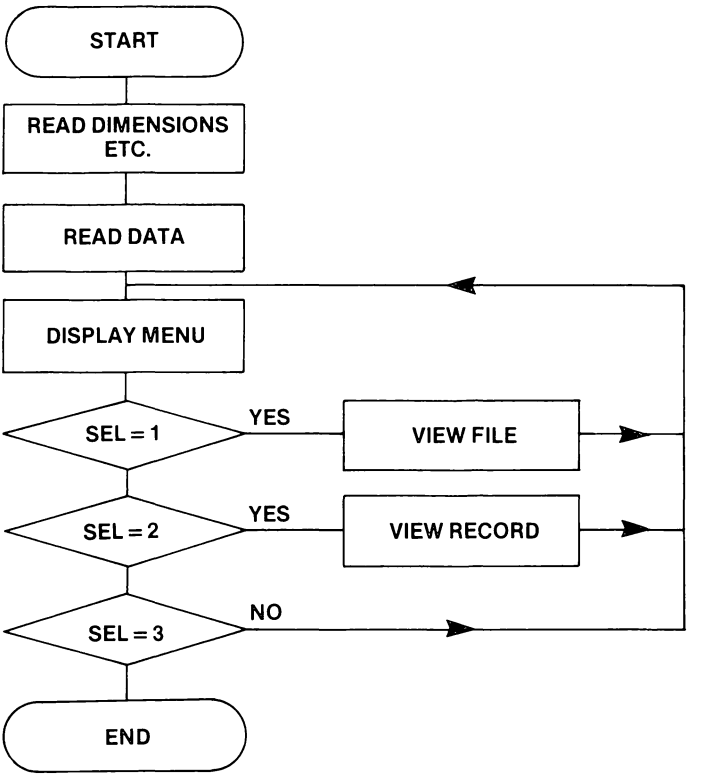


Fig. 3.1. Flowchart for program 3.2.

used by the program to be stored separately from the program itself. This separate information is stored as a data file. Data files cannot be accessed by the normal SAVE and LOAD commands because they are stored in a totally different format. The Commodore User Manual should be studied for detailed syntax but the treatment which follows will be sufficient for our present purpose.

Tape data files

Before data can be stored on tape, it is necessary to open a tape file from within the program by using the OPEN statement, the general form of which is as follows:

> OPEN file number, device number, secondary address, "file name"

The parameters have the following meanings:

File number: an arbitrary number of your own choice. It can be between 0-255 but is normally between 0-10. All subsequent references to the file will be by this number.
Device number: the device number of the cassette unit is fixed at 1.
Secondary address: This can be 0, 1 or 2 and is used for informing the tape unit whether it is to read data from tape or write data to it. 0=read, 1=write and 2=write with end of tape marker. The term 'secondary address' is unfortunate but is standard for the IEE-448 protocol. It really means the particular function required of that device so to call it an 'address' can be misleading.
File name: this can be the literal file name or a string variable.

Example: OPEN 2,1,0"TEST"
The file number is 2, the device is the cassette tape, the cassette is commanded to read and the file name is TEST.

Example: OPEN 5,1,1"TEST"
The file number is 5, the device is again the cassette tape, the cassette is commanded to write and the file name is contained in the variable N$.

Example: OPEN 1,1,2,"PERSONNEL"
The file number is 1, the device is the cassette tape, the cassette is commanded to write and append an end of tape marker.

Once the tape file has been OPENed, data is handled by the following general format:

PRINT#file number,data

Example: PRINT#2,K$
This prints the contents of K$ to file number 2.

Example: INPUT#5,G$
This reads from file number 5 into the variable G$.

Once a file has been OPENed and it has been finished with, it should be closed as follows:

CLOSE file number

Example: CLOSE 4

It is important to close a file when you have finished with it because there may still be data left in the cassette buffer area. The act of closing the buffer forces any remaining buffer contents onto tape.

To illustrate the procedure for writing data to tape, key in Program 3.3.

```
10 REM STORING DATA ON TAPE
20 PRINT CHR$(147)
30 INPUT"ENTER SOME CHARACTERS ";K$
40 OPEN1,1,1,"TEST"
50 PRINT#1,K$
60 CLOSE1
70 END
```

Program 3.3. Storing data on tape.

Before running the program, ensure you have a spare tape in place and rewound to receive the data. Once the data is on tape, key in Program 3.4 which will read back the data stored by the previous program.

```
10 REM READING DATA FROM TAPE
20 PRINT CHR$(147)
30 PRINT"PLEASE REWIND TAPE"
40 OPEN1,1,0,"TEST"
50 INPUT#1,G$
60 CLOSE1
70 PRINT"THE CHARACTERS READ BACK ARE:-"
80 PRINT G$
90 END
```

Program. 3.4. Reading data from tape.

Again, make sure that the data tape is rewound before running this program. Although the variable K$ was used to store and G$ was used to receive, we could use the same variable in both cases.

Serial disk data files

There are two types of file supported by the DOS, *serial* and *random*. The random files are, in turn, divided into relative and random access. We are, at the moment, concerned only with serial files. That is to say, all data items are stored on disk, one after the other, in the same order as they are received. In fact, as far as data organisation is concerned, serial disk files are no different to tape files.

The OPEN statement

The syntax for disk files is a little more complex than for tape so it is

sensible to treat them separately. For example, the third parameter in the OPEN statement is called the *channel number* instead of the 'secondary address'. The channel number must be between 2 and 14. For example:

```
OPEN 3,8,5
```

Here, the file number is 3, the device number for the disk is 8 and the channel number is 5.

However, the above is not a complete OPEN statement for disk. There may be more than one disk drive installed and it is necessary to inform the DOS what *type* of file is required. The complete syntax is, as we shall see, fairly involved:

```
OPEN 3,8,5,"disk command string"
```

The command string has the following syntax for serial files:

```
"drive number:file name,file type,direction"
```

Drive number is either 0 or 1 (normally 0 unless a dual drive is fitted).
File type is S (Serial file). As stated above, we are concerned here only with serial files. (Random files have a slightly different format for the OPEN statement.)
Direction is either W for writing to disk or R for reading from disk.

Example: OPEN 5,8,5,"0:TEST,S,W"

This opens file number 5 on device 8 via channel 5. The file is Serial, called TEST, and is opened for Writing. Be particularly careful when writing OPEN statements to use the exact punctuation as shown.

```
10 REM READ/WRITE DATA FROM DISC
20 PRINT CHR$(147)
30 PRINT"ENTER SOME CHARACTERS"
40 INPUT K$
50 OPEN5,8,5,"@0:TEST,S,W"
60 PRINT#5,K$
70 CLOSE5
80 OPEN5,8,5,"0:TEST,S,R"
90 INPUT#5,G$
100 CLOSE5
110 PRINT"THE CHARACTERS RECEIVED BACK W
ERE :-"
120 PRINTG$
130 END
```

Program 3.5. Read/write data from disk.

The same file opened for Reading would be

```
OPEN 5,8,5,"0:TEST,S,R"
```

Notice that the file number and the channel number are the same in both the above examples. They could be different but it is a well established practice, observed by most Commodore users, to choose them the same. Since they are both arbitrary numbers, anyway, it is less confusing and easier to remember if they are always chosen the same, providing they are within the prescribed limits of 2–14.

When a file is written to disk, an error condition will arise if there is a file of the same name already on disk. If it is desired that an existing file be replaced by a new file of the same name, the character @ should be added before the drive number when OPENing for write.

Example: OPEN 5,8,5,"@0:TEST,S,W"

To illustrate serial file syntax, key in Program 3.5. Make sure you have a previously formatted disk in the drive before you RUN the program. The program acts exactly the same for disk as the tape versions, given in Programs 3.3 and 3.4. Notice that the file replacement character @ is used so be careful that your disk does not already have a file named TEST or it will be overwritten.

Command channel 15

When we discussed channel numbers for serial files, it was stated that they just fall within the range 2–14 because channel number 15 is reserved for special duties. It is called the *command channel* and is responsible, amongst other things, for handling disk error conditions. In fact it is sometimes referred to as the *error channel* although this is a misleading term because, as we shall see in later chapters, it is also used for other purposes.

All major programs employing disk files should OPEN the command channel before any other files are OPENed by writing:

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

Note: The command channel should be left open all the time and finally closed only after all other channels have first been closed.

Providing the command channel is open, any generated error reports can be read into four arbitrarily chosen variables by the subsequent line:

INPUT#15,A,B$,C,D

If an error exists, the variables will be loaded with the following information:

Error number in A (first variable).
Plain language error message in B$ (second variable).
Track number where error occurred in C$ (third variable).
Sector number in D (fourth variable).

You can, of course, choose any other set of variables. The second must be a string variable. The list of error numbers appears on page 42 of the User Manual supplied with the 1541 drive unit where it should be noted that error numbers below 20 are of little significance. The most commonly used variable is the second, B$ in our example, because the plain language description of the error can be displayed by PRINT B$.

To illustrate the use of the command channel and error reporting, key in Program 3.6 which is virtually identical to Program 3.5 except for a few additional lines.

```
10 REM READ/WRITE DATA FROM DISC
20 PRINT CHR$(147)
30 PRINT"ENTER SOME CHARACTERS"
40 INPUT K$
50 OPEN15,8,15
60 OPEN5,8,5,"0:TEST,S,W":GOSUB1000
70 PRINT#5,K$
80 CLOSE5
90 OPEN5,8,5,"0:TEST,S,R":GOSUB1000
100 INPUT#5,G$
110 CLOSE5
120 CLOSE15
130 PRINT"THE CHARACTERS RECEIVED BACK W
ERE :-"
140 PRINTG$
150 END
1000 INPUT#15,A,B$,C,D
1020 IF A<20 THEN 1040
1030 PRINTA" "B$" "C" "D:END
1040 RETURN
```

Program 3.6. Use of command channel.

The bottom four lines constitute the subroutine which prints error messages. It is called up, by GOSUB1000, after each OPEN

statement and, if an error is generated, displays all four error parameters except those having numbers less than 20.

To try it out, leave the drive door open before you RUN for the first time. You should get:

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

Close the drive door and RUN the program twice. The second run should produce:

```
63 FILE EXISTS 0 0
```

Notice that there is no @ character before the drive number in the first OPEN statement so that when the program is re-run, using the same file name 'TEST', the above error message is displayed. It is sound policy to follow all OPEN statements with a subroutine similar to the one in Program 3.6.

Summary

1. Subroutines are good for structure, even if they are only used once in the program.
2. A program-based file is one in which the file data is held in DATA statements.
3. Two-dimensional arrays in this book will use A$(F,R) where F=field and R=record.
4. You cannot SAVE or LOAD a data file.
5. To open a tape file, use:

   ```
   OPEN,fn,1,sa,"name"
   ```

   where fn=file number.
6. The secondary address (sa) is 1 for write and 0 for read.
7. To read a data item, use:

   ```
   INPUT#fn,data item
   ```

8. To write a data item, use:

   ```
   PRINT#fn,data item
   ```

9. To close a file, use:

   ```
   CLOSE fn
   ```

10. If you forget to close a file before the program ends, data is lost.
11. Tape files are always serial files.

12. Disk files are either serial or random but only disk serial files are summarised below.
13. The open statement for disk serial files has the form:

    OPEN fn,8,cn,"command string"

    where cn is the channel number instead of the secondary address.
14. The command string has the form:

    "dr:fn,S,direction"

15. OPEN 6,8,6,"0:PERSONNEL,S,W" opens channel number 6 for writing a serial file and uses drive 0.
16. OPEN 6,8,6,"0:PERSONNEL,S,R" is the same but opened for reading.
17. To overwrite an existing file of the same name, use @ before the drive number.
18. Channel number 15 is reserved for the special command channel.
19. The command channel is opened by

    OPEN 15,8,15

    and should always be opened first, near the beginning of a program, and closed last of all.
20. Providing the command channel remains open, any disk error conditions can be read into four variables by using

    INPUT#15,A,B$,C,D

21. It is wise to call an error reporting subroutine immediately after each file is opened.
22. Error numbers less than 20 can be disregarded.

Self test

Tape files

3.1 A file is to be opened on file number 4 for reading and named NUCLEAR. Write the OPEN statement.

3.2 Write the OPEN statement for a file for writing data to file number 7 with end of tape marker, where the file name is in N$.

Serial disk files

3.3 Write the OPEN statement for the command channel.

3.4 A file for writing, named "GRANADA", is to be opened on drive 1 using channel 6. Write the OPEN statement.

3.5 A file for writing is to be opened on drive 0 using channel 7. The file name is in FN$ and it must overwrite any existing file of the same name. Write the OPEN statement.

3.6 The statement INPUT#15,B,G$,K,L is written for reading the command channel. Write the PRINT statement for displaying only the sector number where the error occurred.

Chapter Four

A Complete RAM-based Serial Filing System

The program listing

Chapter 3 dealt with the general methods of opening and closing serial files, both on tape and disk. This chapter is devoted entirely to the operation and description of a complete RAM-based serial filing program.

The program, as listed, is intended for disk files but information will be given to modify it for cassette tape. As can be seen from the length of the listing, it will take some time to key in and perhaps still longer to rectify the keying errors. Before continuing, it is worthwhile pointing out the difference between a 'bug' and a keying error. If we define a bug as a programming error, causing behaviour other than the programmer intended, then there are no known bugs in Program 4.1. If you key in the listing exactly as shown, the program should work in exactly the manner we intended. After you have used it for some time, you may think certain features can be improved or altered to suit your specific filing needs. In fact you may not need all the options, in which case you can leave one or two of them out in order to leave more room for records. Fortunately, the program is reasonably well structured so it is easy to introduce amendments or omit a few of the options. However, if you think that some of the subroutines can be shortened or made easier, be careful to keep the original listing on tape or disk as a 'fall back' precaution. Strange things can happen if attempts are made to 'simplify' programs.

```
10 REM RAM BASED FILING SYSTEM (DISC)
20 OPEN15,8,15: N$="NOT YET NAMED"
30 L%=0:F1=0:FL%=0:GOSUB12000
40 GOSUB16000
50 PRINT"(1)  CREATE NEW FILE"
60 PRINT"(2)  LOAD FILE"
70 PRINT"(3)  SAVE FILE"
```

```
80 PRINT"(4)  DISPLAY FILE"
90 PRINT"(5)  ADD RECORDS"
100 PRINT"(6)  MODIFY ANY FIELD"
110 PRINT"(7)  SORT BY ANY FIELD"
120 PRINT"(8)  CREATE SUB FILE (SEARCH A
NY FIELD)"
130 PRINT"(9)  DELETE RECORD"
140 PRINT"(10) CHECK BYTES FREE"
150 PRINT"(11) END PROGRAM
160 PRINT:PRINT:INPUT"SELECT OPTION ";S%
170 PRINT CHR$(147):FL%=L%:SL%=L%
180 IF S%<1 OR S%>11 THEN 40
190 IF S%<3 AND F1=1 THEN 11000
200 IF S%>2 AND L%=0 AND S%<>11 THEN PRI
NT"NO FILE PRESENT":GOSUB15000:RUN
210 ON S% GOSUB 1000,2000,3000,4000,5000
,6000,7000,8000,9000,26000,230
220 GOTO40
230 CLOSE15:END
997 REM *
998 REM **
999 REM CREATE FILE SUBROUTINE
1000 PRINT CHR$(147)
1010 PRINT"ENTER FILE SIZE (NUMBER OF RE
CORDS)"
1020 INPUT FS%
1030 IF FS%<1 THEN 1010
1040 PRINT"ENTER NUMBER OF FIELDS REQUIR
ED (2-10)"
1050 INPUT NF%
1060 IF NF%<2 OR NF%>10 THEN 1040
1070 NF%=NF%-1:DIM A$(NF%,FS%)
1080 PRINT CHR$(147)
1090 FOR F=0 TO NF%
1100 PRINT"ENTER FIELD HEADING";F+1
1110 GOSUB25000:A$(F,0)=K$
1130 NEXT
1140 GOSUB5000
1150 F1=1
1160 RETURN
1997 REM *
1998 REM **
1999 REM LOAD FILE SUBROUTINE
2000 GOSUB10000
2010 OPEN 6,8,6,"0:"+N$+",S,R"
```

```
2020 GOSUB27000:IF A>=20 THEN 2100
2030 INPUT#6,FS%,NF%,L%
2040 DIM A$(NF%,FS%)
2050 FOR R=0 TO L%
2060 FOR F=0 TO NF%
2070 INPUT#6,A$(F,R)
2080 NEXT:NEXT
2090 F1=1
2100 CLOSE6
2110 RETURN
2997 REM *
2998 REM **
2999 REM SAVE FILE SUBROUTINE
3000 GOSUB10000
3010 OPEN 6,8,6,"@:"+N$+",S,W"
3020 GOSUB27000:IF A>=20 THEN 3080
3030 PRINT#6,FS%:PRINT#6,NF%:PRINT#6,FL%
3040 FOR R=0 TO FL%
3050 FOR F=0 TO NF%
3060 PRINT#6,A$(F,R)
3070 NEXT:NEXT
3080 CLOSE6
3090 RETURN
3997 REM *
3998 REM **
3999 REM DISPLAY FILE SUBROUTINE
4000 C=1:S=1
4010 PRINT CHR$(147):PRINT"PRESS SPACE B
AR TO REGAIN MENU
4020 PRINT L$
4030 PRINT A$(0,0) TAB(20) A$(C,0)
4040 PRINT L$:SS=S+17
4050 IF SS>FL% THEN SS=FL%
4060 FOR R=S TO SS:PRINT A$(0,R) TAB(20)
 A$(C,R):NEXT
4070 GET K$:IF K$="" THEN 4070
4080 IF K$=CHR$(32) THEN 4180
4090 IF K$="L" THEN C=C-1
4100 IF K$="R" THEN C=C+1
4110 IF K$="U" THEN S=S-18
4120 IF K$="D" THEN S=S+18
4130 IF C<1 THEN C=NF%
4140 IF C>NF% THEN C=1
4150 IF S<1 THEN S=(INT(FL%/18)*18)+1
4160 IF S>FL% THEN S=1
```

```
4170 GOTO4010
4180 RETURN
4997 REM *
4998 REM **
4999 REM ADD RECORDS SUBROUTINE
5000 PRINT CHR$(147):IF L%>=FS% THEN PRI
NT"FILE FULL":GOSUB15000:GOTO5120
5010 L%=L%+1
5020 PRINT"TYPE (EXIT) TO FINISH ENTRY O
F RECORDS"
5030 PRINT
5040 PRINT"RECORD NUMBER ";L%
5050 PRINT:PRINT:F=-1
5060 F=F+1
5070 PRINT A$(F,0)
5080 GOSUB25000:A$(F,L%)=K$
5090 IF A$(F,L%)="EXIT" THEN L%=L%-1:GOT
O5120
5100 IF F<NF% THEN 5060
5110 IF L%<FS% THEN 5000
5120 RETURN
5997 REM *
5998 REM **
5999 REM MODIFY FIELD SUBROUTINE
6000 GOSUB14000:GOSUB13000:PRINT CHR$(14
7)
6010 PRINT"RECORD CONTAINING SELECTED FI
ELD":PRINT:PRINT L$
6020 FOR C=0 TO NF%
6030 PRINT A$(C,0) TAB(20) A$(C,R)
6040 NEXT
6050 PRINT L$:PRINT
6060 PRINT"CURRENT CONTENTS OF FIELD"
6070 PRINT A$(F,R):PRINT
6080 PRINT"ENTER NEW CONTENTS"
6090 GOSUB25000:A$(F,R)=K$
6100 RETURN
6997 REM *
6998 REM **
6999 REM SORT FILE SUBROUTINE
7000 GOSUB13000
7010 IF FL%<2 THEN 7160
7020 PRINT CHR$(147):PRINT"SORTING BY ";
A$(F,0)
7030 N%=FL%
```

```
7040 N%=(N%+2)/3
7050 FOR D=N%+1 TO N%*2
7060 FOR E=D TO FL% STEP N%
7070 FOR R=E TO D STEP -N%
7080 IF A$(F,R)>A$(F,R-N%) THEN 7130
7090 FOR C=0 TO NF%
7100 K$=A$(C,R):A$(C,R)=A$(C,R-N%):A$(C,
R-N%)=K$
7110 NEXT
7120 NEXT
7130 NEXT E
7140 NEXT
7150 IF N%>1 THEN 7040
7160 RETURN
7997 REM *
7998 REM **
7999 REM SEARCH/SHRINK FILE MENU
8000 GOSUB16000
8010 PRINT"SHRINK FILE (NON DESTRUCTIVE)
"
8020 PRINT"SEARCH ANY FIELD FOR:-"
8030 PRINT"(1)  <> A"
8040 PRINT"(2)   = A"
8050 PRINT"(3)  >= A"
8060 PRINT"(4)  <  A"
8070 PRINT"(5)  CHARACTER GROUP"
8080 PRINT:PRINT"FUTHER OPTIONS:-":PRINT
8090 PRINT"(6)  SAVE SUB-FILE"
8100 PRINT"(7)  DISPLAY SUB-FILE"
8110 PRINT"(8)  SORT SUB-FILE BY ANY FIE
LD"
8120 PRINT"(9)  DELETE SUB-FILE FROM MAI
N FILE"
8130 PRINT"(10) RETURN TO MAIN MENU"
8140 PRINT:PRINT
8150 INPUT"SELECT OPTION ";S%
8160 PRINT CHR$(147):FL%=SL%
8170 IF S%<1 OR S%>10 THEN 8000
8180 ON S% GOSUB 17000,17000,17000,17000
,17000,3000,4000,7000,24000,8200
8190 IF S%<10 THEN 8000
8200 RETURN
8997 REM *
8998 REM **
8999 REM DELETE RECORD SUBROUTINE
```

```
9000 GOSUB14000
9010 PRINT CHR$(147)
9020 PRINT"RECORD TO BE DELETED":PRINT:P
RINT L$
9030 FOR C=0 TO NF%
9040 PRINT A$(C,0) TAB(20) A$(C,R)
9050 NEXT
9060 PRINT L$:PRINT
9070 PRINT"DELETE THIS RECORD (Y/N)"
9080 GOSUB25000
9090 IF K$<>"Y" THEN 9170
9100 IF L%=1 THEN 9160
9110 FOR F=0 TO NF%
9120 A$(F,R)=A$(F,R+1)
9130 NEXT
9140 R=R+1
9150 IF R<L% THEN 9110
9160 L%=L%-1
9170 RETURN
9997 REM *
9998 REM **
9999 REM FILE NAME SUBROUTINE
10000 PRINT CHR$(147)
10010 PRINT"ENTER FILE NAME"
10020 GOSUB25000
10030 N$=K$
10040 IF LEN(N$)>16 THEN PRINT"TOO LONG"
:GOTO10010
10050 RETURN
10997 REM *
10998 REM **
10999 REM BELT AND BRACES PROTECTION
11000 PRINT"CAUTION : OPTION DESTROYS LO
ADED FILE"
11010 PRINT:PRINT
11020 PRINT"PRESS SPACE BAR TO CLEAR FIL
E"
11030 PRINT:PRINT"PRESS ANY OTHER KEY TO
 REGAIN MENU"
11040 GET K$:IF K$="" THEN 11040
11050 IF K$=CHR$(32) THEN RUN
11060 GOTO40
11997 REM *
11998 REM **
11999 REM DRAW LINE SUBROUTINE
```

```
12000 L$=""
12010 FOR K=1 TO 39
12020 L$=L$+CHR$(99)
12030 NEXT
12040 RETURN
12997 REM *
12998 REM **
12999 REM FIND FIELD SUBROUTINE
13000 PRINT CHR$(147):F=-1
13010 PRINT"OPERATE ON WHICH FIELD? (GIV
E HEADING)"
13020 GOSUB25000
13030 F=F+1
13040 IF K$=LEFT$(A$(F,0),LEN(K$)) THEN
13070
13050 IF F<NF% THEN 13030
13060 PRINT CHR$(147):PRINT"NO SUCH FIEL
D":GOSUB15000:GOTO13000
13070 RETURN
13997 REM *
13998 REM **
13999 REM FIND RECORD SUBROUTINE
14000 PRINT CHR$(147):R=0
14010 IF S%=6 OR S%=9 THEN F=0
14020 PRINT"GIVE RECORD ENTRY UNDER ";A$
(F,0)
14030 GOSUB25000
14040 R=R+1
14050 IF K$=LEFT$(A$(F,R),LEN(K$)) THEN
14080
14060 IF R<FL% THEN 14040
14070 PRINT CHR$(147):PRINT"NO SUCH RECO
RD":GOSUB15000:GOTO14000
14080 RETURN
14997 REM *
14998 REM **
14999 REM PRESS ANY KEY SUBROUTINE
15000 PRINT:PRINT"PRESS ANY KEY TO CONTI
NUE"
15010 GET K$:IF K$="" THEN 15010
15020 RETURN
15997 REM *
15998 REM **
15999 REM TITLE/STATUS SUBROUTINE
16000 PRINT CHR$(147):PRINT L$
```

```
16010 PRINT"RAM BASED FILING SYSTEM"
16020 PRINT L$
16030 IF F1=0 THEN PRINT"FILE NOT PRESEN
T"
16040 IF F1=1 THEN PRINT"FILE LOADED: ";
N$
16050 PRINT
16060 RETURN
16997 REM *
16998 REM **
16999 REM SEARCH DIRECTOR SUBROUTINE
17000 GOSUB13000
17010 PRINT"SEARCH TO OPERATE ON ";A$(F,
0)
17020 IF S%<5 THEN PRINT"GIVE SEARCH DAT
UM/ENTITY
17030 IF S%=5 THEN PRINT"GIVE CHARACTER
GROUP"
17040 GOSUB25000:B$=K$:L2%=0
17050 ON S% GOSUB 18000,19000,20000,2100
0,22000
17060 SL%=L2%
17070 IF SL%=0 THEN PRINT CHR$(147):PRIN
T"SEARCH IS NEGATIVE":GOSUB15000
17080 RETURN
17997 REM *
17998 REM **
17999 REM SEARCH <>
18000 FOR R=1 TO SL%
18010 IF A$(F,R)<>B$ THEN L2%=L2%+1:GOSU
B23000
18020 NEXT
18030 RETURN
18997 REM *
18998 REM **
18999 REM SEARCH =
19000 FOR R=1 TO SL%
19010 IF A$(F,R)=B$ THEN L2%=L2%+1:GOSUB
23000
19020 NEXT
19030 RETURN
19997 REM *
19998 REM **
19999 REM SEARCH >=
20000 FOR R=1 TO SL%
```

```
20010 IF A$(F,R)>=B$ THEN L2%=L2%+1:GOSU
B23000
20020 NEXT
20030 RETURN
20997 REM *
20998 REM **
20999 REM SEARCH <
21000 FOR R=1 TO SL%
21010 IF A$(F,R)<B$ THEN L2%=L2%+1:GOSUB
23000
21020 NEXT
21030 RETURN
21997 REM *
21998 REM **
21999 REM SEARCH FOR SUBSTRING
22000 LC=LEN(B$)
22010 FOR R=1 TO SL%
22020 LT=LEN(A$(F,R))
22030 IF LC>LT THEN 22080
22040 FOR K=1 TO LT-LC+1
22050 IF B$=MID$(A$(F,R),K,LC) THEN L2%=
L2%+1:GOSUB23000:GOTO22080
22070 NEXT
22080 NEXT R
22090 RETURN
22997 REM *
22998 REM **
22999 REM MOVE RECORD SUBROUTINE
23000 FOR C=0 TO NF%
23010 K$=A$(C,R):A$(C,R)=A$(C,L2%):A$(C,
L2%)=K$
23020 NEXT
23030 RETURN
23997 REM *
23998 REM **
23999 REM DELETE SUB FILE SUBROUTINE
24000 IF SL%=L% THEN 24050
24010 FOR R=SL%+1 TO L%
24020 FOR F=0 TO NF%
24030 A$(F,R-SL%)=A$(F,R)
24040 NEXT:NEXT
24050 L%=L%-SL%:S%=10
24060 RETURN
24997 REM *
24998 REM **
```

```
24999 REM INPUT VALIDATION SUBROUTINE
25000 K$="":INPUT K$
25010 IF K$="" THEN 25000
25020 IF LEN(K$)>18 THEN K$=LEFT$(K$,18)
25030 RETURN
25997 REM *
25998 REM **
25999 REM BYTES FREE SUBROUTINE
26000 PRINT CHR$(147):PRINT"WAIT":PRINT
26010 X=FRE(0)-(SGN(FRE(0))<0)*65535
26020 PRINT"NUMBER OF BYTES FREE";X
26030 GOSUB15000
26040 RETURN
26997 REM *
26998 REM **
26999 REM READ ERROR CHANNEL SUBROUTINE
27000 INPUT#15,A,B$,C,D
27010 IF A<20 THEN 27030
27020 PRINT CHR$(147):PRINT B$:GOSUB1500
0
27030 RETURN
```

Program 4.1. RAM-based multifile system.

Those not yet accustomed to marathon bouts of keyboard bashing may be interested to know there are two ways of accomplishing the task:

(a) Brute force and hope for the best – that is to say, start at line 10 and carry on until the last line. This method must work – eventually!
(b) Notice that the main program only occupies the first 23 lines. These lines should be keyed in first and then all the subroutines necessary to allow option 1, create file, to be tried out. Remember that option 1 calls up a few other subroutines so, naturally, you must enter these as well. It may help if you study Table 4.1 which appears later in this chapter. You can go through the motions of creating a small file of three records, each of three fields, say Name, Age, Trade. Proceed on these lines for each option in turn, making sure that the program runs O.K. up to the point which you have reached. However, there is a world of difference between a program which appears to work O.K. and one which has been subjected to exhaustive tests. These should be delayed until you have explored the complete program.

Using the program

On RUNning the program, the menu is displayed. Unless a file already exists in RAM, there are only two possible choices on the menu, Option 1 'Create file' or Option 2 'Load file'. If any other option is selected, the message 'No file loaded' is displayed and the menu regained by pressing any key. Operating details which follow are treated in the order they are most likely to be used.

Using Option 1: Create file

This begins by asking for your estimate of the maximum number of records intended to be held in the file followed by the number of fields. Don't be too ambitious with the number because the amount of RAM available after the program has been located is about 25K. It would have been easy at this point to have programmed in a check on RAM limits but this would have curtailed some of the user's freedom.

The next prompts asks for the number of fields. The maximum number has been fixed at ten and a trap is included to prevent the limit being exceeded. It is easy to modify the trap if you wish to increase the number of fields. Remember that the space occupied by the file depends on the product of the number of records and the number of fields in each record.

The next prompt is for the headings required for each field. They must not exceed 18 characters in length and if more than 18 are entered, the program will strip off any excess characters. This concludes the work required to create a new file, apart from entering the actual file data. As an aid to subsequent explanations, assume that the file headings have been entered as Name, Telephone and Trade.

After the field headings have been entered, the program is ready to receive data. The record number is displayed at the top of the screen and increments automatically as records are entered. Record entry is terminated by typing EXIT. If you intend to enter a large number of records at one sitting, it is worth terminating entry now and again to select Option 10 (Check bytes free). You can resume entering records again by selecting Option 5 (Add records) which, in this case, would mean adding further records. It would have been easy to arrange for 'bytes free' to be displayed at the top of the screen and automatically updated as each record is added. The trouble with this procedure is the delay which occurs due to the 'house-keeping' action of the FRE(0) statement in Commodore BASIC. The delay time increases alarmingly as the number of records increase.

Option 4: Display file

After the file has been created, the next logical step is to have a look at it. Option 4 has the following appearance:

Press space bar for menu

NAME	TELEPHONE
BLENKINSOP J	666 6778
GUTSWORTHY M	233 6895
GLUGENHEIMER K	233 1212

NAME is the fixed key field but the second field, TELEPHONE, is a variable (in the display sense). Other fields can be rotated into the right-hand position by use of the 'L' and 'R' keys. If more than 18 records are in the file, other blocks of 18 can be moved into view with the 'U' and 'D' keys. If the display is moved beyond the boundaries of the fields or records, *wrap around* occurs. That is to say, if there were 7 fields and we tried to shift in a non-existent eighth field, the display would wrap around to the second field again. The same thing happens if we try to bring in a non-existent block of records, except that we should probably have to describe it as roll around instead of wrap around.

Option 3: Save file

Once the file has been displayed, you will most probably wish to save a copy in store. The only prompt in this option is 'Enter file name'. A maximum of 16 characters is allowed for the file name. If more than 16 are typed in, the message 'Too long' appears. The listing for the subroutine is obviously for disk but necessary modifications for cassette tape are given later in the chapter.

Option 2: Load file

The file name must be entered and the loaded file will overwrite any existing file present in RAM. Again, modifications to the subroutine to suit cassette tape will be given later.

Option 5: Add records

This is used to add further records to an existing file. The operating procedure and display is similar to that described under the 'Create file' option, apart from the record number. This will represent the nth instead of the first record.

Option 6: Modify any field

The keyfield of the required record is first requested, followed by the particular field to be modified. To save the operator having to remember the exact details, the *first one or two characters* are normally sufficient. For example, assuming that the record on GUTSWORTHY M is required, it may be sufficient to enter 'GUTS' or even 'GU' after the request for which record and, say, just 'T' for TELEPHONE after the request for field. The screen should then display the existing record and request the new field data to be entered.

Option 8: Create subfile (Search any field)

On selecting Option 8, an alternative menu is displayed. The first half of the menu offers five ways of selecting which records from the main file are to be treated as a 'subfile'. Building up a subfile from records which share some common attribute is a powerful feature of the program although this, in itself, is no justification for promoting the filing system to the status of a 'database'. It is worth digressing a little at this point to clarify the distinction.

The two terms files and databases are often used as if they were one and the same. They are not. Indeed, there are profound differences between a 'data file' and a 'database'. To avoid any possible confusion, we can think of a database as an integrated collection of records, each containing information on the *relationship* between them. These records could well be distributed throughout a number of different files. On the other hand, a simple file contains records all of the same type. A typical database has the following characteristics:

(a) A collection of *cross-referenced* records, known as the 'database' held within perhaps, a multitude of files.
(b) Cross-references are stored in an organised fashion, within the database itself, so that complex search requirements can be executed rapidly.
(c) A database is accessed from an applications program via a piece of software called a DataBase Management System (DBMS). Thus the applications program is independent of changes in the data structure.

It is obvious from the qualities of a true database that Option 8 has no pretentions in this direction. However, it is certainly capable of carrying out fast search operations within the bounds of a single data file. We shall test this by describing how a subfile can be progressively narrowed down until one particular record survives the final 'shrink'.

Assume that the file loaded from store is named EUROPE and contains four fields - Country,Area(square miles),Population and Capital. We wish to find a country with an area less than 40000 square iles with a population greater than or equal to 500000. The task begins by selecting Option 8 which brings us to the Search menu, displaying the heading 'Shrink file (Non-destructive)'. The steps from then on could vary but the following approach is reasonable:

(1) Select Option 4 (search for < A). The first prompt is

Which field ?

The response should be AREA.
(2) The next prompt is

Search to operate on AREA
Give search datum/entry

The response should be 40000.
On selecting Option 7 (View file), the subfile is displayed with all countries having areas less than 40000 square miles. The search menu is now regained (by pressing the space bar). We now shrink this subfile still further.
(3) Select Option 3 (search for >= A).
The first prompt is

Which field ?

The response should be Population.
(4) The next prompt is

Search to operate on Population
Give search datum/entry

The response should be 500000.

On selecting Option 7 (View subfile), the new subfile is displayed with all countries having an area less than 40000 square miles with a population greater than or equal to 500000 inhabitants. We could, of course, shrink still further by insisting that the capital of the country must contain the substring 'er'. According to our own file on EUROPE (compiled with the help of Whitaker's 1980 edition), the final shrinkage produced only one country - Switzerland. The second half of the search menu offers several straightforward options, including save and sort the subfile at any stage. They are used in exactly the same manner as their counterparts in the main menu. However, it is worth pointing out that the ability to split the main file,

by using Option 9 (Delete subfile from main file) is very useful when the main file grows too large for RAM. For example, we can create a subfile of all NAMEs which begin with letters lower than 'L'. The subfile, after being saved under another file name can then be deleted from the main file. It then becomes a new 'main file' and the original file, now much depleted, can be saved separately. With regard to the above remark, 'letters lower than L', it should be explained that any search option in the subfile menu acts on the ASCII codes if the field is alphabetic. Thus, if we use the operator '<' on an alphabetic field, under NAME, it will compare sequential ASCII characters as necessary for the decision.

Option 7: Sort

Option 7 in the main menu (or 8 in the subfile menu) will sort a file into order by any field heading. The subject of sorting is treated in detail in Chapter 5. In the meantime, it is simply a case of responding to the prompt 'Which field'. If we sort under NAME, it will put all records into alphabetical order by name. On the other hand, a sort under AREA will be numerical although the numbers are still held in string form. Because of this, it is important when creating a file, that all numeric data entered should have the *same total number of digits*. Small numbers must have leading zeros added as packing. If this is not done, numeric sorts will not be carried out correctly. On the other hand, if you never intend to sort under a numeric field there is no need to use leading zeros. However, sorting data into order is important, so the next chapter is devoted entirely to the subject.

Option 9: Delete record

This option deletes any selected record from the main file and closes up the gap left in the file array. The user is prompted to enter the key field of the record to be deleted. The specified record is then displayed on the screen. Because an unwanted deletion can be annoying, the user is then given the chance to confirm or cancel by responding to the message 'Delete this record (Y/N)'.

Variable names used in Program 4.1

FS% = Maximum file size = maximum number of records in file.
NF% = Number of Fields.
H1$ = Heading of field 1.
H2$ = Heading of field 2.

F = a particular field number.
R = a particular record number.
A$(F,R) = An element of the two-dimensional array holding the main file data.
S% = option number selected.
K$ = general purpose variable.
L% = main file length.
FL% = file for subroutine (subfile or main file).
F1 = file loaded flag (1=loaded).
FS% = maximum file size.
L$ = a dotted line.
SS = bottom line of display.
S = top line of display.
L2% = search subfile counter.
SL% = subfile length.
K$,LC and LT are general purpose variables.

Subroutine calls

The program itself is short, occupying only the first twenty or so lines and concerned only with initialising variables, displaying the primary menu and obtaining the option number selected by the user. All subsequent processing is relegated to subroutines. They are all given a title by REM statements so they should be easy to locate on the listing. Continual reference to line numbers has been avoided because nearly all the subroutines start at round figures of 1000. There are no GOTOs to REM lines so these can be omitted from the listing if you wish to save memory space.

Each option has its own first level subroutine which may call on second level which, in turn, may even call on third level subroutines. Table 4.1 shows the subroutine calls and should prove useful if you decide to follow the advice given earlier and key in the program a block at a time. It should also help you to understand the underlying strategy.

The subroutines are fairly straightforward and hardly justify line by line explanations.

Modifications to suit cassette tape files

As it stands, Program 4.1 will only work on compatible Commodore

Table 4.1. Subroutine calls

1st level	2nd and 3rd levels
Create file	Input validation Add records
Load file	File name Read error channel
Save file	File name Read error channel
Display file	(Self contained)
Add records	Input validation
Modify field	Find record Press any key Find field Input validation
Sort file	Find field Press any key
Search/shrink	Title/status Search director Save file Display file Sort file Delete subfile
Delete record	Find record Input validation
Bytes free	Press any key

disk drives but it should be quite easy to modify the program for cassette tape if you follow the directions below.

(1) Replace the Save and Load file subroutines with the lines shown in Program 4.2.
(2) Delete line 20 which OPENs the command channel for disk in Program 4.1.
(3) Delete the CLOSE15 statement in line 230.
(4) Delete the entire Read Error Channel subroutine at the bottom of Program 4.1.
(5) All calls of GOSUB 27000 (Read Error Channel subroutine above) should be deleted from Program 4.1.

```
1997 REM *
1998 REM **
1999 REM LOAD FILE SUBROUTINE
2000 GOSUB10000
2010 OPEN 1,1,0,N$
2030 INPUT#1,FS%,NF%,L%
2040 DIM A$(NF%,FS%)
2050 FOR R=0 TO L%
2060 FOR F=0 TO NF%
2070 INPUT#1,A$(F,R)
2080 NEXT:NEXT
2090 F1=1
2100 CLOSE1
2110 RETURN
2997 REM *
2998 REM **
2999 REM SAVE FILE SUBROUTINE
3000 GOSUB10000
3010 OPEN 1,1,1,N$
3030 PRINT#1,FS%:PRINT#1,NF%:PRINT#1,FL%
3040 FOR R=0 TO FL%
3050 FOR F=0 TO NF%
3060 PRINT#1,A$(F,R)
3070 NEXT:NEXT
3080 CLOSE1
3090 RETURN
```

Program 4.2. Cassette tape load and save.

Use of Program 4.1

The advantage of RAM-based filing systems is the processing speed and the degree of sophistication which can be achieved. The disadvantage, of course, is the limited length of individual files.

Fortunately, the Commodore 64 is reasonably well supplied with RAM so, in spite of the length restriction, the program is certainly worth the effort to key in. The Sort file option is obviously a candidate for experimentation. BASIC sort routines, even the best, are irritatingly slow when there are a large number of records. The speed can be increased dramatically if the BASIC version is replaced by one of the machine code versions described in the next chapter.

Summary

1. A RAM-based filing system requires the entire file, rather than single records, to be loaded from store to RAM.
2. A subfile contains all records in the main file which satisfies criteria specified by the user.
3. Program 4.1 can only sort correctly under a numeric field if each file has the same number of digits. Leading zeros can be used as packing.
4. If a subfile is stored, it assumes the status of a mainfile when reloaded.
5. Program 4.1, as it stands, is written for disk files but it is easy to convert for cassette tape.
6. The sort subroutines can be replaced by machine code versions.
7. Program 4.2 replaces the save and load subroutines if cassette files are to be used.

Self test

4.1 Why was a separate option used for checking how many bytes are free instead of incorporating it in the 'Add record' subroutine?

4.2 Why do leading zeros, added to numbers, solve the numerical sort problem?

Chapter Five

Searching and Sorting

It is often required to sort records into order. This chapter is devoted to the subject and will progress from a simple exchange sort, through bubble and diminished increment to a complete two-dimensional string array sort written in machine code. The latter is capable of sorting a computer full of records in about a second or two. The chapter will treat sorting qualitatively to avoid boring the reader with too much dry academic analysis.

As in all the BASIC sorts in this chapter, the actual sort subroutine starts at line 1000. The preceding lines are simply a test program used to set up the array and display the execution time.

The exchange sort

This is the simplest of all sorting algorithms but is rather slow. The technique, shown in Program 5.1, is based on the comparison of adjacent items in an unsorted list or array. The object is to sort an integer array A%(N) into ascending order. A pass, starting at A%(1), is made through the array until two adjacent elements are found in descending order. The offending array elements are exchanged in position and the cycle restarted at A%(1). The cycle is repeated as many times as necessary until the list is completely ordered. The technique certainly works but is not widely used. Its merit lies in its simplicity.

```
10 REM EXCHANGE SORT DEMONSTRATION
20 PRINT CHR$(147)
30 INPUT"HOW MANY INTEGERS ";K%
40 DIM A%(K%+1)
50 FOR N=1 TO K%
60 A%(N)=K%-N
70 PRINT A%(N)
```

```
80 NEXT
90 PRINT "SORTING"
100 TI$="000000"
110 L%=K%:GOSUB1000
120 T=INT(TI/60+0.5)
130 FOR N=1 TO K%
140 PRINT A%(N)
150 NEXT
160 PRINT"SORTING TIME= ";T
170 END
1000 N=0
1010 N=N+1
1020 IF A%(N)>A%(N+1) THEN T%=A%(N):A%(N
)=A%(N+1):A%(N+1)=T%:GOTO1000
1030 IF N<L%-1 THEN 1010
1040 RETURN
```

Program 5.1. Exchange sort.

The relative timings for various configurations are shown in Table 5.1.

Table 5.1. Exchange sort timings.

Array order	Elements	Execution time
Reverse	20	29 seconds
Reverse	50	417 seconds
In order	20	0 seconds
In order	50	1 second
In order	100	2 seconds
Random	20	21 seconds
Random	50	270 seconds

Studying Table 5.1 reveals that the sorting time increases alarmingly as the number of elements increases. However, the time taken to execute when the array is in order, or nearly in order, is good. The table also reveals that the worst case condition in an exchange sort is when the array is in reverse order. This is only to be expected. The worst case number of comparisons needed (C) is given by:

$$C=\frac{(N^3+5N-6)}{6}$$

where N is the number of elements to be sorted. The time taken will thus be proportional to N^3 if N is large.

The bubble sort

The bubble sort is considered an improvement on the exchange sort because time is not wasted comparing adjacent array elements already correctly positioned in their final order. This is probably the most popular of all sorting routines providing there is only a small number of array elements. A bubble sort demonstration is given in Program 5.2.

```
10 REM BUBBLE SORT DEMONSTRATION
20 PRINT CHR$(147)
30 INPUT"HOW MANY INTEGERS ";K%
40 DIM A%(K%+1)
50 FOR N=1 TO K%
60 A%(N)=K%-N
70 PRINT A%(N)
80 NEXT
90 PRINT "SORTING"
100 TI$="000000"
110 L%=K%:GOSUB1000
120 T=INT(TI/60+0.5)
130 FOR N=1 TO K%
140 PRINT A%(N)
150 NEXT
160 PRINT"SORTING TIME= ";T
170 END
1000 L%=L%-1
1010 FOR N=1 TO L%
1020 IF A%(N)>A%(N+1) THEN T%=A%(N):A%(N
)=A%(N+1):A%(N+1)=T%
1030 NEXT
1040 IF L%>1 THEN 1000
1050 RETURN
```

Program 5.2. Bubble sort.

Program 5.2 consists of an inner and an outer control loop. The pairs of array elements are repeatedly incremented, compared and if necessary swopped in position within the inner loop. The largest integer in the array will always 'bubble' through to the last position. It is no longer necessary to involve this integer in further comparisons,

since it will be in its final array position. The outer loop counter, L%, can thus be decremented by one. On the following series of inner loop cycles, the next largest integer bubbles through to the next to last position in the array, and so on, until the array is completely ordered.

Table 5.2 shows the timing data for the bubble sort program.

Table 5.2. Bubble sort timings.

Array order	Elements	Execution time
Reverse	20	6 seconds
Reverse	50	35 seconds
Reverse	100	138 seconds
In order	20	2 seconds
In order	50	14 seconds
In order	100	54 seconds
Random	20	4 seconds
Random	50	25 seconds
Random	100	96 seconds

Table 5.2 reveals that the bubble sort is, in general, an improvement on the exchange sort in terms of execution time. The average number of comparisons is given approximately by $\frac{1}{2}$ N^2, where N is the number of array elements. Thus it will take four times as long to sort double the number of elements. However, the performance when the array is ordered, or near-ordered, is poorer than for the exchange sort. This can be rectified, without incurring too much additional overhead, by introducing a 'swop flag' as shown in Program 5.3. The process allows early termination of the sort if no swops are detected on any inner loop cycle. The swop flag, CH%, is cleared to zero at the start of each inner loop and is set to one within the loop only if a swop is made. As soon as a cycle ends with CH% clear, the array must have been in order.

```
10 REM BUBBLE SORT (WITH SWOP FLAGS)
20 PRINT CHR$(147)
30 INPUT"HOW MANY INTEGERS ";K%
40 DIM A%(K%+1)
50 FOR N=1 TO K%
60 A%(N)=K%-N
70 PRINT A%(N)
80 NEXT
```

```
90 PRINT "SORTING"
100 TI$="000000"
110 L%=K%:GOSUB1000
120 T=INT(TI/60+0.5)
130 FOR N=1 TO K%
140 PRINT A%(N)
150 NEXT
160 PRINT"SORTING TIME= ";T
170 END
1000 CH%=0:L%=L%-1
1010 FOR N=1 TO L%
1020 IF A%(N)>A%(N+1) THEN T%=A%(N):A%(N
)=A%(N+1):A%(N+1)=T%:CH%=1
1030 NEXT
1040 IF CH%=1 THEN 1000
1050 RETURN
```

Program 5.3. Bubble sort with swop flags.

The comparative timings using the flag system are shown in Table 5.3.

Table 5.3. Bubble/swop flag combination.

Array order	Elements	Execution time
Reverse	20	6 seconds
Reverse	50	38 seconds
Reverse	100	151 seconds
In order	20	0 seconds
In order	50	1 second
In order	100	1 second
Random	20	4 seconds
Random	50	26 seconds
Random	100	106 seconds

As can be seen from Table 5.3, the near-order performance is greatly improved by the use of swop flags with the marginal trade-off in execution time under other conditions.

The diminishing increment sort

This group of sort algorithms is sometimes named after D.L. Shell

who devised his Shellsort in 1959. However, there are numerous variations on the same basic skeleton. The differences are small, but sometimes significant, so we will deal with only two such variations in this Chapter.

Although bubble sort routines are efficient for small numbers of elements, the execution time increases alarmingly when in excess of about 25 or so elements. To see the delay on high numbers try running Program 5.3 with 1000 integers! About a day or so later the array will be sorted! An improvement for large values of N is to use one of the many diminishing increment sort variants of which Program 5.4 is perhaps the simplest example. We noted earlier that the bubble sort is fairly efficient for small numbers of elements. We also noted that the use of a swop flag system significantly speeds up the execution of a bubble sort when the elements are roughly in order. The diminishing increment sort capitalises on both these features. Essentially the array is split up into small sets which are bubble sorted. These are merged to form larger sets which will be roughly in order. These larger sets will be bubble sorted and merged to form even larger sets and so on until we are left with one large roughly ordered list. This is finally bubble sorted, which will be efficient due to the points made earlier. The number of sets are progressively halved each time round the outer loop but the number of elements within a set is doubled. If we go beyond this simplistic treatment, without the aid of mathematics, we run the danger of lapsing into verbal diarrhoea. An understanding of the intricate details of a diminishing increment sort, or indeed any other algorithm, is to make use of a *trace table*. The idea is to follow the program through on paper, using arbitrary test data and taking notes of the various key variables such as loop counters, etc.

```
10 REM DIMINISHING INCREMENT SORT
20 REM DEMONSTRATION VERSION 1
30 PRINT CHR$(147)
40 INPUT"HOW MANY INTEGERS ";K%
50 DIM A%(K%+1)
60 FOR N=1 TO K%
70 A%(N)=K%-N
80 PRINT A%(N)
90 NEXT
100 PRINT "SORTING"
110 TI$="000000"
120 L%=K%:GOSUB1000
130 T=INT(TI/60+0.5)
```

```
140 FOR N=1 TO K%
150 PRINT A%(N)
160 NEXT
170 PRINT"SORTING TIME= ";T
180 END
1000 E=INT(LOG(L%)/LOG(2))
1010 F=2^E
1020 FOR G=1 TO E
1030 F=F/2:M=L%-F
1040 CH%=0
1050 FOR N=1 TO M
1060 IF A%(N)>A%(N+F) THEN T%=A%(N):A%(N
)=A%(N+F):A%(N+F)=T%:CH%=1
1070 NEXT
1080 M=M-1
1090 IF CH%=1 THEN 1040
1100 NEXT
1110 RETURN
```

Program 5.4. Diminishing increment sort (version 1).

The timings for the simple diminishing increment sort are given in Table 5.4.

Table 5.4. Diminishing increment sort timings (version 1)

Array order	Elements	Execution time
Reverse	100	17 seconds
Reverse	500	118 seconds
Reverse	1000	270 seconds
In order	100	6 seconds
In order	500	40 seconds
In order	1000	92 seconds
Random	100	36 seconds
Random	500	542 seconds
Random	1000	1824 seconds

Some interesting features are exposed by Table 5.4. The most striking is the increase in speed, due to the more linear relationship between execution time and the number of array elements. Notice that for this algorithm, worst case operation is no longer the reverse order condition. This honour has been transferred to randomly generated arrays. Although mathematicians are fond of discussing

the randomness of a random number we will be content with the Commodore formula for producing a random number – 1000*RND (1).

Another point worth noting in Program 5.4 is its random access nature – that is to say, comparisons are made with elements spaced far apart. This is fine if the data is in RAM but not if on disk. Program 5.5 is a variation on the simple diminishing increment sort. The only fundamental difference is in the way the sets are divided up and composed. Notice that a constant (2) is added in at line 1010 to help the sets merge better. This purely arbitrary number could be left out without affecting the basic operation of the sort but the performance is likely to be improved by this addition. Analysis of Shellsort is very difficult and estimates of the number of comparisons required have only, to date, been estimated under special conditions. Broadly, Program 5.5 is an improvement on the previous one in terms of execution time. This is evident from the comparison of Table 5.5 and 5.4. The most outstanding improvement is in the sorting of a randomly generated array. The near-order performance, however, is slightly inferior.

```
10 REM DIMINISHING INCREMENT SORT
20 REM DEMONSTRATION VERSION 2
30 PRINT CHR$(147)
40 INPUT"HOW MANY INTEGERS ";K%
50 DIM A%(K%+1)
60 FOR N=1 TO K%
70 A%(N)=K%-N
80 PRINT A%(N)
90 NEXT
100 PRINT "SORTING"
110 TI$="000000"
120 L%=K%:GOSUB1000
130 T=INT(TI/60+0.5)
140 FOR N=1 TO K%
150 PRINT A%(N)
160 NEXT
170 PRINT"SORTING TIME= ";T
180 END
1000 N%=L%
1010 N%=(N%+2)/3
1020 FOR D=N%+1 TO N%*2
1030 FOR E=D TO L% STEP N%
1040 FOR R=E TO D STEP -N%
1050 IF A%(R)>A%(R-N%) THEN 1080
```

```
1060 T%=A%(R):A%(R)=A%(R-N%):A%(R-N%)=T%
1070 NEXT
1080 NEXT E
1090 NEXT
1100 IF N%>1 THEN 1010
1110 RETURN
```

Program 5.5. Diminishing increment sort (version 2).

This particular version of the Shellsort (Program 5.5) was also used in Program 4.1 in the previous chapter and is quite acceptable written in BASIC provided N is not too large.

Table 5.5. Diminishing increment sort timings (version 2)

Array order	Elements	Execution time
Reverse	100	16 seconds
Reverse	500	106 seconds
Reverse	1000	237 seconds
In order	100	8 seconds
In order	500	48 seconds
In order	1000	115 seconds
Random	100	21 seconds
Random	500	115 seconds
Random	1000	362 seconds

It is generally agreed that there is no universally superior sorting algorithm. Each have their relative advantages and disadvantages. The programmer should choose the one most suited to the particular task in hand. An analogy can be found between sort routines and cricketers. Cricketers are either good bowlers or good batsmen but rarely both.

Sorting strings and two-dimensional string arrays

Earlier examples in this chapter have all used integer array variables as the elements. String array variables can be substituted by changing all occurrences of A%(N) to A$(N). The corresponding test part of the programs, of course, would need to be changed.

A common file organisation, encountered in RAM-based filing systems, is the two-dimensional string array. If a RAM-based file is

dimensioned A$(NF%,FS%) then the file A$ can be considered to contain FS% records each of NF% fields. We can define any field in a specific record by A$(F,R) where F is the field number and R is the record number. Thus, to sort records according to a specific field we will need to modify our routines to swop all the fields of the pair of records each time. Another FOR/NEXT loop will take care of this. (See the sort routine of Program 4.1.)

If a string sort is to be used successfully on numeric data fields, the user should ensure that all numbers have a constant number of digits (including leading zeros). An alternative is the conditional use of the VAL statement. However, this would introduce further complication along with an alarming increase in execution time. We used the former method in Program 4.1 in the previous chapter.

Machine code solutions

Whatever sort techniques we use, there is no denying that where large numbers of elements are involved, the execution time can be unacceptable. We are, after all, up against the inherent defects of the BASIC language. It is an interpretive, rather than a compiled, language so execution speed is slow. The solution to the speed problem is to program in machine code.

Even in machine code, it will still be important to select a suitable algorithm. For example, assuming that a list is large, a bubble sort written in machine code wouldn't execute much faster than a Shellsort written in BASIC. However, in order to minimise the tedium of machine code programming it is sometimes advantageous to choose algorithms which are inherently *binary* in nature. Of all the algorithms briefly discussed in this chapter, the diminishing increment sort (version 1) is perhaps the best choice, taking into account both programming effort and general performance. This results from the binary method used to calculate the increment series.

Machine code integer sort

The machine code version of Program 5.4 is given in Program 5.6. Its BASIC test routine is Program 5.7. As you will probably be aware, the Commodore 64 stores array integers sequentially in two-byte form, the least significant byte being highest in memory. Thus, Program 5.6 compares and swops the data in two-byte groups. In

general, this new machine code version will execute about 160 times faster than its BASIC equivalent and will significantly outperform, over the range tested, any of the BASIC sorts described.

```
10 REM DIMINISHING INCREMENT SORT
20 REM (SIGNED INTEGER ARRAY)
30 N%=193:PRINT CHR$(147)
40 B=49152:PRINT"LOADING BYTES FROM $COO
0"
50 FOR L=0 TO N%-1
60 READ D$
70 FD%=ASC(D$)-48
80 SD%=ASC(RIGHT$(D$,1))-48
90 IF FD%>9 THEN FD%=FD%-7
100 IF SD%>9 THEN SD%=SD%-7
110 BT%=16*FD%+SD%
120 POKE B+L,BT%
130 NEXT
140 DATA A9,01,85,57,A9,00,85,5C
150 DATA 85,58,E6,5C,06,57,26,58
160 DATA 38,A5,57,E5,FB,A5,58,E5
170 DATA FC,90,EF,46,58,66,57,38
180 DATA A5,FB,E5,57,85,FD,A5,FC
190 DATA E5,58,85,FE,A9,00,85,FF
200 DATA 85,5A,85,5B,18,A5,2F,69
210 DATA 09,85,5D,A5,30,69,00,85
220 DATA 5E,A5,58,85,59,A5,57,0A
230 DATA 26,59,18,65,5D,85,4E,A5
240 DATA 5E,65,59,85,4F,A0,01,38
250 DATA B1,4E,F1,5D,88,B1,4E,F1
260 DATA 5D,50,02,49,80,10,16,C8
270 DATA 84,FF,B1,5D,AA,B1,4E,91
280 DATA 5D,8A,91,4E,88,10,F3,30
290 DATA 04,D0,B1,D0,9E,E6,5A,D0
300 DATA 02,E6,5B,A5,5D,18,69,02
310 DATA 85,5D,90,02,E6,5E,18,A5
320 DATA 4E,69,02,85,4E,90,02,E6
330 DATA 4F,A5,FD,C5,5A,D0,B6,A5
340 DATA FE,C5,58,D0,B0,A5,FF,F0
350 DATA 13,38,A5,FD,E9,01,85,FD
360 DATA B0,04,C6,FE,A5,FD,D0,C1
370 DATA A5,FE,D0,BD,C6,5C,D0,BB
380 DATA 60
```

Program 5.6. Machine code diminishing increment sort (integer array).

```
10 REM SORT TEST PROGRAM
20 REM ARRAY OF SIGNED INTEGERS
30 PRINT CHR$(147)
40 INPUT"SORT HOW MANY INTEGERS";B%
50 REM FILL AND DISPLAY RANDOM ARRAY
60 DIM A%(B%)
70 FOR N=1 TO B%
80 A%(N)=INT(32000*RND(1))
90 A%(N)=A%(N)-16000
100 PRINT A%(N)
110 NEXT
120 PRINT"SORTING"
130 REM SET UP NUMBER PARAMETER
140 HB%=B%/256
150 LB%=B%-(HB%*256)
160 REM PASS NUMBER PARAMETER
170 POKE 251,LB%
180 POKE 252,HB%
190 REM CALL SORT ROUTINE
200 TI$="000000"
210 SYS 49152
220 T%=TI/60+0.5
230 REM DISPLAY SORTED ARRAY
240 FOR N=1 TO B%
250 PRINT A%(N)
260 NEXT
270 PRINT"SORTED"B%"INTEGERS IN"T%"SECON
DS
```

Program 5.7. BASIC test program for machine code integer sort routine.

Program 5.6 is supplied in the form of a hex loader program which on RUNning loads the machine code bytes (object code) starting at address $C000 (49152 decimal). This is a 4K area of memory protected from BASIC and is ideal for placing machine code programs. Once Program 5.6 has been RUN the machine code routine is loaded and Program 5.6 can be deleted. At this stage Program 5.7 can be loaded and used to test that all has been typed in correctly. Once tested, a final copy of Program 5.6 can be saved for use at any time.

It is beyond the scope of the present book to discuss 6502 machine code but for those interested in a detailed treatment of Program 5.6 and those which follow, we suggest you refer to our book *Advanced Machine Code Programming for the Commodore 64* which includes

line by line treatment of various machine code sorting routines written in assembly language (source code).

It is not necessary to understand machine code in order to use it. The BASIC programmer need only know how to *load* and *call* it. To load it, simply RUN the Program 5.6; the code takes a few seconds to load so be patient. The setting of the variable B in line 40 determines where the object code is to be located. The variable N% in line 30 sets the number of bytes to be loaded. The code, once loaded, is executed from any BASIC program via a series of POKE statements and a SYS49152 command. The procedure for executing the code is shown in lines 140 to 210 of Program 5.7.

The total number of integers in the array is entered into the arbitrary chosen variable B%. This is split into high byte and low byte form in lines 140 and 150. The low byte (LB%) is POKEd into location 251 in line 170 and the high byte (HB%) is POKEd into location 252 in line 180. Once these parameters have been set up the code is executed simply by SYS49152.

It is important to stress that this, and the subsequent machine code sorting routines supplied, will only work on the first array encountered in the array space. Therefore the array which is to be sorted must be the first DIMensioned. For example, if A%(N) is the only array to be sorted and a second array B%(N) is also present in a program, then be careful to use

```
10 DIM A%(N),B%(N)
```

and not the other way round -

```
10 DIM B%(N),A%(N)
```

Table 5.6. Machine code diminished increment sort timings.

Array order	Elements	Execution time
Reverse	100	0 seconds
Reverse	500	1 second
Reverse	1000	2 seconds
In order	100	0 seconds
In order	500	0 seconds
In order	1000	1 second
Random	100	0 seconds
Random	500	3 seconds
Random	1000	12 seconds

or the machine code routine will attempt to sort B%(N) with the parameters intended for A%(N)!

Table 5.6 illustrates the relative advantages to be gained by the transition from BASIC to machine code. The completely ordered or near-ordered performance is very fast and does not increase alarmingly as the number of elements is increased. On the other hand, the random integer performance is acceptable for most sorting requirements.

Machine code string array sort

Practical files usually have more fields holding alpha characters than numeric. Program 5.8 is the machine code necessary to sort string arrays. The corresponding test program is Program 5.9 which sets up a random string array, calls the machine code routine and displays the sorted array. The code is loaded and executed in a similar manner to that described for the integer sort.

```
10 REM DIMINISHING INCREMENT SORT
20 REM (STRING ARRAY)
30 N%=227:PRINT CHR$(147)
40 B=49152:PRINT"LOADING BYTES FROM $C00
0"
50 FOR L=0 TO N%-1
60 READ D$
70 FD%=ASC(D$)-48
80 SD%=ASC(RIGHT$(D$,1))-48
90 IF FD%>9 THEN FD%=FD%-7
100 IF SD%>9 THEN SD%=SD%-7
110 BT%=16*FD%+SD%
120 POKE B+L,BT%
130 NEXT
140 DATA A9,01,85,4F,A9,00,85,4E
150 DATA 85,50,E6,4E,06,4F,26,50
160 DATA 38,A5,4F,E5,FB,A5,50,E5
170 DATA FC,90,EF,46,50,66,4F,38
180 DATA A5,FB,E5,4F,85,FD,A5,FC
190 DATA E5,50,85,FE,A9,00,85,FF
200 DATA 85,51,85,52,18,A5,2F,69
210 DATA 0A,85,57,A5,30,69,00,85
220 DATA 58,A5,50,85,26,A5,4F,0A
230 DATA 26,26,18,65,4F,65,57,85
```

```
240 DATA 59,A5,26,65,50,65,58,85
250 DATA 5A,A0,00,B1,57,85,5F,B1
260 DATA 59,85,60,C8,B1,57,85,5B
270 DATA B1,59,85,5D,C8,B1,57,85
280 DATA 5C,B1,59,85,5E,A0,00,B1
290 DATA 5D,D1,5B,90,11,D0,20,C8
300 DATA C4,5F,F0,1B,C4,60,F0,06
310 DATA D0,ED,D0,A0,D0,8D,A0,02
320 DATA 84,FF,B1,57,AA,B1,59,91
330 DATA 57,8A,91,59,88,10,F3,E6
340 DATA 51,D0,02,E6,52,A5,57,18
350 DATA 69,03,85,57,90,02,E6,58
360 DATA A5,59,18,69,03,85,59,90
370 DATA 02,E6,5A,A5,FD,C5,51,D0
380 DATA 98,A5,FE,C5,52,D0,92,A5
390 DATA FF,F0,13,38,A5,FD,E9,01
400 DATA 85,FD,B0,02,C6,FE,A5,FD
410 DATA D0,B0,A5,FE,D0,AC,C6,4E
420 DATA D0,AA,60
```

Program 5.8. Diminishing increment sort of string array.

```
10 REM SORT TEST PROGRAM
20 REM STRING ARRAY
30 PRINT CHR$(147)
40 INPUT"SORT HOW MANY STRINGS";B%
50 REM FILL AND DISPLAY RANDOM ARRAY
60 DIM A$(B%)
70 FOR N=1 TO B%
80 B$=""
90 A%=10*RND(1)+1
100 FOR Z=1 TO A%
110 R%=26*RND(1)
120 K$=CHR$(R%+65)
130 B$=B$+K$
140 NEXT
150 A$(N)=B$
160 PRINT A$(N)
170 NEXT
180 PRINT:PRINT
190 PRINT"SORTING"
200 PRINT:PRINT
210 REM SET UP NUMBER PARAMETER
220 HB%=B%/256
230 LB%=B%-(HB%*256)
```

```
240 REM PASS NUMBER PARAMETER
250 POKE 251,LB%
260 POKE 252,HB%
270 REM CALL SORT ROUTINE
280 TI$="000000"
290 SYS 49152
300 T%=TI/60+0.5
310 REM DISPLAY SORTED ARRAY
320 FOR N=1 TO B%
330 PRINT A$(N)
340 NEXT
350 PRINT
360 PRINT"SORTED"B%"STRINGS IN"T%"SECOND
S"
```

Program 5.9. BASIC test program for machine code string sort routine.

When a string array is set up by the interpreter, three bytes are used in a similar manner to integers. These bytes are not the strings themselves but the length and address of where the string is stored. These three bytes are referred to as a *string information block* and represent the low byte address, high byte address, and string length respectively. The actual string, consisting of the ASCII codes in sequential memory locations, is stored from the starting address given in bytes 1 and 2 above. A string array is formed by a series of string information blocks stored sequentially in memory. Swopping strings during a string sort is easy because it is necessary only to swop the string information blocks since these tell the interpreter where the strings are stored. There is no need to swop the strings themselves.

As before, all that is necessary to load the code, is to RUN Program 5.8. The BASIC test program, Program 5.9, includes the POKEs into locations 251 and 252 and the machine code call SYS49152 but once the program has been loaded, it can be called from any other program in a similar manner.

Machine code version of a two-dimensional string array sort

Program 5.10 will sort a two-dimensional string array. It is fast. In fact, it will sort a computer full of records in about a second or two, so it was considered unnecessary to include a detailed timing table.

```
10 REM DIMINISHING INCREMENT SORT
20 REM (TWO DIMENSIONAL STRING ARRAY)
```

```
30 N%=271:PRINT CHR$(147)
40 B=49152:PRINT"LOADING BYTES FROM $C00
0"
50 FOR L=0 TO N%-1
60 READ D$
70 FD%=ASC(D$)-48
80 SD%=ASC(RIGHT$(D$,1))-48
90 IF FD%>9 THEN FD%=FD%-7
100 IF SD%>9 THEN SD%=SD%-7
110 BT%=16*FD%+SD%
120 POKE B+L,BT%
130 NEXT
140 DATA A9,01,85,4F,A9,00,85,4E
150 DATA 85,50,E6,4E,06,4F,26,50
160 DATA 38,A5,4F,E5,FB,A5,50,E5
170 DATA FC,90,EF,A5,FD,18,69,01
180 DATA 85,FD,0A,65,FD,85,FD,18
190 DATA A5,FE,0A,65,FE,85,FE,18
200 DATA A5,2F,69,09,65,FD,85,61
210 DATA A5,30,69,00,85,62,46,50
220 DATA 66,4F,38,A5,FB,E5,4F,85
230 DATA 28,A5,FC,E5,50,85,29,A9
240 DATA 00,85,FF,85,51,85,52,A5
250 DATA 61,85,57,A5,62,85,58,A9
260 DATA 00,85,26,85,27,A6,FD,18
270 DATA A5,26,65,4F,85,26,A5,27
280 DATA 65,50,85,27,CA,D0,F0,18
290 DATA A5,57,65,26,85,59,A5,58
300 DATA 65,27,85,5A,A4,FE,B1,57
310 DATA 85,5F,B1,59,85,60,C8,B1
320 DATA 57,85,5B,B1,59,85,5D,C8
330 DATA B1,57,85,5C,B1,59,85,5E
340 DATA A0,00,B1,5D,D1,5B,90,11
350 DATA D0,21,C8,C4,5F,F0,1C,C4
360 DATA 60,F0,06,D0,ED,D0,98,D0
370 DATA 85,A4,FD,88,84,FF,B1,57
380 DATA AA,B1,59,91,57,8A,91,59
390 DATA 88,10,F3,E6,51,D0,02,E6
400 DATA 52,A5,57,18,65,FD,85,57
410 DATA 90,02,E6,58,A5,59,18,65
420 DATA FD,85,59,90,02,E6,5A,A5
430 DATA 28,C5,51,D0,97,A5,29,C5
440 DATA 52,D0,91,A5,FF,F0,13,38
450 DATA A5,28,E9,01,85,28,B0,02
460 DATA C6,29,A5,28,D0,AF,A5,29
470 DATA D0,AB,C6,4E,D0,A9,60
```

Program 5.10. Diminishing increment sort of two-dimensional string array.

```
10 REM SORT TEST PROGRAM
20 REM 2 DIMENSIONAL STRING ARRAY
30 PRINT CHR$(147)
40 INPUT"SORT HOW MANY RECORDS";NR%
50 INPUT"NUMBER OF FIELDS";NF%
60 INPUT"SORT WHICH FIELD";F%
70 REM FILL AND DISPLAY RANDOM ARRAY
80 DIM A$(NF%,NR%)
90 FOR R=1 TO NR%
100 FOR F=1 TO NF%
110 B$=""
120 A%=10*RND(1)+1
130 FOR Z=1 TO A%
140 R%=26*RND(1)
150 K$=CHR$(R%+65)
160 B$=B$+K$
170 NEXT
180 A$(F,R)=B$
190 PRINTA$(F,R)
200 NEXT
210 PRINT
220 NEXT
230 PRINT:PRINT
240 PRINT"SORTING"
250 PRINT:PRINT
260 REM SET UP NUMBER PARAMETER
270 HB%=NR%/256
280 LB%=NR%-(HB%*256)
290 REM PASS PARAMETERS
300 POKE 251,LB%
310 POKE 252,HB%
320 POKE 253,NF%
330 POKE 254,F%
340 REM CALL SORT ROUTINE
350 TI$="000000"
360 SYS 49152
370 T%=TI/60+0.5
380 REM DISPLAY SORTED ARRAY
390 FOR R=1 TO NR%
400 FOR F=1 TO NF%
410 PRINT A$(F,R)
420 NEXT
430 PRINT
440 NEXT
450 PRINT"SORTED"NR%"RECORDS IN"T%"SECON
DS"
```

Program 5.11. BASIC test program for machine code two-dimensional string array sort.

Program 5.11 is a BASIC routine for trying out the machine code sort. You are asked for the number of records, number of fields and which field (number) is to be the subject of the sort. As before, the routine can be called not only from the BASIC test program but by any other program, provided the correct parameters below are POKEd into the locations detailed below. The code is executed by the familiar SYS49152 command.

As can be seen from the test program, Program 5.11, the parameters required by the sort routine are:

NR% (the number of records in the file)
NF% (the number of fields)
F% (which is the sorting field number in the range 0 to NF%.)

Of these, only the parameter NR% is likely to exceed 255, so it must be split up into low byte and high byte form prior to POKEing into locations 251 and 252. The parameters NF% and F% are POKEd straight into locations 253 and 254 respectively.

The routine can sort records with up to 42 fields, which is a limit comfortably above the demands of most files. It can be easily spliced into the RAM-based filing system, Program 4.1, described in the previous chapter. There are, however, a few snags. First, since all the file data resides in a single two-dimensional string array, all numeric fields must be stored and sorted as their *string representation*. Therefore, if we are to sort records by numeric fields, we must ensure, at the data entry stage, that all numeric data in a given field consists of the same number of digits. This could often entail the entry of some leading zeros. This is a common occurrence in the real world: account numbers, etc. frequently contain them. Also, if monitary values are represented in fields it might be necessary to store $28 as $00028.00 in order to sort them. This is because other entries under the same field heading may contain odd cents.

An alternative, which avoids the above restrictions, is to employ two separate sort routines. Ultra-fast sorting of records in alphanumeric fields using the machine code routine (Program 5.10) and a dedicated BASIC Shellsort make use of the VAL statement for sorting numeric fields. However, this solution inevitably trades off execution speed in favour of memory savings enabled by storing numerical data in a more efficient, but undisciplined form.

We will concentrate on how to incorporate the machine code segment to replace directly the BASIC Shellsort of Program 4.1. The procedure is as follows:

(1) Run Program 5.10 so as to load the machine code from address $C000 (49152 decimal) onwards.
(2) Replace the Sort File subroutine (lines 7000–7160) with the subroutine given in Program 5.12.

It should be remembered that Program 5.10 should always be LOADed and RUN before LOADing the modified RAM-based filing program so that the code is present in RAM. In order for the sort routine to function, it should be re-emphasised that the file array must be the first one DIMensioned in a filing program. Also, the elements must be accessed in the form A$(F,R) where F is the field number and R is the record number.

```
6997 REM *
6998 REM **
6999 REM CALL M/C SORT ROUTINE
7000 IF FL%<2 THEN 7090
7010 GOSUB13000
7020 HB%=FL%/256
7030 LB%=FL%-HB%*256
7040 POKE251,LB%
7050 POKE252,HB%
7060 POKE253,NF%
7070 POKE254,F
7080 SYS49152
7090 RETURN
```

Program 5.12. Replacement sort subroutine for Program 4.1.

Sequential or linear search

This is the search algorithm most widely used and involves starting from the beginning of a list and comparing each element in turn with the search key. When the end of the list is reached and no match has been found the search is deemed to have failed. We used this simple technique in Program 4.1. Program 5.13 is an uncluttered demonstration program of this simple technique. The subroutine starting at line 1000 is so simple as not to require further explanation. The preceding lines generate an array containing sequential odd numbers. Thus, all odd numbers will be present in the array and all even numbers in the range will be absent. RUN the program with, say, 1000 elements and see how long it takes either to find or to determine that the specified number entered into G%, in line 90, is not present in the array. Compare the search times with that of the more efficient binary search given in Program 5.14.

```
10 REM LINEAR SEARCH DEMONSTRATION
20 PRINT CHR$(147)
30 INPUT"HOW MANY INTEGERS ";K%
40 DIM A%(K%)
50 FOR N=1 TO K%
60 A%(N)=N*2+1
70 PRINT A%(N)
80 NEXT
90 INPUT"SEARCH FOR ";G%
100 GOSUB1000
110 INPUT"SEARCH AGAIN (Y/N)";K$
120 IF K$="Y" THEN 90
130 END
1000 J%=0
1010 J%=J%+1
1015 IF J%>K% THEN PRINT"ITEM NOT FOUND"
:GOTO1050
1020 IF G%=A%(J%) THEN PRINT"ITEM FOUND
AT ARRAY POSITION ";J%:GOTO1050
1040 GOTO1010
1050 RETURN
```

Program 5.13. Simple sequential search demonstration.

The binary search

Searching for a particular record within a file is a common processing requirement, even more common than sorting. The objective of a search is to find the data along with the key item which identifies it. If the file is unsorted, a sequential search can be used whereby the item searched for is compared with each item in the file, starting with the first and continuing until a match is found or the search has produced negative results. Sequential searching, although relatively easy to understand and program, is obviously slow because, on average, half the file will need to be searched before the required data is found. In other words there will be, on average N/2 comparisons for N items in the search list. A much faster method, assuming that the records are first sorted into order, is called the 'binary' search.

```
10 REM BINARY SEARCH DEMONSTRATION
20 PRINT CHR$(147)
30 INPUT"HOW MANY INTEGERS ";K%
40 DIM A%(K%)
50 FOR N=1 TO K%
```

```
60 A%(N)=N*2+1
70 PRINT A%(N)
80 NEXT
90 INPUT"SEARCH FOR ";G%
100 GOSUB1000
110 INPUT"SEARCH AGAIN (Y/N)";K$
120 IF K$="Y" THEN 90
130 END
1000 LO%=1:HI%=K%
1010 J%=(LO%+HI%)/2
1020 IF G%=A%(J%) THEN PRINT"ITEM FOUND
AT ARRAY POSITION ";J%:GOTO1070
1030 IF G%<A%(J%) THEN HI%=J%-1
1040 IF G%>A%(J%) THEN LO%=J%+1
1050 IF LO%>HI% THEN PRINT"ITEM NOT FOUN
D OR ARRAY NOT IN ORDER":GOTO1070
1060 GOTO1010
1070 RETURN
```

Program 5.14. Binary search demonstration.

Program 5.14, with the exception of the search subroutine at line 1000, is similar in form to the previous one. Assuming that the array has first been sorted into ascending order, the data item in the middle of the list is first compared with the item to be matched. If the item is smaller than the required item, the search continues in the first half of the array. If the item is larger, the search concentrates on the second half of the array. On locating which half, the process continues as before by first testing the middle item in that half. Eventually, by continually halving, and testing, the required data item is either found or declared to be non-existent. On the surface, this may seem a longer process than the simple sequential search but this is only because it has taken longer to explain. The equation of interest is

$$\text{Average number of comparisons} = \log_2(N)$$

where N is the total number of data items to be searched.

This is a startling result and worth an example, if only to illustrate the superiority of the binary search over the simple sequential search. Assume that we wish to locate a specific item from within a total list of 10000 items. We will compare both methods:

(a) Sequential search:

$$\text{Average number of comparisons} = N/2 = 10000/2 = 5000$$

(b) Binary search:

Average number of comparisons=$\log_2(N)=\log_2(10000)=14$ (when rounded up)

Even with one million items, the number of comparisons would only be 20. However, it is only fair to stress once more that a binary search can only be carried out on a previously sorted file, whereas the sequential search makes no demands at all on the order of the file items. If a file is small in size, it may not always be worth troubling to sort it and it certainly would not be sensible to sort it just for the sake of using a binary search. On the other hand, if a file has to be sorted for other reasons, then it would be silly not to employ binary searching. Another point is that many RAM-based files are frequently sorted and stored on tape or disk in alphabetical order, anyway.

Summary

1. The exchange sort is simple but slow. The execution time increases roughly proportionally to the third power of N.
2. The bubble sort is faster than the exchange sort, except when the array is near-ordered. The execution time is roughly proportional to the square of N.
3. Introducing a swop flag to the bubble sort decreases the execution time on ordered or near-ordered arrays, with little overhead.
4. A diminishing increment sort can still use bubble techniques but first splits up the array into small sets. The sorted sets are then progressively merged into larger sets until a single sorted set remains.
5. The diminishing increment sort (version 2) is valuable in cases where the array starts in random order but is not quite so efficient as version 1 if the array is near-ordered.
6. The most convenient sort algorithm for machine coding, consistent with good performance, is the simple diminishing increment sort (version 1).
7. Records can be stored in two-dimensional array form, where the fields occupy one dimension and the records the other.
8. A particular record in a file can be located by either a simple sequential search or, if in order, a binary search.
9. A sequential search can be performed on an unordered file.

Self test

5.1 Using the equation given, calculate – to the nearest million – the number of comparisons needed for an exchange sort if the array contains 500 items in reverse order.

5.2 With reference to 5.1 above, if each comparison cycle in 5.1 above takes 1 millisecond, calculate the approximate sorting time to the nearest hour.

5.3 Calculate the average number of comparisons made when exchange sorting 1000 array elements.

Chapter Six

Serial, Sequential and Indexed Relative Files

The Commodore 64 can support only one magnetic tape recorder. Furthermore, direct access is not readily possible on magnetic tape. This forces disk as the only practical storage medium for the filing systems which follow.

Serial files

A *serial file* is one where records or data are stored simply as they occur, with no attempt made to store them in any particular order. It is a simple file organisation and is very economical in the use of store. The disadvantages are that record retrieval is rather slow and modifications are practically impossible. Although not seriously considered in large-scale commercial computing, serial files are quite useful in microcomputer systems. They are ideal for the storage of records generated in chronological order, or for temporary storage of information pending later processing. We have treated examples of this simple file organisation in previous chapters.

Sequential files

The idea behind sequential file organisation is that the records are always stored in their key field or *primary key* order. A primary key is the field which uniquely *identifies* a record. A secondary key is one which tends to *classify* a record rather than identify it. An example is a three-field file where the cheque number is the primary key and 'to whom payable' and 'amount' are secondary keys. The cheque number will be unique, but the amount, and to whom payable, may not.

Once a sequential file exists it is practically impossible to add,

modify or delete records without the use of special techniques.

Each time data is read back from a sequential file it can only be accessed in the same order as that in which it was stored. In this way, storage medium is not wasted since each record butts up to the last until the end of the file is reached. As a result of this, data already stored will be overwritten if the file is reopened for addition of records at a later date.

As far as modifications are concerned, it is unlikely that a new record will exactly fit into the space occupied by the original record. Thus, read errors will become inevitable. The solution is to build up a separate sorted file of all updated records – that is to say, modifications, deletions and insertions of records. This sorted file, called the *transaction file*, is then merged with the existing sequential file (*master file*) to produce a third, called the *update file*. The update file ultimately becomes promoted to the status of master file next time the applications program is run. To clarify the chicken and egg situation, the first sorted transaction file can be considered to be the initial master file.

This occasional updating process is often referred to as *file reorganisation*, and can be accomplished by the following series of steps:

(a) Read a record from both the master file and the transaction file.
(b) Compare the key fields.
(c) If the keys are the same, write the updated record from the transaction file to the update file. Return to (a).
(d) If the keys are not the same, copy the record with the lower key to the update file. Read in the next record from the same file to replace it. Return to (b).
(e) Once the end of either input file has been reached, copy the remaining records from the other input file to the update file.

Note that in (c) above it may first be necessary to combine information from certain fields of the master file and transaction file. Deletions can be signalled by a marker or 'tombstone', in which case no record will need to be written to the update file.

The file reorganisation process requires three files to remain open at the same time – two for input and one for output (see Fig. 6.1). This is no problem on the Commodore 1541 disk unit.

To make the fullest use of sequential files it is convenient to have a double disk drive connected. The master file can be opened for input on, say, DRIVE 0 and the update file opened for output on DRIVE 1. The transaction file can be either in RAM, if frequent updates are

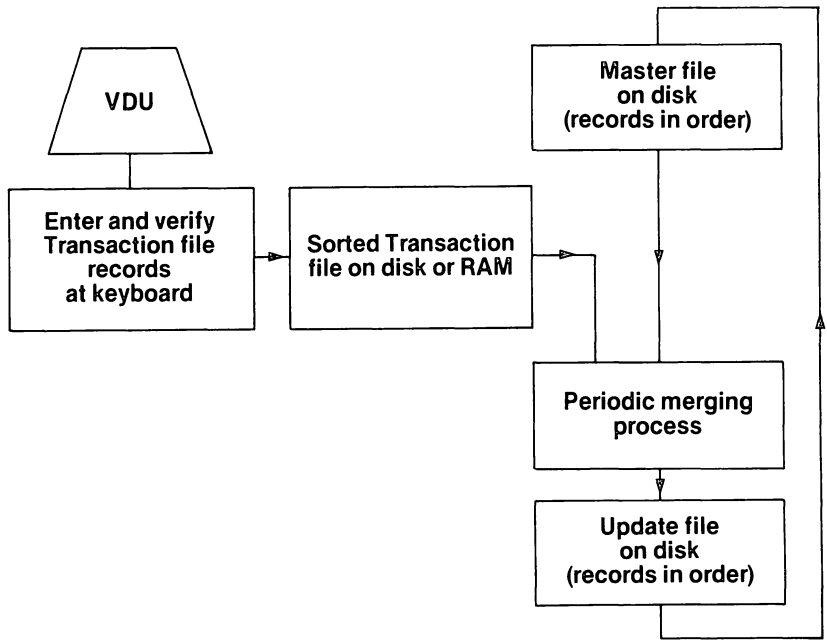


Fig. 6.1. Updating sequential files.

necessary, or stored on one of the other two disks. It would be possible, however, to store all three on one single-sided disk if restricted to a single drive.

Apparently, microcomputer users are seldom interested in this way of keeping sequential files. Perhaps some microcomputer enthusiasts have a total aversion to batch processing and tend to think of it as a relic of the teletype and magnetic core memory era. In spite of this, it is safe to say that sequential files are in common use and are the simplest ordered file organisation.

Direct or random access files

Direct files, also known as *random access files*, are so named because records can be directly accessed irrespective of their position in the file. Although direct files are often referred to as 'random access' files, they must not be confused with random access semiconductor memory. RAM and random access files are in no way related.

The features which characterise direct files are as follows:

(a) Records do not have to be stored in any meaningful sequence.

(b) Any specified record can be accessed without reading any other records in the file.
(c) The time taken to retrieve any record does not depend on processing other records in the file. It depends only on the mechanics of the external store and the disk operating system.
(d) All records and fields must be of the same maximum length.

Use of direct files

Direct files are used in situations where rapid access to file information is required. It may be argued that rapid access is a desirable attribute in any situation. However, bearing in mind that 'nothing is for nothing', we should expect that there may be situations where the necessity for rapid access outweighs all other considerations. We could, for example, consider the case of a small electrical appliance shop which operates a file containing records of each item sold. The records may include the retail cost, VAT charged, the purchaser, length of guarantee, the date of sale and the initials of the assistant who conducted the sale. The computer would be permanently on during shop hours and running the direct access file in standby mode. Any sales would be entered not only in the cash register but also on the computer. If a faulty appliance is later returned for repair, the sales record can be quickly accessed, to determine whether the guarantee is still valid. Speed of access is essential because the customer, who may be irate and in need of pacification, will be waiting in the shop.

Relative files

Two versions of direct or random access files are supported by the Commodore 64 disk operating system. A species called *relative files* are more convenient for BASIC programs involving data handling in files. The other version of random access, although perhaps more potentially versatile, is primitive and thus more suited to machine code programming. For the remainder of this book we will concentrate on exploiting Commodore's 'relative' files as the random access framework for both indexed and direct files.

There is a set of rules governing the use of Commodore's relative files which are laid out in the 1541 disk manual. For reasons of clarity we will briefly recapitulate the essential details and add a few extra remarks.

The OPEN statement

In order to use relative files it is important that the command channel number 15 is OPENed prior to opening the main relative file itself. The command OPEN15,8,15 is sufficient for this. A relative file is OPENed initially at the file creation stage with a command such as:

```
OPEN6,8,6,"0:"+N$+",L,"+CHR$(L%)
```

where OPEN is followed by the file number,device number,channel number,drive number,file name(N$),file type(L denotes relative file) and finally CHR$(L%) where L% is the maximum record length (number of characters). This must be within the range 1 and 254. Notice that the file number and the channel number are chosen to be the same for reasons of simplicity. This reduces a lot of complication later on since some commands require the file number and some, the channel number. If these are the same the programmer need only remember the one number for the duration of writing of the program.

Once a relative file exists, the OPEN statement can be simplified to OPEN6,8,6,"0:"+N$ since the operating system will recognise the syntax as that of a relative file.

Record structure

In order to use relative files, fields and consequently records must be of fixed length. These are chosen at the file creation stage to be sufficient to accommodate the maximum expected number of characters in each field. This may appear a little complicated but enables direct access to any record in the file by setting a file pointer to any chosen record.

According to the 1541 disk manual, relative files are expected to be used in the following way. Records are written to the file into sequential record slots as they occur – that is to say, record 1,record 2,record 3, and so on. Records are transferred from buffer to disk only when sufficient records are entered to fill the buffer (256 bytes) or a CLOSE statement is encountered. Only record retrieval can be considered truly random access.

There appears to be a problem when writing records within the body of relative files, particularly when the chosen record slot happens to *span the junction between disk sectors*. The problem sometimes occurs when deletions, modifications or previously deleted record slots are reused. These occasional crashes usually manifest themselves in either a 'string too long' error or disk runaway when trying to retrieve certain records immediately after writing. Fortunately, there is a solution. We have found that if the relative file

is CLOSEd and reOPENed both *before and after writing records* within the body of the file the problem is cured. This perhaps inelegant solution is a small price to pay for the ability to modify and delete records directly within the file.

When fields in string form are output to disk with the PRINT# command, a carriage return is always needed to terminate it. Otherwise, when reading back, the INPUT# command will grab all characters till a carriage return is recognised. This could lead to a 'string too long' error condition if not realised. It is also recommended that an additional space of one character be left between individual fields. If these guidelines are followed, the maximum record length for the initial open statement above can be calculated by the following formula:

> Maximum record length = (Sum of field sizes) + (2 × number of fields)

Figure 6.2 shows how a record is stored within a relative file.

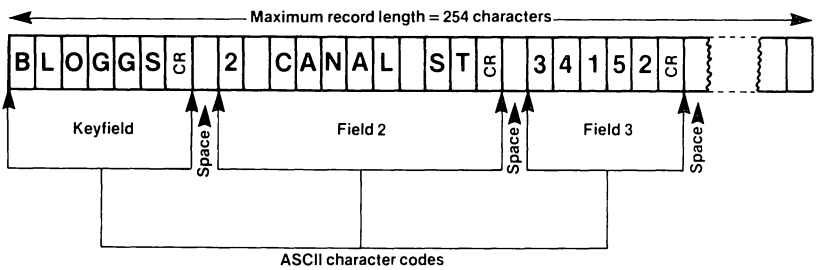


Fig. 6.2. How a record is stored in a relative file.

Relative file creation

Once the maximum record length has been determined the relative file can be set up on disk. The fastest way to do this is to decide on the maximum number of records likely to be in the file and write to the last (highest numbered record) first. This forces the disk operating system to allocate automatically the various record slots of lower numbered records by placing their start addresses into side sectors (a form of index). Subsequent access of a relative file is much faster if the above is adhered to. It is possible to extend the relative file, at a later date, but the operating system forces writing to the next higher record number.

Setting and calculating the file pointer

In order to read or write a record from/to a relative file, the file pointer statement must be previously set up. The syntax of this statement is as follows:

```
PRINT#15,"P"CHR$(6)CHR$(LB%)CHR$(HB%)CHR$
(FP%)
```

The command channel PRINT#15 command is followed by a P denoting 'pointer' and four character string numbers. The first is the channel number. This is followed by the record number in low byte, high byte form. Finally the optional field pointer can be assigned a value in the range (1 to 254) so that any individual field can be accessed directly on disk.

There are no DIV or MOD statements in Commodore 64 BASIC so the record number must be split into low byte, high byte form by alternative methods. A subroutine such as that below is the standard way of doing this:

```
1000 HB%=R%/256
1010 LB%=R%-HB%*256
1020 RETURN
```

This prepares the variables LB% and HB%, from the record number R%, for substitution in the file pointer statement above. Once the file pointer statement is set up any record R% in the file can be written or read directly.

A simple method of using relative files

The simplest method of using relative files is to store records as they occur in consecutive record slots. Immediate access to them can then be made by specifying a unique record number. Examples of this simple method appear in the disk manual so we will not treat it further.

Indexed relative files

As a file becomes larger, access by record number may not be convenient since the user will need to associate a unique record number to the particular record required. The key field usually has the highest mnemonic value in the record, as far as the user is concerned, so it would be useful if records could be accessed in this way. The problem can be overcome at the expense of a little more

disk storage space, by maintaining a separate index file of the type shown in Fig. 6.3.

An index file is very much smaller than the file to which it relates. It need only be a table consisting of the barest minimum of information necessary to access a record. The format usually consists of the *key field* along with the *record number* on disk.

INDEX FILE	
Key	Record number
Albania	1
Denmark	2
Belgium	3
Bulgaria	4
Austria	5
Spain	6
Hungary	7

MAIN RELATIVE FILE			
Country Field 1	Area Field 2	Population Field 3	Capital Field 4
Albania	10700	2594600	Tirana
Denmark	17000	15081747	Copenhagen
Belgium	11800	9837413	Brussels
Bulgaria	43000	8730000	Sofia
Austria	32376	7508400	Vienna
Spain	197000	36240000	Madrid
Hungary	36000	10631000	Budapest

Fig. 6.3. Simple indexed relative file organisation.

The index file

It is always desirable for the index file to fit into RAM at one sitting, even though the file itself must be stored on disk. This organisation obviously has its access speed advantages. If this is not possible it can remain either partly or entirely on disk. From now on we will assume that the index file is read into RAM in its entirety at each session.

If this is the case, then it will be easy to perform a fast internal sort of the index so that records can be accessed in ascending or descending order. Alternative methods involving external (disk) sorting are extremely slow on current microcomputer systems and should be avoided if at all possible. Your mechanical disk drive could well require a few spares after a couple of weeks' use. A further advantage in keeping the index file ordered is that the binary search algorithm can be employed to advantage.

A practical indexed filing program

Program 6.1 is a practical example on implementing indexed relative

files on the Commodore 64. The program uses a linear search technique to look up the RAM resident index. Provision is made to sort the index at a reasonable speed, thus allowing records to be accessed in order if desired.

```
10 REM INDEXED FILING SYSTEM
20 REM VERSION 1
30 OPEN15,8,15
40 GOSUB14000:PRINT#15,"I":GOSUB5000:PRI
NT:PRINT
50 PRINT"(1) CREATE NEW FILE"
60 PRINT"(2) ACCESS EXISTING FILE"
70 PRINT"(3) EXIT PROGRAM"
80 PRINT:PRINT
90 INPUT"SELECT OPTION";S%
100 IF S%<1 OR S%>3 THEN 40
110 ON S% GOSUB2000,1000,280
120 PRINT CHR$(147)
130 OPEN6,8,6,"0:"+N$:GOSUB5000
140 GOSUB14000:PRINT:PRINT
150 PRINT "(1) ENTER A RECORD"
160 PRINT "(2) DISPLAY A RECORD"
170 PRINT "(3) DELETE A RECORD"
180 PRINT "(4) MODIFY A FIELD"
190 PRINT "(5) SORT INDEX"
200 PRINT "(6) DISPLAY FILE"
210 PRINT "(7) CHECK INDEX BYTES FREE"
220 PRINT "(8) END PROGRAM : SAVE INDEX"
230 PRINT:PRINT:INPUT "SELECT OPTION";S%
240 IF S%<1 OR S%>8 THEN 140
250 ON S% GOSUB 6000,7000,8000,9000,1000
0,11000,23000,270
260 GOTO140
270 CLOSE6:GOSUB3000:CLOSE15
280 END
997 REM *
998 REM **
999 REM LOAD INDEX SUBROUTINE
1000 PRINT CHR$(147):GOSUB4000
1010 OPEN 5,8,5,"0:I."+N$+",S,R"
1020 GOSUB5000:PRINTCHR$(147):PRINT"READ
ING HEADING/INDEX FILE"
1030 INPUT#5,NF%,FS%
1040 DIM I$(1,FS%)
1050 FOR F=1 TO NF%
```

```
1060 INPUT#5,H$(F),W(F),P(F)
1070 NEXT
1080 INPUT#5,L%
1090 FOR R=1 TO L%-1
1100 FOR C=0 TO 1
1110 INPUT#5,I$(C,R)
1120 NEXT:NEXT
1130 CLOSE5
1140 RETURN
1997 REM *
1998 REM **
1999 REM CREATE FILE SUBROUTINE
2000 PRINT CHR$(147):GOSUB4000
2010 PRINT"ENTER FILE SIZE (NUMBER OF RE
CORDS)"
2020 INPUT FS%
2030 IF FS%<1 THEN 2010
2040 PRINT"ENTER NUMBER OF FIELDS REQUIR
ED"
2050 INPUT NF%
2060 IF NF%<2 OR NF%>9 THEN PRINT"FIELDS
(2-9) ONLY":GOTO2040
2070 DIM I$(1,FS%):P(1)=1:L%=1
2080 FOR F=1 TO NF%
2090 PRINT CHR$(147)
2100 PRINT"ENTER FIELD HEADING";F
2110 W=255:GOSUB22000:H$(F)=K$
2120 PRINT"ENTER FIELD WIDTH (CHARACTERS
)"
2130 INPUT W(F)
2140 IF W(F)<1 THEN 2120
2150 P(F+1)=P(F)+W(F)+2
2160 NEXT
2170 IF P(NF%+1) > 254 THEN PRINT"TOO LO
NG : START AGAIN":GOSUB15000:GOTO2070
2180 PRINT CHR$(147):PRINT"CREATING MAIN
 FILE ";N$
2190 OPEN 6,8,6,"0:"+N$+",L,"+CHR$(P(NF%
+1))
2200 GOSUB5000
2210 R%=R:GOSUB12000:GOSUB13000
2220 PRINT#6,"END"
2230 CLOSE6
2240 OPEN 6,8,6,"0:"+N$
2250 GOSUB6000
```

```
2260 CLOSE6
2270 RETURN
2997 REM *
2998 REM **
2999 REM SAVE INDEX SUBROUTINE
3000 PRINT CHR$(147)
3010 OPEN 5,8,5,"@0:I."+N$+",S,W"
3020 GOSUB5000:PRINT"WRITING HEADING/IND
EX FILE"
3030 PRINT#5,NF%:PRINT#5,FS%
3040 FOR F=1 TO NF%
3050 PRINT#5,H$(F):PRINT#5,W(F):PRINT#5,
P(F)
3060 NEXT
3070 PRINT#5,L%
3080 GOSUB5000
3090 FOR R=1 TO L%-1
3100 FOR C=0 TO 1
3110 PRINT#5,I$(C,R)
3120 NEXT:NEXT
3130 CLOSE5
3140 RETURN
3997 REM *
3998 REM **
3999 REM GET FILE NAME SUBROUTINE
4000 PRINT"ENTER FILE NAME"
4010 W=14:GOSUB22000:N$=K$
4020 RETURN
4997 REM *
4998 REM **
4999 REM READ ERROR CHANNEL SUBROUTINE
5000 INPUT#15,A,B$,C,D
5020 IF A<20 OR A=50 THEN 5040
5030 PRINT CHR$(147):PRINTB$:GOSUB15000:
RUN
5040 RETURN
5997 REM *
5998 REM **
5999 REM ENTER A RECORD SUBROUTINE
6000 PRINT CHR$(147)
6010 PRINT"ENTER A RECORD"
6020 PRINT:PRINT
6030 FOR F=1 TO NF%
6040 PRINT"MAXIMUM CHARACTERS";W(F)
6050 PRINT "ENTER ";H$(F)
```

```
6060 W=W(F):GOSUB22000:F$(F)=K$
6070 NEXT
6080 GOSUB16000
6090 RETURN
6997 REM *
6998 REM **
6999 REM RETRIEVE RECORD SUBROUTINE
7000 PRINT CHR$(147):PRINT"ENTER KEY FIE
LD"
7010 FL%=0
7020 W=255:GOSUB22000:KY$=K$
7030 PRINT CHR$(147):PRINT"RETRIEVING RE
CORD"
7040 J%=1
7060 IF KY$=I$(0,J%) THEN R%=VAL(I$(1,J%
)):GOSUB19000:GOSUB20000:GOTO7100
7070 J%=J%+1
7080 IF J%>FS% THEN PRINT"RECORD NOT ON
FILE":FL%=1:GOSUB15000:GOTO7100
7090 GOTO7060
7100 RETURN
7997 REM *
7998 REM **
7999 REM DELETE RECORD SUBROUTINE
8000 GOSUB7000:PRINT
8010 IF FL%=1 THEN 8120
8020 PRINT"DO YOU WISH TO DELETE THIS RE
CORD (Y/N)"
8030 INPUT K$
8040 IF K$="N" THEN 8120
8050 IF K$="Y" THEN 8070
8060 GOTO8020
8070 FOR F=1 TO NF%
8080 F$(F)=CHR$(97)
8090 NEXT
8100 I$(0,J%)=CHR$(97)
8110 GOSUB21000
8120 RETURN
8997 REM *
8998 REM **
8999 REM MODIFY RECORD SUBROUTINE
9000 GOSUB7000:IF FL%=1 THEN 9100
9010 PRINT:K=0
9020 PRINT"ENTER HEADING OF FIELD TO BE
MODIFIED"
```

```
9030 W=255:GOSUB22000:M$=K$
9040 FOR F=1 TO NF%
9050 IF M$=H$(F) THEN PRINT"ENTER NEW CO
NTENTS":W=W(F):GOSUB22000:F$(F)=K$:K=F
9060 NEXT
9070 IF K=0 THEN PRINT"NO SUCH FIELD":GO
SUB15000:GOTO9100
9080 IF K=1 THEN I$(0,J%)=F$(1)
9090 GOSUB21000
9100 RETURN
9997 REM *
9998 REM **
9999 REM SORT INDEX SUBROUTINE
10000 PRINT CHR$(147):IF L%<=2 THEN 1016
0
10010 PRINT"SORTING INDEX"
10030 N%=L%-1
10040 N%=(N%+2)/3
10050 FOR D=N%+1 TO N%*2
10060 FOR E=D TO L%-1 STEP N%
10070 FOR R=E TO D STEP -N%
10080 IF I$(0,R)>I$(0,R-N%) THEN GOTO101
30
10090 FOR C=0 TO 1
10100 K$=I$(C,R):I$(C,R)=I$(C,R-N%):I$(C
,R-N%)=K$
10110 NEXT
10120 NEXT
10130 NEXT E
10140 NEXT
10150 IF N%>1 THEN 10040
10160 RETURN
10997 REM *
10998 REM **
10999 REM DISPLAY FILE SUBROUTINE
11000 J%=0
11010 J%=J%+1
11020 R%=VAL(I$(1,J%))
11030 GOSUB19000
11040 GOSUB20000
11050 PRINT:PRINT
11060 PRINT"PRESS SPACE BAR TO STEP THRO
UGH FILE"
11070 PRINT"PRESS ANY OTHER KEY TO QUIT"
11080 GET K$
```

```
11090 IF K$="" THEN 11080
11100 IF J%=L%-1 THEN 11120
11110 IF K$=CHR$(32) THEN 11010
11120 RETURN
11997 REM *
11998 REM **
11999 REM CALCULATE FILE POINTER
12000 HB%=R%/256
12010 LB%=R%-HB%*256
12020 RETURN
12997 REM *
12998 REM **
12999 REM SET FILE POINTER
13000 PRINT#15,"P"CHR$(6)CHR$(LB%)CHR$(H
B%)CHR$(P(F))
13010 RETURN
13997 REM *
13998 REM **
13999 REM TITLE SUBROUTINE
14000 PRINT CHR$(147)
14010 PRINT"INDEXED FILING SYSTEM"
14020 RETURN
14997 REM *
14998 REM **
14999 REM WAIT FOR ANY KEY PRESSED
15000 PRINT:PRINT"PRESS ANY KEY TO CONTI
NUE"
15010 GET K$
15020 IF K$="" THEN 15010
15030 RETURN
15997 REM *
15998 REM **
15999 REM STORE RECORD SUBROUTINE
16000 PRINT CHR$(147)
16010 GOSUB17000
16020 IF T%=1 THEN PRINT"REJECTED : KEY
FIELD NOT UNIQUE":GOSUB15000:GOTO16060
16030 IF L%>FS% AND T%=0 THEN PRINT"REJE
CTED FILE FULL":GOSUB15000:GOTO16060
16040 IF T%=0 THEN R%=L%:GOSUB21000:GOSU
B18000
16050 IF T%=-1 THEN I$(0,I%)=F$(1):R%=VA
L(I$(1,I%)):GOSUB21000
16060 RETURN
16997 REM *
```

```
16998 REM **
16999 REM CHECK KEY (UNIQUE/TOMBSTONE)
17000 T%=0:J%=0
17010 J%=J%+1
17020 IF F$(1)=I$(0,J%) THEN T%=1:GOTO17
050
17030 IF I$(0,J%)=CHR$(97) THEN T%=-1:I%
=J%
17040 IF J%<L%-1 THEN 17010
17050 RETURN
17997 REM *
17998 REM **
17999 REM UPDATE INDEX SUBROUTINE
18000 I$(0,L%)=F$(1)
18010 I$(1,L%)=STR$(L%)
18020 L%=L%+1
18030 RETURN
18997 REM *
18998 REM **
18999 REM READ RECORD SUBROUTINE
19000 GOSUB12000
19010 FOR F=1 TO NF%
19020 GOSUB13000
19030 INPUT#6,F$(F)
19040 NEXT
19050 RETURN
19997 REM *
19998 REN **
19999 REM DISPLAY RECORD SUBROUTINE
20000 PRINTCHR$(147):PRINT"FILE ACCESSED
 :";N$
20010 PRINT"DISPLAY OF RECORD NUMBER";R%
20020 PRINT:PRINT
20030 FOR F=1 TO NF%
20040 PRINT H$(F),F$(F)
20050 NEXT
20060 IF S%=2 THEN GOSUB15000
20070 RETURN
20997 REM *
20998 REM **
20999 REM WRITE RECORD SUBROUTINE
21000 IF R%<L% THEN GOSUB24000
21010 GOSUB12000
21020 FOR F=1 TO NF%
21030 GOSUB13000
```

```
21040 PRINT#6,F$(F)
21050 NEXT
21060 IF R%<L% THEN GOSUB24000
21070 RETURN
21997 REM *
21998 REM **
21999 REM INPUT VALIDATION SUBROUTINE
22000 K$="":INPUT K$
22010 IF K$="" THEN 22000
22020 IF LEN(K$)>W THEN K$=LEFT$(K$,W)
22030 RETURN
22997 REM *
22998 REM **
22999 REM BYTES FREE SUBROUTINE
23000 PRINT CHR$(147):PRINT"WAIT":PRINT
23010 X=FRE(0)-(SGN(FRE(0))<0)*65535
23020 PRINT"NUMBER OF INDEX BYTES FREE";
X
23030 GOSUB15000
23040 RETURN
23997 REM *
23998 REM **
23999 REM REM CLEAR OUT BUFFER TO DISC
24000 CLOSE6
24010 OPEN6,8,6,"0:"+N$
24020 RETURN
```

Program 6.1. An indexed filing system.

Using the program

Before running the program, a disk must be in place, referred to from now on as the 'work disk'. Some of the following details, relating to the use of the program, are much the same as given in the RAM-based file (Program 4.1) described in Chapter 4. In spite of the similarities, there is one major difference. Program 4.1 required the entire file to be resident in RAM which naturally imposes a severe restriction on the file length. Program 6.1 works quite differently. Individual records are transferred between disk and RAM as and when needed. At any one time, the only records likely to be in RAM are those which fit into the disk buffer area – probably three or four, depending on the size of each record.

For example, when adding records to a file, they are first sent to the buffer area associated with the appropriate channel but when the buffer eventually becomes full, the contents are disgorged to disk.

This action is quite automatic. In fact, the only indication that buffer to disk transfer is taking place are the purrs and clicks emanating from the drive mechanism. Because the files in Program 6.1 are store-based the length of a file is limited only by the capacity of a disk.

Creating a new file

On first running the program, the primary menu is displayed and you are asked whether you want to access an existing file or want to create a new file. If you have only just keyed in the program the only choice is to create a new file in which case you will be asked for the file name, which must not exceed 14 characters (two are added by the program as a prefix). You are informed if a file already exists under that file name. The next prompts to be answered concern the number of records and the number of fields. After asking for field headings and maximum field widths the relative file is created. The maximum field widths should be carefully thought out, at this stage, since it will not be possible to change them at a later date. The program asks for the first record to be entered at the file creation stage to start off the index file. The above heading information is always saved along with the index file as a serial file. This file is likely to grow in size so it is customary, where possible, to have only one data file and its associated index per work disk.

Option 1: Enter a record

This option will be used immediately after the file has been created or to add further records to an existing file. Enter the record as prompted a field at a time. After a record is entered, the record is stored on disk, the index is updated and you are returned to the menu. If you want to add another record, you must select Option 1 again and so on. If you are having a long session of record entering, you will occasionally hear a comforting purr from the disk drive as the buffer text is transferred to disk. Ignore it but be thankful for the sound, because it means your data is being preserved. Remember, you should get it every couple of hundred characters or so. If you don't, then it's time to get suspicious. To avoid much teeth-gnashing later, try out the program for the first time by composing a short dummy file of about six records and load one or two of the records back again with the aid of Option 2. This will confirm whether or not the disk transfers are foolproof. If numerical key fields are required, remember that each key must have the same number of digits (including leading zeros). If this guidance is ignored then the index sort (option 5) will not work properly.

Option 2: Display a record

This will perform a sequential search of the index until the required key field is found. The full and exact key field must be supplied – abbreviations are not accepted. If the key field is matched in the index, the file pointer is set to the appropriate record number and the specified record displayed. If the record is not found, the message 'Record not on file' is displayed and the menu regained.

Option 3: Delete a record

The requested record is displayed in a similar manner to option 2 above. An appropriate message is displayed if the record is not on file. Following the record a message 'Do you wish to delete this record (Y/N)' is arranged. If the user replies with 'Y' then the record is deleted and the menu regained. A reply of 'N' returns the menu immediately.

Option 4: Modify a field

This is a very useful option that enables the user to modify any field of any record on file. After prompting the user for the key field, the requested record is displayed on the screen. The user is then prompted for the heading of the field to be modified followed by the new field contents. The new field is placed in the record and the modified record rewritten to disk.

Appropriate error messages are displayed if the user enters a non-existent field heading or key field.

Option 5: Sort index

This option executes a reasonably fast sort of the index by key field. The algorithm is the Shellsort treated in Chapter 5.

Option 6: Display file

This option enables records to be displayed one after the other as determined by the order of the index. If the index is first sorted via the use of option 5 then the complete file can be displayed in alphabetical or numerical order. The file can be stepped through, one record at a time, starting from the first by the use of the space bar. A return to the menu is effected on pressing any other key.

Option 7: Check index bytes free

This option displays how much memory is left for the index. It is well to check this now and again when a file becomes large. To err on the safe side, never let the number of bytes free fall below about 1000.

It may appear 'unfriendly' to leave the responsibility of memory availability to the user. However, due to the time taken to execute the FRE(0) statement on the Commodore 64 it was felt that this method of keeping tabs on the index length was preferable to a fully automatic method.

Option 8: Exit program
Never exit the program by any means other than selecting this option. It is not just a simple exit. Amongst other things, it ensures that the updated index file and any data, still left in the buffers, is transferred to disk. If you leave the program by switching off the power or pressing the STOP/RESTORE keys the file will become invalid and/or corrupted.

How the program works

We will start by giving a list of the variables used in Program 6.1 followed by a table of subroutine structures:

FS%=maximum file size (maximum number of records).
W(F)=width of field F.
F$(F)=is actual field in RAM.
P(F)=pointer to start of field.
NF%=number of fields.
B$=disk error message.
N$=file name.
S%=option selected.
H$(F)=heading of field F.
R%=record number.
A=error number.
HB%=high byte address.
LB%=low byte address.
K$=general purpose variable.
KY$=key field.
T%=Unique/tombstone status flag.
I$(1,FS%)=two-dimensional index array.
L%=current file length.
X=bytes free.
W=temporary variable used to store current field width.
J%=general-purpose variable.
D,E,R,N%,C=loop counters used in Shellsort.
M$=temporary variable.

The main and lower level subroutines used in Program 6.1 are shown in Table 6.1.

Table 6.1. Subroutines used in Program 6.1.

1st level	2nd and 3rd level
Create file	Get file name
	Input validation
	Wait for any key
	Read error channel
	Calculate file pointer
	Set file pointer
	Enter a record
	Store record
	Check key (unique/tombstone)
Access existing file	Load index
	Get file name
	Read error channel
Enter a record	Input validation
	Store record
	Check key field (unique/tombstone)
	Wait for any key
	Write record
	Calculate file pointer
	Set file pointer
	Read error channel
	Update index
Display a record	Retrieve record
	Input validation
	Read record
	Display record
	Wait for any key
Delete a record	Delete record
	Retrieve record
	Input validation
	Read record
	Display record
	Wait for any key
	Write record
	Calculate file pointer
	Set file pointer
	Read error channel
	Update index

Modify a field	Modify record
	Retrieve record
	Input validation
	Read record
	Display record
	Input validation
	Wait for any key
	Write record
Sort index	
Display file	Read record
	Calculate file pointer
	Set file pointer
	Display record
Check index bytes free	Wait for any key
End program	Save index

File creation

In this program we have chosen to incorporate the header file and the index into a single file in Program 6.1. The choice of filenames are not important but it is suggested that the prefix 'I.' be used for the index filename. For instance, if we have a relatively small file on, say, types of Pachyderm then the main file can be stored under the filename 'SPECIES' and the corresponding index file under the name 'I.SPECIES'. The 'create file subroutine' controls all the file creation complexities. The error channel is read each time a file is opened, thus informing the user of the kind of mistake made, if any.

Loading the index file

The serial index file is arranged to include the index itself as well as heading and file information such as field headings, widths, pointers, etc. The index information, loaded into RAM, is contained within a two-dimensional string array. It is important that the array is first DIMensioned I$(1,FS%). Many other methods of accommodating the index in RAM could be used but this is as convenient as any. The subroutine which loads the index file is, not surprisingly, labelled the 'load index subroutine'.

Saving the index file

At the end of a record entering session the updated index will need to be resaved on disk. Notice the use of the '@' character preceding the

filename. This ensures that the old index file is overwritten by the latest version on disk. Consequently, the only way to exit the program should be via Option 8 of the secondary menu which automatically writes the current index file to disk by invoking the subroutine labelled SAVE INDEX.

Entering records

Records are entered a field at a time into the small array F$ by calling the 'enter a record' subroutine. When the record has been received by the user the 'store record' subroutine is called. A temporary excursion is made to the 'check key (unique/tombstone)' subroutine to see if the key field of the entered record has been previously used. If it has, the entire record will be rejected. If not, the record will be written to disk. A call to 'update index' subroutine will add the corresponding index entry to the index array.

The check key (unique/tombstone) subroutine

This subroutine performs a linear search of the key field index array and sets a flag T% as follows:

T%=1 : Key field of new record entered by user is not unique (used before).
T%=0 : Record acceptable for storage.
T%=−1 : Tombstone character found in index. A tombstone (spade character) signifies a previously deleted record space available for re-use.

Store record subroutine

This subroutine acts on the status flag T% set as described above. If T%=1, then a message 'Rejected : key field not unique' is displayed. If T%=0, then the record is stored and the index updated. If T%=−1 then the key field of the entered record replaces the tombstone marker in the index array. The complete new record is stored at the corresponding record number of the previously deleted record in the main file. Finally, if the index has equalled its DIMensioned maximum, a 'Rejected : file full' message is displayed.

Update index subroutine

This is a simple subroutine which places the key field of the entered record F$(1) into the index array I$(0,L%) list. The corresponding record number L% is converted to string form, via the STR$ statement, and stored in the corresponding I$(1,L%) list of the index array. Note that the I$(0,L%) entries will be the key fields and the INDEX$(1,L%) entries will be the corresponding record numbers.

Retrieving records

The process of retrieving a record is handled by the 'retrieve record' subroutine. You will be asked to enter the key field. A linear search of the index array will be made for a match. If a match is found the file pointer will be set to the appropriate record number and the record directly accessed by the 'read record' subroutine. The subroutine labelled 'display record subroutine' will ultimately place the desired record on the screen. If the requested key field is not found in the index, the message 'record not on file' is displayed.

Autosort indexed filing system

Although useful, Program 6.1 is capable of further improvement. Perhaps the most beneficial is the incorporation of machine code sorting and binary searching of the index. Program 6.2 can be used as the basis of a ultra-fast and powerful filing system which, to a large extent, masks the often criticised disk transfer rate of the Commodore 1541 disk system. We think you will find it well worth the additional effort of typing in, and perhaps improving, this more complex version. A print record/file option to suit your particular printer is a suggested project worth tackling, once you have both Programs 5.10 and 6.2 saved and tested.

Machine code sorting

As far as sorting the index is concerned, we already have a suitable machine code solution in the form of the diminishing increment sort of a two-dimensional array (Program 5.10). Program 5.10 must be RUN to load the machine code bytes into memory at address $C000 (49152 decimal) for use by the filing program. Once loaded, the code will be present until the Commodore 64 is switched off. Program 6.2 can then be LOADed and RUN.

Note: It is important to remember that if the code of Program 5.10 is to be successful, the index file I$(1,FS%) must be the first array DIMensioned in the filing program.

```
10 REM INDEXED FILING SYSTEM
20 REM MACHINE CODE SORT-BINARY SEARCH
30 OPEN15,8,15
40 GOSUB14000:PRINT#15,"I":GOSUB5000:PRI
NT:PRINT
50 PRINT"(1) CREATE NEW FILE"
```

```
60 PRINT"(2) ACCESS EXISTING FILE"
70 PRINT"(3) EXIT PROGRAM"
80 PRINT:PRINT
90 INPUT"SELECT OPTION";S%
100 IF S%<1 OR S%>3 THEN 40
110 ON S% GOSUB2000,1000,270
120 PRINT CHR$(147)
130 OPEN6,8,6,"0:"+N$:GOSUB5000
140 GOSUB14000:PRINT:PRINT
150 PRINT "(1) ENTER A RECORD"
160 PRINT "(2) DISPLAY A RECORD"
170 PRINT "(3) DELETE A RECORD"
180 PRINT "(4) MODIFY A FIELD"
190 PRINT "(5) DISPLAY FILE"
200 PRINT "(6) CHECK INDEX BYTES FREE"
210 PRINT "(7) EXIT PROGRAM : SAVE INDEX
"
220 PRINT:PRINT:INPUT "SELECT OPTION";S%
230 IF S%<1 OR S%>7 THEN 140
240 ON S% GOSUB 6000,7000,8000,9000,1100
0,23000,260
250 GOTO140
260 CLOSE6:GOSUB3000:CLOSE15
270 END
997 REM *
998 REM **
999 REM LOAD INDEX SUBROUTINE
1000 PRINT CHR$(147):GOSUB4000
1010 OPEN 5,8,5,"0:I."+N$+",S,R"
1020 GOSUB5000:PRINTCHR$(147):PRINT"READ
ING HEADING/INDEX FILE"
1030 INPUT#5,NF%,FS%
1040 DIM I$(1,FS%)
1050 FOR F=1 TO NF%
1060 INPUT#5,H$(F),W(F),P(F)
1070 NEXT
1080 INPUT#5,L%
1090 FOR R=1 TO L%-1
1100 FOR C=0 TO 1
1110 INPUT#5,I$(C,R)
1120 NEXT:NEXT
1130 CLOSE5
1140 RETURN
1997 REM *
1998 REM **
```

```
1999 REM CREATE FILE SUBROUTINE
2000 PRINT CHR$(147):GOSUB4000
2010 PRINT"ENTER FILE SIZE (NUMBER OF RE
CORDS)"
2020 INPUT FS%
2030 IF FS%<1 THEN 2010
2040 PRINT"ENTER NUMBER OF FIELDS REQUIR
ED"
2050 INPUT NF%
2060 IF NF%<2 OR NF%>9 THEN PRINT"FIELDS
(2-9) ONLY":GOTO2040
2070 DIM I$(1,FS%):P(1)=1:L%=1
2080 FOR F=1 TO NF%
2090 PRINT CHR$(147)
2100 PRINT"ENTER FIELD HEADING";F
2110 W=255:GOSUB22000:H$(F)=K$
2120 PRINT"ENTER FIELD WIDTH (CHARACTERS
)"
2130 INPUT W(F)
2140 IF W(F)<1 THEN 2120
2150 P(F+1)=P(F)+W(F)+2
2160 NEXT
2170 IF P(NF%+1) > 254 THEN PRINT"TOO LO
NG : START AGAIN":GOSUB15000:GOTO2070
2180 PRINT CHR$(147):PRINT"CREATING MAIN
 FILE ";N$
2190 OPEN 6,8,6,"0:"+N$+",L,"+CHR$(P(NF%
+1))
2200 GOSUB5000
2210 R%=R:GOSUB12000:GOSUB13000
2220 PRINT#6,"END"
2230 CLOSE6
2240 OPEN6,8,6,"0:"+N$
2250 GOSUB6000
2260 CLOSE6
2270 RETURN
2997 REM *
2998 REM **
2999 REM SAVE INDEX SUBROUTINE
3000 PRINT CHR$(147)
3010 OPEN 5,8,5,"@0:I."+N$+",S,W"
3020 GOSUB5000:PRINT"WRITING HEADING/IND
EX FILE"
3030 PRINT#5,NF%:PRINT#5,FS%
3040 FOR F=1 TO NF%
```

```
3050 PRINT#5,H$(F):PRINT#5,W(F):PRINT#5,
P(F)
3060 NEXT
3070 PRINT#5,L%
3080 GOSUB5000
3090 FOR R=1 TO L%-1
3100 FOR C=0 TO 1
3110 PRINT#5,I$(C,R)
3120 NEXT:NEXT
3130 CLOSE5
3140 RETURN
3997 REM *
3998 REM **
3999 REM GET FILE NAME SUBROUTINE
4000 PRINT"ENTER FILE NAME"
4010 W=14:GOSUB22000:N$=K$
4020 RETURN
4997 REM *
4998 REM **
4999 REM READ ERROR CHANNEL SUBROUTINE
5000 INPUT#15,A,B$,C,D
5020 IF A<20 OR A=50 THEN 5040
5030 PRINT CHR$(147):PRINTB$:GOSUB15000:
RUN
5040 RETURN
5997 REM *
5998 REM **
5999 REM ENTER A RECORD SUBROUTINE
6000 PRINT CHR$(147)
6010 PRINT"ENTER A RECORD"
6020 PRINT:PRINT
6030 FOR F=1 TO NF%
6040 PRINT"MAXIMUM CHARACTERS";W(F)
6050 PRINT "ENTER ";H$(F)
6060 W=W(F):GOSUB22000:F$(F)=K$
6070 NEXT
6080 GOSUB16000
6090 RETURN
6997 REM *
6998 REM **
6999 REM RETRIEVE RECORD SUBROUTINE
7000 PRINT CHR$(147):PRINT"ENTER KEY FIE
LD"
7010 FL%=0
7020 W=255:GOSUB22000:KY$=K$
```

```
7030 PRINT CHR$(147):PRINT"RETRIEVING RE
CORD"
7040 LO%=1:HI%=L%-1
7050 J%=(LO%+HI%)/2
7060 IF KY$=I$(0,J%) THEN R%=VAL(I$(1,J%
)):GOSUB19000:GOSUB20000:GOTO7110
7070 IF KY$<I$(0,J%) THEN HI%=J%-1
7080 IF KY$>I$(0,J%) THEN LO%=J%+1
7090 IF LO%>HI% THEN PRINT"RECORD NOT ON
 FILE":FL%=1:GOSUB15000:GOTO7110
7100 GOTO7050
7110 RETURN
7997 REM *
7998 REM **
7999 REM DELETE RECORD SUBROUTINE
8000 GOSUB7000:PRINT
8010 IF FL%=1 THEN 8120
8020 PRINT"DO YOU WISH TO DELETE THIS RE
CORD (Y/N)"
8030 INPUT K$
8040 IF K$="N" THEN 8120
8050 IF K$="Y" THEN 8070
8060 GOTO8020
8070 FOR F=1 TO NF%
8080 F$(F)=CHR$(97)
8090 NEXT
8100 I$(0,J%)=CHR$(97)
8110 GOSUB21000
8120 RETURN
8997 REM *
8998 REM **
8999 REM MODIFY RECORD SUBROUTINE
9000 GOSUB7000:IF FL%=1 THEN 9100
9010 PRINT:K=0
9020 PRINT"ENTER HEADING OF FIELD TO BE
MODIFIED"
9030 W=255:GOSUB22000:M$=K$
9040 FOR F=1 TO NF%
9050 IF M$=H$(F) THEN PRINT"ENTER NEW CO
NTENTS":W=W(F):GOSUB22000:F$(F)=K$:K=F
9060 NEXT
9070 IF K=0 THEN PRINT"NO SUCH FIELD":GO
SUB15000:GOTO9100
9080 IF K=1 THEN I$(0,J%)=F$(1)
9090 GOSUB21000
```

```
9100 RETURN
9997 REM *
9998 REM **
9999 REM SORT INDEX SUBROUTINE
10000 IF L%<=2 THEN 10070
10010 R%=L%-1:GOSUB12000
10020 POKE251,LB%
10030 POKE252,HB%
10040 POKE253,1
10050 POKE254,0
10060 SYS49152
10070 RETURN
10997 REM *
10998 REM **
10999 REM DISPLAY FILE SUBROUTINE
11000 J%=0
11010 J%=J%+1
11020 R%=VAL(I$(1,J%))
11030 GOSUB19000
11040 GOSUB20000
11050 PRINT:PRINT
11060 PRINT"PRESS SPACE BAR TO STEP THRO
UGH FILE"
11070 PRINT"PRESS ANY OTHER KEY TO QUIT"
11080 GET K$
11090 IF K$="" THEN 11080
11100 IF J%=L%-1 THEN 11120
11110 IF K$=CHR$(32) THEN 11010
11120 RETURN
11997 REM *
11998 REM **
11999 REM CALCULATE FILE POINTER
12000 HB%=R%/256
12010 LB%=R%-HB%*256
12020 RETURN
12997 REM *
12998 REM **
12999 REM SET FILE POINTER
13000 PRINT#15,"P"CHR$(6)CHR$(LB%)CHR$(H
B%)CHR$(P(F))
13010 RETURN
13997 REM *
13998 REM **
13999 REM TITLE SUBROUTINE
14000 PRINT CHR$(147)
```

```
14010 PRINT"INDEXED AUTOSORT FILING SYST
EM"
14020 RETURN
14997 REM *
14998 REM **
14999 REM WAIT FOR ANY KEY PRESSED
15000 PRINT:PRINT"PRESS ANY KEY TO CONTI
NUE"
15010 GET K$
15020 IF K$="" THEN 15010
15030 RETURN
15997 REM *
15998 REM **
15999 REM STORE RECORD SUBROUTINE
16000 PRINT CHR$(147):PRINT"STORING RECO
RD"
16010 GOSUB17000
16020 IF T%=1 THEN PRINT"REJECTED : KEY
FIELD NOT UNIQUE":GOSUB15000:GOTO16060
16030 IF L%>FS% AND T%=0 THEN PRINT"REJE
CTED FILE FULL":GOSUB15000:GOTO16060
16040 IF T%=0 THEN R%=L%:GOSUB18000:GOSU
B21000
16050 IF T%=-1 THEN I$(0,L%-1)=F$(1):R%=
VAL(I$(1,L%-1)):GOSUB21000
16060 RETURN
16997 REM *
16998 REM **
16999 REM CHECK KEY (UNIQUE/TOMBSTONE)
17000 LO%=1:HI%=L%-1
17010 J%=(LO%+HI%)/2
17020 IF F$(1)=I$(0,J%) THEN T%=1:GOTO17
080
17030 IF F$(1)<I$(0,J%) THEN HI%=J%-1
17040 IF F$(1)>I$(0,J%) THEN LO%=J%+1
17050 IF LO%>HI% THEN T%=0:GOTO17070
17060 GOTO17010
17070 IF I$(0,L%-1)=CHR$(97) THEN T%=-1
17080 RETURN
17997 REM *
17998 REM **
17999 REM UPDATE INDEX SUBROUTINE
18000 I$(0,L%)=F$(1)
18010 I$(1,L%)=STR$(L%)
18020 L%=L%+1
```

```
18030 RETURN
18997 REM *
18998 REM **
18999 REM READ RECORD SUBROUTINE
19000 GOSUB12000
19010 FOR F=1 TO NF%
19020 GOSUB13000
19030 INPUT#6,F$(F)
19040 NEXT
19050 RETURN
19997 REM *
19998 REM **
19999 REM DISPLAY RECORD SUBROUTINE
20000 PRINT CHR$(147):PRINT"FILE ACCESSE
D :";N$
20010 PRINT"DISPLAY OF RECORD NUMBER";R%
20020 PRINT:PRINT
20030 FOR F=1 TO NF%
20040 PRINT H$(F),F$(F)
20050 NEXT
20060 IF S%=2 THEN GOSUB15000
20070 RETURN
20997 REM *
20998 REM **
20999 REM WRITE RECORD SUBROUTINE
21000 IF R%<L%-1 THEN GOSUB24000
21010 GOSUB12000
21020 FOR F=1 TO NF%
21030 GOSUB13000
21040 PRINT#6,F$(F)
21050 NEXT
21060 IF R%<L%-1 THEN GOSUB24000
21070 GOSUB10000
21080 RETURN
21997 REM *
21998 REM **
21999 REM INPUT VALIDATION SUBROUTINE
22000 K$="":INPUT K$
22010 IF K$="" THEN 22000
22020 IF LEN(K$)>W THEN K$=LEFT$(K$,W)
22030 RETURN
22997 REM *
22998 REM **
22999 REM BYTES FREE SUBROUTINE
23000 PRINT CHR$(147):PRINT"WAIT":PRINT
```

```
23010 X=FRE(0)-(SGN(FRE(0))<0)*65535
23020 PRINT"NUMBER OF INDEX BYTES FREE";
X
23030 GOSUB15000
23040 RETURN
23997 REM *
23998 REM **
23999 REM CLEAR OUT BUFFER TO DISC
24000 CLOSE6
24010 OPEN6,8,6,"0:"+N$
24020 RETURN
```

Program 6.2. Autosorted indexed filing system.

Using a binary search

Since a fast sorting routine is on hand it is far less tedious to keep the index in an ordered sequence. In fact the index is automatically sorted each time a record is written to disk. This is why the sort index option is removed from the secondary menu. The worst case execution time to sort the index, even with a thousand records, is one second since only one record needs sorting into the index each time. This enables us to consult the index in a more efficient manner. Two binary search algorithms are employed in Program 6.2 – one connected with index look-up on record retrieval and the other in the 'check key (unique/tombstone)' subroutine. The use of these modifications improves execution speed not only in record retrieval but also in checking the index for deleted records. Any 'space' tombstone characters are sorted to the end of the index each time. Thus, a quick check of the last index entry is sufficient to test for the presence of a tombstone. A tombstone, if you remember, denotes a deleted record and that the file location (and the key field index entry) is available for reuse. These 'holes' in the file take priority over unused record slots when a new record is stored.

An alternative method of loading the machine code sort routine
If required, it is possible to place the machine code bytes generated by Program 5.10 into a serial file via a series of PEEKs. This file could then be loaded automatically under main program control via a series of POKEs. If you are not sure how to do this, see Program 6.3 which shows how to form the serial file under the file name 'SORT', and Program 6.4 which is the subroutine necessary to load back the code

from disk. It must be stressed that Program 5.10 must be RUN before the above file can be created and that the file must be present on disk when RUNning the modified version of Program 6.2. A simple GOSUB25000 at, say, line 5 will autoload the code, albeit slowly.

```
10 REM SAVING MACHINE CODE SORT ROUTINE
20 REM AS A SERIAL FILE
30 OPEN7,8,7,"0:SORT,S,W"
40 S=49152
50 FOR N=0 TO 270
60 PRINT#7,PEEK(S+N)
70 NEXT
80 CLOSE7
90 END
```

Program 6.3 Saving machine code in a serial file.

```
24997 REM *
24998 REM **
24999 REM LOAD M/C CODE SORT ROUTINE
25000 OPEN7,8,7,"0:SORT,S,R"
25010 S=49152
25020 FOR N=0 TO 270
25030 INPUT#7,BT
25040 POKE S+N,BT
25050 NEXT
25060 CLOSE7
25070 RETURN
```

Program 6.4. Loading machine code from a serial file.

Summary

1. A serial file is not stored in any particular order.
2. A primary key or key field is one which identifies a record.
3. A secondary field, other than the key field, which tends to classify the record.
4. A sequential file is stored in primary key order.
5. Sequential files can be modified and records deleted but the method is tedious and involves the use of three files: the master, transaction and update files.
6. The update file eventually becomes the new master file.

7. An indexed relative filing system uses two files: the main file in store and a smaller index file, normally in RAM.
8. The main file need not be in any order, because the records are addressed by the index file which can be ordered.
9. The index file need only contain the key field and the record number.

Self test

6.1 The four fields in a record are Length, Alkalinity, River and Source. Which would you choose as the primary key?
6.2 Current entries first go into the update file. True or false?
6.3 If a disk buffer is only partially filled at the end of a session, how do you ensure that the contents reach the disk?
6.4 What must you be careful of when using the machine code sort routine Program 5.10 with any other program?
6.5 Which character is used as a tombstone marker?
6.6 Why should the programs in this chapter be terminated only via the 'END PROGRAM' option?

Chapter Seven

Hash Coding and Relative Files

An easy method for accessing a record from within a relative file is to quote the record number. However, this is not very convenient for the program user because it would mean keeping a separate list, linking each record with its number. In Chapter 6 we saw one method of directly accessing a record by first consulting an *index* of key fields and record numbers. In this chapter we will look at an entirely different method of achieving the same goal using *hash coding* techniques.

The key field and the record number problem

Key fields can be expressed as a combination of letters, combination of numbers or a mixture of both. The most user-friendly is probably the all-letter form. For example, the name of a person, the name of a species or the name of a town or street is straightforward and does not involve remembering, or looking up, a key in code form. Unfortunately, the nature of direct access files is alien to the concept of retrieval by normal key field. Records in relative files are accessed by a *unique record number*. This means that space in store must be available for as many records as there are possible combinations of the key field characters – even if most of them will never be used! This is a formidable restriction. To see why, let us consider a key field consisting of all letters, such as the surname or clients. We will, for simplicity, restrict the number of letters to 10. There are 26 letters in the alphabet so there are 26^{10} different ways of arranging the letters, which is approximately 10^{14}. This is a truly enormous figure. Obviously, many of the possible combinations would not occur in practice. For example, no-one's name would ever be Zzzzzzzzzz and it is doubtful if a Mr Zbhynskfbo or a Miss Xlypopubum are likely to roam the earth very often. However, even if we are quite ruthless and

say that out of the 10^{14} theoretical combinations, there are only 10^6 practical ones, this is still a large number – 1 million, in fact. And yet, a large file in a microcomputing system is unlikely to contain more than 10000 records while the average small filing system would seldom exceed 1000 records. We could say, therefore, that the list of possible combinations is only 'sparsely populated' with real ones. This leads us to the conclusion that accessing a record, by means of its key field in the form of a name, may certainly be user-friendly but the storage space required, most of it wasted anyway, would stretch the capacity of even a mainframe giant. The situation would be almost as bad if the key field were a string of numerical digits – say, an engine part number or an employee's Social Security number. Even a six-digit number can be arranged in a million ways besides the wastage incurred by discontinuities.

In summary, we can say that it is rarely possible in practice to achieve a one to one functional relationship of a key to a record number. Each key must, therefore, be transformed (mapped) into one of the available range of record numbers allocated to the file.

Key to address transformations

There are two main categories of key to address transformations – *deterministic* procedures and *randomising* techniques.

Deterministic procedures

A deterministic procedure is an addressing algorithm based on the prior knowledge of a set of key values. It is used to compute a unique file address or record number. These algorithms become extremely difficult to construct if records exceed a hundred or so. Because the algorithm is dependent on the spread of keys, the addition of a new record could require the entire algorithm to be rewritten. It follows that deterministic procedures are rarely used other than for small static files. A transformation table could be used in difficult cases, but this is similar to an indexed file!

Randomising techniques

There are two classes of randomising algorithms. The first class is called *sequence maintaining*, where randomisation algorithms are designed, as far as possible, to preserve the original order of the keys. The second class, termed *hashing techniques*, attempts to capitalise

on the uniqueness of the keys. We will deal exclusively with the more popular class of hashing techniques.

Hash functions

Hashing is a brilliant solution to the conflicting interests of storage space and key field range. It is possible to retain the simplicity of access by key field without seriously jeopardising storage capacity.

In the following summary of the technique, we shall assume that the key field is a typical multi-digit number:

(a) First, a conservative estimate is made of the number of records there will be in the file.
(b) The key field digits of each record are then used as input to a suitable 'hash' function.
(c) The output from the hash function is a *transformation* of the key field and specifies a record number within the range of the file.

For example, suppose we estimated there would be 20 records in the file and the original key field was a 6-digit number. The record number, calculated by the hash function, would be within the range 1 to 20. The enormous number of possible key field combinations is thus scaled down by the hash function to just the number required for 20 records. We now have the best of both worlds. The normal key field can be used to locate a record without wasting large areas of memory, most of which might never be used.

Is this too good to be true? Well, it is almost true but, as might be expected, there are one or two snags. Whatever hash function is chosen, and there are many, there is no 100% guarantee that a calculated record number will be unique. There is a probability that, occasionally, two records will bear the same calculated record number. When this happens, the two records are said to *collide*. On the face of it, collisions would appear to have serious, even catastrophic, effects on the behaviour patterns of record access. Fortunately for the reputation of hash coding, careful programming can allow for collisions quite easily. However, before treating collisions in detail, we must examine some actual hash functions.

Common hash functions

A hash function takes a number and mixes it up in a disorganised

fashion. In fact the more the number is mixed up (randomised), the better. In short, it makes a hash of the number, hence its name. The random number generator in a computer works in a similar manner. Apart from mixing up the number, the primary aim is to reduce a multi-digit number to one containing fewer digits. Another requirement is to ensure, as far as possible, that the resultant number is different each time the hash function operates. Some possible methods of generating hash functions now follow:

The mid square method
The original number N is squared and a few digits in the middle are taken as the hashed number.

Examples:
(a) If N=7435, then N^2=55279225 so a two-digit hash-coded number is 79.
(b) If N=8031, then N^2 is 64496961, and the two middle digits are now 96.

The division remainder method
This function is based on modulo arithmetic. The original number N (the key field) is divided by another number (usually the number of records estimated to be in the file) and the remainder taken as the hash result. In BASIC notation this becomes N MOD R.

Example:
Suppose N=3875 and the number of records, R, is 50. Dividing 3875 by 50 yields the whole number 77 leaving a remainder of 25 which is the hash code. There are no keywords in Commodore 64 BASIC for direct manipulation of modulo arithmetic but they can easily be programmed. A better spread of numbers usually results if a *prime number* is used as the divisor. The nearest prime to 50 is 49 so this should be chosen as the divisor. The division remainder method is used in the filing program which follows later in this chapter.

The folding and adding method
Consider a string of numbers, say, 346758, written down on a slip of paper and then folded into an 'S' shape. Taking the fold into consideration, the numbers are then read as three separate pairs and added, as shown in Fig. 7.1 to produce a two-digit hash code. Ignore any final carry.

If you find this confusing, the following simplified description may help:

(a) Split the key field into pairs of digits.
(b) Reverse the order of every other pair of digits.
(c) Add up the pairs but ignore the final carry digit.

Using these rules, the key 346758 becomes 34+76+58=68 (the final carry of 1 is dropped altogether or added to the hash total).

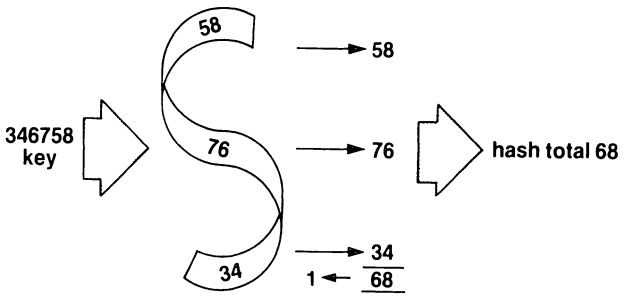


Fig. 7.1. Folded numbers.

Many other hashing functions are described in the literature but it is pointless describing them all. The only requirements of a hash function is that it should produce well-spread random numbers (so as to avoid too many collisions) within a specified range and should execute rapidly. In fact, the particular hash function used is not all that important. Hashing is based on randomising which, after all, is low grade 'energy' and often appears quite unintentionally. As long as the number is well-stirred, like a good curry, almost any hash function will do. A point worth mentioning is that addition and multiplication tend to produce a normal distribution of random values which should be avoided.

Hashing alpha numeric characters

If alphabetical characters are used as key fields in a file, the same techniques apply. All that is necessary is first to convert the string to, as far as possible, a unique number. Just adding together the ASCII codes does not provide a sufficient degree of uniqueness. For example, the number generated for the string ABC will be identical to that of CBA or, in fact, any other combination of these letters. One of many solutions to this problem is to start at one end of the string, cumulatively adding the product of adjacent pairs of ASCII codes. This destroys most of the above combinational relationships. It does not matter about the occasional clash, since this will be handled by

techniques known as *collision resolution* which we shall explain later. The number generated can then be hashed within the available address range of the file by any of the previously described hash functions. We have used, with satisfactory results, the above technique, in conjunction with the division remainder method.

Handling collisions by open addressing

Whatever hash coding techniques are used, some collisions will undoubtedly occur when storing records. The probability of collision is low at first because there are plenty of unoccupied locations, but as the file begins to fill up with records, the probability increases. There are several programming techniques for handling a collision.

If collisions are resolved and stored within the main body of the file itself, then *open addressing* methods are said to be used. Alternative methods make use of separate overflow areas (or files) in order to store colliding records. There are a number of ways to resolve collisions using open addressing such as *linear probing* or *rehashing*.

Linear probing

This method tests the location or record slot to which the record is directed, to see if the location is already occupied. If it is then the next location to it is tried. If this is occupied, the next is tried, and so on, until an unoccupied location is found. Trials all take place in the same direction. If, by chance, the last location in the file has been reached without success, the first one in the file is tried by *wrap around*. On returning to the original transformed location it will be evident that the file is full. In practice, this situation is unlikely because over-allocation of record space by about 40% should be used in direct files to avoid frequent collisions.

If a record is to be retrieved, the search key is compared with the key of its transformed record number. If the record slot is not occupied by the desired key then a sequential search is made in the same direction (wrap around) until the record is either found or deemed not to exist. The latter decision can sometimes be made as soon as a blank record slot is found. However, later deletion of records may trip up the system. To avoid this, the whole file can either be searched sequentially in a wrap-around manner, or *tombstone* markers can be placed into the keys to signify deleted records. The

tombstone would signify that the record slot is free to accept an insertion but can be ignored during the linear probe on retrieval.

Clustering

One of the major disadvantages of linear probing is that when the file becomes 50% to 60% full, the increased collision rate leads to *clustering*. Clustering is a technical term used to describe the tendency of many records to appear in adjacent record slots. This can lead to longer and longer sequential searches being necessary to cope with collisions as the file is filled. This effect can be loosely equated with instability. The system begins to collapse at the onset of instability, thus over-allocation of record space is essential when using linear probing.

Rehashing

A solution to the problem of collision resolution, while at the same reducing the tendency to cluster, is *rehashing*. A second hash function is used to find an alternative record slot for the colliding record. If this location is also filled it will be necessary either to resort to linear probing or rehash again.

A practical direct access filing system using hash coding

The listing appears as Program 7.1.

```
10 REM DIRECT FILING SYSTEM
20 REM EMPLOYING HASHING TECHNIQUES
30 OPEN15,8,15
40 GOSUB1000:PRINT#15,"I":GOSUB5000:PRIN
T:PRINT
50 PRINT"(1) CREATE NEW FILE"
60 PRINT"(2) ACCESS EXISTING FILE"
70 PRINT"(3) EXIT PROGRAM"
80 PRINT:PRINT
90 INPUT"SELECT OPTION";S%
100 IF S%<1 OR S%>3 THEN 40
110 ON S% GOSUB2000,3000,250
120 PRINT CHR$(147)
130 OPEN6,8,6,"0:"+N$:GOSUB5000
140 GOSUB1000:PRINT:PRINT
```

```
150 PRINT "(1) ENTER A RECORD"
160 PRINT "(2) DISPLAY A RECORD"
170 PRINT "(3) DELETE A RECORD"
180 PRINT "(4) MODIFY A FIELD"
190 PRINT "(5) EXIT PROGRAM : CLOSE FILE
S
200 PRINT:PRINT
210 INPUT "SELECT OPTION";S%
220 IF S%<1 OR S%>5 THEN 140
230 ON S% GOSUB 6000,7000,8000,9000,250
240 GOTO140
250 CLOSE6:CLOSE15:END
997 REM *
998 REM **
999 REM TITLE SUBROUTINE
1000 PRINT CHR$(147)
1010 PRINT"DIRECT ACCESS FILING SYSTEM"
1020 RETURN
1997 REM *
1998 REM **
1999 REM CREATE FILE SUBROUTINE
2000 PRINT CHR$(147):GOSUB4000
2010 PRINT"ENTER FILE SIZE (NUMBER OF RE
CORDS)"
2020 INPUT FS%
2030 IF FS%<1 THEN 2010
2040 PRINT"ENTER NUMBER OF FIELDS REQUIR
ED"
2050 INPUT NF%
2060 IF NF%<2 OR NF%>9 THEN PRINT"FIELDS
(2-9) ONLY":GOTO2040
2070 P(1)=1
2080 FOR F=1 TO NF%
2090 PRINT CHR$(147)
2100 PRINT"ENTER FIELD HEADING";F
2110 W=255:GOSUB22000:H$(F)=K$
2120 PRINT"ENTER FIELD WIDTH (CHARACTERS
)"
2130 INPUT W(F)
2140 IF W(F)<1 THEN 2120
2150 P(F+1)=P(F)+W(F)+2
2160 NEXT
2170 IF P(NF%+1) > 254 THEN PRINT"TOO LO
NG : START AGAIN":GOSUB15000:GOTO2070
2180 PRINT CHR$(147):PRINT"WAIT":P%=FS%:
```

```
GOSUB18000
2190 PRINT"CREATING HEADING FILE H.";N$
2200 OPEN 5,8,5,"0:H."+N$+",S,W"
2210 GOSUB5000
2220 PRINT#5,NF%:PRINT#5,FS%:PRINT#5,P%
2230 FOR F=1 TO NF%
2240 PRINT#5,H$(F):PRINT#5,W(F):PRINT#5,
P(F)
2250 NEXT
2260 CLOSE5
2270 PRINT"CREATING MAIN FILE ";N$
2280 OPEN 6,8,6,"0:"+N$+",L,"+CHR$(P(NF%
+1))
2290 GOSUB5000
2300 R%=FS%:F=1
2310 GOSUB12000:GOSUB13000
2320 PRINT#6,CHR$(255)
2330 CLOSE6
2340 RETURN
2997 REM *
2998 REM **
2999 REM LOAD HEADING FILE SUBROUTINE
3000 PRINT CHR$(147):GOSUB4000
3010 OPEN 5,8,5,"0:H."+N$+",S,R"
3020 GOSUB5000
3030 INPUT#5,NF%,FS%,P%
3040 FOR F=1 TO NF%
3050 INPUT#5,H$(F),W(F),P(F)
3060 NEXT
3070 CLOSE5
3080 RETURN
3997 REM *
3998 REM **
3999 REM GET FILE NAME SUBROUTINE
4000 PRINT"ENTER FILE NAME"
4010 W=14:GOSUB22000:N$=K$
4020 RETURN
4997 REM *
4998 REM **
4999 REM READ ERROR CHANNEL SUBROUTINE
5000 INPUT#15,A,B$
5010 IF A<20 OR A=50 THEN 5030
5020 PRINT CHR$(147):PRINT B$:GOSUB15000
:RUN
5030 RETURN
```

```
5997 REM *
5998 REM **
5999 REM ENTER A RECORD SUBROUTINE
6000 PRINT CHR$(147)
6010 PRINT"ENTER A RECORD":PRINT:PRINT
6020 FOR F=1 TO NF%
6030 PRINT"MAXIMUM CHARACTERS";W(F)
6040 PRINT "ENTER ";H$(F)
6050 W=W(F):GOSUB22000:F$(F)=K$
6060 NEXT
6070 PRINT CHR$(147)
6080 PRINT"STORING RECORD ON DISC"
6090 GOSUB19000
6100 RETURN
6997 REM *
6998 REM **
6999 REM RETRIEVE RECORD SUBROUTINE
7000 PRINT CHR$(147):PRINT"ENTER KEY FIE
LD"
7010 FL%=0
7020 W=255:GOSUB22000:KY$=K$
7030 PRINT CHR$(147):PRINT"RETRIEVING RE
CORD"
7040 H$=KY$:GOSUB17000:R%=H%
7050 GOSUB12000:F=1:GOSUB13000
7060 INPUT#6,F$
7070 IF F$=KY$ THEN GOSUB10000:GOTO7130
7080 R%=R%+1
7090 IF R%>FS% THEN R%=1
7100 IF R%=H% OR F$=CHR$(255) THEN 7120
7110 GOTO7050
7120 PRINT"RECORD NOT ON FILE":FL%=1:GOS
UB15000
7130 RETURN
7997 REM *
7998 REM **
7999 REM DELETE RECORD SUBROUTINE
8000 GOSUB7000:PRINT
8010 IF FL%=1 THEN 8110
8020 PRINT"DO YOU WISH TO DELETE THIS RE
CORD (Y/N)"
8030 INPUT K$
8040 IF K$="N" THEN 8110
8050 IF K$="Y" THEN 8070
8060 GOTO8020
```

```
8070 FOR F=1 TO NF%
8080 F$(F)=CHR$(97)
8090 NEXT
8100 GOSUB12000:GOSUB11000
8110 RETURN
8997 REM *
8998 REM **
8999 REM MODIFY RECORD SUBROUTINE
9000 GOSUB7000:IF FL%=1 THEN 9110
9010 PRINT:K=0
9020 PRINT"ENTER HEADING OF FIELD TO BE
MODIFIED"
9030 W=255:GOSUB22000:M$=K$
9040 FOR F=1 TO NF%
9050 IF M$=H$(F) THEN PRINT"ENTER NEW CO
NTENTS":W=W(F):GOSUB22000:F$(F)=K$:K=F
9060 NEXT
9070 IF K=0 THEN PRINT"NO SUCH FIELD":GO
SUB15000:GOTO9110
9080 IF K=1 THEN GOSUB16000:GOTO9110
9090 GOSUB12000
9100 GOSUB11000
9110 RETURN
9997 REM *
9998 REM **
9999 REM READ RECORD SUBROUTINE
10000 FOR F=1 TO NF%
10010 GOSUB13000
10020 INPUT#6,F$(F)
10030 NEXT
10040 GOSUB14000
10050 RETURN
10997 REM *
10998 REM **
10999 REM WRITE RECORD SUBROUTINE
11000 GOSUB21000
11010 FOR F=1 TO NF%
11020 GOSUB13000
11030 PRINT#6,F$(F)
11040 NEXT
11050 GOSUB21000
11060 RETURN
11997 REM *
11998 REM **
11999 REM CALCULATE FILE POINTER
```

```
12000 HB%=R%/256
12010 LB%=R%-HB%*256
12020 RETURN
12997 REM *
12998 REM **
12999 REM SET FILE POINTER
13000 PRINT#15,"P"CHR$(6)CHR$(LB%)CHR$(H
B%)CHR$(P(F))
13010 RETURN
13997 REM *
13998 REM **
13999 REM DISPLAY RECORD SUBROUTINE
14000 PRINT CHR$(147)
14010 PRINT"DISPLAY OF RECORD NUMBER";R%
14020 PRINT:PRINT
14030 FOR F=1 TO NF%
14040 PRINT H$(F),F$(F)
14050 NEXT
14060 IF S%=2 THEN GOSUB15000
14070 RETURN
14997 REM *
14998 REM **
14999 REM WAIT FOR ANY KEY PRESSED
15000 PRINT:PRINT"PRESS ANY KEY TO CONTI
NUE"
15010 GET K$
15020 IF K$="" THEN 15010
15030 RETURN
15997 REM *
15998 REM **
15999 REM MOVE RECORD SUBROUTINE
16000 M%=R%:GOSUB19000
16010 IF T%=0 THEN R%=M%:GOSUB8070
16020 RETURN
16997 REM *
16998 REM **
16999 REM HASH FUNCTION SUBROUTINE
17000 H=0
17010 K=ASC(LEFT$(H$,1))^2
17020 FOR N=1 TO LEN(H$)
17030 H=H+ASC(MID$(H$,N,1))*K
17040 NEXT
17050 K=INT(H/P%):H%=H-K*P%
17060 IF H%=0 THEN H%=1
17070 RETURN
```

```
17997 REM *
17998 REM **
17999 REM CALCULATE HIGHEST PRIME
18000 IF P%<=3 THEN 18070
18010 P%=P%-1
18020 D%=1
18030 D%=D%+1
18040 K=P%/D%
18050 IF K>INT(K) THEN 18030
18060 IF D%<=K THEN 18010
18070 RETURN
18997 REM *
18998 REM **
18999 REM STORE RECORD SUBROUTINE
19000 H$=F$(1):GOSUB17000:R%=H%
19010 GOSUB20000:IF T%=0 THEN GOSUB11000
:GOTO19100
19020 IF T%=-1 THEN PRINT"REJECTED : KEY
 FIELD NOT UNIQUE":GOSUB15000:GOTO19100
19030 R%=R%+1
19040 IF R%>FS% THEN R%=1
19080 IF R%=H% THEN PRINT"REJECTED : FIL
E FULL":GOSUB15000:GOTO19100
19090 GOTO19010
19100 RETURN
19997 REM *
19998 REM **
19999 REM CHECK KEYFIELD UNIQUE
20000 GOSUB12000:F=1:GOSUB13000
20010 INPUT#6,F$
20015 PRINTR%,F$,F$(F)
20020 T%=1
20030 IF F$=CHR$(255) OR F$=CHR$(97) THE
N T%=0
20040 IF F$=F$(1) THEN T%=-1
20050 RETURN
20997 REM *
20998 REM **
20999 REM CLEAR OUT BUFFER TO DISC
21000 CLOSE6
21010 OPEN6,8,6,"O:"+N$
21020 RETURN
21997 REM *
21998 REM **
21999 REM INPUT VALIDATION SUBROUTINE
```

```
22000 K$="":INPUT K$
22010 IF K$="" OR K$=CHR$(255) THEN PRIN
T"INPUT NOT VALID":GOTO22000
22020 IF LEN(K$)>W THEN K$=LEFT$(K$,W)
22030 RETURN
```

Program 7.1. Direct access filing system using hashing techniques.

We should emphasise once more the primary advantage of a direct filing system which employs hashing. Record numbers need not be memorised because normal key fields, numeric or alphanumeric, can be used to access any record directly. However, it is worth pointing out once more that such an advantage is gained only by sacrificing extra disk storage space. You will remember that about 50% extra space should be reserved to allow for collision resolution. So if, for example, you want 100 records, ask for 150 because the frequency of collisions increases as the number of free record slots decrease. The programming of most of the subroutines in Program 7.1 should, by now, be fairly easy to understand. Those which employ techniques specific to direct file handling and hash coding will be described at the end of this chapter. No particular advice is necessary on how to use the program because the options are straightforward and similar to those explained in previous programs.

Variables used in Program 7.1

FS% = maximum file size (maximum number of records).
W(F) = width of field F.
W$(F) = is actual field of RAM.
P(F) = pointer to start of field.
NF% = number of fields.
B$ = disk error message.
N$ = file name.
S% = option selected.
H$(F) = heading of field F.
R% = record number.
A = error number.
P% = calculated prime.
H% = hash number.
HB% = high byte address.
LB% = low byte address.
K$ = general purpose variable.
KY$ = key field.
T% = flag used to indicate record slot status.

The subroutines used are shown in Table 7.1.

Table 7.1. Subroutines used in Program 7.1.

1st level	2nd and 3rd level
Create file	Get file name
	Input validation
	Wait for any key
	Calculate highest prime
	Read error channel
Access existing file	Load heading file
Enter a record	Input validation
	Store record
	Hash function
	Check key field unique
	Wait for any key
	Clear buffer to disk
	Write record
	Set file pointer
Display a record	Retrieve record
	Input validation
	Hash function
	Calculate file pointer
	Set file pointer
	Read record
	Wait for any key
Delete record	Retrieve record
	Input validation
	Hash function
	Calculate file pointer
	Set file pointer
	Read record
	Wait for any key
	Calculate file pointer
	Set file pointer
Modify a field	Retrieve record
	Input validation
	Hash function
	Calculate file pointer
	Set file pointer
	Input validation
	Wait for any key
	Move record
	Calculate file pointer
	Write record
	Set file pointer

Creating a new file

Most of the subroutines in Program 7.1 are straightforward. However, the subroutines for storing and retrieving direct files using hash coding involve fresh principles and so justify detailed discussion. We shall first concentrate on the major steps needed to create a new file. Before the actual file data can be entered, a short header file is produced, containing the essential parameters required by the main file. This is then stored as a small *serial file* on disk. The steps taken to produce the header and main file are out lined below:

(1) Obtain the following parameters from the user:
Maximum number of records, maximum number of fields in each record, field headings, field width and file name. These are assigned to FS%, NF%, H$(F), F(W) and N$ respectively.
(2) From this information, pointers to the start of each field are produced and assigned to an array P(F). Also, a prime number, P%, is calculated which is the highest prime lower than the maximum number of records FS%. The prime is a component for generating the hash code in the main file.
(3) The small serial *heading file* is then OPENed to channel 5 (see line 2200) and stored on disk. It contains NF%, FS%, P% and the three arrays H$(F), W(F) and P(F). These arrays are small enough to escape the need for DIMensioning.
(4) The main, relative file skeleton is then created by the DOS after first OPENing to channel 6 (line 2280). All the empty record slots are then laid down on disk for each record. By creating the last record first, the DOS is forced to create all empty records preceding it. The first byte of each empty record slot is automatically set by the DOS to $FF. This is handy to use as an empty record marker for linear probing. To test if a record is empty it is sufficient to test the *first byte* of a record slot for a CHR$(255), which is the string interpretation of $FF or 255 decimal.

Summarising, creating a filing system entails the production of two disk files, a small header file and the skeleton of the main file. The main file will not yet contain data, other than the CHR$(255) marker, indicating an empty record. It takes rather a long time for the Commodore 1541 DOS to create the file – a few minutes, in fact, for 1000 records, but it is a one-off job. Once the file has been created, storage and retrieval of any record, irrespective of its position within the file, is almost instantaneous – a second or so at most.

Enter a record

After the header and main file have been created on disk, the record data can be entered and stored in the main file. The steps are as follows:

(1) The record data for each field is obtained from the keyboard.
(2) The key field is then assigned to H$ and passed to the subroutine which generates the hash code for R%. At this stage, R% is only the provisional record number because it may cause a collision.
(3) R% is then checked by the 'Check key field unique' subroutine. The purpose of the subroutine is to check whether

(a) that record slot is already occupied;
(b) the key field entered by the operator is unique.

It does this by first setting a flag, T%, to 1. It then calculates and sets the file pointer to the record whose number is R%. The key field of the record is then read into F$. This will either be a CHR$(255) indicating that the record slot is empty or a 'spade', indicating that the slot had once been occupied but has been subsequently deleted (see 'Delete record' below). The spade, CHR$(97), acts as a tombstone marker. The presence of either of these two characters is taken as an indication that the original hashed record number was acceptable for storage and the flag T% is set to 0. If the key field of the entered record has been used before, the flag is set to −1. If some other character is present in F$, it indicates that the slot is already inhabited by another record and the flag is left at 1.
(4) The 'Write record' subroutine is called and the flag, T%, is examined. If this is 0, the record is written to disk. However, there is a snag. When using linear probing in conjunction with relative files, it appears necessary to clear out the current buffer to disk both *before and after writing each record.* This can be achieved by CLOSEing and reOPENing the file. Neglecting to do this can sometimes lead to puzzling results when storage, and subsequent retrieval, takes place across sectors of the disk.

If the flag T% is set to −1, the screen displays the 'Key field not unique' message and the menu is regained.

If the flag T% is set to 1, it indicates that a collision has occurred, so R% is increased by one continuously until an empty slot is found. That is to say, it uses the linear probing technique discussed earlier.

Display record

The subroutine 'Retrieve record' is called and the key field of the required record entered into KY$. The subroutine 'Hash function' then converts KY$ to the hashed record number R%. From R%, the appropriate file pointer is produced and the key field of the stored record is read from disk and compared with that entered at the keyboard. If the two match then the record is displayed. If not, the record number is continually incremented (wrap around) until either a match is found or an empty record slot is detected by the presence of a CHR$(255) marker. Deleted records, denoted by a tombstone marker, are regarded as valid records for retrieval purposes, otherwise the linear probing system will not work properly. Suitable messages are displayed if the record is not on file. This is output as soon as either a CHR$(255) is detected as the first character of the key field or a sequential search has gone full circle. The latter is necessary to cover the case where the user insists on using up every available record location.

Delete record

The required record is retrieved by linear probing, described above, and displayed. Providing the user answers 'Y' to the 'belt and braces' prompt, the file pointer is calculated and set in the usual manner. The existing key field in the record is then replaced by the 'spade' CHR$(97) character which acts as a tombstone marker.

Modify record

The required record is first retrieved in the manner previously described. A variable K is initialised to 0 and the field number to be modified is assigned to K. If K=0, the message 'No such field' is displayed. Providing K is not 1 (the key field), the amended record is written back to memory at the old record number. If, however, the key field was modified, it means that a new hash code must be found which maps from the new key field characters. The subroutine 'Move record' is then called which, in turn, calls on the 'Store record' subroutine. We can see from this that a record with a modified key field is treated as a new record. A tombstone marker is written into the old record slot only if the move was successful.

Exit program

This option should always be used to terminate the program otherwise the file will be invalid and/or corrupted. This option

ensures that all files are closed, both in BASIC and by the DOS. It also clears out the current buffer to disk.

Incidentally, line 20015 may be deleted once the program has been entered and tested. It is included to demonstrate the linear probing technique when storing records.

Summary

1. Each record slot in a relative file has a unique record number.
2. There are a large number of possible combinations in a typical key field. It is not practical to reserve disk slots for all the possible record numbers because so many would not be used.
3. To retain the advantages of accessing a record by key field, a range of record numbers must be produced which are based on a conservative estimate of the file size.
4. As each record is entered, the key field is mapped to one of the record numbers by some method.
5. The most common method of converting a normal key field to a disk address involves a step known as hash coding, or simply 'hashing'.
6. When a key field is hashed, a random number is produced within the previously estimated record range.
7. A hashed number is not necessarily unique. If two different key fields are hashed to the same record number, they are said to collide.
8. Collisions must be resolved, otherwise two records, although having different key fields, could not be independently accessed or stored.
9. Since most key fields consist of, or contain, alpha characters, they must be converted to a pure numerical equivalent before being offered up to the hashing function.
10. The chance of collisions increases as the file population grows.
11. Collision resolution by linear probing is the most popular. If a slot is already occupied, the next is tried and so on, until an empty slot is found.
12. If the file is nearly full, the number of collisions increases and linear probing can lead to clustering. This means that many records will begin to occupy adjacent slots.
13. When the file is initially created, the number of records should be over-estimated by, say, 50% so that there will always be plenty of spare locations to minimise clustering.

14. When using Program 7.1 to create a new file, be prepared for a few minutes' wait while the skeleton of the main file is being laid down on disk.
15. Program 7.1 employs a small serial file for storing the heading information in addition to the main file.
16. A CHR$(255) is used to denote an empty record slot and a CHR$(97) is a tombstone marker indicating a previously deleted record.
17. It is essential to clear out the current buffer to disk before storing a record when using linear probing.
18. When you have finished using Program 7.1, you must use the Exit option. If you leave by the STOP/RESTORE keys or by simply switching off the machine, you will risk corrupting or losing your file.

Self test

7.1 A key field of four characters uses only upper-case letters. How many possible ways are there of arranging all the letters?
7.2 Using the division remainder method, what is the resultant hash code if the original number was 4534 and there are 60 records?
7.3 How can we reduce the possibility of clustering?
7.4 What is meant by linear probing?
7.5 What does a tombstone marker signify?
7.6 Why does the calculated prime number have to be stored on the header file?
7.7 Why must you leave Program 7.1 by the Exit option?

Appendix A
Glossary of Terms

binary search: a fast searching method, requiring a sorted list.
bubble sort: a simple sorting technique.
clustering: a tendency of records to bunch together when using linear probing techniques.
collision: occurs when two keys are mapped to the same address.
command-driven: a program in which an option is selected via a direct command.
contiguous: occupying adjacent positions in a list.
data file: file which contains data accessible only by a resident program.
deterministic procedures: addressing algorithms designed with the prior knowledge of a set of key fields.
direct file: file in which any record, irrespective of its position in the file, can be immediately located.
disk buffer: an area of memory which is used as a bucket for temporary storage.
disk controller: circuitry which provides the hardware interface between computer and disk drive.
disk formatting: laying down track and sector addresses onto a blank disk.
documentation: supporting information which helps in using or modifying a program.
double-sided: a diskette with both surfaces guaranteed.
drive 0: the default drive on a double disk unit.
exchange sort: a sort based on comparison of adjacent items in a list.
field width: number of characters allotted to a field.
file: collection of related information.
file creation: entering preliminary information necessary for starting up a new file.
file reorganisation: a method, or program, which entirely modifies the structure of an existing file.

flag: a data item which can mean either yes or no.
format file: see *heading file.*
hard sectoring: disks which arrive with prerecorded format.
hash function: a function used to calculate the actual address when using hashing techniques.
hashing: a key to an address transformation method involving randomisation techniques.
heading file: a serial file containing any necessary information required to access the main file to which it relates.
indexed file: a file whose records are accessed by first consulting a separate index file. The records themselves need not be in any predefined order.
key: a field which uniquely identifies a record.
master file: the main sequential file periodically updated by a transaction file.
object code: the machine code translation of source code.
open addressing: a term used to describe the act of resolving collisions within the main body of the file.
pointer: a variable representing the address of another variable.
prime: a number which has no divisors other than 1 and itself.
program-based: describes a primitive filing system in which records are kept within the program as DATA statements.
RAM: volatile internal read/write memory.
randomising technique: a key to address transformation which utilises some form of randomising of the key fields.
rehashing: a method of resolving collisions with the use of a second hash function.
relative file: a form of random access file, peculiar to Commodore.
ROM: non-volatile read only memory.
secondary key: a field which tends to classify a record rather than identify it.
sector: subdivision of a disk track.
sector density: number of sectors per track.
sequence maintaining: a randomisation algorithm which tends to preserve the original order of the key fields.
sequential file: one in which records are stored in key field sequence.
sequential search: searching from the first record and continuing sequentially until the required record is found.
serial file: one in which the records follow one another in no particular order.
soft sectoring: disks which must be formatted by the user.

source code: a program written in assembly language or a higher level language.

subfile: part of a file containing records which satisfy search criteria.

substring: selected characters within a string.

swop flag: indicates whether a swop has occurred during a sorting scan.

tombstone: a marker used to denote a deleted record.

tpi: number of disk tracks per inch.

transaction file: a file containing current information, destined for merging into the master file.

update file: a file which combines a transaction file with the existing master file. It subsequently becomes the new master file.

volatile: describes memory which loses stored data if power supply is interrupted.

Appendix B

ASCII Character Codes

Decimal	Hex	Character	Decimal	Hex	Character	Decimal	Hex	Character
32	20	Space	64	40	@	96	60	£
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	−	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	Delete

Appendix C

List of 6502 Assembly Mnemonics

To avoid complication, no distinction is made between 'page zero' and absolute addressing.

Mnemonic Code	Action	Addressing details
TAX	Copy contents of A to X	All op-codes in this section use *implicit* addressing. They have no operands.
TXA	Copy contents of X to A	
TAY	Copy contents of A to Y	
TYA	Copy contents of Y to A	
INX	Add 1 to X	
INY	Add 1 to Y	
DEX	Subtract 1 from X	
DEY	Subtract 1 from Y	
TSX	Copy contents of SP to X	
TXS	Copy contents of X to SP	
PHA	Push A onto stack	
PLA	Pull A from stack	
PHP	Push PSR onto stack	
PLP	Pull PSR from stack	
CLC	Clear C bit in PSR	
SEC	Set C bit in PSR	
CLV	Clear V bit in PSR	
SED	Set D bit in PSR	
CLD	Clear D bit in PSR	
SEI	Set I bit in PSR	
CLI	Clear I bit in PSR	
NOP	No operation at all	
RTS	Return from subroutine	
RTI	Return from interrupt	
BRK	Break (stop)	

Mnemonic Code	Action	Addressing details
BNE	Branch if not equal	All op-codes in this section use *relative* addressing. The operand can be an arbitrary chosen label. Branching depends on the result of the *previous* instruction.
BEQ	Branch if equal	
BPL	Branch is positive (plus)	
BMI	Branch if negative (minus)	
BCC	Branch if C=0	
BCS	Branch if C=1	
BVC	Branch if V=0	
BVS	Branch if V=1	
LDA	Load A	All op-codes in this section use either immediate, absolute or indexed addressing. Either the X or Y register can be used for indexing. All results are in A.
ADC	Add with carry	
SBC	Subtract with carry	
CMP	Compare	
AND	Perform logical AND	
ORA	Perform logical OR	
EOR	Perform logical Exclusive-OR	
STA	Store A	As above but without immediate addressing.
STX	Store X	Absolute or indexed by Y addressing
STY	Store Y	Absolute or indexed by X addressing
LDX	Load X	Immediate, absolute or indexed by Y addressing.
LDY	Load Y	Immediate, absolute or indexed by X addressing.
CPX	Compare X	Immediate and absolute addressing.
CPY	Compare Y	Immediate and absolute addressing.

Mnemonic Code	Action	Addressing details
ASL	Arithmetic shift left	These can act on A alone or on memory using absolute or indexed by X addressing.
LSR	Logical shift right	
ROL	Rotate left	
ROR	Rotate right	
INC	Add 1 to memory	Absolute or indexed by X addressing.
DEC	Subtract 1 from memory	
BIT	Bit test on memory	Absolute addressing.
JMP	Jump to absolute address	Absolute addressing.
JSR	Jump to subroutine	

Answers to Self Test

1.1 It was non-volatile.
1.2 Memory is internal read/write memory. Store is external and non-volatile.
1.3 The home of the first successful audio cassette tape interface.
1.4 43200.
1.5 Disk Operating System.
1.6 Storage is non-volatile and external. Memory is volatile and internal.
1.7 It uses serial instead of parallel data transmission.
1.8 256.
1.9 One.
1.10 Sectors per track.
1.11 18.
1.12 EDSAC.

2.1 No.
2.2 It must be unique.
2.3 Where there are a large number of records.
2.4 Processing is fast.
2.5 102 characters.
2.6 Horizontally.
2.7 When the operator is not sure of the complete field information.
2.8 Because memory space is more limited than in store-based files.
2.9 True.
2.10 False.

3.1 OPEN 4,1,0,"NUCLEAR"
3.2 OPEN 7,1,2,N$
3.3 OPEN 15,8,15
3.4 OPEN 6,8,6,"1:GRANADA,S,W"
3.5 OPEN 7,8,7,"@0:FN$,S,W"
3.6 PRINT L

4.1 Because 'house-sweeping' action takes a longer time as the number of records grow. It would be time-wasting if this happened after each new record entry.
4.2 The sorting always begins with the most significant digit so without leading zeros, a number, such as 55, would be considered a larger number than, say, 255.

5.1 20 million.
5.2 6 hours.
5.3 500000.

6.1 River.
6.2 False. They go into the transaction file.
6.3 CLOSE the file.
6.4 In any program, the two-dimensional array to be stored must be the first array DIMensioned.
6.5 To ensure that any remaining data in the buffer is cleared to disk.

7.1 26^4=456976 ways.
7.2 34.
7.3 By allowing space for more records than will be used.
7.4 Trying consecutive locations until a vacant slot is found.
7.5 A previously deleted record slot.
7.6 Because it will be needed to produce the correct hash code when accessing the file.
7.7 To ensure that all file data is sent to disk.

Index

THE COMMODORE 64 IS A GREAT ORGANISER!

This highly practical book shows how to develop your own filing and database programs to suit your own individual requirements. It is suitable for both cassette and disk systems and presents fast and efficient techniques which the authors have tested and used in their own business.

Many useful general purpose subroutines are set out for you to use in your own programs. You are also shown how to incorporate machine code routines where appropriate, to achieve more speed and utilise memory more economically.

The Authors

A. P. Stephenson has a long and distinguished record as a writer on electronics and computing for the enthusiast. He is a regular contributor to the popular computing journals and is the author of seven other books including *Advanced Machine Code Programming for the Commodore 64.*

D. J. Stephenson is another life-long computer enthusiast. He has contributed to the popular electronics and computing journals including *Computing Today, Practical Electronics* and *Everyday Electronics,* and is a co-author of four other books with A. P. Stephenson.

Front cover illustration by Tony Roberts

COLLINS
Printed in Great Britain
0 00 383011 X

£7.95 net