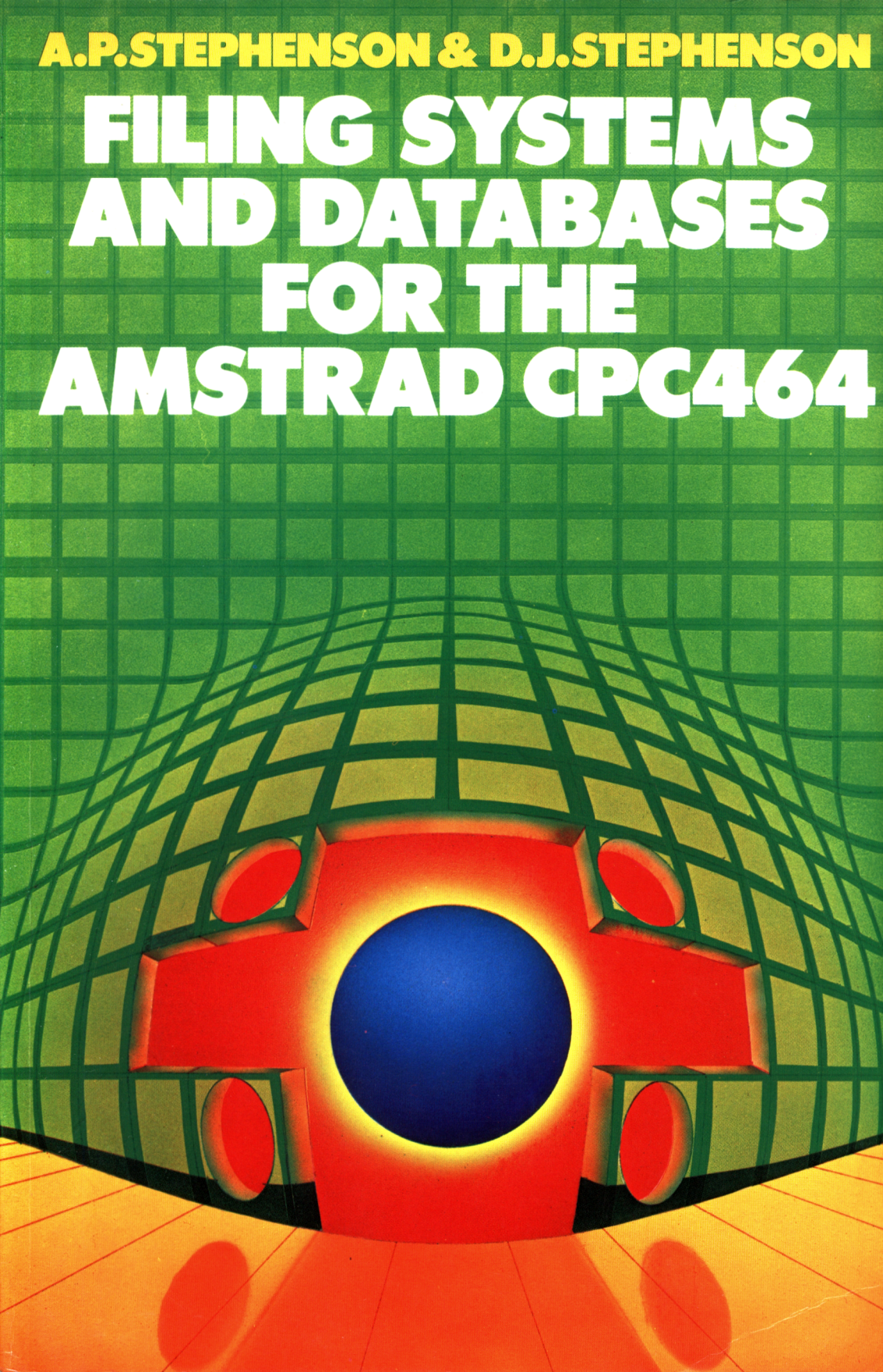
A.P.STEPHENSON & D.J.STEPHENSON

FILING SYSTEMS AND DATABASES FOR THE AMSTRAD CPC464



Filing Systems and Databases for the Amstrad CPC464

Other books for Amstrad users

Amstrad Computing
Ian Sinclair
0 00 383120 5

Sensational Games for the Amstrad CPC464
Jim Gregory
0 00 383121 3

Adventure Games for the Amstrad CPC464
A. J. Bradbury
0 00 383078 0

40 Educational Games for the Amstrad CPC464
Vince Apps
0 00 383119 1

Practical Programs for the Amstrad CPC464
Audrey Bishop and Owen Bishop
0 00 383082 9

Filing Systems and Databases for the Amstrad CPC464

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
Amstrad CPC464.
1. Amstrad CPC464 (Computer) 2. Data base
management
I. Title II. Stephenson, D. J.
001.64'42 QA76.8.A4

ISBN 0-00-383102-7

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 using the cassette tape unit built into the Amstrad CPC464. However, those who have disk drives fitted to their machines may find a good deal of the material relevant to their needs. As the title suggests, the leaning is towards the Amstrad CPC464 version of BASIC and Z80 machine code. However, the contents may also appeal to owners of other machines. It is hoped that the book will interest both the beginner and the experienced user alike although it is assumed that the reader will have already studied the user instructions which arrive with the machine.

Although full listings are given of major programs, they are based on a collection of subroutine listings designed to function as building bricks. It is hoped that they will be used to construct other programs with modifications and extra features to suit specific needs.

The first major program is a general purpose filing system offering a wide range of processing aids, including the ability to construct a subfile from a main file (i.e. extract from a main file all records which possess certain common features, and collect them into a separate subfile). The second program, which is in two parts, illustrates how the computer can be used for conducting multiple choice tests either as classroom checks on progress or in the form of home quiz sessions. Security checks have been included to simulate a formal classroom atmosphere. Apart from the educational value, the programs show that files can be used for storing questions and answers as well as 'records'.

BASIC sorting and searching techniques are described in detail. However, alternative fast machine code sorting routines are given together with full directions for splicing them into main programs. Since many users may not have assembler facilities, the object code is supplied in the form of hex loading programs written in BASIC.

Some terms 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 authorities 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 hair-splitting arguments.

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 Amstrad CPC464 RAM complement consists of a bank of eight chips, each chip capable of storing 64K bits.

Peripheral devices, such as cassette tape, 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 Amstrad CPC464 owners will be content with the cassette unit. However, as funds allow or as interest grows, a proportion of owners may eventually invest in disks. Because storage assumes great importance in filing systems, it is worth studying the properties of cassette and disk units.

Cassette tape storage

Cassette tape has been a boon to home computer users. 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 medium 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, 2000 baud can store $12 \times 60 \times 2000/10 =$ 144000 characters. This is equivalent to just over 140K 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 2000 baud, can store about 70K 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.

BASIC on the CPC464

The majority of the listings in this book are in BASIC but, because there is no such thing as a 'standard' BASIC, it is worth devoting a little space to Amstrad's version of the language. It is a powerful and imaginative version, marred only by the omission of procedures which means we are still stuck with GOSUBs. Defined functions (DEF FN) are available providing they are restricted to a single line. The majority of the familiar keywords, which form the hard core of the language, are still retained so there is no point in redescribing them here. However, there are a number of useful extras, seldom found in other popular versions, which have direct bearing on file handling and data presentation. The user instructions, which arrive with the machine, are accurate and expertly presented. It is possible that newcomers to the world of computer jargon may feel that some of the explanations rely just a little too heavily on Amstrad's 'meta language'. Meta language is concise and elegant but rather forbidding in appearance. The following selection of keywords merit further explanation because of their particular relevance to filing and because they may not already be familiar to users of other BASICs.

Automatic calls to subroutines

The keyword EVERY allows you to build into a program an *automatic* call to a subroutine at regular intervals. The call makes use of any one of the four resident delay timers. The syntax is as follows:

EVERY delay,timer GOSUB line number

The timer is in integer units of $\frac{1}{50}$ second. The timers are referenced Ø,1,2,3. For example,

EVERY 5Ø,3 GOSUB 5ØØØ

This will use Timer 3 to jump to the subroutine at line 5ØØØ every 50 time units (1 second). If this line is entered once, probably near the beginning of the program, it can be used for, say, *interrupting* a program at regular intervals to display a message at the bottom of the screen. In more general terms, the BASIC programmer can make use

of the *internal* interrupt system, a facility normally available only in machine code. The choice of four timers means that four independent timing interrupts can be used in the same program.

Disabling a timed interrupt

If a timed interrupt delay has been programmed, using EVERY, a clash of interests can occur. If a certain part of a program is vulnerable and must not be interrupted under any conditions, it is possible to *inhibit* the interrupt by using the keyword DI (Disable Interrupt). The inhibition will last until the key word EI (Enable Interrupt) is used. For example:

```
500 DI
510 vulnerable part of program
  .   .   .   .   .   .
  .   .   .   .   .   .
600 EI
```

End of file

EOF is a pseudo-variable representing End Of File. During a cassette tape input, it remains at ∅ until the end of the file is sensed, at which point it changes to −1. For example:

```
100 IF EOF < 1, THEN etc., etc.
```

Memory left

When creating a new file, it is reassuring to have a continuous update on the amount of memory left. One of the defects of most BASIC interpreters (the Amstrad version is no exception) is the way in which strings are stored. Untidy heaps of garbage tend to be distributed throughout RAM – relics of unfilled string variables. When there is no room left, the operating system carries out a process known as *garbage collection* in order to recover free space. It is possible to anticipate this by using the pseudo-variable FRE.

FRE(" ") enforces garbage collection before it totalises the free memory bytes. For example, PRINT FRE(" ") can be used at any point in a program. This operation takes a certain amount of time (could be several seconds) depending on the amount of data to be collected. FRE(∅) gives the bytes left but without garbage collection.

Limiting the BASIC area

The memory area above that used by BASIC is, in theory, available

for other purposes – for example, machine code subroutines. The top address of the area currently occupied by BASIC is always available in the dynamic variable HIMEM. The programmer can restrict the BASIC space by assigning a fixed address to the pseudo-variable MEMORY. For example, MEMORY &2ØØØ will set the upper address limit for BASIC. Memory above this, normally claimed by BASIC, is then released for machine code.

Finding a string within a string

The keyword INSTR (meaning 'instring') is useful for determining whether or not a certain group of characters is present within a string variable. The default syntax, using arbitrary variables, is INSTR (A$,B$) where B$ is the substring searched for within A$. If B$ is indeed within A$, the keyword returns the string position in A$ where B$ starts. If B$ is not found at all, then the number returned is zero. For example,

```
1ØØ IF INSTR(A$,B$) = Ø THEN PRINT"Substring not present"
```

This can contribute to user-friendliness in a routine which searches through a file for a given record. An end user may remember a few characters of the search key but not the whole so an escape clause of this nature will be well appreciated.

Converting lower- and upper-case

It is possible to ensure that all characters in a string are either upper-case or lower-case. The two keywords involved are UPPER$ and LOWER$. For example:

A$ = UPPER$(B$) will cause A$ to hold an all upper-case version of B$.
A$ = LOWER$(B$) will cause an equivalent lower-case version.

A useful example is where the operator has to enter a yes/no answer in response to a prompt. The response entered may be in upper-case or lower-case so a lengthy IF/THEN line would otherwise be required to cover both options. UPPER$ can be used to convert the input to upper-case, rendering the IF/THEN line unnecessary. Many other examples may come to mind, such as the sorting of strings when the first letter of each string can be either upper- or lower-case.

Alternative to the FOR/NEXT loop

The standard FOR/NEXT loop has always been the back-bone of

the BASIC language for programming repetitive sequences. Although Amstrad's BASIC naturally includes it, an alternative, and often preferable, structure called the WHILE/WEND loop is provided. The top of the loop is handled by the WHILE condition and the bottom by the single keyword WEND (which means 'While END'). The body of the loop continues to execute until the condition, or conditions, are satisfied. Although it is considered bad programming to jump out of a FOR/NEXT loop before its natural termination, it is quite permissible to avoid the WEND in WHILE/WEND loops.

The Amstrad tape unit

All program and subroutine listings in this book are designed for cassette tape files. The procedure for using tape is given in the user instruction manual supplied with the machine but the additional information which follows may help you gain the maximum benefit from the tape unit.

The cassette tape unit on the majority of home computers is a plug-in device which must be purchased separately. The self-contained unit in the Amstrad has the following advantages, over and above the obvious saving in additional cost:

(a) No untidy connecting cables and, consequently, no trouble with faulty plug connections.
(b) Because of the integrated design, the *signal levels between computer and tape unit are optimised and constant* (no fiddling around with volume and tone controls).
(c) As a direct result of constant signal levels, the speed of data transfer can be high – 2000 baud in fact!

There is a choice of two tape speeds, the normal (default) speed of 1000 baud or a software-selectable higher speed of 2000 baud. The selection procedure, carried out by the keyword SPEED WRITE, is as follows:

To establish the 1000 baud rate, enter	SPEED WRITE ∅
To establish the 2000 baud rate, enter	SPEED WRITE 1

These figures are nominal, as indeed are all baud rate quotations, because they assume that the data transferred consists of regular

alternations of the binary bits, 1 and ∅. In other words, the baud rate quoted is based, quite understandably, on *statistical* averages. Another factor which is often neglected is the presence of formatting data such as *end of block* characters and *inter-block* pauses in the data transfer.

When reading back a file from tape, the Amstrad senses the baud rate which was used when the file was recorded. LOAD"file name" or INPUT"file name" will handle the read process, irrespective of the baud rate used at the time when the file was saved.

Speed of information storage

It is all very well quoting baud rates but, if you are a newcomer to computing and the Amstrad is your first machine, you will appreciate a more down to earth measurement of tape speed. You will probably want to know how long it will take to store a typical page of A4 text. To answer this, we must start with the time it takes to store one keyboard character. One character in the ASCII code requires 8 bits but, as previously stated, at least another couple of bits are required to signal the beginning and end of each character so we end up with a provisional round figure of 10 bits per character. A tape speed of 2000 baud (2000 bits per second) means that the character speed is 2000/10 = 200 characters per second or 12000 characters per minute. If we assume that the average word in the English language contains five letters, this is equivalent to 12000/5 = 2400 words per minute. A typical page of typed text, using double spacing between lines, contains about 28 lines of 13 words per line - 364 words per page. At 2400 words per minute, this means that the speed of data transfer via cassette tape works out to about six and a half pages of typed text per minute. This doesn't seem very fast but, before we start grumbling, we ought to distinguish between a typical page of normal text and a typical computer record. A 'normal page' of text or reading matter, such as you would find in Hansard or a novel, contains a large percentage of *padding*. It may be pleasant, it may be amusing and it may even be important to those of a literary bent but, as far as adding real information value to the page goes, it is still padding. In contrast, a typical computer record is, or should be, almost void of padding. Every word should pull its weight and there should be a minimum of frills and pleasantries. It should be pure information. Any word which can be removed without destroying the information value shouldn't have been there in the first place. It is surprising how much information can be condensed into a record containing as few as fifty words. For example, in a well-designed record layout, fifty words is

normally quite adequate for an individual record containing the name, full address, postal code, telephone number, sex, trade/profession and bank balance. There may even be enough words left over for storing a few intimate characteristics. Taking fifty word records as an example, we obtain a tape transfer speed approaching 2400/50 = 48 records per minute which is fairly acceptable and should provide ammunition to direct against those with fat wallets who habitually sneer at cassette tape files.

Choice of baud rate

In general, the higher the baud rate the greater the chance of a read error so if the data you wish to store on tape is particularly precious it is probably advisable to use the lower 1000 baud rate. However, we should mention that we have used the higher rate consistently during the production of this book and have never yet encountered a read error. It appears to be an extremely reliable system. The reliability of the 2000 baud rate is not quite so good if a program is recorded on one machine and played back on another. This is because there may be slight differences in the alignment or properties of the recording heads. On the same machine, this is not so important because it is the same for both recording and playback. So, if you intend your program to be read on other people's machines, it might be safer to record at 1000 baud.

Dangers of the leader tape

The majority of cassettes available in the high street shops have several inches of blank *leader tape* before the normal oxide coated region. The leader tape makes a strong mechanical joint with the winding spool and cleans the record/playback head but, naturally, it cannot store information. If you initiate the recording process too early – that is to say, before the leader has passed the recording head – the first part of your data may be lost and will be impossible to recover when you later try to load it back. To prevent such a catastrophe, it is worth pressing the PLAY button for a couple of seconds after a full rewind in order for the leader tape to pass over the heads before you begin the recording procedure. A good plan is to advance the tape until the digital counter on the cassette unit reaches 5. You can buy special leaderless cassettes but they are not so freely available. Cassettes without leader tapes can lead to a false sense of security but it is still advisable to wait at least a second to allow the

first few inches of tape to pass. These first few inches can be degraded by the tight coiling around the spool. So-called 'digital quality' tapes often have larger diameter spools so the coiling is less tight on the inner layers. Finally, one word of warning – think carefully before buying bargain-price cassettes. Saving a small amount on the price of a cassette can often be false economy. Good quality cassettes, particularly those designated as suitable for digital data, are manufactured to closer tolerance limits so there is far less chance of *drop-out* areas over the oxide surface. Drop-out (i.e. irregularities in the oxide coating) is relatively unimportant when recording normal music but one bit reversed on a data tape could lead to chaos.

Back-up copies

If a tape fails to load one of your own programs the worst that can happen is a bout of temper and a lot of extra work in retyping the listing. However, if the failure happens to be a data tape containing important business or private records, the effect is much more serious. A file of a few hundred records can represent many hours of careful work entering information, some of which might never again be available. Although the Amstrad tape system seems very reliable, loss of data can sometimes occur due to either a poor cassette or finger trouble on the part of the operator during the initial recording or subsequent playback. It is so easy, particularly when feeling tired, to press PLAY instead of PLAY and REC so that nothing goes on the tape. You may be unaware of the mistake until you attempt to load back the tape at a later date.

To avoid rage and frustration, always keep two or even three copies on the same tape and, for extra safety, have additional copies on separate tape. Taking two copies, one after the other on the same cassette, or on the opposite side, will certainly reduce risk but there is always a chance that it could be mislaid, dropped on a hard surface or placed on the top of a hi-fi loudspeaker. Magnets, especially speaker magnets, have a malignant influence on cassette data and the two should be kept at a safe distance from each other. On the assumption that the same accident couldn't happen to two separate cassettes, you should consider, in spite of the extra expense, keeping a back-up copy on another one. It is not good practice to cram different programs and odd scraps of data onto one tape because (a) it can take some time to locate the wanted part and (b) it causes problems with cassette labels.

If you don't mind wasting unused tape, the best rule to adopt is to use a *separate tape for each program or data file*. All this advice may lead to the impression that keeping computer files on tape is a potentially hazardous business. As already stated, the Amstrad tape system is extremely reliable but this is no reason why every precaution possible should not be taken to preserve the integrity of files. Even in a large professional computer complex, using hardware of high cost and impeccable performance, the taking of file copies, together with their preservation and organisation, is enforced with almost military discipline.

Labelling cassettes

The importance of labelling increases in direct proportion to the number of cassettes in use. If you have only two or three tapes, it is probably a waste of time writing labels because you will be constantly changing the contents. As your collection grows, or if you are using the machine to keep important records, it is wise to adopt a methodical approach to labelling. You can buy labels designed for cassettes but these, in our opinion, are a little overpriced. The best plan is to buy a roll of pre-gummed plain labels and cut them to size as you want them. Stick the labels on the cassette itself, not on the plastic container, because (a) the cassette could get into the wrong case, and (b) once you stick a label on the case, it is very difficult to remove it without scraping and leaving a smudgy mess behind. On the other hand, labels seem to peel off cleanly from the body of the cassette. If cassette contents change, it is better to replace the entire label rather than make alterations. A label with multiple alterations looks scruffy and reduces confidence. Another method is simply to label the cassettes 1, 2, 3, etc. and list the contents in a Tape Register book. The trouble with registers is that initial enthusiasm for their upkeep often wanes. It doesn't really matter what system you use providing you have one and that you stick to it.

Cassette data blocks

A cassette unit, because it is a mechanical system, operates within a time scale which is thousands of times slower than the computer. To allow for the time difference, the Amstrad CPC464 employs a 2K area of memory, called a *block*, which is used as a *buffer* between the cassette and the computer. Data transfers between computer and cassette take place from within this buffer area. When a program is

being loaded, the screen message displays each 2K block number.

Allocation of cassette buffer area

The Amstrad uses *dynamic* allocation for cassette files. This means that space for the cassette buffer is allocated only when needed. There are sound reasons why the designers decided on dynamic allocation rather than the conventional fixed buffer area. Many programs may not use cassette data files so why waste valuable RAM by reserving space that may not be used? For instance, a disk drive may be connected thus making the cassette unit redundant. The Amstrad CPC464 can support two open cassette files. A 2K buffer area is needed for each of the input and output buffers, thus making 4K in total.

The highest available memory location available to BASIC is stored in the variable HIMEM. Strings are stored from HIMEM downwards and are referred to as the 'heap'. In order for the cassette buffers to be allocated, BASIC performs a house-keeping operation to ensure that the heap is as small as possible and then physically moves it down in memory by 4K. HIMEM is subsequently set to this lower value. When the cassette buffer has performed its task, BASIC tries to reclaim the memory taken by the buffer. It can only do this if the buffer area is immediately above HIMEM. With this proviso, the heap is moved back to its original location and HIMEM reset to its default value.

Dynamic allocation of cassette buffers is fine in the majority of cases but is a bit of a nuisance in RAM-based 'database' type programs. The task of house-keeping and moving the heap can take anything up to a couple of minutes to perform, depending on its size. This delay can be annoying but fortunately there is a simple solution. The buffers can be permanently allocated by the following few program lines:

```
10 OPENOUT "Buffer"
20 MEMORY new
30 CLOSEOUT
```

where 'new' is the updated value of HIMEM given by the following relation:

new = default HIMEM−4K−1

The −1 term is needed to ensure that the cassette buffer opened in line 10 is not immediately above HIMEM when the file is closed (see above). BASIC is thus unable to reclaim the memory allocated to the

buffers. However, if, at a later stage in the program, the cassette buffers are needed, the previously allocated space will be reused. Here is a practical example:

Assuming a perfectly standard machine (no disk drives, light pens, etc.) the default HIMEM will be &AB7F. Knocking 4K off HIMEM gives &9B7F and a further reduction of 1, to comply with the above, makes &9B7E. Therefore, the cassette buffers can be statically allocated on a standard machine by the following:

```
1Ø OPENOUT "buffer"
2Ø MEMORY &9B7E
3Ø CLOSEOUT
```

An alternative version is:

```
1Ø OPENOUT "buffer"
2Ø MEMORY HIMEM-1
3Ø CLOSEOUT
```

Although the later example may appear preferable, it suffers a severe disadvantage in that a further 4K will be nobbled each time a program containing the above lines is reRUN. Using the former method, it is easy to reserve a chunk of memory for machine code routines at the same time. For instance, if &2ØØ (512 decimal) bytes are required for a machine code routine then line 2Ø could be changed to:

```
2Ø MEMORY &997F
```

Your machine code can then be assembled from &998Ø onwards. Incidentally, this area is where our machine code sorting routines, described in Chapter 5, are located.

Disk storage

This book is concerned almost entirely with cassette tape files but, for the benefit of readers who may be new to computing and may eventually invest in a disk drive, some brief discussion on this kind of storage system is justified. The most obvious advantage of the disk drive over the cassette is speed of data transfer. Even though we have praised the speed of the Amstrad CPC464 cassette there is no denying that it is still slow in comparison with a well-designed disk system. However, it is worth mentioning that the speed of a disk drive

depends, to a large extent, on the *disk operating system* – The *DOS*. Unless the DOS, which is simply a chunk of complex software, is designed carefully, the inherent speed of the mechanical disk drive is inhibited and the resulting system could, in the worst case, end up only two or three times faster than a good cassette system. In fact, one popular home computer employs a disk drive system which most owners, even the most easily pleased, would describe as 'a little sluggish'. But, in all fairness, the speed of a disk drive is only one of its virtues. As far as filing systems are concerned, the most outstanding advantage of the disk lies in its *random access* nature. This means that it is possible to go straight (or nearly straight) to the exact record or file you need without having to read through all the unwanted data which happens to come before it. This is because the read heads, which pick up the data from the rotating disk, are at the end of an arm which can travel radially across the disk surface until a desired track is reached. The mechanism responsible for the radial movement is called a 'stepping motor' and, because the movements are digitally controlled, can position the read head with great accuracy onto the centre of a desired track. It is this random access property, coupled with the increased disk rotation speed, which is responsible for the popularity of disk-based files. The DOS will normally include a set of commands, appearing as extra BASIC keywords, which allow users to communicate with the disk.

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. Tape speed is conveniently measured in bauds. 1 baud is a rate of one information bit per second.
6. It normally takes 10 bits to represent each character on a tape.
7. Meta language is a specialised set of symbols used to describe the behaviour of BASIC keywords. It is concise and, unlike plain language descriptions, free of ambiguity.
8. The keyword, EVERY, allows access to any of the four interrupt timers. The time unit is $1/50$ second.

9. Interrupts generated by EVERY can be disabled with DI and re-enabled with EI.
10. To enforce garbage collection, use FRE(" ").
11. UPPER$ ensures all upper-case text; LOWER$ ensures all lower-case text.
12. Default tape speed is 1000 baud (SPEEDWRITE ∅). Speed can be doubled when saving by using the command SPEEDWRITE 1.
13. A tape LOAD automatically senses the speed at which the tape was SAVED.
14. Make sure that the leader tape is passed before commencing a recording.
15. Use C12 tapes and, if possible, put only one file on a tape.
16. Information on tape is laid down in a series of blocks. 1 block = 2K bytes.
17. Two 2K areas of RAM are required for cassette buffer purposes.
18. The two cassette buffer areas in RAM are not in fixed addresses. They float up and down dynamically above HIMEM.
19. The buffers can be given fixed addresses by means of a direct assignment.

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 are there in a block?
1.5 Why are there two 2K tape buffer areas in memory?
1.6 EOF is a pseudo-variable containing either ∅ or −1. What is the significance when it contains −1?
1.7 What is the difference between FRE(∅) and FRE(" ")?
1.8 A tape recorded at 2000 baud may play back without fault by the machine on which it was saved but may fail when played back on another machine. Why?

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)":GOTO 100
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 Amstrad CPC464 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 may 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.

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

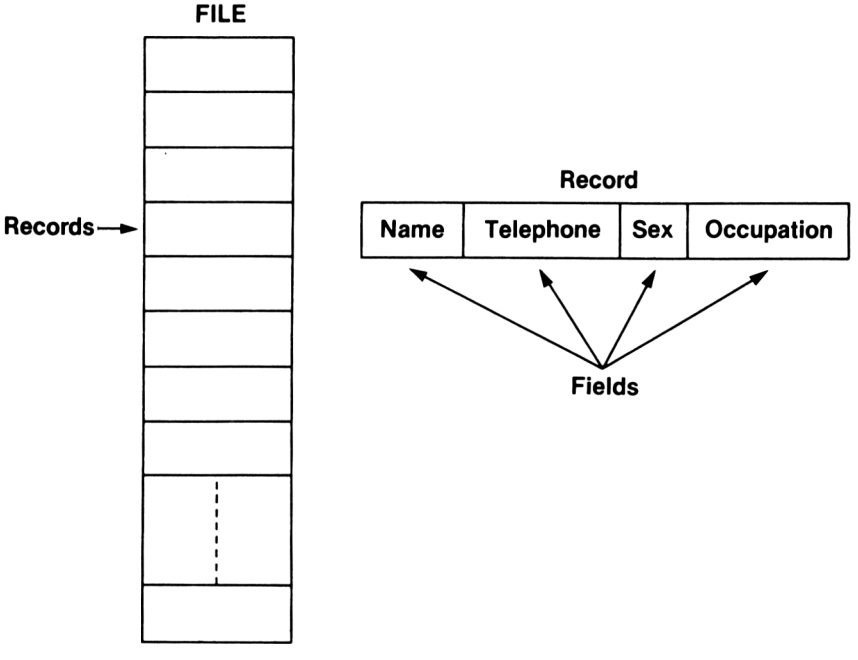


Fig. 2.1. Components of a file.

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. 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, RAM-based files are useful. In cases where the length of the files is 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.

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. However, sorting is 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.

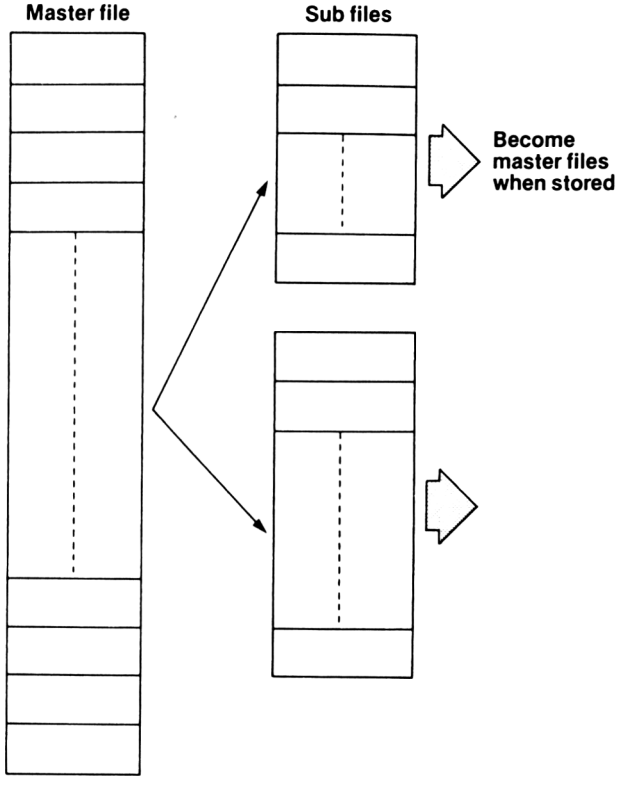


Fig. 2.2. Splitting files.

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!

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 Amstrad 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.

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.

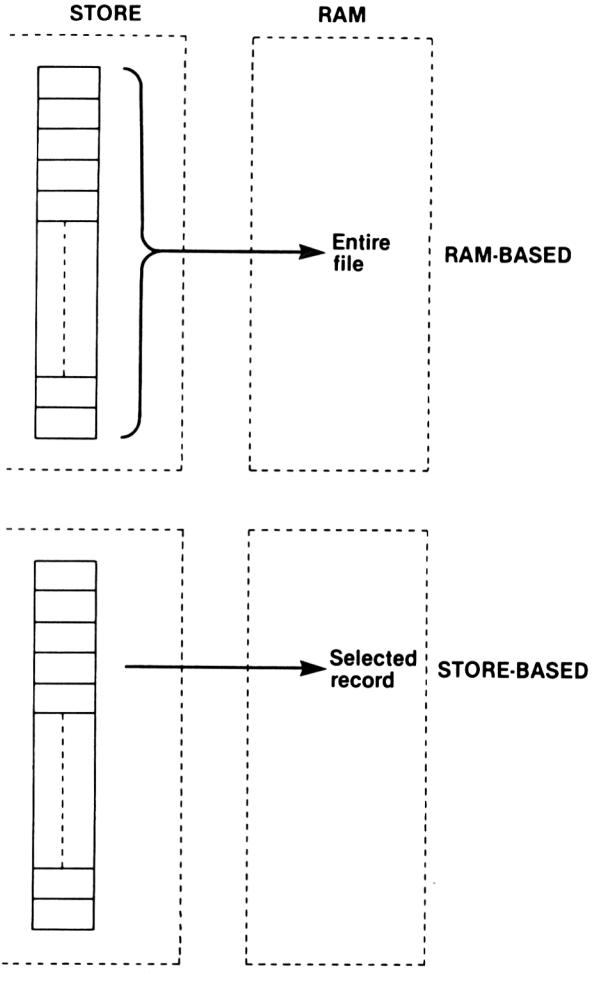


Fig. 2.3. RAM- and store-based files.

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.

This book deals exclusively with RAM-based files that can be used on a standard Amstrad CPC464.

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. Good supporting documentation is an important contribution to user-friendliness.
6. A data file cannot be directly 'SAVEd' or 'LOADed'. It can only be accessed from within a program.
7. A record is a component of a file.
8. A field is a component of a record.
9. The field width is the number of characters in a field.
10. The field heading gives the meaning to be attached to the information within the field.
11. String fields are those which can contain any character (except control characters).
12. Numeric fields contain only numbers, either integers or floating point.
13. The key field is the one which uniquely identifies the record.
14. Numerical key fields are favoured in large files.
15. The file size refers to the number of records it holds.
16. The space a file occupies in store depends on both the file size and the field sizes.
17. 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.

18. 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.
19. Menu-driven filing programs revolve around an option page.
20. Options are chosen in command-driven filing programs from a 'command line' at the bottom of all displays.
21. Sort routines are easy if the file is RAM-based.
22. Subfiles contain selected records within master files. When stored, they become master files in their own right.
23. Print routines can only exploit the full potential of a printer if they are purpose-designed for that model.

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?

Chapter Three

Simple Filing Programs and Building Bricks

The advantage of subroutines

The programs in this book assume that the reader has already gained some experience in the CPC464 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*. A collection of general purpose subroutines are introduced later in this chapter which will serve as 'building bricks' for integrated filing systems.

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. 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 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 introducing chaos.

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 unnecessarily complex. 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 over-packing with graphic symbols and colour information. In any case, it is easy to add these touches afterwards.

Variable names
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 is 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. 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,

```
10 REM DISPLAY DATA STATEMENTS
20 READ filesize%,fields%
30 DIM A$(fields%,filesize%)
40 READ heading1$,heading2$
50 REM READ DATA INTO RECTANGULAR ARRAY
60 FOR R%=0 TO filesize%-1
70 FOR F%=0 TO fields%-1
80 READ A$(F%,R%)
90 NEXT:NEXT
100 GOSUB 150
110 END
120 '
130 '
140 REM VIEW FILE SUBROUTINE
150 CLS
160 PRINT TAB(7) "FILE CONTENTS"
170 PRINT "---------------------------"
180 PRINT heading1$ TAB (19) heading2$
190 PRINT "---------------------------"
200 FOR R%=0 TO filesize%-1
210 FOR F%=0 TO fields%-1
220 PRINT A$(F%,R%) TAB(19);
230 NEXT
240 PRINT
250 NEXT
260 RETURN
270 '
280 '
290 REM NUMBER OF RECORDS AND FIELDS
300 DATA 8,2
310 '
320 '
330 REM FIELD HEADINGS
340 DATA NAME,TELEPHONE
350 '
360 '
370 REM FILE INFORMATION
380 DATA DEREK,678 9993
390 DATA GILL,453 9345
400 DATA CATH,777 5847
410 DATA STEVE,345 6694
420 DATA JOHN,979 6836
430 DATA GRAYHAM,685 3737
```

```
440 DATA PAT,574 6858
450 DATA PAUL,786 8797
```

Program 3.1. Display DATA statements.

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

filesize % = file size = maximum number of records in each file
fields% = number of fields
heading1$ = heading of field 1
heading2$ = heading of field 2
F% = a particular field index
R% = a particular record index
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$(F%,R%) as long as consistency is observed. However, a two-dimensional or *rectangular* array is stored in the CPC464 in column major order. That is to say, the sequential storage sequence of string pointers in memory is A$(Ø,Ø) A$(1,Ø) A$(2,Ø) A$(Ø,1) A$(1,1) A$(2,1) ...A$(Ø,n) A$(1,n) A$(2,n). 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 eight records, each of two fields, giving NAME and TELEPHONE numbers of friends.

The parameters filesize% and fields%, 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 CPC464 BASIC for arrays not exceeding ten but there is always 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 heading1$ and heading2$. 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$(fields%,filesize%) and presents it in humanised form for the screen display. The headings, 'heading1$' and 'heading2$', 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 ";" 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. Mode 1, the 40 characters per line mode, was used in the example because of the increased screen clarity. When there are only two fields to be displayed, 40 columns is normally quite adequate although the mode can easily be changed to allow 80 columns. 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

To maintain consistency, previous variable names have been retained but, as the following list shows, there are one or two extras:

filesize% = file size = maximum number of records in each file
fields% = number of fields
heading1$ = heading of field 1
heading2$ = heading of field 2
F% = a particular field
R% = a particular record
A$(F%,R%) = an element of the two-dimensional array holding the main file data
SEL% = option number selected
K$ = general purpose variable
flag% = is a yes/no search flag (1 = record found, 0 = no record).

```
10 REM SIMPLE PROGRAM BASED
20 REM FILING SYSTEM
30 READ filesize%,fields%
40 DIM A$(fields%,filesize%)
50 READ heading1$,heading2$
60 REM READ DATA INTO RECTANGULAR ARRAY
70 FOR R%=0 TO filesize%-1
80 FOR F%=0 TO fields%-1
90 READ A$(F%,R%)
100 NEXT:NEXT
110 '
120 '
130 GOSUB 210:REM MENU
140 IF SEL%=1 THEN GOSUB 390:GOTO 130
```

```
150 IF SEL%=2 THEN GOSUB 540:GOTO 130
160 CLS:PRINT "EXIT"
170 END
180 '
190 '
200 REM MENU SUBROUTINE
210 CLS
220 PRINT TAB(3) "PROGRAM BASED FILING S
YSTEM"
230 LOCATE 1,6
240 GOSUB 730:REM DRAW LINE
250 PRINT"(1) VIEW FILE"
260 PRINT"(2) DISPLAY RECORD"
270 PRINT"(3) EXIT PROGRAM"
280 GOSUB 730:REM DRAW LINE
290 LOCATE 1,15
300 PRINT"Select option ";
310 K$=INKEY$:IF K$="" THEN 310
320 SEL%=VAL(K$)
330 IF SEL%<1 OR SEL%>3 THEN 210
340 CLS
350 RETURN
360 '
370 '
380 REM VIEW FILE SUBROUTINE
390 PRINT TAB(7)"FILE CONTENTS"
400 GOSUB 730:REM DRAW LINE
410 PRINT heading1$ TAB(19) heading2$
420 GOSUB 730:REM DRAW LINE
430 FOR R%=0 TO filesize%-1
440 FOR F%=0 TO fields%-1
450 PRINT A$(F%,R%) TAB(18);
460 NEXT
470 PRINT
480 NEXT
490 GOSUB 780:REM HOLD DISPLAY
500 RETURN
510 '
520 '
530 REM GET RECORD SUBROUTINE
540 flag%=0
550 PRINT"ENTER "heading1$" REQUIRED ";
560 INPUT K$:K$=UPPER$(K$)
```

```
570 IF K$="" THEN 550
580 CLS:LOCATE 1,4
590 PRINT heading1$ TAB(19) heading2$
600 GOSUB 730:REM DRAW LINE
610 FOR R%=0 TO filesize%-1
620 FOR F%=0 TO fields%-1
630 IF A$(0,R%)=K$ THEN PRINT A$(F%,R%)
TAB(19);:flag%=1
640 NEXT
650 NEXT
660 IF flag%=0 THEN PRINT"THIS "heading1
$" NOT ON FILE"
670 GOSUB 730:REM DRAW LINE
680 GOSUB 780:REM HOLD DISPLAY
690 RETURN
700 '
710 '
720 REM DRAW LINE SUBROUTINE
730 PRINT STRING$(40,CHR$(154));
740 RETURN
750 '
760 '
770 REM HOLD DISPLAY SUBROUTINE
780 PRINT:PRINT"PRESS ANY KEY TO CONTINU
E"
790 K$=INKEY$:IF K$="" THEN 790
800 RETURN
810 '
820 '
830 REM NUMBER OF RECORDS AND FIELDS
840 DATA 8,2
850 '
860 '
870 REM FIELD HEADINGS
880 DATA NAME,TELEPHONE
890 '
900 '
910 REM FILE INFORMATION
920 DATA DEREK,678 9993
930 DATA GILL,453 9345
940 DATA CATH,777 5847
950 DATA STEVE,345 6694
960 DATA JOHN,979 6836
```

```
970 DATA GRAYHAM,685 3737
980 DATA PAT,574 6858
990 DATA PAUL,786 8797
```

Program 3.2. Program-based file.

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 SEL%.
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 'heading1$' is, in this case, NAME. The response is entered into the general purpose variable K$ 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$(0,R%), which is the key field of all records, is compared with the name in K$. Remember that R% is the record variable and 0 is the *first* field which, by definition, is 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% is set to 1. If, after scanning through all records, no match is found, flag% 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 Figure 3.1 will help in following the overall structure.

It would be easy to include a 'search' option for finding a record by

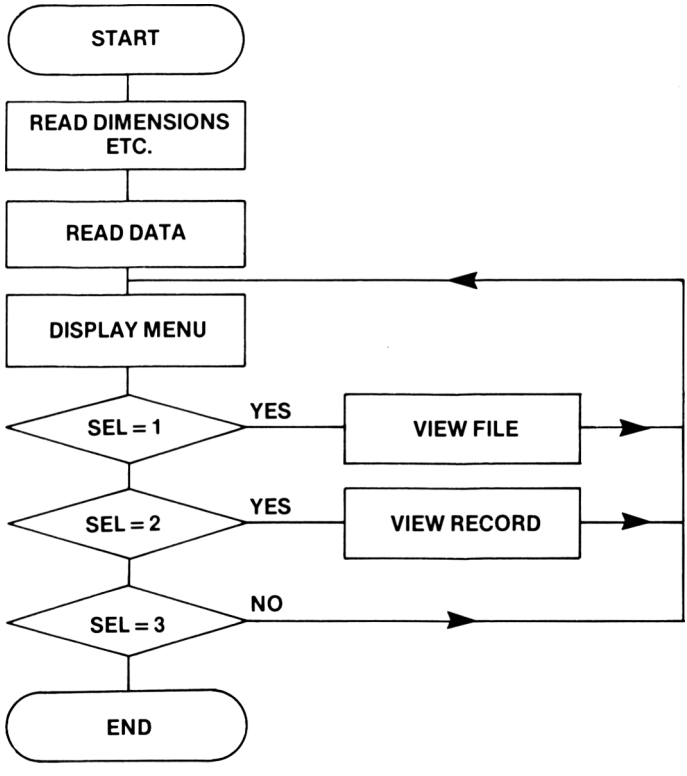


Fig. 3.1. Flowchart for Program 3.2.

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'. Even when a program is SAVEd it is still often referred to as a 'file' although, to distinguish it from *data files*, it should be called a *program file*. The procedure for saving and loading programs has been made easy because they are 'bread and butter' functions on any home computer. It would be irksome and certainly user-unfriendly to force the average end user to worry about opening and closing files each time a normal program needed saving or loading. Instead, most machines, including the CPC464, save and load programs on receipt of the simplified commands SAVE"name" and LOAD"name".

Many programs, filing programs in particular, require the data 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 to that of program files. For example, when an ordinary program is saved, the BASIC keywords are encoded into two-byte groups known as *tokens*. Data files, on the other hand, are stored as straightforward ASCII characters, with special header information and inter-block markers.

Opening a data file for output

At the point in a program where data is to be stored on tape, it is necessary to open a tape file by using the following OPENOUT statement:

```
OPENOUT "file name"
```

The file name can be literal, such as OPENOUT "Personnel", or a string variable which has been previously assigned, such as OPENOUT Personnel$. No more than sixteen characters are allowed for a file name.

Using PRINT #9

Once the file has been opened for output, the data can be written to tape by a special print statement having the format

```
PRINT #9, variable list
```

The '#9' informs the system that the 'printing' is to be carried out on *output stream* '9', which is the cassette tape unit, rather than on the monitor screen by default. The term 'variable list' above means one or more variables, separated by commas such as

```
PRINT #9, filesize%,fields%
```

When a complete data file is to be printed to tape, there is a need to store certain leading particulars of the file as well as the main record items. For example, when the file is loaded, the program which loads it will have no information as to the length of the file or the number of fields in each record. These details must be obtained from the data tape before loop parameters can be set up for reading the bulk of the file.

The baud rate at which the data is sent to tape will be the default

speed of 1000. However, if you wish to send it along at 2000 baud, then follow the OPENOUT line with SPEEDWRITE 1.

Closing a file

After the data has been output to tape it is important to CLOSE the file when you have finished with it because there may still be data left in the cassette buffer area. The act of closing the file flushes out any data that remains in the buffer and writes it to tape. CLOSEOUT is the relevant BASIC keyword to perform this.

Opening a data file for input

To retrieve a data file from tape, it is again necessary to open a file, but this time, the keyword is OPENIN followed by the file name:

OPENIN"filename" or OPENIN string variable

Using INPUT #9

Normally the INPUT statement receives data from the default input peripheral, which is the keyboard. To input data from tape, the equivalent form is

INPUT #9, variable list

For example,

INPUT #9,filesize%,fields%

After all data has been retrieved from tape, the input file should be closed in the usual way with CLOSEIN.

To illustrate the procedure for writing to and reading data from tape, first key in the few lines of Program 3.3.

```
10 REM STORING DATA ON TAPE
20 CLS
30 PRINT"ENTER SOME CHARACTERS "
40 LINE INPUT K$
50 OPENOUT "TEST"
60 PRINT#9,K$
70 CLOSEOUT
80 END
```

Program 3.3. Storing data on tape.

Before running the program, ensure that you have a spare tape in place and rewound to receive the data. Once the data is on tape, perform a hard reset (or switch the computer off and on again). Now key in Program 3.4 which should read back the data stored by the previous program.

```
10 REM READING DATA BACK FROM TAPE
20 CLS
30 PRINT"PLEASE REWIND DATA TAPE"
40 OPENIN "TEST"
50 INPUT#9,G$
60 CLOSEIN
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 to the correct place before running the program. The machine must have been emptied of data by the hard reset after Program 3.3 so the data printed out by Program 3.4 must have originated from the data tape. Although the variable K$ was used for printing the tape and G$ was used to receive it back, the same variable could have been used in both programs.

General-purpose building bricks

The next chapter is dedicated entirely to the description of a single practical filing system. The first, and relatively small part of the program will be recognised as a self-contained *control section*. The remainder of the program is simply a collection of subroutines, called as needed from the control section. Although some of the subroutines are tailor-made to suit this one program, most of them may be considered as standard building bricks which can be spliced into other programs, either as they stand or with slight modification. When devising building bricks for use in a range of programs, there is always the problem of size to consider. If we try and pack too much sophistication into one subroutine, the general purpose nature is endangered and we defeat the main guideline which is that a subroutine should have only one clearly defined function. Like all guidelines, slavish adherence is not necessary or, indeed, recommended. It is often a matter of personal judgment. Collecting a

library of building bricks saves having to re-invent the wheel in every new program.

The variable names used in general purpose building bricks may not necessarily be the most meaningful when sliced into application programs, but changing names is a trivial extra task. However, in the examples which follow, most of the variable names will coincide with those used in the major programs appearing in the remaining chapters of this book.

Subroutine building bricks

Subroutines are often *hierarchical* in structure. That is to say, some subroutines call on other subroutines to carry out minor tasks in the same way that a large organisation passes on some of the less important work to subcontractors. It is convenient to distinguish between *high* and *low level* subroutines. For example, there may be a subroutine for displaying one of the records within a file (a high level subroutine) which, in turn, may call on another subroutine (a low level subroutine) for drawing field marking lines across the screen. Incorporating the line drawing in the display subroutine itself would appear to be the obvious plan but it would not be very efficient. Drawing a line across the screen is a trivial but nonetheless useful task and likely to be wanted in many other subroutines. Although displaying a record is a general purpose task in any filing program, drawing a line is even more general purpose, so the setting up of an independent, low level subroutine is justified.

Subroutine DRAW LINE
Call with GOSUB 233Ø.

```
2320 REM DRAW LINE
2330 PRINT STRING$(40,CHR$(154));
2340 RETURN
```

The STRING$ function should always be used in preference to the simple PRINT"————————————————" form. It is elegant and certainly more economical on memory. CHR$(154) is a special graphics character, forty of which are used to trace out the line. Any other character can be used to draw the 'line' but avoid the asterisk – it is reminiscent of the early days of home computing when gaudy displays were the order of the day.

Subroutine GET LINE INPUT
Call with GOSUB 226Ø.

```
2250 REM GET LINE INPUT
2260 LINE INPUT K$
2270 IF K$="" THEN 2260
2280 K$=UPPER$(K$)
2290 IF LEN(K$)>18 THEN K$=LEFT$(K$,18)
2300 RETURN
```

When a screen prompt requests string input from the keyboard, some responses will be unacceptable. This subroutine rejects the null string caused by an operator mistakenly pressing the ENTER key before entering text. The use of LINE INPUT is preferable in filing programs because text often contains punctuation and other characters which the ordinary INPUT statement would reject. The text is entered into a temporary *global* variable K$. After the subroutine has returned, the text in K$ will normally be assigned to another variable. The subroutine also allows the operator the freedom to enter text in either upper- or lower-case. Use has been made of Amstrad's delightful little keyword UPPER$ which ensures that all input text, irrespective of whether it was entered in upper- or lower-case, is converted to upper-case. It is important, when sorting alphanumeric text, that all characters are of the same 'case' because sorting is carried out on ASCII values. Unless steps are taken to rectify mixed-case text, sorting could produce some weird results.

The subroutine also restricts the length of all input to 18 characters. It does this by cutting off (truncating) characters in excess of 18 rather than rejecting the entire input. This is a specific, rather than general purpose, restriction and the line responsible can always be altered or left out altogether. It has only been incorporated here because it satisfies the needs of Program 4.1 which appears in the next chapter.

Subroutine PRESS ANY KEY
Call with GOSUB 233Ø.

```
2510 REM PRESS ANY KEY
2520 PRINT:PRINT"Press any key to contin
ue"
2530 K$=INKEY$:IF K$="" THEN 2530
2540 RETURN
```

This employs the widely known dodge for freezing a display or halting a program until the operator is ready. Note the appearance again of the ubiquitous K$ variable. Wherever K$ appears in our programs it can be considered as a temporary variable. Using the same global variable over and over again is good practice. It saves memory and simplifies the appearance of a program. Too many different variables in a listing make it difficult to follow.

Subroutine GET FILENAME
Call with GOSUB 237ø.

```
2360 REM GET FILENAME
2370 CLS:PRINT"Enter filename"
2380 GOSUB 2260
2390 IF LEN(K$)>16 THEN PRINT"Too long:E
nter again":GOTO 2380
2400 filename$=K$
2410 RETURN
```

The file name is required at several points in a filing system. The prompt for the name is displayed and the response is handled and checked by the GET LINE INPUT subroutine. The additional reduction to 16 lines is to satisfy the 16-character restriction on file names. Note that the variable, filename$ is assigned to K$ before the final return.

Subroutine SAVE FILE
Call with GOSUB 7øø.

```
690 REM SAVE FILE
700 GOSUB 2370
710 SPEED WRITE 1
720 OPENOUT filename$
730 PRINT#9,filesize%,fields%,L%
740 FOR R%=0 TO L%
750 FOR F%=0 TO fields%
760 PRINT#9,A$(F%,R%)
770 NEXT:NEXT
780 CLOSEOUT
790 RETURN
```

Outline information on opening and closing files has already been given. This is a complete subroutine for printing a data file onto tape. It assumes that certain information resides in the following variables before it is called:

Maximum number of records allowed in filesize%
Maximum number of fields allowed in fields%
Current number of records in L%
All text to be in the array, A$(F%,R%)

The subroutine first calls on GET FILE NAME in order to open the file for output. Three items, constituting the heading information, are first sent to tape. The first two are needed in order to feed the DIMension statement when the data file is read back at some later date. The third item, L%, is needed as a FOR/NEXT loop parameter. The two nested loops print the array to tape. The inner loop takes care of the fields of each record and the outer loop takes care of the complete records. No further screen prompts are required for operating the tape unit because they are issued automatically by the OPEN statement.

Subroutine LOAD FILE
Call with GOSUB 57Ø.

```
560 REM LOAD FILE
570 GOSUB 2370
580 OPENIN filename$
590 INPUT#9,filesize%,fields%,mainsize%
600 flag%=1
610 DIM A$(fields%,filesize%)
620 FOR R%=0 TO mainsize%
630 FOR F%=0 TO fields%
640 INPUT#9,A$(F%,R%)
650 NEXT:NEXT
660 CLOSEIN
670 RETURN
```

After a preliminary call for the file name, the three items of heading information are read in. The first two variables are identical to those used in the SAVE subroutine but the third, although it was saved under the name L%, is changed to mainsize%. The reason for this may be obscure at the moment but it is all to do with programs which may distinguish between a *main file* and a *subfile*. The nested FOR/NEXT loops for reading in the array are identical in form to the SAVE subroutine except, of course, for the substitution of INPUT #9 in place of PRINT #9.

Subroutine DISPLAY FILE
Call with GOSUB 82ø.

```
810 REM DISPLAY FILE
820 F%=1:top%=1
830 WHILE INKEY(47)<>0
840 CLS:PRINT"Press space bar to regain
menu"
850 GOSUB 2330
860 PRINT A$(0,0) TAB(20) A$(F%,0)
870 GOSUB 2330:bottom%=top%+19
880 IF bottom%>L% THEN bottom%=L%
890 FOR R%=top% TO bottom%
900 PRINT A$(0,R%) TAB(20) A$(F%,R%)
910 NEXT
920 CALL &BB03:'KM RESET
930 CALL &BB18:'KM WAIT KEY
940 IF INKEY(8)=0 THEN F%=F%-1
950 IF INKEY(1)=0 THEN F%=F%-1
960 IF INKEY(0)=0 THEN top%=top%-20
970 IF INKEY(2)=0 THEN top%=top%+20
980 IF F%<1 THEN F%=fields%
990 IF F%>fields% THEN F%=1
1000 IF top%<1 THEN top%=(INT(L%/20)*20)
+1
1010 IF top%>L% THEN top%=1
1020 WEND
1030 RETURN
```

This subroutine displays pages of the file in *broadsheet* form (i.e. a page full of records) but, because of the limited screen width, only the key field and one other field can be seen at a time. The key field is always in view but the other fields can be brought in 'sideways' by using the left or right cursor keys. Further pages of records can be rolled into view by means of the up and down cursor keys. The rather mysterious CALL &BB03 is a machine code routine in the operating system which, amongst other things, clears out (resets) the keyboard buffer. The other machine code call to &BB18 is KM WAIT KEY (wait for next key). The letters 'KM' in the appended REMarks refer to the 'Keyboard Manager' and is the group name given by the designers of the system ROM to certain of their operating system routines.

By convention, the headings of the fields – for example NAME,

AGE, TRADE, etc. – are stored in the 'Record ∅' elements of the main array. The key field numbers are ∅ to n, the key field always being in field ∅. This may explain how line 86∅ displays the field headings at the top of the screen. A$(∅,∅) is the key field heading and A$(F%,∅) is the heading of field F%. Lines 89∅ to 91∅ display two fields of data in up to 20 records at a time. Note that the bulk of the subroutine lies within the confines of a WHILE/WEND loop which is testing for a space bar press key (47). The four other INKEY lines which lie further down are testing for a cursor key press.

Subroutine FIND RECORD
Call with GOSUB 267∅.

```
2660 REM FIND RECORD
2670 R%=0
2680 PRINT"Give record entry under ";A$(
F%,0)
2690 GOSUB 2260
2700 R%=R%+1
2710 IF K$=LEFT$(A$(F%,R%),LEN(K$)) THEN
 2740
2720 IF R%<L% THEN 2700
2730 PRINT"No such record on file":GOTO
2670
2740 RETURN
```

A search is conducted through the file for a particular record under one of the field headings so the field number F% must be passed over. If the search is by key field, then F% will be passed as ∅. Assuming that the key field heading is NAME, the first prompt will read

"Give record entry under NAME"

The response from the keyboard is entered into K$, by a call to GET LINE INPUT. Line 271∅ injects a degree of user-friendliness into the keyboard response. The first letter, or first few letters, of the name entered into K$ is sufficient for the search. For example, if the record under name was BREW P, the letters BR might be sufficient to retrieve the record. The search will yield the first record having BR as the first two characters in the name. There is, of course, a chance that BROWN GH may also be on file in which case the search would have to be conducted again, using the first three letters. The end product of the subroutine is the record number left in R%. The search carries on until the record is found or deemed to be not on file.

Subroutine DISPLAY RECORD
Call with GOSUB 2ø4ø.

```
2030 REM DISPLAY RECORD
2040 F%=0:GOSUB 2670:CLS
2050 PRINT:PRINT"Current Record number"R
%:PRINT
2060 GOSUB 2330
2070 FOR F%=0 TO fields%
2080 PRINT A$(F%,0) TAB(21) A$(F%,R%)
2090 NEXT
2100 GOSUB 2330
2110 GOSUB 2520
2120 RETURN
```

The subroutine first calls on FIND RECORD to obtain the record number in R%. The complete record is then displayed, headed by the current record number. The term 'current' is important because the stability of record numbers is not guaranteed. If a record is ever deleted from the file, the record numbers are re-allocated to close up the gap which would otherwise be left. The two calls at the bottom of the subroutine are DRAW LINE and PRESS ANY KEY.

Subroutine FIND FIELD HEADING
Call with GOSUB 257ø.

```
2560 REM FIND FIELD HEADING
2570 F%=-1
2580 PRINT"Operate on which field? (Give
 heading)
2590 GOSUB 2260
2600 F%=F%+1
2610 IF K$=LEFT$(A$(F%,0),LEN(K$)) THEN
2640
2620 IF F%<fields% THEN 2600
2630 PRINT"No such field":GOTO 2570
2640 RETURN
```

Normally, a search through the file is conducted under the key field heading but this subroutine is needed to support searches under any field heading. The response to the prompt will be in K$ after a call to GET LINE INPUT. Line 261ø, which should now be familiar, allows the operator the option of entering the first letter or two of the heading or in full.

Subroutine PRINT FILE
Call with GOSUB 194Ø.

```
1930 REM PRINT FILE
1940 FOR R%=1 TO L%
1950 PRINT#8:PRINT#8,"Record No :"R%
1960 PRINT#8,STRING$(40,"-")
1970 FOR F%=0 TO fields%
1980 PRINT#8,A$(F%,0) TAB(21) A$(F%,R%)
1990 NEXT
2000 NEXT
2010 RETURN
```

Subject to the peculiarities of the printer model in use, this subroutine prints out the entire file as hard copy, including the current record numbers. No provision is made for sending control characters to the printer because they are not standardised but, if needed, they can be entered on a new line immediately below the REM statement.

Subroutine TOTALISE COLUMN
Call with GOSUB 215Ø.

```
2140 REM TOTALISE COLUMN
2150 total=0
2160 GOSUB 2570
2170 FOR R%=1 TO L%
2180 total=total+VAL(A$(F%,R%))
2190 NEXT
2200 PRINT"Column total ="total
2210 PRINT"Column average="total/L%
2220 GOSUB 2520
2230 RETURN
```

The column heading to be totalised is obtained by a call to FIND FIELD HEADING. Both the column total and the column average are printed out. Because the arithmetic is performed using VAL on a string variable, the result will be meaningless if the chosen column contains alpha characters.

Subroutine SORT FILE
Call with GOSUB 149Ø.

```
1480 REM SORT FILE
1490 GOSUB 2570
1500 IF L%<2 THEN 1620
```

```
1510 CLS:PRINT"Sorting by ";A$(F%,0)
1520 N%=L%
1530 N%=(N%+2)\3
1540 FOR D%=N%+1 TO N%*2
1550 FOR E%=D% TO L% STEP N%
1560 FOR R%=E% TO D% STEP -N%
1570 IF A$(F%,R%)<A$(F%,R%-N%) THEN FOR
C%=0 TO fields%:K$=A$(C%,R%):A$(C%,R%)=A
$(C%,R%-N%):A$(C%,R%-N%)=K$:NEXT ELSE 15
90
1580 NEXT
1590 NEXT E%
1600 NEXT
1610 IF N%>1 THEN 1530
1620 RETURN
```

Since the next chapter is devoted entirely to the subject of sorting and searching, no attempt will be made here to explain the subroutine's details apart from noting that the diminishing increment algorithm is used. Considering it is in BASIC, the performance is reasonable but, for long files, an alternative ultra-fast sort in machine code will be given in the next chapter. The heading, under which the sort is conducted, is obtained by a call to GET FIELD HEADING. The complete array A$(F%,R%) is sorted into order, *lowest* record first.

Subroutine ADD RECORDS
Call with GOSUB 1ø6ø.

```
1050 REM ADD RECORDS
1060 CLS: IF mainsize%>=filesize% OR FRE
(0) < fields%*100 THEN PRINT"File full":
GOSUB 2520:GOTO 1240
1070 mainsize%=mainsize%+1
1080 PRINT" Type EXIT to finish entry of
 records"
1090 PRINT:PRINT"Record No :"mainsize%
1100 LOCATE 21,3
1110 PRINT"Bytes Free :"FRE(0)
1120 GOSUB 2330
1130 PRINT:F%=-1
1140 F%=F%+1
```

```
1150 PRINT A$(F%,0)
1160 GOSUB 2260:A$(F%,mainsize%)=K$
1170 X%=POS(#0):Y%=VPOS(#0)
1180 LOCATE 34,3
1190 PRINT USING "#####";FRE(0)
1200 LOCATE X%,Y%
1210 IF A$(F%,mainsize%)="EXIT" THEN mai
nsize%=mainsize%-1:GOTO 1240
1220 IF F%<fields% THEN 1140
1230 IF mainsize%<filesize% THEN 1060
1240 RETURN
```

A file grows by adding more records. Unless checks are built in, the memory can suddenly overflow without warning. There are two mechanisms at work which can lead to this:

(a) The number of records can reach the limit imposed by an existing DIMension statement.
(b) The total number of bytes used can reach the limit imposed by the available RAM complement.

If the amount of text per record is moderate, then the maximum allowed number of records will be the first limiting factor. On the other hand, a moderate number of records, each loaded massively with text, can cause an overflow earlier than expected. This subroutine starts by checking the current number of records already in the file and also the amount of RAM left after allowing for a healthy safety factor. The first line in the subroutine carries out both checks. The amount of safe RAM left has been arbitrarily decided by a constant safety factor equal to 100 × number of fields. If the total free RAM is less than the constant, the message 'File full' is displayed and after a call to PRESS ANY KEY, the rest of the subroutine is skipped by GOTO 124ø.

The records are entered into A$(F%,R%) via a call to GET LINE INPUT. The data is entered, a field at a time, by a loop headed by an increment to F%. If the word EXIT is detected, the subroutine is terminated by GOTO 124ø. While records are being entered, the current record number and the bytes left free are displayed at the top of the screen.

Subroutine CREATE FILE
Call with GOSUB 39ϕ.

```
380 REM CREATE FILE
390 PRINT"Enter file size (number of rec
ords)
400 INPUT filesize%
410 IF filesize%<1 THEN 400
420 PRINT"Enter number of fields require
d (2-15)"
430 INPUT fields%
440 IF fields%<2 OR fields%>15 THEN 420

450 fields%=fields%-1
460 flag%=1
470 DIM A$(fields%,filesize%)
480 CLS
490 FOR F%=0 TO fields%
500 PRINT"Enter field heading ";F%+1
510 GOSUB 2260:A$(F%,0)=K$
520 NEXT
530 GOSUB 1060
540 RETURN
```

This subroutine takes care of the preliminary work necessary to start up a new file. The first two variables 'filesize%' and 'fields%' are obtained directly by asking for an estimate of the maximum number of records the file will eventually hold, and the number of fields respectively. This information is fed to the DIMension line and a flag is set to 1, indicating resident file status. The headings for each field are then obtained via a call to GET LINE INPUT and assigned to the Record ϕ position in the array. The first records in the newly created file are then entered by a call to ADD RECORDS.

Subroutine MODIFY/DELETE RECORD
Call with GOSUB 127ϕ.

```
1260 REM MODIFY/DELETE RECORD
1270 F%=0:GOSUB 2670:K$=""
1280 WHILE K$<>"EXIT" AND K$<>"KILL"
1290 CLS:PRINT"Mod/copy line with CURSOR
/COPY keys"
1300 PRINT"Type KILL to delete record"
1310 PRINT"Type EXIT to regain option pa
ge"
```

```
1320 GOSUB 2330
1330 FOR F%=0 TO fields%
1340 PRINT A$(F%,0) TAB(21) A$(F%,R%)
1350 NEXT
1360 GOSUB 2330:PRINT:GOSUB 2330
1370 LOCATE 1,VPOS(#0)-2
1380 LINE INPUT K$
1390 K$=UPPER$(K$)
1400 IF LEN(K$)>38 THEN K$=LEFT$(K$,38)
1410 FOR F%=0 TO fields%
1420 IF A$(F%,0)=LEFT$(K$,LEN(A$(F%,0)))
 THEN A$(F%,R%)=RIGHT$(K$,LEN(K$)-20)
1430 NEXT
1440 WEND
1450 IF K$="KILL" THEN WHILE R%<=mainsiz
e%:FOR F%=0 TO fields%:A$(F%,R%)=A$(F%,R
%+1):NEXT:R%=R%+1:WEND:mainsize%=mainsiz
e%-1:PRINT FRE("")
1460 RETURN
```

To modify a record, it must first be found from within the file so the first call is to FIND RECORD. The bulk of the subroutine is within a WHILE/WEND loop which checks if the operator enters 'KILL' or 'EXIT'. Lines 133Ø to 135Ø display the record to be amended. The next two calls to DRAW LINE reserve a space on the screen for entering the amended characters which, with the help of VPOS in line 137Ø, appear in the space between the two lines. If KILL is entered by the operator, line 145Ø erases the record and renumbers those which follow in order to close the gap. For more details on the specific use of this subroutine, see Chapter 4.

Subroutine TITLE
Call with GOSUB 326Ø.

```
3250 REM TITLE
3260 CLS
3270 LOCATE 1,2
3280 INK 2,6
3290 PAPER 2
3300 PRINT" MULTIFILING SYSTEM "
3310 PAPER 0
3320 RETURN
```

This, like the subroutine for drawing a line, is trivial but it saves entering a title line at the top of the screen. It is a simple matter to change the border, ink and pen values if the author's choice of colour is felt to be lacking in taste.

Subroutine SEARCH FILE MENU
Call with GOSUB 165ø.

```
1640 REM SEARCH FILE MENU
1650 SEL%=0:GOSUB 3260
1660 LOCATE 1,5
1670 WHILE SEL%<1 OR SEL%>12
1680 PRINT"SHRINK file (Non destructive)
"
1690 PRINT"SEARCH any field for:":PRINT
1700 PRINT"(1)  <> a"
1710 PRINT"(2)   = a"
1720 PRINT"(3)  >= a"
1730 PRINT"(4)  <  a"
1740 PRINT"(5)  Character group"
1750 PRINT:PRINT"Futher subfile options:
":PRINT
1760 PRINT"(6)  Save sub-file"
1770 PRINT"(7)  Display sub-file"
1780 PRINT"(8)  Sort sub-file by any fie
ld"
1790 PRINT"(9)  Delete sub-file from mai
n file"
1800 PRINT"(10) Totalise column"
1810 PRINT"(11) Print sub-file"
1820 PRINT"(12) Return to main menu"
1830 LOCATE 1,24
1840 INPUT"Select option ";K$
1850 SEL%=VAL(K$)
1860 CLS
1870 WEND
1880 CLS:L%=subsize%
1890 ON SEL% GOSUB 2770,2770,2770,2770,2
770,700,820,1490,3170,2150,1940,1910
1900 IF SEL%<12 THEN 1650
1910 RETURN
```

This is a subsidiary menu, which can be called as one of the options given in a main menu. The first call is to TITLE for heading the menu page. Many of the GOSUB numbers called in line 189ø correspond to subroutines already described. Options 1 to 4 allow a main file to be shrunk down to a subfile containing only records which satisfy certain criteria defined by 'a'. Note that selection of any one of these four causes a call to the same GOSUB number, 277ø. This is the subroutine SEARCH DIRECTOR to be described next. Option 5 assumes that the criterion for subfile inclusion is the presence of a particular character or group of characters.

Subroutine SEARCH DIRECTOR
Call with GOSUB 277ø.

```
2760 REM SEARCH DIRECTOR
2770 GOSUB 2570
2780 PRINT"SEARCH to operate on ";A$(F%,
0)
2790 IF SEL%<5 THEN PRINT"Give search da
tum/entity"
2800 IF SEL%=5 THEN PRINT"Give character
 group"
2810 GOSUB 2260:comp$=K$
2820 count%=0
2830 FOR R%=1 TO subsize%
2840 ON SEL% GOSUB 2900,2940,2980,3020,3
060
2850 NEXT
2860 IF count%=0 THEN CLS:PRINT"SEARCH i
s NEGATIVE":GOSUB 2520 ELSE subsize%=cou
nt%
2870 RETURN
2880 '
2890 REM SEARCH <>
2900 IF A$(F%,R%)<>comp$ THEN GOSUB 3100
2910 RETURN
2920 '
2930 REM SEARCH =
2940 IF A$(F%,R%)=comp$ THEN GOSUB 3100
2950 RETURN
2960 '
2970 REM SEARCH >=
2980 IF A$(F%,R%)>=comp$ THEN GOSUB 3100
```

```
2990 RETURN
3000 '
3010 REM SEARCH <
3020 IF A$(F%,R%)<comp$ THEN GOSUB 3100
3030 RETURN
3040 '
3050 REM SEARCH FOR CHARACTER GROUP
3060 IF INSTR(A$(F%,R%),comp$) <>0 THEN
GOSUB 3100
3070 RETURN
3080 ''
3090 REM MOVE RECORD
3100 count%=count%+1
3110 FOR C%=0 TO fields%
3120 K$=A$(C%,R%):A$(C%,R%)=A$(C%,count%
):A$(C%,count%)=K$
3130 NEXT
3140 RETURN
```

To create a subfile, the operator must enter the *field heading* of interest so the first call is to FIND FIELD HEADING. Prompts are issued for the value of 'a', or a character group if SEL%= 5. (SEL% is a parameter passed from the subfile menu.) The response is assigned to comp$ via a call to GET LINE INPUT. The variable comp$ is so named because it contains the parameter which will be used for comparison with each record in the main file. A separate subroutine for each of the comparisons is called from within the FOR/NEXT loop in lines 283Ø to 285Ø. Although the loop extends from R%= 1 to subsize%, it covers the whole of the main file on the first call because subsize% will have been previously assigned to mainsize%. All records which match the entered data in comp$ are brought out to the *front of the main file* with the aid of subroutine MOVE RECORD at line 31ØØ. Because the subfile is the collection of records now at the front of the main file, no additional RAM space is occupied. The variable, subsize%, holds the number of subfile records.

Subroutine DELETE SUBFILE

```
3160 REM DELETE SUB FILE
3170 IF subsize%=mainsize% THEN 3220
3180 FOR R%=subsize%+1 TO mainsize%
3190 FOR F%=0 TO fields%
```

```
3200 A$(F%,R%-subsize%)=A$(F%,R%)
3210 NEXT:NEXT
3220 mainsize%=mainsize%-subsize%:SEL%=1
2
3230 RETURN
```

This subroutine is called only if the subfile is to be deleted from the main file. Line 318Ø contains the clue to its operation because the FOR/NEXT loop starting parameter is subsize%+1. In other words, the mainfile now starts at the first record number higher than the end of the subfile. (You will remember that the records, collected into the subfile, were all repositioned at the head of the main file.)

Initialisation and main menu

```
10 REM INITIALISATION AND MAIN MENU
20 OPENOUT "buffer"
30 MEMORY &9B7E:REM SET TO HIMEM-1
40 CLOSEOUT
50 MODE 1
60 BORDER 0
70 mainsize%=0:flag%=0
80 SEL%=0
90 WHILE SEL%<1 OR SEL%>12
100 GOSUB 3260
110 LOCATE 5,5
120 PRINT"MAIN FILE MENU"
130 LOCATE 1,7
140 PRINT"(1)  Create new file"
150 PRINT"(2)  Load file"
160 PRINT"(3)  Save file"
170 PRINT"(4)  Display file"
180 PRINT"(5)  Add records"
190 PRINT"(6)  Modify any record"
200 PRINT"(7)  Sort by any field"
210 PRINT"(8)  Create sub-file (Search a
ny field)"
220 PRINT"(9)  Print File"
230 PRINT"(10) Display single record"
240 PRINT"(11) Totalise any column"
250 PRINT"(12) End program"
260 LOCATE 1,23
```

```
270 INPUT"Select option ";K$
280 SEL%=VAL(K$)
290 WEND
300 CLS:L%=mainsize%:subsize%=mainsize%
310 IF SEL%>2 AND mainsize%=0 AND SEL%<>
12 THEN PRINT"No file present":GOSUB 252
0:GOTO 70
320 IF SEL%<3 AND flag%=1 THEN 2440
330 ON SEL% GOSUB 390,570,700,820,1060,1
270,1490,1650,1940,2040,2150,350
340 GOTO 80
350 SPEED WRITE 0
360 END
```

This is a control program, not a subroutine. It has three sections:

(1) Lines 1Ø to 8Ø reset initial states, define a *permanent* area in memory for the cassette buffer and establish the mode and border colour.
(2) Lines 9Ø to 29Ø are occupied by a WHILE/WEND loop containing the main menu and the prompt for the operator to select an option number into SEL%. The title at the top of the menu display comes via a call to subroutine TITLE.
(3) Lines 3ØØ to 36Ø reset L% (the current length of the file) and subsize% (the size of the subfile). The validity of the operator's selection is then tested followed by a call to the appropriate subroutine. The default baud rate is finally established in case a call to the SAVE FILE subroutine leaves it at 2ØØØ baud.

Table of subroutine numbers

The subroutines offered as general-purpose building bricks in this chapter are also components of a complete filing program. In fact, the line numbers of the subroutines will be found to correspond exactly with the listing of Program 4.1 in the next chapter. Because the analysis of each subroutine has been described in reasonable depth here, Chapter 4 will concentrate on the instructions for *using* the program. For reference purposes, the subroutine line numbers are set out below. It is worth mentioning at this point that GOTOs and GOSUBs to lines with REM statements is bad practice. The reason is that program *compactors* remove REM lines altogether and fail to make allowances for GOTO and GOSUB. It also makes

manual REM removal easier for a working copy, where more RAM space is required.

Table 3.1. Table of GOSUB numbers (the first REM lines are ignored).

Line number	GOSUB title
233Ø	DRAW LINE
226Ø	GET LINE INPUT
252Ø	PRESS ANY KEY
237Ø	GET FILENAME
7ØØ	SAVE FILE
57Ø	LOAD FILE
82Ø	DISPLAY FILE
267Ø	FIND RECORD
2Ø4Ø	DISPLAY RECORD
257Ø	FIND FIELD HEADING
194Ø	PRINT FILE
215Ø	TOTALISE COLUMN
149Ø	SORT FILE
1Ø6Ø	ADD RECORS
39Ø	CREATE FILE
127Ø	MODIFY/DELETE RECORD
326Ø	TITLE
165Ø	SEARCH FILE MENU
277Ø	SEARCH FILE DIRECTOR
317Ø	DELETE SUBFILE

Summary

1. It is bad practice to jump out of a subroutine before the normal RETURN.
2. A 'program-based' file has records within DATA statements.
3. A$(F%,R%) is a main file array element, or field in our case.
4. The field information index, F%, is written first to comply with the order of string descriptor bytes stored in memory by the interpreter.
5. In Program 3.2, A$(Ø,R%) is the key field of record R%.
6. Data files are saved to tape under a different format altogether from program files.
7. The cassette tape drive is defined as stream #9.

8. The command SPEEDWRITE 1 is used if data is to be recorded at 2000 baud.
9. A file should be properly closed by the keyword CLOSEOUT or CLOSEIN as appropriate.
10. Failure to close a file for output could mean that residual data in the buffer would not go on file.
11. K$ is a temporary variable.
12. The subroutine GET LINE INPUT is used for inputting string variables. It rejects the null string and truncates all input to 18 characters.
13. Record numbers which follow a deleted record are all moved up 1.
14. When records are selected to form a subfile, they are repositioned at the front of the mainfile.
15. The GOSUB numbers in the building bricks are arranged to bypass the leading REMs.

Self test

3.1 If five more records are added to the program, what other line must be altered?
3.2 What is the purpose of flag% in Program 3.2?
3.3 In Program 3.2, what is the purpose of line 31Ø?
3.4 What is the reason behind line 15ØØ in the subroutine SORT FILE?

Chapter Four

A Complete RAM-based Serial Filing System

The program listing

This chapter is devoted entirely to the operation and description of a complete RAM-based filing program.

```
10 REM CASSETTE MULTIFILING SYSTEM
20 OPENOUT "buffer"
30 MEMORY &9B7E:REM SET TO HIMEM-1
40 CLOSEOUT
50 MODE 1
60 BORDER 0
70 mainsize%=0:flag%=0
80 SEL%=0
90 WHILE SEL%<1 OR SEL%>12
100 GOSUB 3260
110 LOCATE 5,5
120 PRINT"MAIN FILE MENU"
130 LOCATE 1,7
140 PRINT"(1)  Create new file"
150 PRINT"(2)  Load file"
160 PRINT"(3)  Save file"
170 PRINT"(4)  Display file"
180 PRINT"(5)  Add records"
190 PRINT"(6)  Modify any record"
200 PRINT"(7)  Sort by any field"
210 PRINT"(8)  Create sub-file (Search a
ny field)"
220 PRINT"(9)  Print File"
230 PRINT"(10) Display single record"
240 PRINT"(11) Totalise any column"
250 PRINT"(12) End program"
260 LOCATE 1,23
```

```
270 INPUT"Select option ";K$
280 SEL%=VAL(K$)
290 WEND
300 CLS:L%=mainsize%:subsize%=mainsize%
310 IF SEL%>2 AND mainsize%=0 AND SEL%<>
12 THEN PRINT"No file present":GOSUB 252
0:GOTO 70
320 IF SEL%<3 AND flag%=1 THEN 2440
330 ON SEL% GOSUB 390,570,700,820,1060,1
270,1490,1650,1940,2040,2150,350
340 GOTO 80
350 SPEED WRITE 0
360 END
370 '
380 REM CREATE FILE
390 PRINT"Enter file size (number of rec
ords)
400 INPUT filesize%
410 IF filesize%<1 THEN 400
420 PRINT"Enter number of fields require
d (2-15)"
430 INPUT fields%
440 IF fields%<2 OR fields%>15 THEN 420

450 fields%=fields%-1
460 flag%=1
470 DIM A$(fields%,filesize%)
480 CLS
490 FOR F%=0 TO fields%
500 PRINT"Enter field heading ";F%+1
510 GOSUB 2260:A$(F%,0)=K$
520 NEXT
530 GOSUB 1060
540 RETURN
550 '
560 REM LOAD FILE
570 GOSUB 2370
580 OPENIN filename$
590 INPUT#9,filesize%,fields%,mainsize%
600 flag%=1
610 DIM A$(fields%,filesize%)
620 FOR R%=0 TO mainsize%
630 FOR F%=0 TO fields%
```

```
640 INPUT#9,A$(F%,R%)
650 NEXT:NEXT
660 CLOSEIN
670 RETURN
680 '
690 REM SAVE FILE
700 GOSUB 2370
710 SPEED WRITE 1
720 OPENOUT filename$
730 PRINT#9,filesize%,fields%,L%
740 FOR R%=0 TO L%
750 FOR F%=0 TO fields%
760 PRINT#9,A$(F%,R%)
770 NEXT:NEXT
780 CLOSEOUT
790 RETURN
800 '
810 REM DISPLAY FILE
820 F%=1:top%=1
830 WHILE INKEY(47)<>0
840 CLS:PRINT"Press space bar to regain
menu"
850 GOSUB 2330
860 PRINT A$(0,0) TAB(20) A$(F%,0)
870 GOSUB 2330:bottom%=top%+19
880 IF bottom%>L% THEN bottom%=L%
890 FOR R%=top% TO bottom%
900 PRINT A$(0,R%) TAB(20) A$(F%,R%)
910 NEXT
920 CALL &BB03:'KM RESET
930 CALL &BB18:'KM WAIT KEY
940 IF INKEY(8)=0 THEN F%=F%-1
950 IF INKEY(1)=0 THEN F%=F%-1
960 IF INKEY(0)=0 THEN top%=top%-20
970 IF INKEY(2)=0 THEN top%=top%+20
980 IF F%<1 THEN F%=fields%
990 IF F%>fields% THEN F%=1
1000 IF top%<1 THEN top%=(INT(L%/20)*20)
+1
1010 IF top%>L% THEN top%=1
1020 WEND
1030 RETURN
1040 '
```

```
1050 REM ADD RECORDS
1060 CLS: IF mainsize%>=filesize% OR FRE
(0) < fields%*100 THEN PRINT"File full":
GOSUB 2520:GOTO 1240
1070 mainsize%=mainsize%+1
1080 PRINT" Type EXIT to finish entry of
 records"
1090 PRINT:PRINT"Record No :"mainsize%
1100 LOCATE 21,3
1110 PRINT"Bytes Free :"FRE(0)
1120 GOSUB 2330
1130 PRINT:F%=-1
1140 F%=F%+1
1150 PRINT A$(F%,0)
1160 GOSUB 2260:A$(F%,mainsize%)=K$
1170 X%=POS(#0):Y%=VPOS(#0)
1180 LOCATE 34,3
1190 PRINT USING "#####";FRE(0)
1200 LOCATE X%,Y%
1210 IF A$(F%,mainsize%)="EXIT" THEN mai
nsize%=mainsize%-1:GOTO 1240
1220 IF F%<fields% THEN 1140
1230 IF mainsize%<filesize% THEN 1060
1240 RETURN
1250 '
1260 REM MODIFY/DELETE RECORD
1270 F%=0:GOSUB 2670:K$=""
1280 WHILE K$<>"EXIT" AND K$<>"KILL"
1290 CLS:PRINT"Mod/copy line with CURSOR
/COPY keys"
1300 PRINT"Type KILL to delete record"
1310 PRINT"Type EXIT to regain option pa
ge"
1320 GOSUB 2330
1330 FOR F%=0 TO fields%
1340 PRINT A$(F%,0) TAB(21) A$(F%,R%)
1350 NEXT
1360 GOSUB 2330:PRINT:GOSUB 2330
1370 LOCATE 1,VPOS(#0)-2
1380 LINE INPUT K$
1390 K$=UPPER$(K$)
1400 IF LEN(K$)>38 THEN K$=LEFT$(K$,38)
1410 FOR F%=0 TO fields%
```

```
1420 IF A$(F%,0)=LEFT$(K$,LEN(A$(F%,0)))
 THEN A$(F%,R%)=RIGHT$(K$,LEN(K$)-20)
1430 NEXT
1440 WEND
1450 IF K$="KILL" THEN WHILE R%<=mainsiz
e%:FOR F%=0 TO fields%:A$(F%,R%)=A$(F%,R
%+1):NEXT:R%=R%+1:WEND:mainsize%=mainsiz
e%-1:PRINT FRE("")
1460 RETURN
1470 '
1480 REM SORT FILE
1490 GOSUB 2570
1500 IF L%<2 THEN 1620
1510 CLS:PRINT"Sorting by ";A$(F%,0)
1520 N%=L%
1530 N%=(N%+2)\3
1540 FOR D%=N%+1 TO N%*2
1550 FOR E%=D% TO L% STEP N%
1560 FOR R%=E% TO D% STEP -N%
1570 IF A$(F%,R%)<A$(F%,R%-N%) THEN FOR
C%=0 TO fields%:K$=A$(C%,R%):A$(C%,R%)=A
$(C%,R%-N%):A$(C%,R%-N%)=K$:NEXT ELSE 15
90
1580 NEXT
1590 NEXT E%
1600 NEXT
1610 IF N%>1 THEN 1530
1620 RETURN
1630 '
1640 REM SEARCH FILE MENU
1650 SEL%=0:GOSUB 3260
1660 LOCATE 1,5
1670 WHILE SEL%<1 OR SEL%>12
1680 PRINT"SHRINK file (Non destructive)
"
1690 PRINT"SEARCH any field for:":PRINT
1700 PRINT"(1)   <> a"
1710 PRINT"(2)    = a"
1720 PRINT"(3)   >= a"
1730 PRINT"(4)   <  a"
1740 PRINT"(5)  Character group"
1750 PRINT:PRINT"Futher subfile options:
":PRINT
```

```
1760 PRINT"(6)  Save sub-file"
1770 PRINT"(7)  Display sub-file"
1780 PRINT"(8)  Sort sub-file by any fie
ld"
1790 PRINT"(9)  Delete sub-file from mai
n file"
1800 PRINT"(10) Totalise column"
1810 PRINT"(11) Print sub-file"
1820 PRINT"(12) Return to main menu"
1830 LOCATE 1,24
1840 INPUT"Select option ";K$
1850 SEL%=VAL(K$)
1860 CLS
1870 WEND
1880 CLS:L%=subsize%
1890 ON SEL% GOSUB 2770,2770,2770,2770,2
770,700,820,1490,3170,2150,1940,1910
1900 IF SEL%<12 THEN 1650
1910 RETURN
1920 '
1930 REM PRINT FILE
1940 FOR R%=1 TO L%
1950 PRINT#8:PRINT#8,"Record No :"R%
1960 PRINT#8,STRING$(40,"-")
1970 FOR F%=0 TO fields%
1980 PRINT#8,A$(F%,0) TAB(21) A$(F%,R%)
1990 NEXT
2000 NEXT
2010 RETURN
2020 '
2030 REM DISPLAY RECORD
2040 F%=0:GOSUB 2670:CLS
2050 PRINT:PRINT"Current Record number"R
%:PRINT
2060 GOSUB 2330
2070 FOR F%=0 TO fields%
2080 PRINT A$(F%,0) TAB(21) A$(F%,R%)
2090 NEXT
2100 GOSUB 2330
2110 GOSUB 2520
2120 RETURN
2130 '
2140 REM TOTALISE COLUMN
```

```
2150 total=0
2160 GOSUB 2570
2170 FOR R%=1 TO L%
2180 total=total+VAL(A$(F%,R%))
2190 NEXT
2200 PRINT"Column total ="total
2210 PRINT"Column average="total/L%
2220 GOSUB 2520
2230 RETURN
2240 '
2250 REM GET LINE INPUT
2260 LINE INPUT K$
2270 IF K$="" THEN 2260
2280 K$=UPPER$(K$)
2290 IF LEN(K$)>18 THEN K$=LEFT$(K$,18)
2300 RETURN
2310 '
2320 REM DRAW LINE
2330 PRINT STRING$(40,CHR$(154));
2340 RETURN
2350 '
2360 REM GET FILENAME
2370 CLS:PRINT"Enter filename"
2380 GOSUB 2260
2390 IF LEN(K$)>16 THEN PRINT"Too long:E
nter again":GOTO 2380
2400 filename$=K$
2410 RETURN
2420 '
2430 REM BELT & BRACES
2440 PRINT"WARNING":SOUND 1,16,10,15
2450 PRINT"This selection destroys EXIST
ING FILE"
2460 PRINT"Press ANY KEY to CANCEL selec
tion"
2470 PRINT"OR press CTRL M keys and RESE
LECT"
2480 CALL &BB18:'KM WAIT KEY
2490 IF INKEY(38)=128 THEN ERASE A$:GOTO
 70 ELSE 80
2500 '
2510 REM PRESS ANY KEY
2520 PRINT:PRINT"Press any key to contin
ue"
```

```
2530 K$=INKEY$:IF K$="" THEN 2530
2540 RETURN
2550 '
2560 REM FIND FIELD HEADING
2570 F%=-1
2580 PRINT"Operate on which field? (Give
 heading)
2590 GOSUB 2260
2600 F%=F%+1
2610 IF K$=LEFT$(A$(F%,0),LEN(K$)) THEN
2640
2620 IF F%<fields% THEN 2600
2630 PRINT"No such field":GOTO 2570
2640 RETURN
2650 '
2660 REM FIND RECORD
2670 R%=0
2680 PRINT"Give record entry under ";A$(
F%,0)
2690 GOSUB 2260
2700 R%=R%+1
2710 IF K$=LEFT$(A$(F%,R%),LEN(K$)) THEN
 2740
2720 IF R%<L% THEN 2700
2730 PRINT"No such record on file":GOTO
2670
2740 RETURN
2750 '
2760 REM SEARCH DIRECTOR
2770 GOSUB 2570
2780 PRINT"SEARCH to operate on ";A$(F%,
0)
2790 IF SEL%<5 THEN PRINT"Give search da
tum/entity"
2800 IF SEL%=5 THEN PRINT"Give character
 group"
2810 GOSUB 2260:comp$=K$
2820 count%=0
2830 FOR R%=1 TO subsize%
2840 ON SEL% GOSUB 2900,2940,2980,3020,3
060
2850 NEXT
2860 IF count%=0 THEN CLS:PRINT"SEARCH i
```

```
s NEGATIVE":GOSUB 2520 ELSE subsize%=cou
nt%
2870 RETURN
2880 '
2890 REM SEARCH <>
2900 IF A$(F%,R%)<>comp$ THEN GOSUB 3100
2910 RETURN
2920 '
2930 REM SEARCH =
2940 IF A$(F%,R%)=comp$ THEN GOSUB 3100
2950 RETURN
2960 '
2970 REM SEARCH >=
2980 IF A$(F%,R%)>=comp$ THEN GOSUB 3100

2990 RETURN
3000 '
3010 REM SEARCH <
3020 IF A$(F%,R%)<comp$ THEN GOSUB 3100
3030 RETURN
3040 '
3050 REM SEARCH FOR CHARACTER GROUP
3060 IF INSTR(A$(F%,R%),comp$) <>0 THEN
GOSUB 3100
3070 RETURN
3080 '
3090 REM MOVE RECORD
3100 count%=count%+1
3110 FOR C%=0 TO fields%
3120 K$=A$(C%,R%):A$(C%,R%)=A$(C%,count%
):A$(C%,count%)=K$
3130 NEXT
3140 RETURN
3150 '
3160 REM DELETE SUB FILE
3170 IF subsize%=mainsize% THEN 3220
3180 FOR R%=subsize%+1 TO mainsize%
3190 FOR F%=0 TO fields%
3200 A$(F%,R%-subsize%)=A$(F%,R%)
3210 NEXT:NEXT
3220 mainsize%=mainsize%-subsize%:SEL%=1
2
3230 RETURN
```

```
3240 '
3250 REM TITLE
3260 CLS
3270 LOCATE 1,2
3280 INK 2,6
3290 PAPER 2
3300 PRINT" MULTIFILING SYSTEM "
3310 PAPER 0
3320 RETURN
```

Program 4.1. RAM-based multifile system.

The program, as listed, is written for cassette tape files but it would be easy to modify the load and save subroutines for a disk system. 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 the manner we intended. After you have used it for some time, you may think certain features could be improved or altered to suit your specific filing needs. You may not even 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 as a 'fall back' precaution. Strange things can happen if attempts are made to 'simplify' programs.

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, start at line 1Ø and carry on until the last line. This method must work – eventually!
(b) Notice that the 'program' itself, terminating with END, occupies about the first 40 lines only; the remainder of the listing is a collection of subroutines. 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 satisfactorily up to the point which you have reached. However, there is a world of difference between a program which appears to work satisfactorily and one which has been subjected to exhaustive tests. These should be delayed until you have explored the complete program.

Using the program

Programming details are given at the end of this chapter for the benefit of those who may wish to introduce modifications or add additional facilities. However, it would be wise to spend some time operating the program as it stands. On RUNning the program, the 12-option menu is displayed. In the first instance, there is no file present in RAM so there are only two choices open on the menu – Option 1 'Create file' or Option 2 'Load file'. If any other option is selected, the message 'No file present' is displayed. The menu is regained by responding to the prompt 'Press any key to continue'. Operating details which follow are treated in the order in which they are most likely to be used rather than in option number sequence.

Create file (Option 1)

When starting up a new file, a number of leading particulars must be obtained from the keyboard in order to fix the dimensions of the main rectangular array. These will include the number of records and the number of fields. The literal headings of each field are also required.

File size

The first screen prompt is 'Enter file size (number of records)'. It pays not to be too ambitious when entering this number because the amount of RAM available, over and above that occupied by the program and cassette buffers, is about 32K. We shall see later that as records are entered into the file, the screen continuously displays the file *status*, including the number of bytes still left in RAM. During subsequent usage, the number of records added to the file can grow to the point where the record number limit is approached or the number of bytes left is looking dangerously low. At this point, it is possible to split the file into two by making use of Option 8, 'Create subfile'.

Number of fields

The next screen prompt is 'Enter number of fields required (2–15)'. The maximum number is fixed at 15 and the minimum at 2. A trap is included to prevent the limit from being exceeded, although it would be easy to modify the upper limit. Remember that the RAM space occupied by the file depends on the product of the number of records, the number of fields in each record and the amount of text in each field.

Field headings

Each field must have a heading, otherwise the data within the field is meaningless. A different heading is required for each field so the screen prompt for, say, field 3 is 'Enter field heading 3'. No heading must exceed 18 characters in length. If more than 18 are entered, the program ignores any excess characters without notification. The limit of 18 is enforced because of the requirements of Option 4 which allows a page of records to be viewed at a time in 'broadsheet' form. The limit of 18 characters per field in Mode 1 allows only two of the fields to be displayed horizontally at one time. (The remaining fields are rotated into view by cursor controls.) It doesn't matter if field headings are entered in lower- or upper-case because the program always converts to upper-case anyway.

Add records (Option 5)

The previous entries were leading particulars only and will normally remain unchanged during the life of the file. Once these have been entered, the display changes automatically to Option 5 'Add records', ready for receiving data for each field. As an aid to subsequent explanations, assume that the file has three fields with the headings Name, Telephone and Age. The prompts would be:

NAME - (you might answer BREW P)
TELEPHONE - (you might answer 2ø8 4672)
AGE - (you might answer 36)

Don't forget that, as with field headings, the data items must not exceed 18 characters. This is not a severe restriction providing the file has been created with the restriction in mind. For example, it would have been bad planning to label one field 'Address' because it is unlikely that a full address could be squeezed into 18 characters. One field should have been 'Number/street', the next 'Town', the next

'County' and the last 'Post code'. Since 15 fields are allowed per record, this still leaves 11 fields for other data items.

When the last data item has been entered, the screen clears ready for the next record to be entered. If no further records are to be added during the current keyboard session, the menu can be regained by keying 'EXIT'. If the number of records reaches the maximum estimated during the creation stage, or if the bytes free are exhausted, the message 'File full' will appear and no further records will be accepted.

If a file already exists in RAM, selecting Option 1 would destroy it so the program issues a warning in case the option has been selected in error, the details of which are given later under the 'Load File' option.

Display file (Option 4)

After the file has just been created or if additional records have just been entered, the next logical step is to see what it looks like on the screen. Assuming that the file has only three records, each giving Name, Telephone and Age, the screen display would appear something like this:

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 left or right cursor keys. For example, if we pressed the right cursor key, the display above might change to:

Press space bar for menu	
NAME	AGE
BLENKINSOP J	24
GUTSWORTHY M	83
GLUGENHEIMER K	18

If more than 20 records are in the file, other blocks of 20 can be moved into view, using the up or down cursor keys. If the display is moved beyond the boundaries of the fields or records, wrap-around occurs. That is to say, if there were seven fields and we tried to shift in a non-existent eighth field, the display would wrap around to the second field again. A similar thing happens if we try to bring in a non-existent block of 20 records, except that it would be better described as roll-around instead of wrap-around.

Display single record (Option 10)

The display file option is useful for gaining a general picture or for observing how values of a particular field vary between records. The eye can run down from top to bottom of a field and easily spot abnormal or significant entries. Option 10, on the other hand, searches through the file for one particular record. It displays the complete record with all fields without the need for cursor key action.

Taking the same three example records shown before, and supposing we ask for the one on GUTSWORTHY M, the search would end with the following display:

Current record number 2	
NAME	GUTSWORTHY M
TELEPHONE	233 6895
AGE	83
Press any key to continue	

The search requires the key field of the record to be entered. In the example case, the prompt would read:

'Enter entry under NAME'

Your response would be GUTSWORTHY M. However, the program is sufficiently user-friendly to realise that you may not remember the exact spelling of each name. The program will conduct the search if you enter the first few letters correctly or even if you enter only the first letter. For example, entering GUT, GU or even G will initiate the search action. The search ends when the first record is found having a key field beginning with the same few letters you have entered. However, the fewer letters you enter, the greater will be the chance of bringing out an unwanted record which, by coincidence, begins with the same letters. If this happens, you can add another letter – if you know it – and start another search. If all this fails, the last resort would be to go over to the Display File option, skim through the key fields until the full name is found and then return to Option 10 for a new search. We appreciate that as far as our three record example goes, such detailed advice might appear absurd but, when a file approaches one hundred records, the advice may be appreciated. If a search through the file fails to find a record which corresponds to given search letters, the message 'No such record on file' is displayed.

Save file (Option 3)

Once the file has been displayed, you will most probably wish to save a copy on tape. The first prompt 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: Enter again' appears. The usual instructions for operating the tape transport will appear after an acceptable file name is received. Although the listing for the subroutine is written for cassette tape it would be easy to modify for disk drives.

Load file (Option 2)

Because the loading of a file will automatically overwrite and therefore destroy any existing file, precautions have been taken by the program to safeguard an existing file should this option be

selected by mistake. If a file already exists, the following warning, accompanied by an audible tone, appears on selecting Option 2 (or Option 1):

This selection destroys existing file
Press any key to cancel
Otherwise press CTRL and M keys together and re-select

Summarising, if Option 2 is selected by mistake, pressing any key will restore the menu without harm to the existing file. If the selection was intentional, the CTRL and M keys must be pressed together to regain the menu before re-selecting Option 2 again. This time, there will be no further warning and the prompt 'Enter file name' appears on the screen. Providing the cassette file is ready in the machine, the loading process can begin. Modifying the subroutine to suit disk drives should again be easy.

Modify any record (Option 6)

This option allows you to modify any field of a selected record. It also allows the record to be deleted entirely from the file. The record to be modified is first searched for in response to the prompt, 'Give record under NAME' – assuming, of course, that the key field is NAME. As we described earlier, the first letter or two of the key field is sufficient to initiate the search. Once the record has been found, the following display appears:

Modify/copy line with cursor/COPY keys Type KILL to delete record Type EXIT to regain Option page
NAME GUTSWORTHY M TELEPHONE 233 6895 AGE 82
X

The cursor will initially rest at the point marked X. Suppose the telephone number is incorrect. The copy cursor is first moved upwards, by pressing SHIFT and 'up arrow' cursor keys, to the letter T at the start of the second line. The SHIFT and COPY keys are then used to move across the screen until the offending figure is reached. It can then be corrected in the normal manner. The corrected version is also duplicated on the bottom line during the editing process. After you press the ENTER key, the corrected version is displayed. When finally satisfied, enter EXIT to regain the menu options.

To delete a record from the file, type KILL when the desired record is displayed. The gap in the array is closed up and garbage collection is forced.

Create subfile (Search any field): Option 8

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 one or more common attributes 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'. At this point, it is worth digressing a little to clarify the distinction between the two.

The terms 'data file' and 'database' are often used as if they were one and the same. This is not strictly true. Indeed, there are profound differences between a data file and a database. To avoid any possible confusion, we should 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 miles with a population *greater than or equal to* 500000. The task begins by selecting Option 8 which brings out the following search menu:

```
SHRINK FILE (Non-destructive)

Search any field for:

(1)     <>a
(2)     =a
(3)     >=a
(4)     <a
(5)     Character group

        Further subfile options:

(6)     Save subfile
(7)     Display subfile
(8)     Sort subfile by any field
(9)     Delete subfile from main file
(10)    Totalise column
(11)    Print subfile
(12)    Return to main menu

        Select option
```

The steps from then on could vary but the following approach is reasonable:

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

    Operate on which field?

The response should be AREA.

(2) The next prompt will be:

    Search to operate on AREA
    Give search datum/entity

The response should be 40000.

On selecting option 7 (Display subfile), 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:

Operate on which field?

The response should be Population.

(4) The next prompt will be:

Search to operate on Population
Give search datum/entity

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 using Option 5 (character group) and insisting that the country must contain the substring 'er'. According to our own file on EUROPE (compiled with the help of *Whitaker's Almanac*), the final shrinkage pointed to one country only Switzerland. The example is illustrated in Figure 4.1.

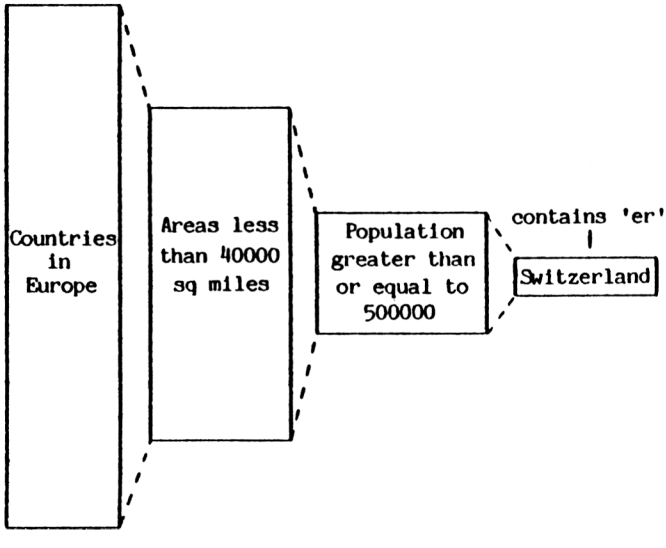


Fig. 4.1. Shrinking a main file on European countries.

The second half of the search menu offers several straightforward options, including save or 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 according to 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.

Sort (Option 7)

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 'Operate on 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. To see why this is, examine the following series of numbers:

```
   45
  763
15777
   94
```

If these are stored in string form, the sorting proceeds from the ASCII code of the most significant character towards the least significant. In the above example, the first characters examined will be the codes for 4,7,1,9 because they take precedence over the less significant characters. Subsequently, if the numbers are to be sorted into highest-first order, the computer would judge the following order to be correct:

```
   94
  763
   45
15777
```

If the numbers had been originally entered with enough leading zeros to make all numbers 5 characters long, the sort would yield the following correct result:

```
15777
ØØ763
ØØØ94
ØØØ45
```

Sorting data into order is a complex business; unless efficient programming techniques are used for handling large files the time delay can be unacceptable, so the next chapter is devoted entirely to the subject.

Print file (Option 9)

This option allows a hard copy printout of a file. Serious use of a filing system frequently demands *hard copy* as well as screen displays so Option 9 deserves special attention. Normally, the keyword PRINT implies 'printing' to the screen because of the default interpretation. The peripheral output lines on the CPC464 are identified by the character '#' followed by a number. The parallel printer socket is identified by the number '8' so print statements, intended for hard copy output, must be of the form:

```
PRINT #8, variables
```

Although the Amstrad DMP-1 is the official printer for the CPC464 and will work without trouble as soon as it is plugged in, there will be many end users who already have a perfectly good printer and would rebel at the thought of buying another. Unfortunately, there is a lack of standardisation, both in printer interfaces and the control characters recognised by the printer software. It is commonly believed that any computer with a parallel (Centronics) socket at the back will always mate successfully with any printer boasting of a parallel interface. Sadly, this ideal situation is seldom realised in practice. Even if the leads and sockets happen to match (and very often they don't) there may still be trouble with the

corresponding pin numbers. For example, some computers send a carriage return out to the printer but, because the printer also generates a carriage return internally, the effect is an unwanted double spacing between line numbers. This can often be cured by studying the manual supplied with the printer and juggling around with control characters. Another possible cure, but not one that should be undertaken without previous experience, is to cut the connecting lead to pin 14 on the ribbon cable – but, here again, this depends on the particular printer.

Whilst on the subject of printers, it is worth remembering the old adage – you get what you pay for. Printers, particularly the so-called 'intelligent' variety, have dropped in price dramatically in spite of increased sophistication. It is very nice to have a bewildering variety of print fonts, underlining, superscripts, subscripts and, in some cases, several different ink colours, providing it has not been obtained at the expense of reliability. Once the initial excitement of trying out all the features has worn off, most people settle down to the simple default typeface because it becomes too much bother to look up the control characters for producing special characters. If the printer is to be worked hard and often, there are four questions relating to performance which will outweigh in importance all others put together:

(a) How robust and reliable is it?
(b) How easy is it to change the print head?
(c) Does it take standard printing paper?
(d) Does it have friction as well as tractor feed without crunching the paper every few sheets?

Totalise any column (Option 11)

The ability to totalise a column only makes sense if the column is purely *numeric* in character. For example, if the column is under the field heading 'Cash in hand' then totalising is a sensible operation. On the other hand, it would be absurd to totalise a column under the field heading 'Name'. The first prompt is 'Operate on which field?'. Providing the response is acceptable and the field exists, the result appears as follows:

```
Column total   = ....
column average = ....
```

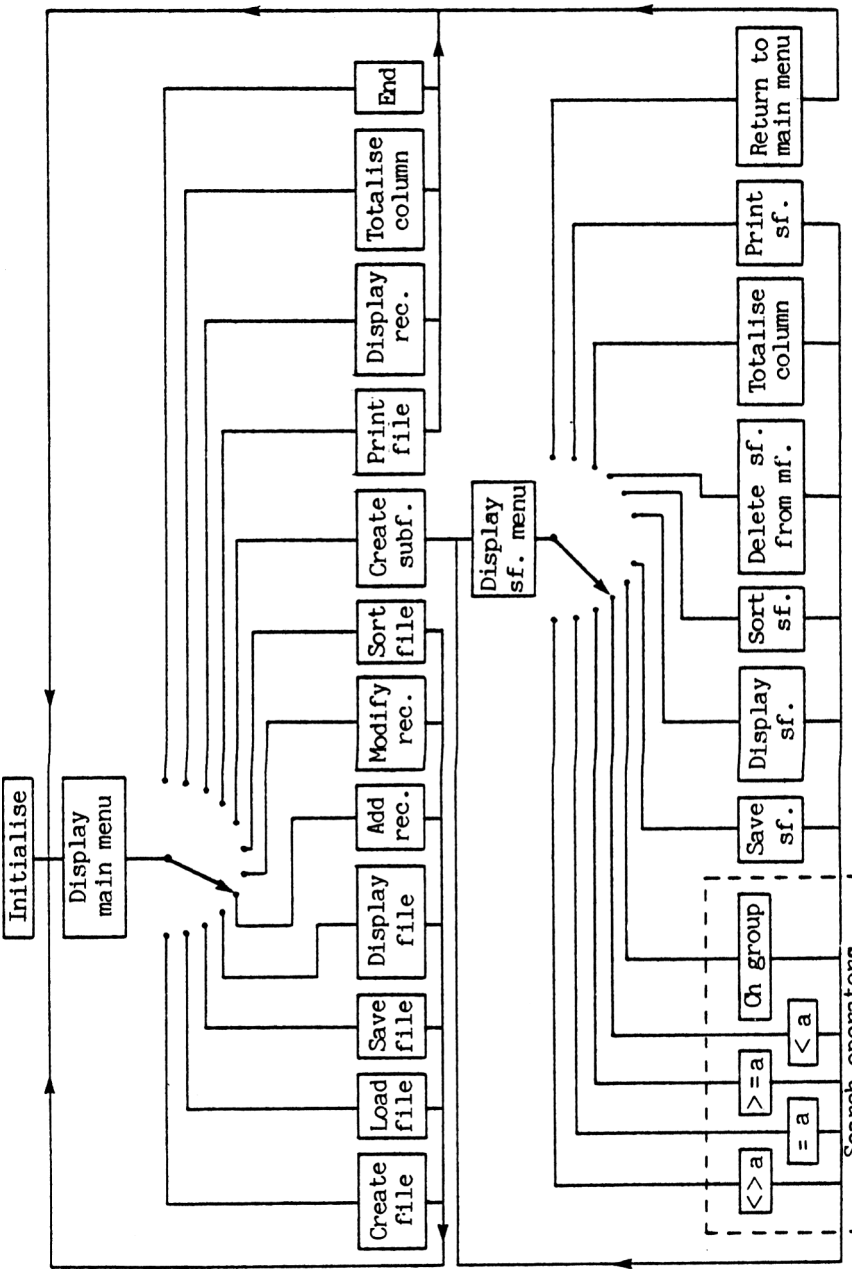


Fig. 4.2. Structure chart of Program 4.1.

End Program (Option 12)

This option simply ends the program and sets the cassette writing speed to its default value. Unfortunately, HIMEM cannot be reset to its default value unless a hard reset is performed. However, if the next program loaded is not heavy on memory, it does not matter.

Analysis of the program

The listing of Program 4.1 shows that the actual 'program' occupies less than 40 lines and extends only as far as the END in line 36ø. The remainder of the listing is a collection of subroutines. Figure 4.2 shows the overall strategy, particularly the relationship between subfile and mainfile options.

Variable names used in Program 4.1
mainsize% = current number of records in mainfile.
subsize% = current number of records in subfile.
L% = temporary variable set to the current number of records in either file as the case may be.
flag% = file status. 1 = file exists in RAM, ø = no file exists in RAM.
filename$ = name of file.
filesize% = maximum number of records that can be in the main file.
fields% = number of fields in file.
K$ = general purpose global variable.
total = column total.
F% = current field number.
R% = current record number.
A$(F%,R%) = two-dimensional file array element (a field).
A$(F%,ø) = heading of field F%.
SEL% = option number selected.
bottom% = bottom of display.
top% = top of display.

Subroutine calls

All subroutines are given a title by REM statements so they should be easy to locate on the listing. In accordance with established practice, there are no GOTOs to REM lines so these can be omitted from the listing if you wish to save memory. Within the body of the

subroutines, REMs have been used sparingly. Too many of them in a listing and continual reference to line numbers in accompanying descriptions can often be self-defeating. Anyway, most of the subroutines may be recognised as belonging to our standard list of building bricks given in the previous chapter.

Table 4.1. Subroutine calls.

Main subroutines	Low level subroutines
Main menu:	
Create file	Get line input
	Belt and braces
	Add records
Load file	Get filename
Save file	Get filename
Display file	Draw line
Add records	Press any key
	Get line input
Modify any record	Find record
	Draw line
Sort by any field	Sort file
	Find field heading
Print file	(no low level subroutines)
Display record	Find record
	Draw line
	Press any key
Totalise column	Find field heading
	Press any key
Subfile menu:	
Create subfile	Search file menu
	Title
	Search director
	Find field heading
	Get line input
	Press any key
	Search <>
	Search =
	Search >=
	Search <
	Search for character group
Delete subfile	(no low level subroutines)

Most options are implemented by first level subroutines which may call on second level which, in turn, may even call on third level subroutines. The table opposite 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.

Use of Program 4.1

The advantages, of RAM-based filing systems are the processing speed and the degree of sophistication which can be achieved. The disadvantage, of course, is the limit imposed by the necessity for the entire file to be loaded, even if we only want to look at one record. Fortunately, the CPC464 is reasonably well supplied with RAM so, in spite of the length restriction, the program will be found highly practical and well worth the effort of keying in.

The sort file option is obviously a candidate for experimentation. BASIC sort routines, even the best, are irritatingly slow when there is 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

All statements below refer to Program 4.1.

1. Unless a file is already in RAM, only Options 1 and 2 can be used.
2. Records can have up to 15 fields.
3. Field headings are limited to 18 characters.
4. Only two fields are visible at one time, one of which is the key field.
5. All data entered is converted to upper-case.
6. EXIT is entered to terminate data entry.
7. When using Option 4, left/right cursors bring other fields into view, up/down cursors bring another block of 20 records into view and the space bar regains menu.
8. The file name cannot exceed 16 characters.
9. If Option 1 (Create file) or Option 2 (Load file) is selected by mistake, pressing any key regains menu.
10. If either of the above two options is selected intentionally, it must be re-selected after pressing CTRL and M keys together.
11. A subfile can be extracted from the mainfile and saved. It then becomes a separate mainfile in its own right.

12. A mainfile can be shortened by deleting the subfile records from it and then being saved separately.
13. Files can be sorted under any field heading.
14. Numeric fields will not sort correctly unless all data was originally entered with the same number of digits, using leading zeros where necessary.
15. The print file option may require slight modification, depending on the printer model.

Self test

All questions relate to the Program 4.1 listing.

4.1 What is the purpose behind lines 2Ø to 5Ø?
4.2 SPEED WRITE Ø is the line before END. Why?
4.3 Why is ERASE A$ used in the BELT and BRACES module?
4.4 Line 137Ø uses VPOS(#Ø)−2. Why?

Chapter Five
Searching and Sorting

Introduction

Throughout this section, searching and sorting will be treated qualitatively to avoid boring the reader with too much dry academic analysis. The choice of algorithms treated is a mere sample of many that have been devised for internal sorting. We will progress from a simple exchange sort, through bubble, diminishing increment and quicksort to a complete two-dimensional string array sort written in machine code. The latter is capable of sorting a computerful of records in a few seconds. Finally, we cover algorithms for searching contiguous lists (arrays), the simple sequential search and the more efficient binary search.

Sorting arrays
The BASIC sort subroutines in this chapter are given line numbers starting at 1ØØØ. Preceding lines, if any, are concerned with a simple test program used to set up the array and display the execution time. The timing tables provided may need clarification. Each group of tests is performed on the following series of array configurations:

(a) Reverse order: The test array is set up with integers in reverse to the order required.
(b) Near ordered: The last element in the test array is sorted into its correct place at the centre array position.
(c) Random order: This is an array filled with random numbers.

The times given are typical and may vary depending on the particular numbers generated.

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 INTEGER SORT DEMONSTRATION
20 CLS
30 INPUT"Sort how many integers";NUMBER%
40 PRINT
50 REM FILL AND DISPLAY RANDOM ARRAY
60 DIM A%(NUMBER%)
70 FOR N%=1 TO NUMBER%
80 A%(N%)=10000*RND
90 PRINT A%(N%)
100 NEXT
110 PRINT:PRINT
120 PRINT"SORTING ARRAY"
130 PRINT:PRINT
140 START=TIME/300
150 GOSUB 1000
160 T=TIME/300-START
170 FOR N%=1 TO NUMBER%
180 PRINT A%(N%)
190 NEXT
200 PRINT
210 PRINT"RECORDS SORTED=";NUMBER%
220 PRINT
230 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
240 END
250 '
260 '
999 REM EXCHANGE SORT SUBROUTINE
1000 N%=0
1010 WHILE N%<NUMBER%-1
1020 N%=N%+1
1030 IF A%(N%)>A%(N%+1) THEN T%=A%(N%):A
```

```
%(N%)=A%(N%+1):A%(N%+1)=T%:N%=0
1040 WEND
1050 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	13.40 seconds
Reverse	50	197.92 seconds
near order	20	1.36 seconds
near order	50	8.53 seconds
near order	100	34.12 seconds
Random	20	8.6 seconds
Random	50	142.92 seconds

Study of Table 5.1 shows that the sorting time increases alarmingly as the number of elements increases. However, the time taken to execute when the array is nearly in order is reasonable. 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 is given by

$$\frac{1}{6}(N^3+5N-6)$$

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.

Program 5.2 consists of an inner and an outer control loop. The pairs of array elements are repeatedly incremented, compared and, if

```
999 REM BUBBLE SORT SUBROUTINE
1000 SIZE%=NUMBER%
1010 WHILE SIZE%>1
1020 SIZE%=SIZE%-1
1030 FOR N%=1 TO SIZE%
1040 IF A%(N%)>A%(N%+1) THEN T%=A%(N%):A
%(N%)=A%(N%+1):A%(N%+1)=T%
1050 NEXT
1060 WEND
1070 RETURN
```

Program 5.2. Bubble sort subroutine.

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. In 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. Incidentally, some purists argue that this algorithm is not a bubble sort at all. They prefer to call it a 'ripple sort' since movement takes place in a downward direction rather than upward.

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	2.25 seconds
Reverse	50	13.79 seconds
Reverse	100	54.83 seconds
Near order	20	1.39 seconds
Near order	50	8.08 seconds
Near order	100	31.54 seconds
Random	20	1.74 seconds
Random	50	10.80 seconds
Random	100	42.13 seconds

Studying 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 near ordered, is only marginally better than that of the exchange sort. This can be improved, 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, SF%, 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 SF% clear, the array must have been in order.

```
998 REM BUBBLE SORT SUBROUTINE
999 REM (WITH SWOP FLAGS)
1000 SIZE%=NUMBER%
1010 SF%=0
1020 SIZE%=SIZE%-1
1030 FOR N%=1 TO SIZE%
1040 IF A%(N%)>A%(N%+1) THEN T%=A%(N%):A
%(N%)=A%(N%+1):A%(N%+1)=T%:SF%=1
1050 NEXT
1060 IF SF%=1 THEN 1010
1070 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	2.38 seconds
Reverse	50	14.50 seconds
Reverse	100	57.62 seconds
Near order	20	1.01 seconds
Near order	50	6.12 seconds
Near order	100	24.27 seconds
Random	20	1.92 seconds
Random	50	10.82 seconds
Random	100	44.33 seconds

As can be seen from Table 5.3, the near-order performance is significantly 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. Although bubble sort routines are efficient for a small number of elements, the execution time increases alarmingly when in excess of about 15 or so. 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, where large values of N are involved, is to use one of the many diminishing increment variants, of which Program 5.4 is perhaps the simplest example.

The execution speed of the bubble sort for large lists is limited by the need to compare only *adjacent* array elements. This means that elements can only be moved by one array position at a time. The diminishing increment sort is an attempt to overcome this problem by making the initial comparisons on elements which are positioned far apart from each other. The sort process begins by making comparisons and exchanges over this large increment. Subsequently, elements closer together are sorted as the increment is progressively reduced at each sorting pass. An important proviso is that the final sorting pass must be performed with an increment of one. The early passes move the elements closer to their final array positions. The final pass, which performs an ordinary bubble sort, executes very quickly since the array is already roughly ordered.

If the above description seems a little difficult to digest at one sitting it is sometimes beneficial to visualise the mechanism another way. We noted earlier that the bubble sort is fairly efficient for sorting 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 mixed to form larger sets which will be roughly in order. These larger sets will be bubble sorted and mixed to form even larger sets and so on until we are left with one large roughly ordered list. This is finally bubble sorted and, due to the points made earlier, will be efficient. In Program 5.4 the number of sets is progressively halved each time round the outer loop but the number of elements within a set is doubled.

One of the best keys to the understanding of the mechanism of a

diminishing increment sort, or indeed any other algorithm, is to utilise 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.

```
998 REM DIMINISHING INCREMENT SORT
999  REM    (VERSION 1)
1000 E%=INT(LOG(NUMBER%)/LOG(2))
1010 F%=2^E%
1020 FOR G%=1 TO E%
1030 F%=F%/2:M%=NUMBER%-F%
1040 SF%=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%:SF%=1
1070 NEXT
1080 M%=M%-1
1090 IF SF%=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 which follows:

Table 5.4. Diminishing increment sort timings (version 1).

Array order	Elements	Execution time
Reverse	100	8.96 seconds
Reverse	200	20.96 seconds
Reverse	300	31.64 seconds
Near order	100	7.54 seconds
Near order	200	17.57 seconds
Near order	300	29.37 seconds
Random	100	21.65 seconds
Random	200	71.19 seconds
Random	300	136.28 seconds

Some interesting features are revealed 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.

Program 5.5 is a variation on the simple diminishing increment sort. The fundamental difference is in the way the sets are divided up and composed. The choice of increments is purely arbitrary as long as the final increment is 1. Other values of increment may work equally well, worse or even better. Notice that a constant (2) is added in at line 1Ø2Ø. We found the performance slightly improved by this addition for large numbers of array elements. Perhaps it helps the sets mix better. Analysis of Shellsort is very difficult and estimates of the number of comparisons required have, to date, only 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 Tables 5.4 and 5.5. The most outstanding improvement is in the sorting of a randomly generated array. The near-order performance, however, is only slightly improved.

```
998 REM DIMINISHING INCREMENT SORT
999 REM    (VERSION 2)
1000 N%=NUMBER%
1010 WHILE N%>1
1020 N%=(N%+2)\3
1030 FOR D%=N%+1 TO N%*2
1040 FOR E%=D% TO NUMBER% STEP N%
1050 FOR F%=E% TO D% STEP -N%
1060 IF A%(F%)<A%(F%-N%) THEN T%=A%(F%):
A%(F%)=A%(F%-N%):A%(F%-N%)=T% ELSE 1080
1070 NEXT
1080 NEXT E%
1090 NEXT
1100 WEND
1110 RETURN
```

Program 5.5. Diminishing increment sort (version 2).

This particular version was modified and used in Program 4.1 in the previous chapter and is quite acceptable written in BASIC provided N is not too large. Please note that the division sign in line 1Ø2Ø is integer division, not normal division.

Table 5.5. Diminishing increment sort timings (version 2).

Array order	Elements	Execution time
Reverse	100	9.92 seconds
Reverse	200	19.82 seconds
Reverse	300	37.14 seconds
Near order	100	7.36 seconds
Near order	200	14.66 seconds
Near order	300	26.11 seconds
Random	100	11.78
Random	200	23.85 seconds
Random	300	45.13 seconds

The Quicksort

The Quicksort algorithm devised and named by C. Hoare is one of the fastest known for sorting random array elements and can approach the theoretical minimum sorting time proportional to $n\log_2 n$. This algorithm often impresses due to its speed when sorting large numbers of random elements.

The fundamental idea behind Quicksort is the divide and conquer technique. We observed earlier that sorting small lists is far more efficient than sorting large lists. It follows that if we have a large array, split it into two sublists containing different ranges of numbers and sort each list separately, we will save a lot of sorting time. To do this, we estimate an array element that, hopefully, will have a median value and call this the *pivot*. All array elements having a value less than the pivot will be placed above it in the lower half of the array and all elements having a value greater than the pivot placed in the higher half. If these two portions of the array are sorted separately, either side of the pivot, then the array will be completely sorted. The estimate of the pivot value is all-important here. For instance, we could choose the first array element or the last array element in a random array. However, if the list is partially sorted, as may occur in practice, the performance may be seriously degraded because the pivot will be too far offset from the median value to make the split worthwhile. This effect can be reduced statistically by choosing the pivot as the mid-point element of the array. There are, of course, many other ways of obtaining the pivot but we will employ this

method. Incidentally, during worst case conditions, where the above effect is dominant, the sorting time can be as poor as that for a bubble sort (approximately proportional to n^2) but this is unlikely to happen in practice.

One method of implementing Quicksort is to keep partitioning lists in this way till we have many lists containing, say, 15 elements at most and then bubble sorting them. Alternatively, if we carry on and take this partitioning process to its limit then each sublist will eventually contain only one element, in which case there will be no need to employ a bubble sort at all.

Normally, Quicksort would be written in a form suitable for exploiting *recursion*. This term is used to define the practice whereby a subprogram calls itself from within its own bounds. This is how the many levels of partitioning are effected. Recursion normally requires that defined *PROCEDURES* with *local variables* are supported. These, although included in the more structured microcomputer languages such as Pascal and BBC BASIC, are sadly omitted in the otherwise impressive Amstrad CPC464 version of BASIC, presumably because of lack of space in the ROM. This does complicate matters but, fortunately – although beyond the scope of this book – a set of rules can be applied to remove recursion from algorithms by employing a *stack*. (A stack is a section of memory where the last variable value stored is the first recalled.) These rules have been applied and Program 5.6 is the result. The array index limits of the lower sublist are p% and q% (head and tail respectively) and r% and s% for the higher sublist. The head and tail parameters of the sublists yet to be sorted are put on the stack. Thus, if we are presently sorting the array between A%(p%) and A%(q%) then r% and s% would be placed on the stack so the sublist A%(r%) to A%(q%) could be sorted later. It transpires that it is better to put the longer sublist limits on the stack and process the shorter sublist immediately. This is the task performed in line 11ØØ. Each time a list is further partitioned in the outer WHILE/WEND loop the process is repeated. On exit of the loop in line 112Ø, the sublists, whose parameters were placed on the stack, are taken in sequence (last in, first out) and sorted in a similar manner.

It is often stated that Quicksort uses a lot of memory because the program listing is longer, it uses more variables and needs to employ a fixed stack area of memory. However, the extra memory used by the stack itself is not excessive. The number of stack levels needed by Quicksort is given by $\log_2 n$. Therefore, to sort, say, 4096 numbers, the number of stack levels needed will be $\log_2(4096)=12$. In Program 5.6

we need two stacks, consisting of at least 12 integer array elements each, making a total of 24. Each array element DIMensioned consumes 2 bytes so 48 bytes of array space will be claimed. This memory need not be permanently allocated on the Amstrad CPC464 since we can DIMension the stack at the start of the subroutine (line 1ØØØ) and ERASE it at line 113Ø thus freeing the memory again for use by the main program.

The demonstration program for Quicksort is Program 5.6 and the sorting time table is Table 5.6. The table reveals that, over the range tested, there is an approximately linear relationship between the execution time and the number of elements sorted. The near-ordered performance was not seriously degraded in this particular test, but we may not be so lucky in some of the other, largely ordered, situations mentioned above.

```
999 REM QUICKSORT SUBROUTINE
1000 DIM stack1%(16),stack2%(16)
1010 sp%=0:head%=1:tail%=NUMBER%
1020 WHILE head%<tail%
1030 pivot%=A%((head%+tail%)\2)
1040 a%=head%:b%=tail%
1050 WHILE A%(a%)<pivot%:a%=a%+1:WEND
1060 WHILE A%(b%)>pivot%:b%=b%-1:WEND
1070 IF a%<b% THEN t%=A%(a%):A%(a%)=A%(b
%):A%(b%)=t%:a%=a%+1:b%=b%-1:GOTO 1050
1080 IF a%=b% THEN q%=b%-1:r%=a%+1 ELSE
q%=b%:r%=a%
1090 sp%=sp%+1:p%=head%:s%=tail%
1100 IF q%-p%<s%-r% THEN stack1%(sp%)=r%
:stack2%(sp%)=s%:head%=p%:tail%=q% ELSE
stack1%(sp%)=p%:stack2%(sp%)=q%:head%=r%
:tail%=s%
1110 WEND
1120 IF sp%>0 THEN head%=stack1%(sp%):ta
il%=stack2%(sp%):sp%=sp%-1:GOTO 1020
1130 ERASE stack1%,stack2%
1140 RETURN
```

Program 5.6. Quicksort.

Table 5.6. Quicksort timings.

Array order	Elements	Execution time
Reverse	100	6.08 seconds
Reverse	200	12.77 seconds
Reverse	300	18.65 seconds
Near order	100	7.45 seconds
Near order	200	15.51 seconds
Near order	300	23.30 seconds
Random	100	10.12 seconds
Random	200	19.32 seconds
Random	300	30.41 seconds
Random	1000	110.00 seconds
Random	2000	243.48 seconds

It is generally agreed that there is no universally superior sorting algorithm. Each has its relative advantages and disadvantages. The programmer should choose the one most suited to the particular task in hand taking into account such factors as the number of items to be sorted, available memory, acceptable execution time, and – most important of all – programming effort. It is pointless to use an elaborate sorting routine, such as Quicksort, to sort a list of say twenty items. It is equally pointless to employ one to sort a fairly large list once only. An analogy can be found between sort routines and cricketers. Cricketers are either good bowlers or good batsmen but rarely both.

Sorting strings

Earlier examples in this chapter have all used integer array elements as the variables. 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. As an example of the conversion necessary, we have produced a string version of Quicksort with an accompanying testing program. The listing is given as Program 5.7. It should be a fairly easy process to convert the earlier subroutines to sort strings. A timing table, Table 5.7, shows typical execution times for sorting various size random string arrays.

Table 5.7. Typical execution times for Program 5.7.

Array size	Typical sorting time
100	9.65 seconds
200	20.37 seconds
300	31.41 seconds
500	54.94 seconds
1000	230.76 seconds

```
10 REM SORT TEST PROGRAM
20 REM STRING ARRAY
30 CLS
40 INPUT"Sort how many strings";NUMBER%
50 PRINT
60 REM FILL AND DISPLAY RANDOM ARRAY
70 DIM A$(NUMBER%)
80 FOR R%=1 TO NUMBER%
90 B$=""
100 A%=6*RND+1
110 FOR Z%=1 TO A%
120 N%=25*RND
130 K$=CHR$(N%+65)
140 B$=B$+K$
150 NEXT
160 A$(R%)=B$
170 PRINT A$(R%)
180 NEXT
190 PRINT:PRINT
200 PRINT"SORTING ARRAY"
210 PRINT:PRINT
220 START=TIME/300
230 GOSUB 1000
240 T=TIME/300-START
250 FOR R%=1 TO NUMBER%
260 PRINT A$(R%)
270 NEXT
280 PRINT
290 PRINT"RECORDS SORTED=";NUMBER%
300 PRINT
310 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
```

```
320 END
330 '
340 '
999 REM STRING QUICKSORT SUBROUTINE
1000 DIM stack1%(16),stack2%(16)
1010 sp%=0:head%=1:tail%=NUMBER%
1020 WHILE head%<tail%
1030 pivot$=A$((head%+tail%)\2)
1040 a%=head%:b%=tail%
1050 WHILE A$(a%)<pivot$:a%=a%+1:WEND
1060 WHILE A$(b%)>pivot$:b%=b%-1:WEND
1070 IF a%<b% THEN t$=A$(a%):A$(a%)=A$(b
%):A$(b%)=t$:a%=a%+1:b%=b%-1:GOTO 1050
1080 IF a%=b% THEN q%=b%-1:r%=a%+1 ELSE
q%=b%:r%=a%
1090 sp%=sp%+1:p%=head%:s%=tail%
1100 IF q%-p%<s%-r% THEN stack1%(sp%)=r%
:stack2%(sp%)=s%:head%=p%:tail%=q% ELSE
stack1%(sp%)=p%:stack2%(sp%)=q%:head%=r%
:tail%=s%
1110 WEND
1120 IF sp%>0 THEN head%=stack1%(sp%):ta
il%=stack2%(sp%):sp%=sp%-1:GOTO 1020
1130 ERASE stack1%,stack2%
1140 RETURN
```

Program 5.7. String version of Quicksort.

Rectangular or two-dimensional string array

A common file organisation, encountered in RAM-based filing systems, is the two-dimensional or rectangular string array. If a RAM-based file is DIMensioned A$(FIELDS%,RECORDS%) then the file A$ can be considered to contain RECORDS% records each of FIELDS% 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.

File stored in RAM as a rectangular array			
	Field Ø	Field 1	Field 2
Field headings	A$(Ø,Ø)	A$(1,Ø)	A$(2,Ø)
Record 1	A$(Ø,1)	A$(1,1)	A$(2,1)
Record 2	A$(Ø,2)	A$(1,2)	A$(2,2)
Record 3	A$(Ø,3)	A$(1,3)	A$(2,3)
Record 4	A$(Ø,4)	A$(1,4)	A$(2,4)
Record 5	A$(Ø,5)	A$(1,5)	A$(2,5)
Record n	A$(Ø,n)	A$(1,n)	A$(2,n)

Fig. 5.1. Storing a data file as a rectangular array.

Figure 5.1 shows how a file of this type is visualised in memory. There are a few points in Figure 5.1 in need of clarification:

(a) When BASIC DIMensions a rectangular array, 3-byte *string descriptors* are reserved for each array element whether they are used or not. Unfortunately, this includes all zero-indexed elements. In order to maximise the use of available memory we should ensure that field zero is used as one of the legitimate fields. If we do this then we can DIMension, say, a 400-record, 5-field file as A$(4,4ØØ) rather than A$(5,4ØØ) which would, at first sight, seem reasonable. Neglecting this point could waste 400×3 = 1200 bytes of memory unnecessarily.

(b) The record zero elements are not wasted since they are convenient for storing the field headings themselves.

Sorting rectangular string arrays

If it is required that records are to be sorted according to a specific field, we need to modify our routines to exchange all fields of the record pairs each time. An extra FOR/NEXT loop will be needed to take care of this.

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

Program 5.8 shows how the diminishing increment sort (version 2) is modified for sorting a rectangular string array. We employed this subroutine for Program 4.1 in the previous chapter. A test program is also given that can generate a chosen number of fixed, three-field, randomly generated records. See Table 5.8 for typical execution times.

Table 5.8. Typical execution times for Program 5.8.

Records	Typical sorting time
100	27.91 seconds
200	109.66 seconds
300	310.70 seconds

```
10 REM TEST PROGRAM
20 REM SORTING A RECTANGULAR STRING
30 REM ARRAY WITH BASIC SUBROUTINES
40 CLS
50 INPUT"Sort how many 3 field records";
RECORDS%
60 FIELDS%=2:REM fields 0,1,2
70 INPUT"Sort which field (0-2)";FIELDNU
M%
80 PRINT
90 REM FILL AND DISPLAY RANDOM ARRAY
100 DIM A$(FIELDS%,RECORDS%)
110 FOR R%=1 TO RECORDS%
120 FOR F%=0 TO FIELDS%
130 B$=""
```

```
140 A%=6*RND+1
150 FOR Z%=1 TO A%
160 N%=25*RND
170 K$=CHR$(N%+65)
180 B$=B$+K$
190 NEXT
200 A$(F%,R%)=B$
210 PRINT A$(F%,R%),
220 NEXT
230 NEXT
240 PRINT:PRINT
250 PRINT"SORTING ARRAY"
260 PRINT:PRINT
270 START=TIME/300
280 GOSUB 1000
290 T=TIME/300-START
300 FOR R%=1 TO RECORDS%
310 FOR F%=0 TO FIELDS%
320 PRINT A$(F%,R%),
330 NEXT
340 NEXT
350 PRINT
360 PRINT"RECORDS SORTED=";RECORDS%
370 PRINT
380 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
390 END
400 '
410 '
998 REM DIMINISHING INCREMENT SORT
999 REM    (VERSION 2)
1000 N%=RECORDS%
1010 WHILE N%>1
1020 N%=(N%+2)\3
1030 FOR D%=N%+1 TO N%*2
1040 FOR E%=D% TO RECORDS% STEP N%
1050 FOR R%=E% TO D% STEP -N%
1060 IF A$(FIELDNUM%,R%)<A$(FIELDNUM%,R%
-N%) THEN FOR C%=0 TO FIELDS%:T$=A$(C%,R
%):A$(C%,R%)=A$(C%,R%-N%):A$(C%,R%-N%)=T
$:NEXT ELSE 1080
1070 NEXT
1080 NEXT E%
```

```
1090 NEXT
1100 WEND
1110 RETURN
```

Program 5.8. Diminishing increment sort of a rectangular string array.

A rectangular string array version of Quicksort, given in Program 5.9, can replace the sort routine used in Program 4.1 of the previous chapter. This might give better all-round performance, but remember that the variables RECORDS% and FIELDNUM% should be changed to L% and F% respectively. Table 5.9 shows the typical sorting times to be expected.

Table 5.9. Typical execution times for Program 5.9.

Records	Typical sorting time
100	15.49 seconds
200	33.42 seconds
300	144.87 seconds
500	596.37 seconds

```
998 REM SUBROUTINE: QUICKSORT
999 REM RECTANGULAR STRING ARRAY
1000 DIM stack1%(16),stack2%(16)
1010 sp%=0:head%=1:tail%=RECORDS%
1020 WHILE head%<tail%
1030 pivot$=A$(FIELDNUM%,(head%+tail%)\2
)
1040 a%=head%:b%=tail%
1050 WHILE A$(FIELDNUM%,a%)<pivot$:a%=a%
+1:WEND
1060 WHILE A$(FIELDNUM%,b%)>pivot$:b%=b%
-1:WEND
1070 IF a%<b% THEN FOR C%=0 TO FIELDS%:t
$=A$(C%,a%):A$(C%,a%)=A$(C%,b%):A$(C%,b%
)=t$:NEXT:a%=a%+1:b%=b%-1:GOTO 1050
1080 IF a%=b% THEN q%=b%-1:r%=a%+1 ELSE
q%=b%:r%=a%
1090 sp%=sp%+1:p%=head%:s%=tail%
1100 IF q%-p%<s%-r% THEN stack1%(sp%)=r%
:stack2%(sp%)=s%:head%=p%:tail%=q% ELSE
stack1%(sp%)=p%:stack2%(sp%)=q%:head%=r%
```

```
:tail%=s%
1110 WEND
1120 IF sp%>0 THEN head%=stack1%(sp%):ta
il%=stack2%(sp%):sp%=sp%-1:GOTO 1020
1130 ERASE stack1%,stack2%
1140 RETURN
```

Program 5.9. Quicksort of a rectangular string array.

Machine code solutions

Whatever sort technique 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 would not execute much faster than 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.

Although the subject of Z80 machine code programming is outside the scope of this volume, it would be a pity if we neglected to include a few powerful routines for sorting arrays. We therefore ask you to accept these routines with only an outline explanation. However, it is worth making use of these routines so the emphasis will be on their use rather than the programming details. If nothing else, it should give you an idea of the speed advantages to be gained from machine code programming.

All the *assembly language* listings in this section have been developed using the inexpensive tape version of the 'HiSoft DEVPAC Editor, Assembler, Disassembler & Monitor' (Amsoft 116). Do not worry if you cannot lay your hands on this particular piece of software. Hex loading programs, written in BASIC, will be

provided so that the machine code bytes can be loaded directly from BASIC DATA statements into a fixed area of memory. This section of memory can then be saved to tape as a binary file.

Machine code string array sort

Practical files usually have more fields holding alpha characters than numeric. When a string array is DIMensioned by the BASIC interpreter, an access table is set up containing three bytes for each string element. 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 *string descriptors* and represent the string length, low byte address and high byte address respectively. The string length byte is the lowest in memory. The actual string, consisting of the ASCII codes in sequential memory locations, is stored at the starting address given in bytes 2 and 3 above. A string array is thus formed by a series of such string descriptors stored sequentially in memory. Swopping strings during a string sort is not as difficult as it may first appear. We can leave the strings themselves where they are in memory and swop the string descriptors, since these tell the BASIC interpreter where the strings are stored.

Program 5.10 is what is known as an assembly code or *source code* listing of a diminishing increment string sort. The final machine code or *object code*, necessary to sort string arrays, is assembled from this and placed into memory from address &998Ø onwards.

The corresponding test program is Program 5.11 which sets up a random string array, calls the machine code routine and displays the sorted array. Finally, for those without an assembler, a hex loading program is given in the form of Program 5.12.

```
 10 ;DIMINISHING INCREMENT SORT
 20 ;OF A STRING ARRAY
 30 begin:   EQU  #9980
 40 top:     EQU  begin+#F0
 50 number:  EQU  top
 60 array:   EQU  top+2
 70 power:   EQU  top+4
 80 cycles:  EQU  top+5
 90 count:   EQU  top+7
100 flag:    EQU  top+9
110 point1:  EQU  top+10
```

```
120 point2: EQU  top+12
130 len1:   EQU  top+14
140 len2:   EQU  top+15
150 addr1:  EQU  top+16
160 addr2:  EQU  top+18
170 size:   EQU  top+20
180         ORG  begin
190 ;Initialise
200         LD   HL,1
210         LD   (size),HL
220 ;Pick up and store
230 ;array start address
240         LD   L,(IX)
250         LD   H,(IX+1)
260         LD   (array),HL
270 ;Pick up number of strings
280 ;and store in number
290         LD   L,(IX+2)
300         LD   H,(IX+3)
310         LD   (number),HL
320 ;Find next power of 2 >= number
330 ;and store in size
340         EX   DE,HL
350         SUB  A
360 loop:   INC  A
370         LD   HL,(size)
380         ADD  HL,HL
390         LD   (size),HL
400         SBC  HL,DE
410         JR   C,loop
420 ;Store corresponding
430 ;index in power
440         LD   (power),A
450 ;Divide size by 2
460 outer:  LD   DE,(size)
470         SRL  D
480         RR   E
490         LD   (size),DE
500 ;Subtract size from number
510 ;and store in cycles
520         LD   HL,(number)
530         SBC  HL,DE
540         LD   (cycles),HL
```

```
550 ;Initialise count and swop flag
560 mid:    SUB  A
570         LD   (count),A
580         LD   (count+1),A
590         LD   (flag),A
600 ;Store array start address
610 ;in point1
620         LD   HL,(array)
630         LD   (point1),HL
640 ;Multiply size by 3 add point1
650 ;store result in point2
660         LD   HL,(size)
670         LD   D,H
680         LD   E,L
690         ADD  HL,HL
700         ADD  HL,DE
710         LD   DE,(point1)
720         ADD  HL,DE
730         LD   (point2),HL
740 ;Get lengths and addresses
750 ;of strings to be compared
760 inner:  LD   HL,(point1)
770         LD   A,(HL)
780         LD   (len1),A
790         INC  HL
800         LD   A,(HL)
810         LD   (addr1),A
820         INC  HL
830         LD   A,(HL)
840         LD   (addr1+1),A
850         LD   HL,(point2)
860         LD   A,(HL)
870         LD   (len2),A
880         INC  HL
890         LD   A,(HL)
900         LD   (addr2),A
910         INC  HL
920         LD   A,(HL)
930         LD   (addr2+1),A
940         LD   B,0
950 ;Compare strings a character at
960 ;a time, branch swop or noswop
970         LD   DE,(addr2)
```

```
 980          LD    HL,(addr1)
 990 comp:    LD    A,(DE)
1000          CP    (HL)
1010          JR    C,swop
1020          JR    NZ,noswop
1030          INC   B
1040          LD    A,(len1)
1050          CP    B
1060          JR    Z,noswop
1070          LD    A,(len2)
1080          CP    B
1090          JR    Z,swop
1100          INC   DE
1110          INC   HL
1120          JR    NZ,comp
1130 ;Initialise byte counter
1140 ;set swop flag
1150 swop:    LD    B,3
1160          LD    (flag),A
1170          LD    DE,(point1)
1180          LD    HL,(point2)
1190 ;Swop string descriptors
1200 ;a byte at a time
1210 loop2:   LD    A,(DE)
1220          LD    C,(HL)
1230          EX    DE,HL
1240          LD    (HL),C
1250          LD    (DE),A
1260          INC   DE
1270          INC   HL
1280          DJNZ  loop2
1290 ;Add 3 to each of
1300 ;point1 and point2
1310 noswop:  LD    DE,3
1320          LD    HL,(point1)
1330          ADD   HL,DE
1340          LD    (point1),HL
1350          LD    HL,(point2)
1360          ADD   HL,DE
1370          LD    (point2),HL
1380 ;Increment count
1390          LD    DE,(count)
1400          INC   DE
```

```
1410          LD    (count),DE
1420 ;Compare count to cycles
1430 ;if not equal jump to inner
1440          LD    HL,(cycles)
1450          SBC   HL,DE
1460          JP    NZ,inner
1470 ;Jump to fclear
1480 ;if swop flag is clear
1490          LD    A,(flag)
1500          CP    0
1510          JR    Z,fclear
1520 ;Decrement cycles and
1530 ;jump to mid if not zero
1540          LD    HL,(cycles)
1550          DEC   HL
1560          LD    (cycles),HL
1570          LD    A,H
1580          OR    L
1590          JP    NZ,mid
1600 ;Decrement power and
1610 ;jump to outer if not zero
1620 fclear:  LD    HL,power
1630          DEC   (HL)
1640          JP    NZ,outer
1650          RET
```

Program 5.10. Diminishing increment sort of string array.

```
10 REM SORT TEST PROGRAM
20 REM STRING ARRAY
30 CLS
40 PRINT"Place object code tape in DATAC
ORDER"
50 OPENOUT "buffer"
60 MEMORY &997F
70 CLOSEOUT
80 LOAD "STRINGSORT",&9980
90 CLS
100 INPUT"Sort how many strings";NUMBER%
110 PRINT
120 REM FILL AND DISPLAY RANDOM ARRAY
130 DIM A$(NUMBER%)
140 FOR R%=0 TO NUMBER%-1
150 B$=""
```

```
160 A%=6*RND+1
170 FOR Z%=1 TO A%
180 N%=25*RND
190 K$=CHR$(N%+65)
200 B$=B$+K$
210 NEXT
220 A$(R%)=B$
230 PRINT A$(R%)
240 NEXT
250 PRINT:PRINT
260 PRINT"SORTING ARRAY"
270 PRINT:PRINT
280 START=TIME/300
290 CALL &9980,NUMBER%,@A$(0)
300 T=TIME/300-START
310 FOR R%=0 TO NUMBER%-1
320 PRINT A$(R%)
330 NEXT
340 PRINT
350 PRINT"RECORDS SORTED=";NUMBER%
360 PRINT
370 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
380 PRINT
390 INPUT"Another test (Y/N)";K$
400 K$=UPPER$(K$)
410 IF K$="Y" THEN ERASE A$:GOTO 90 ELSE
 END
```

Program 5.11. BASIC test program for machine code string sort.

```
10 REM PRODUCING A MACHINE CODE FILE
20 REM SORT STRING ARRAY
30 REM (no assembler needed)
40 CLS
50 ADDRESS%=&9980
60 FOR N%=0 TO &EF
70 READ BYTE$
80 POKE ADDRESS%+N%,VAL("&"+BYTE$)
90 NEXT
100 PRINT"Now produce an object code fil
e on tape"
110 SAVE "STRINGSORT",B,&9980,&F0
120 END
```

```
130 DATA 21,01,00,22,84,9A,DD,6E
140 DATA 00,DD,66,01,22,72,9A,DD
150 DATA 6E,02,DD,66,03,22,70,9A
160 DATA EB,97,3C,2A,84,9A,29,22
170 DATA 84,9A,ED,52,38,F4,32,74
180 DATA 9A,ED,5B,84,9A,CB,3A,CB
190 DATA 1B,ED,53,84,9A,2A,70,9A
200 DATA ED,52,22,75,9A,97,32,77
210 DATA 9A,32,78,9A,32,79,9A,2A
220 DATA 72,9A,22,7A,9A,2A,84,9A
230 DATA 54,5D,29,19,ED,5B,7A,9A
240 DATA 19,22,7C,9A,2A,7A,9A,7E
250 DATA 32,7E,9A,23,7E,32,80,9A
260 DATA 23,7E,32,81,9A,2A,7C,9A
270 DATA 7E,32,7F,9A,23,7E,32,82
280 DATA 9A,23,7E,32,83,9A,06,00
290 DATA ED,5B,82,9A,2A,80,9A,1A
300 DATA BE,38,13,20,26,04,3A,7E
310 DATA 9A,B8,28,1F,3A,7F,9A,B8
320 DATA 28,04,13,23,20,E9,06,03
330 DATA 32,79,9A,ED,5B,7A,9A,2A
340 DATA 7C,9A,1A,4E,EB,71,12,13
350 DATA 23,10,F7,11,03,00,2A,7A
360 DATA 9A,19,22,7A,9A,2A,7C,9A
370 DATA 19,22,7C,9A,ED,5B,77,9A
380 DATA 13,ED,53,77,9A,2A,75,9A
390 DATA ED,52,C2,DC,99,3A,79,9A
400 DATA FE,00,28,0C,2A,75,9A,2B
410 DATA 22,75,9A,7C,B5,C2,BD,99
420 DATA 21,74,9A,35,C2,A9,99,C9
```

Program 5.12. Hex loading program for machine code string sort.

Producing an object code file

In order to use machine code subroutines conveniently from a BASIC program it is necessary to produce a *binary file* of the object code on tape. This file can then be loaded automatically by the BASIC program in the first few lines. There are two ways to produce the object code file:

Method 1: Using an assembler
This method assumes that you are in possession of the HiSoft DEVPAC assembler.

(a) Load the HiSoft DEVPAC assembler at, say, address 65ØØ decimal. You will be prompted for this address.
(b) Type in the assembly code listing exactly as printed in Program 5.10 (this is sometimes called the source code listing).
(c) Save a copy of the source code on tape by typing:

P 1Ø,165Ø,<filename>

The file can be reloaded at some later time with:

G,,<filename>

(d) Assemble the source code by typing in A as an assembler command. When asked for table size, respond with 5ØØ. This is more than adequate space for the symbol table.
(e) Clear typing errors if any errors are reported.
(f) Produce an object code file that can be loaded directly from tape by a main program. This is done automatically by typing the assembler command:

O,,STRINGSORT

Note that the filename STRINGSORT is not enveloped in double quotes.
(g) Perform a hard reset to clear memory.
(h) Type in the test program, Program 5.11. This will automatically load, run and test the STRINGSORT file you have just produced.

The object code itself is loaded into a 512-byte section of memory reserved above HIMEM at &998Ø but below the permanently allocated cassette buffer area (refer back to 'Allocation of cassette buffer area' in Chapter 1).

Method 2: No assembler

Type in Program 5.12 and RUN it. The object code file will be produced automatically. This program is simply a loop which picks up the machine code bytes from DATA statements and dumps them directly into memory from &998Ø onwards. The code is then automatically saved on tape.

Relocation of machine code

The main disadvantage of method 2 is that the code will only execute at address &998Ø. Moving it elsewhere in memory will result in chaos because the object code is said to be not relocatable. If you have a standard machine without disk drives or light pens etc., there is no

problem. However, it may be necessary to resort to Method 1 to reassemble the source code at an alternative address if such peripherals are connected. This is an easy task with Program 5.10; simply change the address in the EQU statement in line 3Ø and reassemble.

Calling STRINGSORT from any user program

To CALL the machine code STRINGSORT routine it is necessary to pass over a few parameters. This is accomplished by the insertion of one line in your program. For instance,

```
CALL &998Ø,NUMBER%,@A$(1)
```

will sort the array A$ from A$(1) to A$(NUMBER%).

The &998Ø term is the execution address of the subroutine. This can be altered if you assemble the code elsewhere in memory. NUMBER%, an integer variable, is the number of strings in the array that are to be sorted. Any other variable will do, as long as it equals this value. For instance, you could use SIZE% instead. The @A$(1) parameter passes over the address of the first string descriptor of the array to be sorted.

Passing @A$(Ø) will sort A$(Ø) to A$(NUMBER%−1) inclusive. Passing @A$(1) will sort A$(1) to A$(NUMBER%) inclusive. Note that the '@' symbol preceding a variable means pass over the address of the variable to the machine code routine, not the variable value itself.

When a CALL statement is executed in BASIC, the parameter list values, following the execution address, are passed over to the machine code subroutine. These are automatically stored in a parameter block offset from an address stored in the IX register of the Z80 microprocessor. The parameter list values are offset from the IX register contents in the reverse order to that which appears in the CALL statement. For instance, with the above CALL, the address of the A$(1) string descriptor is set up in (IX) and (IX+1) and the actual value of NUMBER% is set up in (IX+2) and (IX+3), low byte and high byte respectively. These parameters may then be picked up and stored in a more convenient section of memory by the machine code subroutine itself.

To give an idea of the execution speed that can be expected from Program 5.10, see Table 5.10.

Table 5.10. Typical execution times for Program 5.10.

Array size	Typical sorting time
100	0.53 seconds
200	1.59 seconds
300	2.97 seconds
500	7.23 seconds
1000	22.32 seconds
2000	64.35 seconds

Machine code diminishing increment sort of a rectangular string array

Program 5.13 will sort a two-dimensional string array of the type described earlier in this chapter. It is fairly fast. In fact, it will sort a computerful of records in about 8 seconds. However, successful operation depends on the field index being the first DIMensioned. That is to say, the rectangular array is dimensioned DIM A$(FIELDS%,RECORDS%) and individual fields F%, of record R%, are accessed A%(F%,R%).

```
 10 ;DIMINISHING INCREMENT SORT OF A
 20 ;TWO DIMENSIONAL STRING ARRAY
 30 begin:   EQU  #9980
 40 top:     EQU  begin+#011C
 50 number:  EQU  top
 60 array:   EQU  top+2
 70 power:   EQU  top+4
 80 cycles:  EQU  top+5
 90 bytes:   EQU  top+7
100 count:   EQU  top+9
110 flag:    EQU  top+11
120 point1:  EQU  top+12
130 point2:  EQU  top+14
140 len1:    EQU  top+16
150 len2:    EQU  top+17
160 addr1:   EQU  top+18
170 addr2:   EQU  top+20
180 size:    EQU  top+22
```

```
190 offset: EQU  top+24
200         ORG  begin
210 ;Initialise
220         LD   HL,0
230         LD   (bytes),HL
240         LD   (offset),HL
250         INC  HL
260         LD   (size),HL
270 ;Pick up and store
280 ;array start address
290         LD   L,(IX)
300         LD   H,(IX+1)
310         LD   (array),HL
320 ;Pick up field number
330 ;multiply by 3 and
340 ;store in offset
350         LD   A,(IX+2)
360         LD   B,A
370         SLA  A
380         ADD  A,B
390         LD   (offset),A
400 ;Pick up number of fields
410 ;add 1 and store in bytes
420         LD   A,(IX+4)
430         INC  A
440         LD   B,A
450         SLA  A
460         ADD  A,B
470         LD   (bytes),A
480 ;Pick up number of records
490 ;and store in number
500         LD   L,(IX+6)
510         LD   H,(IX+7)
520         LD   (number),HL
530 ;Find next power of 2 >= number
540 ;and store in size
550         EX   DE,HL
560         SUB  A
570 loop:   INC  A
580         LD   HL,(size)
590         ADD  HL,HL
600         LD   (size),HL
610         SBC  HL,DE
```

```
620          JR    C,loop
630 ;store corresponding
640 ;index in power
650          LD    (power),A
660 ;Divide size by 2
670 outer:   LD    DE,(size)
680          SRL   D
690          RR    E
700          LD    (size),DE
710 ;Subtract size from number
720 ;and store in cycles
730          LD    HL,(number)
740          SBC   HL,DE
750          LD    (cycles),HL
760 ;Initialise count and swop flag
770 mid:     SUB   A
780          LD    (count),A
790          LD    (count+1),A
800          LD    (flag),A
810 ;Store array start address
820 ;in point1
830          LD    HL,(array)
840          LD    (point1),HL
850 ;Multiply size by bytes add
860 ;point1 store result in point2
870          LD    A,(bytes)
880          LD    B,A
890          LD    HL,0
900          LD    DE,(size)
910 mult:    ADD   HL,DE
920          DJNZ  mult
930          LD    DE,(point1)
940          ADD   HL,DE
950          LD    (point2),HL
960 ;Get lengths and addresses of
970 ;strings to be compared
980 inner:   LD    DE,(offset)
990          LD    HL,(point1)
1000         ADD   HL,DE
1010         LD    A,(HL)
1020         LD    (len1),A
1030         INC   HL
1040         LD    A,(HL)
```

```
1050          LD    (addr1),A
1060          INC   HL
1070          LD    A,(HL)
1080          LD    (addr1+1),A
1090          LD    HL,(point2)
1100          ADD   HL,DE
1110          LD    A,(HL)
1120          LD    (len2),A
1130          INC   HL
1140          LD    A,(HL)
1150          LD    (addr2),A
1160          INC   HL
1170          LD    A,(HL)
1180          LD    (addr2+1),A
1190          LD    B,0
1200          LD    DE,(addr2)
1210          LD    HL,(addr1)
1220 ;Compare strings a character at
1230 ;a time branch swop or noswop
1240 comp:    LD    A,(DE)
1250          CP    (HL)
1260          JR    C,swop
1270          JR    NZ,noswop
1280          INC   B
1290          LD    A,(len1)
1300          CP    B
1310          JR    Z,noswop
1320          LD    A,(len2)
1330          CP    B
1340          JR    Z,swop
1350          INC   DE
1360          INC   HL
1370          JR    NZ,comp
1380 ;Initialise byte counter
1390 ;set swop flag
1400 swop:    LD    A,(bytes)
1410          LD    B,A
1420          LD    (flag),A
1430 ;Swop string descriptors for
1440 ;whole dimension a byte at a time
1450          LD    DE,(point1)
1460          LD    HL,(point2)
1470 loop2:   LD    A,(DE)
```

```
1480          LD    C,(HL)
1490          EX    DE,HL
1500          LD    (HL),C
1510          LD    (DE),A
1520          INC   DE
1530          INC   HL
1540          DJNZ  loop2
1550 ;Add bytes to each of
1560 ;point1 and point2
1570 noswop:  LD    DE,(bytes)
1580          LD    HL,(point1)
1590          ADD   HL,DE
1600          LD    (point1),HL
1610          LD    HL,(point2)
1620          ADD   HL,DE
1630          LD    (point2),HL
1640 ;Increment count
1650          LD    DE,(count)
1660          INC   DE
1670          LD    (count),DE
1680 ;Compare count to cycles
1690 ;if not equal jump to inner
1700          LD    HL,(cycles)
1710          SBC   HL,DE
1720          JP    NZ,inner
1730 ;Jump to fclear
1740 ;if swop flag is clear
1750          LD    A,(flag)
1760          CP    0
1770          JR    Z,fclear
1780 ;Decrement cycles and
1790 ;jump to mid if not zero
1800          LD    HL,(cycles)
1810          DEC   HL
1820          LD    (cycles),HL
1830          LD    A,H
1840          OR    L
1850          JP    NZ,mid
1860 ;Decrement power and
1870 ;jump to outer if not zero
1880 fclear:  LD    HL,power
1890          DEC   (HL)
```

```
1900            JP    NZ,outer
1910            RET
```

Program 5.13. Machine code sort of a rectangular string array.

Program 5.14 is a test program produced to set up a random file of 3 field records for testing out Program 5.13. The 3 fields are numbered field zero, field 1 and field 2 respectively and the array is dimensioned DIM (FIELDS%,RECORDS%), where FIELDS% is set to the value 2. If this appears strange, see the remarks earlier in this chapter concerning the use of rectangular arrays. You are asked for the number of records, and which field (Ø to 2) is to be the subject of the sort. As before, the routine can be called not only from the BASIC test program but from any other program by using:

CALL &998Ø,RECORDS%,FIELDS%,FIELDNUM%,
@A$(Ø,1)

The parameters required are:

(1) The execution address. This is &998Ø if the source code is assembled as set by the EQU directive in line 3Ø of Program 5.13.
(2) RECORDS%, which can either be the maximum number of records DIMensioned in the file (assuming the file is full) or the current number of records present in the array.
(3) FIELDS%, the number of fields minus one or the value to which the fields are DIMensioned.
(4) FIELDNUM%, the sorting field number (Ø to n−1) where n is the total number of fields. The routine can sort records with up to 85 fields which is a limit well above the demands of most files!
(5) The @A$(Ø,1) term is the address of the first used array element – that is to say, the first used field of the first record. The zero elements in the record dimension are kept for field headings and thus shuld not be included in the sort.

```
10 REM TEST PROGRAM
20 REM MACHINE CODE SORT OF A
30 REM RECTANGULAR STRING ARRAY
40 CLS
50 PRINT"Place object code tape in DATAC
ORDER"
60 OPENOUT "buffer"
70 MEMORY &997F
80 CLOSEOUT
```

```
90 LOAD"SORT",&9980
100 CLS
110 INPUT"Sort how many 3 field records"
;RECORDS%
120 FIELDS%=2: REM 3 fields (0,1 & 2)
130 INPUT"Sort which field (0-2)";FIELDN
UM%
140 IF FIELDNUM%<0 OR FIELDNUM%>2 THEN 1
30
150 PRINT
160 REM FILL AND DISPLAY RANDOM ARRAY
170 DIM A$(FIELDS%,RECORDS%)
180 FOR R%=1 TO RECORDS%
190 FOR F%=0 TO FIELDS%
200 B$=""
210 A%=6*RND+1
220 FOR Z%=1 TO A%
230 N%=25*RND
240 K$=CHR$(N%+65)
250 B$=B$+K$
260 NEXT
270 A$(F%,R%)=B$
280 PRINT A$(F%,R%),
290 NEXT
300 NEXT
310 PRINT:PRINT
320 PRINT"SORTING ARRAY"
330 PRINT:PRINT
340 START=TIME/300
350 CALL &9980,RECORDS%,FIELDS%,FIELDNUM
%,@A$(0,1)
360 T=TIME/300-START
370 FOR R%=1 TO RECORDS%
380 FOR F%=0 TO FIELDS%
390 PRINT A$(F%,R%),
400 NEXT
410 NEXT
420 PRINT
430 PRINT"RECORDS SORTED=";RECORDS%
440 PRINT
450 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
460 PRINT
```

```
470 INPUT"Another test (Y/N)";K$
480 K$=UPPER$(K$)
490 IF K$="Y" THEN ERASE A$:GOTO 100 ELS
E END
```

Program 5.14. Test program for the machine code rectangular array sort.

Producing an object code file of Program 5.13

The source code assembles its object code where we instruct it to. In the listing this happens to be &998Ø onwards as set in line 3Ø. We now need a copy of this object code on tape for use by any other program. Again there are two methods, depending on whether or not an assembler is available. Brief notes on how to perform this are set out below. For more general instructions and comments refer back to the STRINGSORT example above.

Method 1: Using the HiSoft DEVPAC Assembler

(a) Load the Assembler at, say, &65ØØ decimal.
(b) Type in Program 5.13.
(c) Save the source code file by typing:

P 1Ø,191Ø,<filename>

(d) Assemble the source code file by entering the assembler command A. When asked for table size, respond with 5ØØ.
(e) Save the object code on tape by typing the command:

O,,SORT

(f) Perform a hard reset then type in Program 5.14. This program will load, run and test the object code file you have just produced.

Method 2: No assembler required

Simply type in and RUN Program 5.15. This program produces an object code file named 'SORT' directly. Unfortunately, the machine code generated in this way cannot be relocated and thus executes only at address &998Ø.

```
10 REM PRODUCING A MACHINE CODE FILE
20 REM RECTANGULAR STRING ARRAY SORT
30 REM (no assembler needed)
40 CLS
50 ADDRESS%=&9980
60 FOR N%=0 TO &11B
```

```
70 READ BYTE$
80 POKE ADDRESS%+N%,VAL("&"+BYTE$)
90 NEXT
100 PRINT"Now produce an object code fil
e on tape"
110 SAVE "SORT",B,&9980,&11C
120 END
130 DATA 21,00,00,22,A4,9A,22,B5
140 DATA 9A,23,22,B3,9A,DD,6E,00
150 DATA DD,66,01,22,9F,9A,DD,7E
160 DATA 02,47,CB,27,80,32,B5,9A
170 DATA DD,7E,04,3C,47,CB,27,80
180 DATA 32,A4,9A,DD,6E,06,DD,66
190 DATA 07,22,9D,9A,EB,97,3C,2A
200 DATA B3,9A,29,22,B3,9A,ED,52
210 DATA 38,F4,32,A1,9A,ED,5B,B3
220 DATA 9A,CB,3A,CB,1B,ED,53,B3
230 DATA 9A,2A,9D,9A,ED,52,22,A2
240 DATA 9A,97,32,A6,9A,32,A7,9A
250 DATA 32,A8,9A,2A,9F,9A,22,A9
260 DATA 9A,3A,A4,9A,47,21,00,00
270 DATA ED,5B,B3,9A,19,10,FD,ED
280 DATA 5B,A9,9A,19,22,AB,9A,ED
290 DATA 5B,B5,9A,2A,A9,9A,19,7E
300 DATA 32,AD,9A,23,7E,32,AF,9A
310 DATA 23,7E,32,B0,9A,2A,AB,9A
320 DATA 19,7E,32,AE,9A,23,7E,32
330 DATA B1,9A,23,7E,32,B2,9A,06
340 DATA 00,ED,5B,B1,9A,2A,AF,9A
350 DATA 1A,BE,38,13,20,28,04,3A
360 DATA AD,9A,B8,28,21,3A,AE,9A
370 DATA B8,28,04,13,23,20,E9,3A
380 DATA A4,9A,47,32,A8,9A,ED,5B
390 DATA A9,9A,2A,AB,9A,1A,4E,EB
400 DATA 71,12,13,23,10,F7,ED,5B
410 DATA A4,9A,2A,A9,9A,19,22,A9
420 DATA 9A,2A,AB,9A,19,22,AB,9A
430 DATA ED,5B,A6,9A,13,ED,53,A6
440 DATA 9A,2A,A2,9A,ED,52,C2,FF
450 DATA 99,3A,A8,9A,FE,00,28,0C
460 DATA 2A,A2,9A,2B,22,A2,9A,7C
470 DATA B5,C2,D9,99,21,A1,9A,35
480 DATA C2,C5,99,C9
```

Program 5.15. Hex loading program for the machine code rectangular string array sort.

For an idea of the sorting times to be expected for various numbers of randomly generated three-field records, see Table 5.11.

Table 5.11. Typical execution times for Program 5.13.

Array size	Typical sorting time
100	0.55 seconds
200	1.70 seconds
300	3.21 seconds
500	7.45 seconds

Upgrading the filing system of Chapter 4

The complete filing system program previously described in Chapter 4 would greatly benefit from the increased speed of the following machine code sorting subroutine. The few modifications required are given below:

(1) Replace the existing SORT FILE subroutine of Program 4.1 (lines 148Ø to 162Ø inclusive) with the lines given below:

```
1480 REM SORT FILE
1490 GOSUB 2480
1500 IF L%<2 THEN 1520
1510 CALL &9980,L%,fields%,F%,@A$(0,1)
1520 RETURN
```

Line 15ØØ skips the CALL if less than 2 records are in the file. Line 151Ø calls the machine code routine and passes over the relevant variables from Program 4.1.

(2) Replace the first 4 lines of Program 4.1 (lines 1Ø to 4Ø inclusive) with the following 7 lines:

```
10 REM CASSETTE MULTIFILING SYSTEM
15 REM WITH MACHINE CODE SORT ROUTINE
20 OPENOUT "buffer"
25 MEMORY &997F
30 CLOSEOUT
35 CLS:PRINT"Now load machine code sort
routine"
40 LOAD "SORT",&9980
```

Lines 2Ø to 3Ø set HIMEM to &997F thus reserving a generous 512 bytes for the machine code subroutine. At the same time, the cassette buffers are permanently allocated. The machine code file is loaded in line 4Ø.

It is convenient to ensure that a copy of the object code file is placed on tape immediately after the copy of Program 4.1. This enables Program 4.1 to load the code without the user needing to swap cassettes in the Datacorder.

Using string arrays as data files

A couple of points, concerned with sorting, need to be stressed when using string arrays for data manipulation.

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. Secondly, if monetary values are represented in fields it may be necessary to store $28 as $ØØØ28.ØØ in order to sort them. This is because other entries under the same field heading may contain odd cents.

Machine code sort of multifield, fixed length records

There are two main ways of generating RAM-resident multifield records. The most common is the one we have just treated, the two-dimensional array, where the records occupy one dimension and the fields the other dimension. A totally different method is where the entire field is stored as one string array in which each field/record is of fixed length. The length of the record strings and field substrings are set to the maximum length likely to be used. Any unused portions of fields/records are then padded with spaces. For example, Fig. 5.2 shows the field headings and a pair of records of a simple file padded out with spaces. The start position of each field is at a fixed point along the string for each record.

This technique is sometimes more economical in terms of memory storage since only one string descriptor per record need be set up,

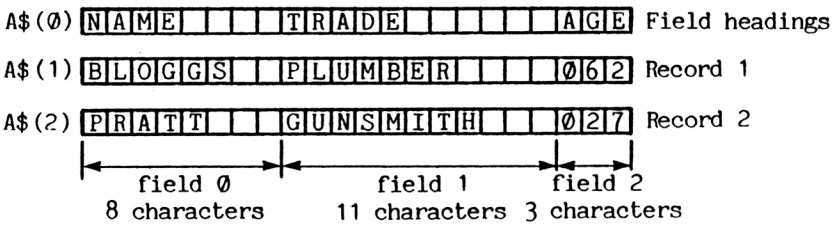


Fig. 5.2. Pair of multifield fixed length records.

whereas in the two-dimensional array, one descriptor is required for each field of a record. The main disadvantages of this method, however, are the restriction of 255 characters per record and the wastage of RAM used for padding out the shorter field entries with spaces.

```
 10 ;DIMINISHING INCREMENT SORT OF
 20 ;MULTIFIELD FIXED LENGTH RECORDS
 30 begin:   EQU  #9980
 40 top:     EQU  begin+#F5
 50 number:  EQU  top
 60 array:   EQU  top+2
 70 power:   EQU  top+4
 80 cycles:  EQU  top+5
 90 count:   EQU  top+7
100 flag:    EQU  top+9
110 point1:  EQU  top+10
120 point2:  EQU  top+12
130 length:  EQU  top+14
140 start:   EQU  top+15
150 addr1:   EQU  top+16
160 addr2:   EQU  top+18
170 size:    EQU  top+20
180          ORG  begin
190 ;Initialise
200          LD   HL,1
210          LD   (size),HL
220 ;Pick up and store
230 ;array start address
240          LD   L,(IX)
250          LD   H,(IX+1)
260          LD   (array),HL
270 ;Pick up and store
280 ;field length in length
290          LD   A,(IX+2)
```

```
300          LD    (length),A
310 ;Pick up field start position
320 ;decrement and store in start
330          LD    A,(IX+4)
340          DEC   A
350          LD    (start),A
360 ;Pick up number of records
370 ;and store in number
380          LD    L,(IX+6)
390          LD    H,(IX+7)
400          LD    (number),HL
410 ;Find next power of 2 >= number
420 ;and store in size
430          EX    DE,HL
440          SUB   A
450 loop:    INC   A
460          LD    HL,(size)
470          ADD   HL,HL
480          LD    (size),HL
490          SBC   HL,DE
500          JR    C,loop
510 ;Store corresponding
520 ;index in power
530          LD    (power),A
540 ;Divide size by 2
550 outer:   LD    DE,(size)
560          SRL   D
570          RR    E
580          LD    (size),DE
590 ;Subtract size from number
600 ;and store in cycles
610          LD    HL,(number)
620          SBC   HL,DE
630          LD    (cycles),HL
640 ;Initialise count and swop flag
650 mid:     SUB   A
660          LD    (count),A
670          LD    (count+1),A
680          LD    (flag),A
690 ;Store array start address
700 ;in point1
710          LD    HL,(array)
720          LD    (point1),HL
```

```
730 ;Multiply size by 3 add point1
740 ;store result in point2
750         LD   HL,(size)
760         LD   D,H
770         LD   E,L
780         ADD  HL,HL
790         ADD  HL,DE
800         LD   DE,(point1)
810         ADD  HL,DE
820         LD   (point2),HL
830 ;Get addresses of strings
840 ;to be compared
850 inner:  LD   HL,(point1)
860         INC  HL
870         LD   A,(HL)
880         LD   (addr1),A
890         INC  HL
900         LD   A,(HL)
910         LD   (addr1+1),A
920         LD   HL,(point2)
930         INC  HL
940         LD   A,(HL)
950         LD   (addr2),A
960         INC  HL
970         LD   A,(HL)
980         LD   (addr2+1),A
990 ;Add start to each of string
1000 ;character start addresses
1010        LD   A,(start)
1020        LD   C,A
1030        LD   B,0
1040        LD   HL,(addr2)
1050        ADD  HL,BC
1060        EX   DE,HL
1070        LD   HL,(addr1)
1080        ADD  HL,BC
1090 ;Compare substrings a character
1100 ;at a time branch swop or noswop
1110 ;finish when length is reached
1120 comp:  LD   A,(DE)
1130        CP   (HL)
1140        JR   C,swop
1150        JR   NZ,noswop
```

```
1160          INC   B
1170          LD    A,(length)
1180          CP    B
1190          JR    Z,noswop
1200          INC   DE
1210          INC   HL
1220          JR    NZ,comp
1230 ;Initialise byte counter
1240 ;set swop flag
1250 swop:    LD    B,3
1260          LD    (flag),A
1270          LD    DE,(point1)
1280          LD    HL,(point2)
1290 ;Swop string descriptors
1300 ;a byte at a time
1310 loop2:   LD    A,(DE)
1320          LD    C,(HL)
1330          EX    DE,HL
1340          LD    (HL),C
1350          LD    (DE),A
1360          INC   DE
1370          INC   HL
1380          DJNZ  loop2
1390 ;Add 3 to each of
1400 ;point1 and point2
1410 noswop:  LD    DE,3
1420          LD    HL,(point1)
1430          ADD   HL,DE
1440          LD    (point1),HL
1450          LD    HL,(point2)
1460          ADD   HL,DE
1470          LD    (point2),HL
1480 ;Increment count
1490          LD    DE,(count)
1500          INC   DE
1510          LD    (count),DE
1520 ;Compare count to cycles
1530 ;if not equal jump to inner
1540          LD    HL,(cycles)
1550          SBC   HL,DE
1560          JP    NZ,inner
1570 ;Jump to fclear
1580 ;if swop flag is clear
```

```
1590          LD    A,(flag)
1600          CP    0
1610          JR    Z,fclear
1620 ;Decrement cycles and
1630 ;jump to mid if not zero
1640          LD    HL,(cycles)
1650          DEC   HL
1660          LD    (cycles),HL
1670          LD    A,H
1680          OR    L
1690          JP    NZ,mid
1700 ;Decrement power and
1710 ;jump to outer if not zero
1720 fclear:  LD    HL,power
1730          DEC   (HL)
1740          JP    NZ,outer
1750          RET
```

Program 5.16. Diminishing increment sort of multifield fixed length fields/records.

The overall structure is similar to that of previous routines with the extra coding 'remarked' on the listing. The machine code can be executed from any program by means of the CALL statement

CALL &998Ø,NUMBER%,start%,fieldlength%,@A$(1)

where

the &998Ø term is the execution address of the routine;
NUMBER%=the current number of records in the array;
start%=the character position of the field by which the records are to be sorted. The permissible range is 1 to 255;
fieldlength% = the number of characters in the field by which the records are to be sorted.
@A$(1)= the first used element in the array. The zero element can be left for headings and excluded from the sort.

Program 5.17 is a test program that produces a number of random, 20-character strings. The string array can be sorted by any fixed length substring starting at any fixed character position. This mimics the sorting of a file structured on the lines of Fig. 5.2. To make the sorted list easier to decipher, the test program inserts spaces between the chosen field limits.

```
10 REM SORT TEST PROGRAM
20 REM MULTIFIELD FIXED LENGTH RECORDS
30 CLS
40 PRINT"Place object code tape in DATAC
ORDER"
50 OPENOUT "buffer"
60 MEMORY &997F
70 CLOSEOUT
80 LOAD"MFSORT",&9980
90 CLS
100 length%=20
110 INPUT"Sort how many 20 character rec
ords";NUMBER%
120 INPUT"Enter field START position (1-
20)";start%
130 IF start%<1 OR start%>length% THEN 1
20
140 INPUT"Enter field LENGTH (1-20)";fie
ldlength%
150 IF fieldlength%<1 OR start%+fieldlen
gth%>length%+1 THEN 140
160 PRINT
170 REM FILL AND DISPLAY RANDOM ARRAY
180 DIM A$(NUMBER%)
190 FOR R%=0 TO NUMBER%-1
200 B$=""
210 FOR Z%=1 TO length%
220 N%=25*RND
230 K$=CHR$(N%+65)
240 B$=B$+K$
250 NEXT
260 A$(R%)=B$
270 PRINT A$(R%)
280 NEXT
290 PRINT:PRINT
300 PRINT"SORTING ARRAY"
310 PRINT:PRINT
320 START=TIME/300
330 CALL &9980,NUMBER%,start%,fieldlengt
h%,@A$(0)
340 T=TIME/300-START
350 FOR R%=0 TO NUMBER%-1
360 PRINT LEFT$(A$(R%),start%-1);SPC(1);
```

```
MID$(A$(R%),start%,fieldlength%);SPC(1);
MID$(A$(R%),start%+fieldlength%,LEN(A$(R
%)))
370 NEXT
380 PRINT
390 PRINT"RECORDS SORTED=";NUMBER%
400 PRINT
410 PRINT"SORTING TIME=";ROUND(T,2);"SEC
ONDS"
420 PRINT
430 INPUT"Another test (Y/N)";K$
440 K$=UPPER$(K$)
450 IF K$="Y" THEN ERASE A$:GOTO 90 ELSE
 END
```

Program 5.17. Test program for the multifield fixed length string sort.

Producing an object code file of Program 5.16

As in the previous examples, there are two ways of producing the object code file needed for use by another program. These are outlined below, but refer back to the description of the simple machine code string sort to cover the general details.

Method 1: Using the HiSoft DEVPAC assembler

(a) Load the assembler when prompted at, say, 65ØØ decimal.
(b) Type in Program 5.16 as listed.
(c) Save the source code on tape by typing:

P 1Ø,175Ø,<filename>

(d) Assemble the source code by entering the command A. When asked for table size, respond with 5ØØ.
(e) Save a copy of the object code on tape by typing the assembler command:

O,,MFSORT

(f) Perform a hard reset, then type in Program 5.17. This will load, run and test the object code file produced.

Method 2: No assembler required

To produce an object code file that loads and executes at &998Ø simply type in Program 5.18 and RUN it.

```
10 REM PRODUCING A MACHINE CODE FILE
20 REM SORT STRING ARRAY OF
30 REM FIXED LENGTH RECORDS
40 REM (no assembler needed)
50 CLS
60 ADDRESS%=&9980
70 FOR N%=0 TO &F4
80 READ BYTE$
90 POKE ADDRESS%+N%,VAL("&"+BYTE$)
100 NEXT
110 PRINT"Now produce an object code fil
e on tape"
120 SAVE "MFSORT",B,&9980,&F5
130 END
140 DATA 21,01,00,22,89,9A,DD,6E
150 DATA 00,DD,66,01,22,77,9A,DD
160 DATA 7E,02,32,83,9A,DD,7E,04
170 DATA 3D,32,84,9A,DD,6E,06,DD
180 DATA 66,07,22,75,9A,EB,97,3C
190 DATA 2A,89,9A,29,22,89,9A,ED
200 DATA 52,38,F4,32,79,9A,ED,5B
210 DATA 89,9A,CB,3A,CB,1B,ED,53
220 DATA 89,9A,2A,75,9A,ED,52,22
230 DATA 7A,9A,97,32,7C,9A,32,7D
240 DATA 9A,32,7E,9A,2A,77,9A,22
250 DATA 7F,9A,2A,89,9A,54,5D,29
260 DATA 19,ED,5B,7F,9A,19,22,81
270 DATA 9A,2A,7F,9A,23,7E,32,85
280 DATA 9A,23,7E,32,86,9A,2A,81
290 DATA 9A,23,7E,32,87,9A,23,7E
300 DATA 32,88,9A,3A,84,9A,4F,06
310 DATA 00,2A,87,9A,09,EB,2A,85
320 DATA 9A,09,1A,BE,38,0D,20,20
330 DATA 04,3A,83,9A,B8,28,19,13
340 DATA 23,20,EF,06,03,32,7E,9A
350 DATA ED,5B,7F,9A,2A,81,9A,1A
360 DATA 4E,EB,71,12,13,23,10,F7
370 DATA 11,03,00,2A,7F,9A,19,22
380 DATA 7F,9A,2A,81,9A,19,22,81
390 DATA 9A,ED,5B,7C,9A,13,ED,53
400 DATA 7C,9A,2A,7A,9A,ED,52,C2
410 DATA E9,99,3A,7E,9A,FE,00,28
```

```
420 DATA 0C,2A,7A,9A,2B,22,7A,9A
430 DATA 7C,B5,C2,CA,99,21,79,9A
440 DATA 35,C2,B6,99,C9
```

Program 5.18. Hex loading program of the object code of Program 5.16.

Searching arrays

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. Although integer arrays are used in the examples, they can easily be converted for use with string arrays.

Sequential or linear search

This is the search algorithm most widely used and involves starting from the beginning of a list and sequentially 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 in the previous chapter. Program 5.19 is an uncluttered demonstration program of this simple technique. The subroutine, starting at line 1ØØØ, 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, 1ØØØ elements and see how long it takes either to find or to determine that the specified integer entered into item%, in line 1ØØ, is not present in the array. Compare the search times with that of the more efficient binary search given in Program 5.20.

```
10 REM LINEAR SEARCH DEMONSTRATION
20 CLS
30 INPUT"How many numbers in list";NUMBE
R%
40 DIM A%(NUMBER%)
50 FOR N%=1 TO NUMBER%
60 A%(N%)=N%*2-1
70 PRINT A%(N%)
80 NEXT
90 PRINT
100 INPUT"Search for ";item%
```

```
110 START=TIME/300
120 GOSUB 1000
130 T=TIME/300-START
140 PRINT"Searching time=";ROUND(T,2);"S
econds"
150 INPUT"SEARCH AGAIN (Y/N)";K$
160 K$=UPPER$(K$)
170 IF K$="Y" THEN 90
180 END
190 '
200 '
999 REM SEQUENTIAL SEARCH SUBROUTINE
1000 J%=1
1010 WHILE J%<NUMBER% AND item%<>A%(J%)
1020 J%=J%+1
1030 WEND
1040 IF item%=A%(J%) THEN PRINT"Item fou
nd at array position";J% ELSE PRINT"Item
 not found"
1050 RETURN
```

Program 5.19. Simple sequential search demonstration.

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.

The binary search

A much faster method of searching for a key field, provided the records are first sorted into order, is called the 'binary' search. Program 5.20 is a binary search demonstration. With the exception of the search subroutine at line 1ØØØ, it is similar in form to the previous one.

```
10 REM BINARY SEARCH DEMONSTRATION
20 CLS
30 INPUT"How many numbers in list";NUMBE
R%
40 DIM A%(NUMBER%)
50 FOR N%=1 TO NUMBER%
60 A%(N%)=N%*2-1
70 PRINT A%(N%)
80 NEXT
```

```
90 PRINT
100 INPUT"Search for ";item%
110 START=TIME/300
120 GOSUB 1000
130 T=TIME/300-START
140 PRINT"Searching time=";ROUND(T,2);"S
econds"
150 INPUT"SEARCH AGAIN (Y/N)";K$
160 K$=UPPER$(K$)
170 IF K$="Y" THEN 90
180 END
190 '
200 '
999 REM BINARY SEARCH SUBROUTINE
1000 low%=1:high%=NUMBER%
1010 WHILE high%>low%
1020 mid%=(low%+high%)\2
1030 IF item%>A%(mid%) THEN low%=mid%+1
ELSE high%=mid%
1040 WEND
1050 IF item%=A%(high%) THEN PRINT"Item
found at array position";high% ELSE PRIN
T"Item not found or array not in order"
1060 RETURN
```

Program 5.20. Binary search demonstration.

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 this 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:

Average number of comparisons= n/2= 10000/2= 500

(b) Binary search:

Average number of comparisons=$\log_2(n)$=$\log_2(1000)$=10 (when rounded)

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 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 mixed into larger sets until a single sorted set remains.
5. The diminishing increment sort (version 2) is more efficient still than version 1.
6. The Quicksort is fast on average but the performance can deteriorate if the array is ordered or near-ordered.
7. The most convenient sort algorithm for machine coding is the simple diminishing increment sort (version 1).

8. Understanding of machine code string sorting is made easier if the details of the string descriptors are known.
9. Records can be stored in two-dimensional array form or, providing the fields/records are all of fixed length, as one simple string array.
10. A particular record in a file can be located by either a simple sequential search or, if in order, a binary search.
11. 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 takes 1 millisecond, calculate the approximate sorting time to the nearest hour.

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

5.4 State a possible disadvantage of the Quicksort.

5.5 In the Shellsort, Program 5.5, why is the constant (2) added in line 1Ø2Ø?

Chapter Six
Knowledge Testing

Introduction

Some readers may feel that an entire chapter devoted to knowledge testing is out of place in a book dedicated to filing systems. Nevertheless, programs which facilitate the composing of questions, their subsequent storage as a data file and their eventual retrieval in a form suitable for student self-testing is, in essence, a filing system which has many applications. The programs which follow are concerned with the so-called *multiple choice* testing method.

Studying any subject, whether it is gardening or astrophysics, involves memorising a number of facts and then learning how to use them. Some modern educationalists tend to despise mere facts on the grounds that they can always be looked up in a suitable reference book as needed. This is excellent advice providing it is not taken too seriously. It is very difficult to assess the relative worth of contradictory arguments if you always have to be within reach of the *Encyclopaedia Britannica*. In any case, most courses of study culminate in an examination of some form and there are not many authorities at the moment who are sufficiently enlightened to let students enter the room carrying armfuls of reference material. The trouble nowadays is the sheer number of facts which must be carried in the head before you can begin serious study of any subject.

Progress testing

Some form of *feedback* is essential in any learning process, whether it be the traditional sarcasm of a tutor, the regular progress test in the classroom or from another member of the family or group. Without periodic tests for progress, it is difficult to assess whether or not there has been any. The question arises as to the form the progress test is to take. If the test is in the home, then a common practice is to ask for a volunteer from the family to conduct the test by asking questions

plucked at random from current textbooks and see if the answers given correspond. This method has some value but can, at times, lead to a certain amount of hostility because

(a) The parent may have difficulty in posing reasonable questions even with the aid of the textbook;
(b) The answer given may be right but, because expressed in words slightly different to the textbook version, is pronounced wrong. (The arguments which follow often result in the abandonment of the test!);
(c) The volunteer, because the textbook is to hand, may gradually acquire an air of superiority which can irritate the learner. Criticism is often accepted in good grace from an outsider but resented from a member of the family or group.

Mutliple choice tests

Whether it is justifiable or not, most people view the computer as an omnipotent machine untainted by personal malice and completely objective in all dealings with humans. Progress tests conducted via a keyboard and VDU screen can take place in a stress-free atmosphere which is conducive to learning. If you give a wrong answer, whilst in the home environment, nobody need know except the computer. A poor score, displayed at the end of the test, can be a secret between you and the computer and one which will probably stimulate a desire to keep repeating the test over and over again until the computer is 'beaten'. An innate desire to beat the computer, so evident in the games addict, can thus be turned to good effect.

Although there are many ways of arranging a progress test on a computer, the multiple choice question with four answers is the most obvious. This kind of test was first used in World War II. It provided a quick method of checking the progress of service personnel on training courses. Traditional examination papers demanded a skilled marking team and took too long. The multiple choice paper could be marked by untrained personnel using only a stencil placed over the answer sheet. For some time after the war, the educational establishments despised the system, considering it a cheap expedient suitable only for marking low grade students under training. (The Establishment has always tended to consider 'training' as inferior to 'education'.) However, due to the advance of technology in the post-war years, the attitude towards the multiple choice test changed and, eventually, even the universities began to accept it as a useful adjunct

to conventional testing. The Open University, in particular, favours this kind of test.

Mechanics of the multiple choice test

The test consists of a series of questions, each with at least four answers A, B, C, and D. The person taking the test puts a cross against the answer which he or she considers is the 'best' one. In the more sophisticated questions, three of the answers may be superficially correct but only one is true in all respects. The following is a straightforward example:

Question: What is the capital of Hungary?
Answer A: Sofia
Answer B: Budapest
Answer C: Warsaw
Answer D: Bucharest

It is not good practice to include one answer which is absurd because it effectively reduces the choice to one of three. We should remember that some degree of guesswork is inevitable during an answering session. In the above example, all four have an east European ring about them so they are all possibles, but to have included New York as an answer would almost certainly be rejected. Much of the antagonism to the multiple choice paper in the early days was the tendency for those preparing the paper to indulge in cheap word tricks. They tried to express the four answers in a way which lured the poor trainee into placing a cross against the wrong answer.

The multiple choice method of testing is used extensively for shortlisting applicants for employment. Conducting personnel interviews is an expensive business, particularly if the post demands technical or business management knowledge. A provisional multiple choice paper eliminates the bulk of the non-runners. Because of this, it is good for all concerned, children as well as adults, to get used to the system. It is one thing to know a subject but quite another when you have to face a question paper in multiple choice format for the first time and under conditions of stress.

The computer display

The Amstrad CPC464 is ideally suited for multiple choice questioning because of the 80-column format in Mode 2 and the presence of the customised video monitor at no extra cost. Several popular computers offer an 80-column format but the display on a TV, particularly if it's a colour set, is almost unreadable. The

Amstrad 80-column display is *excellent on the monochrome monitor* and perfectly legible even on the colour monitor. Of course, it is not essential to choose an 80-column display but there are problems if we try to make do with only 40 columns. It is self-evident that the screen must be able to display the question, the four answers and some form of prompting message. With only 40 columns and, say, 25 lines to play with, it is very difficult to compose thought-provoking questions and answers sufficiently concise to fit into one screen page. With 25 lines, each with 80 columns, there is room for at least 256 characters for the question and 256 each for the four answers, a generous allowance and ample for even the most searching question.

Programming methods

There is a variety of ways in which the test questions can be programmed. A simple form of program could be devised in which the questions and answers are contained in DATA statements. Indeed, programs based on this principle have appeared in one or two of the monthly magazines but, apart from simplicity, they have little to recommend them. The snags are obvious:

(a) The person taking the test can easily LIST the program and examine the DATA to find the right answer.
(b) Unless the person setting the test has some computer experience there will be difficulties in setting up the DATA lines.

The first consideration is to design in user-friendliness. It must be easy for the person to enter the questions as well as for the person who will eventually have to sit the test. It is hoped that the programs which follow will encourage members of a family to construct fresh questions or even modify the programs to suit them.

The most sensible method is to split the task into two, one program for compiling the questions and a separate program for answering the questions. We shall call these programs 'MC Prep' and 'MC Test' respectively. Two such programs are listed later in this chapter together with operating details. It is convenient in the meantime to study some of the options which we have included and to suggest how some of them might be adapted or enlarged to cover individual requirements.

Multiple Choice File Preparation (Program 6.1)

The end result of the program is the production of a *data tape* bearing

the questions and answers together with certain leading particulars relating to the test. It is essential to have a blank tape ready in the cassette drive before running the program. The data tape must be saved for subsequent use in the second program, Multiple Choice Test (Program 6.2).

Operating details

The program is menu-driven. When the program is initially RUN, the menu page is displayed. The top half gives certain leading information and the menu bottom half shows the available options. Until the leading particulars have been entered, the menu page has the following appearance:

```
Multiple Choice Preparation

Subject          :
Test             :
Author           :
Date             :
Ref              :
Time (minutes)   :
File size        :  ∅ questions

1 Prepare file
2 Load file
3 Save file
4 View file
5 Add questions
6 Modify questions
7 Delete question
8 Exit file

Select option:
```

Option 1: Prepare file

This option should be used when preparing a fresh set of questions in order to gather leading particulars from the composer of the questions. Until this is done, the file size (number of questions) remains at ∅. The maximum number of questions has been limited to a hundred but no input is requested for file size since this is

automatically updated as questions are added. The last item requested is the time allowed (in minutes) which the composer of the questions considers reasonable for answering them. After the leading particulars have been entered, the menu page is replaced by the Add Question display. This starts with the following header information:

MC Preparation Bytes free: xxxxx Question number 1

The first prompt asks for the question, the next four for the answers. The last prompt asks which of the four answers A, B, C or D is correct. As each question is entered, the heading information is updated. Knowledge of the remaining bytes is important to the operator to avoid running out of memory. Although the number of questions is limited to a hundred, it is possible to exhaust memory before this is reached if there is an abundance of text in each question. It is important to remember that if a question or answer occupies more than one line, *the ENTER key must not be used to turn the corner to the next line*, because this will terminate the entry. The limit imposed by the Amstrad operating system on the number of characters in a string variable held in RAM is 255, but the program imposes a 254 limit because we have found that data cassette files can sometimes reject the full 255 characters in a string. No question or answer must exceed this or a warning bleep is given and the excess refused. However, it is most unlikely that a question, and even less likely an answer, would consume 254 characters. The majority of questions will rarely exceed 80 characters. As a conservative round-figure calculation, assume each question and each answer takes 100 characters, making a total of 500 bytes, say ½K of memory per question. Thus there is ample room for a file to hold 50 questions because only 25K will be used up, still leaving a comfortable margin.

After the first question and answers have been entered, the program reverts back to the menu. Any existing file resident in RAM is destroyed by Option 1.

Option 5: Add questions

Once the file has been prepared, and Question 1 has been entered, this option is used each time an additional question is to be added to the file. The question number, 'bytes free' and prompts for obtaining the question and answers are the same as described under Option 1 above. The menu is regained after each question, the four answers and the correct answer have been completed so it is necessary to select Option 5 again for each one.

Option 3: Save file

It is not necessary to sit at the keyboard for hours on end until all questions have been entered. Whatever the question number reached, selecting Option 3 can be used to save a partially completed file on tape. Remember to take the program tape out of the cassette and replace it with either a completely blank tape or one that has previously been rewound to a blank area. The only prompt given is for the file name followed by the normal system instructions for pressing RECord and PLAY. The program sets the writing speed to 2000 baud but those of little faith can change back to the normal default speed of 1000 baud.

Option 2: Load file

Providing the tape has been rewound to the correct position and the response to the prompt 'Enter file name' is accepted, the file is loaded. The position in the computer is now exactly the same as it was before that particular file was saved. For example, if the file had been saved when it was complete up to the stage of Question 15, further questions 16 and upwards would be ready to be entered.

Option 4: View file

This option allows the questions in a file to be *stepped through* for examination. The first prompt asks for the question number – which means that the file can be viewed starting from a chosen number. An error message of the form 'Range = x to y' appears if an illegal question number is entered. Irrespective of the first question number, other questions can be brought into view by using the right or left cursor keys. The right cursor advances to the next higher and the left cursor to the next lower question number. The cursor action wraps around at both ends so, for example, if the last question number in the file is 47, the next right cursor action wraps around to question 1 again. To regain the menu, press the space bar.

Option 6: Modify questions

The ability to modify either the question or one or more of the answers is essential. Even if the original questions were typed in correctly, there is always a strong possibility in any multiple choice test that, sooner or later, someone is going to challenge some of them. Either the question is misleading or, more often than not, two of the answers in the same question are judged to be equally correct. For this reason, modification facilities are needed to alter any single part of a particular question number. It should also be possible to alter the

leading particulars in the menu (the Menu Panel). For example, the time which has been allowed for the test. Care has been taken in Option 6 to provide separate access to any of the parts and to make the modification procedure as painless as possible.

On first selecting the option, the display reads:

```
To modify Menu Panel select question ∅
Give question number
```

Thus, if only the leading particulars require modification, you would select question ∅. To modify a question, you enter the appropriate question number.

The next display reads:

```
Modify which part of question?
(1) Question
(2) Answer A
(3) Answer B
(4) Answer C
(5) Answer D
(6) Correct answer A, B, C or D?

Select option:
```

Assuming you select 5 (Answer D), the next display changes to:

```
You can utilise any of the EXISTING TEXT with the
CURSOR/COPY keys
Type EXIT to regain menu
You are modifying field labelled D
-------------------------------------------------
(Existing text displayed here)
-------------------------------------------------
(ENTER NEW TEXT HERE)
```

Once the offending line has been modified, it is redisplayed for a final check. Use the SHIFT and CURSOR/COPY keys to reuse any existing text. To return to the menu, enter EXIT.

Option 7: Delete question
This option simply prompts for the question number to be deleted and closes up the gaps in the file. If the option has been selected by mistake, there is a facility for returning straight to the menu by selecting question ∅. The selection also forces garbage collection of redundant strings held in memory. If there are many questions in the file, this may take some time.

Option 8: Exit program
It is important to use this option, rather than the ESC key, when you have finished with the program because it restores normal default status.

Analysis of Program 6.1

The treatment which follows is related to the listing of Program 6.1.

```
10 REM MULTIPLE CHOICE FILE PREPARATION
20 OPENOUT "buffer"
30 MEMORY &9B7E
40 CLOSEOUT
50 flag%=0
60 DIM A$(100,5),H$(5),P$(5)
70 H$(0)="QUESTION"
80 H$(1)="ANSWER A"
90 H$(2)="ANSWER B"
100 H$(3)="ANSWER C"
110 H$(4)="ANSWER D"
120 H$(5)="CORRECT answer (A,B,C or D)"
130 P$(0)="Subject"
140 P$(1)="Test"
150 P$(2)="Author"
160 P$(3)="Date"
170 P$(4)="Ref"
180 P$(5)="Time (minutes)"
190 BORDER 0
200 MODE 1
210 SEL%=0
220 WHILE SEL%<1 OR SEL%>8
230 GOSUB 2110:GOSUB 2190
240 LOCATE 1,14
250 PRINT"(1)  PREPARE file"
```

```
260 PRINT"(2)  LOAD file"
270 PRINT"(3)  SAVE file"
280 PRINT"(4)  VIEW file"
290 PRINT"(5)  ADD Question"
300 PRINT"(6)  MODIFY Question"
310 PRINT"(7)  DELETE question"
320 PRINT"(8)  EXIT program"
330 LOCATE 1,24
340 PRINT"Select option :";
350 K$=INKEY$:IF K$="" THEN 350
360 SEL%=VAL(K$)
370 WEND
380 CLS
390 IF SEL%>2 AND SIZE%=0 AND SEL%<>8 TH
EN PRINT"No File loaded":GOSUB 1800:GOTO
 200
400 IF SEL%<3 AND flag%=1 THEN GOSUB 185
0
410 ON SEL% GOSUB 470,570,690,810,960,11
20,1460,430
420 GOTO 200
430 SPEED WRITE 0
440 END
450 '
460 REM PREPARE NEW FILE
470 SIZE%=0:K%=22
480 FOR C%=0 TO 5
490 PRINT"Enter ";P$(C%)
500 GOSUB 1950:A$(0,C%)=UPPER$(K$)
510 NEXT
520 GOSUB 960
530 flag%=1
540 RETURN
550 '
560 REM LOAD FILE
570 GOSUB 2050
580 OPENIN filename$
590 INPUT#9,SIZE%
600 FOR Q%=0 TO SIZE%
610 FOR C%=0 TO 5
620 INPUT#9,A$(Q%,C%)
630 NEXT:NEXT
640 CLOSEIN
```

```
650 flag%=1
660 RETURN
670 '
680 REM SAVE FILE
690 GOSUB 2050
700 SPEED WRITE 1
710 OPENOUT filename$
720 PRINT#9,SIZE%
730 FOR Q%=0 TO SIZE%
740 FOR C%=0 TO 5
750 PRINT#9,A$(Q%,C%)
760 NEXT:NEXT
770 CLOSEOUT
780 RETURN
790 '
800 REM VIEW FILE
810 LOW%=1:GOSUB 1730
820 IF Q%<1 THEN Q%=SIZE%
830 IF Q%>SIZE% THEN Q%=1
840 GOSUB 1620
850 FOR C%=0 TO 5
860 PRINT H$(C%);" : "; USING "&";A$(Q%,
C%)
870 NEXT
880 WHILE INKEY(47)<>0
890 CALL &BB18:'KM WAIT KEY
900 IF INKEY(8)=0 THEN Q%=Q%-1:GOTO 820
910 IF INKEY(1)=0 THEN Q%=Q%+1:GOTO 820
920 WEND
930 RETURN
940 '
950 REM ADD QUESTION
960 IF SIZE%>=100 OR FRE(0)<2000 THEN PR
INT"FILE FULL":GOSUB 1800:GOTO 1090
970 SIZE%=SIZE%+1
980 Q%=SIZE%:GOSUB 1620
990 K%=254
1000 FOR C%=0 TO 5
1010 PRINT "Enter ";H$(C%)" : ";
1020 GOSUB 1950:A$(Q%,C%)=K$
1030 X%=POS(#0):Y%=VPOS(#0)
1040 LOCATE 48,2
1050 PRINT USING "#####";FRE(0)
```

```
1060 LOCATE X%,Y%
1070 NEXT
1080 A$(Q%,5)=UPPER$(A$(Q%,5))
1090 RETURN
1100 '
1110 REM MODIFY QUESTION
1120 PRINT"To Modify MENU PANEL select q
uestion 0":LOW%=0:GOSUB 1730
1130 SEL%=0
1140 WHILE SEL%<1 OR SEL%>6
1150 CLS:LOCATE 2,5:PRINT"MODIFY which p
art of QUESTION ";Q%
1160 LOCATE 1,10
1170 FOR C%=1 TO 6
1180 PRINT"(";C%;")  ";
1190 IF Q%=0 THEN PRINT P$(C%-1)
1200 IF Q%>0 THEN PRINT H$(C%-1)
1210 NEXT
1220 LOCATE 1,21:PRINT"Select option"
1230 K$=INKEY$:IF K$="" THEN 1230
1240 SEL%=VAL(K$)
1250 WEND
1260 K$="":WHILE K$<>"EXIT" AND K$<>"exi
t"
1270 GOSUB 1620
1280 PRINT"You can utilise any of the EX
ISTING TEXT with the CURSOR/COPY keys"
1290 PRINT
1300 PRINT"Type EXIT to regain the main
MENU"
1310 PRINT
1320 PRINT"You are MODIFYING the field l
abelled : ";
1330 IF Q%=0 THEN PRINT P$(SEL%-1)
1340 IF Q%>0 THEN PRINT H$(SEL%-1)
1350 PRINT:PRINT:GOSUB 2010
1360 PRINT A$(Q%,SEL%-1)
1370 GOSUB 2010:LOCATE 1,VPOS(#0)+4:GOSU
B 2010
1380 LOCATE 1,VPOS(#0)-5
1390 LINE INPUT K$
1400 IF Q%=0 OR SEL%=6 THEN K$=UPPER$(K$
)
```

```
1410 IF K$<>"EXIT" AND K$<>"exit" THEN A
$(Q%,SEL%-1)=K$
1420 WEND
1430 RETURN
1440 '
1450 REM DELETE QUESTION
1460 PRINT"The DELETE option has been se
lected"
1470 PRINT"To return to MENU select Ques
tion 0"
1480 LOW%=0:GOSUB 1730
1490 IF Q%=0 THEN 1590
1500 PRINT"Wait"
1510 WHILE Q%<SIZE%
1520 FOR C%=0 TO 5
1530 A$(Q%,C%)=A$(Q%+1,C%)
1540 NEXT
1550 Q%=Q%+1
1560 WEND
1570 SIZE%=SIZE%-1
1580 PRINT FRE("")
1590 RETURN
1600 '
1610 REM DISPLAY HEADER
1620 MODE 2
1630 GOSUB 2010
1640 PRINT"MULTIPLE CHOICE PREPARATION"
1650 LOCATE 35,2
1660 PRINT"Bytes free :";FRE(0)
1670 LOCATE 64,2
1680 PRINT"QUESTION :";Q%
1690 GOSUB 2010
1700 PRINT
1710 RETURN
1720 '
1730 PRINT
1740 REM GET QUESTION NUMBER
1750 INPUT"Give Question Number ";Q%
1760 IF Q%<LOW% OR Q%>SIZE% THEN PRINT"R
ANGE ("LOW%;"to";SIZE%")":GOTO 1750
1770 RETURN
1780 '
1790 REM PRESS ANY KEY
```

```
1800 PRINT"PRESS any KEY to CONTINUE"
1810 CALL &BB18:'KM WAIT KEY
1820 RETURN
1830 '
1840 REM BELT & BRACES
1850 PRINT"LOADED FILE AT RISK"
1860 PRINT"Do you wish to RESELECT MENU
(Y/N)";
1870 K$=INKEY$
1880 K$=UPPER$(K$)
1890 IF K$<>"Y" AND K$<>"N" THEN 1870
1900 IF K$="Y" THEN SEL%=0
1910 IF K$="N" THEN flag%=0
1920 RETURN
1930 '
1940 REM GET LINE INPUT
1950 LINE INPUT K$
1960 IF K$="" THEN 1950
1970 IF LEN(K$)>K% THEN K$=LEFT$(K$,K%)
1980 RETURN
1990 '
2000 REM PRINT A LINE
2010 PRINT STRING$(80,CHR$(154));
2020 RETURN
2030 '
2040 REM GET FILENAME
2050 CLS:PRINT"Enter filename"
2060 K%=16:GOSUB 1950
2070 filename$=K$
2080 RETURN
2090 '
2100 REM DISPLAY HEADER 2
2110 CLS
2120 PRINT
2130 INK 2,6
2140 PAPER 2
2150 PRINT" MULTIPLE CHOICE PREPARATION
"
2160 PAPER 0
2170 RETURN
2180 '
2190 REM DISPLAY PANEL
2200 PRINT
```

```
2210 PRINT STRING$(40,CHR$(154));
2220 FOR C%=0 TO 5
2230 PRINT TAB(2) P$(C%) TAB (16) ": "A$
(0,C%)
2240 NEXT
2250 PRINT" File size      :";SIZE%;"QUES
TIONS"
2260 PRINT STRING$(40,CHR$(154));
2270 RETURN
```

Program 6.1. Multiple choice file preparation.

The program divides naturally into two sections. The first, extending down to END at line 44Ø, is the actual 'program'. The remaining lines form a battery of subroutines. The major subroutines, carrying out various menu functions, are called from the 'program' but these, in turn, may call up smaller subroutines which perform general purpose, rather than specialised, functions. The variable names and their corresponding functions are given below:

A$(Q%,C%) = main array for holding questions and answers, dimensioned A$(1ØØ,5).
H$(Ø − 5) = literal prompts for the question and answers.
P$(Ø−5) = literal prompts for heading panel information.
SEL% = chosen option number.
SIZE% = current highest question number in file.
K$ = general purpose global variable.
filename$ = current file name.
Q% = question number.
C% = index to either the question itself, the four answers or the correct answer.
flag% = file status: Ø = file not resident, 1 = file resident.

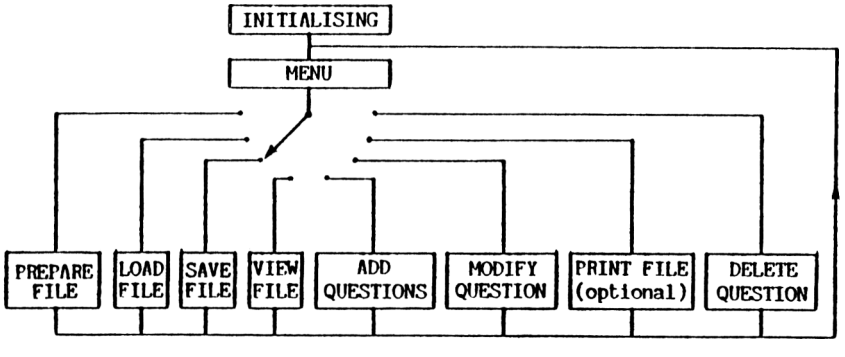


Fig. 6.1. Structure diagram for Program 6.1.

Tracing the subroutine paths

The major subroutines for each option are called from the ON SEL% GOSUB statement at line 5ØØ. They each RETURN to the start of the menu at line 2ØØ, via the GOTO at line 42Ø. The overall structure is shown in Figure 6.1.

Subroutine calls

Some subroutines call up other subroutines. Table 6.1 shows which of them are called for each option.

Table 6.1. Subroutine calls.

Prepare question	Prepare new file
	Get line input
	Add question
Load file	Get file name
Save file	Get file name
View file	Get question number
	Display header
Add question	Get line input
Modify question	Get question number
	Display header
	Print a line
Delete question	Get question number

The first section begins with a routine for reserving a 4K fixed cassette buffer area. In this type of program, it is preferred to the normal dynamic allocation of buffers which cause HIMEM to float up and down. This can be a lengthy and repetitive process because all strings in the 'heap' have to be garbage collected, shifted out to make room for the cassette buffer and, after a load or save, re-positioned again. (Refer back to the paragraph on cassette data blocks in Chapter 1.)

The 40-column Mode 1 is used for the menu page but for preparing and viewing the questions the 80-column Mode 2 is set by the Display header subroutine.

Multiple choice test program

The objective of the last program was the production of a TEST CASSETTE (data file) containing the questions. This one, Program

6.2, uses it to present the questions, accept the answers and display the final score.

Operating the program

A degree of classroom formality is blended into the program, giving a 'supervisor' power to prevent the person taking the test from having more than one attempt. The supervisor should operate the first part of the program as follows:

(1) Load Program 6.2.
(2) Remove the program tape from the cassette drive and have the TEST CASSETTE (data file) to hand.
(3) RUN Program 6.2.

The screen will then read:

```
MULTIPLE CHOICE TESTING
Place TEST CASSETTE in the machine
REWIND tape ready to load DATA
Enter file name
```

When the tape has finished loading, the next prompt is:

```
MEMORISE and ENTER any 6 DIGIT code
```

(Once a code is entered, the program cannot be run again by the person taking the test because the ESCape key is rendered inoperative. If a hard reset is used, the program is lost anyway and only the supervisor has the program tape needed for a re-run.) The last screen prompt, as far as the supervisor is concerned, is for the name of the person taking the test. The computer is then ready for the test to begin.

The student's actions

The first display gives the Subject, Test, Author, Date, Reference and the Time allowed in minutes. This is followed by instructions on which keys to press for answering the questions. The test begins and the time clock starts to count down when the student presses any key to reveal the following (example) display:

Multiple choice: Time ∅1:15:∅5 Question: 1
Question: If N is any integer, which of the following expressions ensures an ODD integer? Answer A: 2N Answer B: 2N+1 Answer C: N+1 Answer D: N/3
No RESPONSE recorded Press A,B,C or D to ANSWER or CHANGE ANSWER

The test starts at question number 1 but the next question does not follow on automatically as each question is answered. The right or left arrow cursor keys must be used to step forward or backward to the next question. Each question is answered by typing A, B, C or D in either lower- or upper-case. The student can change his mind at any time by altering the answer. The message 'No RESPONSE recorded' first appears but once an answer has been received, the message changes to 'Your RESPONSE stored as:' followed by the letter last entered for that question. While the test is progressing, the amount of time left in hours, minutes and seconds, is always visible at the top of the screen together with the current question number. When the time clock eventually counts down to zero, the test ends, irrespective of the number of questions answered. The last 10-second count-down is accompanied by audible beeps which serve as a warning to the student. If the student finishes the test before the allotted time and he/she so wishes, the ESC key can be pressed.

On the count of zero, or pressing the ESC key, the screen displays:

TEST OVER Press any key to obtain results

The test results are then displayed with a format as shown in the following example:

MULTIPLE CHOICE TEST	
Subject	Mathematics
Test	Taylor series
Author	DJ Stephenson
Date	Oct 25 1984
Reference	M/S2/3Ø2
Time (minutes)	37
File size	5Ø questions

RESULTS TABLE: Wharton H	
Total questions	5Ø
Questions attempted	4Ø
Correct answers	2Ø
Incorrect answers	2Ø
Percentage score	4Ø%
Performance grade	(D) Pass

Enter CODE to obtain Menu

Obtaining the menu

Four options are available on completion of the test:

(a) Allow the same student to resit the test.
(b) Allow another student to sit the test.
(c) Load a different set of questions.
(d) Exit the program.

Assuming that the test is formal, the student who has just completed the test will not have access to the menu so the supervisor will take control because only he or she will know the code entered at the start of the test.

As each code character is entered, a high tone is sounded. A low tone is repeated on entry of every sixth character for synchronisation purposes. If the correct code is entered, the menu will be displayed. If not, the low tone is a prompt to indicate that the wrong code has been entered and the entire code must be repeated. The menu page appears as follows:

```
1  Load new question file
2  Resit test (new testee)
3  Resit test (same testee)
4  Exit program (Enable ESC key)

Select option number
```

Informal test

Although the presence of a 'supervisor' has so far been assumed, the test can easily be carried out informally. The person taking the test can key in his or her own code and carry out the test over and over again in order to improve the score. In a free and easy environment, the program can provide a novel driving force for continuous self-improvement. There is, of course, one difficulty – someone must be able to produce the questions in the first place and know how to utilise the facilities of Program 6.1 before Program 6.2 can be used. There are two solutions:

(1) Schools, or more usually technical colleges, professional institutions and universities are often willing to supply previous years' examination papers. Many of these are in multiple choice format although they don't normally supply answers. Nevertheless, there may be someone in the family or circle of friends who may know a few of the answers; even if not, the questions themselves could provide a model for simpler ideas.

(2) Although seemingly absurd, the student could make up his or her own questions. There is no better way of learning a subject than to try and teach it. An important part of any teaching process is the setting of test questions. A student can, with the aid of textbooks, learn quite a lot by composing a set of questions and answers because such a task must stimulate research. It may be argued that the student who set the test would gain little benefit from later taking it since he or she would already know the answers. This argument is valid if the test is taken a day or so after it was set. The results of the same test taken a week or so later when the memory begins to fade may be quite surprising as well as disappointing. With this in mind, students could prepare and keep a set of question files for later revision purposes.

Analysis of Program 6.2

The following should be read in conjunction with the listing of Program 6.2 and the structure diagram shown in Figure 6.2.

```
10 REM MULTIPLE CHOICE TEST PROGRAM
20 OPENOUT "buffer"
30 MEMORY &9B7E
40 CLOSEOUT
50 BORDER 0
60 DIM A$(100,5),H$(5),P$(5),ANS$(100)
70 flag%=0:lastT%=0
80 H$(0)="QUESTION"
90 H$(1)="ANSWER A"
100 H$(2)="ANSWER B"
110 H$(3)="ANSWER C"
120 H$(4)="ANSWER D"
130 H$(5)="Your RESPONSE stored as : "
140 P$(0)="Subject"
150 P$(1)="Test"
160 P$(2)="Author"
170 P$(3)="Date"
180 P$(4)="Ref"
190 P$(5)="Time (minutes)"
200 MODE 1
210 GOSUB 1480:LOCATE 1,5
220 PRINT"Place TEST CASSETTE in the mac
hine"
230 PRINT"REWIND tape ready to LOAD DATA
"
240 PRINT:GOSUB 830
250 PRINT:PRINT"MEMORISE and ENTER any 6
 DIGIT code"
260 INPUT code$
270 IF LEN(code$)=0 OR LEN(code$)>6 THEN
 250
280 CALL &BB03:'KM RESET
290 GOSUB 1480:LOCATE 1,5
300 PRINT"Enter testee's NAME (Max 22 Ch
aracters)"
310 LINE INPUT name$
320 IF name$="" THEN 300
330 IF LEN(name$)>22 THEN name$=LEFT$(na
me$,22)
```

```
340 name$=UPPER$(name$)
350 GOSUB 730
360 GOSUB 950
370 CALL &BB18:'KM WAIT KEY
380 FINISH=TIME/300+VAL(A$(0,5))*60
390 GOSUB 1080
400 SOUND 1,36,20,15
410 MODE 1:LOCATE 16,12
420 PRINT"TEST OVER":PRINT
430 LOCATE 5,14
440 PRINT"PRESS any KEY to DISPLAY RESUL
TS"
450 CALL &BB18:'KM WAIT KEY
460 GOSUB 730
470 GOSUB 1710
480 LOCATE 1,23
490 PRINT STRING$(40,CHR$(154));
500 PRINT"Enter code to OBTAIN MENU"
510 PRINT STRING$(40,CHR$(154));
520 try$="":SOUND 1,758,10,15
530 FOR CHAR%=1 TO 6
540 K$=INKEY$:IF K$="" THEN 540
550 SOUND 1,18,5,15:try$=try$+K$
560 NEXT
570 IF try$<>code$ THEN 520
580 flag%=0:FOR Q%=1 TO SIZE%:ANS$(Q%)="
":NEXT
590 GOSUB 1480:LOCATE 1,10
600 PRINT" (1) Load new QUESTION FILE"
610 PRINT" (2) RESIT test (new testee)
620 PRINT" (3) RESIT test (same testee)
630 PRINT" (4) EXIT program (Enable ESCA
PE key)
640 LOCATE 2,23
650 PRINT"Select option :"
660 K$=INKEY$:IF K$="" THEN 660
670 SEL%=VAL(K$)
680 IF SEL%<1 OR SEL%>4 THEN 590
690 ON SEL% GOTO 210,290,350,700
700 CLS:END
710 '
720 REM DISPLAY PANEL
730 MODE 1:GOSUB 1480
```

```
740 PRINT STRING$(40,CHR$(154));
750 FOR C%=0 TO 5
760 PRINT TAB(2) P$(C%) TAB(16) ": "A$(0
,C%)
770 NEXT
780 PRINT" File size      :";SIZE%;"QUEST
IONS"
790 PRINT STRING$(40,CHR$(154));
800 RETURN
810 '
820 REM LOAD FILE
830 PRINT"Enter filename :";
840 LINE INPUT filename$
850 OPENIN filename$
860 INPUT#9,SIZE%
870 FOR Q%=0 TO SIZE%
880 FOR C%=0 TO 5
890 INPUT#9,A$(Q%,C%)
900 NEXT:NEXT
910 CLOSEIN
920 RETURN
930 '
940 REM INSTRUCTIONS
950 LOCATE 1,14
960 PRINT"STEP through questions (wrap a
round) by"
970 PRINT"pressing one of the following
keys :"
980 PRINT"RIGHT arrow key increments QUE
STION No."
990 PRINT"LEFT arrow key decrements QUES
TION No."
1000 PRINT:PRINT STRING$(40,CHR$(154));
1010 PRINT"To END TEST at any time PRESS
 ESC key"
1020 PRINT STRING$(40,CHR$(154));
1030 PRINT
1040 PRINT"To START the TEST press any k
ey"
1050 RETURN
1060 '
1070 REM DISPLAY & ANSWER QUESTIONS
1080 Q%=1
```

```
1090 WHILE flag%=0
1100 GOSUB 1350
1110 FOR C%=0 TO 4
1120 PRINT H$(C%);" : "; USING "&";A$(Q%
,C%)
1130 NEXT
1140 LOCATE 1,24:GOSUB 1440
1150 WHILE flag%=0 AND INKEY(8)<>0 AND I
NKEY(1)<>0
1160 LOCATE 1,25
1170 IF ANS$(Q%)="" THEN PRINT"No RESPON
SE recorded" ELSE PRINT H$(5);ANS$(Q%)
1180 LOCATE 37,25
1190 PRINT"Press A,B,C or D to ANSWER or
 CHANGE ANSWER"
1200 K$=INKEY$
1210 IF INKEY(66)=0 THEN flag%=1
1220 GOSUB 1580
1230 IF K$="" AND flag%=0 THEN 1200
1240 K$=UPPER$(K$)
1250 IF K$>="A" AND K$<="D" THEN ANS$(Q%
)=K$
1260 WEND
1270 IF INKEY(8)=0 THEN Q%=Q%-1
1280 IF INKEY(1)=0 THEN Q%=Q%+1
1290 IF Q%<1 THEN Q%=SIZE%
1300 IF Q%>SIZE% THEN Q%=1
1310 WEND
1320 RETURN
1330 '
1340 REM DISPLAY HEADER 1
1350 MODE 2
1360 GOSUB 1440
1370 PRINT"MULTIPLE CHOICE TEST"
1380 LOCATE 30,2
1390 PRINT"Time left "
1400 LOCATE 64,2
1410 PRINT"QUESTION :";Q%
1420 GOSUB 1440
1430 RETURN
1440 PRINT STRING$(80,CHR$(154));
1450 RETURN
1460 '
```

```
1470 REM DISPLAY HEADER 2
1480 CLS
1490 LOCATE 1,2
1500 INK 2,6
1510 PAPER 2
1520 PRINT" MULTIPLE CHOICE TESTING "
1530 PAPER 0
1540 PEN 1
1550 RETURN
1560 '
1570 REM TRACK AND DISPLAY TIME LEFT
1580 T%=FINISH-TIME/300
1590 S%=T% MOD 60
1600 M%=T%\60
1610 H%=M%\60
1620 M%=M% MOD 60
1630 LOCATE 41,2
1640 PRINT USING "## : ## : ##";H%,M%,S%
1650 IF T%<=10 AND lastT%>T% THEN SOUND
1,36,5,15
1660 IF T%<=0 THEN flag%=1
1670 lastT%=T%
1680 RETURN
1690 '
1700 REM PROCESS AND DISPLAY RESULTS
1710 SCORE%=0:PASS%=0
1720 FOR Q%=1 TO SIZE%
1730 IF ANS$(Q%)=A$(Q%,5) THEN SCORE%=SC
ORE%+1
1740 IF ANS$(Q%)="" THEN PASS%=PASS%+1
1750 NEXT
1760 PRINT"RESULTS TABLE  : ";name$
1770 PRINT STRING$(40,CHR$(154));:PRINT
1780 PRINT"TOTAL Questions      :";SIZE%
1790 PRINT"Questions ATTEMPTED :";SIZE%-
PASS%
1800 PRINT"CORRECT answers      :";SCORE%
1810 PRINT"INCORRECT answers    :";SIZE%-
PASS%-SCORE%
1820 PERCENT=SCORE%/SIZE%*100
1830 PRINT"PERCENTAGE score     :";ROUND(
PERCENT,2);"%"
1840 PRINT:PRINT"Performance GRADE   : "
```

```
;
1850 IF PERCENT>=90 THEN PRINT"A (EXCELL
ENT!)"
1860 IF PERCENT>=70 AND PERCENT<90 THEN
PRINT"B (GOOD PASS)"
1870 IF PERCENT>=50 AND PERCENT<70 THEN
PRINT"C (CLEAR PASS)"
1880 IF PERCENT>=40 AND PERCENT<50 THEN
PRINT"D (PASS)"
1890 IF PERCENT<40 AND PERCENT>=30 THEN
PRINT"E (FAIL)"
1900 IF PERCENT<30 THEN PRINT"F (BAD FAI
L)"
1910 RETURN
```

Program 6.2. Multiple choice test.

The variable names used and their corresponding functions are as follows:

A$(Q%,C%) = main array for holding questions and answers, dimensioned A$(1ØØ,5).
H$(Ø − 5) = literal prompts for the questions and answers.
P$(Ø − 5) = literal prompts for heading panel information.
SEL% = chosen option number.
SIZE% = highest question number in use.
K$ = general purpose global variable.
filename$ = current file name.
Q% = question number.
C% = index to the question itself, the four answers and the correct answer.
flag% = test status: flag% = Ø, test in progress; flag% = 1, test over.
code$ = 6 character code to gain menu.
M% = minutes.
S% = seconds.
H% = hours.
T% = clock tick.

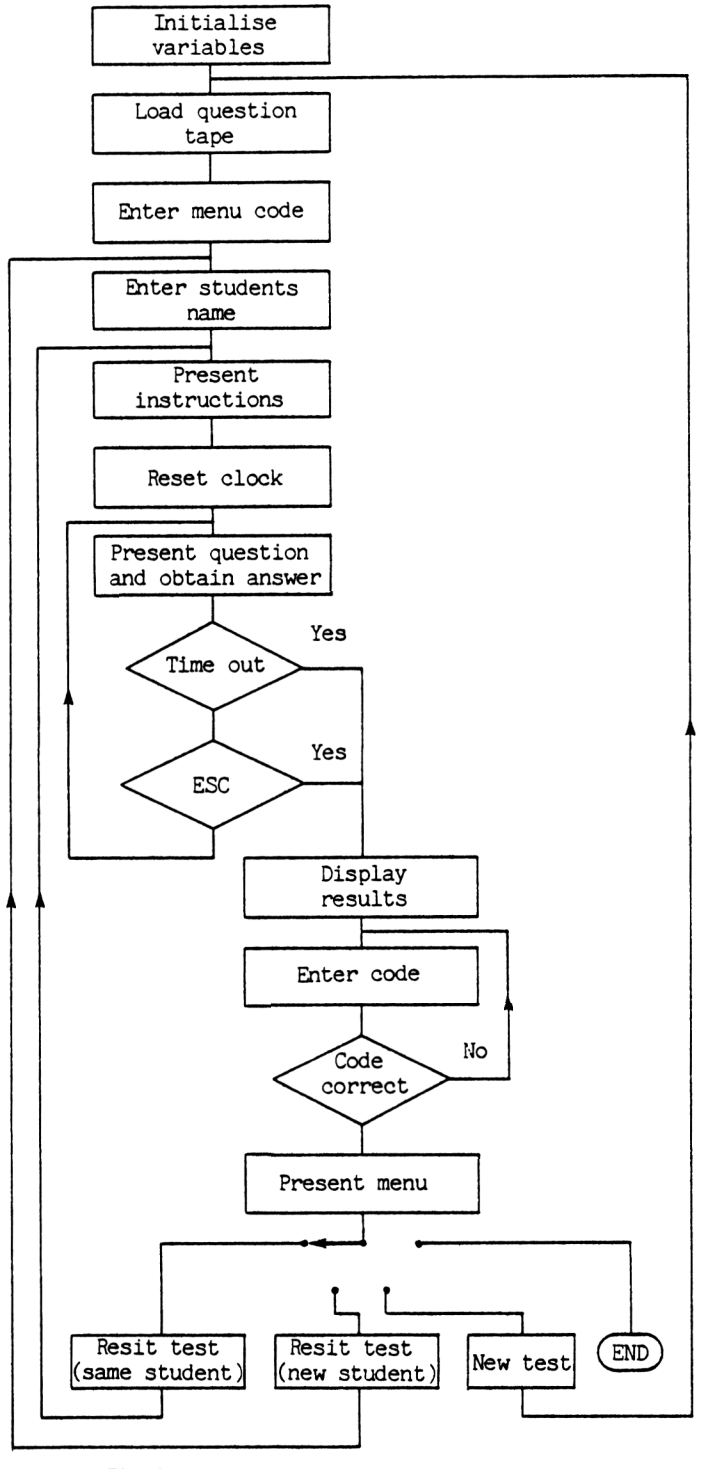


Fig. 6.2. Structure diagram for Program 6.2.

The subroutine calls are shown in Table 6.2.

Table 6.2. Subroutine calls.

Display and answer questions	Display header 1 Print line
Track and display time left	No other calls
Load file	No other calls
Display panel	Header 2
Instructions	No other calls
Display header 1	Print line
Display header 2	No other calls
Process and display results	No other calls

The actual 'program' extends down to the END at line 7ØØ; the remaining lines are occupied by subroutines.

Most of the subroutines are fairly straightforward and, apart from some of the variable names, are similar in form to those discussed in earlier programs. The subroutine at line 1580 assumes you are familiar with the MOD function because it is used to break down the time into seconds, minutes and hours. The screen presentation of the time in groups of two digits is achieved by the PRINT USING statement in line 164Ø.

It should be clear from Figure 6.2 how the subroutines are linked together.

Summary

1. A multiple choice test consists of questions together with four answers, only one of which is considered correct.
2. It is bad practice to use word tricks.
3. An 80-column display is essential for displaying a question together with four answers.
4. Two programs are required, one for preparing the questions and the other for answering them.
5. Both programs involve data tapes.
6. Program 6.1 must save a data tape containing the questions. Program 6.2 loads a data tape before the test begins.
7. When preparing questions or answers which take more than one line, use the space bar (not the ENTER key) to wrap around to a new line.

8. When preparing questions, keep an eye on the 'Bytes free' number.
9. Each question added is via the menu page.
10. Program 6.1 produces a 2000 baud data tape.
11. During preparation, any question can be viewed by quoting the question number.
12. Questions can be deleted but subsequent questions numbers will be decreased by one to maintain a numerical sequence.
13. When taking the test with Program 6.2, the final menu can be protected by a code word entered by the examiner.
14. Whilst the test is in progress, the time left is displayed at the top of the screen.
15. Audible warning is given when only ten seconds are left for the test.
16. During the test, the ESC key is disabled.

Self test

Using Program 6.1 and with the aid of a dictionary, compose a set of fifty questions on spelling. For example:

Question: A word meaning a mental derangement accompanied by feelings of persecution.
Answer A: parinoia
Answer B: paranoia
Answer C: parenoia
Answer D: paronoia

Having produced the data tape, use Program 6.2 for a self test – without using the dictionary.

Appendix A
Glossary

assembly language: a more user friendly method of entering machine code.
baud: a rate of one information bit per second.
binary search: a fast searching method, requiring a sorted list.
block: a unit consisting of a certain number of bytes. In the CPC464, a block is 2K bytes.
bug: a program error causing behaviour other than intended.
byte: a group of 8 bits.
command driven: a program in which an option is selected by a direct command rather than from a menu.
contiguous: occupying adjacent positions within a list.
crash: a situation in which the operator loses keyboard control.
database: a term which, in the home computer environment, is virtually synonymous with a filing system.
data file: file which contains data accessible only by a resident program.
data processing: a term originally used to cover storage and processing of information as distinct from computation.
default: the natural conditions assumed in the absence of commands to the contrary.
direct files: a file organisation which allows any record, irrespective of its position in the file, to be located immediately.
documentation: hard copy information which is intended to describe fully the operation and structure of a program.
drop out: loss of bits from a tape due to defects in the recording film.
exchange sort: a sort based on comparisons between adjacent items.
field: a subdivision of a record.
field heading: the title which defines the meaning of field data.
field width: number of characters allotted to a field.
file: collection of related information.
file creation: entering preliminary information such as field headings etc.

file reorganisation: a method, or program, which modifies the entire structure of an existing file.

file size: normally, the number of records held in the file.

flag: a data item which can be the equivalent of either yes or no.

garbage collection: periodic deletion of redundant strings and subsequent rearrangement of those left.

hard copy: computer output in permanent form, such as a printout on paper.

hierarchical: a system which is structured in order of importance.

indexed file: a file in which the records are accessed by first consulting a separate index file. The records need not be in any pre-defined order.

key field: a field which uniquely identifies a record.

leader tape: the first few inches of unrecordable tape.

machine code: a program written in a format which is immediately recognised by the microprocessor.

menu driven: a program in which the various options are selected from a menu page.

meta language: a concise set of symbols used to describe BASIC keywords without ambiguity.

numerical data: the character subset containing only the digits 0 to 9 inclusive.

object code: the machine code translation of source code.

pointer: a variable, representing the address of another variable.

primary option: one of the main menu options.

program based: a file in which the data items are within DATA statements.

program module: a section of program which, although part of the whole, has a recognisable function.

prompt: a screen message requesting some input from the keyboard.

pseudo variable: a variable in which the value can be changed but not the nature.

record: subdivision of a data file.

recursion: a subprogram or subroutine which calls itself.

secondary key field: a field which tends to classify rather than indentify itself.

secondary option: an option selected from a subordinate set of options.

sequential file: one in which records are stored in key field sequence.

sequential integrity: permanent preservation of original order.

sequential search: searching from the first record and continuing sequentially until the desired record is found.

source code: a program written in assembly or higher level language.
subfile: a file containing only a selection of records from a main file.
substring: a group of characters within a string.
swop flag: a flag indicating whether or not a swop has occurred.
tape buffer: an area in memory from which data is transferred to and from tape.
tokens: the two-byte code used by the interpreter when storing keywords.
trace table: a pencil and paper method of plotting the progress of a program.
user friendly: the built-in quality of a program to recognise and allow for human weaknesses.
volatile: a memory in which the data is lost when the power supply is interrupted.
wrap around: beginning again on reaching the end of a file.

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
Answers to Self Test Questions

1.1 It was non-volatile.
1.2 Memory is internal and volatile. Store is external and non-volatile.
1.3 *BYTE* magazine originally used a normal tape recorder for storing digital information. It was published in Kansas City.
1.4 2K.
1.5 Two tape files can be opened at once.
1.6 End of file has been detected.
1.7 FRE(" ") enforces garbage collection first; FRE(ø) doesn't.
1.8 Because the recording heads may not be aligned quite the same on different machines.
1.9 False.

2.1 No.
2.2 It must be unique.
2.3 When there are a large number of records.
2.4 Processing is fast once the file is loaded.
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.

3.1 Line 3øø should be altered to DATA 13,2.
3.2 To register whether the record is on file.
3.3 To reject a null string input, i.e. ENTER pressed before data is entered.
3.4 You can't sort a file containing only one record.

4.1 To allocate permanently the cassette buffer areas.
4.2 To restore the 1000 baud rate.
4.3 So that the array A$ may be redimensioned.
4.4 To ensure that the edited line appears within the centre of the dotted lines.

5.1 20 million.
5.2 6 hours.
5.3 500000.
5.4 Can have poor near order performance.
5.5 Improved mixing of sets for large values of N.

6.1 No formal answer required.

Index

This book shows how to construct both general purpose and specialised filing systems – using the cassette system – for a variety of applications.

Complete BASIC listings and subroutines are fully described and written in module form so that users should find it easy to tailor them to suit their own individual needs. Fast machine code routines are also included as alternative options where high executing speed is essential.

The Authors

A. P. Stephenson has a long and distinguished record as a writer on electronics and computing for the enthusiast. He has contributed regularly to the popular computing journals and is the author of ten other books.

D. J. Stephenson is another life-long computer enthusiast. He has contributed to popular electronics and computing journals, and is a co-author of four other books.

Other books for Amstrad users

AMSTRAD COMPUTING
Ian Sinclair
0 00 383120 5

SENSATIONAL GAMES FOR THE AMSTRAD CPC464
Jim Gregory
0 00 383121 3

ADVENTURE GAMES FOR THE AMSTRAD CPC464
A. J. Bradbury
0 00 383078 0

40 EDUCATIONAL GAMES FOR THE AMSTRAD CPC464
Vince Apps
0 00 383119 1

PRACTICAL PROGRAMS FOR THE AMSTRAD CPC464
Audrey Bishop and Owen Bishop
0 00 383082 9

Front cover illustration by Godfrey Dowson

COLLINS
Printed in Great Britain
0 00 383102 7

£8.95 net

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/