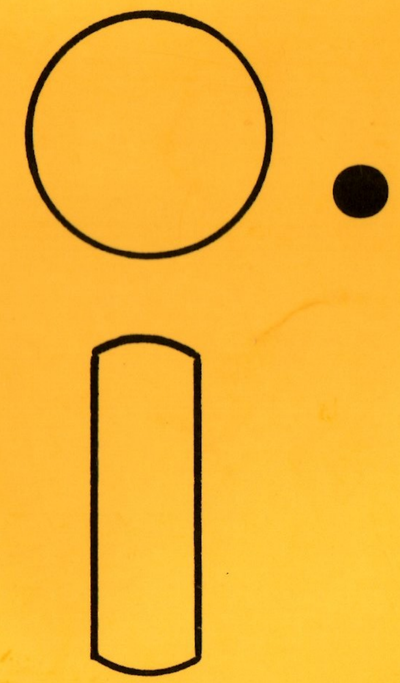
EINSTEIN

DATA FILE HANDLING IN BASIC



Phil Croshaw.

CONTENTS

Chapter 1 Introduction.

This book was written to complement the Tatung Einstein Basic Reference Manual provided with the Einstein microcomputer. Constant reference is made to pages within the Basic Reference Manual. It is therefore suggested that you have a copy close at hand when reading this book.

It is assumed that the user of this book has a reasonable understanding of the Basic programming language, though not necessarily the Tatung/Xtal Basic. However, please note that although the principles of data files descibed in this book are true for most types of Basic, certain parts of this book are only applicable to Tatung/Xtal Basic.

All sample programs are written with 'readability' in mind. It is more than possible that sample programs could be written in such a manner that they RUN faster or take up less memory. It is hoped that this book, as well as showing the reader how to use data files, will provide 'food for thought' when writing file handling programs in the future.

Chapter 2 Introduction to data files.

2.1 Data storage

There are two basic types of files that may be stored on a disk :

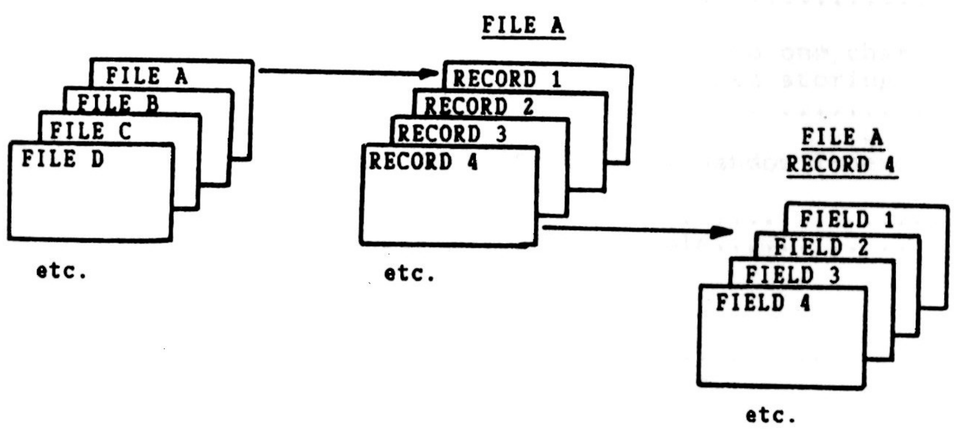
a) Program files
b) Data files

We are concerned with the latter, data files. Data files can be used to store any type of textual and/or numeric information. This information may be updated, deleted, added to or printed when required. Examples of information that may be stored are :

a) Documents, letters, forms etc
b) Employee information
c) Stock control information
d) Accounting figures
e) Names and addresses

The list is endless and depends on what information you need to store on a disk.

A data file may be likened to a desk top card box containing a large amount of cards. Each card could for example, contain information on one customer. The box as a whole is the 'data file'. Each individual card is a 'record'. The record may contain the customers address, postcode, telephone number, credit limit etc. Each single piece of information is known as a 'field'. For example, the postcode is one field, the credit limit is another field, on so on. A single letter or number within a field is known as a 'character' or 'byte'. The following diagram illustrates these divisions of a data file.



One or more data files may be stored on any one side of a 3" disk. The amount of data that may be stored is limited by the amount of free space on the formatted disk. Use the DIR command under DOS to find how much space is available.

A formatted 3" disk with no files on the disk has 192K of free space. The directory listing specifies the amount of free space in units of 1K. One K is 1024 characters or bytes. Therefore an empty formatted disk may contain 196,608 characters. If your data record consists of 100 characters you will be able to store 1966 records on an empty formatted disk.

2.2 Designing the record layout.

Once you have decided on the filename, the next step is to decide on the record layout. In deciding the record layout you will need to take into consideration the file organization you will be using. The choice of file orgainization is discussed in section 2.3.4 of this chapter.

When you are deciding the information to be stored in a record on a file, a good idea is to list the 'output' information you want to see on the printer paper or on the screen. Let us take an example to clarify this task. We will use a "Customers Details" information system in the example. For simplicity we will keep the number of fields in the record to a minimum. The output we might require is as follows :-

a) Customer number
b) Customer name
c) Address line 1
d) " " 2
e) " " 3
f) " " 4
g) " " 5
h) Phone number
i) Credit limit
j) Current credit

From this information, we would require a master list of all customers and all the field information, listed above, for each customer. Another report that would be handy, would be a total count of all monies owed. Also, the amount of credit still available for each customer would be of interest. This would be calculated by subtracting the money owed from the credit limit for each customer.

There are two pieces of information we require that are not stored on the data file.

a) Credit available
b) Grand total of monies owed

These two fields need not be stored on the data file as they can be calculated each time the report program is used. This will save space and therefore increase the number of records you can store on the disk.

The length of each field depends on the file organisation selected. The various file organisations will be dealt with in the remainder of this chapter. Further details on each of the three organisations can be found in chapters 3,4 and 5. Also, look at chapter 6 'Data compaction', for hints on reducing the length of fields and records before deciding on the total length of each field and the total record length.

2.3 File organisations.

There are three main different types of data file structures (also known as 'organisations'). They are :-

a) Sequential
b) Random - Relative
c) Random - Indexed

The following three sections will introduce the beginner to the structures of the files. For programming examples and further detail on each file structure please see the relevant chapter.

2.3.1 Sequential data files.

Sequential data files contain a number of data records with one record after the other. Each record consists of a series of fields with each field terminated by a carriage return/line feed. (In chapter 6 you will see that the number of field terminators can be reduced). A carriage return has the ASCII value 13 and line feed has the ASCII value 10. These two characters form the carriage return/line feed terminator at the end of each data field when stored onto the disk. They are there so that when a program reads the record/fields from the disk, the INPUT# statement can delimit on the carriage return/linefeed.

When reading a sequential file on a disk, the program will have to start by reading the first record in the file before it can read the second record. To read the 50th record would involve the program first reading the 49 records before it can access the 50th. It is a time consuming way of getting to the 50th record if you only want to look at the 50th record. However sequential files do have advantages. Because all records are read prior to reading the record you require, the user may store each record in a variable length format and so save space on a disk.

Any new records created in a sequential file would be added onto the end of the data file. To find the end of the file before writing new records, the data file must be read until the 'End of File' (EOF) mark is detected. Alternatively, the file can be OPENed in APPEND mode which positions the file pointer at the end of the data file ready for appending new records. See page 274 of the Basic Manual

2.3.2 Random data files - Relative.

A random-relative file contains a series of one or more records with each record being of a fixed length. The length must be decided upon before using the 'CREATE' statement. To calculate the length of the record, add up the maximum length of each field within the data record. Then add 2 for each field within the record to allow for the carriage return/line feed. See the previous section on 'carriage return/linefeeds' before continuing with this section. The number of carriage return/linefeeds may be reduced. See chapter 6 for details.

Taking the example of the Customer Data file mentioned before, we can see how the length of the record may be calculated. Below are the suggested maximum lengths :-

Field name	Max length of field	Value range
Customer No.	5	0 to 99999
" name	25	
Address line 1	30	
" 2	30	
" 3	30	
" 4	30	
" 5	30	
Phone Number	12	
Credit limit	7	0.00 to 9999.99
Current credit	7	" " "

In the above example the numeric fields have the maximum value that can be stored in each field on the right hand column. Adding all the field lengths together gives you a total of 206 characters. Add 2 for each field within the record to allow for the carriage return/linefeeds and you have a total record length of 226.

It is important to note that random-relative files are fixed length records, with all records being the same length. Because of the fixed length nature of random files it is possible for the program to go straight to a particular record. If the record length is 226 and you want to read the 100th record, by simple multiplication we can calculate that there are 22374 bytes before the 100th record (99 X 226). Therefore the first byte of the 100th record is the 22375th byte in the file. All the calculations and jumping through the file is done automatically by Xtal Basic, and need not concern the user.

The ability to move about within a data file accounts for the word 'random'. The word 'relative' for this file organisation comes about because of the numbering method used to find a record. Each record is numbered relative to the beginning of the file. Record 98 is the 98th record 'relative' to the beginning of the file.

Compare the random-relative data file structure to the sequential file structure. One can quite easily see that to get an individual record from a random-relative file can take far less time than a sequential file depending on how far down the file the record is. Random speed is at the expense of disc space because random files require fixed length records even though only part of the record may contain information.

2.3.3. Random data files - Indexed.

Please ensure that you are quite familiar with sequential and random-relative files in the previous two sections before reading this section.

A random-indexed file is a data file with fixed length data records. As with random-relative data files, the user can read any single record directly without having to read preceeding records first. The difference between a relative and indexed data file lies in the way an individual record is accessed. With relative data files, an individual record is selected by a number, the number relating to the relative position of the record within the file. With indexed files a record can be accessed via an alphanumeric 'key'. The key is a string of characters, each key being unique in the index. As well as a key there is a 'pointer' associated with each key pointing to the data record. A pointer works in much the same way as the 'relative' record number in a random-relative file. If you look at fig 3 you can see the relationship between the index and the data records.

The index can be held on disk within the same file as the data or seperately. To start with however, we will say that the index is to be stored in a seperate file from the data records. To find a key, the index array will be searched in a FOR/NEXT loop. This can be speeded up by keeping the index sorted and doing a binary chop to find a key in the index (see section 3.6).

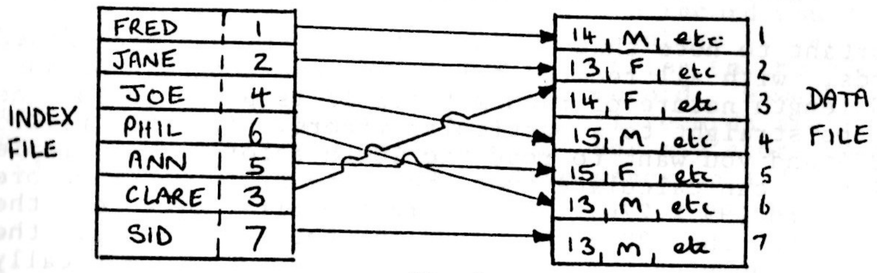


Fig 3

When using index files the first thing to do is load the index from the index file into two arrays. The first array being a string array holding the key, and the second array being an integer array holding the pointers pointing from the key to the data record. With the index in memory, accessing of records is faster than if the index is left on the disk. However, as data records are deleted and added, you must be careful to remember to update the index array in memory as well as the index on the disk.

Initially, as records are added, the index entry and data records are in the same relative positions within their files. As more additions take place, and data is deleted and the index is kept sorted, the postions of the index key/pointer can differ from the postion of the related data record. This shows why the pointers are needed in the index.

At this stage the reader should not expect to be able to write a Basic program using random-indexed files. Chapter 5 gives the reader practical and more precise information on programming with random-indexed files.

2.4 Selecting the file structure required.

After reading the previous three sections the user should start to be in a postion to judge on which type of file organization to use. Below are some brief guidelines to help you select the organisation.

First decide if you need the facility of 'random' access. Files where the user would want to examine individual records, and there are large amounts of data, then a random file should be chosen. Sequential organisation would suit files with a low volume of data or those files that would not require individual access to one record eg a text file containing a letter or a document.

In chosing between relative and indexed you would need to examine the way in which the user of the program (and therefore data) would want to access data on an individual basis. Taking the 'Customer Details' file mentioned earlier as an example, the key for each record if you were using a random-index file, might be the surname or company name. This would be usefull if the user wanted to search for a particular surname. The problem that quickly arises is that each key must be unique. It would be more than possible that there would be two 'Smith's. A way of getting over this 'uniqueness' problem is to give each record a 'customer number'. In giving a number, say 1 for the first customer on the file, 2 for the second and so on we can see that this suits the random-relative file organisation.

Part-codes on a stock system are generally made up of alphanumeric characters eg 123/223/TRE/11. A user would expect to

find details of this part from a data file using the part number as the unique key. The part-code would not be acceptable in a random-relative file on its own because it is not numeric. If all the part-codes were stored on an index, and access to the data was through the key, then a random-indexed file would be needed.

It is difficult to generalise in selecting a file organisation because each problem needs to be looked at individually. As you practice using the various organisations the choice of organisation will become easier.

Chapter 3 Sequential Data Files.

3.1 Storing data.

The first step in using data files is to put some data into the file. Other file operations, such as deleting, reading or updating data, can not take place until data exists in the file.

To create an empty data file use the CREATE statement (Basic Manual P. 53). Creating a data file is a one off operation. After the file has been created, the file should then be opened for use using the OPEN statement (Basic Manual P.163). If you use the CREATE statement to create a file with the same filename as an existing data file, then the existing data will be erased and a new empty data file will be created. Unless you want to reCREATE an empty data file each time, it is usual to write a quick one-off program to create the data file. Another method would be to first use the OPEN statement on the file. If the file does not exist then error number 25 will be generated, and using the ON ERR statement you could direct the program to CREATE the file before resuming the normal processing. The two listings below demonstrate the two options.

```
10 REM **********************
20 REM A QUICK CREATE PROG
30 REM **********************
40 CREATE "SAMPLE.DAT",FD$
50 CLOSE
60 END
```

```
10 REM ************************
20 REM USING ON ERR TO CREATE
30 REM ************************
40 ON ERR GOTO 3000
50 OPEN "SAMPLE.DAT",FD$
60 PRINT "FILE ALREADY EXISTS"
70 GOTO 90
80 PRINT "FILE NEWLY CREATED"
90 REM REST OF PROGRAM
100 GOTO 5000
3000 REM ***ERROR ROUTINE****
3010 IF ERR=25 AND ERL=50 THEN
       CREATE "SAMPLE.DAT",FD$
       :OFF ERR: GOTO 80
5000 CLOSE: END
```

Although the first example looks far simpler, quite often the second example can provide a more flexible program.

Now the file has been created the next step is to put some information into the file. First accept the data for each field from the user. Validate the data if needed, for example, if one field is 'age', then check the data entered is numeric. Store each piece of information (ie each field) into its own variable. When all the data for one record has been entered and validated the information can be stored as the first record on the file with the following statements :-

```
10 OPEN "SAMPLE.DAT",FD$
20 PRINT "ENTER NAME ";
30 INPUT A1$
40 PRINT "ENTER AGE ";
50 INPUT A2$
60 PRINT #FD$
70 PRINT A1$
80 PRINT A2$
90 PRINT #0
100 CLOSE
110 END
```

Line 60 and 90 are very important. There is only one channel linking the computer with other devices eg screen, disc drives etc. Before outputing data to a file on a disc or to the screen you need to tell the computer which file/device is to recieve the data. Individual files are identified via their file descriptor (see Basic Manual p.268). The screen is on number 0. The default is 0, however once the channel has been assigned to another device or file the channel remains linked to that device or channel until the channel is reassigned as in line 90. Any more output after line 90 will be directed to the screen. It is a good idea to set the channel immediately back to the screen after outputting to a file incase an error occurs and an error message needs to be displayed on the screen. Failure to reset the channel back to the screen will result in any error messages for the screen being directed into the data file !! Another way of setting the channel back to the screen is to do a global CLOSE, ie CLOSE all files. This reverts the I/O channel back to the default channel, the screen, but if you want to use the file again the file will need reOPENing.

In the above example, the two fields to be stored in the file, are stored temporarily in A1$ and A2$. If you had, say, 10 fields, then a quicker way would be to use an array for temporary storage, then store the fields on the file using a FOR/NEXT loop. This can be done as follows :

```
.
.
.
70 FOR I=1 TO 10: PRINT A$(I): NEXT I
.
.
```

Using this method, line 80 in the original example would not be needed.

3.2 Reading data.

The file will first need to be opened using the OPEN statement. To be able to read data from a file, some data must exist. Your program should however allow for the possibility of there being no data in the file.

With the file open you can read data from the file using the INPUT# statement (Basic Manual p.113). The INPUT# statement pulls characters from a file or device depending on the channel specified. To set the channel to point to the file you want to read use the INPUT# statement as follows :-

```
INPUT #FD$
```

The above statement forces all further input to be taken from the device or file opened previously with the file descriptor FD$. The first INPUT statement, after the above INPUT #FD$, will read the first field from the first record of the data file. The next INPUT will read the second field and so on. For each field you need one INPUT statement. The Xtal/Basic INPUT statement delimits on the cariage return/linefeed put at the end of each field. The following is a program to read the record from the file SAMPLE.DAT as created in section 3.1 of this chapter.

```
10 OPEN "SAMPLE.DAT",FD$
20 INPUT #FD$
30 INPUT A1$
40 INPUT #A2$
50 PRINT "NAME IS ";A1$
60 PRINT "AGE IS ";A2$
70 INPUT #0
80 CLOSE
90 END
```

In the above example, there are two fields, name and age, which are read into the variables A1$ and A2$. Note line 20 which sets the I/O channel for input from the file with the file descriptor FD$. The channel is set to receive any further input from the screen at line 70. Line 70 can be removed, as line 80 sets all input/output via the screen and CLOSEs all files.

If there were 10 fields to the record, then rather than have 10 INPUT statements, you could use a FOR/NEXT loop as follows :-

```
30 FOR I=1 TO 10:INPUT A$(I):NEXT I
```

In the above example the 10 fields will be put into each of the 10 elements of the array A$( ). Line 40 should be deleted. If there was more than 1 record to be read then a GOTO 30 needs to be inserted at line 75.

Constantly reading a file, one record after another, will work so long as there are records to read. When the program reaches the end of the data file an End of File (EOF) error occurs. See page 74 of the Basic Manual. When EOF is detected the program can be forced to branch off to another part of the program.

As well as the INPUT statement, the INCH$ statement can be used to read data from a data file. However, the INCH$ statement is

not suitable for variable length records as is usually found with sequential data files. The use of he INCH$ statement is dealt with in future chapters.

If you are reading one record after another from a sequential file, Basic has a 'file pointer' that keeps track of which is the next record to read. If you are part way through reading a sequential file and want to start reading from the first record onwards again, you will have to CLOSE the file, and reOPEN it. This has the effect of setting the pointer at the top of the file, pointing at the first record.

3.3 Updating data.

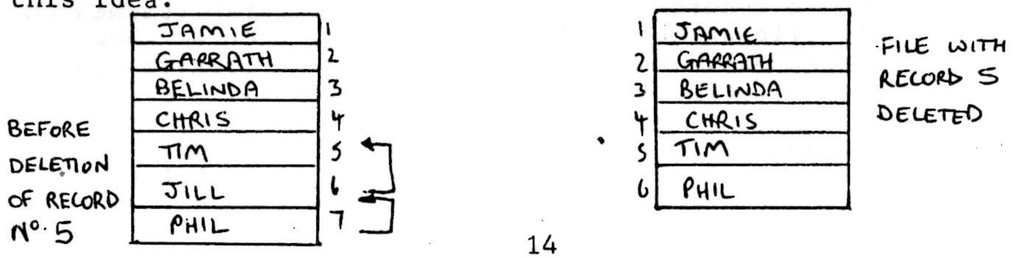
Sequential files can not be updated in a quick and simple way. When you write to a sequential file the EOF marker is moved to the end of the record just written. This is fine if you are at the end of the file. If the file pointer is at the beginning of the file, and a new record is written to the file, the records after the new record will be lost, as the new record written will contain an new EOF marker.

One way around this problem is to store ALL records and their fields in an internal array. Then update the data in the array as required. Finally, before exiting the program, CREATE a new file and write all the data within the array to the created file. This has the serious limitation of taking up a lot of memory and therefore reducing the amount of records that can be stored. On the other side of the coin, records held internally mean very fast access time to find a record as there is no disk access required after the records have initally been loaded into the internal array.

If you require the facility of being able to update individual records then it could be that you need a random file to allow a larger number of records to be stored.

3.4 Deleting data.

Deleting records from sequential files poses a similar problem as that encountered in updating records in sequential files. Individual records can not be updated, or deleted, directly from the disk. As with the previous section, one way around this is to load all records into an internal array. To delete a record, shuffle all the records up the array by one, starting with the record after the one to be deleted. The diagram below illustrates this idea.



The problems with this method is that you are limited to the size of the internal array by memory size. If the record to be deleted is the first element in the array then ALL elements below have to be moved up. This can be time consuming according to the size of the array.

If you require the facility of being able to delete individual records then it is suggested that you consider a random file, depending on the number of records to be stored on disk.

3.5 Sorting data.

This book contains two types of sort routines.

a) Bubble sort
b) Binary chop sort

There are many other types of sorts, but one of the above should satisfy the needs of most programs. The Binary Chop Sort is quicker than the Bubble Sort when sorting large volumes of data with many records out of sequence. The detailed description below should allow you to decide on the routine to use.

Bubble Sort.

I have also heard this being called a 'ripple ' sort, as well as many other things ! The Bubble sort can take place in an internal array or be sorted directly to/from the disk.Because sequential files do not allow updating of individual records, records can not be sorted directly to/from the disk. If you want to sort records on the disk, without the use of an array, then opt for a random file.

When sorting sequential files the first step is to OPEN the file and read each record from the data file. All records should be placed into an array, the first record in the first element of the array, second record in the second element and so on. When all records are stored in the array, make sure the 'number of records' to be sorted is kept in a variable. Set up two FOR/NEXT loops with the outer loop being 'FOR 1 TO the number of records to be sorted'. The inner loop is 'FOR 1 TO the number of records to be sorted minus 1 each loop'. With each pass an element of the array is compared with the next element of the array. If the contents of the current element is greater than the contents of the next element then these two elements are swapped around. This process continues through the array. The first pass forces the element with the highest value to the end of the array. Therefore, on the second pass there is no need to look at the last element, so the number of elements to be compared is reduced by 1. The Bubble Sort in BASIC code is as follows :

```
 1  REM C EQUALS THE NUMBER OF RECORDS TO SORT
 2  REM A$( ) ARRAY CONTAINS THE RECORDS TO BE SORTED
 5  DIM A$(100)
 10 C=100:C1=C
 20 FOR I=1 TO C
 30   C1=C1-1
 40   FOR I1=1 TO C1
 50     IF A$(I1) > A$(I1+1) THEN SWAP A$(I1),A$(I1+1)
 60   NEXT I1
 70 NEXT I
```

The SWAP statement can be found on page 222 of the Basic Manual. If you want to use this piece of coding with a Basic language without the SWAP statement then substitute the SWAP statement with the following :

```
..... THEN T$=A$(I1+1): A$(I1+1)=A$(I1): A$(I1)=T$
```

I have put leading spaces in the coding above routine to show clearly the start and end of each FOR/NEXT loop. The records to be sorted are in the array A$( ). Make sure that the array is DIMensioned to allow the maximum number of records possible or expected.

If the array is sorted, except for one record which is at the beginning of the array, and needs to be sorted down to the bottom part of the array, then after one pass the unsorted record will be positioned correctly. However, the sort will continue to loop through. To stop this unnecessary processing you can add a bit of coding to get the program to jump out of the sort if all records are sorted. The coding consists of a flag being set if a SWAP has taken place. If during a complete pass through the inner loop the flag is not set, then there are no more records to be swapped so you can stop the sort. The lines of coding below may be inserted into the Bubble Sort to catch this condition :

```
15 F=0
50 ..... :F=1
65 IF F=1 THEN F=0: ELSE GOTO 80
80 REM Rest of program
```

Binary chop sort.

If you are not aware of a 'binary chop' then read section 5.6 before reading this section. A binary chop sort works on the principle of pre-sorting subsets of records which reduces the transfer of records. A binary chop sort compares, and swaps if necessary, records up to half the file apart, and thus records move to their correct postion much more quickly than if only adjacent records were compared and swapped.

Suppose there are a thousand items to be sorted. If one of the

items needs to be moved to the other end of the file and a bubble sort is used, it must be swapped a thousand times with its neighbours before it gets there. If we use a binary chop sort, it is first compared with the item 511 records away, then with the record 255 records further on, then 127 more, then 63, 31 and finally a 15 record jump. This sort can move a record one thousand postions up the file in 6 jumps when a bubble sort would have moved it only 6 postions of 1000 moves!

The folowing program listing shows a routine to perform a binary chop sort. The array A$( ) contains the records to be sorted, while N contains the number of records to be sorted. The value D is used to 'chop' the array up.

```
1010  REM Binary chop sort routine
1020  LET D=1
1030  LET D=D*2
1040  IF D<=N THEN GOTO 1030
1050  LET D=INT((D-1)/2)
1060  IF D=0 THEN GOTO 1199
1070  FOR I=1 TO N-D
1080      LET J=I
1090      LET L=J+D
1100      IF A$(L) < A$(J) THEN GOTO 1120
1110      GOTO 1140
1120      SWAP A$(J),A$(L): J=J-D
1130      IF J>0 THEN GOTO 1090
1140  NEXT I
1150  GOTO 1050
1199  RETURN
```

The RETURN at 1199 allows the above routine to be GOSUBed. Remove line 1199 if it is not required. Before using the above routine don't forget to set N with the number of records to be sorted, store the records into the array A$( ), and set the DIM statement to the correct size for A$( ).

If you are sorting data on a disk or in an array with the data taken from the disk, then it is advisable to use a binary chop sort. If you only require a small number of records in an internal array then opt for a bubble sort.

It is a good idea to display a counter on the screen to show that the data is being sorted, and that the computer hasn't gone dead. If so, put the PRINT #0 statement after the NEXT of the inner loop with the Bubble Sort or with the Binary Chop Sort put the PRINT statement at line 1055.

Chapter 4 Random Relative Data Files.

Section 2.3.2 of this book gives a background to Random - Relative files. Also, if you look at page 283 of the Basic Manual you will find an example of using a Random - Relative file. Please note that throughout the Basic Manual, where the manual refers to Random Access files, the author is in fact talking about what this book calls Random - Relative files. Both names are acceptable.

4.1 Storing data in a Random - Relative file.

The first step in storing data on a random - relative file is to CREATE the file. This has been dealt with in some detail in section 3.1 on Sequential Files. Also see page 53 of the Basic Manual.

We will continue this chapter with an example. The data to be stored consists of information for a garage Parts Department :

a)	Part description	length	20	alphanumeric
b)	" quantity in stock	"	4	numeric
c)	" re-order level	"	3	numeric

With relative files, access is gained to a particular record via a number relating to the record's postion from the beginning of the file. We must have some way of knowing what record number relates to what part record. One way would be to make the 'part number' the same as the record number. The part number does not need to be stored on the file as a seperate field. The part number for the first record in the file will be 1, 2 for the second record and so on. Generally part numbers contain quite a lot of information, besides a record number. For example, the part number for a Volvo 244/DL 1968 saloon hub cap might be V244/DL68/W124 where the W stands for the wheel assemely. The information contained in the part number tells the operator a lot, and is obviously needed. To give the part a record number we could append a record number onto the dealers part number so the above becomes V244/DL68/W124/12 where the 12 on the end tells us that this particular part is the twelveth record in the file. The record number can be stripped off the complete part number with the RIGHT$ statement (Page 200 of the Basic Manual).The full part number would need to be stored as a 4th field. This shows how you can use meaningful 'key numbers', however for ease of understanding, the part number in this example will be a number only relating to the records postion in the file.

The file is to contain the three fields listed above. The length of each field is given alongside as is the type of data to be stored. The number of records allowed will depend on the amount of free space on the disk. The first allowable record number is record 0. In this example we will use record 0 as a count of the number of records on the file at any one time. Keeping a

count of the number of records within the file makes it easier to FOR/NEXT through the file and checking where to add a new record. Using a counter at the beginning of the file requires you to set the counter to zero when you CREATE the file. This is coded as follows :

```
100 CREATE "PARTS.DAT",FD$,33
110 PRINT #FD$,0;0
120 CLOSE
```

Line 100 is similar to the CREATE discussed in detail in the previous chapter, with the exception of the record length. This must be specified (33 in this case) because Random files require the length of the record to calculate the start of any record. Don't forget to add on 2 for the carriage return at the end of each field. Doing this to the Parts File described above gives us a record length of 33. It is quite common in business applications to allow a number of spare bytes for expansion of the file to include additional information at a later date. Line 110 sets the record count to zero.

Any time that you wish to add a new record, the new record will generally be added onto the end of the existing records. The exception is if a record has been deleted, then the new record could be inserted at the postion of the deleted record (see section 4.4). In adding a new record first OPEN the file and pick up the record count. To pick up the record count, use the following coding :

```
10 OPEN "PARTS.DAT",FD$,33
20 INPUT #FD$,0;N
30 REM N=THE RECORD COUNTER
```

The record count taken from record zero needs 1 to be added to it to give the record number for the new record to be added. If the record count is 123 then the new record will become the 124th record on the file. If the record count is zero then the new record will become record 1. So to continue the coding above, to add a record the following coding is required :

```
40 PRINT "ENTER PART DESCRIPTION ";: INPUT A$(1)
50 PRINT "ENTER QUANTITY IN STOCK ";: INPUT A$(2)
60 PRINT "ENTER RE-ORDER LEVEL ";: INPUT A$(3)
70 N=N+1: PRINT #FD$,N
80 FOR I=1 TO 3: PRINT A$(I): NEXT I
```

The above program does not have any validation on the items input. Each field should have an IF/THEN statement to check the maximum length allowed for the field has not been exceeded. Also check that the 'quantity in stock' and 're-order level' are numeric, before storing on the file. Note at line 70, as well as incrementing the record count, the line also directs all output to the file OPENed on channel FD$. The same statement tells the program that the data is to be sent to the Nth record.

Now the record has been stored on the file, the record count in record 0 needs to be updated. This should be done as soon as the new data record has been stored in case the computer crashes. To update the record count add the following line :

```
90 PRINT #FD$,0;N: PRINT #0
```

The variable N contains the up to date record count. Don't for get to set the output channel back to the screen with a the PRINT #0. Otherwise CLOSE the file. Note that a CLOSE is the securest way of safeguarding data in the event of a system crash. The final output to the disk can be lost in the event of a crash. This is because the operating system writes information to the disk in 'blocks'. A block is only written onto the disk when data for a different block is required to be written to the disk. A CLOSE however writes any remaining data to the file and reverts input/output back to the screen. It is a good idea to put a close after storing one, or a batch of new records.

Chapter 6 gives you information on compacting data and so save disk space.

4.2 Reading data - Random Relative file.

Use the OPEN statement to open up the input channel for access :

```
10 OPEN "PARTS.DAT",FD$,33
```

Then pick up the record count from record zero and store in a numeric variable :

```
20 INPUT #FD$,0;N
```

You can now set up a FOR/NEXT loop to read through the whole of the file or just read one record using the record number. First let us look at using a FOR/NEXT loop to read all the data records on the file.

```
30  FOR I=1 TO N: INPUT #FD$, I
40      FOR I1=1 TO 3
50          INPUT A$(I1)
60      NEXT I1
70      PRINT "PART NUMBER      ";I
80      PRINT "DESCRIPTION      ";A$(1)
90      PRINT "QTY IN STOCK     ";A$(2)
100     PRINT "RE-ORDER LEVEL   ";A$(3)
110 NEXT I : CLOSE
```

The outer FOR/NEXT loop (I) makes N number of passes, reading one record at a time. The inner loop (I1), makes three passes, one for each field in a record. Note how the record number, I, is also used as the part number. The INPUT #FD$,I at line 30 sets the file pointer to the start of the Ith record.

If you only want to read an individual record then at some stage the Part Number/record number needs to be input by the user. Putting this into code form you would arrive at something like the following :

```
30  PRINT "ENTER PART NUMBER ";: INPUT I$
40  I=VAL(I$)
50  IF I<1 OR I>N THEN PRINT "INVALID PART NUMBER":  GOTO 30
60  INPUT #FD$, I
70  FOR I1=1 TO 3: INPUT A$(I1): NEXT I1
80  PRINT "PART NUMBER      ";I
90  PRINT "DESCRIPTION      ";A$(1)
100 PRINT "QTY IN STOCK     ";A$(2)
110 PRINT "RE-ORDER LEVEL   ";A$(3)
120 CLOSE
```

The part number entered at line 30 becomes the record number at line 60. The FOR/NEXT loop at line 70 reads the three fields from the file, and finally the data is displayed. The CLOSE at line 120 reverts the input channel to the keyboard.

Reading Random-Relative files is quite easy once you know the record number. It is a good idea to always make the record number a meaningful field within the record ie in the above case the record number has become the Part Number. If you do not check that the record number is greater than the highest record number already existing on the file then trying to read past the last record will result in an 'End of Text' error (see line 40 in the program above).

4.3 Updating data in a Random Relative file.

There are two steps in updating data :

1. Read the record so the user can check its the correct record

2. Overwrite the old record with the new data

The first step is not obligitory, although to save the possible update of the wrong record or field, it is a very good idea to check with the user that the correct record has been selected for update. We will say that we will be using both steps in updating a part record. Using the above coding to read and display an individual record, we can add new coding from line 120 onwards. With the record details displayed on the screen, go through and prompt for new data for each field within the record. Coding will have to be included to allow for the data to be left alone. We could say that if ENTER is pressed on its own then the field is to be left as it is. The updating of the record could be enhanced and made more user friendly by using cursor postioning. For the following example cursor postioning has been omitted for readability :

```
.
.
Lines 10 to 110 as above
.
.
120 PRINT "ENTER NEW DESCRIPTION ";: INPUT A$
130 IF A$<>"" THEN A$(1)=A$
140 PRINT "ENTER NEW QTY IN STOCK ";: INPUT A$
150 IF A$<>"" THEN A$(2)=A$
160 PRINT "ENTER NEW RE-ORDER QTY ";: INPUT A$
170 IF A$<>"" THEN A$(3)=A$
180 PRINT #FD$, I
190 FOR I1=1 TO 3
200     PRINT A$(I1)
210 NEXT I1
220 CLOSE
```

Line 120 to 170 accepts the new data from the user, for each field on the file. If A$<>"" (A$ is not equal to "") then the user has NOT pressed ENTER on its own, therefore a new value has been entered for the field, so store in A$( ) array. The last step in updating a record is to PRINT the new data over the old data. This is done in the same manner as writing a new record. Using the record number of the record to be updated, set the file pointer, using the record number/part number (see line 180), then use a FOR/NEXT loop to PRINT the three fields. Do not forget to CLOSE the file if you are going off to do another task within the program. A system crash after PRINTing the updated record, but before the CLOSE statement, could result in you losing the update.

Generally, when updating, you will re-PRINT all fields within the record, regardless of wether a field was unchanged or not. The updating of each individual field is 'sequential' in its manner. The update process updates the first field, then the second field and so on. Therefore, you can not jump to the 3rd field and update, then say, jump to the 6th field. and update. You could however, update the first and second field, then ignore the trailing fields within the record. This would only be required where the record is quite large, and requires a lot of disk access to read a few records.

4.4 Deleting records from Random - Relative files.

You can not delete a record from Xtal/Basic in the true sense of the word. You can only simulate a deletion. A common method of 'deleting' records in Basic, is to update the record filling it with control-characters or null characters. Control-characters and null characters can not be directly entered from the keyboard by the user, therfore the only way that these characters should of gotten onto the data file is by the program being coded to do so.

Because 'deletion' is a glorified update most of the work

involved has been described in the previous section. Use the two steps described above :

1 Read the record so the user can check it is the correct record to be deleted.

2 Overwrite the record with one or more control characters to show it has been deleted.

Refer to the previous section to see how an individual record was read from the file and displayed on the screen. With the record displayed on the screen you can now ask the user if this is the correct record to be deleted. If they answer 'yes' then the next stage is to update one or more of the fields. It is quite a common practice to use null characters (Ascii value 0) to signify deletion of a record, or high values (Ascii value 255). When you have chosen the characters, place them into one or more of the fields. Which, or how many fields will contain the 'deletion' characters depends. If there is a possibiliy that you will want to 'unset' the deletion at a later date, then a good idea is to have a seperate one character field, set to ascii 0 if it is deleted, or set to ascii 1 if not deleted. This method means that your data is not actually overwritten and may be recovered at a later date. When you read a file using this deletion method don't forget to check to see if the record has been flagged as deleted.

Following on from the 'one character field' deletion method, another way is to set the whole record with ascii 0 and then rewrite the record. The following coding shows two programs, the first sets the first field to ascii 0's (deleted), after first checking that the record hasn't already been deleted !! The second program reads a record from the file and checks to see if it is flagged as deleted. We will use the PARTS.DAT used before :

1.
```
1  REM THIS DELETES AN INDIVIDUAL PARTS RECORD**************
10 OPEN "PARTS.DAT", FD$,33
20 INPUT #FD$,0,N: INPUT #0
30 PRINT "ENTER PART NUMBER TO BE DELETED ";: INPUT I$
40 I=VAL(I$)
50 IF I<1 OR I>N THEN PRINT "DOES NOT EXIST": GOTO 30
60 INPUT #FD$,I
70 FOR I1=1 TO 3: INPUT A$(I1): NEXT I1: INPUT #0
80 IF A$(1)=MUL$(CHR$(0),20) THEN PRINT "ALREADY  DELETED":
                                  GOTO 30
90 PRINT "PART NUMBER      ";I
100 PRINT "DESCRIPTION      ";A$(1)
110 PRINT "QTY IN STOCK     ";A$(2)
120 PRINT "RE-ORDER LEVEL   ";A$(3)
130 PRINT: PRINT "DO YOU WANT TO DELETE THIS PART ? ";
140 INPUT A$: IF A$<>"Y" AND A$<>"N" THEN GOTO 130
150 IF A$="N" THEN GOTO 30
160 A$(1)=MUL$(CHR$(0),20)
170 PRINT #FD$,I;I: PRINT A$(1): PRINT #0
180 PRINT "PART DELETED": GOTO 30
```

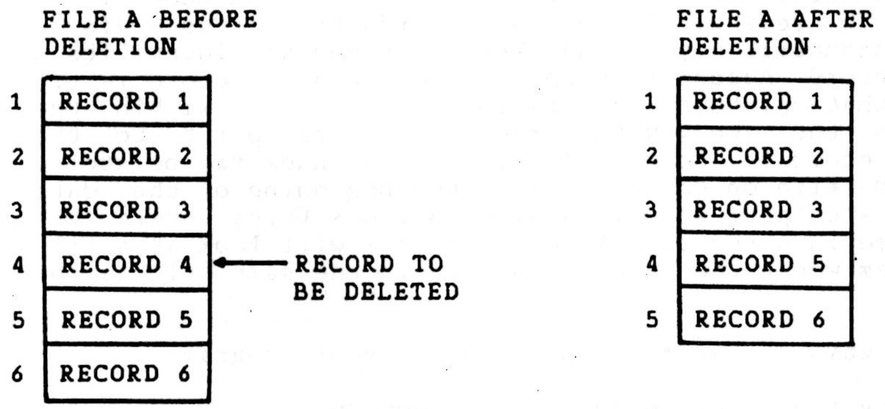
2.
```
1   REM THIS PROGRAM READS INDIVIDUAL RECORDS AND CHECKS TO
2   REM SEE IF THEY HAVE PREVIOUSLY BEEN DELETED
10  OPEN "PART.DAT",FD$,33
20  INPUT #FD$,0;N: INPUT #0
30  PRINT "ENTER PART NUMBER ";: INPUT I$ : I=VAL(I$)
50  IF I<1 OR I>N THEN PRINT "PART DOES NOT EXIST": GOTO 30
60  INPUT #FD$,I
70  FOR I1=1 TO 3: INPUT A$(I1): NEXT I1
80  IF A$(1)=MUL$(CHR$(0),20) THEN PRINT "ALREADY DELETED":
                                    GOTO 30
90  PRINT "PART NUMBER    ";I
100 PRINT "DESCRIPTION    ";A$(1)
110 PRINT "QTY IN STOCK   ";A$(2)
120 PRINT "RE-ORDER LEVEL ";A$(3)
130 GOTO 30
```

The first field in the file is the Part Description. This field has been used to set the record to deleted. Using this method there is no way that the description can be recovered from the same field, so be careful when chosing wether to have a seperate 'delete flag' field, or using an existing field.

If the record does not need to be kept for possible historic reasons, or possibly to be 'undeleted' at a later date, then that record will provide an empty space for a new record. However, to find any possibly deleted records, in which to insert a new record, the file needs to be read from the start until a record flagged as deleted is found. Reading through a file to find one possible deleted record can be a time consuming job with all the disc I/O involved. There are several ways around this. One way is to read through the whole file when first starting the program, and store all the record numbers of the records deleted. Another way would be to do a reorganisation every now and again. This involves periodically reading through the data file and transfering all none-deleted records across to another file, maybe on the second disk drive if you have one. When all none-deleted records have been transfered, you can ERAse the old file and RENAME the newly created file TO the old file name. This process is particularlly suitable to a 'batch processing' system.

A final method of deleting a record that does not have to be kept for historic reasons or possible 'un-deletion' is to move the last record from the end of the file and write it over the deleted record. If you do this, don't forget to reduce the 'record counter' in record 0, by 1. This method is not suitable to systems where the 'record number' is relavant to a particuler data record. For example, in the Parts System mentioned before in this chapter, the record number also acts as the 'Part Number'. If a record containing details on say a '13 amp plug' has the Part Number of 67, then the record will be found in the 67th record from the start of the file. Moving this record from the end of the file and overwriting it onto a deleted record means that Part Number 67 will no longer be in record 67 as expected. So be carefull in using this deletion method. The diagram below

shows how this last system operates.



Chapter 5 Random index files.

Section 2.3.3 decribes briefly the theory behind Random Index data files. To review, individual records are accessed via a 'list' of alphanumeric keys. Each key is unique and identifies a single data record. Access from the key to the data record is via a 'pointer' that points to the postion of the record within the data file, in the same way that record numbers point to the postion of a record in 'Relative' files. The index may be stored in a seperate file on the disk or at the beginning of the data file. In this chapter we will deal with index files where the index is stored in a seperate file. Chapter 7 will look at a file handling system where the index is stored in the same file as the data.

There are two ways of handling the index in your program :

1) Holding the index in an array in memory
2) Holding the index on the disk

The first option has the advantage of being faster when searching for a single index key but is limited by the number of keys that can be held in an array in memory. We will deal with the first option. I have had very few problems with indexes in arrays blowing memory. I generally find that the disk size for data storage is the limitation, not the memory size. Memory size problems can be overcome using the HOLD function or splitting a program up into several programs etc. If we have a data record 100 bytes long and each index key is 10 bytes long, then the total storage is 110 bytes per record. A formatted disc is 196k or 200704 bytes (1024 X 196). 200704 divided by 110 shows that we could store upto 1824 records on a blank 3" disk. If we had 1824 index keys (the maximum in this case) and each key is 10 bytes long, then the array would take up 18k.

This chapter will continue to show you how to use indexes with data files through an example. The data file is to hold information on employees in a company. Each record contains the following :

Field description	Field type	Field length
Employee name	alpha	15
National Ins No	numeric	10
Weekly rate of pay	numeric	6
Tax code	alha/numeric	5

Each data record is to be accessed via an alphnumeric key 6 characters long. If you also add the 2 caracters to allow for the carriage return, this brings the length of each key, to 8 bytes. The data record contains 36 bytes of data, plus 2 bytes for each field brings the record length to 44. We will add 10 bytes onto the length of the data record to allow for possible extra data to be stored at a later date. The key will consist of the first

three characters of the employees surname, followed by a 3 digit number. The reason for this number is to keep the keys unique. If there are 6 Smith's, then the first Smith we will give a key of SMI001. The second, a key of SMI002 and so on.This way, no two keys will be the same. We mustn't forget to add on the number of bytes to be allocated to the key for the pointer. If we allocate 4 bytes, then this will allow a maximum value of 9999 and increase the key length to 12 bytes. This will be more than enough. To check how many records, along with the keys, you can store on a disk, add the key length of 12 to the data record length of 54, and divide into 198656. This shows we can store a maximum of 3009 data records and keys. It may seem a lot, but bear in mind, the record layout has been shortened to 4 fields for ease of coding this example. In a real situation there is likely to be between 100 and 200 bytes per record.

The first record of the index file (record 0) will contain the count of the number of keys on the index at any one time. This is the same as the relative file discussed in the previous chapter. When using these files, the first step is to CREATE the empty index and data files. The index file needs to have record zero (count of the number of keys) set to 0. The following coding will do this.

```
10 REM THIS PROGRAM WILL CREATE AN EMPTY INDEX AND DATA FILE
20 CREATE "WORKERS.DAT",DAT$,54
30 CREATE "WORKERS.IND",IND$,12
40 PRINT #IND$,0;0
50 CLOSE
```

When writing file handling programs, allways keep a copy of the 'Create program' on your disk incase the index or data file gets screwed up during the testing stage or even later.

We are now ready to start storing data onto the files.

5.1 Storing data on Random Indexed data files.

There are several steps involved in storing data and updating the index. These steps are :

1. Look through the array index for a deleted key
2. If none deleted, then add the key onto the end of the index
3. Look through the pointers for an empty data record. (Any empty data records will not have a pointer in the index.)
4. Store the data in the data record
5. Add the new key to the index array in memory and update the index on disk.

6. If the key was added onto the end of the index then add 1 to the counter in record 0 of the index file

It can be seen that random index files require quite a lot more coding and thought, however, if you code in a modular form then the coding could be quite easily used for further programs. We will assume that you have created the index and data file, though we will not assume that some data records already exist.

```
1   REM ************** READ INDEX INTO ARRAY *******
10  OPEN "WORKERS.DAT",DAT$,54
20  OPEN "WORKERS.IND",IND$,12
30  INPUT #IND$,0;N
40  DIM IN$(1000)
50  DIM RE%(1000)
60  FOR I=1 TO 1000: RE%(I)=0: NEXT I
70  FOR I=1 TO N
80     INPUT #IND$,I: IN$(I)=INCH$(10)
90     RE%(VAL(RIGHT$(IN$(I),3))=1
100 NEXT I: INPUT #0
110 REM ************** ENTER NEW RECORD ************
120 PRINT "ENTER EMPLOYEE NUMBER (XXXNNN) ";:INPUT A$(1)
130 FOR I=1 TO N
140   IF IN$(I)=A$(I) THEN PRINT "ALREADY  EXISTS":GOTO 120
150 NEXT I
160 FOR I=1 TO 1000
170   IF RE%(I)=0 THEN REC%=I: GOTO 220
180 NEXT I
190 PRINT "NO ROOM IN DATA FILE": GOTO 9999
220 PRINT "ENTER EMPLOYEE NAME ";: INPUT A$(2)
230 PRINT "ENTER NAT INS NO    ";: INPUT A$(3)
240 PRINT "ENTER RATE OF PAY   ";: INPUT A$(4)
250 PRINT "ENTER TAX CODE      ";: INPUT A$(5)
260 REM ***************** STORE DETAILS IN FILES **********
260 PRINT #DAT$,REC%
270 FOR I=1 TO 4: PRINT A$(I): NEXT I
290 FOR I=1 TO N
300     IF IN$(I)=MUL$(CHR$(0),10) THEN GOTO 320
310 NEXT I: I=I+1
320 IND%=I: PRINT #IND$,IND%: FMT 3,0: REC$=STR$(REC%)
330 IN$(REC%)=A$(1)+REC$: PRINT IN$(REC%): PRINT #0:
340 IF IND% > N THEN N=N+1: PRINT #IND$,0;N
9999 CLOSE: END
```

Line	Description
10 - 20	OPEN the previously CREATED data and index files.
30	Get the number of the last index key in the index file
60	Initialise 'data record numbers available for new data' array. 0 means available, 1 means already contains a data record.
70 - 90	Go through the index and get each key from the index file and store in the array IN$( ). The last 3 characters of each key is the pointer to the data record so set the data record (RE%(?)) to 'full'.

120	Ask user to enter key for new record. Must be 6 characters long. Store in A$(1).
130 - 150	Loop through the index array to see if the key entered already exists on the index.
160 - 180	Loop through to find an empty data record. Store the new pointer in REC%
190	If the program gets here then all 1000 data records are full.
220 - 250	Ask the user to enter the data fields. No length or numeric validation has been written, but this would be required.
260	Set the output channel pointing to the new data record slot.
270	Output the new data record to the data file.
290 - 310	Now the data record has been stored, you are ready to store the new key in the index. Look through the index array for a deleted key (all null characters in this case) and store the position in IND%. If no deleted keys are found then add 1 and store on the end of the index.
320	Set the output channel ready for PRINTing the new key on the index file. Convert the pointer to 3 characters by adding leading zeroes.
330	Store the new key in the index array, and PRINT the new key on the index file.
340	If the new key was added onto the end of the index then the counter in record zero will need to be updated by +1.

The above example is not by any means the quickest or most effeicent way of coding the above, but hopefully it is quite 'readable' for the beginner. Ways of making it more efficient include the following :

1) Add all new records onto the end of index regardless of whether any keys have been deleted.

2) Only look for the first deleted key, if any, and look for the next deleted key if a second new data record is to be added.

3) Have some periodical batch processing which physically removes deleted keys and shuffles the remaining keys up the index.

A good idea is to have a Menu selection with add,delete and amend as 3 seperate options. Then only if the 'add' option is selected do you need to look for new data and index slots.

When having two seperate files for the index and data records then it is a good idea to first add the new data record THEN add the new key to the index. This way, if a computer crash occours between the two file updates the index will not be corrupted. You will however loose the data record entered, as this will appear as deleted or past the end of the index.

5.2 Reading data from a random indexed file.

Reading is a far simpler process than adding a record. The steps involved are :

1) Open the data and index file

2) Get the counter of the number of keys in the index

3) Load the index from the index file into an array

4) Ask the user to enter the key of the record they require

5) Search the index array for the key. If it is not there then send error meaasge, 'not found', to the user.

6) If it is there then get the pointer from the key and go and read the data record and display/print.

Converting the above steps into coding, and using the Employee File used in the last section, you should end up with something like this :-

```
10 REM **********READ A SELECTED RECORD FROM AN INDEX FILE**
20 OPEN "WORKERS.DAT",DAT$,54
30 OPEN "WORKERS.IND",IND$,12
40 DIM IN$(1000)
50 INPUT #IND$,0;N
60 FOR I=1 TO N
70     INPUT #IND$,I: IN$(I)=INCH$(10)
80 NEXT I: INPUT #0
90 PRINT "ENTER THE RECORD KEY ";: INPUT K$
100 FOR I=1 TO N
110     IF IN$(I)=K$ THEN GOTO 140
120 NEXT I
130 PRINT "DOES NOT EXIST": GOTO 90
140 REC%=VAL(RIGHT$(IN$(I),3)): IND%=I
150 INPUT #DAT$,REC%
160 FOR I=1 TO 4 : INPUT A$(I): NEXT I: INPUT #0
170 PRINT "EMPLOYEE NUMBER ";LEFT$(IN$(IND%),6)
180 PRINT "EMPLOYEE NAME   ";A$(1)
190 PRINT "NAT INS NUMBER  ";A$(2)
200 PRINT "RATE OF PAY     ";A$(3)
210 PRINT "TAX CODE        ";A$(4)
220 CLOSE
```


The above program explains how to read an individual record. To read all the records within a data file is slightly quicker in coding terms :

```
10 REM ******* TO READ ALL THE RECORDS IN AN INDEX FILE*****
20 ...line 20 to 80 as in the above example
.
.
80
90 FOR I=1 TO N
100     REC%=VAL(RIGHT$(IN$(I),3))
110     INPUT #DAT$,REC%
120     FOR I1=1 TO 4: INPUT A$(I1): NEXT I1
130     PRINT "EMPLOYEE NUMBER ";LEFT$(IN$(I),6)
140     PRINT "EMPLOYEE NAME   ";A$(1)
150     PRINT "NAT INS NUMBER  ";A$(2)
160     PRINT "RATE OF PAY     ";A$(3)
170     PRINT "TAX CODE        ";A$(4)
180 NEXT I
190 CLOSE
```

To read all the data records, access must be via the index. You can not read the data records directly, as deleting involves removing the key entry only, and not a physical deletion of the data record. Also, the key is an integeral part of the data for each record (the employee number in this case) and therefore needed. Direct reading of the data could give unexpected results and 'garbage' data.

5.3 Updating data in random index files.

A decision needs to be made, as to whether or not the user of a random-index file is allowed to change the index key at any stage. For example, customer numbers are generally set once for each customer, and remain that number. If the user stops trading with a customer, the number is held back just in case trading is resumed. Quite often, customer details such as address etc are stored in a seperate file. These details are accessed by the customer number held in a transaction record. If the customer number on the customers detail file was changed, you would get a 'can't find customers details' error when the customer number on the transaction file tries to find the corresponding customers details. This is sometimes the same with part numbers, tax reference codes etc. However, ther are cases where the key may be changed at any time to allow for typing errors etc.

It is quite simple to allow for the two above possibilities. If an ammendment of the key is not allowed, display the field, but do not prompt the user to enter a new value.

In any updating of a data record, it is quite common, though not always the case, to allow the user to see the records fields on the screen before asking the user to enter new values for one or more of the fields. So the first operation will be to ask the

user to enter the record key. Check the key exists on the index, read the data files using the pointer then display the record details on the screen. Now you can go through each field (excluding or including the key field ?) and prompt for a new value for the field. Don't forget to validate the data and check the length to avoid truncation of the field data. Finally you can rePRINT the record using the pointer as the record locator. To summarise the steps :

1) Ask the user to enter the record key

2) Search through the index array to check the key exists

3) If it exists then read the data record from the disk and display

4) Prompt for a new value to be entered for each field and validate the input

5) Rewrite the updated record onto the disk using the same record pointer as when the file was read

6) Close the file

Below is an example of how the above could be coded. The previous example of the Employees File will be used. The key, employee number, can be changed in this example. Generally employee numbers would remain for one particular employee even after the employee has left the company. This is because tax offices may refer to an employers previous employees using the original employee number as the reference etc.

```
10 REM ******** INITIALISE INDEX ARRAY ********
20 OPEN "WORKERS.DAT",DAT$,54
30 OPEN "WORKERS.IND",IND$,12
40 DIM IN$(1000)
50 INPUT #IND$,0;N
60 FOR I=1 TO N
70     INPUT #IND$,I: IN$(I)=INCH$(10)
80 NEXT I: INPUT #0
90 REM ********* ASK USER FOR KEY AND GET THE RECORD *****
100 PRINT "ENTER EMPLOYEE NUMBER (XXXNNN)  ";: INPUT E$
110 IF LEN(E$) <> 6 THEN PRINT "INVALID NO.": GOTO 100
120 FOR I=1 TO N
130     IF IN$(I)=E$ THEN IN%=I: GOTO 160
140 NEXT I
150 PRINT "DOES NOT EXIST IN INDEX ARRAY": GOTO 100
160 REC%=RIGHT$(IN$(IN%),3): INPUT DAT$,REC%
170 FOR I=1 TO 4: INPUT A$(I): NEXT I: INPUT #0
180 PRINT "EMPLOYEE NUMBER    ";LEFT$(IN$(IN%),6)
190 PRINT "EMPLOYEE NAME      ";A$(1)
200 PRINT "NAT INS NUMBER     ";A$(2)
210 PRINT "RATE OF PAY        ";A$(3)
220 PRINT "TAX CODE           ";A$(4)
```



```
230 REM ****** PROMPT FOR NEW DATA FOR EACH FIELD**********
240 REM ****** IF ONLY ENTER IS PRESSED CURRENT DATA ******
250 REM ****** IS LEFT UNCHANGED **************************
260 PRINT "ENTER NEW EMLOYEE NUMBER  ";
270 INPUT Z$: IF ASC(Z$)=13 THEN GOTO 300
280 PRINT #IND$;IN%:FMT 3,0
290 Z$=MUL$("      ",7-LEN(Z$))+Z$+STR$(REC%)
290 PRINT Z$
300 PRINT "ENTER EMPLOYEE NAME  ";: INPUT Z$
310 IF ASC(Z$) <> 13 THEN A$(1)=Z$
320 PRINT "ENTER NAT INS NUMBER  ";: INPUT Z$
330 IF ASC(Z$) <> 13 THEN A$(2)=Z$
340 PRINT "ENTER RATE OF PAY ";: INPUT Z$
350 IF ASC(Z$) <> 13 THEN A$(3)=Z$
360 PRINT "ENTER TAX CODE  ";: INPUT Z$
370 IF ASC(Z$) <> 13 THEN Z$(4)=Z$
380 PRINT #DAT$,REC%
390 FOR I=1 TO 4:PRINT A$(I):NEXT I
400 CLOSE
```

Line 280 and 290 makes sure that the new index key is 10 characters long and that the data record pointer is the last three characters of the key.

If you do not wish to allow for the update of the index key then omit lines 260 to 290. It is a good idea to CLOSE the data file and index file after updating a single record and index key. This ensures that the contents of the file buffer have been written to disk and that there will be no loss of new data in the event of a system crash. The problem with doing this is one of time. Any disk I/O slows down the running of any program, and this includes a CLOSE. If therefore, you are reading through all records in a file and updating without prompting from the user then the CLOSE should be left out to keep the updating time to a minimum.

5.4 Deleting a data record.

The data record in a deletion, is not physically deleted, all that is done is that the index key is removed from the index. This has the effect of removing the pointer from the index so that there is no way that a program accessing data via the index would give access to the data record. Deletion of the key can be done in a variety of ways.

1) Bring the key from the end of the index and overwrite the key to be deleted. The count in record zero of the number of keys in the index, would have to be reduced by 1.

2) Set the key to spaces or a certain character eg ascii 0, so when reading the index, any index key containing these characters are treated as deleted and may be used for storing a new record.

The first method requires a sort after deletion and/or adding a record to put the keys into alphanumeric order. The second method

only requires a sort after a new record key has been entered. Taking the Employee data file example as used before, the following example will show you how to set a key to 'deleted' by putting 10 ascii 0 characters into the key. This was used in the example in section 5.1 'Storing a record'.

The first part of the coding is the same as the previous coding example above. The user will have to enter the employee number of the record to be deleted. Once this is known, the program can scan the index for the key entry and display the record and allow the user to confirm that the record is to be deleted.

```
10
.
.
.
.
.   same as previous example
.
.
.
220
230 PRINT:PRINT "IS THIS THE RECORD TO BE DELETED ? ";
240 INPUT Z$: IF Z$ <> "N" AND Z$ <> "Y" THEN GOTO 230
250 IF Z$ = "N" THEN GOTO 270
260 Z$=MUL$(CHR$(0),10): PRINT #IND$,IN%: PRINT Z$
270 CLOSE
```

5.5 Sorting the index.

When adding a new key onto the end of the index or into a key flagged as deleted the index soon becomes unsorted. This destroys the beauty of an index random file. Also, as will be seen in the next section, a sorted index can be searched at a far greater speed than an unsorted index. In previous examples the index has been stored in an internal array. If you refer back to section 3.5 there is an example and disscussion on Binary Chop sorts and Bubble sorts. Either of the two may be used in sorting the array.

When sorting, it may be a good idea to take the opportunity to tidy up the index. The tidying up involves physically removing keys flagged as deleted. This can be done when reading the index off the disk and storing in an array. If the key is flagged as deleted, do not store the key in the array and go and read the next key from the index file.

The following example sorts the Employee index and removes any keys flagged as deleted using the Bubble sort routine.


```
10 OPEN "WORKERS.IND",IND$,12
20 INPUT #IND$,0;N: C=0
30 DIM A$(1000)
40 FOR I=1 TO N
50 Z$=INCH$(10)
60 IF Z$=MUL$(CHR$(0),10) THEN GOTO 80
70 C=C+1: A$(C)=Z$
80 NEXT I
90 IF C  < 2 THEN GOTO 165
100 C1=C
110 FOR I=1 TO C
120 C1=C1-1
130 FOR I1=1 TO C1
140 IF A$(I1) > A$(I1+1) THEN SWAP A$(I1),A$(I1+1)
150 NEXT I1
160 NEXT I
165 REM
```

The above coding reads the index into A$( ). Any deleted keys are ignored at line 60, the array is Bubble Sorted. The next step is to update the index file on the disc with the sorted keys.

```
170 CREATE "WORKERS.IND",IND$,12
180 FOR I=1 TO C
190 PRINT #IND$,I:PRINT A$(I)
200 NEXT I
210 PRINT #IND$,0;C
220 CLOSE
```

The line at 210 updates the key count in record zero. If the above coding fails due to a system crash the index will be lost forever. A way around this would be to store the newly sorted index into a file named "WORKERS.TMP" (temporary file) and use the RENAME verb to change the file name. Using this method will keep the original index until the sort has been completed so that a system failure would mean that the user could always backup the original index in the event of a system crash.

The binary chop sort could be used to possibly speed up the sort.

5.6 Fast searching of an index - Binary chop.

This routine could be used to find one particular key within a SORTED index. The theory behind the routine involves halfing the index. If the key being searched for is less than the key in the middle of the index then the number of keys in the bottom half of the index is halved again. If the key being searched for is in the top quarter then the quarter is halfed and the sequence is performed again. This halving continues until the key being searched for if found. Using this method reduces the search time many fold and so reduces the search time accordingly. The coding is as follows. The variable N contains the number of keys in the index and the keys are stored in the array A$( ).

```
100 PRINT "ENTER KEY TO FIND ";: INPUT A$
110 A1$=A$+MUL$(" ",7-LEN(A$)
120 LL=0: LAST = N : UL= LAST : REC=INT(UL-LL)/2
130 B$=LEFT$(A$(REC),7)
140 C=C+1
150 IF A1$=B$ THEN GOTO 1000
160 IF A1$=LEFT$(A$(REC-1),7) THEN REC=REC-1: GOTO 1000
170 IF A1$=LEFT$(A$(REC+1),7) THEN REC=REC+1: GOTO 1000
180 IF LL=REC-1 AND UL=REC+1 THEN PRINT "NOT HERE": GOTO 1000
190 IF A1$>B$ THEN LL=REC:REC=INT((UL-REC)/2+LL) : GOTO 130
200 IF A1$<B$ THEN UL=REC:REC=INT((REC-LL)/2): GOTO 130
1000 PRINT "KEY FOUND IN THE ";REC;"TH ELEMENT OF THE INDEX"
```


Chapter 6. Data compaction.

6.1 Reducing the carriage return/line feed to 1 character.

When storing each record in the files discussed so far, each field within a record has two bytes added to the record length to allow for the carriage return and line feed. These two characters per field are used as delimiters (end of field) by the INPUT statement when reading a field from a disk.

The user can use the IOM(6,0) statement (page 119 Basic Manual) to output a C/R only when PRINTing data to a file. Use the statement in the following manner :

```
100 IOM(6,0): PRINT #FD$,1
110 FOR I=1 TO 4
120 PRINT A$(I)
130 NEXT I
140 IOM 6,1: CLOSE
```

In the above example assume the file is OPEN. The FOR/NEXT loop PRINTs 4 fields to a file record. Using the IOM statement the overall length of the record has been reduced by 4 bytes. Removing the L/F character does not affect the INPUT statement as the INPUT statement is just as happy to delimit on a C/R only. Set bit 6 back to auto line feed character straight after PRINTing to the file, so that other PRINTing statements in your program are not affected.

6.2 Removing C/R from between fields.

Using semi-colons between each field will mean that no C/R or L/F will be PRINTed. The effect of the PRINT statement with semi-colons is similar to the PRINT statement on the screen with semi-colons. If you have 4 fields within a record, and each fields is 10 bytes long then using the IOM statement mentioned above the record length will be 44 bytes. By putting semi-colons between each field the total length of the record will be 10+10+10+10+1. The one byte at the end is the C/R resulting from the PRINT statement with IOM(6,0) set. Put into coding as follows :

```
100 PRINT #FD$,1: IOM(6,0)
110 PRINT A$(1);A$(2);A$(3);A$(4)
120 CLOSE: IOM(6,1)
```

When using this method of storage each individual field must be a set length decided upon at data file creation stage. The reason is because when the INPUT statement is used it delimits on the C/R, so all the four fields will be dumped into the INPUT variable in one go. The individual fields can be broken down into their own variables using LEFT$,RIGHT$ and MID$. Using these three statements requires the user knowing where the start and

end of each field is. Coding to extract the above record from the disk and break in down to fields may be as follows:

```
100 INPUT #FD$,1
110 INPUT Z$
120 FOR I=0 TO 3
130 A$(I)=MID$(Z$,I*10+1,10)
140 NEXT I
```

Always make sure that the FOR/NEXT loop is set up for the correct number of fields and that the array A$( ) has been dimensioned accordingly.

There is a limit to the amount of characters that can be PRINTed to a data file in one PRINT statement. The limit is 128 characters. If the record is for example, 150 characters long excluding any C/R's or L/F's then the printing of the record will have to be broken down into 2 PRINT statements as follows. Assume that the record contains 5 fields, each field 30 bytes long, and held in the array A$( ).

```
100 PRINT #FD$,1: IOM(6,0)
110 PRINT A$(1);A$(2);A$(3): PRINT A$(4);A$(5)
120 CLOSE: IOM(6,1)
```

The first PRINT statement will put a C/R at the end of the first three fields and the second PRINT will put a C/R at the end of the third and fourth field. Therefore with C/R's the record will contain 152 bytes. Not using IOM(6,0) would mean 154 bytes being required per record. Without the semi-colons and IOM(6,0) the record length would be 160 bytes.

Reading the 5 fields from the disk could be done using two INPUT statements and the splitting each field up using MID$, LEFT$ and RIGHT$. Again, the length of each field should be decided and kept to, when writing the record to the disc.

6.3 Storing numeric data.

When storing a number, say 232323.33, storing as a string would involve 9 bytes being taken up. Storing the number 122 would take up 3 bytes. Storing numbers as strings, whilst acceptable, it is somewhat wastefull of disk space. A disk saving method is to break numbers down into ascii codes using a simple formula. Described below are three numeric ranges in which numbers may be stored. The ranges may be extended by the user however, by expanding the coding. The ranges covered in this manual are:

1) 0 to 255 (stored as 1 byte)

2) 0 to 65025 (stored as 2 bytes)

3) 0 to 16581375 (stored as 3 bytes)

With all three ranges there are two steps in using them. Before storing on disk, the numeric data needs to be 'converted' from its numeric value to its ascii value. When reading from the disk the ascii value needs to be 'reconverted' to its numeric value before any computation may be performed.

Numbers in the range 0 to 255 may be stored in 1 byte on the disk. The number is converted for storage using the CHR$( ) statement. (see page 47 of the Basic manual). Therefore the number is stored as an ascii character, not as the number entered. Storing the number 125 would be coded as follows :

```
100 PRINT #FD$,1: PRINT CHR$(125)
```

The coding stores the number in the file OPEN on FD$ in record number 1.

To read the number back from the file and convert back to a numeric value is coded as follows :

```
110 INPUT #FD$,1: INPUT N$: N%=ASCII(N$)
```

The ASCII statement can be found on page 36 of the Basic manual. There is an important note at the end of this section to allow for the program not being confused by CHR$(9) (TAB) and CHR$(25) (End of file).

Numbers in the range 0 to 65025 can be stored in two bytes on the disk. The conversion from numeric to a two byte ASCII string involves the number being divided by 255. The number by which the value can be divided by 255 is stored in the first byte and the remainder stored in the second byte. The coding, as a GOSUBed routine is as follows :

```
1000 REM ****** N is the number to be stored ******
1010 A1$=CHR$(INT(N/255))
1020 A2$=CHR$(N-INT(ASCII(A1$)*255)
1030 Z$=A1$+A2$:
1040 RETURN
```

Z$ contains the 2 byte string that represents the number N. The variable Z$ may be PRINTed to the data file.

When reading the 2 byte string from the file the string needs to be converted back to a numeric item. N, below, is the coding for the string to numeric conversion.

```
2000 REM *******CONVERTS A TWO BYTE STRING TO A NUMBER****
2010 REM *******Z$ IS THE INCOMING 2 BYTE STRING *********
2020 A1$=LEFT$(Z$,1): A2$=RIGHT$(Z$,1)
2030 N=ASCII(A1$)*255+ASCII(A2$)
2040 RETURN
```

There is an important note at the end of this section to allow for the program not being confused by CHR$(9) (TAB) and CHR$(25) (End of file).

Storing a number in the range 0 to 16581375 in a three byte string works on the same principle as with the two byte string. The following two routines convert from a number to a three byte string and vica versa.

```
3000 REM **** CONVERTS NUMBER N INTO A THREE BYTE ********
3010 REM **** STRING Z$
3020 A1$=CHR$(INT(N/65025)): N=N-(ASCII(A1$)*65025)
3030 A2$=CHR$(INT(N/255)):   N=N-(ASCII(A2$)*255)
3040 A3$=CHR$(N): Z$=A1$+A2$+A3$
3050 RETURN
```

If you were to store the number 12345678 in a three byte string, the ASCII value of each byte will be 189, 219, 108. This can be checked by the calculation :

```
189 X 65025  =  12289725+
219 X 255    =     55845+
108                  108+
                ---------
                12345678=
                ---------
```

```
4000 REM **** CONVERTS A THREE BYTE STRING Z$ TO *******
4010 REM **** A NUMBER N                         *******
4020 A1$=LEFT$(Z$,1): A2$=MID$(Z$,2,1): A3$=RIGHT$(Z$,3)
4030 N=ASCII(A1$)*65025
4040 N=N+ASCII(A2$)*255
4050 N=N+ASCII(A3$)
4060 RETURN
```

As mentioned before, when handling ASCII values 9 (TAB) and 25 (End of file) Basic treats them as described in the brackets. This can be overidden using the IOM statements IOM(3,0) and IOM(7,0). The first IOM causes all 'end of file' markers to be ignored. The second IOM stops the expansion of the TAB character to spaces. See pages 118 and 120 of the Basic manual). Use these two statements before PRINTing and INPUTing to/from the disk. Reset them after, if required. Note that IOM(3,0) inhibits the use of SHIFT-BREAK which is very useful when testing/debugging programs!

If you require negative numbers, as well as positive numbers, to be stored then add a leading or trailing byte onto the string with the sign stored. Alternatively, half the maximum number to be stored, ie 65025 in a two byte string, and allow the user to store any number in the range -32512 and +32512. Using this method requires the value 32512 to be added to the number when converted from a number to a two byte string. After converting from a three byte string to a number subtract 32512 to give the original number stored. This principle can be used on 1 and three

byte number strings.

To store 2 decimal places multiply the number by 100 before converting to a string, and divide the number by 100 after converting from a string to a number. This will reduce the range of numbers that may be stored by a factor of 100.

Adding a small amount of extra coding to the above routines, the user may create four, five and more byte strings which may be used to store numbers. Which ever length string is used, ensure that two byte strings are converted back to a numeric using the two byte conversion routine, not the one byte etc. It may be a good idea to put range checks in the coding, with appropriate user error messages to stop the possibility of the user trying to store the number 300 in one byte and so on.

Chapter 7. Advanced techniques in using Random Indexed files.

There are several ways in which the Random Index files discussed in chapter 5 can be improved upon. Disk access slows down the overall speed at which a program operates. The more disk access, the slower the program operation. When the computer reads the disk it pulls in a chunk of characters in the memory. If you are reading the index keys in, then the more compact the data the less number of reads to pull all the index keys into an array. Keeping the index in an array brings us to another problem. The size of the array is inflexable and depends on the DIM statement. Also, if the array contains 2000 keys and pointers, and a large amount of random access is required, then scanning through the index array for each key can amount to a noticable time delay.

A third problem is keeping the array in a sorted order. Slotting a new key into the index invloves finding where to put the new key, then shuffling all the keys below down by one array element. Or, altenatively, the index can be sorted periodically. This last method involves a period where the user is kept waiting for some time according to the number of keys to sort.

All the above problems can be overcome using more advanced file structure techniques. This chapter is written to describe a possible index structure, but without programming examples. The system is quite complicated and any coding in this manual would not help in grasping the overall understanding of the principles involved.

7.2 Detailed description of Advanced Random Indexed Data files.

The index to the file will usually be structured over two levels, the 'low level' and the 'high level' index.

The low level index will contain the key and pointer to the file for each data record on the file. The low level index will be subdivided into smaller 'units'. The enteries will be in strict sequence within the low level index. Following reorganisation the low level index 'units' will only contain a percentage of the maximum entries possible, to allow for insertions.

The high level index will contain the lowest key in each unit of the low level index. This will allow the program to quickly search through the high level index and decide which of the low level units to search for the index key and pointer.

A 'header record' (previously record 0 has been used as a header record with the 'number of records on file count') will be required to hold the following information :

1) Pointer to the first free record
2) Number of actual records on file
3) Maximum number of records on file
4) Number of bytes per record
5) Key length
6) Index unit density.

The above numeric items stored as a two byte string (see section 6.3) will allow a maximum of 65025 records to be held.

An example of an index design using this method of access on a file of 1000 data records, each record of 128 byte with a key of eight bytes and an index density of 80% follows.

Each unit of the low level index will be 128 bytes.

The header record and index units will be held at the beginning of the data file. They could be stored in a seperate file if required.

Header record. Record number 0.

Description of field.	length.	type.
Pointer to the first free record	2	numeric string
Number of record on the file	2	" "
Maximum number of records on file	2	" "
Number of bytes in data record	2	" "
Key length	2	" "
Index density	2	" "

High level index record. 6 records - Records numbers 1 to 6.

Description of the field.	length.	type.
Lowest key in low level index unit 1	8	string
Lowest key in low level index unit 2	8	"
Lowest key in low level etc etc		
Lowest key in low level index unit 15	8	"

The record layout for the other 5 high level indexes is as above. The second high level index record will contain the lowest keys for low level index units 16 to 30. There will be 84 low level units.

Low level index. Repeated 84 times. Record numbers 7 to 90.

Description	length	type
Key to data record	8	string
Logical pointer to data record	2	numeric string

The above key and pointer is repeated up to 12 times according to the number of data records on file.

Keys within a low level index block are kept in strict ascending sequence.

Data records. 1000 records max. Record numbers 91 to 1090.

Description	length	type
Data.....	128	?

If the data record is not in use (free) the format of the record is :

Pointer to the next free record	2	numeric string
Spare	126	?

Accessing a record - assuming index structure as per example.

The key required will be compared to the keys in the first high level index and the second high level index until the entry higher than the key required is found. The low level index block indicated by the previous entry will then be searched. If the key is found the associated pointer can be used to access the data record. If the key is not found a 'record not on file' error should be passed to the user.

Inserting a new record into the file.

The key of the new record will be used in a search of the index, as if accessing the record. If the key is found a 'record already on file' error will be passed to the user. If the key is not found the header record will be accessed to update the number of records on file and to obtain the pointer to the first free record, this will be replaced by the pointer to the NEXT free record held in the first free record. The new data will be inserted to the file in the free record previously pointed to. The key and pointer to the new data record will be entered into the low level index unit in the appropriate position. Prior to any actual insertion the number of records on the file will be compared to the maximum number of records and warning messages should be passed to the user if nearly full. A message, warning the user that a low level unit has reached its INDEX DENSITY (in Header Record) should be passed before reorganisation. A record cannot be inserted if it's index unit is full.

Deleting a record from the file.

The key of the record to be deleted will be used in a search of the index, as if accesssing the record. If the key is not found a 'record not on file' is passed to the user. If the key is found then verification can be obtained from the user by displaying the

data record. The data record will be zeroised (filled with ASCII 0's) and the first two bytes set to point to the next free record (the previous pointer to the first free record held in the header block). The key and pointer to the deleted record will be removed from the low level index unit and the index block re-ordered. If the key was the lowest or the only key in the low level index unit the appropriate entry in the high level index will need to be altered/removed and the high level unit reordered if necessary. The number of records held in the header record will be reduced by 1 and the pointer to the first free record re-set to point to the address previously occupied by the deleted record.

Reorganising the index.

Reorganisation space equal to the size of the low level index will be created. The average number of the entries per index record will be calculated by dividing the number of records on file by the number of low level index blocks. The index entries will be transfered to the original file. The high level index will be re-created from the lowest entries in each unit of the low level index.

Maintenance programs.

One or more maintenance programs will be required to create, maintain and reorganise random index files designed in the above way. There will be required, .ASC files containing add, search and delete routines that can be appended to Basic programs when required. A third level index may be held in the header record to reduce to access time when searching through the high level indexes. This third level index would work under the the same principle as the high level index is to the low level index. The 'create' maintainance program, as well as setting up the header record and initialise pointers, will have to calculate the number of indexes required for the maximum number of data records.

The above Random Indexed file description is meant to give an outline for more serious data processing programmers. Whilst its use on 3" disk may not be worth while, serious consideration must be given to this file structure where large amounts of records are being randomly accessed on a hard disk.

Chapter 8. Round up.

8.1 Summary.

It is hoped that this manual has given you a broader understanding of using data files with Xtal Basic. Please note however, with the Xtal Basic specific comments removed, this manual will assist any programmer writing Basic programs for data files.

For those persons put off from writing file handling in Basic, there are programming languages available that maintain the indexes automatically. I have had experience of CIS Cobol which operates under C/PM and is available for the Einstein.

This manual was not written as being the B'all and end'all of Data File handling manuals, merely as a stepping stone for those persons with Basic programming experience but a bit lacking in data file structures.

REQUEST FOR AN APPLICATION FORM TO JOIN THE "UK Einstein User Group"

Please send me details of membership to the UK Einstein User Group

Name : _____

Address : _____

_____

_____

_____

_____

Postcode : _____

Please post to :
Graham Bettany, 80 Dales Road, Ipswich, Suffolk, IP1 4JR.
Tel (0473) 45907

****

NOTES