RANDOM ACCESS FILES

by Dilwyn Jones

(with much appreciated updates from Martyn Hill)

GLOSSARY OF TERMS

DATABASE - A file which holds information as a list of entries

RECORD - One complete entry in database, equivalent to one page or one entry in an address book.

FIELD - One part of a record, e.g. a street name in an address list

FILE POINTER - a value which indicates how far into the file we are, or where the next piece of data will be read from or written to.

ARRAY - A SuperBASIC list, which can store strings, floating point numbers, or integer numbers, depending on what type it was declared as in a DIM statement.

CHANNEL - A number defined in an OPEN, OPEN_IN, or OPEN_NEW statement which indicates which file or device the data is to be printed to or inputted from.

In this article I hope to show how to set up and use data files on disk so that you can create your own database and general data files.

Data files are created using the OPEN_NEW command or similar in SuperBASIC. Once you have opened the file, commands such as PRINT and PUT can send the data in an ordered fashion to the file, allowing it to be read back later.

The simplest type of data file is a simple list of names or words. In SuperBASIC we can simply hold the information in an array and just use PRINT to send it all to a file one after the other, then use the CLOSE command to finish it all off.

Here's a very simple example, where we dimension an array to hold up to 10 entries, and then the program opens a file, prints the elements of the array to the file and closes it, so that it is stored for later use.

```
100 DIM entry$(9,20)
110 CLS
120 FOR a = 0 TO 9
130   INPUT "Entry ";(a);": ";entry$(a)
140 END FOR a
150 INPUT"Filename ";filename$
160 OPEN_NEW #3,filename$
```

```
170 FOR a = 0 TO 9 : PRINT #3,entry$(a)
180 CLOSE #3
```

If you now copy the file to the screen, you can see that the data file looks like any old text file - each entry is stored to all intents and purposes like a line of text. For example, if you entered numbers when invited you should get a list which looks like this:

```
one
two
three
four
five
six
seven
eight
nine
ten
```

There are a few points of interest to look at in the above program.

On the QL, arrays dimensioned include one element of subscript 0, so DIM entry$(9,20) actually creates 10 strings (0 to 9) with a possible length of up to 20 characters. If you try to enter something longer than the value dimensioned (20) it is truncated to the first 20 characters. This is because the DIM command sets aside space for the amount of data specified and once it is set you can't change this maximum space without using another DIM command which of course destroys the original data and sets up a new definition. This is something we will need to consider when loading the information at a later date - the array should be dimensioned to match those which were used to create the file in the first place.

Note, how in line 130, the variable a is enclosed in brackets. This turns it into an expression, so that INPUT prints the value of the variable rather than inputting it.

Line 170 prints all 10 elements of the array to the file. An alternative to this is to change line 170 to print the entire array in one go using the much simpler form 170 PRINT #3,entry$ - SuperBASIC recognises that "entry$" is an array, and no subscript or slice has been specified, so it prints the whole array with a linefeed between each part of the array. The slight advantage of using a FOR loop to do the printing though is that it is a little bit clearer to read.

If we want to read the array back in later, we need to use an OPEN_IN command to open a channel to the file and then use INPUT to read the values back into the array, like this:

```
200 CLS : INPUT "Filename ";filename$
210 DIM entry$(9,20)
220 OPEN_IN #3,filename$
230 FOR a = 0 TO 9 : INPUT #3,entry$(a)
240 CLOSE #3
```

One way of improving our simple little file program is to ensure that we add data to the file which describes the dimensions of the array, to make sure we use exactly the same array

when we load the file at a later date. To do this, we add two extra variables with the two dimensions of the array, and save these at the start of the file to allow this information to be recovered before the data itself is read back.

So our program to create the file now looks like this:

```
100 INPUT "Maximum length of each item ";wide
110 INPUT "How many entries ";high
120 DIM entry$(high-1,wide)
130 CLS
140 FOR a = 0 TO high-1
150   INPUT "Entry ";(a);": ";entry$(a)
160 END FOR a
170 INPUT"Filename ";filename$
180 OPEN_NEW #3,filename$
190 PRINT #3,wide
200 PRINT #3,high
210 FOR a = 0 TO 9 : PRINT #3,entry$(a)
220 CLOSE #3
```

And the program to read data back has to be modified to recover and use this information:

```
300 CLS : INPUT "Filename ";filename$
310 OPEN_IN #3,filename$
320 INPUT #3,wide
330 INPUT #3,high
340 DIM entry$(high-1,wide)
350 FOR a = 0 TO high-1 : INPUT #3,entry$(a)
360 CLOSE #3
```

What we have discussed thus far describes the very simplest type of file, called a **SERIAL** data file, so called because when loading or saving it, we always start from the beginning and work our way through to the end. This is a very simple type of data file to use, and is perfectly adequate for small, simple files.

The other type of data file is called a **RANDOM ACCESS** type. This is rather more complex than the SERIAL type, and as its name implies you can grab any piece of data from somewhere in the middle of the file and just manipulate individual items of data without having to load it all into memory. This allows very large files to be handled, but the programming involved can be much more complex.

The basis of a random access file is that all the entries in the file are of equal length or padded out to be equal length. This in turn lets us just work out how far into a file the data should be, position the file pointer there and we can fetch one item out of the middle of the file or write an item there to update the old value.

To visualise how a random access file would look, we'll use our example array above, and pad it out with spaces (I'll use a dot here to represent a space. Each line would end with a linefeed (shown by <LF> below). Our array "entry$" just contains the words "one", "two" and

so on up to "ten". Our file would look like this:

```
one.................<LF>
two.................<LF>
three...............<LF>
four................<LF>
five................<LF>
six.................<LF>
seven...............<LF>
eight...............<LF>
nine................<LF>
ten.................<LF>
```

We dimension the array to be up to 20 characters wide. When we print it to the file, we make sure that exactly 20 characters are sent to the file and if necessary we add some spaces to pad it all out to the required length. Therefore, each entry becomes exactly 20 characters wide plus the linefeed which PRINT adds, so a total of 21 characters per entry.

```
100 CLS : INPUT"Filename ";filename$
110 DIM entry$(9,20)
120 FOR a = 0 to 9
130   INPUT "Entry number ";(a);": ";entry$(a)
140 END FOR a
150 OPEN_NEW #3,filename$
160 FOR a = 0 TO 9
170   PRINT #3,entry$(a);FILL$(' ',20-LEN(entry$(a)))
180 END FOR a
190 CLOSE #3
```

The important line here is line 170. This ensures that enough spaces are added by a FILL$ command to ensure that each entry in the file is exactly 20 bytes long plus the linefeed after it.

This has now created a file where we can work out where in the file each entry lies. This lets me introduce the concept of the file pointer, which is an operating system pointer which tells us how far into the file we are at the moment, in other words where the next entry will be taken from. It starts at 0, which means the beginning of the file (or how many bytes of data there are from the beginning of the file to here). In this case, we know that each entry is 20 bytes long and there is a linefeed after each one, so the first entry starts at 0, the next one at 0 plus 21, then next one at 0 plus 21 plus 21 and so on. So to skip the first entry we point just after the first 21 bytes. This first entry occupies bytes 0 to 20, so we set the pointer to 21 and INPUT the element starting there. So, to specify this mathematically we need to point to 21 x entry_number (where entry number starts from 0 like an array subscript). To get the third record we would set the file pointer to 42 (or 2*21) and so on.

RECORDS AND FIELDS

At this point we'll digress to discuss some terminology - RECORDS and FIELDS.

A Record is one complete entry in a database. A Field is one part of the complete entry. Suppose we are building a database of names and addresses. A record is one particular name and address. The lines within the name and address are each a different field:

```
Fred Bloggs
123 The Street
Anytown
Any County
England
AB12 3CD
```

If we use a different array for each line of the name and address, each array is a field, for example:

```
DIM name$(9,20)     : REMark name field
DIM address1$(9,20) : REMark address line 1
DIM address2$(9,20) : REMark address line 2
DIM address3$(9,20) : REMark address line 3
DIM address4$(9,20) : REMark address line 4
DIM postcode$(9,20) : REMark postcode field
```

So, this database contains 6 fields in each record, the first being the person's name, the next four contain the address and the final field contains the postal code.

This example uses fields which have the same width (20 characters). Where this is true, it may have been easier to use one array with an extra dimension:

```
DIM name$(9,5,20)
```

In this way, the '9' represents the number of records the file can handle (0 to 9) and the '5' represents that the database will have six fields (0 to 5).

If it helps to remember which term is which, think of an address book. Each page is a 'record'. The number of pages in the book is how many 'records' the book can store. Each part of the page, each line if you like, is a field within that record.

In practice, fields in a file like this are rarely the same size. A postal code is rarely more than 8 to 10 characters, while 20 characters is really a bit short for names and addresses.

Another way of storing the data in an array is to dimension an array wide enough to hold the names and addresses in a single line, rather like columns of text. We need to work out how many characters we need in total. In this case, we dimensioned six arrays, all up to 20 characters wide, so we need 6*20 = 120 characters in total, and we can store the whole lot in an array called entry$:

```
DIM entry$(9,120)
```

For this to work we need to make sure that each record (each array entry) is a fixed size and padded out to 120 characters. The best way of doing this is to pad it out with spaces:

```
FOR a = 0 TO 9 : entry$(a) = FILL$(' ',120)
```

Important: when dimensioning arrays, you should make sure it's an even last dimension, otherwise the QL may round it up for you and cause unpredictable side effects. So DIM entry$(9,10) is OK. DIM entry$(9,11) is not, and you may never be sure if the QL made each string 10 or 12 characters long.

Having all fields in a single array dimension works by virtue of the fact that SuperBASIC lets you assign data to a string slice as long as you are careful to assign the right length and pad the fields out with spaces to make them the right length and make sure that old data is deleted before you assign any updated data.

Since each field happens to be the same length in our database, we need to make sure that when we use a LET statement to put text into the array, we start the assignment at the right place and make sure we put the right length of data into the array. So to put the name field into entry$(0), the first field in the first record, we could input the name and use a LET statement to put it into the right place:

```
INPUT name$
IF LEN(name$) > 20 THEN name$ = name$(1 TO 20) : REMark too
long?
IF LEN(name$) < 20 THEN name$ = name$&FILL$(' ',20-LEN(name$))
LET entry$(0,1 TO 20) = name$
```

It does have the disadvantage that the name is padded with spaces, so you'll always get back the padded 20 character length of string and you don't really know the real width of the name unless you strip off spaces at the right of the name when you read it back later. It can be a positive advantage in some cases, e.g. where data is displayed in columns and rows, like a spreadsheet or text file with text in columns. Ultimately, it is up to you which type of array method you use to store your database, depending on what is most appropriate to what type of database you build, and how the data is to be displayed.

INTERNAL DATA FORMATS

So far, we have used the PRINT and INPUT commands to send data to and recover data from a file. In many cases, this is fine. It writes the data to the file pretty much like text, followed by a linefeed character. It becomes a little restrictive in that numbers are actually stored as text.

Toolkit 2 introduced new commands to SuperBASIC which allow us to write and read data in what is called Internal Format. This is how the QL stores data in its own memory and can be more convenient (but harder to view, read and understand by humans!) to use when it comes to random access files. These new commands and functions (PUT, GET, BPUT, BGET) allow us to write and read data using the same format as the QL uses internally:

Strings are stored using a 2 byte value for the length, followed by the text of the string. This allows strings, in theory (memory permitting), up to about 32 kilobytes long.

Integers are stored as a 2-byte value, storing values from -32768 to +32767

Floating point values are stored as 6 byte values.

Byte values can also be written and read. Byte values can hold values from 0 to 255.

A very real advantage of this is the predictability of length of data. Using PRINT, the value 32000 needs 5 bytes plus a linefeed, whereas 99 requires only two bytes plus the linefeed. If we use internal data types, an integer value like those two always need the same number of bytes (2), so the length stays the same in a file. Likewise, floating point numbers always need 6 bytes - the length doesn't depend on the number of digits. And with text, the length is always the number of characters in the string plus the two bytes used for the length. So a string would look like this:

<2 bytes for length of string>Text follows the length bytes

We use the commands PUT and BPUT to write such data to a file. PUT knows from the name of the variable which type it is to use. Names which end with $ are treated as strings, names ending with % are integer values and names ending with neither of those symbols are handled as floating point numbers. BPUT always sends the data as a byte value even if you use an integer% name or a floating point variable.

The command is followed by a channel number indicating which file data is sent to or read from, then the name of the variable being handled:

```
100 OPEN_NEW #3,flp1_testfile
110 LET a$ = "QL Today" : REMark a string
120 LET year_no% = 2007 : REMark an integer number
130 LET float_num = 1.5 : REMark a floating point number
140 LET byte% = 255     : REMark a byte value
150 :
160 REMark send these variables to the file
170 REMark in internal format
180 PUT #3,a$         : REMark send a string
190 PUT #3,year_no%   : REMark send an integer
200 PUT #3,float_num : REMark send a floating point number
210 BPUT #3,byte%     : REMark send one byte value
220 CLOSE #3
```

This simple example sets up a few variables and opens a new file, then shows how to use PUT and BPUT in lines 180 to 210 to write the values of the variables out to file in QL internal format.

So what we get is as follows:

```
<2 length bytes>"QL Today"<2 byte integer><6 byes float value><byte value>
```

Note that there's no linefeed (newline) characters between them all. Try copying this file to the screen to view it and it will look pretty meaningless, with a little text you can read. This shows a little disadvantage with this method of storing data - it is not easy for humans to read, although it is great for a QL to handle!

Here is how we read the data back into the QL later:

```
100 OPEN_IN #3,flp1_testfile
110 GET #3,a$         : PRINT a$
120 GET #3,year_no%   : PRINT year_no%
130 GET #3,float_num  : PRINT float_num
140 BGET #3,byte%     : PRINT byte%
150 CLOSE #3
```

The prgram fetches the values of the variables from the file and prints them to the screen to prove that they are the same ones we wrote to the file previously.

Although I happened to use the same variable names, you don't have to do so, as long as you use the right type of variable. The first one must be string. The second one must be an integer, the third one must be a floating point number, and the fourth one must be either an integer (best) or a floating point variable, either can accept the one byte value. Sometimes, QL type coercion can convert from one type to another, but you are best advised to use the same type of variable where possible to avoid any mix-ups.

FILE POSITION

We can see where in the file the QL's file pointer is looking at by using the FPOS function from Toolkit 2. This returns a number which tells us how far from the beginning of the file we are. In other words, it gives us a value indicating where the next value will be taken from. Give it a channel number and it will tell you this pointer's value. As the QL can have more than one file channel open at the same time, you need to be careful you are using the correct channel number. Assuming that the file is alread open:

```
PRINT FPOS(#3)
```

or

```
LET file_postn = FPOS(#3)
```

We can use FPOS in our last example to help us understand how the file is stored, by printing its values as we read each item back from the file. This can help us debug a program when we make a mistake and things don't quite work as expected:

```
100 OPEN_IN #3,flp1_testfile
```

```
110 fp = FPOS(#3) : GET #3,a$         : PRINT fp;':';a$
120 fp = FPOS(#3) : GET #3,year_no%   : PRINT fp;':';year_no%
130 fp = FPOS(#3) : GET #3,float_num  : PRINT fp;':';float_num
140 fp = FPOS(#3) : BGET #3,byte%     : PRINT fp;':';byte%
150 CLOSE #3
```

You should get a list like this:

```
0:QL Today
10:2007
12:1.5
18:255
```

Which shows that when a file is first opened, the file pointer is at position 0, zero bytes in from the beginning of the file. Then, we read in the string, which we expect to be 8 characters long plus the two length bytes. So, after we read in the value of a$, the file pointer moves 10 bytes further into the file, pointing to the next item, which is the integer value 2007. We read in two bytes for the integer variable year_no%, which then leaves the file pointer at 12, ready to read in the value for the floating point number float_num, which in turn leaves the file pointer moved 6 bytes further on pointing at the byte value ready to be fetched to the variable byte%.

Hopefully, after that example we can now start to appreciate how convenient to use these internal storage formats are for us to use in files, because the numbers of each type are always stored using exactly the same amount of bytes in a file, and strings are always their own length plus 2 bytes for the length. When it comes to handling individual records in a large file, these "known length" elements will come in very useful.

We know how to check the file pointer position, but we can set it too, using a special option of the GET and BGET commands. If we use a backslash symbol after the channel number, this tells the command that the number or expression which follows is a file pointer position value. So we could change the value of the integer year_no% above and use this to alter the value stored in the file, without altering anything else. This will help explain bthe term RANDOM ACCESS FILE because we are able to point to any item in the file pretty well randomly and read or write that item as long as we are careful to use the correct data type. To open a file to allow us to read and write to it, we use the OPEN command:

```
100 OPEN #3,ram1_testfile
110 year_no% = 2008
120 BGET #30
130 PUT #3,year_no%
140 CLOSE #3
```

A note here about the various types of OPEN commands.

OPEN_NEW opens a new file. If a file of that name already exists, it will ask if you wish to overwrite it or not, as long as you have Toolkit 2 (if not, it just gives an 'Already Exists' error report). Only one channel can be opened to this file at a time to write to it - two separate

programs must not try to change the data at the same time or absolute chaos could result!

OPEN_IN opens a file for input only. It is not possible to change any of the file content - attempts to do so would result in a 'Read Only' or 'Write Protected' error message. But since it is not being written to, several programs can open_in channels to input data from the same file.

OPEN opens a file in such a way that you can write to the file and input from it at the same time. OPEN_NEW creates a new, blank file, whereas OPEN just opens a channel to both read and write with an existing file. That's really the only difference between OPEN_IN and OPEN - both open an existing file, but OPEN_IN doesn't let you alter the file at all.

As long as the file is still open in exclusive mode (either with OPEN_NEW, OPEN_OVER, or just OPEN, any subsequent attempt to open for reading (shared mode - OPEN_IN) will result in an "in use" error. Same would apply if you attempt to open exclusively a file still open in shared mode - this time the exclusive open call would fail with the error. It's a QDOS limitation that the **entire file** is locked in exclusive mode rather than just at the record or field level.

If you have Toolkit 2, you will also have an OPEN_OVER command. This one deletes any existing file and then opens the file like an OPEN_NEW command. Use with care, of course, it is only too easy to delete a file by mistake!

NOTE: It's worth pointing out that you should always ensure you CLOSE any files opened (in any of the exclusive modes) after any data is changed - in order to persist those changes and before any attempt to re-open in shared mode is made.

Think of an example where we have a small call centre handling calls from customers. All need to be able to read information from the database. Only the supervisor can make changes to it. So the supervisor's QL opens a channel to the database using an OPEN command, while the other operators have their QLs open a channel to the file using OPEN_IN. This allows them to search for the customer's details, but to add changes of address for example, they have to put the caller through to their supervisor who is the only person to have permission (write access) to change anything in the database.

I hope that by now we are starting to get a picture of how the two types of files work. As the names imply:

* **SERIAL files** - we start at the beginning and work our way through the file in order.

* **RANDOM ACCESS files** - we can also jump back and forth by moving the file pointer about within the file, meaning we can change a specific entry in the file almost at will, and add more data to the end of the file without first having to read it all in.

SETTING UP A DATABASE

A good deal of forethought is needed when designing a database using random access files, since changing the structure of a database (adding or removing fields or changing data types) may prove difficult.

We would need to assess how many fields are needed, and how large each needs to be. We also need to know which fields are to hold text, which ones integer numbers, which ones floating point numbers and so on, so that we can make a note of how much space each field and each record will take in the database, to enable us to calculate how far we need to jump from part to part.

A simple little subscriptions database will illustrate this. We decide that we need text fields for subscriber's name, four lines of address, one line for postal code, and a final number field for how many issues of the magazine is left until the subscription expires.

```
max% = 100 : REMark maximum number of subscribers allowed
DIM name$(max%-1,20)
DIM addr1$(max%-1,30)
DIM addr2$(max%-1,30)
DIM addr3$(max%-1,30)
DIM addr4$(max%-1,30)
DIM postcode$(max%-1,10)
DIM issues%(max%-1)
```

Looking at this we can work out how much space each record is to take, by assuming that each string needs its maximum nnumber of characters plus two bytes for the string length. Integer numbers need two bytes each:

```
name$ needs 2+20=22 bytes per entry
addr1$ needs 2+30=32 bytes per entry
addr2$ needs 2+30=32 bytes per entry
addr3$ needs 2+30=32 bytes per entry
addr4$ needs 2+30=32 bytes per entry
postcode$ needs 2+10=12 bytes per entry
issues% needs 2 bytes per entry
```

When we use PUT to send these field variables to the file, we will need a total of 22+32+32+32+32+12+2 bytes (164 bytes in total) per record. In other words, every full entry in the database will need 164 bytes. As we set a maximum of 100 entries (the variable max%), a completely full database will need 164*100 bytes, or 16,400 bytes. This is the beauty of using PUT and BPUT commands, everything can be handled as nice neat predictable lengths of data - so we can jump around and fetch any data we like using multiples of the field length. To go to the second entry in the database, we simply skip the first 164 bytes. In other words, we skip 152 bytes times the record number less one. To get to the 10th entry:

```
bytes_per_record * (entry_number-1)
164*(10-1) = 164*9 = 1476
```

So, assuming everyone's membership number is actually their position in the database (the

first is 1), we can write a line or two of SuperBASIC to calculate where their entry in the database should be:

```
INPUT"Membership Number > ";memb_num
postn = 1476 * (memb_num-1)
OPEN_IN #3,flp1_database
BGET #3
```

and we are quickly at the member's records, which we can read in with a few GET commands, into suitable variables.

```
GET #3,n$ : REMark name
GET #3+22 : REMark next field starts here
GET #3,a1$ : REMark address line 1
GET #3+22+32 : REMark next field starts here
GET #3,a2$ : REMark address line 2
GET #3+22+32+32
GET #3,a3$ : REMark address line 3
GET #3+22+32+32+32
GET #3,a4$ : REMark address line 4
GET #3+22+32+32+32+32
GET #3,p$ : REMark postal code
GET #3+22+32+32+32+32+12
GET #3,i% : REMark issues left
```

By now, I hope you can see that accessing any part of the data is as easy as calculating its position in the file and fetching it from there. It is important that even if the fields are shorter than their maximum possible length that we pad out the file with enough spaces or zero codes to make sure the record is always the right length even if this seems a little wasteful.

When adding a new record to the file we point to the end of the existing data, remember that position, send the same number of bytes as the maximum record size, point back to where this new record started, and then PUT the data in the right places as required.

An easy way to set the file pointer to the end of the file is to use FLEN(#channel_number):

```
fl = FLEN(#3)
BGET #3
```

We can then add enough bytes of space by printing the right number of spaces or nulls to the file. In this case, our records were 1476 bytes long:

```
PRINT #fl,FILL$(" ",1476);
```

Then, point to the new position agin to start PUTting the new record there:

```
BGET #3
PUT #3,new_name$
```

```
BGET #3+22,new_addr1$
PUT #3,new_addr1$
```

and so on.

Having a database in a known layout like this makes it fairly easy to search through it as well. Using our subscriber database example above, suppose that the subscriber called us, but didn't know his membership number. We could then search for his postcode -

```
INPUT"Search for which postcode?";pcd$
FOR a = 0 TO number_of_records-1
  REMark work out where each record starts
  fp = a*1476
  REMark now work out how far into the record is the postcode
  BGET #3+22+(4*32)
  GET #3,postcode$
  IF pcd$ == postcode$ THEN
    REMark we have found the subscriber details
    REMark extract them here
    ...
    EXIT a
  END IF
END FOR a
```

COMPARISONS AND CONCLUSIONS

Learning about how random access files work is good background knowledge for using Archive, since it works in a similar, though somewhat more advanced way. Many database programs you can find for the QL such as K-Base and EasyBase also use these types of principles and so it's all good background knowledge, and would also stand you in good stead when it came to writing customised databases using the DataDesign or QBase database programming languages. Although not all database programs use fixed size records for random access files like this (Flashback and DataDesign are notable exceptions which use their own unique format flexible record lengths and so on), the grounding in knowing what fields and records are, and a little experience of writing your own code will really help you understand what other free and commercial database programs do and to understand the possibilities of what all these programs can really do if we can achieve fairly sophisticated things just using a few lines of SuperBASIC.