File Creation and Manipulation

A file is a collection of related data kept on secondary storage. Files containing programs are referred to as *program files* (with a .exe extension appended to them), whereas the files we will be creating now are *data files*; they store data to be processed by our programs.

A major advantage of *data files* is that they are kept on secondary storage, such as jump drives or hard drives. Data kept on secondary storage, unlike that kept in the computer's RAM, is not lost when the system is turned off. Another advantage of using *data files* is that many users can access the same *data* (portability). In addition the file(s) can be created by one program then accessed by other program(s) at a later time. Files make it easy for us to update data.

Files are usually made up of a group of records. The individual data items within these records are called *fields*. A computer file is often compared to a filling cabinet. A particular filing cabinet drawer often contains information on one general topic, such as the records of students. Each student record is kept in a separate folder. The individual pieces of information in each folder are similar to *record fields*.

Secondary Storage

The most commonly used types of secondary storage for micro-computer systems are the hard disks, CDs, jump drives, etc… with hard disks being the most popular storage media.

File Access Methods

An ***access method*** is the way in which the computer transmits data between secondary storage (hard disks, CDs, jump drives, etc…) and primary storage (RAM). Because of the way in which secondary storage devices are structured, two different access methods can be used with them: ***sequential*** and ***random***.

When using ***sequential*** access, the data within the file is accessed in the order in which it is physically stored within the files. This means that if we want to access a particular data item, all the data that precedes the needed item must be accessed first. We can think of sequential access as being similar to a one-way street. If we visit the tenth house on the block and then wish to go back to the second house, we must drive around the block to the first house and then go on the second house. ***Random*** access allows a record to be accessed directly, usually by using its numbered location in the file. A hard disk's read/write heads allows it to move quickly to a specific disk location. *Random* access can be compared to dialing a telephone number. The number allows the system to connect us with a single telephone line out of many thousands of lines.

QBasic allows us to create both types of files. When we create a ***sequential*** file we must access the data within it sequentially. The records in ***random*** access files, however, can be accessed directly (non-sequentially).

Using Sequential Files

File Position Pointers

Before we discuss the different statements used with ***sequential*** files, it is important to understand the concept of the ***file position pointer***. Associated with each file is a file position pointer that "points to" the next record to be processed. The computer adjusts the ***file position pointer*** when a file statement is executed in a program. For instance, when we first

access a file to read its records, the ***file position pointer*** is set to the beginning of the file so that the first record in the file is the one available for processing. After the program reads the first record, the ***file position pointer*** automatically advances the next record. The ***file position pointer*** has no effect on the data contained in the file just like an array index number has no effect the data in an array: it simply indicates the position of the next record to be processed.

Creating and Accessing a Sequential File

To use data files, we must be able to create a new file or access an existing ones. Both operations are performed by the `OPEN` statement. The `OPEN` statement provides QBasic with the following information:

1. ***The name for the file***. The same rules apply to naming data files that apply to naming program files: the name may have from one to eight characters and an optional extension (from one to three characters). We often assign the same name to both the program file and its data file and use their extension to differential them. For example, if `INVEN.BAS` is an inventory program, the file storing the inventory data could be named `INVEN.DAT`.
2. ***The mode, or way, in which the file is to be used***. The mode can be `OUTPUT`, `INPUT`, or `APPEND`.
   - `OUTPUT` (to the file) indicates that data will be written to the file from the program.
   - `INPUT` (from the file) the contents of the file are being read into the program.
   - `APPEND`S allows new records to be added (appended) to an existing file.
3. ***The number of the buffer to be associated with the file***. A *buffer* is a reserved spot in primary storage used as a temporary storage area for data that is being written to or read from a file. When data is being written to a file:
   - It is first copied from the variable to the *buffer*.
   - Next it is copied from the *buffer* into the correct location in the file.

   Each file must be assigned a *buffer* number.

The following statement creates a file named `TICKETS.DAT` on the Q drive, sets its ***mode*** to `OUTPUT` and associates it with ***buffer*** `#1` (note the double quotation marks enclosing the entire file name):

```
OPEN "q:\TICKETS.DAT" FOR OUTPUT AS #1
```

This statement instructs the system to do the following:

1. Create a new file named `TICKETS.DAT` on the Q drive.
   (*Note: If a file with this name already exists, it is destroyed, and a new, empty file is created.*)
2. Prepare the file to receive data (the word `OUTPUT` indicates that this file will have data written to it).
3. Associate the file `TICKETS.DAT` with *buffer* `#1`. Until the file is closed the file buffer `#1` will be used to temporarily store data before it is written to the file on disk. Other statements in the program use this *buffer number* `#1` to identify the file.

The `OPEN` Statement Format for Sequential Files

```
OPEN "file-name" FOR mode AS [#]file-number
```

Note: The available modes are `OUTPUT` (to be written to), `INPUT` (to be read), or `APPEND` (to have new records added to the end).

Closing a File

When a program is through using a file, the CLOSE statement closes the file. Closing the file causes its contents to be stored permanently on disk. If the file is opened for OUTPUT, the CLOSE statement moves any data remaining in the *buffer* (#1) to the file (this is often referred to a as *dumping the buffer*). No input or output can be performed on a closed file. The following statement closed the file that was previously opened:

```
CLOSE #1
```

Notice that the CLOSE statement does not use the name of the file (which is TICKETS.DAT), but rather its *buffer number* (#1). Any number of files can be closed in a single statement:

```
CLOSE #1, #3
```

If we use the CLOSE statement alone, it closes all files that have been opened in a program:

```
CLOSE
```

If we wish to change the ***mode*** of an open file, we must first close that file and then re-open it in another ***mode***.

Note: The file can be re-opened with the same or a different file number.

The CLOSE Statement Formant

```
CLOSE [#file-number1, #file-number2, …]
```

Note: If no file-number is listed, all opened files are closed.

Writing to a Sequential File

When a file needs to have new data written to it, it ***mode*** must be set to OUTPUT. The new file is created, but at this point it is empty. To store data in a new file we perform the following steps:

1. Open the file for OUTPUT.
2. WRITE writes date to the file.
3. CLOSE the file to protect its contents.

The WRITE #*buffer-number* statement places records in the file. It operates much like the PRINT statement, except that it sends values to a specific file (in secondary storage) rather than displaying them on the screen. For example, the following statement sends the values in the variables artist$, numTickets, ticketCost, and percent to the file number #1:

```
WRITE #1, artist$, numTickets, ticketCost, percent
```

QBasic places the values of artist$, numTickets, ticketCost, and percent in the file at the storage location indicated by the ***file position pointer*** and advances the pointer to the next record positon. The file now contains the data for one record.

The `WRITE` Statement for Sequential Files

```
WRITE #file-number, data1, [data2,]…
```

The following program creates a ***sequential*** file `TICKETS.DAT` and allows the user to write data to it. The purpose of this file is to keep track each artist's total ticked sales at concerts. Each record contains the artist's name, the number of tickets sold at a single concert, the cost of each ticket, and the percentage of total sales to be given to the artist.

```
'*****************************************************************
'                          CREATE TICKET SALES FILE
'
'This program creates a sequential file, TICKETS.DAT.
'The user is prompted to enter the data which is then written to the file.
'
'File used:
'  q:\TICKETS.DAT  A Sequential file containing ticket sales information.
'
'Major variables used:
'  artist$        Name of the individual or group.
'  numTickets     Number of tickets sold.
'  ticketCost     Cost of each ticket.
'  percent        Percentage of ticket sales that
'                  the artist receives.
'*****************************************************************

OPEN "q:\TICKETS.DAT" FOR OUTPUT AS #1

CLS

INPUT "Do you want to enter information on an artist (Y/N)"; answer$
DO WHILE UCASE$(LEFT$(answer$, 1)) = "Y"
   INPUT "Enter the artist's name: ", artist$
   INPUT "Enter the total number of tickets sold: ", numTickets
   INPUT "Enter cost per ticket: ", ticketCost
   INPUT "Enter percentage artist is to receive: ", percent
   WRITE #1, artist$, numTickets, ticketCost, percent
   INPUT "Do you want to enter information on another artist (Y/N)";
          answer$
LOOP
CLOSE #1

END
```

Example data: *(note: each line represents a single record in the file).*

```
Pete and the Piranhas, 850, 25.00, 10
Working Stiff, 384, 32.5, 8
Working Stiff, 458, 25.00, 8
Astrofly, 592, 24.00, 8
Astrofly, 634, 30.00, 10
```

Appending Records to a Sequential File

If it is necessary to add records to a file that is open for INPUT, the file must first be closed using a CLOSE statement, then re-opened using an OPEN statement with the ***mode*** set to APPEND. For example:

```
CLOSE #1
OPEN "q:\TICKETS.DAT" FOR APPEND AS #1
```

When a file is first opened for input or output, the ***file position pointer*** is placed at the first record in the file; however, new records can be written only at the end of a ***sequential*** file. The APPEND clause sets the ***file position pointer*** to the end of the specified file and thus permits the user to add new records without losing any of the old ones. The three basic steps for appending data to an existing sequential file are:

1. Open the file for APPEND.
2. WRITE #*buffer-number* writes the new data to the end of the file.
3. CLOSE the file.

The following code snippet adds data records to the TICKETS.DAT file created by the pervious program. The only change is in the OPEN statement. When adding records to an existing file, be careful to open the file for APPEND rather than OUTPUT. The OUTPUT clause destroys the contents of an existing file.

```
CLOSE #1
OPEN "q:\TICKETS.DAT" FOR APPEND AS #1

FOR ctr = 1 to 3
   INPUT "Enter the artist's name: ", artist$
   INPUT "Enter the total number of tickets sold: ", numTickets
   INPUT "Enter cost per ticket: ", ticketCost
   INPUT "Enter percentage artist is to receive: ", percent
   WRITE #1, artist$, numTickets, ticketCost, percent
NEXT ctr
```

After the preceding snippet is executed the last three records are appended to the end the file TICKETS.DAT which will now contains the following values:

```
Pete and the Piranhas, 850, 25.00, 10
Working Stiff, 384, 32.5, 8
Working Stiff, 458, 25.00, 8
Astrofly, 592, 24.00, 8
Astrofly, 634, 30.00, 10
Next Time, 743, 36.50, 10
Next Time, 875, 40.00, 8
Next Time, 657, 36.50, 10
```

Reading from a Sequential File

Writing data to a file is useful only if you are able to use that data at a later time. The transfer of data from a file to the computer's RAM is referred to as reading a file. The three steps involved in reading f from a sequential file are:

1. Open the file for INPUT #*buffer-number*.
2. INPUT #*buffer-number* reads the data from the file.
3. CLOSE the file.

The INPUT #*buffer-number* statement reads data from a file and is similar to the INPUT statement. Whereas the INPUT statement accepts data from the keyboard, the INPUT #*buffer-number* statement takes values from a file and assigns them to the listed variables on a one-to-one basis. The following code snipped reads four data items (the first record) from the TICKETS.DAT file and assigns them to the variable artist$, numTickets, ticketCost, and percent.

```
OPEN "q:\TICKETS.DAT" FOR INPUT AS #2
INPUT #2, artist$, numTickets, ticketCost, percent
```

These statements cause variables artist$, numTickets, ticketCost and percent to be assigned the values Pete and the Piranhas, 850, 25.00, 10, respectively. The INPUT #*buffer-number* statement reads the record to which the file position pointer is currently pointing, and places its contents in the variables listed. After the INPUT #*buffer-number* statement is executed, the file positon pointer advances automatically to the next record. In a ***sequential*** file, an INPUT #*buffer-number* operation begins with the first record in the file (that is, where the OPEN statements has set the ***file position pointer***). Each successive INPUT #*buffer-number* statement retrieves the next values in the file and places them in the listed variables.

The computer places a special character, called an ***end-of-file marker***, after the last data item in a file. An attempt to read past this marker results in an error message and termination of program execution.

The ***end-of-file marker*** acts as a "trailer/sentinel value" that is quite useful when reading the contents of a file. A special function, called the EOF (end-of-file) function, is used to check for ***end-of-file marker***. For example, the following expression determines whether the ***end-of-file marker*** for file #1 has been reached:

```
EOF(1)
```

If there are more records to be read, EOF returns FALSE; when the ***end-of-file marker*** is reached, EOF returns TRUE.

Because the EOF function results in a ***Boolean*** value of TRUE or FALSE, it can be used to control the execution of a loop. The following loop reads and prints all of the data in file #1; TICKETS.DAT:

```
DO WHILE NOT EOF(1)
      INPUT #1, artist$, numTickets, ticketCost, percent
      PRINT TAB(10); artist$; TAB(30); numTickets; TAB(45); ticketCost;
            TAB(60); percent
LOOP
```

In this loop, every record in file #1;TICKETS.DAT is accessed.

Now let's consider a situation where just one record from a file is needed. Suppose we wanted to display the 50th record of file #1; TICKETS.DAT. The rules of ***sequential*** access dictate that all proceeding records must be accessed first, starting at the beginning of the file. We would have to do this by reading (but not printing) the first 49 records of file #1; TICKETS.DAT. Remember that an INPUT #*buffer-number* statement automatically advances the ***file position***

pointer to the next record in the file. After the second record is read, the ***file position pointer*** is set to the third record, and so forth, until the 50th record is reached (*note: **random**-access files offer a solution to this time-consuming process*).

The `INPUT #`*buffer-number* Statement Format for Sequential Files

```
INPUT #file-number variable1[, variable2, …]
```

Using Random-Access Files

Random access files have a distinct advantage over ***sequential*** files. Their records can be accessed directly. By using a ***random*** access file it is no longer necessary to access the first 49 records in a file in order to update the 50th record. (*note: as we will see **random**-access files can also be accessed sequentially*).

Creating a Random-Access File

Random access files consist of records (think `TYPE`). These records are declared using a `TYPE` statement. As an example, we will create a ***random*** access file containing Student records by first creating a `STUDENT_RECORD structure.`

```
TYPE STUDENT_RECORD         'Declare the record TYPE – STUDENT_RECORD
     name AS STRING * 25
     IDNumber AS STRING * 11
     DOB AS STRING * 8
     GPA AS SINGLE
     credits AS INTEGER
END TYPE
```

A DIM statement must be used to create a record:

```
DIM student as STUDENT_RECORD
```

Before a ***random*** access file can be used, it must be opened. The `OPEN` statement for the file named `STUDENT.DAT` containing Student records might look like this:

```
OPEN "q:\STUDENT.DAT" FOR RANDOM AS #3 LEN = LEN(student)
```

Notice that the `LEN` function at the end of the statement line. It is used to determining the size of each record. QBasic must know how much space to allot for each record so that it can properly store each records. After we are done accessing a *random-access* file, it must be closed in the same way as a ***sequential*** file.

The OPEN Statement Format for Random-Access Files

```
OPEN "file-name" FOR RANDOM AS [#]file-number LEN = LEN(record-variable)
```

Storing Records in a Random-Access File

Just as before data is assigned to each record by placing a period between the record name and the field name. For example, the following statement assigns `2.8` to the `GPA` field of the `student` record:

```
student.GPA = 2.8
```

After data has been assigned to each field, the PUT statement is used to write that data to the specified record location. For example, the following statement writes the current contents of student to record number 4 in file #3:

```
PUT #3, 4, student
```

Notice that the file number #3 must be the same as the one in the OPEN statement; this is how QBasic knows the record should be written to file STUDENT.DAT.

We also use the PUT statement to overwrite existing records. So, if record fields need to be updated, we simply place the new values in the fields then write them to the old record location using the PUT statement.

The PUT Statement Format for Random-Access Files

```
PUT [#]file-number[, record-number] [, record-variable]
```

Reading Records from a Random-Access File

The GET statement is similar to a ***sequential*** file's OPEN statement with its ***mode*** set to INPUT. GET performs the reverse process of the PUT statement by allowing us to assign the contents of a file record to a record variable. File records are access directly by using their record number.

The GET Statement Format

```
GET[#]file-number[, record-number] [, record-variable]
```

For example, the following statement access record number 6 of file #3 and assigns the record's fields to the object student:

```
GET #3, 6, student
```

So the values stored in record number 6 are placed into the fields of the student record.

In the following program we will perform two tasks:

1. Allow the user to add a new record to a file.
2. Allow the user to display the contents of an existing record.

In the following program we will use the PUT statement to write each new record to the file and the GET statement to access the specified record.

```
DECLARE SUB AddRecord (student AS ANY)
DECLARE SUB DisplayRecord (student AS ANY)

TYPE STUDENTRECORD
   sName AS STRING * 25
   IDNumber AS STRING * 11
   DOB AS STRING * 8
   GPA AS SINGLE
```

```
   credits AS INTEGER
END TYPE

DIM student AS STUDENTRECORD
DIM choice AS INTEGER


OPEN "q:\STUDENT.DAT" FOR RANDOM AS #3 LEN = LEN(student)

DO UNTIL (choice = 3)
   CLS
   LOCATE 2, 1
   PRINT "1. Add a new record"
   PRINT "2. Display an existing record"
   PRINT "3. Stop"
   PRINT
   LOCATE 1, 1
   INPUT "Enter number of operation you wish to perform: ", choice

   SELECT CASE choice
      CASE 1
         CALL AddRecord(student)
      CASE 2
         CALL DisplayRecord(student)
      CASE 3
         CLS
         PRINT "Goodbye"
      CASE ELSE
         PRINT "Invalid Choice."
   END SELECT
LOOP

CLOSE #3

END

'*************************************************************************
SUB AddRecord (student AS STUDENTRECORD)
   DIM recNum AS INTEGER

   CLS

   INPUT "Enter student's record number: ", recNum
   INPUT "Enter student's name: ", student.sName
   INPUT "Enter student's identification number: ", student.IDNumber
   INPUT "Enter student's date of birth: ", student.DOB
   INPUT "Enter studernt's garde point average: ", student.GPA
   INPUT "Enrer number of credits: ", student.credits

   PUT #3, recNum, student

END SUB
```

```
'*************************************************************************
SUB DisplayRecord (student AS STUDENTRECORD)
   DIM recNum AS INTEGER
   DIM continue AS STRING

   CLS

   INPUT "Enter the student's number: ", recNum
   GET #3, recNum, student

   PRINT
   PRINT "Name"; TAB(28); "ID Number"; TAB(41); "Birth Date"; TAB(53); "GPA";
         TAB(59); "Number of Credits"
   PRINT student.sName; TAB(27); student.IDNumber; TAB(42); student.DOB;
         TAB(52); student.GPA; TAB(65); student.credits
   PRINT
   INPUT "Press <Enter> to return to main menu.", continue
END SUB
```

If we want to read the entire contents of a ***random*** access file, we can use a `FOR…NEXT` loop. We will start with the first record and iterate through the file using the following method:

- Use the `LOF` (length-of-file) function to determine the number of bytes in the entire file.
- Use the `LEN` function on the object `student` to determine the number of bytes per record.
- Divide the size of the file (`LOF`) by the number of bytes per record (`LEN`), to determine the number of records in the file.
- Now a `FOR…NEXT` loop can be used to read the entire file.

When we include a `GET` statement inside the `FOR…NEXT` loop we can read each record in turn.

For example, the following statement will cause a loop to be executed one time for each record in a file containing `student` records:

```
FOR ctr = 1 to LOF(3) / LEN(student)
     GET #3, ctr, student
NEXT ctr
```

```
'*************************************************************************
'                              STUDENT FILE #2
'
'This program creates a random file of student records. Each record contains
'the following fields:
'  Name
'  Identification Number
'  Date of Birth
'  Grade point average
'  Number of credits student has earned
'The user can add a new record or display an existing record
'
'A SUB ROUTINE named PrintRecords has been added.It uses a FOR...NEXT loop and
'a GET statement to print all of the records in the file.
'
'File used:
'  q:\STUDENT.DAT
'
'Major variables:
'  student     Student Record
'  recNum      Number of the current record
'*************************************************************************

DECLARE SUB AddRecord (student AS ANY)
DECLARE SUB DisplayRecord (student AS ANY)
DECLARE SUB PrintRecords (student AS ANY)

TYPE STUDENTRECORD
   sName AS STRING * 25
   IDNumber AS STRING * 11
   DOB AS STRING * 8
   GPA AS SINGLE
   credits AS INTEGER
END TYPE

DIM student AS STUDENTRECORD
DIM choice AS INTEGER


OPEN "q:\STUDENT.DAT" FOR RANDOM AS #3 LEN = LEN(student)
DO UNTIL (choice = 3)
   CLS
   LOCATE 2, 1
   PRINT "1. Add a new record"
   PRINT "2. Display an existing record"
   PRINT "3. Print file"
   PRINT "4. Stop"
   PRINT
   LOCATE 1, 1
   INPUT "Enter number of operation you wish to perform: ", choice

   SELECT CASE choice
      CASE 1
         CALL AddRecord(student)
```

```
      CASE 2
         CALL DisplayRecord(student)
      CASE 3
         CALL PrintRecords(student)
      CASE 4
         CLS
         PRINT "Goodbye"
      CASE ELSE
         PRINT "Invalid choice."
   END SELECT
LOOP

CLOSE #3

END

SUB AddRecord (student AS STUDENTRECORD) STATIC
   DIM recNum AS INTEGER
   recNum = 1

   CLS

'   INPUT "Enter student's record number: ", recNum
   INPUT "Enter student's name: ", student.sName
   INPUT "Enter student's identification number: ", student.IDNumber
   INPUT "Enter student's date of birth (MM/DD/YY): ", student.DOB
   INPUT "Enter studernt's garde point average: ", student.GPA
   INPUT "Enrer number of credits: ", student.credits

   PUT #3, recNum, student
   recNum = recNum + 1

END SUB

SUB DisplayRecord (student AS STUDENTRECORD)
   DIM recNum AS INTEGER
   DIM continue AS STRING

   CLS

   INPUT "Enter the student's number: ", recNum
   GET #3, recNum, student

   PRINT
   PRINT "Name"; TAB(28); "ID Number"; TAB(41); "Birth Date"; TAB(53); "GPA";
TAB(59); "Number of Credits"
   PRINT student.sName; TAB(27); student.IDNumber; TAB(42); student.DOB;
TAB(52); student.GPA; TAB(65); student.credits
   PRINT
   INPUT "Press <Enter> to return to main menu.", continue
END SUB

SUB PrintRecords (student AS STUDENTRECORD)
   DIM ctr AS INTEGER
```

```
    CLS
    PRINT TAB(2); "Student Name"; TAB(20); "I.D."; TAB(29); "DOB"; TAB(38);
    PRINT "GPA"; TAB(44); "Credits"
    PRINT STRING$(50, "*")

    FOR ctr = 1 TO LOF(3) / LEN(student)
       GET #3, ctr, student
       PRINT RTRIM$(student.sName); TAB(20); RTRIM$(student.IDNumber); TAB(27);
       PRINT student.DOB; TAB(37); student.GPA; TAB(45); student.credits
    NEXT ctr

END SUB
```

Comparison of Random-Access and Sequential Files

The following points compare sequential files and random-access files:

- Data items in sequential files are written to the disk one after the other, starting at the beginning, whereas records in random-access files may be written in any order desired.
- Data in sequential files is read from the disk in the same order in which it was written, whereas random-access file records may be read in any order desired.
- Records in sequential files can be different lengths, whereas records in random-access files must be all the same length.
- Both sequential and random-access files must be opened before they can be accessed and closed when processing is completed.