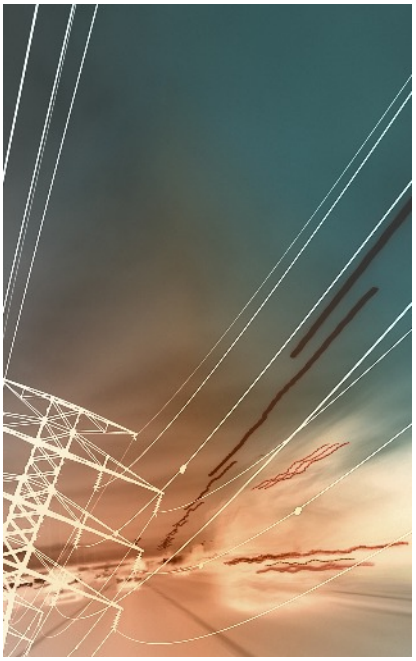
Input / Output - Part 2



In Part 2, we take a look at reading and writing data to disk files.

It's unlikely that anyone is going to be writing major commercial programs, or processing large data files in a simple language like Creative Basic. So our data handling will usually be fairly simple.

Files and Folders

A program accesses a file by its name, in the directory (folder) in which it has been stored. (eg. **D:\folder\Data.txt**)

So, you will need to set up a directory (folder) on your disk in which to store the file. Often this will be in the same directory from which the program is launched - sometimes, it will be in a separate directory.

The first step is to decide what the name of your file is going to be, and in which directory it will be stored. Files can have any name you like, but it's best to choose a meaningful name and file extension.

As an example, let us suppose a file is to be named '**Scottish.dat**', and that it is to be stored in the same directory from which the program is launched.

When your program is executed, it will have no idea whether the file already exists.

It will need to check whether the file exists, and if not, create an initially empty version of it. The actual data will be added as the program proceeds.

There are two types of storage format - an **ASCII** (text) file, or a **Binary** file.

ASCII text files can be read using any text editor such as Notepad. Binary files can not.

A binary file cannot be read by a casual user, or even by a programmer if the file's data structure is unknown.

There are two ways in which data can been written to a file - **Sequentially**, or in **Random Access** format.

A **Sequential** file is where data records are written to the file one after another. Of course, this means that the data has to be read back again in the same order. There is no way to get to the last record without reading all those which precede it.

A **Random Access** file solves this problem, because any record can be read independently of the other records.

Writing and Reading Sequential data files

So let us jump straight in and create a data file on the hard drive.

First step, is to check your computer's date and time settings are correct. When you create a file, it's attributes will contain this information.

Set up a suitable directory (folder) to hold the program and the data file.

Now we need some data to store.

For this example, I'll use some statistics from the Scottish football league, which look like this:

Team	Played	Won	Drawn	Lost	Goals For	Against	Points
Celtic	21	13	4	4	40	15	43
Inverness	21	8	10	3	44	35	34
Motherwell	22	9	7	6	37	30	34
Hibernian	22	9	5	8	31	29	32
Aberdeen	22	8	7	7	30	28	31

Notice we have a mixture of string and integer data in the table - the team names, and the associated statistics.

We are cheating a little bit by setting up the data directly in the program. Normally, the user would enter the data, and then save it to the file. It just simplifies things for the examples.

Creative Basic uses the following command to open a new file ..

OPENFILE (FileVariable, FilePath, Operation)

FileVariable, is defined as a variable of type **File** using the usual **DEF** statement.
For our example:

DEF DataFile : file (where **DataFile** is the chosen name for the file variable).

FilePath, is the fully qualified location of the file. Since we are going to use the same folder for the file and the program, we use **GetStartPath + "scottish.dat"** as the path.

The **Operation** flag **"w"** specifies that the file is to be **Written**. (An "**r**" would specify a **Read** operation, and an "**a**" would be an **Append** operation to an existing file).

So here is a test program to create a new **ASCII (Text)** data file, and then to read the data back again from the stored file. Copy the program to your chosen folder before loading and running it. The program will store the data to your hard drive in this folder. It can be deleted later if you wish.

Sequential ASCII (Text) Files

```
' Disk File Test Program 1 ..
' Creates a Sequential ASCII (Text) File in the same Folder as the program.
' This method of handling the data will write 1 byte for each name character
' and 4 bytes for each integer - with separator characters - 214 bytes in all.

openconsole
cls

def DataFile:file
def name[5]:string
def stats[7,5],team,ival:int

autodefine "off"

' a few Scottish team names are held in array name[] as strings ..
name[0] = "Celtic","Inverness","Motherwell","Hibernian","Aberdeen"
' (remember arrays are zero based)

' the statistics for each team are in array stats[ival,team]. The values are -
'   0   Matches Played
'   1   Won
'   2   Drawn
'   3   Lost
'   4   Goals For
'   5   Goals Against
'   6   Points

' Note that a list of values will automatically load from a given starting element ..
' so all the loading starts at element 0 for each of the 5 teams (0 - 4)
stats[0,0] = 21,13,4,4,40,15,43             :' Celtic
stats[0,1] = 21,8,10,3,44,35,34             :' Inverness
stats[0,2] = 22,9,7,6,37,30,34              :' Motherwell
stats[0,3] = 22,9,5,8,31,29,32              :' Hibernian
stats[0,4] = 22,8,7,7,30,28,31              :' Aberdeen

' open a file for writing (this will destroy any existing contents of the file)
openfile(DataFile,getstartpath + "scottish.dat","w")
for team = 0 to 4
   write DataFile,name[team]
   print:print name[team] + chr$(9),
   for ival = 0 to 6
     write DataFile,stats[ival,team]
      print stats[ival,team],
   next ival
next team
closefile DataFile                          :' always close a file after using it

print:print:print "Data File Written":print
print "Reading Data back again"

' read the data back again, in the same order it was written ..
openfile(DataFile,getstartpath + "scottish.dat","r")
   for team = 0 to 4
      read DataFile,name[team]
      print:print name[team] + chr$(9),
      for ival = 0 to 6
        read DataFile,stats[ival,team]
         print stats[ival,team],
      next ival
   next team
closefile DataFile

print:print:print "Read back completed":print
print "Check the file 'scottish.dat' created in the starting directory"
print "It should be readable in any text editor"
print:print "Straightforward storage, but some waste space."

do:until inkey$<>""
closeconsole
end
```

Hopefully, the program worked, and you now have the file '**scottish.dat**' in your chosen directory. If you examine the file in 'My Computer', you will see it is 214 bytes long.

You could have determined this yourself, while the file was still open, by using the **LEN**( ) function:

```
length = LEN( DataFile )
```

The **LEN( )** function confirms the file to be 214 bytes long.

This test program was written as a console application, but a Windows program would have used the same commands.

Of course there are other ways of representing the football data other than using separate arrays. Use whichever method you are comfortable with.

Writing and Reading Binary data files

Now we'll take a look at the same example, but using **Binary** data files.

The Scottish football data in our example remains exactly the same, but we make one small but important change when we define the **'DataFile'** variable. We use the **bfile** type.

DEF DataFile : bfile (where **DataFile** is the chosen name for the file variable).

It seems to be a small change, but it has significant effects.

From the user's point of view, the data file created on the hard drive will have the same name, but it will look like gibberish if viewed by any other program.

From the programmer's point of view, Binary files enable entire arrays to be written and read using only their names. For example:

```
write DataFile, name, stats
```

However, Binary files are not particularly efficient for storing strings of data. Strings are written using the full 255 characters, even if all the characters are not used. Integers take up 4 bytes each.

This results in a total file size of 1415 bytes - much larger than the previous example.

Binary File

```
' Disk File Test Program 2 ..

' Creates a Sequential Binary File in the same Folder as this program.
' This method of handling the data will write 255 bytes for each string
' and 4 bytes for each integer - 1415 bytes in all.

openconsole
cls

def DataFile:bfile                          :' Declare the file as Binary format
def name[5],text:string
def stats[7,5],team,ival:int

autodefine "off"

' a few Scottish team names are held in array name[] as strings ..
name[0] = "Celtic","Inverness","Motherwell","Hibernian","Aberdeen"
' (remember arrays are zero based)

' Note that a list of values will automatically load from a given starting element ..
' so all the loading starts at element 0 for each of the 5 teams (0 - 4)
stats[0,0] = 21,13,4,4,40,15,43             :' Celtic
stats[0,1] = 21,8,10,3,44,35,34             :' Inverness
stats[0,2] = 22,9,7,6,37,30,34              :' Motherwell
stats[0,3] = 22,9,5,8,31,29,32              :' Hibernian
stats[0,4] = 22,8,7,7,30,28,31              :' Aberdeen

printout                                    :' display the data

' open a file for writing (this will destroy any existing contents of the file)
openfile(DataFile,getstartpath + "scottish.dat","w")
write DataFile,name,stats
' notice that arrays can be written using just their names
closefile DataFile                          :' always close a file after using it

print:print
print "Binary Data File Written"
print:print "Reading Data back again"
print

' now read the data back in again (re-open the file for Reading)..
openfile(DataFile,getstartpath + "scottish.dat","r")
read DataFile,name,stats
' notice that arrays are read back using just their names.
closefile DataFile

print "Read back completed"

printout                                    :' display the recovered data

print:print:print "Check the file 'scottish.dat' created in the starting directory"
print "As a binary file, it is not now readable in a text editor"
print
print "Not very efficient storage - even short strings use 255 characters."

do:until inkey$<>""
closeconsole
end

sub printout
' print the data on screen ..
for team = 0 to 4
   print chr$(10) + name[team] + chr$(9),
   for ival = 0 to 6
      print stats[ival,team],
   next ival
next team
return
```

User Defined Data Type

Now we'll take a look at using a User Defined data type (UDT) with the same example.

```
' Disk File Test Program 3 ..

' Creates a Sequential ASCII (Text) File in the same Folder as the program.
' This version uses a User Defined Type structure to hold the team data.
' Storage is very efficient, requiring a total of only 55 bytes.

openconsole
cls

def DataFile:file                                   :' Declare the file as ASCII format
def name[5]:string
def stats[7,5],i:int

type team                                           :' declare a data structure for each team
   def n:string                                     :' Name
   def m:int                                        :' Matches Played
   def w:int                                        :' Won
   def d:int                                        :' Drawn
   def l:int                                        :' Lost
   def f:int                                        :' Goals For
   def a:int                                        :' Goals Against
   def p:int                                        :' Points
endtype

def t[5]:team                                       :' a data structure array for the 5 teams

autodefine "off"

' 5 Scottish team names are loaded in array name[] as strings 0 to 4 ..
name[0] = "Celtic","Inverness","Motherwell","Hibernian","Aberdeen"
' (remember arrays are zero based)

' Note that a list of values will automatically load from a given starting element ..
' so all the loading starts at element 0 for each of the 5 teams (0 - 4)
stats[0,0] = 21,13,4,4,40,15,43                     :' Celtic
stats[0,1] = 21,8,10,3,44,35,34                     :' Inverness
stats[0,2] = 22,9,7,6,37,30,34                      :' Motherwell
stats[0,3] = 22,9,5,8,31,29,32                      :' Hibernian
stats[0,4] = 22,8,7,7,30,28,31                      :' Aberdeen

' load the data into the Team structure ..
for i = 0 to 4
   t[i].n = name[i]
   t[i].m = stats[0,i]
   t[i].w = stats[1,i]
   t[i].d = stats[2,i]
   t[i].l = stats[3,i]
   t[i].f = stats[4,i]
   t[i].a = stats[5,i]
   t[i].p = stats[6,i]
next i

printout                                            :' display the data

' open a file for writing (this will destroy any existing contents of the file)
openfile(DataFile,getstartpath + "scottish.dat","w")

for i = 0 to 4
   write DataFile, t[i]                             :' Write each team structure
next i
closefile DataFile                                  :' always close a file after using it
```

(Continued)

... continuation

```
print:print "Data File Written"
Print:print "Reading Data back again"
print

' now read the data back in again (re-open the file for Reading)..

openfile(DataFile,getstartpath + "scottish.dat","r")

for i = 0 to 4
   read DataFile, t[i]                          :' Read each team structure
next i
closefile DataFile

print "Read back completed"
print

printout                                        :' display the read-back data

print:print
print "Check the file 'scottish.dat' created in the starting directory."
print "As a file containing structures, it is not readable in a text editor"
print "Very efficiently stored using only 55 bytes."

do:until inkey$<>""
closeconsole
end

sub printout
' print the data on screen
   for i = 0 to 4
      print t[i].n, chr$(9), t[i].m, t[i].w, t[i].d, t[i].l, t[i].f, t[i].a, t[i].p
   next i
return
```

If you run this version of the test program, you will see that the file created is only 55 bytes long. That's pretty impressive and I've no idea how the data is packed so tightly.

Things to notice, are that using a User Defined Type is mostly self documenting - you know what each item of data is. (Sorry for the short item names - I hate typing lengthy ones).

```
type team                                   :' declare a data structure for each team
   def n:string                             :' Name
   def m:int                                :' Matches Played
   def w:int                                :' Won
   def d:int                                :' Drawn
   def l:int                                :' Lost
   def f:int                                :' Goals For
   def a:int                                :' Goals Against
   def p:int                                :' Points
endtype
```

Also, the entire data structure can be written in one go, without using loops for array indices. In our example:

```
write DataFile, t[i]                        :' Write each team structure
```

The only downside, is having to use the object-like dot format such as **Team.Name** when referring to the various data elements.

Still, that's a small price to pay for the clarity of using UDT's.

Our previous examples have covered three types of Sequential data files - Text files, Binary files, and User Defined Type (UDT) text files.

Text files are fine for simple data applications.

You write all your data to a file from the working arrays in your program.

To use the data in a subsequent run of the program, you read the data back into the working arrays in the same order as it was written.

Binary files effectively encrypt the data in the file, since it can not be read without knowing the data structure which created it.

The size of the file may get quite large if there are strings involved. All 255 bytes of each string are written to the file. (This can be avoided to some extent by using **Istrings** of defined length - but working with arrays of IStrings can give you a headache, since an Istring is itself an array ..)

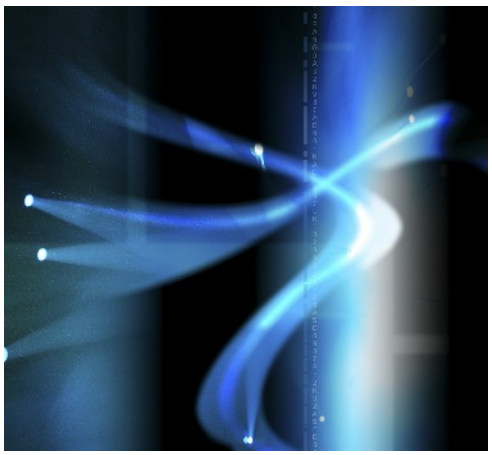
The neatest form of Text File, which also results in the smallest file size, is the User Defined Type approach.

The problem with Sequential files, is that they are 'Write it all Out', 'Read it all In' methods.

They are not so good for applications where the user needs to add, change, or delete a few records of the file at any one time.

For that type of application, we need **Random Access files**.

Writing and Reading Random Access data files



Random Access files contain a number of **Records,** each containing the same data items.

Unlike Sequential data files, you can access any of the records randomly, which is why they are called "random access" files.

Once you read a random file record, you can modify the data, and then re-write the record directly without needing to close and re-open the file.

Random Access files are always in **Binary** format.

Creative Basic uses the same command to open a new Random Access file, as is used for a Sequential file, but the file type is always **Binary** ...

OPENFILE (FileVariable, FilePath, Operation)

FileVariable, is defined as a variable of type **BFile** using the usual **DEF** statement.
For our example:

DEF DataFile : BFile (where **DataFile** is the chosen name for the file variable).

FilePath, is the fully qualified location of the file. Since we are going to use the same folder for the file and the program, we use **GetStartPath + "scottish.dat"** as the path.

The **Operation** flag **"w"** specifies that the file is to be **Written**. (An "**r**" would specify a **Read** operation, and an "**a**" would be an **Append** operation to an existing file).

When you're finished reading or writing to the file, you should close it to make sure that system resources are returned to Windows.

CLOSEFILE DataFile

So far, there seems to be no difference from a Sequential file. The difference comes from the way data is written and read from a Random Access file.

Get and Put

One of the limitations of Random Access files, is that they work with fixed length **Records**, each containing the same number of data items. This is the price you pay for random access to the data.

Each data item must be designed with a fixed size, and once you specify it, you cannot make a change. As a result, the size in bytes of each record is constant. That is how the position on the disk is easily found and retrieved.

In other words, a Random Access file is well suited to using User Defined Data types.

You use the **PUT** command to write a record to the file.

PUT FileVariable, record, variable(s)

FileVariable, is the file variable defined in the **OPENFILE** statement.

Record, is the record number in the file, **starting from 1**.

Record numbers do not have to be contiguous - you can write record 14 when only records 1 through 4 have previously been written.

Alternatively, you can have a **PUT** statement in a loop, which will write a sequence of records.

Variable, is usually a variable of the **User Defined Data (UDT) type**, which specifies the fixed Record length.

When working with an existing file, all data items in each record , need to be appropriately filled before writing the record to the file. Otherwise there is a risk of overwriting or deleting previously stored values.

The GET command is used to retrieve records.

GET FileVariable, record, variable(s)

You use the same list of variables in the GET statement as are used in the PUT statement that created the record.

Random Access File

To investigate using a Random Access file, it will be convenient to use the previous User Defined Type example program.

The data is exactly the same, and we have already seen how to load the data into a User Defined type structure. So most of the work is already done.

The only detail that needs to be considered, is that the User Defined array **t[ ]** is zero based, as are all arrays by default.

Random Access file record numbers **start at 1**. For example, we need the record for Celtic, held in array **t[0]**, to be output to **file record 1**.

This is easily achieved, by using the existing **FOR** loop **variable 'i'** , to give the Random Access record number **'i+1'.**

The same translation of the index is used to Read the data back again.

Here is the modified program ..

```
' Disk File Test Program 4 ..

' Creates a Random Access File in the same Folder as the program.

openconsole
cls

def DataFile:bfile                          :' Declare the file as Binary format
                                            :' (Needed for Random Access files)
def name[5]:string
def stats[7,5],i:int

type team                                       :' declare a data record for each team
   def n:string                                 :' Name
   def m:int                                    :' Matches Played
   def w:int                                    :' Won
   def d:int                                    :' Drawn
   def l:int                                    :' Lost
   def f:int                                    :' Goals For
   def a:int                                    :' Goals Against
   def p:int                                    :' Points
endtype

def t[6]:team                                   :' a data structure array for the 5
                                                :' teams (plus one extra for testing)

autodefine "off"

' 5 Scottish team names are loaded in array name[] as strings 0 to 4 ..
' (remember arrays are zero based)
name[0] = "Celtic","Inverness","Motherwell","Hibernian","Aberdeen"

' Note that a list of values will automatically load from a given starting element ..
' so all the loading starts at element 0 for each of the 5 teams (0 - 4)
stats[0,0] = 21,13,4,4,40,15,43             :' Celtic
stats[0,1] = 21,8,10,3,44,35,34             :' Inverness
stats[0,2] = 22,9,7,6,37,30,34              :' Motherwell
stats[0,3] = 22,9,5,8,31,29,32              :' Hibernian
stats[0,4] = 22,8,7,7,30,28,31              :' Aberdeen

' load the data into the Team structure (zero based in array t[ ])..
for i = 0 to 4
   t[i].n = name[i]
   t[i].m = stats[0,i]
   t[i].w = stats[1,i]
   t[i].d = stats[2,i]
   t[i].l = stats[3,i]
   t[i].f = stats[4,i]
   t[i].a = stats[5,i]
   t[i].p = stats[6,i]
next i

printout                                    :' display the data

' open a file for writing (this will destroy any existing contents of the file)
openfile(DataFile,getstartpath + "scottish.dat","w")

' Note: Since we are writing a Random Access file,
' the data will not be readable in a text editor.

for i = 0 to 4
   put DataFile,i+1,t[i]                    :' Output each record (numbers start from 1)
next i
closefile DataFile                          :' always close a file after using it
```

(Continued)

... continuation

```
print:print "Data File Written"
print
print "Reading Data back again"
print

' now read the data back in again (re-open the file for Reading)..
openfile(DataFile,getstartpath + "scottish.dat","r")
for i = 0 to 4
   get DataFile,i+1,t[i]                        :' Read each record starting at record 1)
next i

print "Read back completed":print

printout                                        :' display the read-back data

print:print "Check the file 'scottish.dat' created in the starting directory."
print "As a Random Access file, it is not readable in a text editor"
print "File size is 1415 bytes - the same as for a Sequential Binary file."

print:print "Accessing Record 2 .."
get DataFile,2,t[5]
print t[5].n, chr$(9), t[5].m, t[5].w, t[5].d, t[5].l, t[5].f, t[5].a, t[5].p

closefile DataFile

do:until inkey$<>""
closeconsole
end

sub printout
' print the data on screen
   for i = 0 to 4
      print t[i].n, chr$(9), t[i].m, t[i].w, t[i].d, t[i].l, t[i].f, t[i].a, t[i].p
   next i
return
```

The Random access file proves to be nothing more than a Binary format, User Defined type file.

It does however provide the ability to add, change and delete records randomly, which can be useful for some applications.

Putting it all together

Once you've decided on the name, type and desired location of your file, the program needs to orientate itself each time you run it.

Does the file already exist in the desired folder - or does a new file need to be created ?

Here is a typical program which handles the situation - creating a new file if necessary, or using the existing file if one is found.

The data file itself will be very small, comprising just one line of text. That will be sufficient to demonstrate the method used.

A typical program

This program checks to see if the file '**OneLine.txt**' exists in the folder from which the program is launched.

If it is found, the program will use it - otherwise a new file will be created.

```
' File Application Program ..
' Checks for an existing File "OneLine.txt" in the same Folder as the program.
' If it doesn't exist, it creates a new one.

openconsole
cls

def File1:file
def text,workfile,filename:string
def exists,dir:int

autodefine "off"

' check for a Previous data file ...
' if it doesn't exist, create it ...
workfile = "OneLine.txt"
exists = 0                        : ' assume the file does not exist

dir = FINDOPEN(GETSTARTPATH + "*.txt")
if (dir > 0)
  do
     filename = findnext(dir)
     if (filename = workfile) then exists = 1
  until filename = ""
  findclose dir
endif

if (exists > 0)
' **** EXISTING FILE ****
   if(OPENFILE(File1,GETSTARTPATH + "OneLine.txt","A") = 0)
' read the previously stored data ...
       read File1, text
       print:print "File OneLine.txt already exists - it reads:"
       print:print text
   else
       print "Error - The file could not be read." + chr$(10)
   endif

else
' *** NEW FILE ****
' we need to create a new file ...
   if (OPENFILE(File1,GETSTARTPATH + "OneLine.txt","W") = 0)
       text = "Creative Basic - A good choice."
       Write File1, text
       print "A new file 'OneLine.txt' has been created."
       print:print "Re-start the program to read the file."
   else
       print "OneLine.txt file could not be created." + chr$(10)
   endif

endif

closefile File1

do:until inkey$<>""
closeconsole
end
```

The test program uses a few Creative Basic functions useful for file handling.

The first is the **FindOpen** function, used to check if a directory exists containing files of a given type.

The syntax is:

handle = FINDOPEN(directory)

'handle' holds an integer reference to the directory, or zero if the directory could not be opened for reading the specified file type.

'directory' is the fully specified directory we wish to examine, plus any wildcard symbols for file matching.

In our example, we assumed the file would be in the same directory from which the program was launched. So the full specification for 'directory' uses the **GETSTARTPATH** function, and the file wildcard description ***.txt**

This should locate any **.txt** files in the directory from which we started the program. If there are no files in the directory with the **.txt** extension, the 'handle' will be zero.

Once a directory is successfully opened, the names of all the .txt files can be tested using the **FINDNEXT** function.

The syntax for this function is:

filename = FINDNEXT(handle)

'handle' is the integer directory value returned by **FINDOPEN**.

'filename' is the name of the file we are interested in.

If the file we are looking for does not exist, filename returns an empty string "".

These functions can therefore be used to test whether a file exists or not, and then branch to an appropriate code block.

When you are finished reading a directory, you must close it using the **FINDCLOSE** function. If the handle is not closed, memory loss will occur.

The syntax for this function is:

FINDCLOSE handle

See the Creative User Guide for other file and directory commands which are available.