# File Handling: Part 1

*by Brian Long*

**It's a funny thing, this Delphi. Before it came along, people who bought a Pascal compiler would probably learn how to talk to normal files using standard file management routines. If they wanted to talk to established database formats they would need to obtain an appropriate add-on library, such as the Paradox Engine or the Borland Database Engine, and negotiate the many APIs it required in particular orders and the appropriate parameter sets. Delphi has made database access of practically any description a breeze. So much has it turned the tables that many Delphi users might not know where to look for file management routines which are not centred around a database table.**

The purpose of this series of articles is to explore what Delphi has to offer on the subject of file manipulation. The principal benefactors will be those readers new to file handling, but I hope to include enough bits and bobs so that even the experienced will pick up one or two things along the way. Subjects to be covered include basic file access using file variables (text, typed and untyped), file handles, one or two efficiency tips, file sharing, record locking (and no, I'm not sneakily introducing a database section!), text file device drivers and streaming.

So what do we need to know how to do, with respect to files? Things we may wish for include creating files, writing data to them, both sequentially and in a random access fashion, reading from files (again sequentially and random access), renaming, copying, deleting, making directories, finding files and probably more besides. Fortunately, Delphi has routines available to do all these and more.

## File Variables

First things first, though: how do we represent a file in a program? Pascal offers two ways: the historic Pascal file variable and the more recent file handle, corresponding to the options C programmers have. We'll deal with file variables first and come back to file handles later.

There are three forms of file variable, and the type we use depends on what form of file we wish to address. If we are interested in a text file, we use the `Text` type, defined in the System unit (the unit that is implicitly used by every unit and main program we write, ie the unit which implements the core run-time library). Although this is a valid type to use in Delphi, it is also the name of a property of several components and so Delphi introduces another term, `TextFile`, to use in its place. In the previous Borland Pascal we can use `Text`, in Delphi we should use `TextFile` or at least fully qualify `Text` and use `System.Text` to tell the compiler exactly which unit it should look in to find the routine we want.

If we are looking at a non-text file we need to identify if there is a regular record structure to the file. If so, we can use a typed file variable, otherwise an untyped file variable. Some example variable definitions are shown in Listing 1, where `ATextFile` is clearly intended for use with a text file. The variable `AnUntypedFile` can be used for any arbitrary file and `ATypedFile` can be used for a file which is made up of `Double` values. You can see that a typed file definition is an extension of an untyped file, specifying the type of data constituting the file.

➤ *Listing 1*

```
var
  ATextFile: TextFile; { Could use System.Text instead }
  AnUntypedFile: File;
  ATypedFile: File of Double;
```

A file variable is not much use until it has been associated with a real file name – this is done with the `Assign` procedure. Again, Delphi has a number of methods called `Assign`, so it introduces a substitute called `AssignFile`. A file assignment looks like this:

```
AssignFile(ATextFile,
  'C:\AUTOEXEC.BAT');
```

Now we are ready to run: we can open the file. How we do this depends on whether we want to read or write the file, or both. A text file can be opened for reading or writing; typed and untyped files can be opened for reading, writing or both. The `Rewrite` procedure will create a new file and open it for writing, for any file type. `Append` can be used to open an existing text file so more text can be added at the end.

The `Reset` procedure operates differently depending on the file type. If given a text file, `Reset` opens the file read-only, but if it is given a typed or untyped file, it opens it in read/write mode (by default – we'll come back to how to change this) to allow random access reading and writing.

`Reset` and `Rewrite` are often used in combination to open an existing file, ensuring it exists first:

```
if not Exists(FileName) then
  {create file if not found}
  Rewrite(FileName);
Reset(FileVar); {open the file}
```

Closing the file is done with `Close`, although Delphi adds `CloseFile` as a synonym to avoid conflicts with the many `Close` methods in the VCL.

```
Reset(AnUntypedFile);
{ do something }
CloseFile(AnUntypedFile);
```

## Records And Buffers

A typed file has a clear concept of records, each component item in the file is termed a record and the record size for the file is the size of each item. For `ATypedFile` (Listing 1), the record size is the size of a `Double`, ie `SizeOf(Double)` or 8 bytes. With an untyped file it is not so clear. An impulse might be to think that an untyped file is simply a collection of bytes and so it has a record size of 1 byte. This is not necessarily the case – if it were a file of bytes, why not insist it is declared as `File of Byte`? In fact, by default, `Reset` gives an untyped file a record size of 128 bytes. It isn't quite so important these days with clever disk caching that we all tend to have, but historically, reading and writing files 128 bytes at a time is considerably more efficient than reading one byte at a time. If you want to change this default untyped file record size, you can use an optional second parameter to `Reset`, eg for a record size of 2048 bytes:

```
Reset(AnUntypedFile, 2048);
```

Text files do not have a record size as such, but there is a buffer associated with each text file which also defaults to 128 bytes and it is used to read or write text files 128 bytes at a time for efficiency. Again, you can change this if you wish, this time using the `SetTextBuf` routine, which you pass a buffer of any type to and this will be used until `Assign` or `AssignFile` is next called for the text file variable. Make sure the buffer will be in existence for as long as the text file will be open – bad things will happen if you pass a local variable which goes out of scope before the file is closed. Also, do not call `SetTextBuf` after accessing the file or you are likely to lose data. See Listing 2 for an example.

If you don't want the file to use all the buffer, you can use an optional third parameter to limit how much it uses. This extra parameter defaults to the size of the passed in buffer, here it's limited to half of the buffer size:

```
SetTextBuf(ATextFile, Buffer,
  SizeOf(Buffer) div 2);
```

Since there isn't really a record size associated with a text file, there's just a buffer that fills up as you write to it, you can end up with data written to the text file variable sitting in memory for some time (until the file is closed). If you want to ensure the data is sent to the operating system to do with what it will, but don't feel like closing the file, you can call the `Flush` routine:

```
Flush(ATextFile);
```

## Reading And Writing

So that's some housekeeping out the way; now we need to find how to access data in the file. To read and write data from text files or typed files, we use the `Read` and `Write` routines.

These will be very familiar to Borland Pascal users as the way you also read and write to the standard Input/Output devices (keyboard and screen), but perhaps not so to many Delphi users where by default there are no standard I/O devices (though you can set some up as shown later).

`Read` and `Write` are unusual calls since they can take a variable number of arguments (mind you, so do `Reset` and `SetTextBuf` and I didn't raise an eyebrow at them). It is not possible to define your own routines to do the same, although Delphi does allow it with the penalty of an extra pair of square brackets (eg, consider the `Format` function, and the `TTable.FindKey` method).

The first parameter you need to pass is the file variable and this is followed by the data you wish to write. In the case of a text file this can be integers, floating point numbers etc, each of which get written to the file as their textual equivalent. Additionally, for text files, you can use `ReadLn` and `WriteLn` to read or write whatever values, if any, are passed followed by a carriage return and line feed. For typed files, the values must match the component type of the file. Listing 3 shows examples.

Untyped files can't use those routines; instead they must use `BlockRead` and `BlockWrite`. To see the extent of file variable support in the libraries, Table 1 lists all the routines that work solely with text file variables and Table 2 shows those that work with any file variables except text files. Table 3 lists untyped file variable routines and

➤ *Listing 2*

```
var
  ATextFile: TextFile;
  Buffer: array[1..2048] of Char;
procedure DoFileStuff;
begin
  AssignFile(ATextFile, 'BIGFILE.DAT');
  Reset(ATextFile);
  try
    SetTextBuf(ATextFile, Buffer);
    { do something }
  finally
    CloseFile(ATextFile);
  end;
end;
```

➤ *Listing 3*

```
var
  ATextFile: TextFile;
  ATypedFile: File of Double;
procedure DoMoreFileStuff(Dbl: Double);
begin
  WriteLn(ATextFile, 1, ' Hello ', 5.5);
  Write(ATypedFile, Dbl);
end;
```

Table 4 has those that work with all file variables, regardless of type.

Okay, let's see some code, firstly for text files. Listing 4 shows a routine which dumps a text file to the screen by reading lines from the file until the end of file is encountered. It's used by the project DUMPTEXT.DPR on the disk, which writes out your AUTOEXEC.BAT.

Note that in a GUI Windows program there are no standard input or output devices to get data from or present information to the user by default. In Delphi 1.0x we can use the WinCrt unit to emulate them and in Delphi 2.0 we will be able to generate console mode (text mode) applications by setting a linker option. Then, whenever a `Write/WriteLn` or `Read/ReadLn` instruction appears without a file variable the standard input (keyboard) will be read from or the standard output (CRT emulation or text window) will be written to. Without WinCrt, such an instruction in a 16-bit application will give run-time error 104 (*file not open for input*) or 105 (*file not open for output*). In a 32-bit application you will get no error.

The previous routine dealt with lines as individual entities, but often this will not be a sensible approach.

Consider a tab separated data file, having 5 entries per line interspersed with tab characters. A better way to treat such a file is to use `Read` instead of `ReadLn` and check for the end of the line with `Eoln` or `SeekEoln`. To show the difference between these two functions, try

➤ *Listing 4*

```
procedure DumpTextFile(
  const FileName: String);
var
  F: TextFile;
  S: String;
begin
  try
    AssignFile(F, FileName);
    Reset(F);
    while not EOF(F) do
    begin
      ReadLn(F, S);
      WriteLn(S);
    end;
  finally
    CloseFile(F);
  end;
end;
```

| *System unit (Delphi and Borland Pascal):* | |
|---|---|
| Append | Opens a text file in write-only mode allowing text to be added at the end |
| Eoln | Returns True if the file pointer is at the end of a line |
| Flush | Flushes text file's internal 128-byte (by default) buffer to DOS |
| ReadLn | Reads a line of a text file |
| SeekEof | Same as Eof but skips past white spaces and end of lines before performing the test |
| SeekEoln | Same as Eoln but skips past white spaces before performing the test |
| SetTextBuf | Replaces the 128 byte I/O buffer with a different (usually bigger) one for more efficient text file operations |
| WriteLn | Writes a line of text at the current file pointer |

➤ *Table 1: File variable routines – text files only*

| *System unit (Delphi and Borland Pascal):* | |
|---|---|
| FilePos | Returns position of file pointer of an open file in terms of records |
| FileSize | Returns file size in terms of records |
| Seek | Moves the file pointer to a particular record |
| Truncate | Sets the current file position as the end of file |

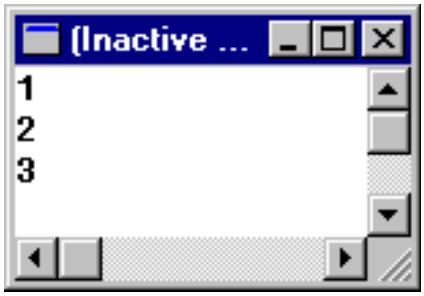
➤ *Table 2: File variable routines – non-text files only*

the routine in Listing 5 (from the program READDATA.DPR on the disk) which starts off using `Eoln`.

Notice the numbers are written out to the first line of the file with tab separators (ASCII value 9), and when all of them have been written, a line terminator is written by calling `WriteLn(F)`. The numbers 1, 2 and 3 are written to the file, but the numbers read back and written to the screen are 1, 2, 3 and 0 (see Figure 1).

The problem here is that after the 3, another tab character is written out. During the read loop, when the 3 has been read back in, the end of the line has not been reached – the tab character remains, and so another iteration of the loop takes place. No number exists after the 3, so the `Read` returns a zero. In short, the extra


➤ *Figure 1*




➤ *Figure 2*

| *System unit (Delphi and Borland Pascal):* | |
|---|---|
| BlockRead | Reads an arbitrary number of records from a file |
| BlockWrite | Writes an arbitrary number of records to a file |

➤ *Table 3: File variable routines – untyped files only*

| *System unit (Delphi and Borland Pascal):* | |
|---|---|
| Assign | Associates a file variable with an file name |
| AssignFile | Delphi substitute for Assign to avoid scoping problems |
| Close | Closes a file |
| CloseFile | Delphi substitute for Close to avoid scoping problems |
| Eof | Returns True if file pointer is at the end of the file |
| Erase | Deletes a file |
| Read | Reads data from a file |
| Rename | Renames a file |
| Reset | Opens a file – text files are read-only, other files are read/write but can be changed with `FileMode` variable |
| ReWrite | Creates and opens a new file |
| Write | Writes data to a file |
| *WinDos unit (Delphi and Borland Pascal) & Dos unit (B Pascal only):* | |
| GetFAttr | Finds a file's attributes, Delphi introduces `FileGetAttr` |
| GetFTime | Used with `UnpackTime` to find file's last modification time and date; Delphi introduces `FileGetDate` |
| SetFAttr | Sets a file's attributes. Delphi introduces `FileSetAttr` |
| SetFTime | Used with `PackTime` to set file's last modification time and date; Delphi introduces `FileSetDate` |

➤ *Table 4: File variable routines – any files*

white space mucked things up. This is where `SeekEoln` comes in. `SeekEoln` eats up any white space before deciding whether it is at the end of the line. Changing `Eoln` to `SeekEoln` changes the program to do what we would expect, as shown in Figure 2.

## Untyped Files

Now for untyped files. The example I'm using here is operating on a Windows 3.1x executable file.

In an article in *Microsoft Systems Journal* (September, 1994), Matt Pietrek tells us that Windows 95 can be of particular help for the badly behaved applications which

➤ *Listing 5*

```
procedure WriteAndReadDataFile;
var
  F: TextFile;
  Loop, Num: Byte;
begin
  AssignFile(
    F, 'C:\DELETEME.DAT');
  try
    Rewrite(F);
    for Loop := 1 to 3 do
      Write(F, Loop, #9);
    WriteLn(F);
    Reset(F);
    while not Eoln(F) do begin
      { Change Eoln to SeekEoln
        to work properly }
      Read(F, Num);
      WriteLn(Num);
    end;
  finally
    CloseFile(F);
  end;
end;
```

➤ *Below: Listing 6*

```
program ChangApp;
{$ifdef WINDOWS}
uses WinCrt;
{$else}
  {$ifndef CONSOLE}
  'Turn on Project | Options | Linker | Generate console application'
  {$endif}
{$endif}
type
  TWordRec = record
    Lo, Hi: Byte;
  end;
procedure UpdateApp(const FileName: String);
var   F: File;
      Num, NewExeOffset: Word;
const Sig: String[2] = '  ';
begin
  AssignFile(F, FileName);
  try
    Reset(F, 1);
    BlockRead(F, Sig[1], SizeOf(Word));
    if Sig <> 'MZ' then begin
      WriteLn('Not a valid EXE file');
      Exit;
    end;
    Seek(F, $18);
    BlockRead(F, Num, SizeOf(Word));
    if Num < $40 then begin
      WriteLn('Not a Windows EXE file');
      Exit;
    end;
    Seek(F, $3C);
    BlockRead(F, NewExeOffset, SizeOf(Word));
    Seek(F, NewExeOffset);
    BlockRead(F, Sig[1], SizeOf(Word));
    if Sig <> 'NE' then begin
      WriteLn('Not a Windows EXE file');
      Exit;
    end;
    Seek(F, NewExeOffset + $3E);
    BlockRead(F, Num, SizeOf(Word));
    with TWordRec(Num) do
      WriteLn('Current expected Windows version is ',
        Hi, '.', Lo);
    WriteLn('Setting to 4.0');
    Seek(F, NewExeOffset + $3E);
    Num := $400;
    BlockWrite(F, Num, SizeOf(Word));
  finally
    CloseFile(F);
  end;
end;
var Buf: Char;
    Num: Integer;
begin
  if ParamCount = 0 then begin
    WriteLn('Pass the filename as a command-line parameter');
    { ... }
    { SEE FILE CHANGAPP.DPR ON DISK FOR FULL CODE LISTING }
```

plagued Windows 3.1. In the executable file header is a number indicating the expected Windows version. Windows 3.1 applications use a word where the high byte is 3 and the low byte is 10, indicating 3.10. If this is changed to 4.00, Windows 95 will tidy up Windows resources left hanging around by sloppy programming. A Windows executable is a good example of a file with no regular structure: there are various tables and values. The format is partially documented in the Windows API help file, available in the Delphi help in the topic "Executable-File Header Format."

Listing 6 is from the program CHANGAPP.DPR on the disk. This program takes a command-line parameter (you can set one up in the Delphi IDE by selecting `Run | Parameters`) which should be the name of a Windows 3.1x executable file. The program will change the expected version to 4.0.

As it turns out, all values the program reads and writes are word-sized, but not all of them are `Word` types. The program opens the file with a record size of 1 (using `Reset(F, 1)`), so that the offsets in the .EXE file format documentation can be used readily (forgetting to amend the record size in `Reset` from the default 128 bytes is a common cause of errors when using untyped files). The first thing it does is check the first two bytes are "MZ" which implies it is an executable file. `BlockRead` is used to read them into a two byte string to facilitate this comparison.

Then the word at offset $18 is tested against $40. If the value is greater or equal, it means the value at offset $3C represents the offset in the file of the Windows header, and that header is tested to ensure it starts with "NE". $3E bytes past the start of the Windows header is the expected Windows version: this is read in and written out to the screen. A typecast is used to allow the high and low bytes to be written individually.

An alternative and perhaps more readable version of that particular section would have involved reading two individual byte values, since that is what is being written out, but never mind. Isn't hindsight a wonderful thing? If you are unfamiliar with the principles of typecasting, see my articles on the subject in Issues 3, 4 and 5.

To prove that non-text files can be used as random access read/write devices, the file pointer is moved back to the place where the version is stored and a new version is written with `BlockWrite` and finally the file is closed.

You may have noticed that I defined a record `TWordRec` to access the two bytes of a word with, in the context of a typecast. Whilst there is a perfectly suitable `WordRec` type in the SysUtils unit, I chose not to use that unit, as it carries a lot of baggage, that will be added into my EXE, of which I will be making no use, such as date and time formatting information, hardware exception handling etc.

## Typed Files

Text files and untyped files we can tick off now. Onwards to typed files. A typed file is essentially a binary data file which happens to have some order or structure.

Let's consider that we might like to store numbers from zero to 90 along with their sines, cosines and tangents in a file to save calculating them all the time in a program which has heavy trigonometry usage. Well, yes, I know I'm stretching reality a bit here – it's hardly likely to be more efficient what with all the file access overhead, but it will suffice for the purposes of our demonstration. We can define a record to hold the required information as:

```
type
  TMaths = record
    Val: Byte;
    Sine, Cosine, Tangent:
      Double;
  end;
```

A routine to set up the file could look like Listing 7 (see MATHS.DPR on the disk for an example).

If you run the program and look at the resultant file, you will see that it is an unreadable binary file. If the file is opened with `Reset` it will be a random access read/write file, just like the previous untyped file example. The `Seek` routine would allow us to seek to any individual record in the file and read or write it. If we go to the end of the file, possibly by using the call `Seek(FileSize(F))`, and write new values, the file would be extended.

```
procedure SetupFile;
var
  Maths: TMaths;
  Loop: Byte;
  F: File of TMaths;
begin
  AssignFile(
    F, 'C:\DELETEME.DAT');
  try
    Rewrite(F);
    for Loop := 1 to 90 do begin
      with Maths do begin
        Val := Loop;
        Sine := Sin(Loop);
        Cosine := Cos(Loop);
        Tangent := Sine/Cosine;
      end;
      Write(F, Maths);
    end;
  finally
    CloseFile(F);
  end;
end;
```

➤ *Listing 7*

## Next Time

We'll carry on looking at file variables, file handles and general file access routines next time.

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com