File Handling 3: File Handles

by Brian Long

A file handle is an integer number. Its meaning is not relevant for programmers other than to say that a value of -1 means that your file didn't open because something went wrong. We are advised usually not to refer to literal values like -1, but instead to use constants. When programming for 16-bit Windows in C and C++ there is a constant defined which is called `hFile_Error` and has a value of -1. Unfortunately this is one of the few items which has no equivalent definition in Delphi 1. Delphi 2 gives us both `hFile_Error` and the new Win32 constant for -1, which is `Invalid_Handle_Value`.

Apparently, a file handle is what Microsoft call a 'magic cookie,' meaning you can use it to refer to a file without understanding its value. Historically in DOS (and I think this is still the case in Windows 3.1x, but am unaware of what goes on in Win32) each program has an array of 20 bytes (though this number can be increased) called a Job File Table (JFT). A file handle is an index into your JFT. Each JFT entry was itself an index into DOS's System File Table (SFT), whose size is dictated by the `FILES=` statement in CONFIG.SYS. Each SFT entry contains information about an open file including a file name, file position, file size and date and time stamps.

So, in short, a file handle is a number. Remember that we can find a file variable's file handle by using `TFileRec(FileVar).Handle` or `TTextRec(TextVar).Handle`. Conceptually you could consider operations using a file handle to be quite similar to operations on a `file of byte`; however, file handle operations don't raise exceptions. Also, you can write data blocks greater than 64Kb to a file via a file handle. Delphi 1 file variables can't deal with data blocks over 64Kb in size. The file handle support routines in Delphi 1 and 2 are listed in Table 1.

File Handle Routines

➤ SysUtils unit:

`FileClose`	Closes a file. Synonym for `_lclose` (1.0x), `CloseHandle` (2.0).
`FileCreate`	Creates a new file.
`FileGetDate`	Replaces `GetFTime` to get file modification time/date. Used with `FileDateToDateTime`.
`FileOpen`	Opens existing file in mode specified by file open mode constants.
`FileRead`	Reads data from a file. Synonym for `_hread` and `ReadFile`.
`FileSeek`	Moves file pointer position. Synonym for `_llseek` or `SetFilePointer`.
`FileSetDate`	Replaces `SetFTime` to set file modification time/date. Used with `DateTimeToFileDate`.
`FileWrite`	Writes data to a file. Synonym for `_hwrite` and `WriteFile`.

Windows File Handle Routines

➤ WinProcs unit (Delphi 1 and Borland Pascal):

`_hread`	Like `_lread` but handles blocks over 64Kb. Not predefined in Delphi.
`_hwrite`	Like `_lwrite` but handles blocks over 64Kb. Not predefined in Delphi.
`_lclose`	Closes a file.
`_lcreat`	Creates or opens a file.
`_llseek`	Repositions the file pointer.
`_lopen`	Opens a file.
`_lread`	Read data from a file.
`_lwrite`	Writes data to a file.
`OpenFile`	Creates, opens, reopens or deletes a file.
`SetHandleCount`	Changes the number of file handles available to a task. The Controls unit uses this API to set 255 file handles.

➤ Windows unit (Delphi 2):

`_hread`	For compatibility with 16-bit Windows.
`_hwrite`	For compatibility with 16-bit Windows.
`_lclose`	For compatibility with 16-bit Windows.
`_lcreat`	For compatibility with 16-bit Windows.
`_llseek`	For compatibility with 16-bit Windows.
`_lopen`	For compatibility with 16-bit Windows.
`_lread`	For compatibility with 16-bit Windows.
`_lwrite`	For compatibility with 16-bit Windows.
`CloseHandle`	Close file.
`CreateFile`	Create/open/truncate a file.
`FlushFileBuffers`	Write files to disk.
`GetFileSize`	Find size of file.
`LockFile`	Lock area of a file.
`LockFileEx`	Not implemented in Windows 95.
`OpenFile`	For compatibility with 16-bit Windows.
`ReadFile`	Read data from a file.
`ReadFileEx`	Not implemented in Windows 95.
`SetEndOfFile`	Sets the current file position as the end of file.
`SetFilePointer`	Repositions the file pointer.
`SetHandleCount`	For compatibility with 16-bit Windows.
`UnlockFile`	Unlock area of file.
`UnlockFileEx`	Not implemented in Windows 95.
`WriteFile`	Writes data to a file.
`WriteFileEx`	Not implemented in Windows 95.

➤ *Table 1: File handle support routines in Delphi 1 and 2*

If you are still working with Delphi version 1.00 or 1.01 (your DELPHI.EXE time stamp won't be 8:02) then your Help page for FileOpen erroneously says that it is an internal routine. So does FileCreate's page, but at least that describes the function. These mistakes were corrected in the 1.02 maintenance release, but for those without that version, here is the text (with a spelling correction):

"FileOpen *opens the specified file using specified access mode. The access mode is constructed by* Or*-ing one of the* fmOpenXXX *constants with one of the* fmShareXXX *constants. If the return value is positive, the function was successful and the value is the file handle for the opened file. If the return value is negative, an error occurred and the value is a negative DOS error code.*"

Note that the return value for FileOpen (and indeed for some other routines) isn't necessarily -1 for an error condition: it could be any of a range of negative numbers. This is in contrast to Delphi 2 where an error does yield -1 (or hFile_Error) and additional error information can be gleaned by using the GetLastError API. The reason for the difference is that Delphi 1 implements FileOpen, among other routines, by calling DOS interrupts. Calling interrupts is taboo in 32-bit and so these routines are now implemented by calling appropriate Win32 APIs (ie CreateFile with appropriate parameters).

The likely DOS error values for the 16-bit FileOpen and FileCreate are shown in Table 2.

If you have been paying attention to what functionality was available for file variables, you may feel the 16-bit file handle support is a bit limited. There is no direct support for finding a file's size, or the current file position. But fear not, because we can still get hold of this information. The routine FileSeek (which is implemented by a call to _llseek in Win16 or SetFilePointer in Win32) can move to any place in your file and then return that file position. We specify where to move to by giving it an offset (a number of bytes) and a symbol indicating an origin to seek from which can mean from the beginning of the file (0), from the current position (1) or from the end of the file (2). The symbols have different constant names, depending which function you are calling, as shown in Table 3.

To find our current position, we can seek zero bytes from the current position and we will be told how far through the file we are. To find the file size, we need to do a similar thing, but save the current position, then seek zero bytes from the end of the file, recording the position, which will be the file size. To get back to where we were we can seek from the beginning of the file, specifying our saved position as the number of bytes.

Words Of Warning

When dealing with files, the data types used for storage need to be thought about if portability of data files between 16-bit and 32-bit Delphi programs is to be maintained. If integral values are being stored, remember to use Smallint or Word, rather than Integer or Cardinal for signed and unsigned 16-bit numbers respectively. Delphi 2, like Delphi 1, interprets Smallint and Word as 16-bit values, whereas Integer and Cardinal become 32-bit values in Delphi 2.

Records get laid out in memory differently by default. Instead of each field immediately following the previous one, Delphi 2 ensures each one starts at a suitable boundary (dependent on the field size) for efficient access. This means there may be spare bytes in your records which will cause the record size to increase and break programs that read data from Delphi 1 days. To prevent problems, precede the keyword record with the keyword packed (this used to be ineffective, but it is now significant), as shown in Listing 1.

Also, string data needs to be considered carefully. If you have 16-bit programs writing data out to non-text files which include strings, ensure you either disable huge string support for the areas of your Delphi 2 program that deal with the file I/O, or alternatively explicitly declare your strings as short strings (eg S: String[255] – the length limit in square brackets makes a Delphi 1 compatible short string).

Another gotcha with strings in Delphi 2 comes up with FileWrite. The second parameter of FileWrite is an untyped variable: you pass some data and it takes the address of that data and passes it through. In Delphi 1 you can pass a string in, maybe something like the example in Listing 2.

➤ *Listing 1*

```
type
  TUnsafeRecord = record
    Ch: Char;
    L: Longint;
    B: Boolean;
  end; { 12 bytes in Delphi 2,
         6 bytes in Delphi 1 }
  TSafeRecord = packed record
    Ch: Char;
    L: Longint;
    B: Boolean;
  end; { 6 bytes in Delphi 1 & 2}
```

➤ *Table 2: DOS error values for 16-bit FileOpen and FileCreate*

Value	Meaning
-2	File not found
-3	Path not found
-4	Too many open files
-5	Access denied
-12	Invalid access mode

➤ *Table 3: File seeking constants*

Origin	Value	FileSeek Constant	_llseek Constant	SetFilePointer Constant
Beginning of file	0	soFromBeginning	Seek_Set	File_Begin
Current position	1	soFromCurrent	Seek_Cur	File_Current
End of file	2	soFromEnd	Seek_End	File_End

```
var
  Handle: Integer;
  S: String;
...
Handle := FileCreate('c:\delme.dat');
FileClose(Handle);
...
Handle := FileOpen('c:\delme.dat', fmOpenReadWrite or fmShareDenyNone);
S := Edit1.Text;
FileWrite(Handle, S, SizeOf(S));
...
FileSeek(Handle, 0, soFromBeginning);
FileRead(Handle, S, SizeOf(S));
FileClose(Handle);
Caption := S;
```

➤ *Listing 2*

```
var
  Handle: Integer;
  S: String;
  Len: Byte;
...
Handle := FileCreate('c:\delme.dat');
FileClose(Handle);
...
Handle := FileOpen('c:\delme.dat', fmOpenReadWrite or fmShareDenyNone);
S := Edit1.Text;
Len := Length(S);
FileWrite(Handle, Len, SizeOf(Byte));
SetLength(S, 255);
FileWrite(Handle, S[1], 255);
SetLength(S, Len);
...
FileSeek(Handle, 0, soFromBeginning);
FileRead(Handle, Len, SizeOf(Byte));
SetLength(S, 255);
FileRead(Handle, S[1], 255);
SetLength(S, Len);
FileClose(Handle);
Caption := S;
```

➤ *Listing 3*

This writes a whole string variable out, a 256 byte block of data. Because of the use of a `var` parameter, things turn pear-shaped in Delphi 2. The new huge strings are implemented via an implicit pointer. A string variable is really a pointer to the string data. If you try and pass a huge string to `FileWrite`, it passes the address of the pointer and all you get in the file is the value of that pointer plus a load of garbage. Instead, you need to pass `S[1]`, so the address of the first character would be passed.

If a Delphi 1 application wrote a string out as above, a Delphi 2 huge string program would need to be fiddly to match its operation and maintain the file structure and layout. Strings are managed by dynamic allocation of memory, which is increased when a string is written to using normal string operations, but not using a memory write operation as `FileRead` does. Bearing this in mind, we could use code like that in Listing 3.

The first `SetLength` in the writing section makes sure there are definitely 255 valid bytes which `FileWrite` can write to the file. If there were only, say, 5, then we would risk an access violation. The second one restores the string to its old length. The first `SetLength` in the reading section causes enough memory to be available to write the 255 characters to and then the second one ensures that the string thinks of itself with the correct length. However, the string storage here is inefficient: 256 bytes for each string. Let's change the Delphi 1 writing and reading code to this:

```
FileWrite(Handle, S, Length(S) + 1);
...
FileRead(Handle, S[0], 1);
FileRead(Handle, S[1], Length(S));
```

The Delphi 2 reading code now turns into:

```
Len := Length(S);
FileWrite(Handle, Len, 1);
FileWrite(Handle, S[1],
  Length(S));
...
FileRead(Handle, Len, 1);
SetLength(S, Len);
FileRead(Handle, S[1], Len);
```

Sample Implementations

To do a bit of a recap of what has been covered so far in these articles I have implemented a reasonably simple program that uses a structured data file. To see the different options available for file handling, the program will be first written using a typed variable, and then with an untyped file variable, and yet again with file handles.

The program is not particularly adventurous, for simplicity of reading: it allows storage of records of information consisting of a name and a date of birth. The file will not exist to start with and so the program needs to know how to create it and must obviously support adding records. It will also attempt to handle the various error situations which may arise, using the natural error mechanism of the file system used (ie exceptions for file variables).

The program (NAMES1.DPR) is split over two units. The first is the main form unit, NAMES1U.PAS, and implements the user interface. I won't go into what it does in any detail, though I'll mention that it spends some time ensuring that clipboard-oriented menu items and speed buttons are enabled and disabled when appropriate, and the same for navigational menu items and speed buttons. Additionally it implements the functionality behind those buttons/menus. Figure 1 shows the program.

The important part of the code is left to the second unit (called NAMES1U2.PAS), which has nothing to do with the user interface. This unit implements a class to represent a data file and also defines the data record. The plan is that only this second unit will need adjusting for the different file types.

The TDataFile class interface will remain static, but the implementation will change, as you can see in Listing 4.

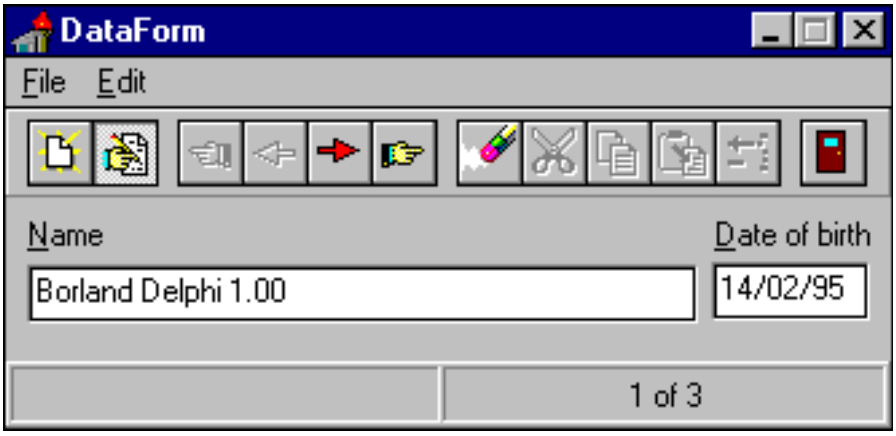
There are seven principal methods of interest (mostly property access methods) which all deal with the file variable. Some are fairly simplistic, like GetCount which returns the number of records in the file and GetCurrent which reports the position in the file. You can see from Listing 5 that they are simple wrappers around FileSize and FilePos. The code in SetCurrent is almost only a call to Seek, but there is a check to ensure that the caller is not requesting a position past the end of the file.

The more interesting methods are the constructor and destructor, and also the Records property access methods GetRecord and SetRecord.

The constructor changes to the directory where the program's EXE file resides and sets FileMode up for normal sharing. If there is no data file found an attempt is made to make one, followed by the normal call to Reset which should open the file in sharing mode. If there is a problem, for example another program already has the file open in a mode that excludes us from writing, then an exception is raised and caught. Before re-raising the error with a custom message, code executes to ensure the file is definitely shut, taking advantage of the file variable structure record TFileRec and one of the file open mode constants, fmInOut.

The destructor ensures the file is closed before letting the object destroy itself. If you try and close a file that is not open, you get an exception. Some would say that these days you should be in the mindset of exception handling and just close the file, but trap an exception if one occurs. However, doing it the way I've shown demonstrates a use for the Mode field in the TFileRec structure.

GetRecord retrieves a specified record by positioning the file pointer and using Read. If Read yields any kind of problem an *Index out of bounds* exception is raised to suggest that maybe an invalid



➤ *Figure 1*

```
type
  TDataRec = packed record
    { The form's edit box has its MaxLength property set to 30 }
    Name: String[30];
    { Only interested in the date portion of this date/time value }
    DOB: TDateTime;
  end;
  TDataFile = class
  private
    FDataFile: File of TDataRec;
  protected
    function GetCount: Longint;
    function GetCurrent: Longint;
    function GetRecord(Index: Longint): TDataRec;
    procedure SetCurrent(RecNo: Longint);
    procedure SetRecord(Index: Longint; const DataRec: TDataRec);
  public
    constructor Create;
    destructor Destroy; override;
    property Count: Longint read GetCount;
    property Current: Longint
      read GetCurrent write SetCurrent;
    property Records[Index: Longint]: TDataRec
      read GetRecord write SetRecord; default;
  end;
```

➤ *Listing 4*

record was requested. SetRecord is less trivial – it tries to lock a record in the file so it can exclusively write a record without fear of conflict with anyone else. If the lock cannot be placed an I/O exception is raised, otherwise the record is written and the lock is removed. The locking unit NETLOCK.PAS is a slightly updated version of the file supplied with the last issue (it has conditional compilation for Win32) and is included on the disk of course.

The reason for including the TDataRec((@DataRec)^) element in the call to write is simply to show how you can get around the normal restrictions on const parameters, ie that they can't be passed as var parameters. Normally I wouldn't write such an expression. It would be clearer to declare a local variable of type TDataRec, assign DataRec to it and pass that instead.

Version 2

When rewriting this for untyped files (see the project NAMES2.DPR on the disk), there are very few changes to make (four in total). First the definition of FDataFile needs to change to be:

```
FDataFile: File;
```

To give the untyped file a record size, we modify the call to Reset in the constructor:

```
Reset(FDataFile,
  SizeOf(TDataRec));
```

Finally, GetRecord and SetRecord

```
uses
  Forms, NetLock, Consts, Classes;
const
  FileName = 'DataFile.Dat';
constructor TDataFile.Create;
begin
  { Make current directory where EXE file is, just in case }
  ChDir(ExtractFilePath(Application.ExeName));
  AssignFile(FDataFile, FileName);
  FileMode := fmOpenReadWrite or fmShareDenyNone;
  try
    { Make file if it ain't there }
    if not FileExists(FileName) then
      Rewrite(FDataFile);
    Reset(FDataFile);
  except
    on E: EInOutError do begin
      { In case Rewrite succeeded but Reset failed }
      if TFileRec(FDataFile).Mode = fmInOut then
        CloseFile(FDataFile);
      { Customise the exception and re-raise it }
      E.Message := 'Failed to create or open ' + FileName;
      raise;
    end;
  end;
end;
destructor TDataFile.Destroy;
begin
  if TFileRec(FDataFile).Mode = fmInOut then
    CloseFile(FDataFile);
  inherited Destroy;
end;
function TDataFile.GetCount: Longint;
begin
  Result := FileSize(FDataFile);
end;
function TDataFile.GetCurrent: Longint;
begin
  Result := FilePos(FDataFile);
end;
function TDataFile.GetRecord(Index: Longint): TDataRec;
begin
  try
    Current := Index;
    Read(FDataFile, Result);
    { Go back to the beginning of the read record }
    Current := Index;
  except
    raise EListError.CreateRes(SListIndexError);
  end;
end;
procedure TDataFile.SetCurrent(RecNo: Longint);
begin
  { Anything past EOF is considered EOF }
  if RecNo > Count then
    RecNo := Count;
  Seek(FDataFile, RecNo);
end;
procedure TDataFile.SetRecord(Index: Longint; const DataRec: TDataRec);
var X: EInOutError;
begin
  Current := Index;
  if not LockFileVar(FDataFile, Current, False) then begin
    X := EInOutError.Create('Cannot lock file');
    { Set up a file access denied type exception }
    X.ErrorCode := 5;
    raise X;
  end;
  try
    { DataRec is passed as a const (pass by reference, but }
    { not allowed to be treated/passed as a var parameter). }
    { We can get around this by dereferencing its }
    { address with an appropriate typecast }
    Write(FDataFile, TDataRec((@DataRec)^));
    { Go back to the beginning of the written record }
    Current := Index;
  finally
    LockFileVar(FDataFile, Current, False);
  end;
end;
```

➤ *Listing 5*

must use `BlockRead` and `BlockWrite` instead of `Read` and `Write`:

```
BlockRead(FDataFile, Result,
  1, Count);
...
BlockWrite(FDataFile,
  TDataRec((@DataRec)^),
  1, Count);
```

The fourth parameter is an optional `Word` variable which is used to detect problems. It returns the number of records (since we set a record size) read or written. If it is less than we requested, ie if it is zero, then there was a file error. Note that the third and fourth parameters of `BlockRead` and `BlockWrite` are both of `Word` type in Delphi 1 (ie range 0..65535), but are `Integer` types in Delphi 2 (ie range -2Gb..2Gb-1). This change is to cater for the 32-bit file system which deals in 32-bit values. In my program they are defined as `Cardinal`, so they will be 16-bit in Delphi 1 and 32-bit in Delphi 2.

Version 3

The version for file handles (NAMES3.DPR) has a few more involved changes in NAMES3U2.PAS. `FDataFile` is now defined as an `Integer`. The constructor contains no exception handling, as file handle routines don't generate exceptions, although it does generate some exceptions if there is a problem.

In order to cater for the lack of a `FileSize` routine I implemented one using the logic described earlier. This isn't necessary in Win32, since it has its own `GetFileSize` routine. There is conditional compilation to call the relevant one in `GetCount`. `GetRecord` and `SetRecord` are modified to use `FileRead` and `FileWrite`, but also `SetRecord` uses the `LockFileArea` call instead of `LockFileVar`. As an example of the changes, `SetCurrent` is shown in Listing 6.

More Routines

We will not be looking at the Windows file handling APIs as they tend to vary between Windows version. However, I have listed them in the various tables so you

```
procedure TDataFile.SetCurrent(
  RecNo: Longint);
begin
  { Anything past EOF is
    considered EOF }
  if RecNo > Count then
    RecNo := Count;
  FileSeek(FDataFile,
    RecNo * SizeOf(TDataRec),
    soFromBeginning);
end;
```

➤ *Listing 6*

can look up their details in the online help.

To give you even more food for thought, Tables 4, 5 and 6 list a plethora of file-oriented bits and pieces, including miscellaneous routines, directory and disk routines and also constants and data structures.

In the next article we will delve into streaming.

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

System unit (Delphi and Borland Pascal):

IOResult	Returns value of last I/O error if I/O checking is disabled.

SysUtils unit:

ChangeFileExt	Given a file name, this returns a string with the extension changed.
DateTimeToFileDate	Replaces PackDate to turn Delphi date/time into DOS date/time. Used with FileSetDate.
DeleteFile	Deletes a file.
ExpandFileName	Returns absolute file specification. Replaces FileExpand, FExpand.
ExtractFileExt	Given a full file specification, returns the extension. Replaces FileSplit, FSplit.
ExtractFileName	Given a full file specification, returns the name including extension. Replaces FileSplit, FSplit.
ExtractFilePath	Given a full file specification, returns the path. Replaces FileSplit, FSplit.
FileAge	Used with FileDateToDateTime returns age of file.
FileDateToDateTime	Replaces UnpackDate to turn DOS date/time into Delphi date/time for FileAge, FileGetDate, TSearchRec.
FileExists	Returns True if file exists.
FileGetAttr	Replaces GetFAttr for finding file attributes.
FileSearch	Locates a file on a given path.
FileSetAttr	Replaces SetFAttr for setting file attributes.
FindClose	Terminates a FindFirst, FindNext sequence.
FindFirst	Finds first occurrence of a file specification (can handle wildcards).
FindNext	Finds next occurrence of a file specification (can handle wildcards).
RenameFile	Renames a file.

WinDos unit (Delphi 1 and Borland Pascal):

FileExpand	Returns absolute file specification. Delphi introduces ExpandFileName.
FileSearch	Locates a file on a given path.
FileSplit	Splits a file specification into path, directory, name and extension. Delphi introduces ExtractFileExt, ExtractFileName and ExtractFilePath.
FindFirst	Finds first occurrence of a file specification (handles wildcards).
FindNext	Finds next occurrence of a file specification (handles wildcards).
PackTime	Became DateTimeToFileDate. Used with SetFTime to turn DOS date/time to DateTime or TDateTime.
UnpackTime	Became FileDateToDateTime. Used with GetFTime, TSearchRec, SearchRec to turn DateTime or TDateTime into DOS date/time.

Windows unit (Delphi 2):

DeleteFile	Deletes a file.
CopyFile	Copies an existing file to a new file.
FindClose	Terminates a FindFirstFile, FindNextFile sequence. Note the same name as the System unit procedure.
FindFirstFile	Finds first occurrence of a file specification (handles wildcards).
FindNextFile	Finds next occurrence of a file specification (handles wildcards).
GetBinaryType	Identifies application type (eg Win16, Win32, DOS, OS/2).
GetFileAttributes	Gets attributes of a file or directory.
GetFileTime	Gets file time stamp.
GetFullPathName	Returns full path and file name of a file (even an 8.3 file name).
GetShortPathName	Returns the short path form of the specified input path.
MoveFile	Renames a file or directory.
MoveFileEx	Not implemented in Windows 95.
SearchPath	Searches for a specified file.
SetFileAttributes	Sets file attributes.
SetFileTime	Sets file timestamp.

➤ *Table 4: Miscellaneous routines*

System unit (Delphi and Borland Pascal):

`ChDir`	Changes current directory. Replaces `CreateDir`.
`GetDir`	Gets current directory. Replaces `GetCurDir`.
`MkDir`	Makes new directory. Replaces `CreateDir`.
`RmDir`	Removes a directory. Replaces `RemoveDir`.

SysUtils unit:

`DiskFree`	Returns free disk space.
`DiskSize`	Returns disk size.

WinProcs unit (Delphi 1 and Borland Pascal):

`GetDriveType`	Determines if a drive is removable, fixed or remote.
`GetSystemDirectory`	Retrieves Windows system directory path.
`GetTempDrive`	Returns a drive letter where temporary files may be stored.
`GetTempFileName`	Creates a temporary file.
`GetWindowsDirectory`	Retrieves Windows directory path.

WinDos unit (Delphi 1 and Borland Pascal):

`DiskFree`	Returns free disk space.
`DiskSize`	Returns disk size.
`CreateDir`	Changes current directory. Delphi introduces `MkDir`.
`GetCurDir`	Gets current directory. Delphi introduces `GetDir`.
`RemoveDir`	Makes new directory. Delphi introduces `RmDir`.
`SetCurDir`	Removes a directory. Delphi introduces `ChDir`.

Windows unit (Delphi 2):

`CreateDirectory`	Makes a new directory.
`CreateDirectoryEx`	Makes a new directory with the attributes of another directory.
`GetCurrentDirectory`	Gets current directory.
`GetDiskFreeSpace`	Gives information about free space on disk.
`GetDriveType`	Identifies drive type.
`GetLogicalDrives`	Identifies currently available drives.
`GetLogicalDriveStrings`	Returns names of currently available drives.
`GetSystemDirectory`	Retrieves Windows system directory path.
`GetTempFileName`	Creates a temporary file.
`GetTempPath`	Returns a path where temporary files may be stored.
`GetVolumeInformation`	Gives various information about a file system and volume.
`GetWindowsDirectory`	Retrieves Windows directory path.
`RemoveDirectory`	Removes a directory.
`SetCurrentDirectory`	Changes current directory.
`SetVolumeLabel`	Changes file system volume label.

➤ *Table 5: Directory/disk routines*

System unit (Delphi and Borland Pascal):

`File`	Untyped file type.
`File of ****`	Typed file type, eg `File of Double`.
`FileMode`	Affects how `Reset` opens a file.
`Text`	Text file type.
`TextFile`	Delphi substitute for `Text` to avoid scoping problems.

SysUtils unit:

`fa****`	Used by `FileSetAttr`, `FileGetAttr`, `TSearchRec`, eg `faReadOnly`.
`fm****`	File open mode constants, used by `FileMode` variable, eg `fmOpenReadWrite`.
`fm****`	File mode constants, used by `TTextRec` and `TFileRec`, eg `fmClosed`.
`TFileName`	Generic file name type.
`TFileRec`	Non-text file internal representation.
`TSearchRec`	Search record used by `FindFirst`, `FindNext`.
`TTextRec`	Text file internal representation.

Classes unit:

`soFrom****`	Seek origin constants for `FileSeek` and `TStream.Seek` eg `soFromEnd`.

WinTypes unit (Delphi 1):

`Seek_****`	Seek origin constants for `_llseek`, eg `Seek_End`.

Windows unit (Delphi 2):

`File_****`	Seek origin constants for `SetFilePointer` and `_llseek`, eg `File_End`.

WinDos unit (Delphi 1 and Borland Pascal):

`DosError`	Where DOS file errors are reported.
`fa****`	File attribute constants, used by `GetFAttr`, `SetFAttr`, `TSearchRec`, eg `faReadOnly`.
`fc****`	File component constants, used by `FileSplit`, eg `fcDirectory`.
`fm****`	File open mode constants and file mode constants, used by `TTextRec` and `TFileRec`, eg `fmClosed`.
`fs****`	File name component string lengths, eg `fsPathName`, `fsExtension`.
`TDateTime`	Used by `GetFTime`, `SetFTime`, `PackTime`, `UnpackTime`.
`TFileRec`	Non-text file internal representation.
`TSearchRec`	Search record used by `FindFirst`, `FindNext`.
`TTextRec`	Text file internal representation.

For more information on all the routines, types, variables and constants listed in these tables, check Delphi's online help

➤ *Table 6: File types, variables and constants*