C:\WINDOWS\WORD\DOT\!5X8.DOT

NS_Text

v0.6

tTextFile

C:\WINDOWS\WORD\DOT\!5X8.DOT

Preface

The *System* unit is Turbo Pascal's run-time library. It implements low-level, run-time support routines for all built-in features, such as file I/O, string handling, floating point, dynamic memory allocation, and text files. The *System* unit is used automatically by any unit or program and doesn't need to be referred to in a **Uses** clause.

The *WinDOS* unit implements a number of operating system and file-handling routines. None of the routines in the *WinDOS* unit are defined by standard Pascal, so they have been placed in their own module.

The *NS_Text* unit binds all the functions, procedures, types, constants, and variables from the *System* and *WinDOS* units into 1 object thus allowing the programmer to treat each file as a separate entity (instance) and freeing him from most of the lower level parameter and exception (error checking) maintenance. *NS_Text* can be used as a platform to build more specialized text handling objects, devices, typed or untyped files. (When creating special device handling objects from *NS_Text* some routines may not apply. To circumvent this situation you might wish to override (nullify) them with empty methods.)

+ In order to use the features of the *NS_Text* object, you must include a reference to ***NS_Text*** in your **Uses** clause.

C:\WINDOWS\WORD\DOT\!5X8.DOT

What's New to v0.6

- € Added methods SetFAttr & GetFAttr to manipulate file attributes
- € Added methods SetFTime & GetFTime to manipulate file time stamps
- € Added ability to copy file within its instance (i.e. no longer need to destruct instance, copy file, & recreate instance)
- € Added ability to instantate leaving file closed (via NS_Closed enumerated type)
- € Added method FileSplit to return subcomponents of the file name
- € Methods made recursive where possible to reduce code / increase integrity
- € Private method CClose added to increase integrity

C:\WINDOWS\WORD\DOT\!5X8.DOT

An Introduction to Using Text Files

A Pascal file variable is any variable whose type is a file type. There are three classes of Pascal files: *typed, text,* and *untyped.* The NS_Text unit was designed with the *text* class in mind. So we will ignore the other two classes in this document.

Before a file variable can be used, it must be associated with an external file through a call to the *Assign* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be opened to prepare it for input or output. An existing file can be opened via the *Reset* procedure and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only.

+ When using the *Rewrite* procedure, it ***will*** overwrite any existing files with the same name. Take precautions to make a backup if there is a possibility of loosing important data.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. Each component has a component number. The first component of a file is considered to be component zero.

Files are normally accessed sequentially; that is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write,* the current file position moves to the next numerically-ordered file component.

When a program completes processing a file, the file must be closed using the standard procedure *Close*. After closing a file completely, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors: if an error occurs, the program terminates, displaying a run-time error message. This automatic checking can be turned on and off using the **{$I+}** and **{$I-}** compiler directives. When I/O checking is off that is, when a procedure or function call is compiled in the **{$I-}** state an I/O error does not cause the program to halt. To check the result of an I/O operation, you must instead call the standard function *IOResult.*

+ In Turbo Pascal the type *Text* is distinct from the type *File of Char*.

Text Files

When a text file is opened, the external file is interpreted in a special way: it is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a line-feed character).

For text files, there are special forms of *Read* and *Write* that allow you to read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type Integer variable, will read a sequence of digits, interpret that sequence as a decimal integer, and store it in *I*.

Turbo Pascal defines two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input file (typically the keyboard), and the standard file variable *Output* is a write-only file associated with the operating system's standard output file (typically the display).

Since Windows doesn't directly support text-oriented input and output, the *Input* and *Output* files are by default unassigned in a Windows application, and any attempt to read or write to them will produce an I/O error. However, if an application uses the *WinCrt* unit, *Input* and *Output* will refer to a scrollable text window. *WinCrt* contains the complete control logic required to emulate a text screen in the Windows environment, and no Windows-specific programming is required in an application that uses *WinCrt*.

Some of the standard procedures and functions associated with text files do not need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* are assumed by default, depending on whether the procedure or function is input or output oriented. For instance, *Read(X)* corresponds to *Read(Input, X)* and *Write(X)* corresponds to *Write(Output, X)*.

If you do not specify a file when calling one of the procedures or functions in this section, the file must have been associated with an external file using *Assign*, and opened using *Reset, Rewrite*, or *Append*. An error message is generated if you pass a file that was opened with *Reset* to an output-oriented procedure or function. Likewise, it's an error to pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Devices

Turbo Pascal and the DOS operating system regard external hardware, such as the keyboard, the display, and the printer, as *devices*. From the programmer's point of view, a device is treated as a file, and is operated on through the same standard procedures and functions as files.

Turbo Pascal supports two kinds of devices: DOS devices and text file devices.

DOS Devices

DOS Devices are implemented through reserved file names that have a special meaning attached to them. DOS devices are completely transparent in fact, Turbo Pascal is not even aware when a file variable refers to a device instead of a disk file. For example, the program

```
Var
  Lst : Text;

Begin
  Assign(Lst, 'LPT1');
  ReWrite(Lst);
  WriteLn(Lst, 'Hello World...');
  Close(Lst);
End.
```

writes the string *Hello World...* on the printer, even though the syntax for doing so is exactly the same for a disk file.

The devices implemented by DOS are used for obtaining or presenting legible input or output. Therefor, DOS devices are normally used only in connection with text files.

In general, you should avoid using DOS device devices under Windows and you should use the device I/O functions provided by the Windows API instead. Although some DOS devices, such as LPT1, may work others, such as CON, will not function properly.

Text File Devices

Text file devices are used to implement devices unsupported by DOS or to provide another set of features similar to those supplied by another DOS device. A good example of a text file device is the CRT window implemented by the *WinCrt* standard unit. It provides a terminal-like text screen in a window and allows you to create Standard I/O applications under Windows with a minimum of effort.

Unlike DOS devices, text file devices have no reserved file names; in fact, they have no file names at all. Instead, a file is associated with a text file device through a customized *Assign* procedure. For instance, the *WinCrt* standard unit implements an *AssignCrt* procedure that associates text files with the CRT window.

A *text file device driver* is a set of four functions that completely implement an interface between Turbo Pascal's file system and some device. These four functions are *Open, InOut, Flush,* and *Close.* The function header of each function is

```
Function DeviceFunc(Var F:TTextRec) : Integer;
```

where *TTextRec* is the text file record type defined in Chapter 3 of the Programmers Guide. Each function must be compiled in the **{$F+}** state to force it to use the far call model. The return value of a device interface function becomes the value returned by *IOResult*. The return value of 0 indicates a successful operation.

To associate the device interface functions with a specific file, you must write a customized *Assign* procedure (like the *AssignCrt* procedure in the *WinCrt* unit). The *Assign* procedure must assign the addresses of the four device interface functions to the four function pointers in the text file variable. In addition, it should store the *fmClosed* magic constant in the *Mode* field, store the size of the text file buffer in *BufSize*, store a pointer to the text file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device interface functions are called *DevOpen, DevInOut, DevFlush,* and *DevClose*, the *Assign* procedure might look like this:

```
Procedure AssignDev(Var F:Text);
  Begin
    With TextRec(F) Do
      Begin
        Mode     := fmClosed;
        BufSize  := SizeOf(Buffer);
        BufPtr   := @Buffer;
        OpenFunc  := @DevOpen;
        InOutFunc := @DevInOut;
        FlushFunc := @DevFlush;
        CloseFunc := @DevClose;
        Name[0]  := #0;
      End;
    {EndWith}
  End
{EndProcedure}
```

The device interface functions can use the *UserData* field in the file record to store private information. This field is not modified by the Turbo Pascal file system at any time.

The Open Function

The *Open* function is called by the *Reset, ReWrite,* and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput, fmOutput,* or *fmInOut* to indicate whether the *Open* function was called from *Reset, ReWrite,* or *Append*.
The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified in *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutPut* before *Open* returns.
Open is always called before any of the other device interface functions. For that reason, *Assign* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode, Open* can then install pointers to either input or output oriented functions. This saves the *InOut, Flush,* and *Close* functions from determining the current mode.

The InOut Function

The *InOut* function is called by the *Read, ReadLn, Write, WriteLn, Eof, Eoln, SeekEof, SeekEoln,* and *Close* standard procedures and functions whenever input or output from the device is required.
When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into *BufPtr^*, and returns the number of characters read into *BufEnd*. In addition, it stores 0 in *BufPos*. If the *InOut* function returns 0 in *BufEnd* as a result of in input request, *Eof* becomes True for the file.
When *Mode* is *fmOutPut*, the *InOut* function writes *BufPos* characters from *BufPtr^*, and returns 0 in *BufPos*.

The Flush Function

The *Flush* function is called at the end of each *Read, ReadLn, Write,* and *WriteLn*. It can optionally flush the text file buffer.
If *Mode* is *fmInput*, the *Flush* function can store 0 in *BufPos* and *BufEnd* to flush the remaining (un-read) characters in the buffer. This feature is seldom used.
If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer, exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text will not appear on the device until the buffer becomes full or the file is closed.

The Close Function

The *Close* function is called by the *Close* standard procedure to close a text file associated with a device. (The *Reset, ReWrite,* and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutPut*, then before calling *Close*, Turbo Pascal's file system calls *InOut* to ensure that all characters have been written to the device.

Predeclared *System* Variables

Besides procedures and functions, the Turbo Pascal *System* unit provides a number of predeclared variables.

Variable	Type	Init	Val	Description
Input	Text	N/A	Input	standard file
Output	Text	N/A	Output	standard file
ErrorAddr	Pointer	nil	Run-time	error address
ExitProc	Pointer	nil	Exit	procedure
InOutRes	Integer	0	I/O	result buffer

The *ExitProc* and *ErrorAddr* variables are used to implement exit procedures. These exit procedures can be helpful in closing any open files upon the applications exit but will not be discussed here. You can find more detail on these variables in Chapter 18 of the Programmers Guide.
The built-in I/O routines use *InOutRes* to store the value that the next call to the *IOResult* standard function will return.
Input and *Output* are the standard I/O files required by every Pascal implementation.

Predeclared *WinDOS* Variables

File Mode Constants

The file-handling procedures use these constants when opening and closing disk files. The mode fields of Turbo Pascal's file variables will contain one of the values specified in the following:

Constant	Value

fmClosed	$D7B0
fmInput	$D7B1
fmOutput	$D7B2
fmInOut	$D7B3

File Attribute Constants

These constants test, set, and clear file attribute bits in connection with the *GetFAttr, SetFAttr, FindFirst,* and *FindNext* procedures:

Constant	Value
faReadOnly	$01
faHidden	$02
faSysFile	$04
faVolumeID	$08
faDirectory	$10
faArchive	$20
faAnyFile	$3F

The constants are additive, that is, the statement

```
FindFirst('*.*', faReadOnly + faDirectory, S);
```

will locate all normal files as well as read-only files and subdirectories in the current directory. The *faAnyFile* constant is simply the sum of all attributes.

File String Constants

These constants are the maximum file name component string lengths used by the functions *FileSearch* and *FileExpand*.

Constant	Value
fsPathName	79
fsDirectory	67
fsFileName	8
fsExtension	4

FileSplit Return Flag Constants

These return flags are used by the function *FileSplit*. The return value is a combination of the *fcDirectory, fcFileName,* and *fcExtension* bit masks, indicating which components were present in the path. If the name or extension contains any wildcard characters (* or ?), the *fcWildCards* flag is set in the return value.

Constant	Value
fcExtension	$0001
fcFileName	$0002
fcDirectory	$0004
fcWildCards	$0008

File Record Types

The record definitions used internally by Turbo Pascal are also declared in the *WinDOS* unit. *TFileRec* is used for both typed and untyped files, while *TTextRec* is the internal format of a variable of type Text.

```
Type
  TFileRec = Record                    { Typed and untyped files }
    Handle    : Word;
    Mode      : Word;
    RecSize   : Word;
    Private   : Array [1..26] Of Byte;
    UserData  : Array [1..16] of Byte;
    Name      : Array [0..79] Of Char;
  End; {Record}
  PTextBuf = ^TTextBuf;                      { TextFile record }
  TTextBuf = Array [0..127] Of Char;
  TTextRec = Record
    Handle    : Word;
    Mode      : Word;
    BufSize   : Word;
    Private   : Word;
    BufPos    : Word;
    BufEnd    : Word;
    BufPtr    : ^TTextBuf;
    OpenFunc  : Pointer;
    InOutFunc : Pointer;
    FlushFunc : Pointer;
    CloseFunc : Pointer
    UserData  : Array [1..16] of Byte;
    Name      : Array [0..79] Of Char;
    Buffer    : TTextBuf;
  End; {Record}
```

TDateTime Type

Variables of *TDateTime* type are used in conjunction with the *UnpackTime* and *PackTime* procedures to

examine and construct 4-byte, packed date-and-time values for the *GetFTime, SetFTime, FindFirst,* and *FindNext* procedures.

```
Type
  TDateTime = Record
    Year, Month, Day, Hour, Min, Sec : Word
  End; {Record}
```

TSearchRec Type

The *FindFirst* and *FindNext* procedures use variables of type *TSearchRec* to scan directories.

```
Type
  TSearchRec = Record
    Fill : Array [1..21] Of Byte;
    Attr : Byte;
    Time : LongInt;
    Size : LongInt;
    Name : Array [0..12] Of Char;
```

DOSError Variable

DOSError is used by many of the routines in the *WinDOS* unit to report errors.

```
Var
  DosError : Integer;
```

The values stored in *DOSError* are DOS error codes. A value of 0 indicates no error; other possible error codes include:

DOS Error Code	Means
2	File not found
3	Path not found
5	Access denied
6	Invalid handle
8	Not enough memory
10	Invalid environment
11	Invalid format
18	No more files

Using NS_Text
TTextFile Object Reference

Types

FileIntegType An enumerated type containing the identifiers *NS_Normal*, and *NS_High*. Used in the *Integrity* method.

+ Setting the DOS verify flag (via *SetVerify*) to true will ensure the highest integrity when used in conjunction with *NS_High*. Note, though, this configuration will severely hamper performance of the entire system.

FileNameType A PChar type. The maximum length is defined by the *fsPathName* constant (which is 79) plus 1.

FileOpenType An enumerated type containing the identifiers *NS_Reset, NS_ReWrite, NS_Append* and *NS_Closed.* These identifiers correspond with the three modes in which a text file can be opened for processing, Reset, ReWrite, and Append, respectively. *NS_Closed* leaves the file closed upon initialization, and may be useful when using *FCopy, GetFAttr,* or *SetFAttr.*

PTextFile A pointer to the *TTextFile* object.

TTextFile The *TTextFile* object.

Unit Variables (typed constants)

TextError A byte type set to 0 when TextOK is True. When TextOK is False, TextError holds the first Turbo Pascal run-time error code which caused the methods result to become unreliable.

TextOK A boolean type set to True if the method completed reliably. TextOK is set to False when the methods result will be unreliable upon termination; usually requiring the programmer (you) to further define the exception logic.

+ The term *unreliable* is used in place of the more general *failed* because TTextFile methods automatically compensate for many internal functions or procedures generating a run-time error (i.e. a run-time error may simply redirect the flow of logic within the method).
+ With **{$I-}**, *IOResult* returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Instance Variables

F The text-file variable associated with the external file *FileName*.

FileInteg A *FileIntegType* type identifying the level of integrity to be maintained for this file instance.

FileMode A *FileOpenType* type identifying the mode in which the file was opened for I/O.

FileName Holds the DOS path and file name passed with the *Init* constructor. *FileName* is of type *FileNameType*. 80 bytes of heap space are allocated via *GetMem* in the *Init* constructor and released via *FreeMem* in the *Close / Erase* destructors.

FlushBuffer A boolean type which is set to true when data is buffered to disk (i.e. when a *Write / WriteLn* has been issued but before *Flush* has been called). *FlushBuffer* is initialized to False.

Methods

Append **Procedure Append;**

Closes and reopens an existing file for appending. *Append* reopens the external file with the name assigned to *FileName* for write-only output. The current file position is set to the end of the file. If there is no external file, *Append* will first create one. If the external file was already open, it is first closed and then re-opened. If a *Ctrl+Z* is present in the last 128-byte block of the file, the current file position is set to overwrite the first *Ctrl+Z* in the block. In this way, text can be appended to a file that terminates with a *Ctrl+Z*. If *FileName* is an empty name (such as *x*.Init(' ', NS_Append);), then all calls will refer to the standard Output file (standard file handle number 1).

Close **Destructor Close; Virtual;**

The *Close* destructor closes the open file and removes the objects instance from memory. The external file is completely updated and its DOS file handle is freed for reuse.

EOF **Function EOF:Boolean; Virtual;**
Returns the end-of-file status of a text file. *EOF* operates on the file specified in the *Init* procedure and returns True if it is beyond the last character of the file or if the file contains no components; otherwise, *EOF* returns False.

EOLN **Function EOLN:Boolean; Virtual;**
Returns the end-of-line status of a file. *EOLN* operates on the file specified in the *Init* procedure and returns True if the current file position is at an end-of-line marker (or if *EOF* is True); otherwise, *EOLN* returns False.

Erase **Destructor Erase; Virtual;**
The *Erase* destructor erases the external file and removes the objects instance from memory. The external file is first closed, freeing its DOS file handle for reuse.

FCopy **Procedure FCopy(DestName:FileNameType); Virtual;**
Copies an external file to the new external file *DestName*. *DestName* is of type *FileNameType*. The original external file remains intact. After the file has been successfully copied, it will be reopened according to the state identified in *FileMode*. The current file position is set to the beginning of the file (unless *FileMode* is NS_Append, and then it is set to the end of the file) and the value of *FlushBuffer* is set to False. Further operations will operate on the original external file. A method Borland should certainly include!

FileSplit **Function(Dir, Name, Ext:PChar); Virtual;**
Splits the file name maintained in *FileName* into its three components. *Dir* is set to the drive and directory path with any leading and trailing backslashes, *Name* is set to the file name, and *Ext* is set to the extension with a preceding period. Notice that the returned values are of type PChar; be sure to check for **nil** pointers. The maximum lengths of the strings returned in *Dir, Name*, and *Ext* are defined by the *fsDirectory, fsFileName,* and *fsExtension* constants. The returned value is a combination of the *fcDirectory, fcFileName*, and *fcExtension* bit masks, indicating which components were present in *FileName*. If the name or extension contains any wildcard characters (* or ?), the *fcWildCards* flag is set in the returned value.

Flush **Procedure Flush; Virtual;**
Flushes the buffer of a text file open for output. When a text file has been opened for output using *ReWrite* or *Append*, a call to *Flush* will empty the file's buffer. This guarantees that all characters written to the file at that time have actually been written to the external file. If an I/O run-time error occurs, *Flush* remains in its current state (that is, prior to the call). *Flush* has no effect on files opened for input.

GetFAttr **Procedure GetFAttr(VAR Attr:Word); Virtual;**
Returns the attributes of a file. The attributes are examined by ADDing them with the file attribute masks defined as constants in the *WinDOS* unit.

GetFTime **Procedure GetFTime(VAR Time:LongInt); Virtual;**
Returns the date and time a file was last written. The time returned in the *Time* parameter may be unpacked through a call to *UnPackTime*.

Init **Constructor Init(Name:FileNameType; State:FileOpenType);**
Opens the file name specified in the *.Init Name* parameter for I/O. *Name* is expanded into a fully qualified file name and stored in the *FileName* instance variable. The *FlushBuffer* instance variable is set to false and the *FileMode* instance variable is set equal to the enumerated type specified in the *State* parameter. *FileName* is assigned to the instance variable *F* and according to the enumerated type specified in the *State* parameter, opened for I/O.

Integrity **Procedure Integrity(Integ:FileIntegType);**
Takes a parameter of type FileIntegType to determine the level of integrity Write / WriteLn are to maintain. If Integrity is *NS_High* then the buffer is flushed after each Write / WriteLn.

Read **Procedure Read(VAR Item:String); Virtual;**
Reads a file component into a variable. This method is intended to be overridden, unless you intend to read in 1 component of type string.

ReadLn **Procedure ReadLn(VAR Item:String); Virtual;**
Executes the *Read* procedure then skips to the next line of the file.

ReName **Procedure ReName;**
Closes then renames an external file. *NewName* is of type *FileNameType*. The external

file is renamed to *NewName*. After the file has been successfully renamed, it will be reopened according to the state identified in *FileMode*. The current file position is set to the beginning of the file (unless *FileMode* is NS_Append, and then it is set to the end of the file) and the value of *FlushBuffer* is set to False. Further operations will operate on the external file with the new name.

Reset **Procedure Reset;**

Closes and reopens the existing external file for read-only input specified in *FileName*, setting the current file position to the beginning of the file. If *FileName* is an empty name (such as *x*.Init(' ', NS_Reset);), then all calls will refer to the standard input file (standard file handle number 0). After a call to *Reset*, *EOF* is True if the file is empty or did not exist; otherwise, *EOF* is False.

ReWrite **Procedure ReWrite;**

Closes recreates and reopens the external file specified in *FileName* for write-only output. If an external file with the same name already exists, it is deleted and a new empty file is created in its place. The current file position is set to the beginning of the empty file. If *FileName* is an empty name (such as *x*.Init(' ', NS_ReWrite);), then all calls will refer to the standard output file (standard file handle number 1).

+ Reset, ReWrite, and Append are core methods called by other methods, such as Init and ReName. For this reason they have not been defined as virtual.

SeekEOF **Function SeekEOF:Boolean; Virtual;**

Returns the end-of-file status of a file. *SeekEOF* corresponds to *EOF* except that it skips all blanks, tabs, and end-of-line markers before returning the end-of-file status. This is useful when reading numeric values from a text file.

SeekEOLN **Function SeekEOLN:Boolean; Virtual;**

Returns the end-of-line status of a file. *SeekEOLN* corresponds to *EOLN* except that it skips all blanks and tabs before returning the end-of-line status. This is useful when reading numeric values from a text file.

SetFAttr **Procedure GetFAttr(VAR Attr:Word); Virtual;**

Sets the attributes of a file. The attribute value is formed by ADDing the appropriate attribute masks defined as constants in the *WinDOS* unit.

SetFlush **Procedure SetFlush;**

Used with procedures and functions which extend the *TTextFile* object, *SetFlush* sets the value of *FlushBuffer* to true, indicating that data may be buffered and needs to be flushed. This method does not affect the value of *IORresult*.

SetFTime **Procedure GetFTime(VAR Time:LongInt); Virtual;**

Sets the date and time a file was last written. The *time* parameter can be created through a call to *PackTime*.

Write **Procedure Write(Item:String); Virtual;**

Writes a variable to a file component. This method is intended to be overridden, unless you intend to write 1 variable of type String.

WriteLn **Procedure WriteLn(Item:String); Virtual;**

Executes the *Write* procedure then skips to the next line of the file.

+ When overriding *Write / WriteLn*, *SetFlushBuffer* must be called after each Write / WriteLn call for *Integrity* to work properly.

C:\WINDOWS\WORD\DOT\!5X8.DOT

Error Codes

DOS Error Codes

2 **File not found.**
Reported by *Append, Erase, ReName,* or *Reset* if the name assigned to the file variable does not specify an existing file.

3 **Path not found.**
Reported by *Append, Erase, ReName, Reset, ReWrite, GetFAttr,* or *SetFAttr.* If the name assigned to the file variable is invalid or specifies a nonexistent subdirectory.

4 **Too many open files.**
Reported by *Append, Reset,* or *ReWrite* if the program has too many open files. DOS never allows more than 15 open files per process. If you get this error with less than 15 open files, it may indicate that the CONFIG.SYS file does not include a FILES=*xx* entry or that the entry specifies too few files. Increase the number to some suitable value, (20 for instance).

+ It has been recommended and confirmed countless times on the Microsoft and Borland CIS forums that a more realistic value for FILES= is 60. This is especially true if you are running in Enhanced mode.

5 **File access denied.**
Reported by *Reset* or *Append* if *FileMode* allows writing and the name assigned to the file variable specifies a directory or read-only file. Reported by *ReWrite* if the directory is full or if the name assigned to the file variable specifies a directory or an existing read-only file. Reported by *ReName* if the name assigned to the vile variable specifies a directory or if the new name specifies an existing file. Reported by *Erase* if the name assigned to the file variable specifies a directory or a read-only file. Reported by *GetFAttr* and *SetFAttr*.

6 **Invalid file handle**
Reported by *GetFTime* and *SetFTime* if an invalid file handle is passed to DOS. It should never occur; if it does, though, it is an indication that the file variable has been corrupted.

17 **Cannot rename across drives.**
Reported by *ReName* if both names are not on the same drive.

IOResult Error Codes

101 **Disk write error.**
Reported by *Close, Flush, Write,* or *WriteLn* if the disk becomes full.

102 **File not assigned.**
Reported by *Append, Erase, Rename, Reset,* or *ReWrite* if the file variable has not been assigned a name through a call to *Assign*.

103 **File not open.**
Reported by *Close, EOF, Flush, Read,* or *Write* if the file is not open.

104 **File not open for input.**
Reported by *EOF, EOLN, Read, ReadLn, SeekEOF,* or *SeekEOLN* if the file is not open for input.

105 **File not open for output.**
Reported by *Write*, or *WriteLn* if the file is not open for output.

106 **Invalid numeric format.**
Reported by *Read*, or *ReadLn* if a numeric value read from a text file does not conform to the proper numeric format.

+ Because *TTextFile* does its own internal I/O error checking, and because a call to the IOResult function clears the internal I/O error flag, IOResult will not function properly outside of the *TTextFile* objects (when used in regards to the file that particular object is associated with. This is not to say, though, IOResult will not function properly if you include *NS_Text* into your code.) To avoid any confusion, when interfacing with a *TTextFile* object you should simply use *TextOK / TextError* instead.