How to Use File Handles

By Shayanna Gatchalian, Technical Services Engineer, 4D Inc.

Technical Note 23-04

Table of Contents

Abstract

Data can be stored in various forms, from something as simple as saving a number value to a variable to keeping millions of records in a database table. In a world where data is often shared between different contexts, it is imperative that there are established conventions to formally retrieve data from one form and convert it to another that is usable in a different setting. One such common conversion deals with reading content from a file, handling its data in a program, and possibly writing back new data to the opened file or a newly created one. With the newly released 4D v19 R7, FileHandle objects provide a way to easily do this with its well-designed properties and functions.

Introduction

4D v19 R7 showcases various new features including file handles, a brand-new object that allows a more efficient way to handle file content. Its associated FileHandle class has several properties and functions that make the development quick and feasible.

Before the newest R-release, file handling was already possible; the **Document to text** and **TEXT TO DOCUMENT** commands allow the developer to retrieve and rewrite the entire content of a file, while a combination of the **GET/SET DOCUMENT POSITION**, **RECEIVE PACKET**, and **SEND PACKET** commands make it possible to read and write parts of a file.

However, learning about file handles serves as a great addition to one's 4D development skills as the object can inherently read or write individual lines of text, follows object-oriented programming concepts, does not require a manual file close, and much more. This Technical Note will take a deeper look into the FileHandle object, highlight its benefits over the classic way of file handling, and show how to set up a file handle with a basic logger demo application.

What are file handles?

As mentioned earlier, file handles are objects based on the File object introduced back in v17 R5. The FileHandle class contains properties and functions that allow the user to not only access any part of a file and either read or write content, but also customize the way it does so—further details will be explained in the "How to use a file handle" section. A file handle can be opened in one of three modes (e.g., read, write, or append), and each one has an **.offset** property to pinpoint its specific location within the file. But, if there is already a way to handle file content, why choose file handles over its classic counterpart? Let's compare each way's process of file handling to get a better overall picture.

Classic Way

The following steps outline the typical order of operations when executing file handling in the classic way:

1) The file is first opened using the **OPEN DOCUMENT** command, in which a document reference number is returned.
2) The offset is automatically set at the very beginning of the file but can be set to a later position in the document by using the **SET DOCUMENT POSITION** command; similarly, the **GET DOCUMENT POSITION** command would return the current position of the offset.
3) Reading from a file
    a. Reading from a file uses the **RECEIVE PACKET** command, which takes in either an integer value of the number of characters to read or a text value called *stopChar* that 4D will continue reading until.
4) Writing to a file
    a. Writing to a file uses the **SEND PACKET** command, which either writes text or BLOB content directly to the file at the specified offset position.
5) The file must then be manually closed using the **CLOSE DOCUMENT** command with its corresponding document reference number. Forgetting to execute this command in runtime will lock the file, and it can only be released by restarting the database; therefore, you must make sure to execute this command even when debugging.

New FileHandle objects

Compared to the classic way, file handles do require a little more setup; however, they offer more flexibility and options for file content manipulation. The following steps outline the typical pipeline of creating and working with file handles:

1) A File object is first created to either load a disk file into 4D or generate an entirely new file to be written to the disk.
2) Then, a File Handle object is created with the **.open()** function of the File class. It can be opened in one of three modes: read, write, or append. The default placement of the offset depends on the mode the file handle is opened in, and its value can be either retrieved or set using the **.offset** property.
3) Read mode

   a. The offset/reading head is placed at the beginning of the file by default. There are three options of how you would like to read from a file:
      i. **.readLine()**
      ii. **.readText()**
      iii. **.readBlob()**

4) Write/append mode
   a. In write mode, the offset/writing head is initially placed at the beginning of the document, while in append mode, it is set to the end of the file by default. Both modes share the same set of functions for writing to the file:
      i. **.writeLine()**
      ii. **.writeText()**
      iii. **.writeBlob()**
5) With file handles, closing the document is done a little differently from the classic way. Once the FileHandle object is no longer referenced in the code, the file handle is automatically closed. This can be done in either one of two ways: either a process containing the reference to the file handle is terminated or the reference to the file handle is manually set to null. In both scenarios, the reference to the file object itself would no longer be available, allowing 4D to close the file handle.

Aside: Manipulating the entire content of a file

When it comes to reading and writing a file's entire content, there is no difference in performance between the classic commands and the File class's get and set functions; it simply comes down to whether you happen to be using the File object or not. If a File object has not been created, use the following commands:

- **Document to text**
- **TEXT TO DOCUMENT**
- **DOCUMENT TO BLOB**
- **BLOB TO DOCUMENT**

If a File object is already in use, you have the option to use the following functions of the File class:

- **.getText()**
- **.setText()**

- **.getContent()**
- **.setContent()**

Benefits of file handles

Now that we looked at the different protocols for file handling, here are the reasons why file handles are more beneficial and should be utilized.

- Unlike the classic way of manipulating file content, file handles ***have built-in functions to read and write individual lines of a file***, with the **.readLine()** and **.writeLine()** function. While it was possible to implement this before v19R7, it requires a more roundabout way of programming; to do this, the *stopChar* parameter would be used, but it would not inherently consider the different end-of-line delimiters used between Mac and Windows environments.
- The **design of the FileHandle object** allows for more feasible programming on the 4D developer's end, as entire tasks are encapsulated into single-line functions.
- File handles also **integrate easily with object-oriented programming** as it is an object and utilizes dot notation (e.g., the **.offset** property) to more easily customize the file handling process.
- With three available modes, you **can have a file handle in each mode running at the same time**, each keeping track of their own offset. In the classic way, a document can only be opened in one context, meaning the developer would have to manually keep track of the reading and writing heads if he or she intends to read and write content at the same time.
- As mentioned in the "New FileHandle objects" section, there is **no need to close the file manually**. This is a huge benefit as errors can occur when a document is not closed properly and is locked until the database can be restarted.

With these many benefits in mind, it is easy to see why file handles are the better choice when implementing file handling code. Its class was designed with that very purpose, unlike the classic way, in which the **RECEIVE PACKET** and **SEND PACKET** commands could also be used in conjunction with serial ports. The next section will briefly introduce File and Folder objects, as they are essential in creating and working with file handles.

Brief introduction of File and Folder objects

Before working with file handles, it is best to first be familiarized with File and Folder objects, since file handles are an extension of the File object. If you believe you already have extensive knowledge on this topic, you may skip to the next section.

File objects

As mentioned many times before, a File object is needed to create a file handle. To do this, the File object must be instantiated by either inputting a path or specifying a system/built-in 4D document constant value.

It must be noted that the 4D File object is an entirely different entity from the file stored on the disk. If you are loading an existing document into 4D, it is advised to check its validity by using the **.exists** property; or, if you intend to write an entirely new file to the disk, the **.create()** function must be used. Take a look at the code snippets below.

Using the *path* parameter

```
$path:=Get 4D folder(Database folder)+"Resources"+Folder
separator+"test.txt"
$file:=File($path; fk platform path)
If ($file.exists=False)
   $file.create()  // write new file to the disk
   // or insert handling code here
End if
```

Using the *fileConstant* parameter

```
$file:=File(Backup log file)
If ($file.exists=False)
   // insert handling code here
End if
```

After that, the File object is then created in the context of 4D. With object notation, you may obtain information as well as manipulate the object itself using the File object's predefined properties and functions (refer to the table below).

Properties and functions of the File object

Property	**Function**
.creationDate	4D.File.new()
.creationTime	.copyTo()
.exists	.create()
.extension	.createAlias()
.fullName	.delete()
.hidden	.getAppInfo()
.isAlias	.getContent()
.isFile	.getIcon()
.isFolder	.getText()
.isWritable	.moveTo()
.modificationDate	.open()
.modificationTime	.rename()
.name	.setAppInfo()
.original	.setContent()
.parent	.setText()
.path	
.platformPath	
.size	

More information on each of these properties and functions may be found on the 4D documentation page here: https://developer.4d.com/docs/API/FileClass/

Folder objects

The Folder object is essentially the same as the File object, except for a few minor differences. It is instantiated using the *path* or *folderConstant* parameters, and the same rules apply when inputting the platform path instead of the default POSIX path. Like File objects, Folder objects are a separate entity from any corresponding folders on the disk.

The Folder object also uses object notation, so you may use its properties and functions to work with it however you would like. Most of these attributes are the same as the File object, except the Folder object has additional functions to retrieve information on its contained files and subfolders. You can see the full list of properties and functions in the table below.

Properties and functions of the Folder object

Property	**Function**
.creationDate	4D.Folder.new()
.creationTime	.copyTo()
.exists	.create()
.extension	.createAlias()
.fullName	.delete()
.hidden	.getIcon()
.isAlias	.getText()
.isFile	.file()
.isFolder	.files()
.isWritable	.folder()
.modificationDate	.folders()
.modificationTime	.moveTo()
.name	.rename()
.original	
.parent	
.path	
.platformPath	

More information on each of these properties and functions may be found on the 4D documentation page here: https://developer.4d.com/docs/API/FolderClass/

Basic knowledge on 4D File and Folder objects is plenty for 4D developers to handle their disk files and folders from an outside scope. File handles, on the other hand, operate in the inner scope of the file's content itself. The next section will go over how to create and work with the File Handle object.

Aside: POSIX vs. platform paths

When working with paths, it is important to consider the different formats between Mac and Windows. By default, File and Folder objects take in the universal POSIX path; if you would like to use the machine's native format, the *pathType* parameter must also be defined. Within the object, the POSIX path is saved into the **.path** property, while the platform path is separately saved into the *.platformPath* property.

POSIX path

```
$posixPath:="/Users/Username/Documents/SampleDatabase/Resources/new.
txt"
$file:=File($posixPath)
```

Platform path (Mac)

```
$platformPath:="Macintosh
HD:Users:Username:Documents:SampleDatabase:Resources "
$file:=Folder($path; fk platform path)
```

Platform path (Windows)

```
$platformPath:="C:\\Users\\Username\\Documents\\SampleDatabase\\Reso
urces\\test.txt"
$file:=File($path; fk platform path)
```

How to create and use a file handle

Creating a file handle is done simply by using the File object's **.open()** function. There are two options for its input parameters: a *mode* text value which specifies the mode to open the file handle in (e.g., read, write, append) or an *option*s object with attributes to customize the file handling process.

Mode parameter

```
$fileHandle:=$file.open("read")
```

Options parameter

```
$o:=New object()
$o.mode:="append"
$o.charset:="UTF-8"
$o.breakModeRead:=Document with CRLF
$o.breakModeWrite:=Document with CRLF
$fileHandle:=$file.open($o)
```

The options object consists of four attributes. The *mode* attribute specifies the file handle mode, as the name implies. *Charset* defines the character set the file is written in. And the *breakModeRead* and *breakModeWrite* attributes describe the end-of-line delimiters used in the file.

Aside: Overview of the three file handle modes

The table below outlines the differences between the three file handle modes. If the mode attribute is not defined, the file handle is opened in read mode by default; moreover, the read file handle is unique in that multiple file handles created in this mode can run on the same File object. One important thing to note is that only either a write handle or append handle can be opened on the same File object (both cannot exist at the same time)*.

	Read	**Write**	**Append**
Function	Reads file	Writes to *beginning of file*	Writes to *end of file*
When File = Null	Returns error	Creates file on disk	Creates file on disk
# Handles Per File Object	>1	Only 1*	Only 1*

The .offset property

The offset is defaulted to the beginning of the file in read and write modes, while it is positioned to the end of the file in append mode. You may obtain its exact location or set its position using the **.offset** property.

Getting and setting the **.offset** property

```
// get position
$offset:=$fileHandle.offset

// set position
$fileHandle.offset:=5  // 5th character in the file
```

Read mode

Read file handles have three unique functions that each offer a different way to fetch parts of the file content.

- **.readLine():** reads a single line from the document (the file handle detects the return character defined by the **.breakModeRead** attribute)
- **.readText():** reads a stream of characters from the offset up to the character immediately before the string of characters defined in the *stopChar* parameter
- **.readBlob():** reads a certain number of bytes specified by the *bytes* parameter and returns a BLOB value

The three read mode functions

```
// .readLine()
$line:=$readHandle.readLine()

// .readText()
$text:=$readHandle.readText("file handles")

// .readBlob()
$blob:=$readHandle.readBlob(25)
```

Write/append mode

File handles opened in either write or append mode use the same three functions. These work in the exact same way in both modes; the only difference is their default offset positions.

- **.writeLine()**: writes a single line of text ending with a return character, designated by the **.breakModeWrite** attribute
- **.writeText()**: writes a string to the file without the end-of-line delimiter
- **.writeBlob()**: writes a BLOB value to the file (does not include a new line character)

The three write/append mode functions.

```
// .writeLine()
$writeHandle.writeLine("The quick brown fox")

// .writeText()
$appendHandle.writeText("hello, world!")

// .writeBlob()
$text:="To be or not to be"
TEXT TO BLOB($text; $blob)
$writeHandle.writeBlob($blob)
```

Once you are completely done with the file handle, it is always a good idea to ensure that the document is closed. If the file handle was created locally within a method, the object will no longer be referenced once the process is terminated. Otherwise, if the file handle was created in a more global context (e.g., saved within form data or a process/interprocess variable), you may set this reference to null.

For more information on the FileHandle object, you can find the documentation page here: https://developer.4d.com/docs/API/FileHandleClass/

As you can see, developing file handles is super easy and intuitive thanks to the FileHandle object's included read and write functions. With these options available to the 4D developer, he or she can put less time into file handling implementation and more time into other features of the database. With a general understanding of how the FileHandle object works, let's see this 4D object in action with a walkthrough of a basic logger demo application.

File Handle Demo Application Walkthrough

The file handle demo application can be found in the *FileHandleDemo* folder that came along with this technical note. The purpose of this sample database is to show how file handles work in practice. This section of the technical note will go over how to use the application; for information on the actual implementation, comments are provided in each 4D method, detailing the method's purpose as well as algorithm explanation.

The application is split into two contexts: Text and Blob. Under the Resources folder, there are two .txt files: *FoxInSocks_Text.txt* and *FoxInSocks_Blob.txt*. While both files essentially have the same exact content, the first one deals with the text functions of the file handles (a.k.a., **readLine()**, **readText()**, **writeLine()**, **writeText()**) while the second deals with the blob functions (a.k.a., **readBlob()**, **writeBlob()**).

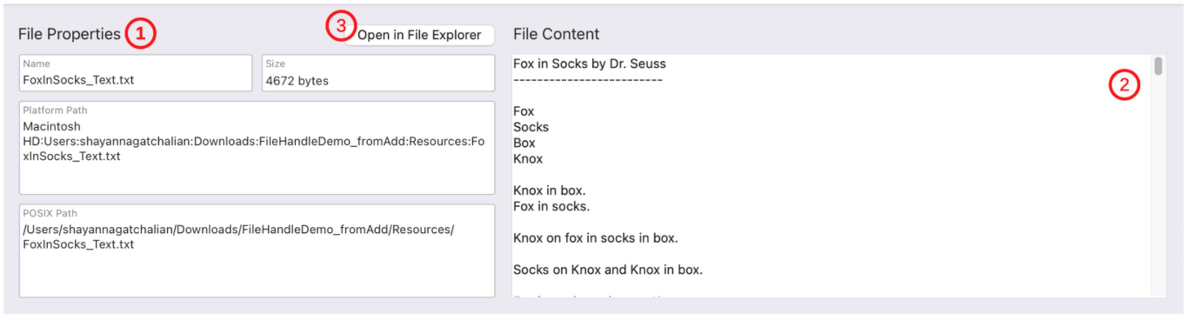
If you would like to use different files to import into the demo application, you may edit the paths in the loadFile method.

Text File Handle Demo

The *Text File Handle Demo* window demonstrates the text functions of the FileHandle object. These functions are only used on the *FoxInSocks_Text.txt* file.

1. The top of this page first displays the *FoxInSocks_Text.txt* file's properties and content and only shows the properties relevant to file handles. For a complete list of the File class properties, you may refer to the File class documentation page (https://developer.4d.com/docs/API/FileClass/) or the table on pages 7-8 (under the *File objects* section of this technical note).
2. The *Text_FileContent* text field then displays the entire content of the file, using the **getText()** function.
3. A button on the top right of the window opens the file explorer to show the location of the *FoxInSocks_Text.txt* file.

File section of the *Text* tab



Name
FoxInSocks_Text.txt

Size
4672 bytes

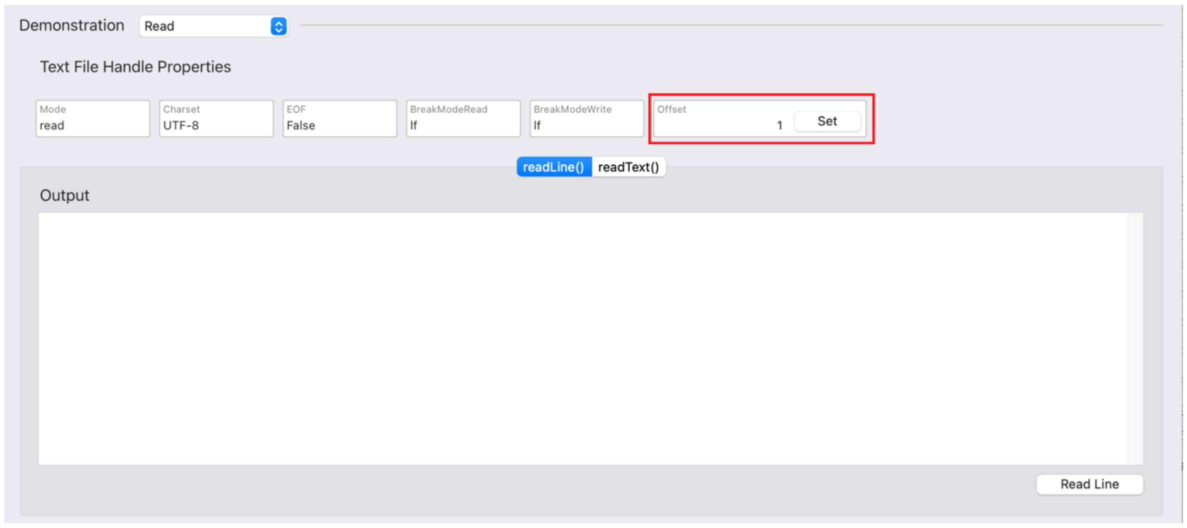
Platform Path
Macintosh HD:Users:shayannagatchalian:Downloads:FileHandleDemo_fromAdd:Resources:FoxInSocks_Text.txt

POSIX Path
/Users/shayannagatchalian/Downloads/FileHandleDemo_fromAdd/Resources/FoxInSocks_Text.txt

```
Fox in Socks by Dr. Seuss
-------------------------

Fox
Socks
Box
Knox

Knox in box.
Fox in socks.

Knox on fox in socks in box.

Socks on Knox and Knox in box.
```

The next section deals with the actual file handles. The first section presents the file handle properties; the offset for each file handle can be manually set by inputting an offset value into the field shown below.

File Handle section of *Text File Handle Demo*


Demonstration: Read

Text File Handle Properties

Mode: read
Charset: UTF-8
EOF: False
BreakModeRead: lf
BreakModeWrite: lf
Offset: 1

The dropdown menu on the top-left allows you to switch between the read, write, and append file handle modes on the same *FoxInSocks_Text.txt* file. The file handle functions are split in the following way:

Table of file handles and functions on the *FoxInSocks_Text.txt* file

File Handle	**Function**	**Input**	**Output**
Read Handle	readLine()	*Text_ReadLineButton* button click	Line of text from the read handle offset to the end of the line -> displays in *Text_ReadHandleOutput* text field
	readText()	*stopChar* text input	Text value from read handle offset to right before *stopChar* in the file -> displays in *Text_ReadHandleOutput* text field
Write Handle	writeLine()	*lineOfText* text input	Writes *lineOfText* to *FoxInSocks_Text.txt* at the write handle offset -> displays in *Text_FileContent* text field

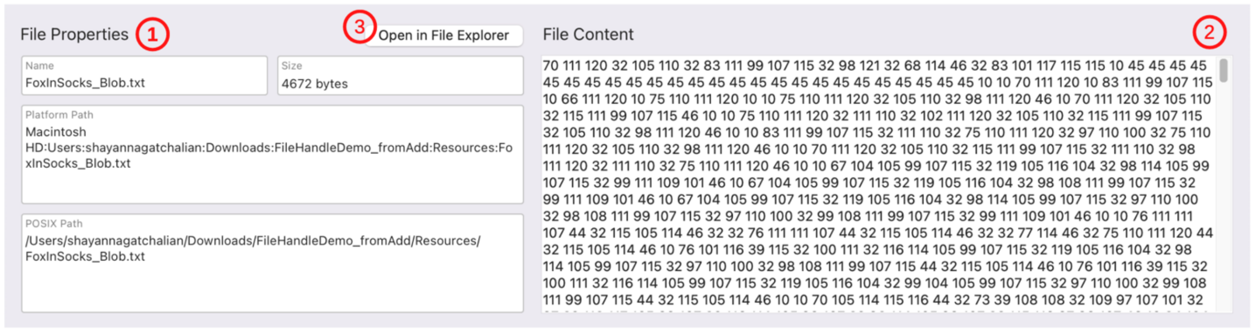
	writeText()	*stopChar* text input	Writes *textToWrite* to *FoxInSocks_Text.txt* at the write handle offset -> displays in *Text_FileContent* text field
Append Handle	writeLine()	*lineOfText* text input	Writes *lineOfText* to *FoxInSocks_Text.txt* at the append handle offset -> displays in *Text_FileContent* text field
	writeText()	*stopChar* text input	Writes *textToWrite* to *FoxInSocks_Text.txt* at the append handle offset -> displays in *Text_FileContent* text field

Blob File Handle Demo

The *Blob File Handle Demo* window works very similarly to its text counterpart. This side of the application works with the *FoxInSocks_Blob.txt* file, and only deals with the **readBlob()** and **writeBlob()** functions of the FileHandle class.

1. The first section of this page shows the relevant File object properties of the *FoxInSocks_Blob.txt* file.
2. The *Blob_FileContent* text field is implemented a little differently from the *Text_FileContent* text field. The **getContent()** function returns a 4D.Blob object rather than a BLOB value. This object contains a collection of BLOB octets stored as real values. The collection is iterated through using a loop, converting each real value into a string, and then concatenating that string to an overall output string to be displayed in the *Blob_FileContent* text field.
3. This section also features a button that will lead to the location of *FoxInSocks_Blob.txt* file in the file explorer.

File section of the Blob File Handle Demo



Name
FoxInSocks_Blob.txt

Size
4672 bytes

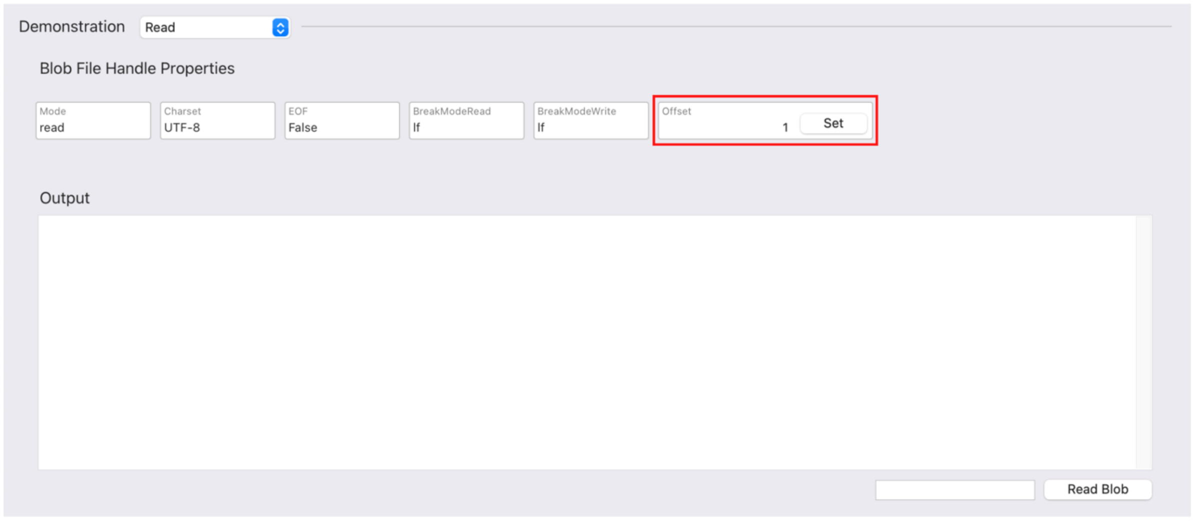
Platform Path
Macintosh HD:Users:shayannagatchalian:Downloads:FileHandleDemo_fromAdd:Resources:FoxInSocks_Blob.txt

POSIX Path
/Users/shayannagatchalian/Downloads/FileHandleDemo_fromAdd/Resources/FoxInSocks_Blob.txt

```
70 111 120 32 105 110 32 83 111 99 107 115 32 98 121 32 68 114 46 32 83 101 117 115 115 10 45 45 45 45
45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 10 10 70 111 120 10 83 111 99 107 115
10 66 111 120 10 75 110 111 120 10 10 75 110 111 120 32 105 110 32 98 111 120 46 10 70 111 120 32 105 110
32 115 111 99 107 115 46 10 10 75 110 111 120 32 111 110 32 102 111 120 32 105 110 32 115 111 99 107 115
32 105 110 32 98 111 120 46 10 10 83 111 99 107 115 32 111 110 32 75 110 111 120 32 97 110 100 32 75 110
111 120 32 105 110 32 98 111 120 46 10 10 70 111 120 32 105 110 32 115 111 99 107 115 32 111 110 32 98
111 120 32 111 110 32 75 110 111 120 46 10 10 67 104 105 99 107 115 32 119 105 116 104 32 98 114 105 99
107 115 32 99 111 109 101 46 10 67 104 105 99 107 115 32 119 105 116 104 32 98 108 111 99 107 115 32
99 111 109 101 46 10 67 104 105 99 107 115 32 119 105 116 104 32 98 114 105 99 107 115 32 97 110 100
32 98 108 111 99 107 115 32 97 110 100 32 99 108 111 99 107 115 32 99 111 109 101 46 10 10 76 111 111
107 44 32 115 105 114 46 32 32 76 111 111 107 44 32 115 105 114 46 32 32 77 114 46 32 75 110 111 120 44
32 115 105 114 46 10 76 101 116 39 115 32 100 111 32 116 114 105 99 107 115 32 119 105 116 104 32 98
114 105 99 107 115 32 97 110 100 32 98 108 111 99 107 115 44 32 115 105 114 46 10 76 101 116 39 115 32
100 111 32 116 114 105 99 107 115 32 119 105 116 104 32 99 104 105 99 107 115 32 97 110 100 32 99 108
111 99 107 115 44 32 115 105 114 46 10 10 70 105 114 115 116 44 32 73 39 108 108 32 109 97 107 101 32
```

Like *Text File Handle Demo*, the second half of the form also features a subsection where the read, write, and append handles' properties are displayed. The offsets for these file handles can also be manually set.

Demonstration section of the Blob File Handle Demo



Mode	Charset	EOF	BreakModeRead	BreakModeWrite	Offset
read	UTF-8	False	lf	lf	1

The file handle section also has a similar tab control to switch between the read, write, and append handles. The *Blob File Handle Demo* module is much simpler than the first, since there is only one function available for each file handle mode when working with BLOB values. The function split is as follows:

Table of file handles and functions on the *FoxInSocks_Blob.txt* file

File Handle	**Function**	**Input**	**Output**
Read Handle	readBlob()	*bytes* real input	4D.Blob object with BLOB stored as collection, reads file from read handle offset up until the number of bytes specified in *bytes* input -> displays in *Blob_ReadHandleOutput*
Write Handle	writeBlob()	*blob* 4D.Blob input	Writes *blob* to *FoxInSocks_Blob.txt* at the write handle offset -> displays in *Blob_FileContent* text field
Append Handle	writeBlob()	*blob* 4D.Blob input	Writes *blob* to *FoxInSocks_Blob.txt* at the append handle offset -> displays in *Blob_FileContent* text field

Conclusion

This Technical Note first compared the classic and new ways of file handling and underscored the benefits of File Handle object newly introduced in v19R7. File handles allow for easier development from the 4D developer's perspective as they integrate well with object-oriented programming, have a multitude of built-in properties and functions, and most importantly allow for read/write on sections of a file for a more efficient runtime. With these benefits in mind, it is hard to pass up using this 4D object for any features that require file content handling or for any current databases that already do so. Give file handles a try, and you will understand how great it is to not only implement but also execute in deployment.